ANALYSIS OF SHA-3 FINAL ROUND CANDIDATE ALGORITHMS AND DESIGN OF VARIANT TO SKEIN HASH FAMILY

A Thesis Submitted to Lovely Professional University

For the Award of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE AND ENGINEERING

By
RAJEEV SOBTI

Guide
Dr. G. GEETHA

FACULTY OF TECHNOLOGY AND SCIENCES
LOVELY PROFESSIONAL UNIVERSITY
PUNJAB, INDIA
MAY, 2016

DECLARATION

I declare that the thesis entitled **'Analysis of SHA-3 Final Round Candidate Algorithms and Design of Variant to Skein Hash Family'** has been prepared by me under the guidance of **Dr. G. Geetha**, Professor of School of Computer Applications, Lovely Professional University**.** No part of this thesis has formed the basis for the award of any degree or fellowship previously.

Rajeev Sobti
School of Computer Science and Engineering
Lovely Professional University
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab, INDIA
DATE : 11 May 2016

CERTIFICATE

I certify that **Rajeev Sobti** has prepared his thesis entitled **'Analysis of SHA-3 Final Round Candidate Algorithms and Design of Variant to Skein Hash Family'**, for the award of Ph.D. degree of Lovely Professional University**,** under my guidance. He has carried out the work at the School of Computer Science and Engineering, Lovely Professional University.

Dr. G. Geetha
Professor and Associate Dean
School of Computer Applications
Lovely Professional University
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab, INDIA
Date : 11 May 2016

ABSTRACT

Technological advancements and globalization have made faster exchange of information a very vital aspect of human life and society. The exponential growth of networks, internet, and portable devices have contributed to proliferation of information. With all this, securing information has become equally important and challenging.[1] Cryptography in general and Cryptographic Hash Functions in particular are extensively used to accomplish security of information. Cryptographic Hash Functions are crucial in implementing multiple security goals and have led their way into various security applications like: digital signatures, storing passwords, digital time stamping, constructing block ciphers, generating pseudorandom numbers, maintaining secure web connections, encryption key management, virus scanning, indexing data in hash tables, and detecting accidental data corruption as checksums etc.[2]

Among all hash functions being used, those from SHA (Secure Hash Algorithm) family covering SHA-0, SHA-1, SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512) have been the most commonly used ones. This SHA family of functions was developed by National Security Agency (NSA) and certified as Federal Information Processing Standard (FIPS) by National Institute of Standards and Technology (NIST), US Department of Commerce. All these are based on MD4 and MD5 algorithms, commonly known as MDx family of hash functions. A few other important hash functions like HAVAL and RIPEMD are also based on MDx family. Around year 2004 and later, majority of hash functions based on MDx family (MD4, MD5, HAVAL, RIPEMD, SHA-0 and SHA-1) were attacked and cryptanalysis of these functions brought serious weaknesses in them to the fore.[2] Given that SHA-2 functions are in the same family and share a common heritage and design principles as the earlier broken functions, these attacks shook the long term confidence of cryptographers in nearly all hash functions. A question that perturbed everybody's mind was what if SHA-2 is compromised or successfully cryptanalyzed or broken and what could be its repercussions? If this proved

[1] R. Sobti, A. Bagga and G. Ganesan, "Security of Online Social Networks," in *International Conference on Control, Communication, Computer & Mechanical Engineering*, New Delhi, 2012

[2] R. Sobti and G. Ganesan, "Cryptographic Hash Functions: A Review," *IJCSI International Journal of Computer Science,* vol. 9, pp. 461-479, 2012

true, the world would not be left with any option because SHA-2 was the best that we had at that time.

To handle this situation, NIST, initiated a design competition (public open competition) in November 2007 for designing next generation of hash functions. The objective of the competition was to design a new hash standard named 'SHA-3' to augment current standard (SHA-2). NIST received 64 hash function submissions from over 200 cryptographers around the world. NIST also invited the public to evaluate the submissions and consequently a lot of cryptanalysis and public review were carried out.[3] In December 2010, five algorithms (Blake, Grøstl, JH, Keccak, and Skein) advanced to the final round.

The 'Reference platform' announced by NIST for SHA-3 competition consisted of general purpose machine (Windows Intel machines). Considerable domain of architectures like the ones prevalent in Smart Cards, Embedded systems, and Mobile platforms were ignored.

The first objective of this thesis revolves around these five SHA-3 final round candidate algorithms and analysis of their performance on architecture other than the one specified in 'Reference platform' and thus in its way contribute to NIST's public call to evaluate and compare performance of these candidate algorithms on platform other than Reference platform. For this objective, ARM architecture was selected as the Target platform for evaluation. The choice of the Target platform was a two-step decision. In the first step, the decision to go for embedded and mobile platform was directed by the recent surge in usage of these devices. In the second step, for zeroing down on ARM architecture, its market dominance and technical features were the main consideration. The rationale behind selection of ARM architecture is thoroughly discussed in the thesis. ARM Cortex-A8, Cortex-M4, and ARM7TDMI processors were picked to cover all possible range of processor series from ARM portfolio. Evaluation on Cortex-A8 and Cortex-M4 was done on hardware (OpenBoard-AM335x and Stellaris® LM4F232 Evaluation Board respectively) and for ARM7TDMI, evaluation was done on simulator. The Methodology and approach used for evaluation on these processors differ from each other because of the nature of these devices. Cortex-A8 based device runs Linux OS and

[3] S. Chauhan, R. Sobti and S. Anand, "Cryptanalysis of SHA-3 Candidates - A Survey," *Research Journal of Information Technology,* vol. 5, no. 2, pp. 149-159, 2013

has MMU, while Cortex-M4 based device is a bare machine without any Operating system. For Cortex-A8, Coprocessor CP15's registers were accessed to compute cycles and for Cortex-M4, Data Watchpoint and Trace Unit's CYCCNT counter was accessed for computing cycles. The tools, methodologies, and approach used for different processor series are detailed in the thesis. Cycles per Byte (CPB) was used as performance metric and results were obtained for short and long messages separately as defined in KAT (Known Answer Test) values specified by NIST. Comparison was done for 224, 256, 384 and 512-bit hash values.

The obtained results reflect that, for almost all algorithms, CPB consumption by 224-bit and 384-bit hash match with those of 256-bit and 512-bit hash respectively. As input size increases, consumption of Cycles per Byte decreases for almost all algorithms on all ARM architectures.[4,5,6,7] This trend is prominently visible in Skein and Grøstl. Skein, Blake, and Keccak showed good performance on all ARM processors. Position of the **best performer** or **No. 2 / 3** performer changes with change in *hash size* or *input message type (long or short messages)* or *ARM processor used* for evaluation. For example, for long messages on Cortex-A8, Skein exhibits the best performance with average of 325 CPB, followed by Keccak at No. 2 with 360 CPB, and Blake at No. 3 with 367 CPB to generate 512-bit hash size. With the same ARM processor (Cortex-A8) and input message type (long messages), but for different hash size (256-bit in place of 512-bit), the relative performance of algorithms changes. In this case, Blake outperforms others with average of 197 CPB, followed by Keccak at No. 2 with 207 CPB, and Skein at No 3 with average of 325 CPB. Performance of JH and Grøstl is found to be quite slow compared to other three algorithms on all processor series of ARM. Grøstl is No. 4 performer on all ARM processors.[5,6,7] Depending on *hash size*, *input message type* and *ARM processor* used, Grøstl takes 47% to 220% more CPB than No. 3 performer. [5,6,7] JH is even slower and

---

[4] R. Sobti, G. Ganesan and S. Anand, "Performance Comparison of Grøestl, JH and BLAKE – SHA-3 Final Round Candidate Algorithms on ARM Cortex M3 Processor," in *2012 International Conference on Computing Sciences*, 2012

[5] R. Sobti and G. Ganesan, "Performance Comparison of Keccak, Skein, Grøstl, Blake and JH: SHA-3 Final Round Candidate Algorithms on ARM Cortex A8 Processor," *International Journal of Security and Its Applications,* vol. 9, no. 12, pp. 353-370, December 2015

[6] G. Singh and R. Sobti, "SHA-3 Blake Finalist on Hardware Architecture of ARM Cortex A8 Processor," *International Journal of Computer Applications,* vol. 123, no. 13, pp. 22-27, August 2015

[7] R. Sobti and G. Ganesan, "Performance Evaluation of SHA-3 Final Round Candidate Algorithms on ARM Cortex-M4 Processor," *International Journal of Information Security and Privacy.* (Accepted for Publication)

depending on the *hash size*, *input message type*, and *ARM processor*, consumes between 29% to 235% more CPB than Grøstl.[5,6,7] Skein exhibits another important characteristic that CPB does not increase as we increase the size of message digest from 224/256 bits to 384/512 bits. On ARM architeture as a whole, use of Skein with 512-bit hash is recommended for long messages whereas Blake and Keccak are recommended for short messages like Password hashing.

The second objective of the thesis is to design a hash function that can act as a variant to Skein hash family and perform better than Skein on 'Reference platform' (announced by NIST) and 'Target platform' (chosen in this study for evaluation of SHA-3 final round candidate algorithms). To achieve the second objective, this study presents a new primitive named **M**odified **C**haCha **C**ore (MCC) that can be used to construct a stream cipher or block cipher or compression function of cryptographic hash function. MCC is an improvisation over Salsa and ChaCha core and experiment conducted during this study reflects that MCC creates more diffusion than its counterparts. On an average, MCC′s Quarter round results in gain of 16% and 88% over Quarter rounds of ChaCha and Salsa respectively.[8] This study also contributes to improvement in Salsa and ChaCha core by evaluating their diffusion properties for different rotation constants and suggest alternative constants that result in more diffusion.

Using MCC, a new ARX based hash function named as ***Cocktail*** is presented that builds its compression function from this core (MCC) and infuses sub-keys in alternate rounds. This newly function ***Cocktail,*** the proposed algorithm, is a simple, flexible, and efficient hash function that blends security with speed. ***Cocktail*** can work on 32-bit as well as 64-bit word size and generates hash output of variable sizes. It is secure and does not suffer from any generic attack. The thesis presents detailed specifications of ***Cocktail***.

---

[5] R. Sobti and G. Ganesan, "Performance Comparison of Keccak, Skein, Grøstl, Blake and JH: SHA-3 Final Round Candidate Algorithms on ARM Cortex A8 Processor," *International Journal of Security and Its Applications,* vol. 9, no. 12, pp. 353-370, December 2015

[6] G. Singh and R. Sobti, "SHA-3 Blake Finalist on Hardware Architecture of ARM Cortex A8 Processor," *International Journal of Computer Applications,* vol. 123, no. 13, pp. 22-27, August 2015

[7] R. Sobti and G. Ganesan, "Performance Evaluation of SHA-3 Final Round Candidate Algorithms on ARM Cortex-M4 Processor," *International Journal of Information Security and Privacy.* (Accepted for Publication)

[8] R.Sobti and G. Ganesan, "Analysis of Quarter Rounds of Salsa and ChaCha Core and Proposal of an Alternative Design to Maximize Diffusion," *Indian Journal of Science and Technology*, vol. 9, no. 3, January 2016

In addition to a normal hash function, ***Cocktail*** can be used in different other operating modes like HMAC, CMAC, Randomized hashing, PRF ensemble. ***Cocktail*** may be used for various security applications that require use of hash functions like Digital Signature, Pseudo Random Number Generator, Password hashing, Digital Time Stamping, Identifying file or data, Verifying File Integrity etc. ***Cocktail's*** compression function can also be used to construct block cipher, processing 512-bit message blocks, in any operation mode like ECB (Electronic Codebook) or CBC (Cipher Block Chaining) or CFB (Cipher Feedback) mode. ***Cocktail*** may also work with Random Key Chaining Mode (RKC), an authenticated encryption mode designed by these researchers, which is enlisted as one of the 14 recommended authenticated encryption modes by NIST, Computer Security Division, U.S. Department of Commerce [9].

On the performance front, ***Cocktail*** can generate 256-bit message digest with average speed of 13 Cycles per Byte and 512-bit message digest with average speed of 8.7 Cycles per Byte on Intel x86_64 Architecture. It can achieve considerably high level of parallelism and performs better than Skein on ARM architecture (Target platform opted) and x86 architecture (Reference platform). ***Cocktail*** consumes 45% lesser operations than Skein. In comparison to Skein, ***Cocktail*** consumes 15% to 87% lesser Cycles Per Byte depending on the architecture used and size of hash output. Besides Skein, ***Cocktail*** performs faster than SHA-3 winner (Keccak) as well as other SHA-3 final round candidate algorithms on Intel x86 and ARM platform.

---

[9] P. Kaushal, R. Sobti and G. Ganesan, "Random Key Chaining (RKC): AES Mode of Operation," *International Journal of Applied Information Systems,* vol. 1, no. 5, pp. 39-45, February 2012.

ACKNOWLEDGEMENTS

It is a pleasure for me to thank all those who have helped me to accomplish this Ph.D. thesis. In the first place, I wish to express my deepest gratitude to Dr. G. Geetha for guiding me throughout this research work. She was the one who introduced me to research in cryptography that has now turned into my passion. She has been a great adviser and an outstanding support.

I am pleased to acknowledge the help, assistance, and support of Sami Anand, Puneet Kaushal, and Shilpa Chauhan, my M.Tech students; Mr. Nikesh Bajaj, my ex-colleague from School of Electronics and Communication Engineering; Col. T. S. Sangha from Content Development Cell, Mr. Gurpreet from Imbuent Technologies, and the ever helpful Mr. Varun Kumar from Department of Mathematics, LPU.

I would like to extend my special thanks to my dearest friend Aman, who has always been a great support and encouraged me when it mattered the most - the time when little was known and much was obscure and doubted.

I am grateful to the Management of LPU - Mr. Ashok Mittal, Ms. Rashmi Mittal, and my seniors - Dr. Lovi Raj Gupta and Dr. Sanjay Modi - for all the support and encouragement they have been rendering from so long.

I also want to express my thanks to my friends and colleagues at LPU, Mr. Gaurav Sethi and Mr. Navdeep Dhaliwal, for their encouragement, interest, and comments. Many thanks go to my colleagues Mr. Amandeep Nagpal, Mr. Lalit Bhalla, and Mr. Hitesh, who helped me to concentrate on my work by shouldering some of my administrative responsibilities.

Special thanks go to examiners of term end reports and reviewers of the journals who vetted my submissions and gave valuable comments to improve the work further. I am really thankful to SERSC Korea branch for providing the necessary financial support towards publishing research on evaluation of Algorithms on ARM Cortex-A8.

Finally, I wish to express my profound gratitude to my mother, father, wife, son and all other members of my family. Their love, support and unshakable faith in me provides umpteen strength to succeed in all the goals of the life. I take this opportunity to express my gratitude to all my teachers who have shaped me and have contributed immensely to my knowledge and skills development since my childhood.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Status of Cryptographic Hash Functions in Cryptology 4

1.2 Formal Definition of Cryptographic Hash Functions 5

1.3 Cryptographic Competitions 7

1.4 Motivation, Research Gap, and Objectives 9

1.5 Contribution to the Field of Computer Science 13

1.6 Outline and Main Contributions 14

CHAPTER 2: REVIEW OF LITERATURE

2.1 Security Services of Cryptographic Hash Functions 15

2.2 Iterative Structure of Cryptographic Hash Functions 20

2.3 Security Properties of Cryptographic Hash Functions 26

2.4 Methods of Attack on Cryptographic Hash Functions 30

2.5 Type of Cryptographic Hash Functions Based on Design of Underlying Compression Function 40

2.6 Migration from SHA-2 to SHA-3 50

2.7 About This Thesis 53

CHAPTER 3: PERFORMANCE ANALYSIS OF SHA-3 FINAL ROUND CANDIDATE ALGORITHMS

3.1 Selection of Target Platform and the Rationale behind It 54

3.2 A Brief about ARM Architecture 59

3.3 ARM Processor Portfolio and Finalization of Processor(s) for Analysis 63

3.4 Introduction to SHA-3 Final Round Candidate Algorithms 66

3.5 Performance Analysis of Algorithms on ARM Cortex-A8 Processor 71

3.6 Performance Analysis of Algorithms on ARM Cortex-M4 Processor 93

3.7 Performance Analysis of Algorithms on Classical Processor-ARM7TDMI 102

3.8 Concluding Remarks on Performance of SHA-3 Final Round Candidate Algorithms on ARM Architecture 108

CHAPTER 4: DESIGNING AND DEVELOPMENT OF ‘Modified ChaCha Core’ - A CRYPTOGRAPHIC PRIMITIVE, LEADING TO THE DESIGN OF '*Cocktail*' – AN ARX BASED NEW HASH FUNCTION

4.1 Analysis of Quarter Rounds of Salsa and ChaCha Core and Proposal of an Alternative Design (MCC) for Maximizing Diffusion 112

4.2 Introduction to ***Cocktail*** and Description of Notations and Operations Used 131

4.3 Iterated Construction of ***Cocktail*** 134

4.4 Specifications of ***Cocktail-512*** 135

4.5 Specifications of ***Cocktail-1024*** 144

4.6 Complexity of ***Cocktail*** 147

4.7 Design Philosophy, Design Decisions, and Its Rationale 148

4.8 Using ***Cocktail*** 166

4.9 Security Aspects of ***Cocktail*** 172

4.10 Concluding Remarks 176

CHAPTER 5: *Cocktail's* PERFORMANCE COMPARISON WITH OTHER SHA-3 FINALISTS

5.1 Performance Results of ***Cocktail*** 178

5.2 Comparison of ***Cocktail*** and Skein 182

5.3 ***Cocktail*** and SHA-3 final Round Candidate Algorithms 193

5.4 Concluding Remarks 196

CHAPTER 6: CONCLUSIONS AND FUTURE ENHANCEMENTS

6.1 Conclusion 198

6.2 Future Enhancements 201

LIST OF TABLES

Table 1. 12 Secure Schemes to Design Hash Function from Block Cipher 42
Table 2. List of Processors Selected for Evaluation of SHA-3 Final Round Candidate Algorithms 66
Table 3. Details of c9 Register of CP15 Coprocessor 78
Table 4. Mean, Median and Quartiles of CPBs Consumed by SHA-3 Finalists on Cortex-A8 91
Table 5. Comparison of Skein-512 and Skein-1024 for Generating 512-bit Hash 92
Table 6. Mean, Median and Quartiles of CPBs Consumed by SHA-3 Finalists on Cortex-M4 99
Table 7. Comparison of Performance on Cortex-A8 and Cortex-M4 102
Table 8. Mean, Median, and Quartiles of CPBs Consumed by SHA-3 Finalists on ARM7TDMI 106
Table 9. Diffusion Matrix 121
Table 10. Diffusion Matrix of Salsa's Quarter Round 123
Table 11. Diffusion Matrix of ChaCha's Quarter Round 123
Table 12. Top 12 Sets of Rotation Constants with Mean ≥ 7.70 127
Table 13. Diffusion Matrix of MCC's Quarter Round 127
Table 14. ***Cocktail's*** Overview 132
Table 15. Round Constants for Key Derivation Words(***Cocktail-512***) 139
Table 16. Diffusion Matrix for One Double Round of MCC 143
Table 17. Round Constants for Key Derivation Words (***Cocktail-1024***) 146
Table 18. Diffusion Matrix for MCC's Quarter Round (64-bit version) 156
Table 19. Diffusion Matrix of One Double Round of MCC (64-bit version) 157
Table 20. Spread of Diffusion in Four Column Quarter Rounds followed by Four Row Quarter Rounds 162
Table 21. Spread of Diffusion in Interlaced Column and Row Quarter Rounds 162
Table 22. Performance Results of ***Cocktail*** on Intel x86 Architecture 180
Table 23. Performance Results of ***Cocktail*** on ARM Architecture 181

LIST OF FIGURES

Figure 1. Types of Cryptography 3
Figure 2. Hash Function SHA-2 Generating 256-bit Hash Result 6
Figure 3. Contribution of the Thesis in the Field of Computer Science as per ACM CCS 13
Figure 4. Usage of Hash Function in Implementing Digital Signature (RSA approach) 17
Figure 5. Merkle-Damgard Construction 20
Figure 6. Wide Pipe Design 22
Figure 7. The Sponge Construction for Hash Functions 24
Figure 8. Twin Pipe Based 3C Construction 26
Figure 9. Classification of Attacks on Hash Functions 31
Figure 10. Compression Function Based on Block Cipher 41
Figure 11. History of MDx-type Hash Functions. 49
Figure 12. Time Spent with Digital Media in USA 56
Figure 13. Time Spent with Internet, by Device, in the USA 56
Figure 14. ARM Market Share as per ARM Holdings 2012 Q4 results 58
Figure 15. ARM Dominance in Different Market Segments 59
Figure 16. ARM Register Set 61
Figure 17. Program Status Register Fields 61
Figure 18. Four of the 72 Rounds of Threefish 512 68
Figure 19. Working of Grøstl 68
Figure 20. Local Wide Pipe of Blake 69
Figure 21. Compression Function of JH 70
Figure 22. Cortex-A8 Based OpenBoard-AM3359 from PHYTEC 72
Figure 23. Host and Target Machine Setup for OpenBoard-AM335x 73
Figure 24. Format of Performance Monitor Control Register 79
Figure 25. Format of Count Enable Set Register 80
Figure 26. Format of User Enable Register 82
Figure 27. Results on Cortex-A8 for Short Messages (224-bit Message Digest) 85
Figure 28. Results on Cortex-A8 for Short Messages (256-bit Message Digest) 86
Figure 29. Results on Cortex-A8 for Short Messages (384-bit Message Digest) 86
Figure 30. Results on Cortex-A8 for Short Messages (512-bit Message Digest) 87
Figure 31. Results on Cortex-A8 for Long Messages (224-bit Message Digest) 88
Figure 32. Results on Cortex-A8 for Long Messages (256-bit Message Digest) 88
Figure 33. Results on Cortex-A8 for Long Messages (384-bit Message Digest) 89
Figure 34. Results on Cortex-A8 for Long Messages (512-bit Message Digest) 89

Figure 35. Cortex-M4 Based Stellaris LM4F232 Board from Texas Instruments 94
Figure 36. Host and Target Machine Setup for Stellaris EK-LM4F232 95
Figure 37. Results on Crotex-M4 for Short Messages (256-bit Message Digest) 98
Figure 38. Results on Cortex-M4 for Short Messages (512-bit Message Digest) 98
Figure 39. Results on Cortex-M4 for Long Messages (256-bit Message Digest) 100
Figure 40. Results on Cortex-M4 for Long Messages (512-bit Message Digest) 101
Figure 41. Results on ARM7TDMI for Short Messages (256-Bit Message Digest) 104
Figure 42. Results on ARM7TDMI for Short Messages (512-Bit Message Digest) 105
Figure 43. Results on ARM7TDMI for Long Messages (256-bit Message Digest) 106
Figure 44. Results on ARM7TDMI for Long Messages (512-bit Message Digest) 106
Figure 45. Performance of Blake, Keccak, and Skein for Long Messages on Cortex-A8 (512-bit Message Digest) 109
Figure 46. Quarter Round of Salsa Core 115
Figure 47. Quarter Round of ChaCha Core 117
Figure 48. Quarter Round of MCC 119
Figure 49. Mean and Standard Deviation of Diffusion Matrices of Salsa's Quarter Round 124
Figure 50. Mean and Standard Deviation of Diffusion Matrices of ChaCha's Quarter Round 125
Figure 51. Mean and Standard Deviation of Diffusion Matrices of MCC's Quarter Round 126
Figure 52. Sequence of Parameters Ensure Uniform Diffusion 128
Figure 53. Comparison of Quarter Rounds of Salsa Core, ChaCha Core and MCC 129
Figure 54. Zoomed Version of Comparison of Three Designs for Mean ≥ 7.0 130
Figure 55. Iterative Structure of ***Cocktail*** Hash Function 135
Figure 56. Message 'M' after Padding in ***Cocktail-512*** 135
Figure 57. Structure of 10 Round Compression Function ***Cocktail*** 138
Figure 58. Process of Computing Key Derivation Words 140
Figure 59. Working of Column and Row Round of ***Cocktail-512*** 142
Figure 60. The Output Transformation (O) of ***Cocktail*** 144
Figure 61. Message 'M' after Padding in ***Cocktail-1024*** 145
Figure 62. Mean and Standard Deviation of Diffusion Matrices of MCC's Quarter Round (64-bit) 156
Figure 63. HMAC with ***Cocktail*** for Achieving Message Integrity and Authentication 167
Figure 64. CMAC using ***Cocktail's*** Compression Function 168
Figure 65. Password Based Authentication Using ***Cocktail*** 169
Figure 66. Performance of Skein and ***Cocktail*** for 512-bit Internal State on Intel x86_64 Machine 187
Figure 67. Performance of Skein and ***Cocktail*** for 1024-bit Internal State on Intel x86_64 Machine 187
Figure 68. Performance of Skein and ***Cocktail*** for 512-bit Internal State on Intel x86_32 Machine 188
Figure 69. Performance of Skein and ***Cocktail*** for 1024-bit Internal State on Intel x86_32 Machine 188

Figure 70. Performance of Skein and ***Cocktail*** for 512-bit Internal State on Cortex-A8 Processor 189
Figure 71. Performance of Skein and ***Cocktail*** for 1024-bit Internal State on Cortex-A8 Processor 189
Figure 72. Performance of Skein and ***Cocktail*** for 512-bit Internal State on Cortex-M4 Processor 190
Figure 73. Performance of Skein and ***Cocktail*** for 1024-bit Internal State on Cortex-M4 Processor 190
Figure 74. Performance of Skein and ***Cocktail*** for 512-bit Internal State on ARM7TDMI Processor 191
Figure 75. Performance of Skein and ***Cocktail*** for 1024-bit Internal State on ARM7TDMI Processor 191
Figure 76. Comparison of Skein and ***Cocktail's*** Recommended and Extended Rounds on Different Architectures (512-bit Internal State) 193
Figure 77. Comparison of Skein and ***Cocktail's*** Recommended and Extended Rounds on Different Architectures (1024-bit Internal State) 193
Figure 78. Comparison of ***Cocktail*** and All SHA-3 Final Round Candidate Algorithms 195
Figure 79. Performance of JH and Grøstl on Various Platforms 196

LIST OF APPENDICES

APPENDIX I. Guidelines to Setup and Use MiniCom 222

APPENDIX II. Inline Assembly and Kernel Module Used for Accessing CP15 Registers 225

APPENDIX III. Using Code Composer Studio 229

APPENDIX IV. Using IAR Embedded Workbench 233

APPENDIX V. Process of Generating Initial Values for ***Cocktail*** 236

APPENDIX VI. Full Diffusion in ***Cocktail*** and Skein 237

APPENDIX VII. Test Vectors of ***Cocktail*** 252

CHAPTER 1: INTRODUCTION

"The last thing that we find in making a book is to know what we must put first."

Blaise Pascal

The digital era, with dominance of internet in all spheres of human activities, bears testimony to a paradigm shift in creation, exchange, and preservation of information. The modes of interacting with people, reading news, playing games, listening songs, shopping, transacting businesses, financial payments, preparing documentation, giving instructions, making representations have all witnessed a total revolution. Information, in the contemporary society, has become a valuable asset as never before and keeping the information secure has become equally crucial.

Earlier, all these were not only dependent on human beings, their messengers, drum beaters, trained flight carriers be it pigeons, hawks and eagles but were also restricted. The restrictions were either due to the mindfulness or due to the lack of resources or competency on how to make, transact, give, and share information. The creation of cyber space, evolution of internet, availability and accessibility of high speed networks, and addictive use of technology have all contributed to unprecedented proliferation of information. With all this, challenges in securing information have also increased manifold. The security aspect is important for all the users of information be it an individual, a business, an organization, a government or the world as a whole.

The principal objectives of security of information are directed by the need of ensuring **confidentiality** i.e. the information should be rendered unintelligible to an unauthorized person; **integrity** i.e. to ensure that the information does not get altered in storage or during transmission without the knowledge of the sender or the receiver; **authentication** i.e. both the sender and the receiver can identify each other as well as the source and destination of the information; **availability** i.e. the information created and stored by an individual or organization is available to authorized entities when required.

In earlier days, people used different ways to achieve these security objectives. Confidentiality was achieved using sealed envelopes, lockers etc. Signatures, fingerprints

and specific government seals helped in maintaining authenticity of information (integrity as well authentication). In ancient times in India, tallies (a piece of wood scored across with notches for the items of an account and then split into halves, each party keeping one) were used as tool to identify individuals and the business they transacted. Communication used to take place either in person or through confidential emissaries. The chances of eavesdropping were little. **Nowadays**, majority of our data is getting digitized and communication is taking place through phones or internet and thus challenges of maintaining confidentiality, integrity, and authentication are getting bigger and bigger. The data might be present on, or being communicated through, open networks or cloud from where it might be easy for an adversary to read, copy, modify or delete. The adversary could be anyone who wants to do something against the wishes of the communicating parties. Adversaries are enemies and may disrupt and thwart the goals of communication between the two parties. An adversary may exist in the shape of a human or a machine programmed and controlled by any human. So it becomes imperative to develop tools and techniques to protect ourselves from the tricks of our adversaries and outwit them.

Cryptography comes as a rescue and helps us to secure information from these adversaries and thus achieve objectives of security. **Cryptography** is the science of writing in secret codes and thus provides a method of storing and transmitting information so that only those who are intended to receive it can read, understand and process it.

The origin of cryptography technique is dated back to circa 2000 BC., where Egyptians used hieroglyphics, complex pictograms, the full meaning of which was only known to an elite few. Julius Ceaser (100 BC to 44 BC) is also generally credited as the first user of modern cipher. He did not trust his messengers. So, to communicate with his governors and officers, he created a system in which each character in the messages was replaced by a character three positions ahead of it in the Roman alphabet.

With the passage of time, a lot of advancements have taken place in the field of cryptology. Cryptology consists of two complimentary fields: cryptography and related **cryptanalysis**. A cryptographer is concerned with the development of new schemes or algorithms, to protect the security of the information being communicated through any communication channel. A cryptanalyst on the other hand is concerned with the development of attack methodologies that break a cryptographic algorithm, allowing, for

example, unauthorized people access to secret information or gain ability to forge documents.

For many years, **cryptology** in its modern form was practiced exclusively by military and government organizations. History also credits the victory of the Allies in World War II due to the decryption of ENIGMA. United State's National Security Agency (NSA), and its peers in other countries including USSR, UK, Israel, France etc., have spent billions of dollars in securing their own information and communication while trying to access and decrypt everyone else's [1]. Kahn in [2] has also given a comprehensive account of how cryptographic techniques have been used for many centuries to protect military and diplomatic secrets.

In last 30 – 35 years, the scenario has changed considerably and research in cryptography has evolved and is being carried out outside the walls of military and government organizations also. Undoubtedly, classical cryptography has been into practice by ordinary citizens for centuries but computer based cryptography was restricted within the walls of military and governmental organizations. However nowadays, even individuals practice the field of cryptography and can thus protect their data against the most powerful of adversaries.

There are, in general, three types of cryptographic schemes to accomplish the objectives of security.

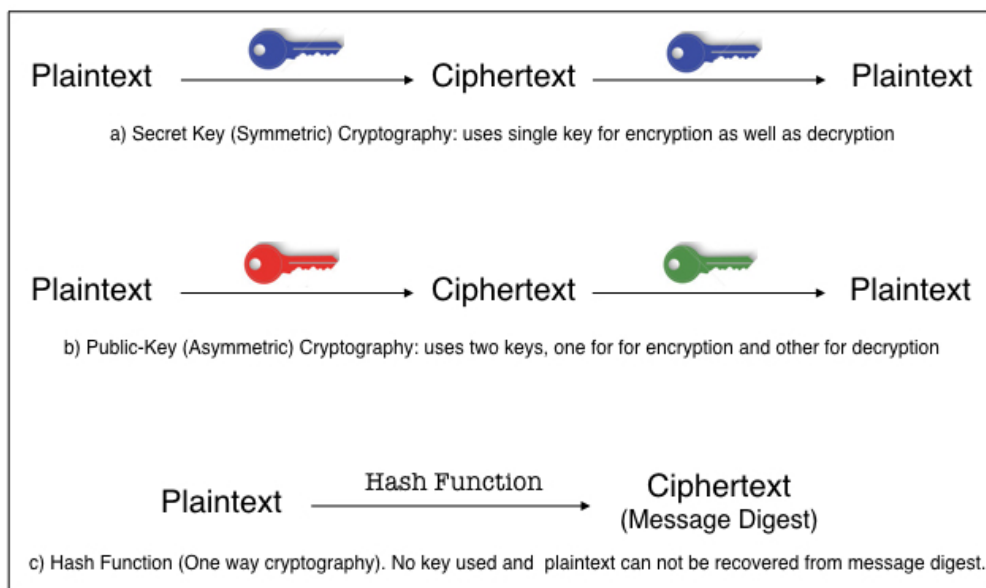


Figure 1. Types of Cryptography

These are:

a) Symmetric Cryptography (also known as Secret key or Private key cryptography)
   - Uses single secret key for both encryption and decryption
b) Asymmetric Cryptography (also known as Public key cryptography)
   - Uses two keys; one key for encryption and the other for decryption (one of the key is secret (private) while the other is public).
c) Cryptographic Hash Functions:
   - Uses mathematical transformation to encrypt information and obtain message digest that is irreversible i.e. plaintext can not be obtained from this message digest.

The main focus of this thesis is third category of cryptographic algorithms i.e. **Cryptographic Hash Functions.**

1.1 Status of Cryptographic Hash Functions in Cryptology

Until mid 70s, the term cryptography referred to safeguarding confidentiality of the information and cryptanalysis referred to breaking the confidentiality. So through out the history of cryptology, **confidentiality** has taken the primary seat and it was believed that if confidentiality / privacy of information is achieved (using symmetric encryption) then other security goals such as **authentication** and **integrity** would be automatically achieved. The logic was, if decryption of an encrypted text results in a meaningful message it must have been constructed by someone who knows the secret key.

The trend changers were Diffie and Hellman, who are credited for advent of public key cryptography in mid 70s. Their seminal paper "New Directions in Cryptography" [3] gave birth to a number of relevant concepts like digital signatures that are still extensively used. Diffie and Hellman differentiated the problem of confidentiality from authentication and to quite an extent initiated the development of cryptographic schemes for the protection of authenticity (authentication and integrity). These schemes make use of cryptographic primitive under discussion in this thesis i.e. **Cryptographic Hash Functions**.

In the past, cryptographic hash functions received much less attention from the cryptologic community than encryption schemes. Rompay in his thesis [4] quoted the example of NESSIE (New European Scheme for Signature Integrity and Encryption) project to illustrate how cryptographic hash functions had been ignored in the past. In

NESSIE project, seventeen block ciphers and six stream ciphers were submitted as candidates (both are categories of encryption schemes), but only one un-keyed hash function and two keyed hash functions (also known as MAC – Message Authentication Code) were submitted. Rompay also gave example of open competition used by the National Institute of Standards and Technology (NIST) in the United States to decide on the block cipher to be used as Advanced Encryption Standard [5]. This competition had fifteen candidates out of which Rijndael block cipher was finally chosen. On the other hand, for its hash function standard [6], NIST simply chose the SHA hash functions, designed by NSA without disclosure of their design strategy or any supporting cryptanalytic results.

However, the trend has changed in recent years because of the wide range of applications areas of cryptographic hash functions. One of the first application of hash function was to construct efficient Digital signatures. Ever since then, hash function has led their way into various security applications like: storing passwords, digital time stamping, constructing block ciphers, generating pseudorandom numbers, maintaining secure web connections, encryption key management, virus scanning, indexing data in hash tables and detecting accidental data corruption as checksums etc. [7]. Cryptographic hash functions also form an integral part in the functionality of other cryptographic tools like key distribution protocols and zero knowledge proofs. Usage of hash functions in achieving security goals in several information processing applications is much more widespread than block or stream ciphers.

Considerable research has been undergoing in the field of cryptographic hash functions. Hash functions are being generated from existing primitives like block ciphers e.g. Whirlpool [8] and Skein [9] as well as being explicitly and specially constructed from scratch like MD4 [10], MD5 [11] and SHA family [12] of hash functions. A new ARX (Arithmetic, Rotation and XOR) based hash function designed using **M**odified **C**haCha **C**ore is also presented in this thesis. The details are given in Chapter 4.

1.2 Formal Definition of Cryptographic Hash Functions

Hash function is defined as a function that processes an arbitrary length input message into fixed length digest known as hash code (also known as hash value or hash result or message digest). However, if it satisfies some additional requirements (as detailed

further), then it can be referred as 'Cryptographic Hash Functions' and can be used for cryptographic applications like ensuring authenticity of message over an insecure communication channel.

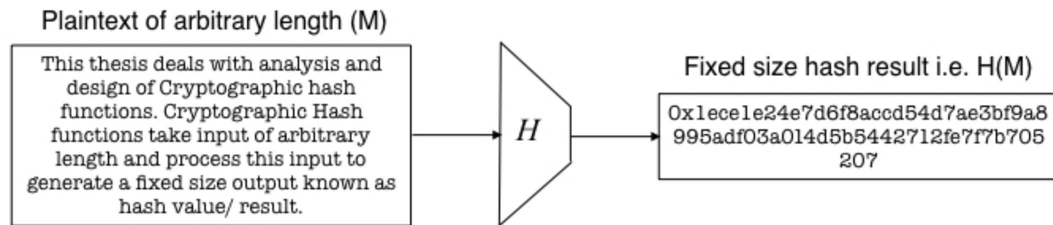


Figure 2. Hash Function SHA-2 Generating 256-bit Hash Result

Formally, hash function may be defined as:

A hash function is a function H: $D \rightarrow R$, *where the domain* $D = \{0,1\}^*$ *and* $R = \{0,1\}^n$ *for some* $n \geq 1$.

Cryptographic hash functions are broadly of two types i.e. Keyed hash functions; the one which uses a secret key, and Un-keyed hash functions; the other one which does not uses a secret key. The Keyed hash functions are referred to as **M**essage **A**uthentication **C**ode (MAC). Generally, the term hash functions refer to un-keyed hash functions and this thesis concentrates on Un-keyed hash functions only**. MACs (i.e. Keyed hash functions)** are commonly constructed from Un-keyed hash functions. Un-keyed or simply Hash functions (sometimes also known as MDC – Manipulation Detection Code) can be further classified into OWHF (One-Way Hash Functions) or CRHF (Collision Resistant Hash Functions) depending on various properties satisfied by them.

1.2.1 One-Way Hash Functions (OWHF)

OWHF as defined by Merkle [13] is a hash function H that satisfies the following requirements:

I. H can be applied to block of data of any length. (In practice, 'any length' may be actually be bounded by some huge constant, larger than any message we ever would want to hash.)

II. H produces a fixed-length output.

III. Given H and M (any given input), it is easy to compute message digest $H(M)$.

IV. Given H and $H(M)$, it is computationally infeasible to find M.

V. Given H and $H(M)$, it is computationally infeasible to find M and M' such that $H(M) = H(M')$

The first three requirements are must for practical applications of a hash function to message authentication and digital signatures. The fourth requirement, also known as **Pre-image resistance or One-way property**, states that it is easy to generate a message code given a message but hard (virtually impossible) to generate a message given a code. The fifth requirement, also known as **Second pre-image resistance** property, guarantees that an alternative message, hashing to the same code as a given message, cannot be found.

1.2.2 Collision Resistant Hash Functions (CRHF)

One of the early definitions of Collision Resistant Hash functions (CRHF) was given by Merkle [14]. Based on the same, CRHF may be defined as a hash function H that satisfies all the requirements of OWHF (As listed in '1.2.1 One-Way Hash Functions (OWHF)') and in addition satisfies the following **collision resistance** property:

VI. Given H, it is computationally infeasible to find a pair (M, M') such that $H(M) = H(M')$

1.3 Cryptographic Competitions

Cryptographic community has a long tradition of open competitions (focused or broader) for identifying new efficient cryptographic primitives that can act as new protocols or standards for government and public use. These competitions and projects have been a great motivator and have helped a lot in understanding of cryptographic primitives and have given a great push to the research in cryptology. The important cryptographic competitions, that have been instrumental in identification of cryptographic primitives and finalization of standards, are listed below:

a) **Process for DES (1973 – 1977):** On 15 May 1973, National Bureau of Standards (NBS) – US standards body, now-a-days known as NIST (National Institute of Standards and Technology), after identifying the need for a government-wide standard for encrypting unclassified sensitive information, invited proposals for a cipher that would meet prescribed design criteria. Candidate submitted by IBM was eventually selected and a modified version of this algorithm was published as a Federal

Information Processing Standard (FIPS) for United States in 1977 [15]. The standard was widely known as Data Encryption Standard (DES).

b) **AES competition (1997 – 2000):** Announced by NIST, AES competition was initiated to replace DES. The competition attracted 15 block cipher submissions from 50 cryptographers around the world. All submissions were subjected to extensive cryptanalysis and finally Rijndael was chosen as Advanced Encryption Standard (AES) [5]. The competition is considered as one of the most successful, open and transparent cryptographic competition. The whole process helped a lot in increasing confidence in the security aspect of winning algorithm which is still being used as encryption standard.

c) **NESSIE project (2000 – 2003): "**New European Schemes for Signatures, Integrity, and Encryption" was a project of European commission with an objective to establish a portfolio of strong cryptographic primitives comprising of algorithms for encryption, authentication and digital signatures. 42 algorithms were submitted and cryptanalysis by cryptographic community went for about 2 years. 12 algorithms were selected as portfolio algorithms. [16]

d) **eSTREAM (2004 – 2007):** A project initiated by ECRYPT - **E**uropean Network of Excellence in Cryptology, an European research initiative, called for submission of new stream ciphers suitable for widespread adoption. This call attracted 34 stream-cipher submissions from 100 cryptographers around the world. Just like AES (but on larger scale), hundreds of security evaluations and performance evaluations were done. Eventually, the eSTREAM committee selected a portfolio containing several stream ciphers. [17]

e) **SHA-3 competition (2007 – 2012):** In Nov 2007, NIST announced an open competition [18] for development of a new cryptographic hash standard named "SHA-3". This competition attracted 64 hash-function submissions from 200 cryptographers around the world, and then a tremendous volume of security evaluations and performance evaluations were carried by the cryptographic community. **This research also contributes to performance analysis of the candidate algorithms.** Eventually NIST chose Keccak [19] as SHA-3.

f) **PHC - Password Hashing Competition (2013 – 2015):** Inspired by the success of AES and SHA-3 competition, "Password Hashing Competition" was announced in

year 2003 to select a new standard algorithm for password hashing. In the light of allegations that NSA may influence NIST for some specific algorithm, this competition was run by independent panel of cryptographer and security practitioners. The competition attracted 24 entries of which Argon2 was selected the winner. [20]

g) **CAESAR (2013 onwards):** A new competition CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) is under progress at the time of writing this thesis. The objective of this competition is to identify a portfolio of authenticated ciphers that (1) offer advantages over AES-GCM and (2) are suitable for widespread adoption. As this thesis is being written, the second round of the competition is going on and out of 56 entries received from about 200 cryptographers around the world, 29 algorithms have advanced to the second round. [21]

Motivation for this research is derived **from SHA-3 competition** that was under progress during the same time when literature for this study was being reviewed and problem formulation was being worked upon.

1.4 Motivation, Research Gap, and Objectives

As stated, this research originated from and was motivated by cryptographic competition conduced by NIST, USA for development of a new cryptographic hash standard named "SHA-3".

The competition was a necessity. Around 2005, majority of commonly used hash functions belonged to SHA (Secure Hash Algorithm) family. These were SHA-0 [22], SHA-1 [23], SHA-2 {SHA-224, SHA-256, SHA-384, SHA-512} [6]. All these were developed by NSA and certified as FIPS by NIST. SHA functions as mentioned above are based on MD4 [10] and MD5 [11] algorithms, commonly known as MDx family of hash functions. The practical attack on MDx family, followed by attack on SHA-0 and SHA-1 were carried out in year 2004 and 2005. Various other attacks like Joux Multicollisions and attack on HAVAL and RIPEMD were also reported. The details are discussed in ‘Chapter 2: Review of Literature’. SHA-2, the best known hash function at that time, also belonged to same family and shared a common heritage and design principles as the earlier broken functions. This created doubt in everybody’s mind, as to what would happen if SHA-2 also gets compromised or successfully

cryptanalyzed/broken. If this proved true, the world would not be left with any reliable hash standard.

Motivated by the big success of AES (Advanced Encryption Standard) competition that took place a few years ago and forced by recent attacks on existing family of widely used hash functions, NIST decided to hold a public competition in search of a new hash standard. The idea was to have the new algorithm parallel to current SHA-2 and this new algorithm is to be used in case the worst came to pass i.e. SHA-2 faces some serious challenge. In Nov 2007, to augment the hash algorithms currently specified in the FIPS 180-4, Secure Hash Standard [12], NIST announced a public competition [18] for development of a new cryptographic hash standard named "SHA-3".

By October 31, 2008, NIST received sixty-four entries (in comparison to the 15 entries received in AES Competition). NIST selected fifty-one algorithms that fitted their guidelines and advanced to the first round on December 10, 2008. There was a lot of cryptanalysis and many hash functions were broken and some were found unappealing on performance parameters. Fourteen algorithms advanced to the second round on July 24, 2009. A year was allocated for the public review of the fourteen second-round candidates. Significant feedback was received from the cryptographic community. Based on the public feedback and internal reviews of the second-round candidates, NIST selected five SHA-3 finalists, which advanced to the third (and final) round of the competition on December 9, 2010. These five SHA-3 finalists were Skein [9], Keccak [19] , Grøstl [24], JH [25] and Blake [26].

The concurrency of on going SHA-3 competition and Literature review for this study had a compelling influence in finalizing the problem formulation and its research objectives. The following two objectives were finalized:

1.4.1 Objective 1: Performance Analysis of SHA-3 Final Round Candidate Algorithms

A) Motivation and Research Gap

For SHA-3 competition, NIST had announced a 'Reference Platform and Complier' [27] on which the candidate algorithms were to be evaluated. The Reference platform and compiler, announced by NIST, was Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit

(x86) and 64-bit (x64) Edition and ANSI C compiler in the Microsoft Visual Studio 2005 Professional Edition. Because of this Reference platform, there was high probability that final round candidates had optimized their code for the Reference platform and compiler.

But a considerable domain of architecture like 8 /16 /32-bit architectures used by Smart Cards, Embedded systems, Portable devices like mobile phones etc. might have skipped the attention of cryptographers of final round. It may happen that a seemingly fast function on Reference platform is not performing well on embedded system or mobile devices. So a performance analysis of SHA-3 final round candidate algorithms on other platforms is desired. NIST notice introducing the competition [27] also invited the public to evaluate candidate algorithms on platforms other than Reference platforms and compare results. Motivated by NIST's public call, we decided to contribute to the efforts of analysing the final round candidate algorithms (Cryptographic Hash functions) that would be used for multiple applications in future on variety of platforms.

B) Scope

Scope of the first objective is to analyse performance of SHA-3 final round candidate algorithms on any one of the platforms other than the one specified as 'Reference platform' by NIST. The platform may be ARM processors or 8-bit/16-bit machine or various languages and compilers like C or Java.

1.4.2 Objective 2: Design of a New Hash Function that can Act as a Variant to Skein Hash Family

A) Motivation and Research Gap

Motivated by the current scenario of existing cryptographic hash functions and excited by SHA-3 competition, we decided to contribute to the field of cryptography in general and hash function in particular by designing of a new hash function that can perform better than the existing algorithms under discussion.

Skein (one of the finalists of SHA-3) as submitted by Schneier and his team is a family of hash functions with three different internal state sizes: 256-bit, 512-bit and 1024-bit [9]. In October 2010, an attack combining rotational cryptanalysis with rebound attack was published by Khovratovich and his team. The attack found rotational collisions for 53 of 72 rounds in Skein-256, and 57 of 72 rounds in Skein-512. It affected Threefish cipher [28]. This was a follow-up to the earlier attack published in February 2010 which

broke 39 and 42 rounds respectively [29]. So, Skein was **initially found to be prone to Rotational Cryptanalysis**. We thought that an alternative was desired and it motivated us to design a new hash function that can act as a variant to Skein hash family.

However, during revised submissions, Schneier and his team tweaked (key schedule constant was changed) Skein to overcome the threats posed by rotational cryptanalysis. Major findings that came out after going through various papers in initial months of 2012 were that, the security five SHA-3 finalists had been carefully analysed in the last 2 – 3 years and cryptographic community did not expect serious flaws or vulnerabilities to be found in any of them. NIST's computer scientists Quynh Dang & Tim Polk in their presentation during 83rd meeting of the Internet Engineering Task Force (IETF) reinstated the similar facts [30] as given below:

a) Confidence in the security of SHA-3 candidates is very high.
b) **SHA-3 Candidates are based on new constructions and thus are not vulnerable to well known attacks on MD5, SHA-1 and SHA-2 and on Merkle-Damgard construction (e.g. length extension attacks) on which all three are based**.
c) NIST also may not expect across the board performance increase from SHA-3 as they got with AES (Advanced Encryption Standard) compared with triple DES.
d) SHA-3 could be faster than SHA-2 in some environments e.g. Skein & Blake is faster in software on high-end computing platforms and Keccak is fast in hardware in general.
e) SHA-3 potentially offers significantly better performance for hash based MAC on short message.
f) SHA-256 is competitive in low-end SW platforms and "Constrained" HW. Depending on the selected algorithms,
g) SHA-2 collision resistance seems fine but **SHA-3 candidates have greater security margins**.
h) **All candidates have much higher multi-collision resistance than SHA-2** and fix other generic limitations of Merkle-Damgard.

The above evidence was enough to conclude that that cryptographic community did not expect serious flaws or vulnerabilities to be found in any of SHA-3 finalists. No

algorithm was knocked out by cryptanalysis and different algorithms had different depth of cryptanalysis. However, performance-characteristics vary from one platform to other.

From the available scenario, it looked quite evident that NIST's decision to select SHA-3 algorithm may be based on performance of the finalists in hardware and software. The trend was shifting towards performance. It was also visible in the research being carried out at that time. The cryptographic community had been putting much efforts dedicated to optimizing software or hardware implementations of the SHA-3 finalists on different platforms. So, with the aforementioned proceedings in mind, it was decided to focus on performance characteristics of the new hash function to be developed for achieving the second objective of this research.

B) Scope

Scope of second objective is to design a new hash function that can act as a variant to Skein Hash family and perform better than Skein on Reference platform prescribed by NIST and Target platform selected for the first objective.

1.5 Contribution to the Field of Computer Science

The following Venn Diagram (Figure 3) represents the contribution of this thesis to the field of Computer Science as per ACM Computing Classification System.

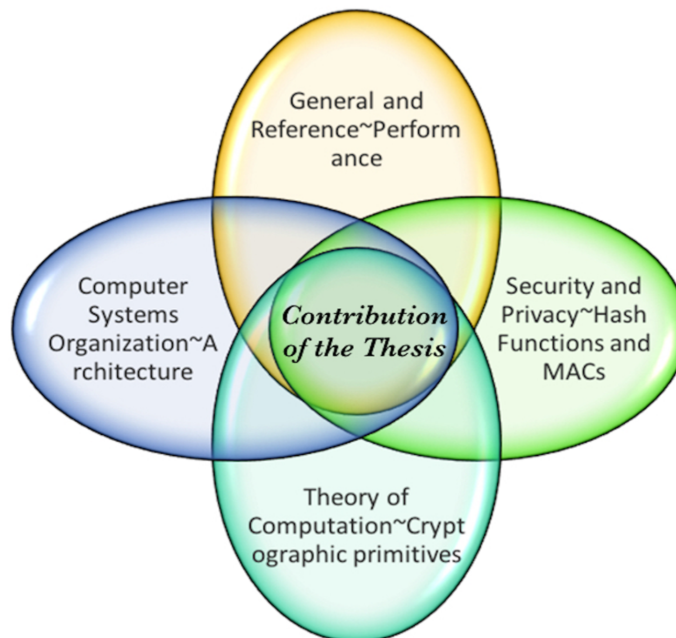


Figure 3. Contribution of the Thesis in the Field of Computer Science as per ACM CCS

1.6 Outline and Main Contributions

The outline of the thesis is as below:

⇒ **Chapter 1** introduces the thrust of our thesis and the background of the motivation of our research on cryptographic hash functions.

⇒ **Chapter 2** presents the review of literature and is organized as applications of hash functions, their internal structures, their security properties, ways of designing the hash functions and various type of attacks that hash functions are prone to. This study of cryptographic hash functions was also published as a review paper in [7].

⇒ **Chapter 3** concentrates on the first objective of the research i.e. Performance analysis of SHA-3 final round candidate algorithms - Keccak, Skein, Grøstl, Blake and JH. The chapter provides a detailed explanation of the rationale behind selection of Target platform for performance analysis of these algorithms, gives a brief introduction to Target platform and these algorithms, discusses methodology, and shares results of performance of these algorithms on Target platform.

⇒ **Chapter 4** fulfils the second objective of this thesis and presents a new ARX based hash function function (named ***Cocktail***) that has been designed using **M**odified **C**haCha **C**ore. Initial sections concentrate on design and performance of **M**odified **C**haCha **C**ore and in later part of the chapter, specifications of new hash function (***Cocktail***) are given followed by its design philosophy and rationale behind design decisions. The chapter ends with discussion on various modes of using ***Cocktail*** and its security aspects.

⇒ **Chapter 5** presents the performance results of ***Cocktail*** on Intel x86 and ARM architecture. The Comparison of ***Cocktail*** with Skein and other SHA-3 final round candidate algorithms is also discussed in this Chapter.

⇒ In **Chapter 6,** Conclusion and future enhancements are listed.

CHAPTER 2: REVIEW OF LITERATURE

"Literature is an investment of genius which has dividends to all subsequent times."

John Burroughes

Cryptographic Hash function (also known as Hash function), an important cryptographic primitive, is used extensively to achieve number of security goals. The literature reviewed for this study is presented by organizing this chapter into following headings:

- Security Services of Cryptographic Hash Functions **(2.1)**
- Iterative Structure of Cryptographic Hash Functions **(2.2)**
- Security Properties of Cryptographic Hash Functions **(2.3)**
- Methods of Attack on Cryptographic Hash Functions **(2.4)**
- Types of Cryptographic Hash Functions based on Design of Underlying Compression Function **(2.5)**
- Migration from SHA-2 to SHA-3 **(2.6)**

2.1 Security Services of Cryptographic Hash Functions

The review of literature, in relation to various security services and applications of cryptographic hash functions is detailed here:

2.1.1 Achieving Integrity and Authentication

Integrity and Authenticity is of utmost importance in computer systems and networks. Two parties communicating over an insecure channel always require mechanism to validate the integrity and authenticity of information being communicated. There are multiple ways to implement message Integrity and authentication. Symmetric Encryption based mechanisms may be used but they have their own drawbacks.

Tsudic in [31] has highlighted drawbacks like speed, cost factor, optimization for data sizes etc. Such methods combine the confidentiality and authentication functions. However, there are instances where encrypting full message (confidentiality) is not

required. For such applications keeping message secret is not important as authenticating it is. For example, in SNMP (Simple Network Management Protocol), it is usually important for a managed system to authenticate incoming SNMP commands (like changing the parameters of the managed system), though concealing the SNMP traffic is not essential. **Tsudic** in [31] has detailed a protocol for achieving message authentication and integrity goals with one-way hash functions without the use of symmetric encryption.

Rompay in [4] has also detailed the ways of ensuring authentication using hash functions alone as well as using hash functions with encryption.

Bellare et al. in [32] have stated that one of the alternative ways to implement integrity and authentication is with MACs but even MACs are being constructed extensively using cryptographic hash functions. As per the authors, usage of hash functions for message authentications and ensuring message integrity has surged because majority of hash functions are faster than block ciphers in software implementation and these software implementations are readily and freely available.

2.1.2 Implementing Efficient Digital Signature

In a cryptosystem, Digital Signatures achieve goal of authenticity and also provide the security service of non-repudiation. Non-repudiation means the creator or sender of information can not deny his/her role or intentions in creation or transmission of the information.

Diffie and Hellman in [3] were the first one to realize the need of a message dependent electronic signature (fingerprint) so that disputes between sender and receiver could be avoided and RSA by **Rivest et al.** [33] was the first public key cryptosystems with digital signature capabilities.

However, **Singh** in [34] has put forward an interesting aspect of this invention. As per S. Singh; James Ellis, Clifford Cocks, and Malcolm Williamson from GCHQ (Government Communication Head Quarters), Cheltenham, Britain perhaps invented the idea of Public key in 1972. The three Britons had to sit back and watch as their discoveries were rediscovered by Diffie, Hellman, Merkle, Rivest, Shamir, and Adleman over the next three years because of the policies of GCHQ that all work was top secret and could not be shared with anyone.

Use of hash functions to optimize the digital signature schemes is nicely presented by **Stalling** in [35]**.** Without the use of hash, the signature will be of same size as message.

The idea is that instead of generating the signature for the entire message, the sender of the message only signs the digest (hash result) of the message using a signature generation algorithm. The sender then transmits the message and the signature to the intended receiver. The receiver verifies the signature of the sender by computing the digest of the message using the same hash function as used by the sender and comparing it with the output of the signature verification algorithm. It is obvious that this approach saves a lot of computational overhead involved in signing and verifying the messages in the absence of hash functions.

Figure 4 explains how hash functions are used to generate efficient digital signatures. Here, for generating signatures, private key of the sender is used and for verifying the signature, public key of sender is used.

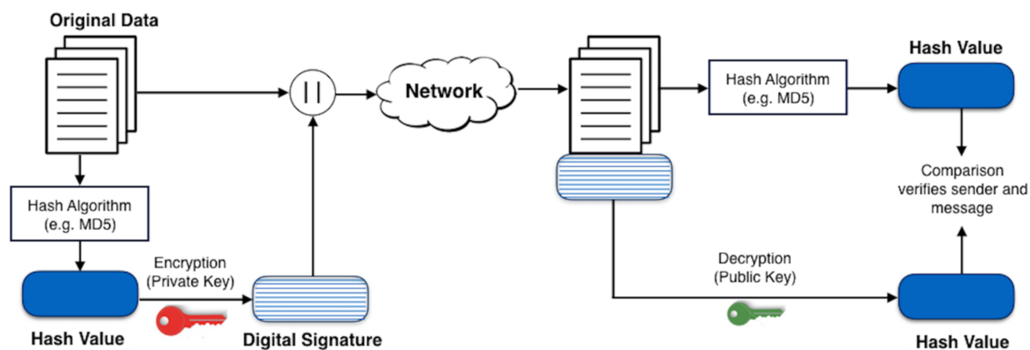


Figure 4. Usage of Hash Function in Implementing Digital Signature (RSA approach)

2.1.3 Authenticating Users of Computer Systems

Hash functions may be used to authenticate the users at the time of login. The passwords are stored in the form of message digest to avoid access of the same even to Database Administrators (because of pre-image resistance of hash digest). Whenever a user tries to login and enter the password, the message digest of the entered password is computed and compared with the digest stored in the database. If it matches, then login is successful, otherwise user is not authenticated. Specialized efficient hash functions for passwords have also been suggested in literature. A **Password Hashing competition** [20] was conducted to select specific password hash functions that can be recognized as recommended standard. **Argon2** [36] has been selected as the winner of this competition.

Hatzivasilis et al. [37] have conducted a survey and presented an analysis of password hash functions submitted in this competition.

2.1.4 Digital Time Stamping

Majority of text, audio, and video documents are available in digital format and a number of simple techniques and tools are available to change digital documents. So, some sort of mechanism is required to certify when such a document was created or last modified. Digital timestamp solves the purpose by providing a temporal authentication.

Rompay [4] in his study has suggested multiple ways - simple scheme based on trusted third party, scheme that links timestamps into temporal chain, and the other one that make use of Merkle Tree. He has highlighted that digital time stamp helps in protecting intellectual property rights, ensuring strong auditing procedures, and implementing true non-repudiation services.

Before Rompay**, Haber and Stornetta** [38] have also detailed how one-way hash functions and digital signatures can be used to implement digital time stamping.

2.1.5 Hash Functions as PRNG

Hash functions can be used to implement PRNG (Pseudo Random Number Generator). A very simple technique can be starting from an initial value (s) known as seed and computing $H(s)$, $H(s+1), H(s+2)$ and so on. **Bellare et al.** in [39] and **Haitner et al.** in [40] have given some other ways of constructing Pseudo random strings from hash functions.

2.1.6 Deriving Session Keys

Hash functions as one-way functions can be used to generate sequence of session keys that are used for the protection of successive communication sessions. Starting from a master key $K0$, the first session key can be $K1 = H(K0)$ and second session key can be $K2 = H(K1)$ and so on. **Matyas et al.** in [41] have described the key management scheme based on control vectors which uses hash functions and encryption functions for generating session keys.

2.1.7 Construction of Block Ciphers

Block ciphers can be used to construct a cryptographic hash function. However, the inverse is also true and there has been block ciphers designed using hash functions.

Handschuh and Naccache in [42] proposed to use the compression function of cryptographic hash function SHA-1 [23] in encryption mode. The name of the cipher was SHACAL. SHACAL-1 (originally named SHACAL) and SHACAL-2 are block ciphers based on SHA-1 [23] and SHA-256 [12] respectively. SHACAL-1 is 160-bit clock cipher and SHACAL-2 is 256-bit block cipher. Both were selected for the second phase of NESSIE project. In 2003 SHACAL-1 was not recommended for NESSIE portfolio because of concerns about its key schedule, while SHACAL-2 was finally selected as one of the 17 NESSIE finalists.

2.1.8 Identifying File or Data

A hash result can be used to reliably identify a file. A number of Source Code Management systems like *Git, Monotone etc*. use *'sha1sum' or 'shasum'* (computer programmes that calculate and verify SHA hashes.) of directory tree, file content etc. to uniquely identify them. Not only in Source Code Management systems, hash functions is also used for identifying files on peer to peer file-sharing networks.

2.1.9 File Verification

D. Armstrong in [43] has nicely explained how hash functions have been used in verifying the integrity or authenticity of a file on computer. Rather than comparing files bit by bit, their message digests (hash values also known as checksums in this case) are stored for future comparisons. If a file got corrupted because of faulty storage media, transmission errors, errors while copying or moving or because of software bugs, then comparison of checksums (message digests of files) with previous calculated checksums can let us know whether the file is corrupted or not. Software utilities like 'md5deep' etc. use the same methodology to verify integrity of files. The same methodology is also used by anti-virus software to cross check whether file has been corrupted by any bot or virus.

2.1.10 Email, IP and Web Security Services and Protocols

Cryptographic hash functions are crucial in implementing E-mail, IP, and Web security. As explained in [35], Email security services like PGP (Pretty Good Privacy) and S/MIME (Secure/Multipurpose Internet mail Extension), Web security services like SSL (Secure Socket Layer), TLS (Transport Layer Security), and IPSec, and X.509 Authentication service, all make use of hash functions to provide secure internet services.

2.2 Iterative Structure of Cryptographic Hash Functions

All Cryptographic hash functions create a fixed-size output (known as message digest / hash value / hash result / hash code) out of a variable-size message input. To process this variable size input, iteration is used. Instead of creating a function with variable-size input, a function with fixed-size input (called compression function) is designed and same is called iteratively until an arbitrary length message is processed completely. **Lai and Massey** [44] named such schemes as **Iterated Hash Structure**.

The commonly used iterative design of cryptographic hash functions are listed below:

2.2.1 Merkle-Damgard Iterated Hash Design (MD Structure)

At Crypto '89, **Damgard** [45] and **Merkle** [14] independently proposed a similar iterative structure to construct a collision resistant hash function using fixed length input collision resistant compression function. Both independently provided proofs in their papers that if there exists a fixed length collision resistant compression function: $f\colon \{0,1\}^n \ X\ \{0,1\}^l \rightarrow \{0,1\}^n$ then by iterating this compression function, one can design a variable length input collision resistant hash function $H\colon \{0,1\}^* \rightarrow \{0,1\}^n$. It means if compression function is vulnerable to some attack, then hash function will also be vulnerable but converse of this is not always true. Originally named "Merkle's Meta Method", this scheme is now mostly called the Merkle-Damgard construction.

Figure 5 represents the structure of hash function based on Merkle-Damgard construction:

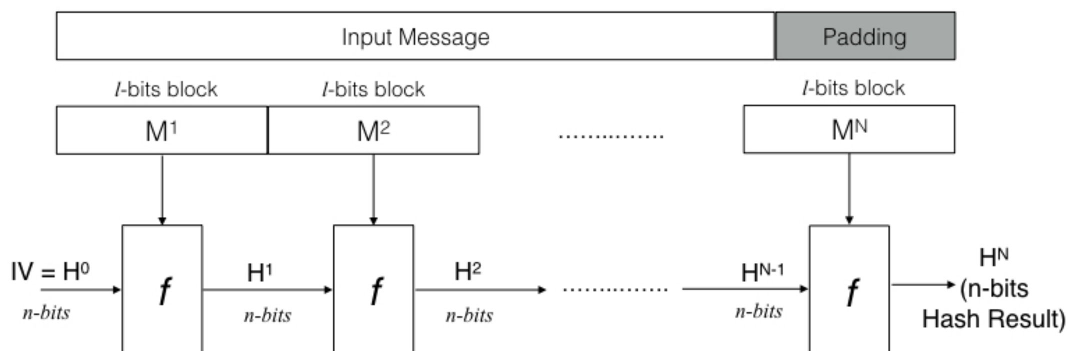


Figure 5. Merkle-Damgard Construction

Merkle–Damgard construction consists of the following steps:

a) Every hash function fixes an n-bit Initial Value (IV) and maximum length of the message that it can process. Let us assume the length of the message is $2^L - 1$.

b) The input message is extended by padding bit 1, followed by enough 0 bits so that extended message is L bits short of one full block. After 0 bits, length of the message in binary form is appended.
c) After this padding, extended message can be evenly divided into block of l bits each i.e. the whole message can be viewed as N blocks of l bits each. First block is named M^1 , second as M^2 and the last one as M^N.
d) Each block M^i is iterated using the compression function computing $H^i = f\left(H^{i-1}, M^i\right), where\ i\ varies\ from\ 1\ to\ N$. This iterative process produces hash value $H(M) = H^N$.

H^i is also known as internal state (chain value). The padding technique as mentioned above may vary from one algorithm to the other. Merkle [14] and Damgard [45] suggested that if IV is not fixed then finding second pre-image or collision is trivial and also if length is not padded then attacks based on fixed points can be used to break iterated hash structure. Both independently provided proof that if IV is fixed as well as length padding is used then hash function will be collision resistant if compression function is collision resistant. The process of fixing IV and adding length padding is known as **MD-strengthening**.

Majority of hash functions launched in recent years and being used these days follow the iterated hash function. MD4 [10], MD5 [11], SHA-1, SHA-256 and SHA-512 [12] are all influenced by the Merkle-Damgard's iterated hash design as explained above. The new hash function designed in this thesis also uses a variation of Merkle-Damgard construction.

Merkle-Damgard construction as explained above has some drawbacks like it suffer from some generic attacks (to be discussed subsequently) like Joux Multicollision [46], Herding attacks [47], Extension attacks [48] etc. To overcome these structural weaknesses, some other constructions have been suggested in the literature.

2.2.2 Wide Pipe Iterated Hash Design

Lucks in [49], proposed an improvement over Merkle-Damgard structure named 'Wide Pipe Hash Design'. The idea was to overcome Length extension and Joux multicollision attacks. Wide pipe design is quite similar to MD design, but it has larger internal state size (chain value). **Lucks** [49] suggested that Joux multicollisions [46] and

length extension are mainly based on internal collisions which can be avoided if we widen the internal pipe from n bits to $w \geq n$ bits. If a hash of m bits is desired, then two functions f and g will be required. Function f is known as compression function and function g is referred as Output transformation. Generally, $w = 2 * n$ is chosen. Figure 6 represents the Wide Pipe Design.

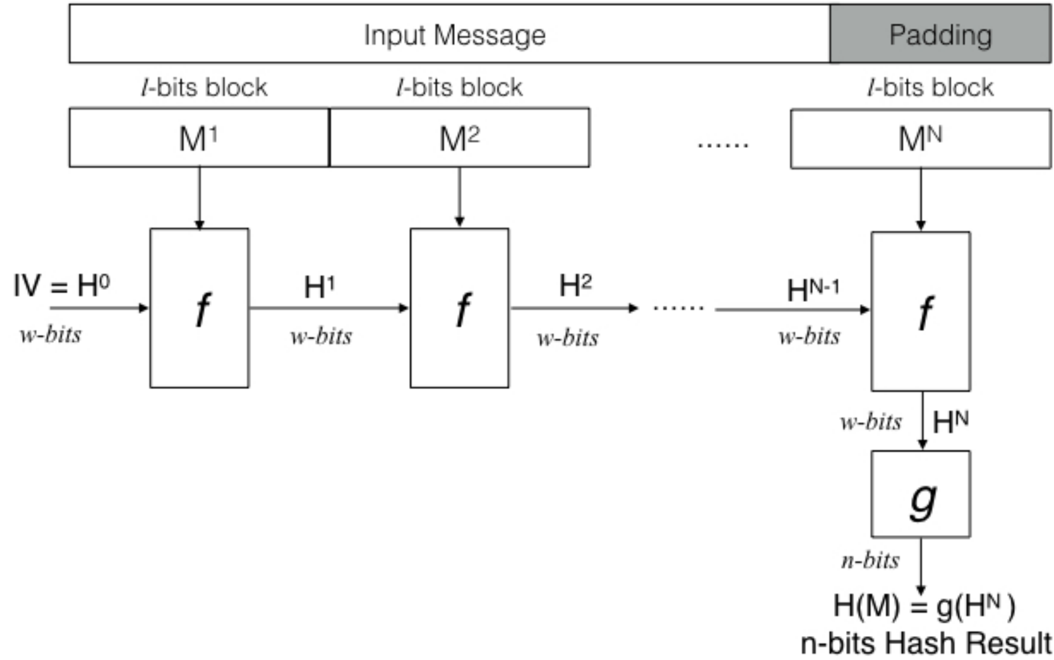


Figure 6. Wide Pipe Design

The compression function, Output transformation, and Wide Pipe Iterated hash design may be defined as follows:

*A **compression function** is a function $f : D \rightarrow R$ where $D = \{0,1\}^w \; X \; \{0,1\}^l$ and $R = \{0,1\}^w$ for some $w, l \geq 1$.*

*An **output transformation** is a function $g: D \rightarrow R$ where $D = \{0,1\}^w$ and $R = \{0,1\}^n$, for some $w, n \geq 1$ and $w \geq n$.*

*Suppose that a compression function $f : \{0,1\}^w X \{0,1\}^l \rightarrow \{0,1\}^w$ and an output transformation $g: \{0,1\}^w \rightarrow \{0,1\}^n$ are given. Then an **iterated hash function** is the hash function $H: \{0,1\}^* \rightarrow \{0,1\}^n$ defined by $H(M) = H(M^0, M^1, \ldots, M^N) = g(H^N)$ where $H^i = f(H^{i-1}, M^i)$ for $1 \leq i \leq N$. The input block $M^i (1 \leq i \leq N) = \{0,1\}^l$ and Initial chaining value $H^0 = IV \in \{0,1\}^w$*

SHA-224 and SHA-384 are based on the same design and are derived from SHA -256 and SHA-512 respectively.

In addition to wide pipe, **Lucks** [49] has also proposed double-pipe hash (twined pipe) design. A variant of twin pipe, suggested by **Gauravram** in [50] and named as 3C and 3C-X Construction, is briefly explained under the head 'Other constructions' in this section only.

2.2.3 Hash Iterated Framework (HAIFA)

Biham and Dunklemnn [51] in 2006 proposed the HAIFA structure to overcome many of the pitfalls observed in Merkle-Damgard Construction. The main idea behind HAIFA are the introduction of number of bits that were hashed so far and a salt value into the compression functions. Formally, instead of using a compression function of the form $f : \{0,1\}^m \text{ X } \{0,1\}^l \rightarrow \{0,1\}^m$, **Biham and Dunkleman** [51] proposed to use form $f : \{0,1\}^m \text{ X } \{0,1\}^l \text{ X } \{0,1\}^{\#bits} \text{ X } \{0,1\}^s \rightarrow \{0,1\}^m$ *i.e.* in HAIFA chaining value H^i is computed as

$$H^i = f\ (H^{i-1}, M^i, \#bits, s)$$

#bits are number of bits hashed so far and *s* is a salt value.

2.2.4 Fast Wide Pipe Design (FWP)

A further improvement of wide pipe design was suggested by **Nandi and Paul** [52] in 2010. They proposed that FWP was nearly twice as fast as the Wide-pipe for a reasonable selection of the input and output size of the compression function. The idea behind FWP was to divide the internal state (i.e. wide pipe chaining value) into two halves. One half is inputted to the compression function but the other half is not fed to the current compression function. The other half is feed-forwarded and XORed with the output of the current compression function and result of XOR act as input to next compression function.

2.2.5 Sponge Construction

Bertoni et al. in [53] [54] [55] proposed sponge construction to design hash functions that closely map the random oracle. In the context of cryptographic hash functions, *sponge functions* provide a particular way to generalize hash functions to more general functions whose output length is arbitrary. **Bertoni et al**. in [54] explained that sponge functions are only distinguishable from random oracles by the detection of inner collisions and the probability of inner collisions can be made arbitrarily small by

increasing a security parameter, called the capacity. As per **Bertoni et al.** [55] the *sponge construction* is a simple iterated construction for building a function F with variable-length input and arbitrary output length based on a fixed-length transformation (or permutation) f operating on a fixed number of bits b, known as $width$.

The sponge construction operates on a state of $b = r + c$ bits; r is called bitrate and c as capacity. Initially all the b bits of state are set to zero and I/P message is padded and divided into block of r bits each. Then sponge construction proceeds in two phases: Absorbing phase and Squeezing Phase.

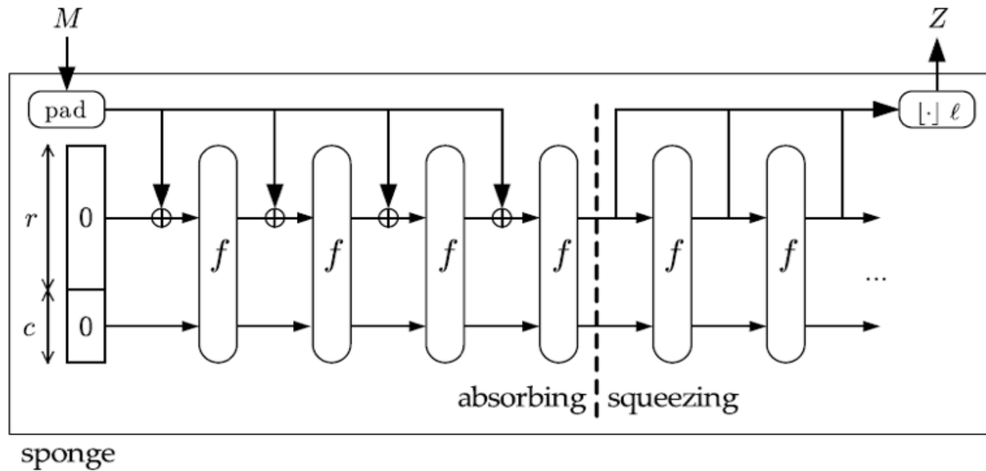


Figure 7. The Sponge Construction for Hash Functions. M is input, Z is hashed output [55]

In first phase input is "absorbed" into the hash state at a given rate, then an output hash is "squeezed" from it at the same rate. To absorb r bits of data, the data is XORed into the leading bits of the state, and the block permutation is applied. To squeeze, the first r bits of the state are produced as output, and the block permutation is applied if additional output is desired.

Central to the Sponge construction is capacity c of hash function and it can be adjusted based on security requirements. SHA-3 [18] winner Keccak [19] is based on Sponge construction and it sets a conservative capacity value i.e. $c = 2n$, where n is the size of the output

2.2.6 Other Constructions

Other important constructions that have been proposed in literature include **Enveloped Merkle-Damgard** (EMD Construction) by **Bellare and Ristenpart** [56], **Randomized**

Hashing (also known as RMX construction) by **Halevei and Krawczyk** [57], and **3C/3C-X construction** by **Gauravaram** [50].

EMD construction [56] preserves collision resistance, pre-image resistance, and pseudo-randomness of the compression function. The message blocks are processed in the same manner as Merkle-Damgard up to pre-final message block. Final message block (M^N) is concatenated with internal state value (H^{N-1}), and is given as input to next iteration of compression function with distinct Initial Value named IV_2. Using the same notation as used in Merkle Damgard construction, EMD construction can be described as:

$$H^0 = IV$$
$$H^i = f\left(H^{i-1}, M^i\right) for\ i = 1,2, \ldots, N-1$$
$$H(M) = H^N = f\ (H^{N-1} \oplus M^N, IV_2)$$

RMX construction [57] is based on the idea of randomization of the message before padding. A random string r of length between the smallest number of padding bits and message block size is generated. Then three other strings r_0 (by appending 0 bits to r), r_1 (by repeating r as many times as required) and r_2 (by extracting some part of r) are generated from r. r_0 is attached to the beginning of the message, r_1 is XORed with all message blocks and r_2 is XORed with padded block. After obtaining Hash result, r is sent (or stored) along with hash result. RMX construction can be represented as:

$$H^1 = f\ (IV, r_0)$$
$$H^i = f\left(H^{i-1}, r_1 \oplus M^{i-1}\right) for\ i = 2,3, \ldots, N$$
$$H^{N+1} = f\ (H^N, r_2 \oplus M^N)$$
$$H^{N+1} = H^{N+1}$$

3C and 3C-X construction [50] are similar to double pipe hash construction. In 3C construction, one line (main pipe) iterates in the same way as MD construction. However, there is an additional line that takes input from main line and iterates similarly. At the end, output of both lines are fed to compression function to generate hash result. Figure 8 represents the twin pipe based 3C construction. For 3C-X construction, the author replaced compression function (except the last compression function) in second line with an XOR operation to make it lighter.

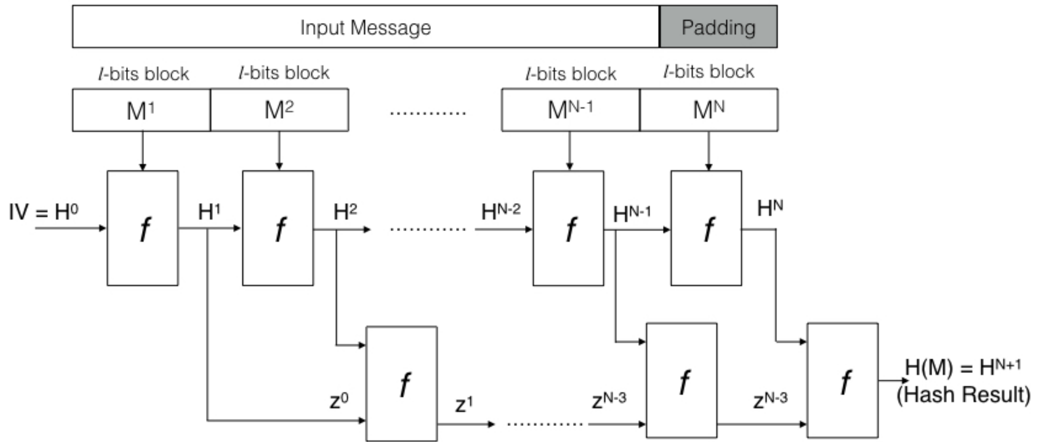


Figure 8. Twin Pipe Based 3C Construction

Andreeva et al. have proposed two related hash structures named **RMC** [58] and **ROX** [59] with an aim to preserve the collision resistance of the compression function along with pre-image resistance, second pre-image resistance, and all their possible variants.

Cascaded constructions have also been discussed in the literature to build large hash values by concatenating several smaller hashes. For example, given two hash functions $H1$ and $H2$, the concatenation $H1(M)||H2(M)$ can be used to generate large hash value for message M. In this construction, $H1$ and $H2$ can either be two completely different hash functions or two slightly different instances of the same hash function. But **Joux** [46] using multi-collisions proved that If $H1$ and $H2$ are good iterated hash functions with no attack better than the generic birthday paradox attack, then the large hash function $H1(M)||H2(M)$ obtained by concatenating $H1$ and $H2$ is not really more secure than either $H1$ or $H2$ taken individually.

Biham and Dunkleman in [51] has explained how multiple constructions like wide pipe design, EMD construction, randomized hashing, RMC and ROX can be instantiated with their proposed structure named HAIFA (Hash Iterated Framework).

2.3 Security Properties of Cryptographic Hash Functions

2.3.1 Basic Security Properties

Basic notion of security of hash functions revolves around ***pre-image resistance, second pre-image resistance*** *and* ***collision resistance*** as defined in Chapter 1 under the heading '1.2 Formal Definition of Cryptographic Hash Functions'. In literature, Collision

resistance property is referred to as collision freeness or strong collision resistance; second pre-image resistance as weak collision resistance; and pre-image resistance as one-wayness [50]**.** It is easy to see that collision resistance implies second pre-image resistance i.e. if a hash function H is collision resistant then H is also second pre-image resistant. However, second pre-image resistance and one-wayness are incomparable (the properties do not follow/imply one another). Hash functions which are one-way but not second pre-image resistant are quite contrived. In practice, collision resistance is the strongest property among all three, hardest to satisfy and easiest to breach, and breaking it is the goal of most attacks on hash functions [44].

Rogaway and Shrimpton [60] extended the notion of hash function security and defined seven different security notions, three on pre-image resistance, three on second pre-image resistance, and one on collision resistance.

2.3.2 Avalanche Criteria and Completeness

From a good hash function, it is desired that the output for two different inputs, should be completely different, regardless of difference in inputs. **Frouzan and Mukhopadhay in** [61] has formalized this with two properties i.e. Completeness and Avalanche effect. **Avalanche effect** (sometime referred as Diffusion property) represents a property when small change in input result in a significant change in message digests. **Completeness** (also known as Full Diffusion) represents a property when each input bit affects all output bits.

Webster and Tavares [62] combined both properties of avalanche effect and completeness and coined the term Strict Avalanche Criterion. It represents a property when a change in one bit of input results in changing every bit of the output (message digest) with a probability of ½. If this **criterion** is not satisfied, then the probability of successful attack on the hash functions increases considerably.

2.3.3 Certification Properties and Weaknesses

In addition to basic properties, some certificational properties have been defined in literature from time to time. For example **Mironov** [63] and **Gauravram** [50] suggested *near collision resistance, partial pre-image resistance, free start collision resistance, pseudo collision resistance, semi Free start collision* as certificational properties for hash functions and / or underlying compression functions. Lack of resistance of these

properties is termed as certificational weaknesses. Certificational properties for hash functions and compression functions on the surface appear desirable but cannot be shown as necessary properties of hash functions. Certificational weaknesses does not result in breaking a hash function directly but is enough to cast doubt on its design principles and may lead to full collision under certain circumstances. Certificational Properties or weaknesses w.r.t. hash function may be defined as a whole or for underlying compression function only. These certificational properties, weaknesses, and possible attacks on these properties are briefly touched upon in this section:

A) Certificational Properties of Hash Functions

Near Collision resistance: As per **Mironov** [63]**,** a hash function is said to be Near Collision resistant if it is hard to find two messages $M \text{ and } M'$ such that the hamming distance between $H(M) \text{and } H(M')$ is small (typically a few bits). Near collision may also be termed as almost collision and can be defined for underlying compression function also. With respect to underlying compression function, almost / near collision means that two message blocks are found for which the difference between the outputs has a low Hamming weight. **Gauravram** [50] quoted the example of how near collisions in case of hash functions with truncated outputs can lead to full collision. If we have a truncated hash function that makes use of leftmost 224 bits of output after chopping rightmost 32 bits then if near collision is found such that message digests differs only in the rightmost 32 bits then such a near collision is practically full collision only.

Partial Pre-image resistance: A hash function is said to be partial pre-image resistant if the difficulty in finding a partial pre-image is same as finding pre-image from a given digest. Also it is hard to find the input if part of the input is known along with digest.

B) Certificational Properties of Compression Function

Certificational properties or weaknesses of the compression functions used in the MD or other similar iterative structures are classified based on the $IV\ /\ H^0$ (Initial value) used. These classifications and nomenclature vary from author to author. For example, *Pseudo collision resistance,* as defined by **Boer and Bosselaers** in [64], is termed as *Special pseudo (type-3) collision resistance* in [50]. Similarly, **Rompay** in [4] named a specific attack as *Random IV collision* and the same attack is named in [50] as *Semi free start collision*. Furthermore, **Mironov** in [63] has defined *Pseudo Collision resistance* and

Free Start collision resistance as two separate properties. On the other side, **Gauravram** in [50] and **Knudsen** [65] have termed *pseudo collision resistance* and *free start collision resistance* as one and the same thing. In this sub section we use the terminology and classification done by Gauravram in [50] as it has been found the most exhaustive and clear but at the same time we also list the alternative nomenclature used by different authors.

Type - 1 collision: Type-I collision resistance is not a certificational property but it is discussed here as it related to other certificational properties based on initial value. Type-I collision refers the collision in a compression function using an IV (initial value) specified in the specification of the hash functions for two distinct messages. Corresponding property may be defined as:

It is hard to find two message blocks $M \text{ and } M'$ *for compression function* $f: \{0,1\}^n \, X \, \{0,1\}^m \rightarrow \{0,1\}^n$ *such that* $f(H, M) = f(H, M')$*, where* H *represents the initial value (IV) specified in the specification of hash function.*

Type-1 collision is also referred to as strong collision.

Type - 2 collision: Type – 2 collision resistance is also termed as Random IV Collision resistance [4] or Semi Free Start collision resistance [50]. Type-2 collisions are the collisions using the same random (arbitrary) initial values for two distinct message inputs. Corresponding property may be defined as:

It is hard to find two message blocks $M \text{ and } M'$ *for compression function* $f: \{0,1\}^n \, X \, \{0,1\}^m \rightarrow \{0,1\}^n$ *such that* $f(H, M) = f(H, M')$*, where computation starts from an arbitrary (random) value H for the input chaining variable.*

Type – 3 collision: Type - 3 collision resistance is also termed as Pseudo collision resistance [50] or Free start collision resistance [65]. Type-3 collisions are the collisions of compression function using two different initial values for two distinct message inputs. Corresponding property may be defined as:

It is hard to find two pairs $(H, M) \text{ and } (H', M')$ *for compression function* $f: \{0,1\}^n \, X \, \{0,1\}^m \rightarrow \{0,1\}^n$ *such that* $f(H, M) = f(H', M')$*, where* $H \,/\, H'$ *represent initial / intermediate chaining value and M/M' represent message blocks.*

Special Type – 3 collision: Special Type – 3 collision are the collisions of the compression function using two different initial values on the same message block. Corresponding property may be defined as:

It is hard to find two pairs (H, M) *and* (H', M) *for compression function* $f: \{0,1\}^n X \{0,1\}^m \to \{0,1\}^n$ *such that* $f(H, M) = f(H', M)$*, where* H / H' *represent initial / intermediate chaining value and M represents message blocks.*

Note that **Rompay** in [4] and **Boer and Bosselaers** in [64] use pseudo collision resistance to represent this property. However, **Gauravram** [50] categorized it as a special category of Pseudo collision resistance and named it as Special pseudo collision resistance.

Inner (almost) collisions: As defined by **Rompay** [4], these are collisions or almost-collisions for the temporary values of the chaining variable (for two distinct message blocks) at some stage of the compression function (for example after $s1$ step operations where $s1 < total\ steps$). This may be helpful for an attacker who tries to generate a collision in the output of the compression function.

The collision attacks on compression functions as described above are also applicable on the iterative modes of their hash functions. Type-1 collision attacks are practical ones and can be used to attack applications that in turn make use of Type-1 susceptible hash functions. Attack demonstrated by **Mikle** in [66] is such an example. Type-2 or Type -3 attacks are not practical but create doubts on the hash functions. Attacks by **Boer and Bosselaers** in [64] and **Dobbertin** in [67] are examples of Type-2 or Type-3 attacks. In [64], Boer and Bosselaers gave an early, although limited, result of finding a 'pseudo collision' (Type- 3) of the MD5 compression function i.e. two different initialization vectors which produce an identical digest. In [67], Dobbertin published an attack (Type-2), without details, that found a collision in MD5 with an IV (Initial value) chosen by him that was different from the one actually used in MD5. While this was not an attack on the full MD5 hash function, it was close enough for cryptographers to recommend switching to a replacement, such as SHA-1. Attacks demonstrated by **Wang et al**. in [68] and [69] are also examples of Type-1 attacks.

2.4 Methods of Attack on Cryptographic Hash Functions

Attacking a hash function means breaking one of its security properties (basic, extended or certificational property). For example, breaking pre-image resistance means that the adversary is able to break the pre-image property. In other words, the adversary is able to create a message that hashes to a specific hash. Breaking certificational

properties may not yield a practical attack but are an important warning to reflect weakness in the hash / compression function. Switching to a strong hash function is recommended when an attack on certificational properties is observed. In an iterated hash function, if a pre-image or collision (Type-1 collision only) can be found for compression function (f), the same can be extended and an attack on hash function can be derived. So attacks may focus on structure of hash function or on algorithm of compression function. In this sub section we will review different types of attacks on hash functions. Depending upon its nature these attacks can be classified into two broad categories - Brute Force Attacks and Cryptanalytical Attack.

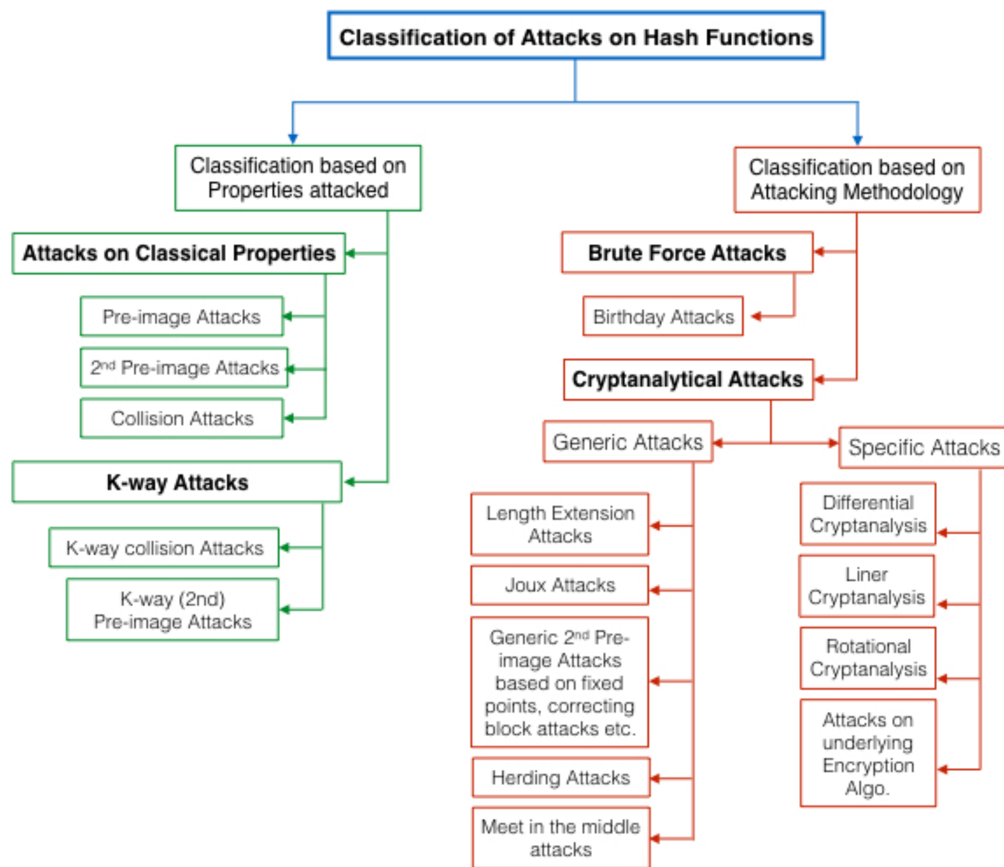


Figure 9. Classification of Attacks on Hash Functions

2.4.1 Brute Force Attacks

Brute force attacks work on all hash functions independent of their structure and any other working details. They are similar to exhaustive search or brute-force key recovery attacks on the encryption schemes to extract the secret key of the encryption scheme. The security of any hash function lies on its output bit size. For a hash code of length n, the level of effort required for brute force classical attacks on hash functions is given below:

Pre-image attack: Effort required for brute force attack is of the order of 2^n(approximately $0.69 * 2^n$). In this attack, for a given n-bit digest H^N of the hash function, the attacker evaluates $H(M)$ with every possible input message M until the attacker obtains the value H^N.

2nd Pre-image attack: Effort required for brute force attack of the order of 2^n(approximately $0.69 * 2^n$+1). In this attack, for a given message M and the hash function $H(\,)$, the attacker tries $H(\,)$ with every possible input message $M' \neq M$ until the attacker obtains the value $H(M)$.

Collision attack: Effort required for brute force attack =$2^{n/2}$. In this attack, for a given hash function $H()$ the attacker tries to find two messages $M\ and\ M'$such that $M \neq M'$and $H(M) = H(M')$. On average the opponent would have to try $\frac{2^n}{2} = 2^{n-1}$ messages to find one that matches the hash code of the intercepted message. However because of Birthday Paradox as illustrated by **Bellare and Kohno** in [70] , the effort required for finding collision in a hash function is of the order of $2^{n/2}$ in place of 2^{n-1} (approximately $1.18 * 2^{n/2}$). Collision attack is also referred as Birthday attack.

In addition to the classical attacks discussed above, the following natural extensions have also been studied by different authors. The nomenclature as given by **Lucks** in [49] is written here:

K-Way collision attack for $K \geq 2$: Find K different messages M^i such that $H(M^1) = \cdots = H(M^K)$.

K-Way (2nd) pre-image attack for $K \geq 1$: Given Y (or M with $H(M) = Y$) , find K different messages M^i, with $H(M^i) = Y\ and\ M^i \neq M.$

2.4.2 Cryptanalytical Attacks

Cryptanalysis of hash functions focuses on the underlying structure of hash function and/or on the algorithm of compression function. Due to fixed size of the hash values compared to much larger size of the messages, collisions must exist in hash functions. However, for the security of the hash function, they must be computationally infeasible to find. Collisions in hash functions are much easier to find than pre-images or 2nd pre-images.

Informally, a hash function is said to be "broken" when a reduced number of evaluations of the hash function compared to the complexities of brute force attack and

the strengths estimated by the designer of the hash function are used to violate at least one of its properties immaterial of the computational feasibility of that effort. For example, assuming that it requires 2^{90} evaluations of the hash function to find a collision for a 256-bit hash function. Though it is impractical to generate this amount of computational power today, the hash function is said to be broken as this factor is less than the 2^{128} evaluations of the hash function required by the Birthday attack. It should be noted that hash functions are easier to attack practically than encryption schemes because the attacker does not need to assume any secrets and the maximum computational effort required to attack the hash function is only upper bounded by the attacker's resources not user's gullibility. This is not the case with block ciphers where the maximum practical count of executions of the block algorithm is limited by how much computational effort the attacker can get the user to do [50].

Collision finding algorithm and attacks may be classified as single block attacks or multi block attacks depending on whether that attack uses single block (i.e. one compression function) or more than one block (i.e. more than one iteration of compression function) for finding collision or pre-images. Cryptanalytical attacks on hash function may be categorized in two types i.e. Generic and Specific attacks.

A) Generic Attacks

The attacks that work on a general hash function construction are called Generic attacks. For example, attacks on the Merkle-Damgard construction that work on all hash functions designed using Merkle Damgard construction are the generic attacks. Generic attacks are applicable even if we replace the underlying compression function by some abstract oracle. Length extension attacks [48] , Joux multicollision attacks [46], Generic 2nd pre-image attacks (like the one based on Fixed points, correcting block attack), Herding Attacks, and Meet in the Middle attacks are example of generic cryptanalysis attacks.

A-I. Length Extension attacks: Length extension, also known as 'message extension' or 'padding' attack, is well known weakness of MD construction. **Biham and Dunkleman** in [51], **Lucks** in [49], and many other authors have highlighted this pitfall of MD construction. Given $h = H(M)$, it is straightforward to compute $M' and\ h'$, such that $h' = H(M||M')$ even for unknown M (but for known length $|M|$). The attack uses $H(M)$ as an internal hash for computing $H(M||M')$. **Kaliski and Robshaw** [71] have

also highlighted how hash of a message and its length may be used to compute hash of longer messages that start with the initial message and include the padding required for the initial message to reach multiple of block size. **Tsudic** [31] also studied length extension attack way back in 1992. However, even these days' certain vulnerabilities based on this simple attack are being observed. **Duong and Rizzo** [72] in 2009 showed a vulnerability in the Flickr (one of the best online photo management and sharing application in the world) signing process making use of Flickr authentication API. This vulnerability allowed an attacker to generate valid signatures without knowing the shared secret. So, using Flickr's API by exploiting vulnerability, an attacker could send valid arbitrary requests on behalf of any application. When combined with other vulnerabilities and attacks, an attacker can gain access to accounts of users who have authorized any third party application.

A-II. Joux Multicollision attacks: Joux in [46] studied the generic multicollision attack on iterated hash functions. Joux showed that finding multicollisions, i.e. r-tuples of messages that all hash to the same value, is not much harder than finding ordinary collisions (i.e. pairs of messages), even for extremely large values of r. More precisely, the ratio of the complexities of the attacks is approximately equal to the logarithm of r i.e. constructing 2^d collisions cost d times as much effort as building ordinary 2 collisions. In this attack, it is assumed that collision finding algorithm (say C) for compression function f exists. A call to algorithm C takes chain value (H^i) as input and generates two message blocks $M \text{ and } N$ as output such that $f\left(H^i, M\right) = f\left(H^i, N\right)$. To start with, the attacker calls this collision finding algorithm for the compression function with the initial state H^0 and algorithm return two messages $M1 \text{ and } N1$ such that $f(H^0, M1) = f(H^0, N1) = H^1$. Then the attacker calls this algorithm with state H^1 and algorithm returns two message block $M2 \text{ and } N2$ such that $f(H^1, M2) = f(H^1, N2) = H^2$. Similarly, successive calls to algorithm can be made. If only two calls are made, then we have obtained $2^2 = 4$ different messages that maps to digest H^2.

$$f(f(H^0, M1), M2) = f(f(H^0, M1), N2) = f(f(H^1, N1), M2) = f(f(H^1, N1), N2) = H^2$$

If we assume collision finding algorithm was based on brute force attack and every call takes time $2^{n/2}$ then it took $O(2 * 2^{\frac{n}{2}})$ time to find 4-collisions. **Joux** [46] demonstrated that this technique requires $O(d * 2^{\frac{n}{2}})$ time for finding 2^d-collisions instead of a brute

force collision finding algorithm that may require $\Omega(2^{n*k})$ time where $k = (2^d - 1)/2^d$. Here n is the size of message digest.[10]

A-III. Multi (2nd) pre-image attacks based on Joux technique: The notion multi (2nd) pre-image represents multiple pre-images as well as multiple 2nd pre-images. The technique presented by Joux [46] can be extended and multi (2nd) pre-images can be found at a cost less than the brute force complexity of finding multiple (2nd) pre-images.

A-IV. Generic 2nd pre-image attacks: In a generic 2nd pre-image attack on a hash function of length n bits, the attacker tries to find second pre-image M' for a target message M such that $M \neq M'$ and $H(M) = H(M')$ with an effort less than 2^n. A number of techniques have been suggested to produce generic 2nd pre-image attacks. Correcting Block attacks as defined in [4] can be used to generate generic 2nd pre-image attacks. **Dean** [73] used Fixed Point attacks to generate generic 2nd pre-images and **Kelsey and Schneier** [74] made use of Joux multi-collisions for generating 2nd pre-image attacks. In this subsection we provide a brief overview of these attacks:

a) **Correcting Block attack:** As stated by **Preneel** in [75]**,** Correcting Block attack may be used for collision attack as well as pre-image attacks in addition to generic 2nd pre-image attacks. Preneel has also referred it by chosen prefix attack. For a pre-image attack, one arbitrary message X is chosen and then adversary tries to find one or more correcting blocks Y such that $H(X||Y)$ comes out to a specific desired value. For a 2nd pre-image attack on target message $X||Y$, adversary freely choose message $X'$and then searches one or more correcting blocks Y' such that $H(X||Y) = H(X'||Y')$. X'may be chosen equal to X also. For a collision attack, two pair of messages $X \text{ and } X'$ are chosen such that $X \neq X'$ and then one or more correcting blocks $Y \text{ and } Y'$are chosen so that $H(X||Y) = H(X'||Y')$.
   To implement the above for 2nd pre-image, opponent uses a pre-existing <message, digest> pair and tries to change one or more message blocks such that the resulting digest remains same. To generate a second pre-image X' for a target message X, the adversary chooses one of the input blocks X_i and replaces it with an alternative block X_i' so that $f(H_i, X_i') = f(H_i, X_i)$. If all other blocks of the alternative

[10] Formally the symbol O and Ω are used for representing expected running time. O asymptotically represent "at most" and Ω asymptotically represent "not less than".

message X' are equal to the corresponding blocks of target message X, then the same hash result will be obtained and a second pre-image can be found.

As per **Rompay** [4], if the size of the internal state i.e. chaining variable is c bits and block size is b bits and $b > c$, then the number of blocks X_i' satisfying the property $f(H_i, X_i') = f(H_i, X_i)$ are approximately $2^b/2^c$ i.e. 2^{b-c}. The challenge is that such blocks are small subset of all possible blocks and to find one such block for an ideal hash function, about 2^c operations are needed. One round of MD5 has been detected for this attack. In MD5, the attacker takes a message block X (consisting of 16 words), fixes the 11 words of X, modifies one word and calculates the remaining 4 words to generate a message block X' which maps to the same digest.

Correcting block attack is possible if the pre-images for compression function can be obtained with the computation starting from pre-specified chaining values. Fixing the value of IV helps in thwarting the attack thus MD strengthening in case of Merkle-Damgard construction prevents this attack from working on complete hash functions [50].

Kelsey and Sheiner [74] have also improved the generic correcting block attack using the notion of expandable messages such that it bypasses the defense provided by MD strengthening.

b) Fixed Point attack: Bard [76] and **Preneel** [75] has given a nice explanation of Fixed Point attack. In this attack, adversary looks for a fixed point in the compression function f. A fixed point is chaining variable H_i such that $f(H_i, X_i) = H_i$. Few authors refer the pair (H_i, X_i) as fixed point. Whenever fixed point exists, the presence of message block X_i does not affect the message digest. To generate pre-images of message X, one may insert arbitrary number of blocks with value X_i to the message X where chaining variable takes the value H_i. Fixed point attack can be avoided by inserting the message length at the end of message. As MD strengthening pad the message length at the end of original message, MD strengthening thwarts fixed point attacks from affecting complete hash functions. However, if fixed points occur at more than one iteration of compression function, then attack may become practical. In such a case the attacker can insert message block X_i at stage i such that $f(H_i, X_i) = H_i$ and can

remove X_i from X at some later stage j, such that $f(H_j, X_j) = H_j$. Even in this case attack is only possible if the initial value is not fixed or the attacker chooses $IV = H_i$, or if fixed points can be found for a significant fraction of all chaining values. **Dean** in [73] presents different techniques that make use of fixed points to produce attack on complete hash functions even in the presence of Merkle-Damgard strengthening. One very simple technique proposed by Dean [73] for MD4 and MD5 hash functions is to repeat the fixed point block 2^{55} times, which adds 2^{64} bits to the input. Since the message length in MD4 and MD5 is computed modulo 2^{64}, this effectively adds 0 to the length field, and the proper hash value comes out.

A-V. Herding attack: Kesley and Kohno in [47] presented a new attack on hash functions based on MD structure, called the Herding attack. In herding attack, an attacker, who can find many collisions on the hash functions by brute force, can first provide the hash of a message, and later "herd" any given starting point of a message to that hash value by the choice of an appropriate suffix. With this attack Kesley and Kohno identified an essential security property for hash functions called Chosen Target Forced Prefix (CTFP) pre-image resistance. CTFP pre-image resistance as defined by Kesley and Kohno in [47] is reproduced here:

*In the first phase of the attack, adversary performs some pre-computation and then outputs an n-bit hash value H: H is his "Chosen Target". The challenger then selects some prefix **P** (picks uniformly at random from large but finite set of strings) and supplies it to adversary; **P** is the "Forced Prefix." In the second phase of attack, adversary computes and outputs some String S. Adversary is said to compromise the CTFP pre-image resistance if it takes less than 2^n evaluations of the hash function to find S such that **hash**(P||S) = H.*

As per **Kesley and Kohno** [47] the following steps are used for applying herding attack:

- In the first phase of a herding attack, the attacker repeatedly applies a collision-finding against a hash function to build a diamond structure[11].
- In the second phase of the attack, the attacker exhaustively searches for a string *S'* such that $P \mid\mid S'$ collides with one of the diamond structure's intermediate states.

[11]Diamond structure is a data structure reminiscent to a binary tree. Diamond structure is a structure of messages constructed to produce large multicollisions. Details are available in [47].

- Having found such a string S', the attacker can construct a sequence of message blocks Q from the diamond structure, and thus build a suffix S = S' || Q such that hash (P||S) = H.

Kesley and Kohno [47] also described the various contexts in which herding attack can be used. Nostradamus attack, stealing credits for inventions, tweaking a signed document, and Random number fixing are examples of such contexts explained in [47].

A-VI. Meet in the Middle attack: This attack is a variation of birthday attack and is applicable to hash function that make use of compression function f invertible to the chaining variable H_i or the message block X_i. As per **Bakhtiari et al.** [77]**,** this attack allows the attacker to construct messages that correspond to certain digest. To apply this attack, the adversary generates $r1$ samples for the first and $r2$ samples for the last part of a bogus message. The adversary then moves forward from the initial value and goes backward from the hash value. The probability that the two intermediate values are same is given by $P \approx 1 - \exp\left(-r1 * \frac{r2}{2^n}\right)$, where n = length of initial value or chaining value or message digest. If the meeting point is found, then the concatenation of the message parts creates a bogus message that results in the target hash value.

B) Specific Attacks

The attacks that work on specific hash function or the algorithm of its compression function are called specific attacks. Attacks illustrated by **Wang et al.** on MD4, MD5, RIPEMD, HAVAL, SHA-0, SHA-1 in [68] [69] [78] [79] [80] and by **Biham et al. in** [81] are examples of collision attacks on the specific hash functions. Attacks using differential cryptanalysis, linear cryptanalysis, rotational cryptanalysis and attack on the underlying encryption algorithms are type of specific cryptanalysis attacks. The most successful of these are the attacks based on differential cryptanalysis.

B-I. Differential cryptanalysis: Differential cryptanalysis, introduced by **Biham and Shamir** [82] was a technique mainly devised to analyse block ciphers. It is used to study the correlation between the difference in input and output. If X and X' are two inputs, then the difference between them is defined as $\Delta X = X \; op \; X'$. If H and H' are two corresponding message digests, then the difference between them is defined as $\Delta H = H \; op \; H'$. The difference operation op can be XOR operation or Integer subtraction or any other operation. For differential cryptanalysis attack, the attacker searches for specific

difference in inputs (ΔX) that result in specific difference in output (ΔH) with high probability. In case of hash function, the difference in output should be zero to result in collisions. Examples of specific attacks using differential cryptanalysis are [68] [69] [78] [79] [80] [81] [83] [84].

B-II. Linear cryptanalysis: Linear cryptanalysis was proposed by **Matsui** in Eurocrypt'93 [85] as a theoretical attack on DES and later successfully used in the practical cryptanalysis of DES. Linear cryptanalysis tries to take advantage of high probability of having linear relationships between plaintext bits, ciphertext bits and sub-key bits. **Heys** has exhaustively explained linear cryptanalysis in [86]. The basic idea is to estimate the operation of a part of an encryption algorithm with a linear equation of the following type:

$$X_{i_1} \oplus X_{i_2} \oplus X_{i_3} \oplus \dots \oplus X_{i_m} \oplus Y_{j_1} \oplus Y_{j_2} \oplus Y_{j_3} \oplus \dots \oplus Y_{j_n} = 0$$

where X_i represents the ith bit of the input and Y_j represents the jth bit of the output. If an encryption algorithm shows a high probability of satisfying or not satisfying the above equation, then the cipher is weak and lacks randomization abilities. Hash functions based on the encryption algorithm can be susceptible to linear cryptanalysis, but till date not many successful attacks on hash functions using linear cryptanalysis has been reported.

B-III. Rotational cryptanalysis: The term Rotational cryptanalysis was coined in February 2010 by **Khovratovich and Nikolic** in [29]. The attack may also be classified as generic attack because, as per the authors, this attack may be applied on all the algorithms that are based on three operations - modular addition, rotation, and XOR (ARX for short). However, we have placed it under the category of specific attacks as this attack has been demonstrated by Khovratovich and Nikolic against reduced round Threefish cipher – part of Skein hash function [9], a SHA3 competition [18] candidate only. Secondly as per classification done in this thesis, the generic attacks are applicable to all the hash functions falling under a particular structure like Merkle-Damgard, so it is better to consider rotational cryptanalysis as a specific attack. In October 2010, a follow up attack that combines rotational cryptanalysis with the rebound attack was presented by the same authors along with **Rechberger** in [28]. In rotational cryptanalysis, the authors studied the propagation of rotational pair $(X, \vec{X})$ through out the cryptographic primitive. The authors presented that operation XOR and rotation both preserve the

rotational pair with probability of 1, while modular addition does it with probability up to 3/8 (and that depending on rotation constant with which $\vec{X}$ is obtained). So with a high probability, a rotational pair of inputs is converted into a rotational pair of output bits.

B-IV. Attacks on underlying encryption algorithm: If the underlying compression function of hash function is implemented using the Encryption algorithm, then the weakness in encryption algorithm can be exploited to attack hash functions. Encryption function may have complementation property or weak keys or may have fixed points and the same may be used to attack complete hash function based on encryption algorithm. **Miyaguchi et al.** in [87] analysed the hash functions from the standpoint of the complementation property and weak keys of the block ciphers used in them and notified their weaknesses.

2.5 Type of Cryptographic Hash Functions Based on Design of Underlying Compression Function

From the discussion in heading '2.2 Iterative Structure of Cryptographic Hash Functions', it is evident that for processing arbitrary length of input the iterative structure of hash function (may be MD structure or any other) is desired and the crucial part of this iterative structure is Compression Function. The classification of hash function can be done, based on the way the compression function is built. This section deals with various ways of building compression function of a cryptographic hash.

2.5.1 Hash Functions Based on Block Cipher as Compression Functions

One of the possible approaches that have been studied by the researchers is to design a compression function from an existing cryptographic primitive like block ciphers. The advantages as highlighted by **Rompay** in [4] is that the existing implementations in hardware or software can be reused. Secondly, some existing block ciphers like DES [88] or AES [5] have received a lot of scrutiny, and thus there is a lot of trust in their security properties.

At the same time a number of drawbacks of block cipher based hash functions have also been observed. **Stalling** in [35] put forward the arguments that the block ciphers do not possess the properties of randomizing functions. For example, they are invertible. This lack of randomness may lead to weakness that may be exploited. Secondly, the

differential cryptanalysis is easier against block operations in hash functions than against block operations used for encryption; because the key is known so several techniques can be applied. The attacks presented by **Preneel et al.** in [89] and **Rijmen and Preneel** in [90] are examples where techniques of differential cryptanalysis are used for attacking hash functions based on block ciphers. Thirdly, it has been suggested that block cipher based on hash functions are significantly slower than hash functions based on compression function specially designed for hash functions. It is also felt that use of a block cipher for a purpose for which it was not designed may reveal some other weaknesses which may not be relevant in case of encryption.

However, with the adoption of AES, there has been renewed interest in developing a secure hash function based on strong bock cipher and these have been exhibiting good performance [35]. Hash functions based on block ciphers can be further classified as follows:

A) Single Block Length Construction

These are the schemes in which size of hash code equals the block size of underlying block cipher. A number of proposals have been made and the basic concept is to construct compression function ***f*** from block cipher. The general scheme works like:

$$H_0 = IV$$

$$H_i = E_A(B) \oplus C$$

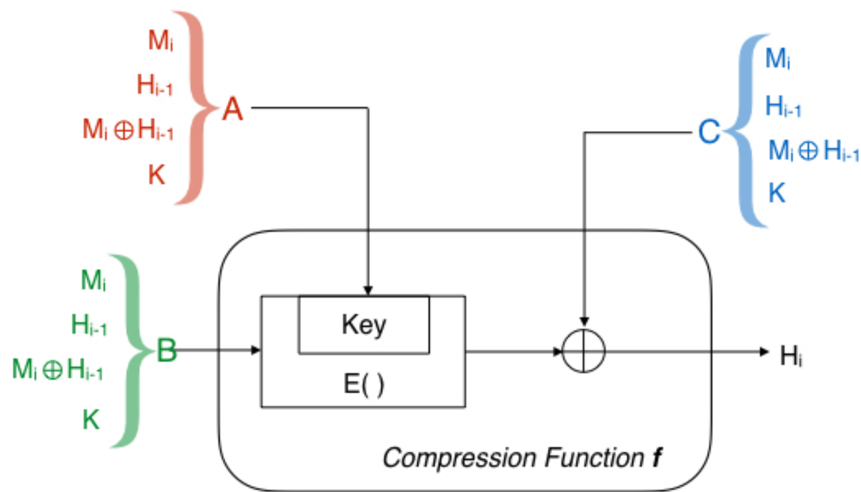


Figure 10. Compression Function Based on Block Cipher

E is the block cipher that encrypts input B with key A. A, B, C can be either M_i, $H_{i-1}, M_i \oplus H_{i-1}$ or a constant K (may be equal to zero or any other value). H_0 is some

random initial value IV. The message is divided into blocks and padding is performed in the same or similar way as done in MD-structure. Figure 10 represents the same.

The three different variables $A, B,\ and\ C$ can take on one of the four possible values. So there are 64 total schemes of this type. **Preneel et al.** [91] studied all these schemes and showed that fifteen of these schemes are trivially weak because the result does not depend on one of the inputs, thirty-seven are insecure for subtle reasons, and remaining 12 of these schemes, depicted in Table 1 , are considered secure. **Schneier** in [1] has also highlighted that first 4 schemes as mentioned in the Table 1, are secure against all attacks and the last 8 are secure against all but a fixed-point attack, which is not really worth worrying about. A formal proof of security of the 12 schemes is given by **Black et al**. [92].

Table 1. 12 Secure Schemes to Design Hash Function from Block Cipher

Secure Schemes based on Block cipher to generate Compression function	**Other Common Name for the scheme as per the Literature**
$H_i = E_{H_{i-1}}(M_i) \oplus M_i$	Matyas-Meyer-Oseas Scheme [93]
$H_i = E_{H_{i-1}}(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$	--
$H_i = E_{H_{i-1}}(M_i) \oplus H_{i-1} \oplus M_i$	Miyaguchi – Preneel Scheme. Independently proposed by Miyaguchi [94] and Preneel [95]
$H_i = E_{H_{i-1}}(M_i \oplus H_{i-1}) \oplus M_i$	--
$H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$	Known as Davies-Meyer Scheme in literature e.g. in [96] [97]
$H_i = E_{M_i}(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$	--
$H_i = E_{M_i}(H_{i-1}) \oplus M_i \oplus H_{i-1}$	--
$H_i = E_{M_i}(M_i \oplus H_{i-1}) \oplus H_{i-1}$	--
$H_i = E_{M_i \oplus H_{i-1}}(M_i) \oplus M_i$	--
$H_i = E_{M_i \oplus H_{i-1}}(H_{i-1}) \oplus H_{i-1}$	--
$H_i = E_{M_i \oplus H_{i-1}}(M_i) \oplus H_{i-1}$	--
$H_i = E_{M_i \oplus H_{i-1}}(H_{i-1}) \oplus M_i$	--

B) Double Block Length Construction

A hash function, generating 64-bit (or 128-bit) digest is insecure as brute force collision will require 2^{32}(or 2^{64}) operations only. Using the single block length

construction schemes as mentioned in previous sub-section, we will get a 64-bit digest with DES as underlying block or 128-bit digest with AES as underlying block cipher. To increase the digest size of hash function and to make it more secure, double length block construction is suggested. These are the schemes in which size of hash code doubles the block size of underlying block cipher. This means DES will result in a 128-bit hash function and AES in a 256-bit hash function. As suggested by **Rompay** [4]**,** the best known schemes in this class are MDC2 (designed by **Brachtl et al.** in [98]) and MDC4 (designed by **Meyer and Schilling** in [99]). MDC-2 is sometime called as Meyer-Schilling scheme. The compression function of MDC2 makes uses of two parallel computations of Matyas-Meyer-Oseas scheme [93]. Explanation of MDC-2 as given in [4] is reproduced here using the terminology used in previous subsection.

Let $C^L \, and \, C^R$ denote the left and right halves of a $b - bit$ value. Then the compression function of MDC-2 can be described by

$H_{i+1}||\overline{H_{i+1}} = f\ (H_i||\overline{H_{i+1}})$, and these values depend on following computations:

$$C_{i+1} = E_{H_i}(M_i) \oplus M_i$$

$$\overline{C_{i+1}} = E_{\overline{H_i}}(M_i) \oplus M_i$$

$$H_{i+1} = C_{i+1}^L || \overline{C_{i+1}^R}$$

$$\overline{H_{i+1}} = \overline{C_{i+1}^L} || C_{i+1}^R$$

If the left and right halves are not switched as done above, then the two chains $(H_{i+1}, \overline{H_{i+1}})$ will be independent and could thus be attacked independently.

The compression function of MDC-4 consists of two sequential executions of MDC-2 compression function. For the second MDC-2 compression, the keys are derived from the outputs (chaining variables) of the first MDC-2 compression, and the plaintext inputs are the outputs (chaining variables) from the opposite sides of the previous MDC-4 compression.

A few other authors like **Lai** in [100], **Lai and Massey** in [44], **Preneel et al.** in [101], **Quisquater and Girault** in [96], and **Hohl et al.** in [102] have also presented different mechanism of constructing hash functions based on block ciphers. **Lai** in [100] **and Lai and Massey** in [44] modified the Davies–Meyer scheme to use IDEA cipher for generating 64-bit hash value. The scheme proposed by **Preneel et al.** in [101] is also called Preneel-Bosselaers-Govaerts-Vandewalle scheme (in the name of authors) and this scheme produces hash value twice the block length of encryption cipher. The scheme

proposed by **Quisquater and Girault in** [96] also generates hash twice the block length. This scheme appeared in 1989 Draft ISO standard [1] but was dropped later. Scheme by **Hohl et al.** [102], known as Parallel Davies Meyer, also produces hash twice the block length.

C) Few More Examples of Hash Functions Based on Block Ciphers

A few other famous hash functions based on block ciphers are listed below:

GOST hash function – This hash function comes from Russia and is specified in the GOST R.34.11-94. It uses the GOST block encryption algorithm. The algorithm is defined in [103].

AR hash function: AR hash function was developed by Algorithmic Research Ltd. and has been distributed by the ISO for information purposes only. Its basic structure is a variant of underlying block cipher (variant of DES) in Cipher Block Chaining mode. The AR hash function is defined in [104]

Whirlpool hash function: Whirlpool is one of the only two hash functions endorsed by NESSIE (New European Scheme for Signatures, Integrity and Encryption). It has also been adopted by International Standard Organization (ISO) and International Electro-Technical Commission (IEC). Unlike virtually all other proposals for a block-cipher based hash function, Whirlpool uses a block cipher that is specifically designed for use in the hash functions and that is unlikely ever to be used as a standalone encryption function. The block cipher used by Whirlpool is substantially modified version of AES [5] and uses Miyaguchi-Preneel scheme [94] [95] for generating hash function. The hash function is explained in [8].

Skein hash function: Skein hash function was one out of the five finalists in the NIST hash function competition held to design SHA-3 standard. The algorithm is based on Threefish tweakable Block Cipher. The algorithm is described in [9] and introduction of the same is given under the heading '3.4 Introduction to SHA-3 Final Round Candidate Algorithms'.

2.5.2 Hash Functions Based on Modular Arithmetic

Compression function can also be designed using modular arithmetic. This allows the reuse of existing implementations of modular arithmetic such as in asymmetric cryptosystems. The idea of cryptosystems based on modular arithmetic is to reduce the

security of a system to the difficulty of solving the problems in number theory. Two important hard problems in number theory which can act as a base for generating cryptosystems are factorization and Discrete logarithm. **Rompay** in [4] has referred to design of two variants of MASH hash functions based on modular arithmetic. The advantage of such hash functions is that the level of security can be easily enhanced by choosing modulus M of appropriate length but hash functions based on modular arithmetic are very slow, even slower than block cipher based hash functions. Also many such constructions have been broken in the past.

2.5.3 Dedicated Hash Functions

Dedicated hash functions are the one which are designed for the explicit purpose of hashing. Compression functions of dedicated hash functions are not based on the existing cryptographic primitives like block ciphers and are not constrained to reuse existing components such as block ciphers or modular arithmetic. This means that they can be designed with optimized performance in mind. A number of such hash functions have been designed. A few of the famous dedicated hash functions and the status of attacks on these hash functions are as follows:

A) MDx Family of Hash Functions

MD2, MD4, and MD5 are three hash functions from MDx family based on Merkle-Damgard structure. Compared to the other two, MD2 is slower and has not obtained much success. However, the dedicated hash functions to have received the most attention in practice are those that are based on MD4 algorithm. Proposed by **Rivest** in 1990 [10], MD4 is a hash function designed specifically towards software implementation on 32-bit platforms. **Rivest** in 1991 came up with MD5 [11], a conservative version, to replace the earlier hash MD4 because of security concerns.

To process variable length input to 128-bit hash, MD5 divides the input message into blocks of 512-bit each (arranged in 16 words of 32-bit each). The padding process is same as defined under the heading '2.2.1 Merkle-Damgard Iterated Hash Design (MD Structure)'. The internal state (chain value), a 128-bit value represented as 4 words of 32-bit each, is denoted by $A, B, C, and\ D$ and initialized to specific constants. The algorithm calls each 512-bit block of input to modify 4 words of internal state. For each message

block of 512-bits, four rounds (each of 16 operations) are used. Four rounds use non-linear function $F, G, H, and\ I$ as defined below:

$F(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z)$ -- Used in Round 1

$G(X, Y, Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$ -- Used in Round 2

$H(X, Y, Z) = X \oplus Y \oplus Z$ -- Used in Round 3

$I(X, Y, Z) = Y \oplus (X \vee \neg Z)$ -- Used in Round 4

Round 1, calls 16 operations one for each word of 512-bit message block. All these 16 operations are quite similar and make use of function F and few constants to update internal state. The same process is carried out in Round 2 to Round 4 using different functions (as described above) for their 16 operations.

As such, MD5 became a milestone in the development of Hash. It was a well-known iterated hash function (generating 128-bit output), widely used in various applications including SSL/TLS, IPsec, and many other cryptographic protocols. It was also commonly-used in implementations of time stamping mechanisms, commitment schemes, random-number generation, and integrity-checking applications for online software. Type-2 (Semi free start collision) and Type-3 (Pseudo collision) attacks on MD5 were reported in [64] [67]. Strong collisions (Type-1 collisions) on MD4 and MD5 have been reported by Wang et al. in [68] [69] [78] and these attacks made the further usage of these hash functions questionable.

B) SHA-1 and SHA-2 Family of Hash Functions

Secure Hash Algorithm (SHA) developed by the National Institute of Standards and Technology (NIST) was also designed on the same principle as MD4 and was published as Federal Information Processing Standard **(FIPS 180)** in **1993** [22]. A revised version, generally referred to as SHA-1, was issued as **FIPS180-1** [23] in **1995**. When the revised version of SHA-1 was published, details of the weaknesses found in SHA-0 (originally SHA) were not provided. SHA-1 produces a hash value of 160 bits. **In 2002**, NIST produced a revised version of the standard known as **FIPS180-2** [6]. Three new versions of SHA with digest lengths of 256, 384 and 512 were defined and these are known as SHA-256, SHA-384, and SHA-512 respectively. Collectively known as SHA-2, these functions include significant changes from its predecessor SHA-1. **In October 2008**, FIPS 180-2 was replaced by **FIPS 180-3** [105] and in this new standard, SHA-224 was added which is similar to other SHA algorithms and produces 224-bit message digest. **In**

March 2012, standard was updated in **FIPS 180-4** [12] by adding SHA-512/224 and SHA-512/256 making SHA-2 family as collection of six hash functions.

SHA-256 and SHA-512 are main hash functions based on 32-bit and 64-bit words respectively and all other hash functions of SHA-2 family are derived from these functions. A brief description of SHA-256 is given here.

To process input of variable length, the input message is divided into blocks of 512-bit each. The padding process is almost similar to MD-structure. The internal state (chain value) is of 256 bits, represented as 8 words of 32-bit each, and is initialized to some specific constants (known as initial value). The algorithm calls each 512-bit block of input to modify 8 words of internal state. Message block of 512-bit is represented as 16 words of 32-bit each. To process a single block, the algorithm generates 64 more words (known as message schedule) from 16 words of the message block. All the message blocks go through 64 steps updating the internal state at each step. After all the 64 steps, internal state is added to the chain value that existed before processing these 64 steps. The whole process makes use of following six logical operations:

$$Ch(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge \neg Z)$$

$$Maj(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

$$\textstyle\sum_0^{512}(X) = ROTR^2(X) \oplus ROTR^{13}(X) \oplus ROTR^{22}(X)$$

$$\textstyle\sum_1^{512}(X) = ROTR^6(X) \oplus ROTR^{11}(X) \oplus ROTR^{25}(X)$$

$$\sigma_0^{512} = ROTR^7(X) \oplus \sigma ROTR^{18}(X) \oplus SHR^3(X)$$

$$\sigma_1^{512} = ROTR^{17}(X) \oplus \sigma ROTR^{19}(X) \oplus SHR^{10}(X)$$

Here $ROTR^n(x)$ represents circular rotation of x towards right-side (i.e. less significant bits) by n bits. $SHR^n(x)$ represents right shift (towards less significant bits) by n bits. Out of this $\sigma_0^{512}, \sigma_1^{512}$ are used for generating message schedule and all the rest are used in 64 steps.

SHA-512 is identical in structure but differs in word size (64-bit in place of 32-bit), block size (1024-bit in place of 512-bit), rotation and shift constants used in logical operations, and number of operations for each block (80 in place of 64). SHA-224 and SHA-384 are truncated versions of SHA-256 and SHA-512 respectively. Both SHA-512/224 and SHA-512/256 are truncated versions of SHA-512. The technique for generating initial values in SHA-512/224 and SHA-512/256 is also a bit different compared to other functions of SHA-2 family as defined in FIPS 180-4.

Attacks on SHA-0 and SHA-1 have been reported in [79] [80] [81]. Till date no practical attack has been reported on SHA-2.

C) RIPEMD Family of Hash Functions

RIPEMD family of hash functions consists of RIPEMD, RIPEMD-128, RIPEMD-160, RIPEMD-256, and RIPEMD-320. RIPEMD, a 128-bit hash function based on MD4 algorithm, was developed in the framework of the EU (European Union) project RIPE (RACE Integrity Primitives Evaluation) by Dobbertin, Bosselaers, and Preneel. Compared to MD4, the order of message words is modified and two instances of algorithms are run in parallel. Both instances are same except the constants. After each block, output of both instances are added to internal state. This arrangement makes this algorithm quite strong against cryptanalysis [1].

RIPEMD-160 [106] was an improved version of RIPEMD. The 128-bit version was intended only as a drop-in replacement for the original RIPEMD, which had been found to have questionable security. The 256 and 320-bit versions diminish chances of accidental collision and don't have higher level of security compared to RIPEMD-160. RIPEMD-160 was designed in the open academic community, compared to NSA designed SHA-1 and SHA-2. However, RIPEMD has been used less frequently than SHA.

A collision on RIPEMD was reported in [68] but that does not affect RIPEMD-160. Till date no practical attack has been observed on RIPEMD-160.

D) HAVAL Hash functions

HAVAL hash function was invented by **Zeng, et al**. in 1992 [107] . To certain extent it takes the motivation from MD4 hash function only. However, HAVAL can produce hashes of different length i.e. 128, 160, 192, 224 or 256 bits. In addition, HAVAL has a parameter that controls the number of passes a message block (of 1024 bits) is processed. A message block can be processed in 3, 4 or 5 passes. By combining output length with pass, authors provided fifteen (15) choices for practical applications to meet the different levels of security requirements. Algorithm was designed for 32-bit computers. Experiments showed that HAVAL is 60% faster than MD5 when 3 passes are required, 15% faster than MD5 when 4 passes are required, and as fast as MD5 when full 5 passes are required. Researchers have uncovered weaknesses which make further use of

HAVAL (at least 128-bit variant with 3 passes) questionable. The strong collision attack on HAVAL was reported by Wang et al. in [68].

All the above dedicated hash functions somehow derive motivation from MD4 algorithm only and are therefore sometime collectively known as MDx type hash functions. Figure 11, as referred in [7], represent the status of MDx family of hash functions. The vertical line shows the year when a hash function was invented and functions that have been attacked are shown crossed with red lines.

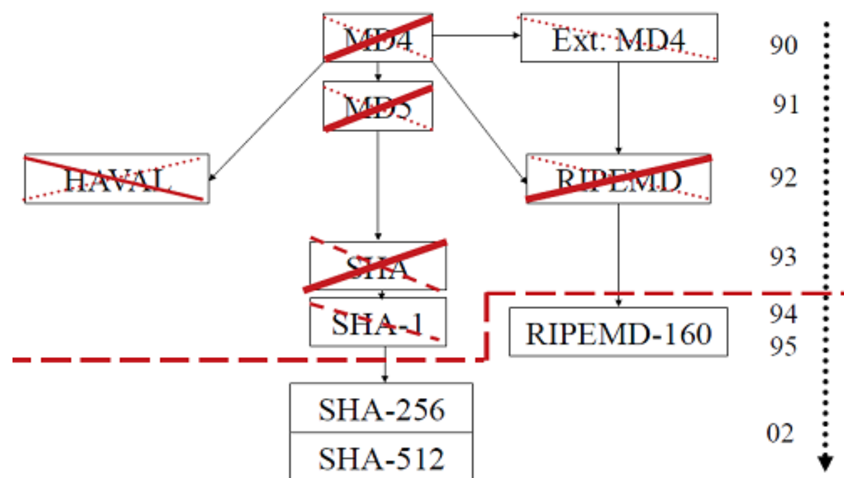


Figure 11. History of MDx-type Hash Functions.

E) More Examples of Dedicated Hash Functions

Some other famous dedicated hash functions reported in literature are **Snefru** [108], **Tiger** [109], **JH** [25]**, Keccak** [19]**,** and **Blake** [26] . **Snefru**; designed by **Merkle** in 1990, like Khufu and Khafre block ciphers was named after an Egyptian Pharaoh. Snefru can generate message digest of 128-bit or 256-bit. Snefru's initial design as well as modified design have been shown to be insecure against differential cryptanalysis [110]. **Tiger** hash function was designed by **Anderson and Biham** in **1995** mainly for 64-bit platforms. It is quite efficient on software but its inherent use of large S-Boxes, makes implementation in hardware or small microcontrollers difficult. Tiger hash function is frequently used in Merkle Hash tree form where it is referred to as Tiger Tree hash (TTH). TTH is used by many clients on Direct Connect and Gnutella file sharing networks. The last three in the list i.e. JH, Keccak and Blake were among the five finalists in the NIST hash function competition [18] to design SHA-3 standard. Structure of these algorithms

with other SHA-3 finalists is introduced separately in subsequent section '3.4 Introduction to SHA-3 Final Round Candidate Algorithms'

2.5.4 Other Approaches to Design Compression Function

There have been few hash functions that are not based on existing cryptographic primitives like block ciphers or modular arithmetic. These are rather based on some hard problems like Knapsack problem, Cellular automata or Discrete Fourier transformations.

Hash function based on knapsack was proposed by **Damgard** in [45] but the same was shown to be broken in [111] and [112]. Cellular automata based hash function was proposed by **Wolfram** in [113] and by **Daemen et al.** in [114]. **Schnorr** proposed hash functions based on discrete Fourier transformations called **FFT-Hash-II** in [115]. This hash function was an improved variant of the algorithm FFT-Hash I presented by the same author in the rump session of CRYPTO'91. **Schnorr** along with **Vaudenay** proposed parallel FFT Hashing in [116]. First two modifications, FFT-Hash I and FFT-Hash II, were broken a few weeks after the proposal [117] [118]. Third modification is quite slow. As a whole, all these approaches (based on knapsack or cellular automata or FFT) did not find much success and are not generally used these days.

2.6 Migration from SHA-2 to SHA-3

The first widely used dedicated hash function was MD4. Developed by Rivest in 1990, MD4 started encountering attacks after which Rivest created a stronger function named MD5 in 1992. However, **Boer and Bosselaers** [64] and **Dobbertin** [67] reported semi free start collision and pseudo collision attack on MD5.

As discussed earlier, Secure Hash Algorithms (SHA-0 and SHA-1) published as FIPS by NIST were also based on MDx family of hash functions. In Crypto' 98, **Chabaud and Joux** [119] reported attacks on SHA-0. In 2002, NIST came up with revised version of the standard (FIPS180-2) adding SHA-256, SHA-384, SHA-512 (collectively known as SHA-2) and SHA-224 was added to these few years later.

This development up to year 2004, has been nicely summarized by **Kelsey** from NIST in [120]. Kesley observes "By year 2004, we as a cryptographic community thought we knew what we were doing". We were well aware that MD4 had been broken, and MD5 known to have weaknesses, as reported by Den Boer, and Bosselaers and Dobbertin, was still widely used. SHA-0 was not used because of weaknesses. SHA-1 was considered to

be very strong. SHA-2 appeared promising and Merkle Damgard was considered a normal way to build hash functions.

The situation changed in years 2004 and 2005, after the practical attack on MDx family followed by attacks on SHA-0 and SHA-1 created doubts about the security of existing hash functions. These attacks have been referred to in the previous section also under the heading '2.5.3 Dedicated Hash Functions' but are discussed here again to clarify the sequence of events that led to SHA-3. Joux [46] showed generic multi collision attack on Merkle Damgard based hashes and reflected that cascaded hashes do not help security much. A number of attacks were reported mainly by **Biham and Chen** [121], **Biham et.al.** [81] and also by a team of researchers led by **Wang** from Shandong University in Jinan, China. The same team also broke HAVAL-128, RIPEMD, and SHA-1 [68] [69] [78] [79] [80].

Looking at the variety of hash functions that had been attacked by this team, it looked as if their approach might explore vulnerability in all cryptographic hashes in the MDx family, including all variants of SHA. **Burr** from US National Institute of Standards and Technology also concurred with this possibility in [122]. Burr pointed out, "With SHA-1 and SHA-2 in its cryptographic toolkit, NIST had hoped to be done with hash functions for a long time. Besides a near break of MD5 by Dobbertin [67] in 1996, researchers made little progress in hash function analysis until mid-2004. Since then, Wang, Joux and Biham have attacked nearly all the early hash functions, including SHA-1. Cryptographers have learned much about hash functions and how to attack them in the past couple of years, and yet cryptanalysts generally agree that practical attacks on the SHA-2 hash functions are unlikely in the next decade. However, attacks and research results could reduce their strength well below theoretical work levels (2^{112}, 2^{128}, 2^{192}, and 2^{256} operations for SHA-224, SHA-256, SHA-384, and SHA-512, respectively)". [122]

In 2006, **Hoch and Shamir** [123] studied the multi collisions on Iterated Concatenated Expanded (ICE) Hash Functions. They extended the idea presented by **Joux** [46] in 2004 which showed that in any iterated hash function it is relatively easy to find exponential sized multicollisions, and thus the concatenation of several hash functions does not increase their security. But Joux attack does not work on ICE i.e. it does not work in the situation when message **E**xpansion is added in addition to **I**terated and **C**oncatenated hash function technique i.e. each iterated function process message block more than once.

Hoch and Shamir [123] considered the general case (ICE) and proved that even if we allow each iterated hash function to scan the input multiple times in an arbitrary expanded order, their concatenation is not stronger than a single function. Finally, Hoch and Shamir extended their result to tree-based hash functions with arbitrary tree structures. They showed that a large class of natural hash functions (ICE and its generalization TCE) is vulnerable to a multicollision attack, and hoped that the techniques they developed would help in creating multicollision attacks against even more complicated types of hash functions. **Such a conclusion was perhaps hinting to probable attack on SHA 2 family of hash functions.**

Pre-image attacks on 41 steps SHA-256 and 46 steps SHA-512 presented by **Sasaki et al.** in [124] reduced the security margin of SHA-256 and SHA-512 also. SHA-256 and SHA-512 having 64 and 80 steps, respectively, then seemed secure but created a doubt for future.

As highlighted earlier under the heading '1.4 Motivation, Research Gap, and Objectives', these attacks called into question all the widely used hash functions. As SHA-2 function belonged to the same family and shared a common heritage and design principles, so these attacks raised many questions on the future of Hash Standards. The cryptographic community was thinking what would happen if SHA-2 was compromised or successfully cryptanalyzed or broken.

Under such circumstances, NIST started public competition to design a new hash standard named SHA-3. The competition was NIST's response to advances in cryptanalysis of hash algorithms. NIST, through this open public competition, was looking to design one or more additional hash algorithms that could be used in place of SHA-2.

As discussed earlier, starting in Nov. 2007 and after few rounds of analysis, NIST selected five SHA-3 finalists; Skein [9], Keccak [19] , Grøstl [24], JH [25], and Blake [26], who advanced to the third (and final) round of the competition on 9th December, 2010. All finalists were tweaked in the final round. BLAKE and JH increased the number of rounds, Grøstl changed the Q permutation, Keccak modified the padding technique, and Skein tweaked the key-schedule constant. Keccak was announced the winner on 2nd October 2012. On 2nd February 2013, Keccak team made a presentation at NIST and NIST's SHA-3 standardization plans were presented at various forums as mentioned in

[120] [125] [126] [127]. NIST announced Draft FIPS 202, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions" [128] on May 28, 2014 and invited public comments on it. However, SHA-3 standard has not been implemented at the time of writing this thesis. NIST policy on hash function [129] suggests no need of transiting applications from SHA-2 to SHA-3.

2.7 About This Thesis

This thesis revolves around five SHA-3 final round candidate algorithms and analyses their performance on platform other than Reference platform as discussed in '1.4 Motivation, Research Gap, and Objectives'.

A serious effort has been made to present a new hash function that outperforms Skein [9] and other SHA-3 final round candidate algorithms on reference and Target platform. The structure and design of new hash functions is based on the accumulated wisdom of the literature that was reviewed.

CHAPTER 3: PERFORMANCE ANALYSIS OF SHA-3 FINAL ROUND CANDIDATE ALGORITHMS

"Performance, and performance alone, dictates the predator in any food chain."

SEAL Team saying

This chapter is focused on the first objective of the research i.e. Performance Analysis of SHA-3 final round candidate algorithms on Target platform (platform other than the Reference platform) as discussed in '1.4.1 Objective 1: Performance Analysis of SHA-3 Final Round Candidate Algorithms'. This chapter is organized into various headings covering:

- Selection of Target Platform and the Rationale behind It **(3.1)**
- Brief about Architecture of Target Platform (**3.2**)
- Various Types of Processors (Processor Series) Available in Target Platform and Finalization of Processors for Analysis of Algorithms **(3.3)**
- Introduction to SHA-3 Final Round Candidate Algorithms **(3.4)**
- Performance Analysis on ARM Cortex-A8 **(3.5)**
- Performance Analysis on ARM Cortex–M4 **(3.6)**
- Performance Analysis on ARM7TDMI **(3.7)**
- Concluding Remarks (**3.8**)

3.1 Selection of Target Platform and the Rationale behind It

The prime decision of looking for a Target platform, on which these final round candidate algorithms were to be analysed, was viewed as a two-step process. In the first step, it was decided to go for an architecture prevalent in Embedded and Mobile platforms. The second step was zeroing down to ARM series processors for their being leaders in Embedded and Mobile platforms. The motivation and rationale behind this is provided here.

3.1.1 Decision – 1: Going for Embedded and Mobile Platform

The narrowing down to the Embedded and Mobile platform was driven by the following two observations:

A) Reference Platform Having Ignored Embedded and Mobile Platforms

The Reference platform, announced by NIST, was a General Purpose system (Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition and ANSI C compiler in the Microsoft Visual Studio 2005 Professional Edition). The fact that had been overlooked was that the 'General Purpose computers' comprise of hardly 1% of total computing devices. The remaining 99% comprises of Embedded and Mobile devices like mobile phones, digital TVs, mass storage controllers, smart cards, smart sensors, automotive body electronics, printers, networking devices like routers etc. The architecture of these Embedded and Mobile systems is considerably different from the General Purpose machines and is generally characterized by small size, limited processing resources, and low power consumption. Importantly, Embedded and Mobile systems, particularly smart phones and netbooks etc., do use hash functions for various security applications. So, logic dictates that evaluation of these hash functions should be carried out on architecture from Embedded and Mobile segment.

B) Recent Surge in Usage of Mobile and Embedded Devices

Another factor that buttressed this decision is the present surge in mobile usage. In recent years the trend of mobile devices has increased considerably and now more and more people prefer to use mobile devices as compared to desktops. Some of the statistics from various sources that vindicate this observation are listed below:

a) ITU (International Telecom Union) collects statistics from 200 economies and over 100 indicators. The latest ICT Facts and Figures 2015 report [130] clearly shows exponential surge in mobile broadband penetration. Globally, mobile broadband penetration reaches 47% in 2015, a value that increased 12 times since 2007, clearly signifying the increasing trend towards mobile usage.
b) Mary Meeker, an analyst from Kleiner Perkins Caufield Byers (KPCB), reviews technology trends and publishes them in and around May every year. She had predicted in 2008 that mobile will overtake the fixed internet access by 2014. The latest "2015 Internet Trend Reports" [131] from KPCB reflecting that we have

crossed that tipping point proves Meeker's prediction. One of the slides from her 2015 report, reproduced below, reflects that time spent with digital media per adult per day in the USA through mobile phones is higher at 52% compared to that through desktops (42%).

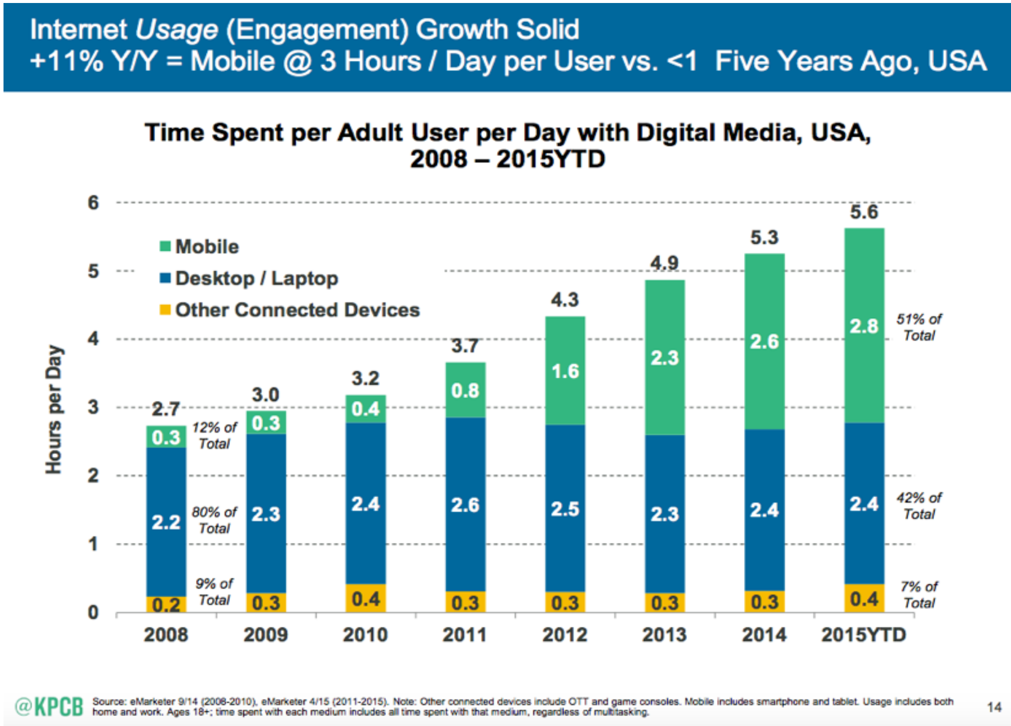


Figure 12. Time Spent with Digital Media in USA [131]

c) Murtagh in [132] clearly presents the fact that Mobile now exceeds PC as far as usage of internet is concerned.

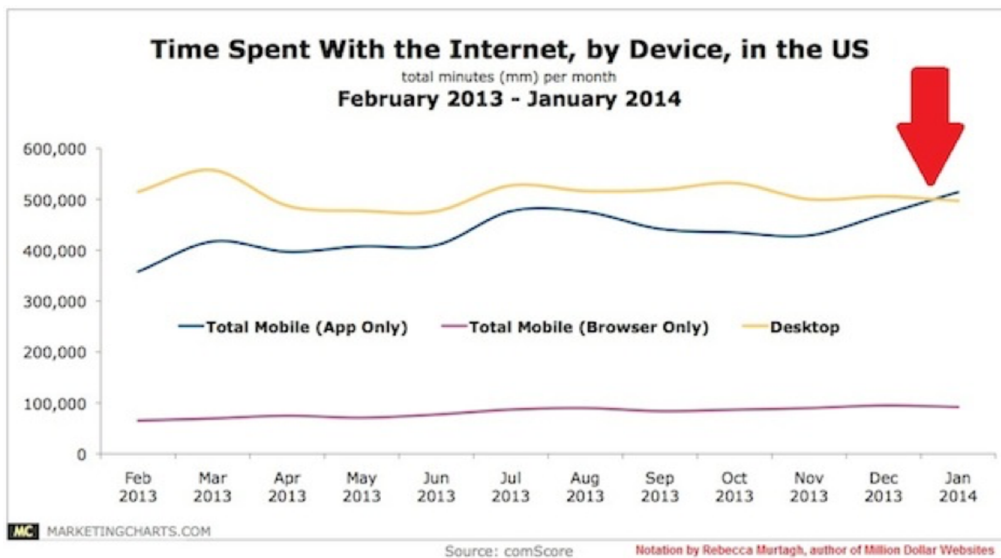


Figure 13. Time Spent with Internet, by Device, in the USA [132]

d) The statistics presented by Bosomworth in [133] and CNET survey results of 2014 presented in [134] [135] , unambiguously reflect the surge in mobile usage and its domination over desktops. Even famous online shopping websites like Flipkart, Myntra etc. have started giving more thrust to their businesses through mobile phones compared to PCs and laptops. India's e-commerce leader Flipkart is taking big steps towards going app-only. For example, its five-day Big Billion-day sale was launched only on its app [136]. Launch of 'Flipkart-Lite' and 'Snap-Lite', new mobile web applications, are targeting audience that access shopping websites through mobile phones [137] [138].

As evident from the foregoing statistics, mobile usage is increasing day by day and mobile platforms are being used to access internet, read emails, do purchases, and consequently all such things do require use of hash functions for security applications. So, opting for architecture prevalent in Mobile platform is quite justifiable.

3.1.2 Decision – 2: Zeroing Down to ARM Architecture

Once the choice narrowed down to Embedded and Mobile platforms, the decision to be made was to select a suitable candidate from among the prevalent architectures in this segment like ARM, MIPS, AVR32, PowerPC etc. The final decision to select ARM architecture was based on its market dominance and performance characteristics that are discussed hereunder.

A) ARM designs cores that are based on RISC (Reduced Instruction Set Computing) architecture. ARM cores have been specifically designed to be small, reduce power consumption, yet have high code density. Both these features make it apt for battery operated devices and also for devices having limited on-board memory like Mobile Phones, PDAs etc. Certain features - like variable instruction cycle for specific instructions (e.g. load-store-multiple instruction), Inline barrel shifter, Thumb 16-bit and Thumb2 instruction set providing high code density, ability of core to switch between ARM and Thumb instruction state, seven different operating modes, and multiple addressing modes including modes that allow direct bit shifting and conditional execution of instructions - make ARM architecture unique and also improve its performance. Multiple extensions developed by ARM like Jazelle, TrustZone, and SIMS/NEON technologies also give ARM an extra edge in this Embedded and Mobile market.

B) The present study found that ARM (Advanced RISC machines) has grown to become world's leading microprocessor IP (Intellectual Property) company, and the ARM processor portfolio covers every area of microprocessor applications from very low-cost embedded microcontrollers to high-performance multicore processors. ARM designs scalable and energy efficient processors and related technologies which are found in many of the digital devices in different market segments including Smartphones, Smart watches, Netbooks, eReaders, PDAs, Digital TVs, Home Gateways, Automotive breaking systems, Storage Controllers, Microcontrollers, Smart sensors, Servers, and Networking etc. The whole ARM market may be divided into the following segments:

- Mobile App Processors & Devices for Home– covering smartphones, tablets, laptops, PDAs, smart watches, consumer entertainments etc.
- Enterprise Infrastructure – covering servers, hard disks, SSDs, networking infrastructures like routers etc.
- Embedded Intelligence – covering automotive app processors, automotive chips, microcontrollers, smartcards etc.

The data of ARM market share in different segments, as obtained from ARM Holdings 2012 Q4 results [139], is shown in Figure 14.

	Devices Shipped (Million of Units)	2012 Devices	Chips/ Device	TAM 2012 Chips	2012 ARM	2012 Share
Mobile	Smart Phone	730	3-5	2,500	2,200	90%
Mobile	Feature Phone	460	2-3	1,200	1,100	95%
Mobile	Low End Voice	730	1-2	730	700	95%
Mobile	Portable Media Players	130	1-3	250	220	90%
Mobile	Mobile Computing* (apps only)	400	1	400	160	40%
Home	Digital Camera	150	1-2	230	180	80%
Home	Digital TV & Set-top-box	420	1-2	640	290	45%
Enterprise	Desktop PCs & Servers (apps)	200	1	200	-	0%
Enterprise	Networking	1,200	1-2	1,300	420	35%
Enterprise	Printers	120	1	120	85	70%
Enterprise	Hard Disk & Solid State Drives	700	1	700	620	90%
Embedded	Automotive	2,600	1	2,600	210	8%
Embedded	Smart Card	6,000	1	6,000	710	13%
Embedded	Microcontrollers	8,700	1	8,700	1,500	18%
Embedded	Others **	2,000	1	2,000	300	15%
	Total	**25,500**		**27,000**	**8,700**	**32%**

Year	Market Share
2007	17%
2008	20%
2009	22%
2010	25%
2011	29%
2012	32%

Source: Gartner, IDC, SIA, and ARM estimates

* Including tablets, netbooks and laptops ** Includes other applications not listed such as headsets, DVD, game consoles, etc

Figure 14. ARM Market Share as per ARM Holdings 2012 Q4 results [139]

ARM core is being used by more than 1000 partners. These partner companies, which make chips based on ARM architecture, include **Apple**, **Samsung**, **Qualcomm, Atmel, Broadcom, Freescale Semiconductor, Nvidia, ST Microelectronics, and Texas Instruments**.

Globally, ARM is one of the most widely used processor architecture. Till date, more than 50 billion ARM processors have been sold and out of which 10 billion were shipped in 2013 alone [140] . As per ARM Holdings 2013 Strategic report [141], ARM technology is now reaching 75% of people in the world and ARM partners are shipping more than 2.5 billion ARM based chips every quarter. Another statistics states that ARM based chips are found in nearly 60 percent of the world's mobile devices and if these chips are laid end-to-end, they would encircle the globe a dozen times [142]. All this vindicates Steve Fuber's statement in [143] that ARM 32-bit architecture is the most widely used architecture in mobile devices and is one of the most popular 32-bit architectures in embedded systems. The latest reports of ARM holdings for year 2015 obtained from [139], also vindicates the fact that ARM chips are being used by leaders of different industry segments. Figure 15 shows the market dominance of ARM in various segments.

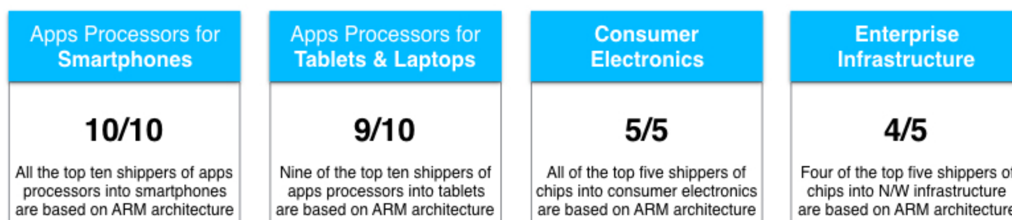


Figure 15. ARM Dominance in Different Market Segments [139]

The way ARM has come to dominate this segment of computing, it becomes quite logical that evaluation of the algorithms under discussion be done on ARM architecture.

3.2 A Brief about ARM Architecture

ARM processor was built in 1985 by Acorn Computer, a British company, with the objective of using it for low cost personal computers (PCs). Later, Acorn introduced an advanced RISC machine and changed ARM from **A**crorn **R**ISC **M**achines to **A**dvanced **R**ISC **M**achines [144].

ARM itself does not manufacture its own electronic chips, but licenses its designs to other semiconductor manufacturers known as partners (as referred to in the previous section). From time to time, ARM upgrades its design and releases these under license to

manufacturers for manufacturing the products. Till now, ARM has released eight such designs from ARMv1, ARMv2 to ARM v7 and ARM v8, which are presently operational.

Since the first ARM1 prototype in 1985, ARM designs have come a long way. Now ARM processor has become a key component of many 32-bit embedded systems. The success of ARM is attributed to its simple and powerful original design based on RISC (**R**educed **I**nstruction **S**et **C**omputing), which is continuously being improved with technical innovations [145]. Pure RISC philosophy is based on five major design rules; a) less complexity at hardware level and have simple instruction that executes in single cycle, b) compiler / programmer handle complex operations by combining simple operations, c) majority of instructions of fixed length resulting in efficient pipelines, d) large general purpose registers used for data as well as addresses, and e) usage of Load-Store instructions for transferring data between register-banks and external memory. However, the difference of ARM instruction set from pure RISC definition in certain ways makes it more suitable for embedded and mobile applications. These are:

3.2.1 Variable Cycle Execution for Certain Instructions

Execution of every ARM instruction does not take place in a single cycle. Some of the instructions like Load – Store may require more cycles when called to transfer multiple registers. Such execution of instructions of variable cycles improves code density as multiple register transfer is common operation at the start and end of function/subroutine call.

3.2.2 Inline Barrel Shifter

Inline Barrel Shifter is a hardware component that can pre-process one of the operands before it is used for operation. It enhances the capability of many instructions and improves the code density and performance.

3.2.3 Unique Register Set and Seven Operating Modes

ARM cores implement the concept of General purpose registers and Banked registers, all of 32-bit size. ARM core has total of 37 registers – 31 general purpose registers and 6 status registers. But all the registers are not accessible all the time. The visibility and operation on register depends on the operating mode of ARM processor. The processor can work in seven different operating modes with maximum of 18 registers accessible in

an operating mode. These processor operating modes are: *abort, fast interrupt, request, interrupt request, supervisor, system, undefined, and user.* For example in the user mode, we can access 17 registers ($r_0 to\ r_{15}, and\ cpsr$). Registers r_{13}, r_{14}, r_{15} are also tagged as $sp - stack\ pointer, lr - link\ register\ and\ pc - program\ counter$.

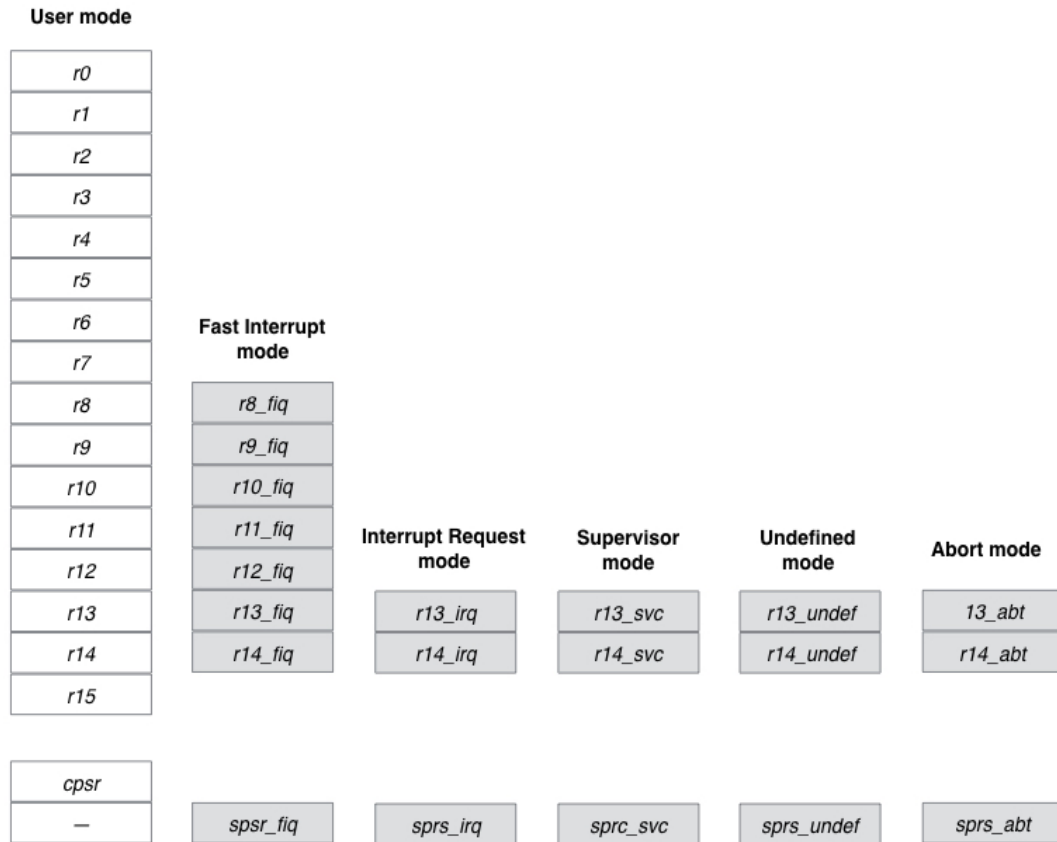


Figure 16. ARM Register Set [145]

All registers, except r_{14}, r_{15} are orthogonal, and $cpsr$ (Current Program Status Register) is used to monitor and control internal operations. Status register has four fields - Flags, Status, Extension, and Control.

Figure 17. Program Status Register Fields

Shaded parts in Figure 17 are not used in several ARM cores and are reserved. Flags are used for implementing conditional execution. Control field has three parts – 'Mode' to select processor operating mode, 'T' to switch to thumb state, and 'interrupt mask'.

Except *user* mode, all other modes are privileged and can read as well as write all fields of $cpsr$. However, in user mode, flag fields can be read as well as written but

control field can only be read. *Supervisor* mode is the mode in which kernel of operating system works. Processor operating mode can be changed by writing in control field of $cpsr$. However, this can not be done in user mode. It can be done in privilege mode only. Change in mode also takes place whenever core responds to interrupts or exceptions.

All modes (except *system* mode) have set of banked registers (coloured grey in Figure 16). Banked registers map one to one onto a user mode register. Whenever the operating mode is changed, the banked registers from the new mode will replace an existing register. For example, whenever these is a change in the operating mode from *user* mode to *interrupt request* mode because of some interrupt, the user registers r_{13} and r_{14} are banked and replaced by r_{13_irq} and r_{14_irq} respectively and $cpsr$ also gets stored in $spsr_irq$. However, $cpsr$ is not stored when mode is changed by writing into control field of $cpsr$.

This study uses the concepts briefed in this subsection to read Cycle Count Register for profiling the SHA-3 final round candidate algorithms on ARM architecture.

3.2.4 Conditional Execution

Most ARM Instructions can be executed only if certain conditions are met (based on setting of Flags). This helps in reducing branch instructions and thus achieve better performance and improve code density.

3.2.5 Thumb, Jazzele, and SIMD Instruction Sets

ARM cores support multiple instruction sets. For example, ARM cores, since ARM7TDMI, have *Thumb* instruction set for improving code density. When the processor executes in Thumb state, it executes a compact 16-bit subset of ARM instruction set. This helps in saving of space by limiting some possibilities compared to full ARM instruction set or by making certain operands implicit. *Thumb2* technology, an extension of Thumb instruction set, has certain additional 32-bit instructions like bit manipulation, conditional execution etc. added to *Thumb* set to give it more breadth. Certain ARM cores also support 8-bit Jazzele instruction set for efficient execution of Java byte codes. Certain Cortex-A Series processors of ARM also support SIMD and Advanced SIMD instruction set (also known as NEON instruction set) which is a combined 64 and 128-bit SIMD instruction set for media and signal processing applications.

The state of ARM core can be switched from ARM to Thumb or to Jazzele instruction set by changing the relevant bits in $cpsr$. This helps in achieving efficiency and high code density.

3.2.6 Pipeline

ARM implements the pipeline execution characteristics of a pure RISC design philosophy by fetching next instruction while instructions in hand are still being decoded and processed for speeding up execution. The pipeline design differs from one family of ARM cores to the other. For example, ARM7 uses three-stage pipeline, ARM9 uses five-stage pipeline, and ARM10 extends it further to six-stage pipeline, though the pipeline executing characteristics remains the same.

3.2.7 Core Extensions

Core Extensions, the hardware components placed next to ARM core, improve performance, manage resources, and provide extra functionality. The Hardware extensions coupled with ARM core will differ from one ARM family to the other, but a few common ones that are wrapped around ARM core in majority of ARM families are: Cache Memory Modules, Memory Management, Coprocessors. Coprocessor CP15 is used in this study to compute cycles consumed by coded algorithms.

More than one coprocessor can also be attached to ARM core. Coprocessors extend the processing features by extending instruction set or sometime by providing configuration registers. To access coprocessor, ARM provides group of dedicated instructions. For example, ARM core uses Coprocessor CP15 to control the cache, memory management etc. Coprocessors can also provide specialized instruction sets like vector floating point operations.

3.3 ARM Processor Portfolio and Finalization of Processor(s) for Analysis

3.3.1 Different ARM Processors

ARM designs a number of processors that are grouped into different families. The ARM processor portfolio [146] consists mainly of:

A) **Cortex–A Series:** Also known as ARM Cortex Application Processors, Cortex-A series processors are high performance processors for devices undertaking complex computational tasks like hosting a feature rich operating system and supporting multiple software applications. These processors are used in high performing consumer, embedded, and enterprise devices like Smartphones, Tablets, Netbooks, Smart TVs, Servers etc. that have memory management system controlled by rich operating systems like Android, iOS or some sort of Linux flavour. ARM Cortex-A8 architecture empowers mobile processors that are used in mobile phones and tablets in today's world. The commonly used processors in this range include Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A17. All these processors are based on ARMv7-A architecture and all support traditional ARM, Thumb, and high performance Thumb2 instruction set.

B) Cortex–R Series: Also known as ARM Cortex Real Time processors, Cortex-R processors are used for deeply embedded and real time markets where high availability, fault tolerance, and deterministic real-time responses are crucial. These processors are commonly used in ASIC, ASSP, and MCU SOC applications in Automotive sector (e.g. airbag, braking, engine management etc.), Storage (hard disk and solid state drive controllers), Enterprise (inkjet and multi functional printers), and Home (Blu-ray players, cameras) etc. The main processors in this series are Cortex-R4, Cortex-R5, Cortex-R7.

C) Cortex-M Series: Also known as ARM Cortex Embedded Processors, Cortex-M processors are targeted towards smart and embedded applications. These processors are generally optimized for Micro Controller Units and mixed signal devices such as IoT, connectivity (router etc.), human interface devices, consumer products, and medical instrumentation. The main processors in this series are Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4, Cortex-M7.

D) Classic Processors: ARM Classic processors refer to the processor families before the launch of Cortex series. These families include ARM11, ARM9 and ARM7 processors. Classic ARM processors have been in the market for more than 15 years and are still widely licensed through out the world, providing cost effective solutions to many of today's applications. Of these processors, ARM7TDMI is still the highest shipped 32-bit processor in the market. As per ARM website, with more than 500 licenses and shipping of over 1 billion devices every quarter, these processors are at the heart of over

1/4th of all electronic products shipped. The main classic processors currently being used are ARM7TDMI, ARM926EJ, ARM946E, ARM968E, ARM1136J(F), ARM1156T2(F), and ARM1176JZ(F).

E) SecurCore Processors: SecurCore processors are mainly used for Smartcards. The three processors in this series are SC300, SC100, and SC000 which are either based on Cortex-M series processors or Classic ARM7 processors.

3.3.2 Finalization of ARM processor(s) for Analysis of Algorithms

From the foregoing discussion on all ARM processor series (processor families), it can be concluded that:

a) Application areas of **Cortex-R processors** do not require usage of hash functions. So **R-series processor family** has been **excluded** from this analysis of five SHA-3 final round candidate algorithms (hash functions).

b) On the other side, **Cortex-A series** processors are **most commonly used** in devices (like Smartphones, Netbooks, Smart TVs, and servers) that run applications requiring use of hash functions for security concerns like achieving integrity and authentication, implementing digital signatures and digital time stamping, generating session keys, maintaining secure web connections, and verifying integrity of files etc. So it was **decided to carry out** analysis of SHA-3 final round candidate algorithms (hash functions) on processor from Cortex-A series.

c) **Cortex-M series** processors are generally not independently used in devices with rich operating systems. However, their usage in human interface devices (like gaming), consumer devices, networking devices like routers **may require usage** of hash functions for achieving one or other security objectives. So it was **decided to carry out** analyses of SHA-3 final round candidate algorithms on processor from Cortex-M series.

d) **Classic Processors:** It was **decided to analyse** the selected candidate algorithms on processor from the classic family also as these processors are **still widely used** in many application areas.

e) **SecurCore Processors** are either based on Cortex-M series or Classic ARM7 series [146]. As processors from both these families have been already included, so no separate evaluations were done on SecurCore Processors.

The matrix in Table 2 represents the final list of processors from different processor series that were finalized for evaluating performance of SHA-3 final round candidate algorithms.

Table 2. List of Processors Selected for Evaluation of SHA-3 Final Round Candidate Algorithms

ARM Processor Series	**Processor and Tools used**
ARM **Application** Processors (Cortex-A series)	**Cortex-A8** processor • Using Open Board AM-3359 from Phytec. • **Running Embedded Linux**
ARM **Embedded** Processor (Cortex-M series)	**Cortex-M4** processor • Using Stellaris LM4F232H5QD Evaluation Board • **Without Any Operating System**
Classical ARM Processor (ARM7 family)	**ARM7TDMI** processor • Using IAR Embedded Workbench **(Simulator)**

3.4 Introduction to SHA-3 Final Round Candidate Algorithms

The five SHA-3 final round candidate algorithms are briefly introduced in this section.

3.4.1 Keccak

The SHA-3 winner Keccak has nothing that looks like MD4 / MD5. It uses Sponge construction (as explained under the heading '2.2.5 Sponge Construction') for building hash function $Keccak - F$ with variable length input and arbitrary output length based on a fixed length permutation $Keccak\ - f$ operating on fixed number of bits b. The building block $Keccak - f$ uses one of the permutation out of the seven permutations named as $Keccak - f[b]$ and width b is defined as $b\ =\ 25 * 2^l\ where\ l\ varies\ from\ 0\ to\ 6$.

As per the above relation, width b can be 25, 50, 100, 200, 400, 800 or 1600. $Keccak - f[b]$, characterized by two parameters: bit rate r and capacity c, holds the relation $b\ =\ r\ +\ c,$ and operates on state a. The inner state a is a three-dimensional array of elements of GF (2) and is written as $a[5][5][2^l]$. For *Keccak – f[1600]*, state a

will be three dimensional array $a[5][5][64]$ and for *Keccak – f [100]*, state a will be three dimensional array $a[5][5][4]$. Initially, all the bits of state are initialized to zero. The input message is padded using multi-rate padding and divided into blocks of r bits each. The sponge construction as explained in "2.2.5 Sponge Construction" then proceeds in two phases: 'Absorbing Phase' in which each block of input message is XORed with r bits of state followed by application of Keccak-*f*[b], and all this is succeeded by 'Squeezing Phase', in which first r bits of the state are returned as output block, interweaved with application of *Keccak-f[b].* The number of blocks are decided by the user depending on the desired hash output size. The last c bits of the state are neither affected by the input messages nor are outputted during the squeezing phase.

Each permutation *'f'* (also termed as *Keccak – f[b]*) as mentioned in Figure 7 is an iterated permutation that makes use of n_r rounds (indexed from $0\ to\ n_r - 1$) and each round R carries out multiple operation in GF(2). For SHA-3, NIST needed four hash sizes i.e. 224, 256, 384, and 512 bits and for these sizes, Keccak recommended the following fixed output length variants:

For output size of 224: b = 1600, r = 1152, c = 448; for output size of 256: b = 1600, r = 1088, c = 512; for output size of 224: b = 1600, r = 832, c = 768; for output size of 224: b = 1600, r = 576, c = 1024. Full details are available in [19].

3.4.2 Skein

Skein is defined for three different internal state sizes: 256-bit, 512-bit, and 1024-bit. As per [9], Skein – 512 is prime proposal, Skein-1024 is ultra-conservative variant, and Skein-256 is low memory variant. Skein can produce variable length hash output as desired by the user. Skein uses tweakable block cipher – Threefish, as the basic building block and UBI (Unique Block Iteration) chaining mode to process arbitrary input size to generate desired output.

Threefish uses three mathematical operations - XOR, Addition, and Rotation (with a constant) - all of which are done on **64-bit words**. The core of Threefish is MIX function which is used in different rounds. Every round of Threefish–512 uses four MIX functions followed by a permutation, named 'Permute' of the eight 64-bit words. A sub-key is inputted every four rounds. Figure 18 illustrates four of the 72 rounds of Threefish-512. The rotation constant of MIX function are chosen that are repeated every eight rounds.

UBI calls multiple instances of Threefish to process variable size input to fixed size output. Skein is detailed in [9].

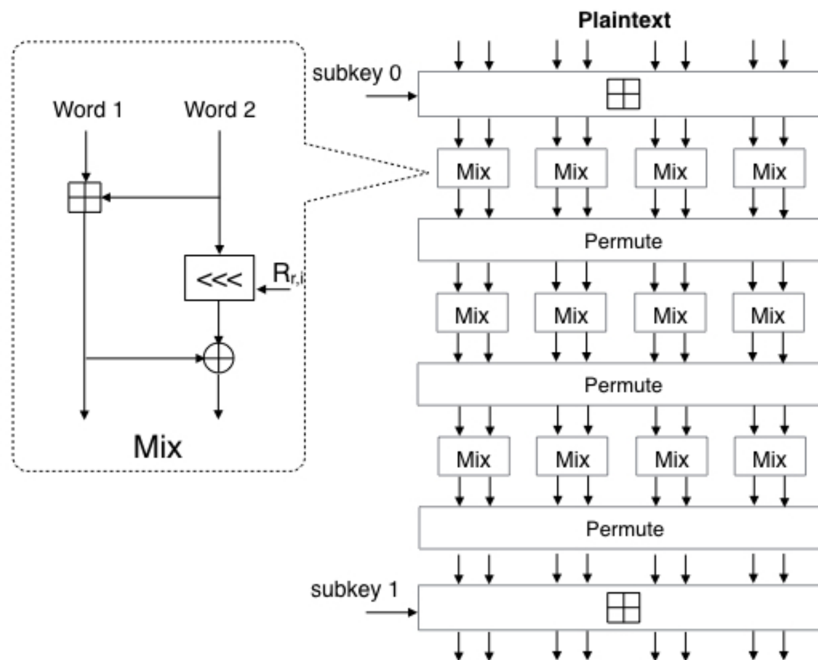


Figure 18. Four of the 72 Rounds of Threefish 512 [9]

3.4.3 Grøstl

Grøstl is an iterated hash function and its compression function is built from two large distinct fixed permutations. It is a byte oriented SP (Substitution and Permutation) network that makes use of S boxes and diffusion layer similar to AES. It is based on the wide pipe design i.e. size of internal state is considerably larger than the hash output size. Grøstl can output message digest of any number of bytes from 1 to 64 i.e. 8 bits to 512 bits in steps of 8 which covers SHA-3 submission requirement i.e. to have message digests of 224, 256, 384, and 512 bits.

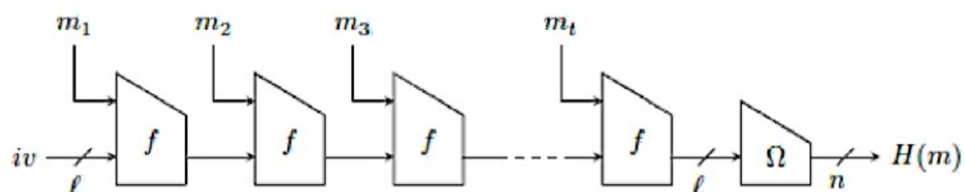


Figure 19. Working of Grøstl [24]

For 256-bit or lesser output, Grøstl uses the state size (and block size) of 512 bits but for higher output sizes, it uses 1024 bits as state (and block) size. The input message is divided into blocks of l bits each (512 or 1024 as stated above) and an initial value H_0 =

IV is defined, and subsequently the message blocks m_i are processed as $H_i = f\ (H_{i-1}, M_i)$ where i varies from 1 to maximum number of blocks. The final output is truncated to the desired width in a final output transformation Ω.

The compression function f is based on two permutation functions $\boldsymbol{P}$ and $\boldsymbol{Q}$ and can be defined as $f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h$. Two types of compression functions have been defined - one for hash size up to 256 bits and the other for hash sizes of more than 256 bits. For 256-bit hash size, the state size is defined as 8x8 matrix whereas for larger hash size, the state matrix has 8 rows and 16 columns. Just like the AES, each round consists of four operations: Add-Round-Constant, Substitute-Bytes, Shift-Bytes, and Mix-Bytes. S boxes are also similar to AES and value of constants vary for P and Q blocks. For 256-bit hash sizes, a total of 10 rounds are operated but for longer hash sizes, 14 rounds are operated. The final output is obtained using $\Omega(x) = trunc_n\ (P(x) \oplus x\ )$ where $trunk_n$ discards all but trailing n bits of x. Details of various round operations are available in [24].

3.4.4 Blake

The Blake hash function is based on HAIFA iteration mode. For 256-bit hash, Blake operates on 32-bit word size and 512-bit block size whereas for 512-bit hash it operates on 64-bit word size and 1024-bit block size. Blake-224 is derived from Blake-256 while Blake-384 is derived from Blake-512 with changed initial values. The compression function uses local wide pipe design making use of salt, counter (number of message bits processed so far) to compress each message block distinctively. As illustrated in Figure 20, the large inner state is initialized from salt, counter, and initial values which is updated with message dependent rounds. At the end, state is compressed to return chain value for the next call of compression function for another message block.

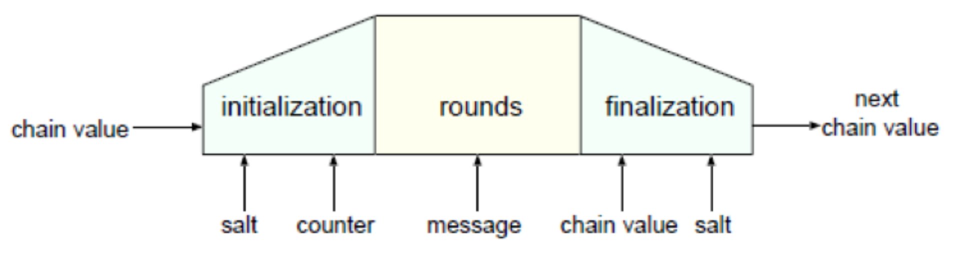


Figure 20. Local Wide Pipe of Blake [26]

The compression function takes Chain value (8 words - h_0 to h_7), Counter (2 words - t_0, t_1), Salt (4 words - $s_0 to$ s_3) and Message block (16 words - m_0 to m_{15}) as input to generate a new chain value of eight words (h'_0 to h'_7) as output. The initial value used in Blake is same as that of SHA-2 but Blake also makes use of 16 constants (c_0 to c_{15}) and ten permutations. The inner state of compression function is of 16 words (32-bit or 64-bit word depending on the hash size) and is arranged in a matrix of 4x4. The compression function of Blake-224/256 iterates a series of 14 rounds whereas that of Blake-384/512 iterates a series of 16 rounds. In each round, all the four columns of inner state are updated independently after which four disjoint diagonals are updated. While updating column or diagonal, two message words are injected according to round dependent permutation. To minimize similarity, each round is parameterized by a distinct constant. After all the rounds of compression function, a new chain value is extracted from state v_0 to v_{15} with an input of initial chain value and salt. Blake is detailed in [26].

3.4.5 JH

JH uses large block cipher with constant key to generate compression function. It utilizes generalized AES design methodology to design a large block cipher from small components [25]. The compression function structure, as given in Figure 21, compresses 512-bit message block $M^{(i)}$ and 1024-bit $H^{(i-1)}$into 1024-bit $H^{(i)}$.

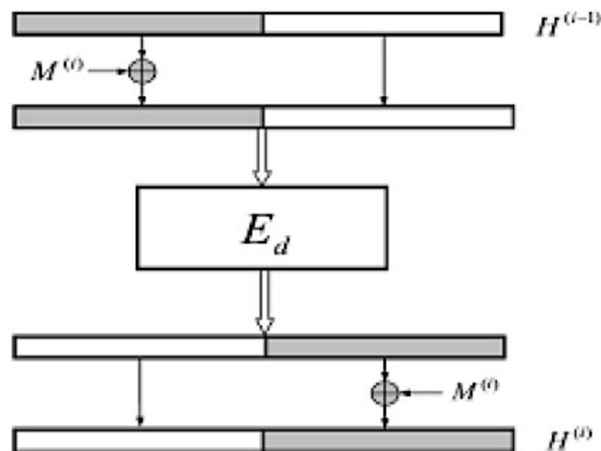


Figure 21. Compression Function of JH [25]

The JH hash function consists of the following steps : **(a)** Pad the message M so that it is in multiples of 512 bits, **(b)** Parse the padded message into N 512-bit blocks named as $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$ and each 512-bit block is expressed as four 128 bit words e.g.

$M_0^{(i)}, M_1^{(i)}, M_2^{(i)}, M_3^{(i)}$ are four 128 bit words of ith block **(c)** Set the initial value H^0 **(d)** Compute $H^{(N)}$by compressing $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$ iteratively as mentioned in Figure 21. **(e)** Generate the message digest by truncating $H^{(N)}$. The last 224, 256, 384, or 512 bits of $H^{(N)}$ are selected as message digest for JH-224, JH-256, JH-384, KH-512 respectively. Operations carried out by E_d and other operations are detailed in [25].

3.5 Performance Analysis of Algorithms on ARM Cortex-A8 Processor

Cortex application series processors, as discussed earlier, are extensively used in environment that require usage of hash function for achieving multiple security goals. This section will present in detail the technique, procedures, tools and methodology used to evaluate SHA-3 final round candidate algorithms on ARM Cortex-A8 processor. The work discussed in this section has been published in [147].

3.5.1 Hardware and Software Tools Used

A) Cortex – A8 Core

From among the Cortex Application series, ARM Cortex-A8 processor was picked for this study. ARM Cortex-A8, a high-performance single core 32-bit processor, introduced by ARM Holdings in 2005, was the first processor supporting the ARMv7-A architecture. Cortex-A8 was also the first Cortex design to be adopted for mass consumption [148]. It has been widely shipped in a range of mobile consumer devices. A few of the Systems on Chips (SOC) that have implemented Cortex-A8 core includes Apple A4, Samsung Exynox 3110, Texas Instruments (TI) Sitara ARM Processors, TI OMAP3, Rock chip RK2918, Freescale Semiconductor i.MX51. Key features of Cortex-A8 [149] are:

- Frequency up to 1GHZ giving 2 Dhrystone MIPS / MHZ.
- Supports Thumb, Thumb-2, NEON(64-bit) and VFPv3 Floating point ISA
- Optimized and integrated L1 and L2 cache integrated into the processor to ensure power efficiency and optimal performance. L1 cache can give single cycle access time.
- Implements Dynamic Branch predictor with >95% accuracy and houses a full MMU that enables Cortex-A8 to run rich OS in variety of applications.
- Support integer pipeline depth of 13-stage.

B) Target Machine: OpenBoard-AM335x Kit with phyCORE-AM335x SOM

For evaluation, OpenBoard-AM335x kit from PHYTEC [150] was used. This Development kit houses phyCORE-AM335x System on Module (SOM) that features Texas Instruments' Sitara^{TM} ARM Cortex^{TM} – A8 CPU. Texas Instruments' AM335x family of processors deliver up to 720 MHz performance. The OpenBoard-AM335x also includes graphics chip, communication subsystem, 512 MB DDR3 RAM, 512 MB NAND, 10/100/1000 Ethernet with dual-port switch and UART (RS232) interface. Image of the board is given in Figure 22.

Figure 22. Cortex-A8 Based OpenBoard-AM3359 from PHYTEC [151]

The OpenBoard has boot loader, kernel, and file system. It runs embedded Linux kernel 3.2.0. PHYTEC also provided board support packages including GCC C/C++ cross development tool chains and bash shell script to prepare the host machine for development [151].

C) Host Machine Running Linux

This study needed a host system that could be used to code the algorithms, cross compile the same for target machine (OpenBoard-AM335x), access the target machine thorough 'Minicom' (explained in next section), and transfer files to the target machine through SCP (Secure Copy protocol) or such other tools.

For this, Dell Inspiron 3542 Laptop housing 1.7 GHz Intel Core i5 4210U processor with 4 GB DDR3 RAM was used as host machine. This machine runs 32-bit Linux Ubuntu 12.04 LTS version.

D) Host and Target Machine Setup

The host machine (Dell Inspiron laptop running Linux Ubuntu) and the target machine shared **two** communication links one through Ethernet ports and the other through USB and Serial port (RS232) (using MCP2200 USB to Serial converter microchip).

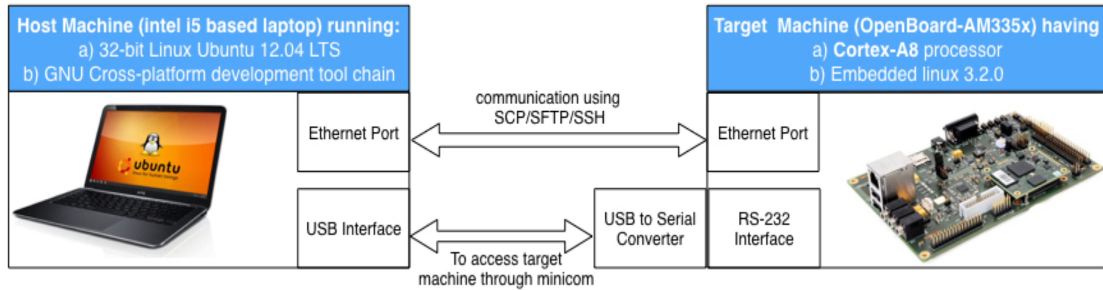


Figure 23. Host and Target Machine Setup for OpenBoard-AM335x

E) Minicom Software and Its Initial Setting

'Minicom' is a utility software used as text-based terminal emulation programme in Linux and other such operating systems. It is commonly used for setting up remote serial console (based on RS-232 interface). Minicom was used to emulate OpenBoard-AM335x from host machine through serial port. Minicom provides command line interface to access Linux target board on host machine. Using the terminal emulation provided by Minicom, one can configure Linux on target board (like installing boot loaders, flashing kernel) and run SCP or TFTP to transfer files from host to board and vice versa. As host machine did not have serial port so USB to serial converter (MCP2200) was used for connecting host and target machine (board). The instructions to setup and use Minicom are given in Appendix -1.

F) GNU Cross-Platform Development Tool-chain

The host machine, on which the algorithms were coded, is an x86 machine running Linux Ubuntu and the target machine on which the algorithms were to be evaluated is ARM Cortex-A8 based machine running embedded Linux. So, a cross compiler was required that could create executable code for platform other than the one on which the compiler itself is running. In fact, to generate binaries of coded algorithms that can execute on target Cortex-A8 board, not only cross compiler, but full cross-platform tool-chain is required. Tool-chain includes linkers, assembler, C (or other language) compiler, library, and headers.

Configuring and building an appropriate cross-platform GNU development tool-chain is a complex and delicate operations. Yaghmour has given detailed steps in chapter 4 of [152] to configure and build tool chains. For the present study, tool chain provided by PHYTEC has been used.

G) SCP Program and Interface Configuration Utility

For transferring files between the host and target machines, we used SCP program that implements SCP (Secure Copy Protocol) as a service daemon. SCP uses Secure Shell (SSH) for data transfers. Command line ***scp*** is provided in most of the SSH implementations. The syntax of ***scp*** is similar to ***cp*** command:

Copying from host:

scp source_file user@host:directory/targetfile

Copying to host:

scp user@host:directory/sourcefile target_file

To use *scp* program, both the host and target machine of the setup need to be connected to Ethernet cable (as shown in Figure 23) and identified by unique IP address. To assign unique addresses to both the host and target machine, interface configuration (***ifconfig***) utility was used. ***Ifconfig*** is quite a versatile utility and is used to configure, manage, and query network. The command used to assign address to the host and target machines was:

ifconfig eth0 192.168.1.12 up

The same command was used on both the host and target side but with different IP addresses. For the setup, IP 192.168.1.12 was assigned to the host machine and 192.168.1.11 to the target machine.

H) Additional Development Tools by PHYTEC

The Ubuntu does not come with all the pre-requisite archives for development. So to prepare the host for development, the bash shell script provided as Board support package with OpenBoard-AM335x [151] was also downloaded. The commands used were:

wget ftp://ftp.phytec.de/ Products/India/OpenBoardAM335x/.../elinux_pkg.sh

chmod +x elinux_pkg.sh and then run *sh elinux_pkg.sh*

3.5.2 Methodology Used

A) CPB as Performance Metric

For evaluating the performance of various algorithms on Cortex A8 processor, the performance parameter used was ‘Cycles per Byte’ (CPB) where CPB is the number of cycles consumed by the hash function divided by the number of input bytes.

For performance metric, execution speed in units of time was the other option. For this study, CPB was preferred as performance metric because contrary to execution speed, CPB is architecture oriented, and does not change with the frequency of the device used. Thus CPB is a better measure than execution speed.

Secondly, for computing CPB, the cycles after computation were divided by total bytes with padding. For example, if data of 100 bytes (800 bits) is inputted to Grøstl-512 algorithm, then initialization functions will do padding to generate at least one block of 1024 bits. So, the cycles consumed were divided by 1024 (bits after padding) rather than 800 bits (input data without padding).

B) Hash Function as a Whole rather than Compression Phase Alone

The five candidate algorithms, just like other hash functions, have three stages i.e. Initialization, Compression, and Finalization. **Initialization** involves padding the input data, setting the initial value to internal state and parameters, and generation of lookup tables etc. The **compression phase** is the main part that consists of multiple calls to compression function depending on the number of input blocks. In every call to compression function, the internal hash state is updated using the current state and one message block. The amount of time spent in this phase is directly proportional to the message length and accounts for the largest part of overall execution time or cycles consumed. The **Finalization phase** usually involves the processing of final block (including padding block) and a final call to output transformation that gives the resulting value. Many researchers often measure the compression phase only for benchmarking purpose as it is a major part of the total execution time. Undoubtedly, initialization and finalization phases take relatively lesser time than the compression phase. However, these phases also vary considerably with the algorithm concerned. So, in this study for performance evaluation, the clock cycles consumed by all the three phases were measured rather than only the compression phase.

C) Optimized 32-bit Implementation of Hash Algorithms Used

As desired for SHA-3 submission, the five finalist algorithms can produce hash output of 224, 256, 384, and 512 bits. The federal notice for SHA-3 competition [27] had asked for reference implementation as well as two optimized implementations – one optimized for 32-bit platform and another for a 64-bit platform. As the Target platform (Cortex A8)

selected for this study is a 32-bit platform, so a 32-bit optimized submission was used and all five SHA-3 finalists were implemented for 224, 256, 384 and 512-bit hash output.

The algorithms were coded with minimum optimization for Target platform (i.e. ARM) as the purpose was to analyse performance of the algorithms with minimum optimization on Target platform to get realistic results rather than optimized results which may vary according to the level of optimization of the code. Also all coded hash functions were compiled without any compiler's optimization level using GNU cross-platform development chain provided along with OpenBoard335x. The syntax of the same is:

```
arm-cortexa8-linux-gnueabi-gcc    algorithm_code.c    -o    algo_binary
```

D) Computation of Cycles Using System Control Coprocessor

To measure the cycle consumed, System Control Coprocessor CP15 available with Cortex A8 was used. The CP15 controls and provides status information like cycle counts. Out of various system control registers of CP15, 'c9' register was used. The detailed methodology and approach of using CP15's 'c9' for accessing cycle count is detailed in section '3.5.3 How Coprocessor CP15 is Used to Access Cycle Count Register (CCNT)'. Using this methodology, the function for reading the Cycle Count register was written and this function was called before and after the call to $Hash$ () function. The difference in the two readings gave us the cycles consumed by the hash algorithm.

E) Averaging the Cycle Count and Subtracting the Overhead

The two calls to a function, one before a function *Hash* () and another after function *Hash* (), that reads Cycle Count register will measure the exact cycles consumed by the function *Hash* () as well as the cycles spent in other processes or in the kernel. There is no way to restrict the measurement to a single thread. To reduce this effect, the cycle consumption was measured multiple times and then the average was calculated to record the readings. Also the above function, used to read Cycle Count Register, is not free. It also has some overhead. This overhead however is fixed and it was computed multiple times on an idle system and averaged out to find the exact overhead.

3.5.3 How Coprocessor CP15 is Used to Access Cycle Count Register (CCNT)

For counting cycles consumed by a set of instructions, System Control Coprocessor CP15, available in Cortex-A8, was used. The purpose of CP15 was to control and provide

status information for the functions implemented in the processor. The main functions of the System Control Coprocessor CP15 as detailed in [149] are:

- Overall system control and configuration
- Cache configuration and management
- Memory Management Unit (MMU) configuration and management
- Preloading engine for L2 cache
- System performance monitoring.

And out of the above, the main concern of this study was **System performance monitoring function**. **Performance monitoring function** keeps track of system events such as cycle counts, cache misses, TLB misses, pipeline stalls, and other related features for enabling system developers to profile the performance of their systems. Before proceeding further, a brief about coprocessor instructions is being given as the same will be used to access various registers of CP15.

Syntax of Coprocessor Instructions: For reading and writing from Coprocessor register, MRC and MCR instructions are used. MRC is used to read coprocessor registers and MCR instructions is used to write coprocessor registers.

The template of instruction is:

$$< MRC|MCR > \ \{< cond >\}\, cp, opcode1, Rd, Cn, Cm\, \{, opcode2\}$$

- cp represent the coprocessor number. In this case it will be $p15$ $(CP15)$.
- $opcode$ fields represent operation to take place on the coprocessor.
- The Cn, Cm describe registers within coprocessor. Cn is primary register, Cm is the secondary register, and $opcode2$ is a secondary register modifier. Secondary register is also known as ‘extended or operation register’.

With respect to $CP15$, the example of MRC and MCR instructions is given below:

$$MRC\ p15, 0, < Rd >, c9, c12, 0\ ;\ Read\ PMNC\ Register$$

$$MCR\ p15, 0, < Rd >, c9, c12, 0\ ;\ Write\ PMNC\ Register$$

The first instruction in the above example reads Performance Monitor Control Register (addressed by $c9, c12$ $coprocessor\ registers$) and copies the contents to Rd register of ARM core (Rd may be any register from r_0 to r_{13} as detailed in ‘3.2.3 Unique Register

Set and Seven Operating Modes'). Similarly, the second instruction writes content to Performance Monitor Control Register.

CP15 Coprocessor Registers used for Cycle Count: CP15 has multiple System Control Coprocessor registers and of these the most important register for this research is $c9$, which has 16 operation registers from $c0$ to $c15$. The register allocation details of $c9$ (extracted from [149]) is given below in Table 3.

Table 3. Details of c9 Register of CP15 Coprocessor

Sr. No.	Primary Register	Opcode1	Operation Register	Opcode2	Detail (Register or operation)
	c9	0	c0 to c11	0 to 7	undefined
A	**c9**	**0**	**c12**	**0**	**Performance Monitor Control register**
B				**1**	**Count Enable Set Register**
				2	Count Enable Clear
C				**3**	**Overflow Flag Status register**
				4,5	Multiple purposes not related to this study
				6, 7	Undefined
D	**c9**	**0**	**c13**	**0**	**Cycle Count register**
				1,2	Multiple purposes not related to this study
				3 to 7	Undefined
E	**c9**	**0**	**c14**	**0**	**User enable register**
				1	Interrupt enable set
F				**2**	**Interrupt enable clear register**
				3 to 7	Undefined
	c9	0	c15	1 to 7	Undefined

The items in bold font (and serial numbered from A to F) are important for accessing cycle count from programme running in user mode. These primary and secondary (operation) registers, important for accessing cycle count, are discussed below.

A) $c9, c12$ ($opcode2 = 0$)-Performance Monitor Control Register:

The purpose of the Performance Monitor Control register (PMNC) is to control the operation of **a)** the four Performance Monitor Count Registers, and **b)** the Cycle Counter Register (CCNT).

PMNC is read/write register and is accessible as determined by "$c9, c14\ (opcode2 = 0)$- User enable Register" (Sr. No. E in Table 3). The PMNC Register is always accessible in privileged modes.

31–24	23–16	15–11	10–6	5	4	3	2	1	0
IMP	IDCODE	N	Reserved	DP	X	D	C	P	E

Figure 24. Format of Performance Monitor Control Register [149]

The details of various bits shown in Figure 24 are available in [149]. The important ones that were accessed for measuring cycle count are:

E (Enable Bit) : Set to 1 to enable all counters including CCNT.

P (Performance Counter Reset) : Set to 1 to reset all performance counters to zero.

C (Cycle Counter Reset) : Set to 1 to reset CCNT to zero

D (Cycle Count Divider) : Kept it to 0 as we wanted cycle counter to be incremented after every cycle rather than after 64 processor cycles.

The code used to implement the setting of PMNC bits is given below.

```
static inline void initialize (int32_t reset, int32_t  divider) {
        int32_t value = 1;  // This will initialize bit ‘E’ of PMNC and thus enable all counters
        // For resetting the counters
        if (reset)
        {
                value  |= 2;     // Set Bit ‘P’ as well ‘E’. So enable and reset all counters to 0
                value  |= 4;     // Set Bit ‘C’ thus also reset cycle counter to zero.
        }
        if (divider)
                value  |= 8;     // enable "by 64" divider for CCNT. Not done in our case.
        value  |= 16;  // Enable export of events from the event bus to monitoring block
        // program the PMNC register:
        asm volatile ("MCR p15, 0, %0, c9, c12, 0\n" :: "r"(value));
}
```

The *initialize ()* function as defined in the code is called with parameter 1 and 0 i.e. 32-bit variable *reset* is assigned with value 1 and *divider* is assigned with value 0. The comments in red colour explain the various bits set by the code. *asm volatile* keywords are used to embed assembly instruction MCR in ‘C’ code. The comparison of MCR instruction, used in this code with template given in the beginning of this section, clarifies

that instead of ARM register Rd, the immediate value is used. $opcode1$ and $opcode2$ is assigned the value 0 in this instruction. A brief introduction of writing inline assembly in C is also given in Appendix-II.

B) $c9, c12$ $(opcode2 = 1)$-Count Enable Set Register:

The purpose of the CouNT ENable Set register (CNTENS) is to enable or disable any of the Performance Monitor Count Registers. The CNTENS Register is also accessible as determined by "$c9, c14$ $(opcode2 = 0)$- User enable Register" (Sr. No. E in Table 3).

The format of CNTENS is given in Figure 25.

31 30 4 3 2 1 0

C Reserved

P3 P2 P1 P0

Figure 25. Format of Count Enable Set Register [149]

The bit C is set to Enable Cycle Counter and P3, P2, P1 and P0 are set to enable all counters. In *initialize(int32_t reset, int32_t divider)* function, the following instructions has been used:

```
/* Immediate value passed to source register make sure all bits as CNTENS are set. Opcode2 also changed to 1 to access CNTENS register */
        asm volatile  ("MCR p15, 0, %0, c9, c12, 1 \n" :: "r"(ox8000000f));
```

C) $c9, c12$ $(opcode2 = 3)$-Overflow Flag Set Register:

The purpose of the Overflow Flag Status register (FLAG) is to enable or disable any of the performance monitor counters producing an overflow flag. While writing this register, any overflow flag written with a value of 0 is ignored i.e. it does not get updated. Any overflow flag written with a value of 1 clears the counter overflow flag to 0. The FLAG Register is also accessible as determined by "$c9, c14$ $(opcode2 = 0)$- User enable Register" (Sr. No. E in Table 3).

Format of FLAG register (i.e. bit positions required to be set) are exactly same as CNTENS register and the following instruction in *initialize(int32_t reset, int32_t divider)* function are used to clear overflows.

```
// Immediate value passed to source register make sure all bits as FLAG register are set.
Opcode2 changed to 3 to access FLAG register
        asm volatile  ("MCR p15, 0, %0, c9, c12, 3 \n" :: "r"(ox8000000f));
```

D) $c9, c13$ $(opcode2 = 0)$-Cycle Count Register:

The purpose of the Cycle CouNT (CCNT) register is to count the number of clock cycles since the reset of register. The CCNT Register is also accessible as determined by "$c9, c14$ $(opcode2 = 0)$- User enable Register" (Sr. No. E in Table 3).

In all previous cases, the specific bits of registers were enabled by writing using MCR coprocessor instruction, but this register will be read using MRC instruction. The following code has been used to access CCNT register after calling *initialize(int32_t reset, int32_t divider)* function.

Two calls to *Read_cycle(void)* function are made, one before *Hash( )* and other after *Hash( ).* The difference in these values give the cycles consumed by *Hash( )* function.

```
static inline unsigned int Read_cycle(void)
{
   unsigned int value;  // This variable will be used to store cycles clicked since last reset
   /* Read CCNT register. In this case operational register has changed to c13 and opcode 2
      has been set to 0 */
   asm volatile  ("MRC p15, 0, %0, c9, c13, 0 \n" :: "r"(value));
   return (value); }
```

E) $c9, c14$ $(opcode2 = 0)$-User Enable Register:

This is one of the most important register for accessing all the preceding registers. In all previous registers (PMNC/CNTENS/FLAG/CCNT), it was mentioned that these registers are accessible as determined by '$c9, c14$ $(opcode2 = 0)$-User Enable Register". Also a few of the above registers are accessible only in privileged mode e.g. PMNC register. However, setting of EN bit in "$c9, c14$ $(opcode2 = 0)$- User enable Register" enables user mode access of all performance monitor registers.

USERN is writable only in privileged mode and readable in any processor mode. The format of USERN register is given in Figure 26.

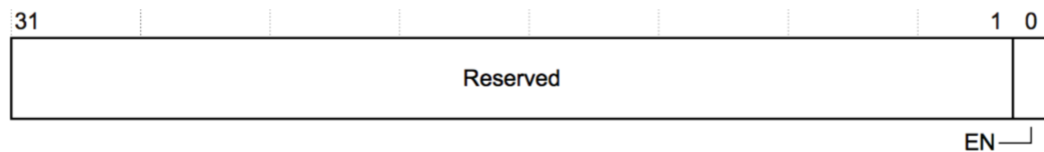


Figure 26. Format of User Enable Register [149]

For enabling all performance monitor registers, EN bit needs to be set. The instruction used to set the same in the present study is as follows:

```
asm volatile  ("MCR p15, 0, %0, c9, c14, 0 \n" :: "r"(1));
```

However, this instruction is neither written in *Read_cycle(void)* function nor in *initialize(int32_t reset, int32_t divider)* function. In fact, the above instruction acts as a prerequisite for both these functions (*Read_cycle* and *initialize*). These functions will work only if all performance monitoring registers have been enabled using the above instruction. Also this instruction can not run in *user* mode and this instruction has to be executed in supervisor mode. More details about this are in the next section.

F) $c9, c14$ $(opcode2 = 2)$-Interrupt Enable Clear Register:

The purpose of the INTerrupt ENable Clear (INTENC) register is to determine if any of the Performance Monitor Count Registers, PMCNT0 to PMCNT3 and CCNT, generates an interrupt on overflow. The INTENC is a read/write register and its format is exactly same as of CNTENS (given in Figure 25). When reading this register, any interrupt overflow enable bit as 0 means the interrupt overflow flag is disabled and bit 1 indicates the interrupt overflow flag enabled. When this register is written, any interrupt overflow bit written with 0 is ignored i.e. not updated and any interrupt overflow bit written with 1 clears the corresponding interrupt overflow enable bit to 0.

The following code is used to disable counter overflow interrupts.

```
asm volatile  ("MCR p15, 0, %0, c9, c14, 2 \n" :: "r"(0x8000000f));
```

Practically this situation will not happen in any evaluation being done in this thesis. The maximum value that a 32-bit register can store is 4,294,967,295 and none of the coded algorithm under discussion in this study consumes so many cycles. Cycle count is also reset after every computation.

This register is also accessible only in privileged mode. So just like USEREN, there is a challenge in writing into INTENC register as well. The next section deals with this challenge.

3.5.4 Writing Kernel Module for Accessing USEREN and INTENC Register

The challenge in hand was to write USEREN and INTENC register of coprocessor CP15. As discussed in the foregoing section, these registers can not be written in user mode as these are accessible only in privileged mode. This challenge can be overcome with the help of a Loadable Kernel module which are piece of code that can be loaded and unloaded from the kernel on demand. This offers an easy way to extend the functionality of a base kernel without having to rebuild or recompile the kernel again and also allow execution of a code in privileged supervisor mode. Majority of device drivers are also implemented as Linux kernel modules.

So, a kernel module containing the MCR instructions for enabling bits in USEREN and INTENC register was written. The source code of kernel module written for this thesis (named *accessCP15.c*) is given in Appendix–II. The ***makefile*** used to ***make*** the code *accessCP15.c* is also given in Appendix-II. The ***make*** commands result in *accessCP15.ko* file which is the desired loadable Kernel module.

The kernel module is loaded using ***insmod*** *accessCP15.ko* command and unloaded using ***rmmod*** *accessCP15.ko* command.

3.5.5 Step-wise Process at Host and Target Machine for Performance Analysis

A) At Host Machine

I. Create a Linux Kernel Module as mentioned under the head '3.5.4 Writing Kernel Module for Accessing USEREN and INTENC Register'.

II. Code all the SHA-3 final round candidate algorithms in C with the same methodology as specified under the head '3.5.2-C) Optimized 32-bit Implementation of Hash Algorithms Used'. All codes follow API Profile for SHA-3 Submissions as mandated by NIST in [153]. The output of coded algorithms must match the test vectors submitted in respective NIST submissions.

III. Write *Read_cycle(void)* and *initialize(int32_t reset, int32_t divider)* functions in all the coded algorithms to access Cycle Count Registers as detailed in '3.5.3 How Coprocessor CP15 is Used to Access Cycle Count Register (CCNT)'.

IV. Generate binaries for all algorithms, coded in step III, using GNU cross-platform development chain provided with OpenBoard-AM335x. The command used for compilation is:

arm-cortexa8-linux-gnueabi-gcc algo_code.c -o algo_binary

V. Transfer the following files from the host machine to target machine using the ***scp*** programme explained in '3.5.1-G) SCP Program and Interface Configuration Utility'

a. Loadable kernel module – *accessCP15.ko* (created in Step-I)

b. The binaries as obtained in step IV.

c. All input files on which binaries will run and generate results.

The command used for transfer is *scp source_file root@192.168.1.11:/home*

A) At Target Machine

I. Access target machine through Minicom serial console using following command and enter password.

sudo minicom.

II. For confirmation, check the listing of files to ensure files sent from host machine has been copied.

III. Load the Kernel module *accessCP15.ko* using following command

insmod accessCP15.ko.

IV. Run object file (binaries sent using **scp** command from host machine) one by one. The algorithms were coded to read KAT files provided by NIST and the codes generate output files containing message digests and cycle counts for each separate input message. The output files are transferred back using ***scp*** command to host machine for further analysis.

3.5.6 Results and Discussion

For each algorithm, results were obtained for four different hash sizes i.e. 224, 256, 384, and 512 bits. For each hash size of an algorithm, results were recorded for 2554

different inputs – input values starting from 0 bit, 1 bit up to 34304 bits. The various input values chosen were same as KAT – Known Answer Test values asked by NIST [154]. The input values in KAT were classified as 'Short' and 'Long' message values and the results presented here are also categorized accordingly. The 'Short' input values range from 0 bit to 2040 bits (255 bytes) in a step of 1 bit each and 'Long' input values vary from 2048 bits to 34304 bits (256 bytes to 4288 bytes) in steps of 64 bits each. So, for each hash size of an algorithm, 2554 inputs were given and the obtained results were recorded for analysis.

In this section, the results of all the five algorithms for 224-bit, 256-bit, 384-bit, and 512-bit hash sizes are presented in graphs.

A) Results for Short Messages

The results for Short messages are presented in Figure 27, Figure 28, Figure 29, and Figure 30. The major observations on the results are as follows:

A-I) Blake, Keccak, and Skein compete closely

i. **For 224 and 256-bit** hash output, Blake stands out as the most efficient algorithm, closely followed by Keccak. Skein, with an average CPB of 469, about double than 213 CPB of Blake, is considerably far at No. 3.

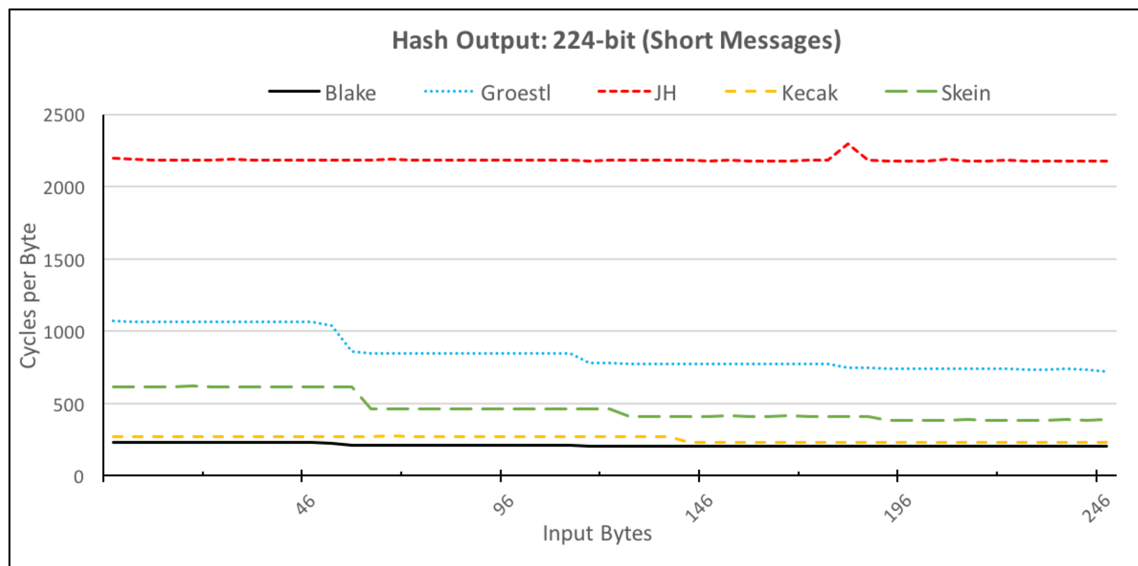


Figure 27. Results on Cortex-A8 for Short Messages (224-bit Message Digest)

ii. For **384-bit** output, Keccak outperforms Blake. For shorter messages, Blake perform better than Skein but as message size increases, Skein starts improving and performs almost as good as Blake.

iii. **For 512-bit** hash output, these three algorithms are quite close. For shorter messages, Keccak is slightly better than Blake followed by Skein. However, as the message size increases, performance of all algorithms come quite close to each other.

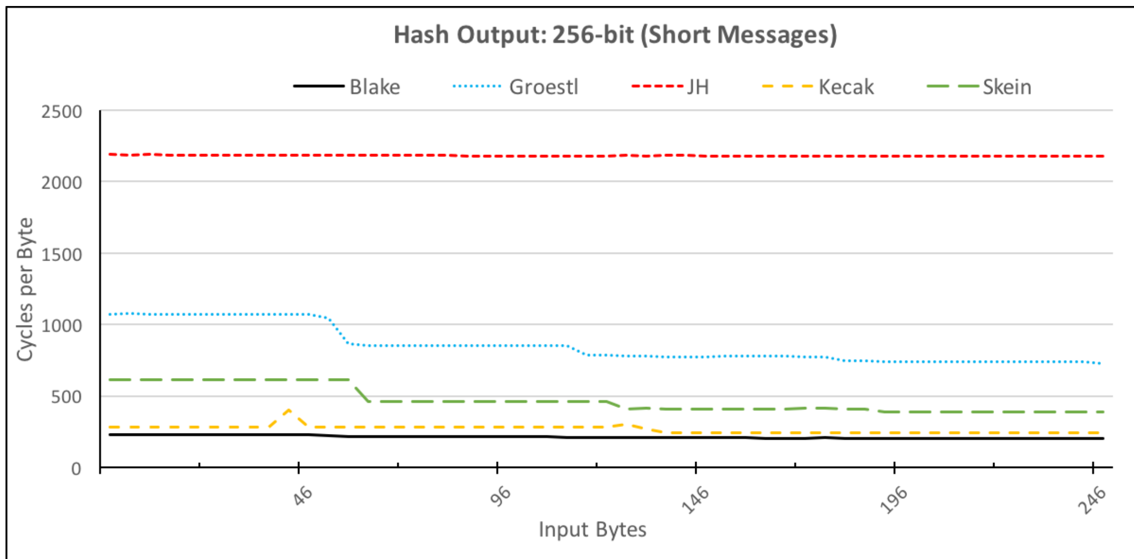


Figure 28. Results on Cortex-A8 for Short Messages (256-bit Message Digest)

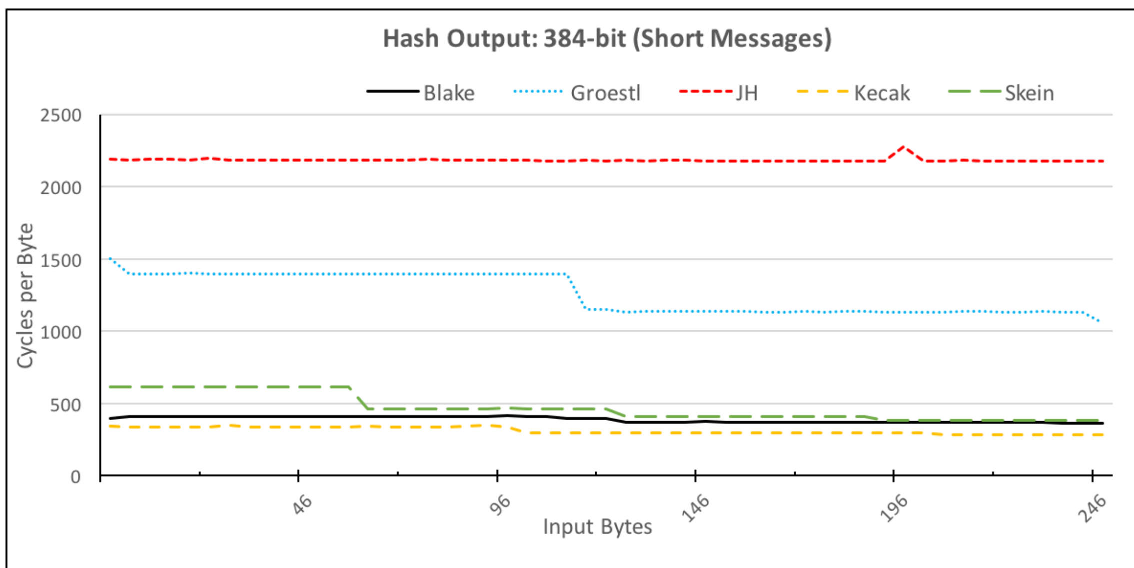


Figure 29. Results on Cortex-A8 for Short Messages (384-bit Message Digest)

A-II) Grøstl is at No. 4

i. For all output sizes (224, 256, 384 and 512-bit), **Grøstl** stands at No. 4.

ii. **For 224 and 256-bit** output, on an average, **Grøstl** takes approximately **1.8 times more** clock cycles per byte than No. 3 performer and **for 384 and 512-**

bit output, **Grøstl** takes about **2.6 times more** clock cycles per byte than No. 3 performer

A-III) JH is at the last

i. For all output sizes (224, 256, 384 and 512 buts), **JH** stands at No. 5.
ii. **For 224 and 256-bit** output, on an average, JH takes approximately **2.5 times more** clock cycles than No. 4 performer and **for 384 and 512-bit** output, JH takes about **1.7 times more** clock cycles than No. 4 performer.

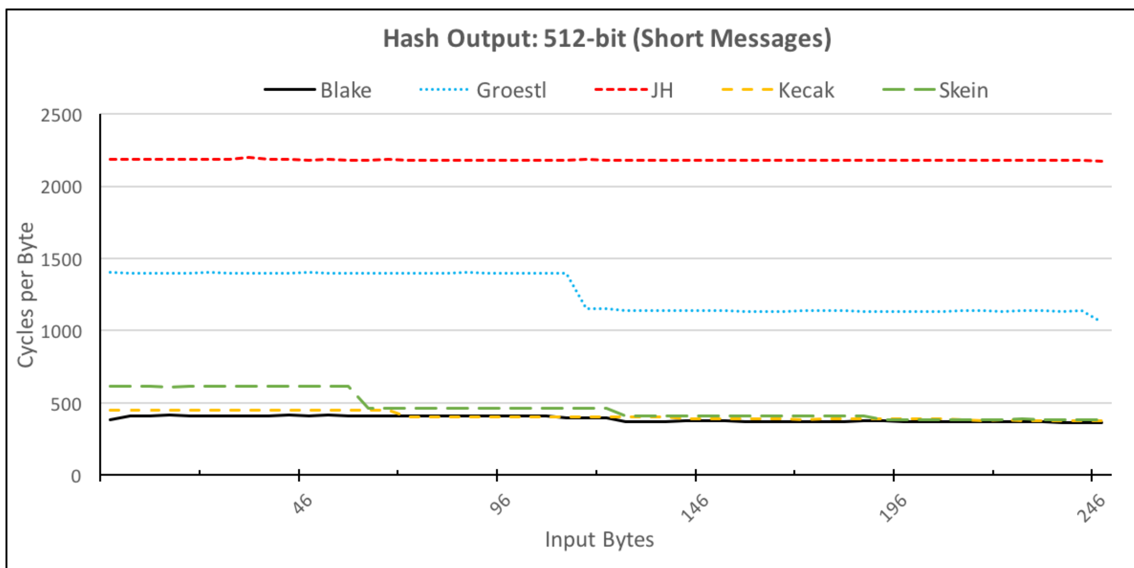


Figure 30. Results on Cortex-A8 for Short Messages (512-bit Message Digest)

A-IV) CPB improves with input size: CPB consumed by an algorithm starts improving as the input size increases. This is more obvious in Skein and Grøstl compared to other algorithms. A close look at Table 4 clarifies it better. For Grøstl and Skein, the CPB values decreases from 1st quartile to median to 3rd quartile. But for other algorithms this change is less evident. For JH, it is the least visible.

B) Results for Long Messages

For Long messages (256 to 4288 bytes) results are shown in Figure 31, Figure 32, Figure 33, Figure 34 and have been elaborated upon as done in case of Short messages.

B-I) Blake, Keccak, and Skein compete closely

i. **For 224 and 256-bit** output, Keccak and Blake perform almost equal and stand out from other algorithms. Skein, with average of 352 CPB compared to approximately 198 for Keccak and Blake, is at No. 3.

ii. **For 384-bit** output, Skein narrowly outperforms Blake, but Keccak is slightly better than Skein.

iii. **For 512 bit** output, Skein stands at No. 1 as it narrowly overtakes Keccak which is at No. 2 followed by Blake at No. 3.

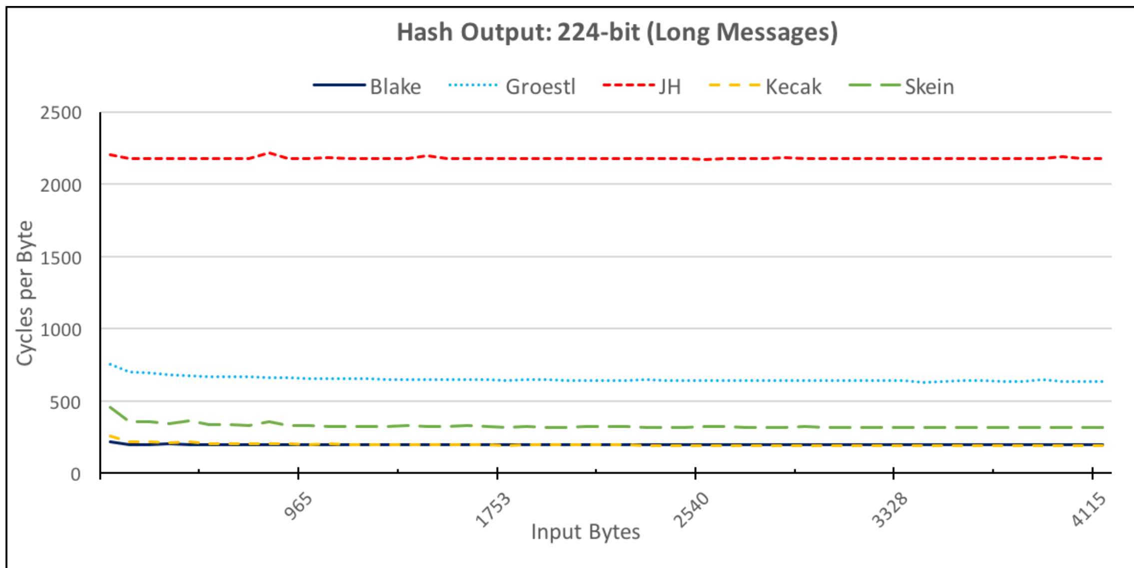


Figure 31. Results on Cortex-A8 for Long Messages (224-bit Message Digest)

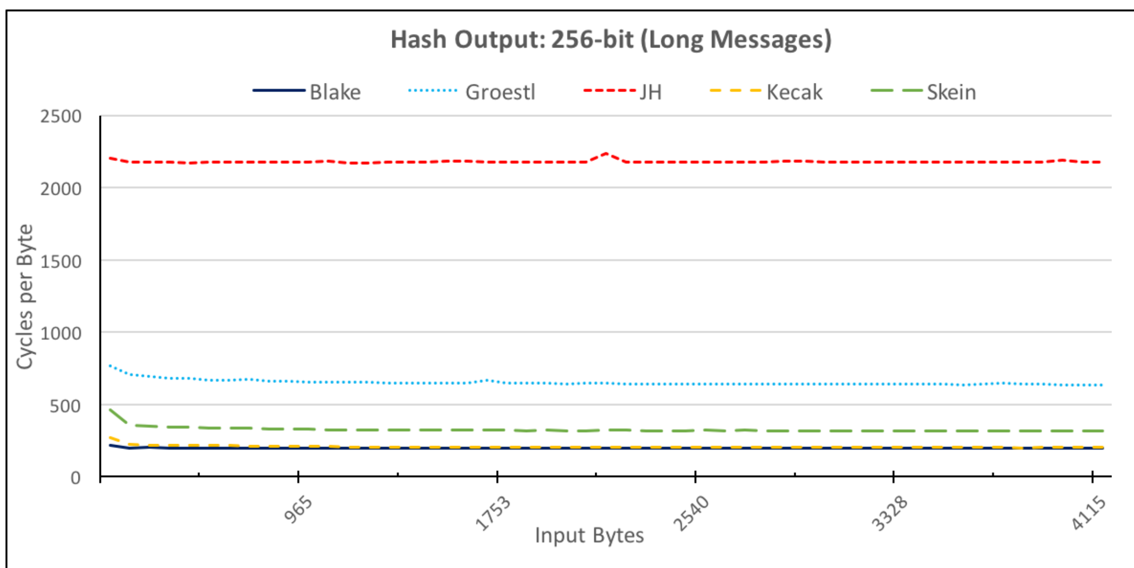


Figure 32. Results on Cortex-A8 for Long Messages (256-bit Message Digest)

B-II) Grøstl at No. 4 and JH at No. 5

i. Just like Short messages, for all output sizes (224, 256, 384, and 512-bit), **Grøstl** stands at No. 4 and **JH** at No. 5**.**

ii. On an average, **Grøstl** takes approximately **2 times more** CPB than No. 3 performer for 224 and 256-bit hash, and **2.5 times more** CPB than No. 3 performer for 384 and 512-bit hash.

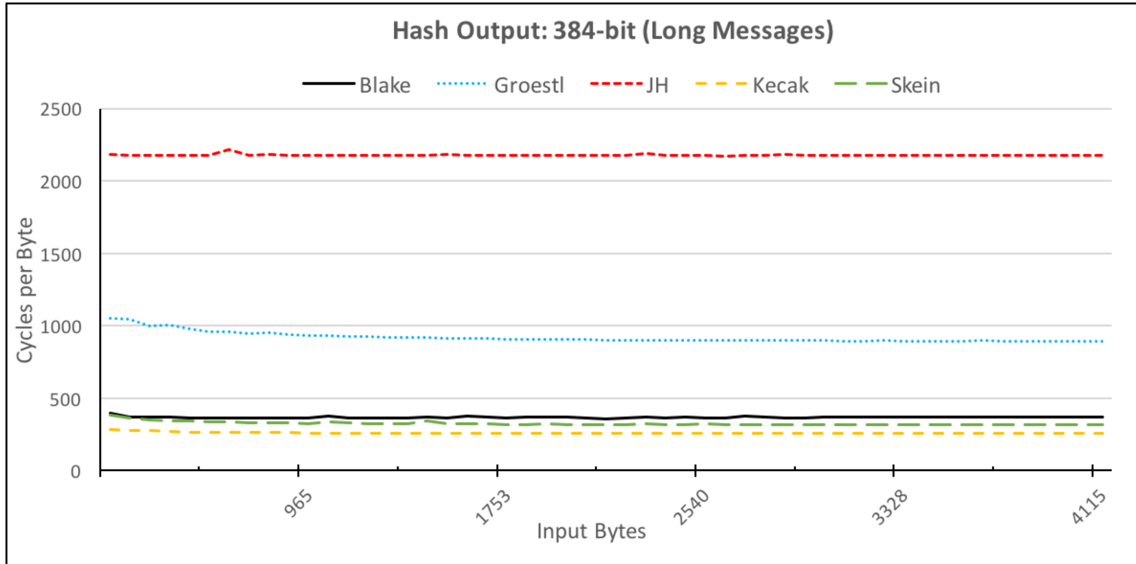


Figure 33. Results on Cortex-A8 for Long Messages (384-bit Message Digest)

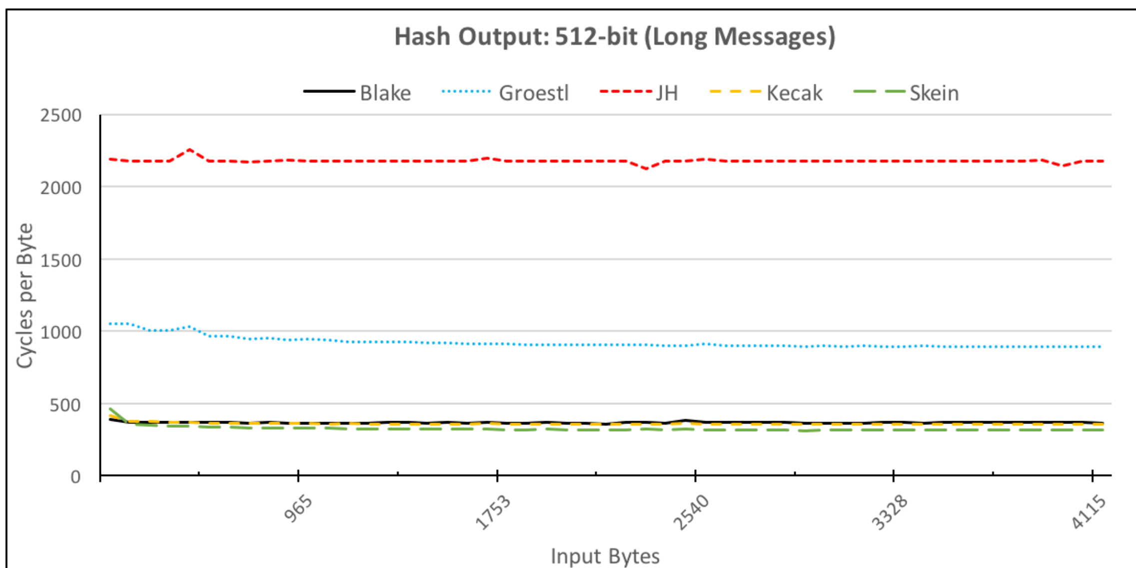


Figure 34. Results on Cortex-A8 for Long Messages (512-bit Message Digest)

iii. On an average, JH takes approximately **3.35 times more** CPB than No. 4 performer for 224 and 256-bit hash, and **2.35 times** more CPB than No. 4 performer for 512-bit hash.

B-III) Change in CPB with input size: Just like Short Messages, in Long messages also the 'Cycles per Byte' consumed by an algorithm improve (reduce) as the input size increases. However, this change is less evident as compared to the change observed in 'Short' messages. This trend is considerably visible in Grøstl and least visible in JH. For other three algorithms, CPB do reduce but in comparison to Grøstl the change is lesser.

C) Few Important Observations and Recommendation

To conclude the performance analysis of of SHA-3 final round candidate algorithms on Cortex-A8 processor, in addition to the graphs shown above, Table 4 is presented which contains median, 1st quartile (QL), 3rd quartile (QL), and mean (calculated after arranging data in decreasing order) of CPB consumed by various algorithms. The major observations and conclusions drawn from the above mentioned graphs and Table 4 are listed here.

i. For all algorithms, the **results are same for 224-bit and 256-bit hash sizes** i.e. CPB taken by an algorithm for 224-bit hash and 256-bit hash are almost the same. Similarly, the results for **384-bit and 512-bit hash are quite close**. The reason for such a behaviour lies in the internal structure of all algorithms. All authors have deigned their algorithms keeping 256-bit hash and 512-bit hash as their prime proposal. Generally, 224-bit hash is extracted from 256-bit hash proposal and 384-bit hash is extracted from 512-bit hash proposal using truncation and/or output transformation. The rest of mechanism like padding etc. is same and thus number of calls to compression function are also same for 224 and 256-bit hash (and similarly for 384-bit and 512-bit hash). Based on this observations, this study has considered only 256-bit hash and 512-bit hash for comparisons on remaining ARM architectures.

ii. As we increase message digest from 224/256-bit to 384/512-bit, the cost in terms of CPB increases in Keccak, Blake, and Grøstl i.e. more cost for better security margins. But on this front, **Skein looks better as CPB does not increase for Skein as we increase the size of message digest from 224/256 bits to 384/512 bits**. The logic behind this lies in the internal state size used by Skein. A close insight into the submission of Skein algorithm reflects that for 224 and 256-bit hash output, the Skein's authors used 512-bit internal state i.e. the internal state is atleast double

the size of the hash output which is in line with recommendation of Stefan Lucks in [49] and many other authors. Lucks in [49] suggested that Joux multicollisions [46] and length extension attacks [48] are mainly based on internal collisions which can be avoided if we widen the internal pipe (internal state) and make it at least double the size of hash output. Designs by Biham and Dunkleman [51] and Nandi and Paul [52] also utilize wide internal state.

Table 4. Mean, Median and Quartiles of CPBs Consumed by SHA-3 Finalists on Cortex-A8

SHA-3	Msg. Type	Short Messages (input from 0 bits to 2040)				Long Messages (input from 2048 bits to 34304)			
	Hash Size	224	256	384	512	224	256	384	512
Skein	Average	468.9	468.6	468.4	468.9	325.2	324.8	325.8	325.6
	1st QL	543.4	537.8	539.2	539.7	326.5	326.0	327.1	326.1
	Median	461.7	460.9	460.0	461.6	320.4	320.2	320.3	320.3
	3rd QL	411.0	410.4	410.8	410.8	318.0	317.7	317.9	317.5
Keccak	Average	253.9	264.0	312.7	407.9	197.9	207.0	259.9	360.3
	1st QL	271.4	282.4	340.0	447.4	198.7	207.9	260.2	360.9
	Median	270.8	281.4	298.4	403.7	195.0	204.3	257.6	358.3
	3rd QL	231.2	241.7	297.6	387.1	193.4	202.8	256.2	357.4
Blake	Average	212.5	212.9	388.8	389.7	196.7	196.9	367.3	367.4
	1st QL	213.5	214.1	408.6	410.1	197.0	197.2	369.9	370.0
	Median	207.0	207.3	391.8	383.3	196.1	196.2	367.8	367.7
	3rd QL	203.8	204.0	369.4	370.1	195.6	195.7	364.3	364.3
Grøstl	Average	846.6	851.1	1255.9	1258.3	650.1	650.6	919.6	919.4
	1st QL	854.9	864.9	1395.1	1397.9	653.3	653.6	926.6	926.4
	Median	783.7	786.3	1148.4	1150.0	643.9	643.7	904.7	904.9
	3rd QL	744.6	746.8	1134.2	1135.6	639.6	639.6	896.0	895.8
JH	Average	2182.2	2181.2	2182.9	2181.6	2176.1	2177.7	2177.6	2176.7
	1st QL	2183.6	2183.3	2183.5	2183.0	2176.9	2176.9	2177.2	2177.2
	Median	2180.8	2180.5	2180.9	2180.4	2175.4	2175.4	2175.4	2175.3
	3rd QL	2178.8	2178.4	2178.8	2178.5	2174.4	2174.5	2174.4	2174.3

However, for 512-bit hash output, Skein's authors did mention that Skein-1024 (with internal state of 1024-bit) may also be used but their submission to NIST uses Skein-512 (with internal state of 512-bit only) i.e. the size of internal state is not more than hash output and this characterstic may be exploited to generate inner collisions that may lead to length extension and joux multicollision attacks.

Contrary to Skein, other competing SHA-3 submissions have internal state considerably wider than hash output size. If the size of internal state of Skein for 512-bit hash output is increased, the benefit that Skein algorithm reflects on this front (CPB does not increase with increase in size of message digest i.e. CPB cost remains same with increased security margins) may dilutes a little. Experiment was carried to cross check this and obtained results reflect that to generate 512-bit hash, Skein's CPB consumption with wide internal state for short messages does increase but it decreases for longer messages. Table 5 gives average CPB consumed for generating 512-bit hash on small messages and long messages using Skein-512 and Skein-1024 (wide internal state).

Table 5. Comparison of Skein-512 and Skein-1024 for Generating 512-bit Hash

Short Messages (up to 255 bytes)		Long Messages (more than 255 bytes)	
Skein-512	Skein-1024 (Wide Pipe)	Skein-512	Skein-1024 (Wide Pipe)
468.9 CPB	600.4 CPB	324.9 CPB	243.3 CPB

Also till date, no specific attack on Skein based on this characterstic (internal state same as hash output) has been reported and hence author's proposal i.e. Skein-512 (with internal state of 512 bit) is used for the comparison on ARM architecture (Crotex-A8 and other processors).

iii. As the input size increases , CPBs decreases for all algorithms except JH. This trend is more prominent in Skein and Grøstl in comparison to other algorithms. For example, Skein consumes about 385 CPB for a 255 byte message and this CPB consumption reduces to 317 **(by about 17.6%)** as the size of message is increased to 4200 bytes. Other algorithms show similar behaviour but percentage change may be lesser. It means hash functions handle large input in better manner because for smaller size input the bare minimum block(s) {because of padding and MD stregthening} and certain other initialization and finalization phases are always processed.

iv. Certain spikes are also visible in the graphs which are because of some system interrupts that could have happened while recording the clock cycles on our target board.

v. **Recommendation:** From the above graphs and tables it is evident that for security applications requiring use of hash functions in ARM Cortex A8 based devices, **Grøstl** and **JH** are **not good choices**. Skein seems a good option for long messages as the cost in terms of Cycles per Byte does not increase when used for bigger message digest of 512 bits (rather than 224/256 bit for better security margins). Skein is also the fastest of the lot for long messages producing 512 bit output. For short messages and particularly for 256-bit hash size, Blake is a good option as it outperfoms Keccak also for 224/256 and 512 bit hash lengths.

3.6 Performance Analysis of Algorithms on ARM Cortex-M4 Processor

This section concentrates on performance evaluation of SHA-3 final round candidate algorithms on **ARM Cortex-M4** processor. Details of Hardware and Software tools used, techniques/ approach adopted, and results arrived at are given in this section.

3.6.1 Hardware and Software Tools Used

A) Cortex–M4 Core

From the Cortex Embedded series, ARM Cortex-M4 processor was picked. This is a 32-bit embedded processor and is considered conceptually as an enhanced version of Cortex–M3 with DSP (Digital Signal Processing) instructions and optional FPU (Floating point unit). The processor which contains core with FPU is referred to as Cortex-M4F otherwise it is simply Cortex-M4. Cortex-M4 is based on ARMv7E-M architecture, supports Thumb and Thumb2 instruction set, and has 3-stage pipeline as well branch speculator [146]. Important point about Cortex-M4, and for that matter about all embedded processors series, is the absence of MMU (Memory Management Unit) as these processors will be used in environment without operating system. Cortex-M4 is used in broad range of devices including microcontrollers, control systems, wireless networking, and sensors. A few important microcontroller chips based on Cortex-M4 are Texas Instruments LM4F, STMicroelectronics STM32 F3, NXP LPC 4000, LPC4300, Toshiba TX04.

B) Target Machine: Stellaris® LM4F232 Evaluation Board (EK-LM4F232)

For evaluation, Stellaris® LM4F232 Evaluation Board (EK-LM4F232) from Texas Instruments was used. Board is an evaluation platform for Stellaris LM4F232H5QD

microcontroller which is based on ARM Cortex^{TM}-M4F processor. The board supports USB2.0 OTG/Host/Device interface that helped to connect it with the host machine. Evaluation kit also supported On-board Stellaris® In-Circuit Debug Interface (ICDI) which is required for debugging the code burnt on the MCU. Board also features 256-KB Flash memory (where code can be burnt) and 32-KB SRAM (for storing data), RTC and a battery backed hibernation module. The MCU operates on 80-MHz [155].

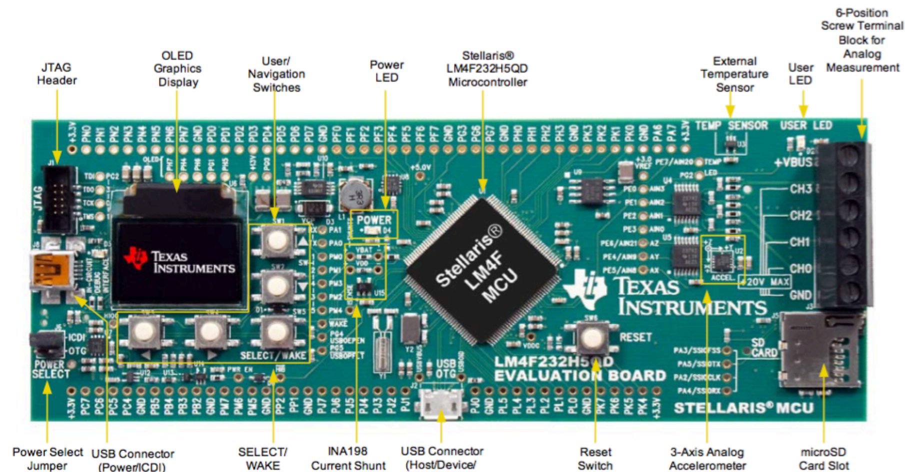


Figure 35. Cortex-M4 Based Stellaris LM4F232 Board from Texas Instruments [155]

Contrary to the Cortex-A8 based board (used in the last section), this target machine does not have any operating system. However, TI provided Code Composer Studio^{TM} IDE to be used on host machine for connecting and using the target board.

C) Host Machine

Dell Inspiron N410 Laptop housing 2.67 GHz Intel Core i5 M480 processor with 4 GB DDR3 RAM was used as a host machine. The host machine runs Microsoft Windows 7.0.

D) Host and Target Machine Setup

The host machine (Dell Inspiron laptop running Microsoft Windows) and Target machines were connected using USB connection as the target machine also provided USB2.0 interface.

In order to use the target machine for debugging, downloading, and running the executable files of coded SHA-3 programmes in the microcontroller’s Flash memory and for using Virtual COM Port connectivity using Stellaris based In-Circuit Debugging Interface (ICDI), certain drivers (Stellaris Virtual Serial Port, Stellaris ICDI JTAG,

Stellaris ICDI DFU) as provided by TI are required to be installed on the host machine. The detailed instructions, used to install drives and setup the host machine so that it can be used to connect and work with EK-LM4F232, are given in [156].

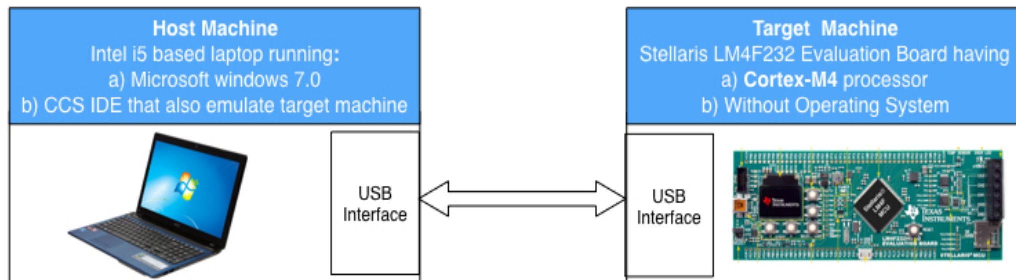


Figure 36. Host and Target Machine Setup for Stellaris EK-LM4F232

E) Code Composer Studio

Code Composer Studio is an Integrated Development Environment provided by Texas Instruments for TI embedded processors only. It includes debugger, compiler, editor, and simulator. The IDE is built on Eclipse open source software framework that has been extended by TI to support device capabilities. The details of installing CCS on host machine are available in [156]. CCS may be used as a simulator to compile, debug, and run the codes without connecting the target machine. However, for this study, CCS was used for compiling, debugging, burning, and running the code on Stellaris LM4F232 board. This required considerable settings to be done on CCS so that executable could be generated for the target machine and then executed on target machine after burning the binary on flash of MCU (target machine). As the target board is a bare machine without any OS, so even the basic settings like size of stack and size of heap to be used are required to setup, otherwise code may not run properly. The step wise process followed for using CCS (version 5.2.1.00018) to debug, burn, and run the code are given in Appendix-III.

3.6.2 Methodology Used

Methodologies and techniques adopted for performance evaluation on Cortex-A8 are used here also. There are:

a) CPB is used as performance metric {already discussed in Section (3.5.2 – A)}
b) Evaluation is done for hash function as a whole rather than for Compression Phase alone {already discussed in section (3.5.2 – B)}

c) Optimized 32-bit implementation of Hash Algorithms is used {discussed already in Section (3.5.2 – C)}. However, for compilation, in place of GNU cross-platform development chain, the inbuilt compiler of CCS (Code Composer Studio) was utilized.

d) Averaging the Cycle Count and Subtracting the Overhead {already discussed in Section (3.5.2 – E)}. However, the overhead in this case is quite less as the target machine is without OS. So, processor is dedicated in executing the coded algorithm burnt on flash.

e) **Data Watchpoint and Trace Unit (DWT) of Cortex-M4F used for counting cycles**

Cortex-M4 core offers many options to add on to the basic core. DWT (Data Watchpoint and Trace Unit) is an important add-on component that is used for debugging and profiling along with other such components like ITM (Instrumentation Trace Macrocell) and ETM (Embedded Trace Macrocell). DWT consists of counters for Clock Cycle (CYCCNT), folded instructions, Load Store Unit (LSU) operations, sleep cycles, Interrupt overhead etc. CYCCNT counter of DWT was used for profiling the codes under this study.

Code Composer Studio provides an important feature named 'Count Events' that helps to read these counters and provide necessary profiling information on graphical interface. For using this feature ('Count Events'), breakpoint needs to be put around the instructions which we want to profile. Then breakpoint properties can be setup to count any of the events like clock cycles (CYCCNT), exception overheads, sleep cycles, and LSU (load-store unit) operations etc. as mentioned above. The properties were set to count **clock cycles**. Hardware configuration of breakpoints can also be configured to reset clock cycle count at each breakpoint or let clock cycle count increment irrespective of breakpoint and give cycle count since start of the execution of the code in MCU. This study made use of the first option i.e. hardware configuration of breakpoint which was configured to reset clock cycle count at each breakpoint. So the event reading, recorded on second breakpoint, gave the clock cycles consumed by the code written between two breakpoints. The detailed steps that were followed to do profiling (in terms of clock cycles consumed by the code) on CCS are detailed in Appendix – III.

f) **No support for file handling**

Cortex-A8 based board, that were used in previous section, runs on embedded Linux and thus supports file handling. File handling could help record results for all 2554 different input values specified by NIST in [154] as KAT – Known Answer Test values. However, **Cortex-M4 based boards do not run any operating system** and therefore do not support file handling either. For every input value, the cycle count reading had to be recorded individually as mentioned in previous paragraph. Each single input value required two breakpoints to be setup (one before the code and the other after the code) and then use 'count event' feature to compute and record the clock cycles consumed by the code. This process could not be iterated automatically, contrary to what could be done on board supporting operating system and file handling, as it required considerable manual intervention. So for each algorithm, results were recorded for 100 different input values (short as well as long messages) ranging from 8 bits to 34304 bits.

3.6.3 Results and Discussions

Based on our observations mentioned under the head '3.5.6-C) Few Important Observations and Recommendation', it was concluded to conduct comparison of algorithms under discussion for 256-bit hash and 512-bit hash outputs only. The results for Short message (input message up to 2040 bits i.e. 255 bytes) and Long messages (from 2048 bits to 34304 bits i.e. 256 bytes to 4288 bytes) are presented separately on similar lines as done for Cortex-A8.

A) Results for Short Messages

Figure 37, Figure 38, and first part of Table 6 present the performance of SHA-3 final round candidate algorithms on Cortex-M4 processor for short messages. The major observations are listed below:

A-I) Blake performs the best among the lot: For 256 as well as 512-bit hash outputs, Blake is clearly the best performer. For 256-bit hash output, Blake on an average takes 108 Cycles per Byte (CPB) which is about 27% lesser than that of No. 2 performer and for 512-bit hash output this value improves further and goes up to 40%.

A-II) Keccak and Skein share the 2nd and 3rd position: For 256-bit hash, Keccak with 149 CPB is clearly at No. 2 and Skein with average of 255 CPB is at No. 3. However,

clear cut distinction between these two near performers diminishes somewhat for 512-bit hash. Keccak's performance with an average of 258 CPB and median of 256 is consistent. Skein also gives average around 263 CPB but its performance improves only with increase in input size. For small input size Skein is behind Keccak but outperforms Keccak when input size increases beyond 130 bytes.

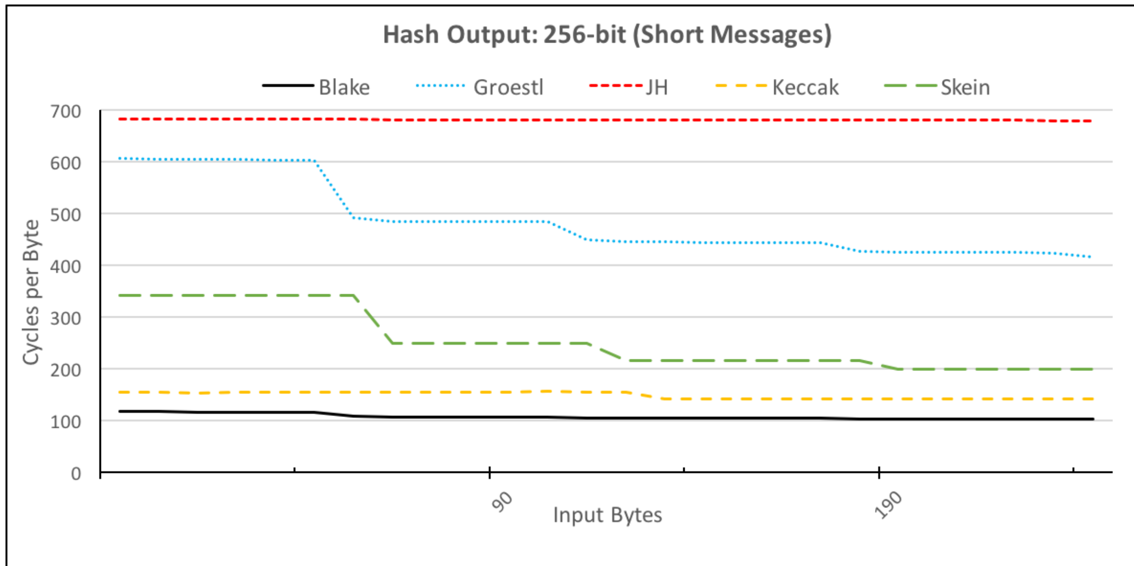


Figure 37. Results on Crotex-M4 for Short Messages (256-bit Message Digest)

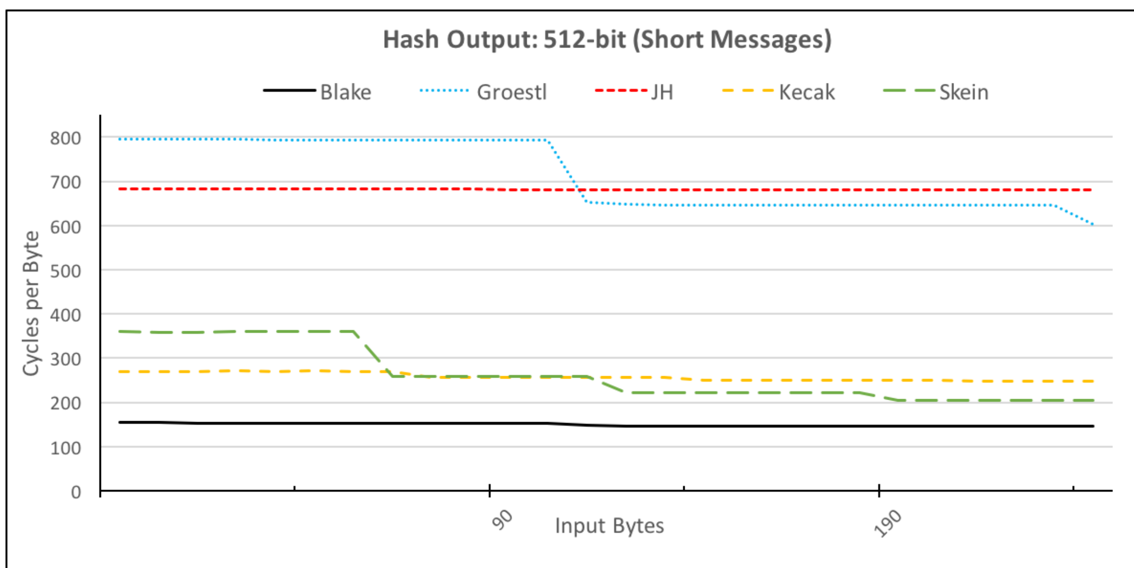


Figure 38. Results on Cortex-M4 for Short Messages (512-bit Message Digest)

A-III) Grøstl and JH share the last two positions

i. **Grøstl** and **JH** are at the last two positions. **For 256-bit** hash JH is at No. 5 and Grøstl is at No. 4 with considerable difference in performance. Grøstl, on

an average consumes 486 CPB and JH consumes 682 i.e. about **1.4 times** more CPB than Grøstl.

Table 6. Mean, Median, and Quartiles of CPBs Consumed by SHA-3 Finalists on Cortex-M4

SHA-3 Finalists	Msg. Type	Short Msg. (input from 0 to 2040 bits)		Long Msg. (input from 2048 bits to 34304)	
	Hash Size	**256**	**512**	**256**	**512**
Skein	**Average**	254.7	263.7	159.7	160.5
	1st Quartile	319.2	333.8	160.7	161.6
	Median	233.4	240.6	156.6	157.1
	3rd Quartile	216.9	222.6	155.0	155.3
Keccak	**Average**	149.2	257.6	131.9	240.8
	1st Quartile	154.5	270.2	132.3	241.2
	Median	154.3	255.9	131.1	240.1
	3rd Quartile	142.8	250.4	130.6	239.7
Blake	**Average**	107.9	149.6	99.4	141.4
	1st Quartile	109.0	152.8	99.5	141.5
	Median	105.3	147.6	99.1	140.9
	3rd Quartile	104.2	146.5	98.9	140.7
Groestl	**Average**	486.0	713.2	375.4	525.6
	1st Quartile	490.3	793.9	376.6	532.8
	Median	447.5	650.3	371.8	517.0
	3rd Quartile	432.4	646.6	369.6	510.5
JH	**Average**	681.6	681.8	679.1	679.0
	1st Quartile	682.3	682.6	679.1	679.0
	Median	681.2	681.4	678.8	678.9
	3rd Quartile	680.6	680.7	678.8	678.8

ii. F**or 512-bit**, JH and Grøstl shuffle their positions. In fact, JH continues to consume the same CPB but performance of **Grøstl deteriorates to 713 CPB** taking it to the last position.

iii. The difference in performances of JH and Grøstl in comparison to the first three performers is quite a lot. **For 256-bit** hash output, No. 4 performer takes **1.9 times more** clock cycles than No. 3 performer and **for 512-bit** hash output the difference **widens and increases up to 2.6**.

A-IV) Change in CPB with input Size: Skein and **Grøstl** show considerable improvements with increase in input size. Generally, the CPB consumption reduces as input size increases. **For Skein**, this change is **more than 25%** (for both 256-bit hash and 512-bit hash) as we move from **1st quartile to median** and **for Grøstl** it is around **9% for 256-bit hash** and **18% for 512-bit hash.** There is **no visible change** for JH. Performances of Keccak and Blake do improve but not as considerably as those of Skein and Grøstl**.**

B) Results for Long Messages

For Long messages (256 to 4288 bytes), results are shown in Figure 39, Figure 40 and Table 6. The results are quite similar to the results of short messages. The top performers are same differing only in percentage improvement. Details are given below:

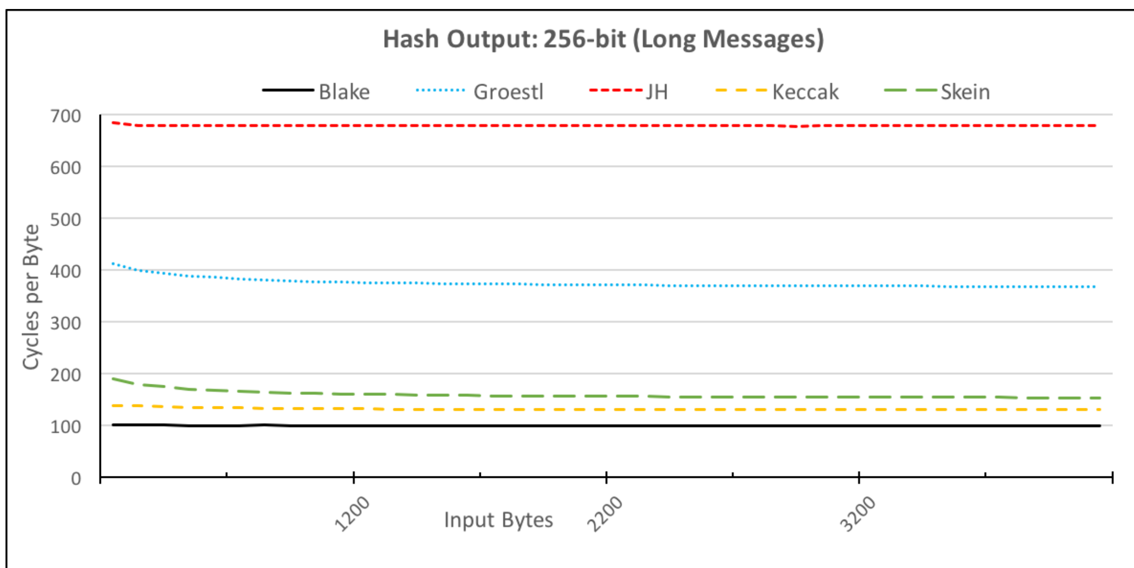


Figure 39. Results on Cortex-M4 for Long Messages (256-bit Message Digest)

B-I) Blake performs best out of the lot: Similar to Short messages, Blake seen to perform better than the other algorithms for both 256 and 512-bit hash. Its performance (in terms of CPB) is around 37.7% better than the 2nd position holder for 256-bit hash. However, for 512-bit hash, this value goes up to 11.9% only.

B-II) Keccak and Skein share 2nd and 3rd Position

i. **For 256-bit** hash, Keccak with an average of 132 CPB is at No. 2 and Skein with average of 160 CPB is at No. 3.

ii. However, for **512-bit** hash, Skein outperforms Keccak and with better margins. In this case Skein consumes 160 CPB compared to 241 by Keccak.

B-III) Grøstl and JH share last two positions

i. Contrary to short messages (where JH and Grøstl switch their position with change in hash size), for long messages **JH is at last position** for **256-bit** as well **512-bit hash (**with about 679 CPB).

ii. Grøstl is at No. 4 position and looks much better than JH in terms of performance for 256-bit hash. However, the difference in performance narrows down for 512-bit hash.

iii. Compared to the other 3 algorithms, performance of these two is quite slow. Grøstl consumes more than double CPB compared to No. 3 performer. JH is even slower taking about 1.8 times more CPB than Grøstl for 256-bit hash and 1.3 times more in case of 512-bit hash.

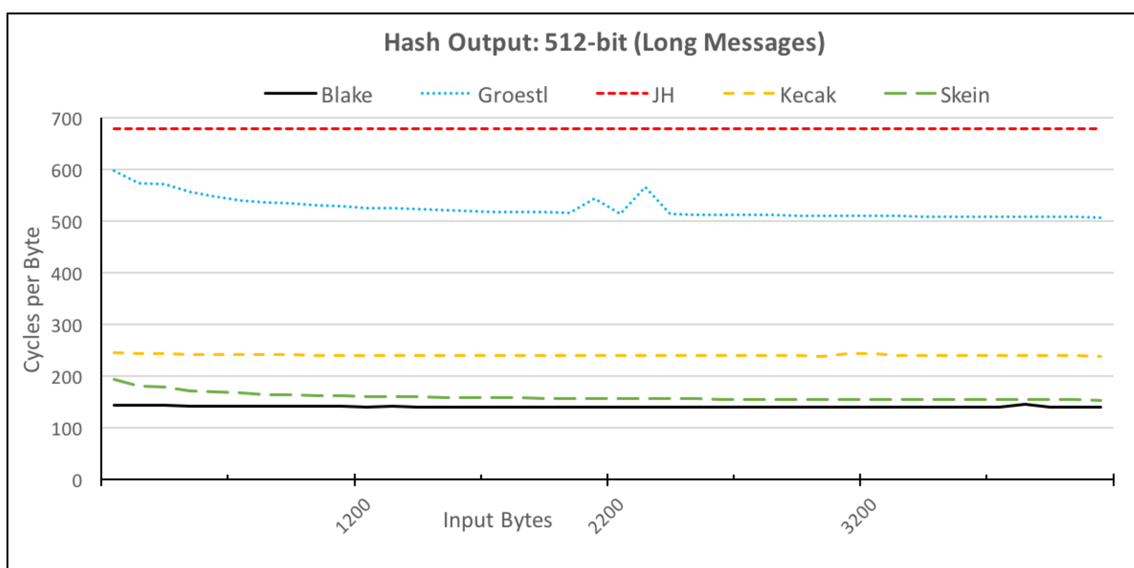


Figure 40. Results on Cortex-M4 for Long Messages (512-bit Message Digest)

B-IV) Change in CPB with input Size: For long messages**,** this trend is visible but not as evident as it is for short messages. **Skein** and **Grøstl** show this behaviour more than the other three algorithms

C) Few Important Observations and Recommendations

i. For ARM Cortex-M4 based devices, **Blake** is the best choice as it performs better than all the algorithms for all message digests irrespective of input sizes. **Skein is**

also a good second option in cirumstances where, for higher security margins, we need 512-bit hash compared to 256-bit hash. The reason to recommend Skein is based on its characterstic that cost in terms of CPB does not increase as we move from 256-bit hash to 512-bit hash.

ii. Comparison of results of all the algorithms on Cortex-M4 with the results obtained on Cortex-A8 processor reflects that cycles consumed by all the algorithms on Cortex-M4 are lesser than those by Cortex-A8. The following table gives a glimpse of the same:

Table 7. Comparison of Performance on Cortex-A8 and Cortex-M4

	256-bit Hash (Short Msg.)	256-bit Hash (Long Msg.)	512-bit Hash (Short Msg.)	512-bit Hash (Long Msg.)
Top performer on Cortex-A8	212.9	196.9	389.7	325.6
Top Performer on Cortex-M4	107.9	99.4	149.6	141.4

This result looks interesting because of the fact that though Cortex-A8 is more powerful core as compared to Cortex-M4 but algorithms took more cycles on the former. The reason behind this lies in the environment in which these cores are being utilized. Cortex-A8 is running an operating system with memory management unit and also supports multiple other devices. So whenever a code is run from user space, considerable time is spent to run other kernel threads also which are out of the control of the user. However, in case of Cortex-M4 based machine, there was no operating system and only the code burnt on its flash was being executed on this bare machine so no other threads were consuming CPU cycles. Because of this, results reflect lesser consumption of cycles for Cortex-M4 based machine compared to Cortex-A8 based machine.

3.7 Performance Analysis of Algorithms on Classical Processor-ARM7TDMI

The last processor, that we picked for evaluation of SHA-3 final round candidate algorithms was ARM7TDMI from ARM7 series of processors. As per ARM website

[146], ARM7TDMI has been one of the highest shipped core from ARM. Its 32-bit core, that supports Thumb 16-bit instruction set, has multiplier unit, and implements Embedded ICE logic to provide powerful debugging environment. It also supports JTAG connection for debugging. ARM7TDMI is based on ARMv4 architecture and a few important chips based on ARM7TDMI are: NXP LPC 2100/2200/2300/2400, Samsung S3C44B0X, ST Microelectronics STR7. Nokia 6110 was also designed using ARM7TDMI core.

3.7.1 Tools Used

The evaluation of the algorithms on ARM7TDMI is done **on simulator** rather than actual hardware. **IAR Embedded Workbench** by IAR Systems was used for this study. The IAR Embedded workbench is the leading C/C++ compiler, debugger, and simulator tool suite for applications based on 8, 16, and 32 bits MCUs (Microcontroller Units). The process was started with the '*Kickstart edition v6.30 (32k)*' as available from IAR systems web site. However, the final results were obtained on version 7.40.5 of 'IAR Embedded Workbench' for ARM. There are two different trial versions available on ARM website, one with size limit of 32K and other one is a time limited edition that has full functionality but works for a month. The latter one (time limited version) was used in this study as certain algorithms required more than 32K of data and code space. The other hardware that we required was a windows machine that could support IAR. The same machine, which was used as host machine for evaluation on Cortex-M4, was used in this case also. Except IAR Embedded Workbench and this host machine, no other hardware or software tool was used.

3.7.2 Methodology Used

Methodology used for evaluation was the same as mentioned for Cortex-M4 processor. The only difference lay in the way of calculation of clock cycles. The function profiler and Timeline tool of IAR embedded workbench was used to simulate the cycles consumed by the code. Function profiler and timeline tool provides the graphical interface showing cycles consumed by different functions of the code. To start the use of IAR Embedded Workbench, certain settings like selection of processor variant, endian mode, debug drivers etc. are to be configured. The settings and procedure followed to setup and use IAR for profiling the algorithms are given in Appendix-IV.

3.7.3 Results and Discussions

The results obtained on ARM7TDMI processor core through this simulator are discussed here in this section. The number of input values and approach used to present the result is same as done for Cortex-M4. First are presented the results for Short message (input message up to 2040 bits i.e. 255 bytes) and then those for Long messages (from 2048 bits to 34304 bits i.e. 256 bytes to 4288 bytes).

A) Results for Short Messages

Figure 41, Figure 42 and Table 8 present the result of SHA-3 final round candidate algorithms on ARM7TDMI processor for Short messages.

A-I) Keccak, Blake and Skein compete closely

i. **Contrary to results obtained on Cortex-M4**, no single algorithm is clearly outperforming other algorithms. Keccak, Blake, and Skein's performances are quite close to each other with **Keccak performing marginally better** than the others **for 256-bit hash**. The difference in performance of these three algorithms is marginal.

ii. For **512-bit** hash, Blake does better than Keccak but as input size increases above 120 – 130 bytes even Skein marginally outperforms Keccak (it is the same behaviour as was visible on Cortex-A8 also).

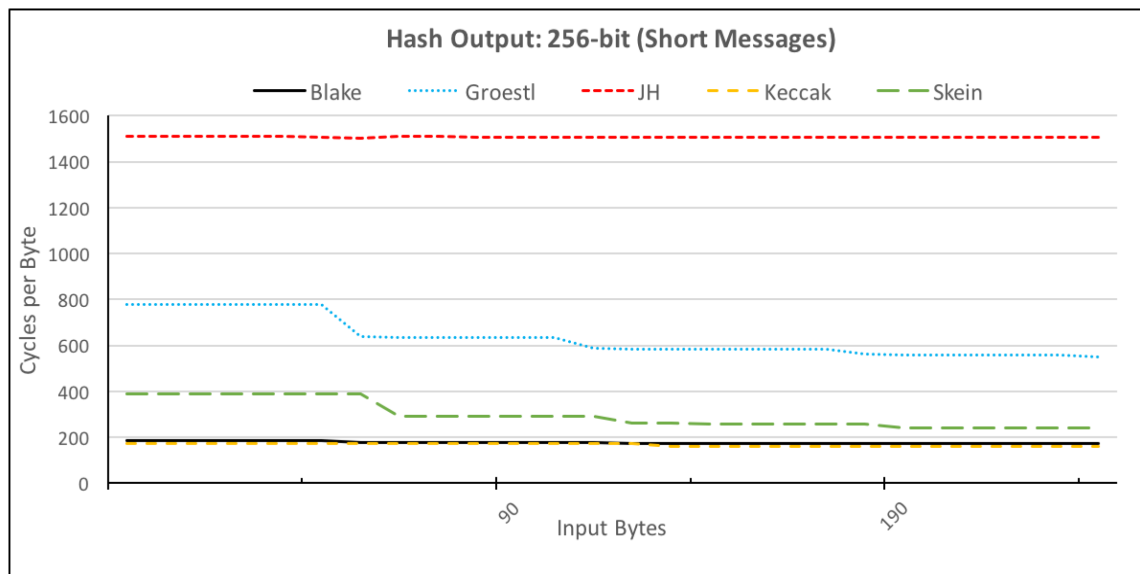


Figure 41. Results on ARM7TDMI for Short Messages (256-Bit Message Digest)

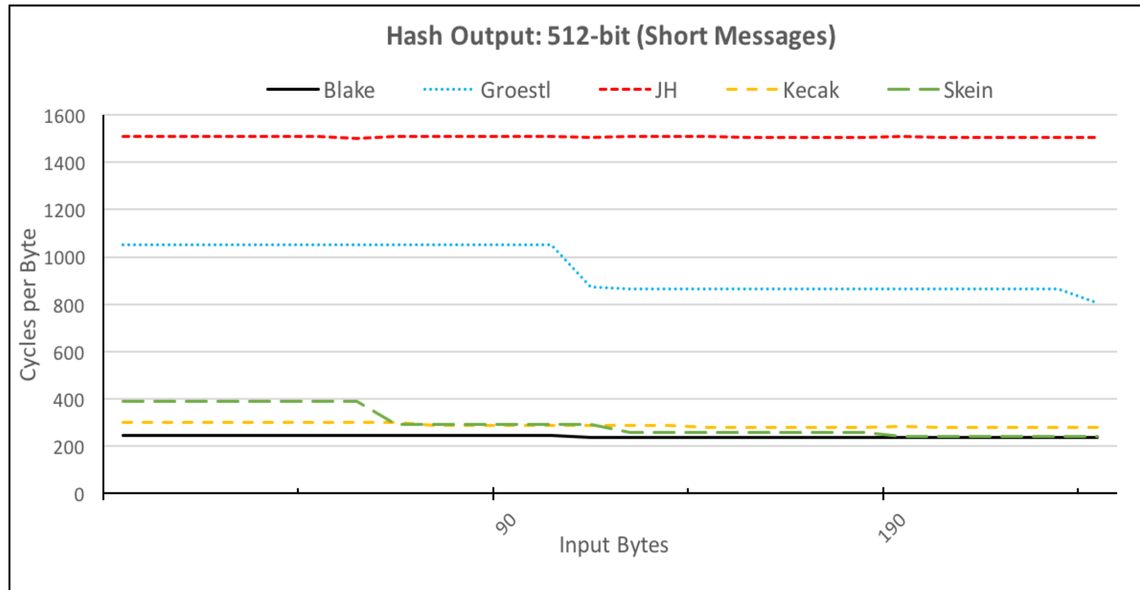


Figure 42. Results on ARM7TDMI for Short Messages (512-Bit Message Digest)

A-II) Grøstl is at No. 4 and JH is at No. 5

On ARM7TDMI also, Grøstl and JH are at the last two positions; Grøstl is at No. 4 and JH is at No. 5. JH's performance is consistent at around 1508 CPB for both 256 and 512-bit message digests. However, Grøstl does better while generating 256-bit hash (634 CBP) compared to 512-bit hash (950 CPB).

A-III) CPB reduces with increase in input size

This characteristic is evident prominently in Skein and Grøstl in comparison to the other algorithms. As we move from 1st quartile to median, change in Skein is around 25% and for Grøstl it is around 8% and 17% for 256-bit and 512-bit hash respectively.

B) Results for Long Messages

For Long messages (256 to 4288 bytes), results are shown in Figure 43, Figure 44 and Table 8.

B-I) Keccak and Skein share the first position while Blake is at No. 2

i. The performance of Keccak, Skein, and Blake is comparable but Keccak stands out as the most efficient algorithm **for 256-bit hash** (150 CPB compared to 169 by No. 2 performer, Blake).

ii. **For 512-bit** hash, Skein with about 200 CPB stands out as the most efficient algorithm compared to No. 2 performer Blake with 232 CPB.

iii. Blake retains No. 2 position **for both 256-bit hash and 512-bit hash.**

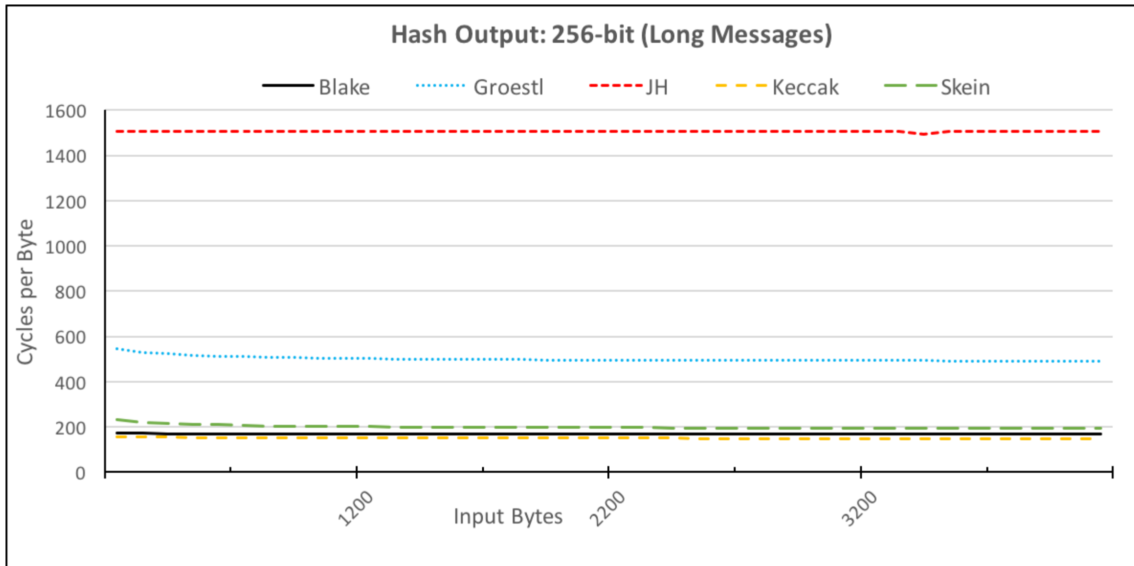


Figure 43. Results on ARM7TDMI for Long Messages (256-bit Message Digest)

B-II) Grøstl and JH are at the last two positions: Grøstl is at position No. 4 and JH is at No. 5. The performance of JH is consistent irrespective of size of message digest or size of input messages. Grøstl performs better for 256-bit hash as compared to 512-bit hash. Similar behaviour was visible for short messages also.

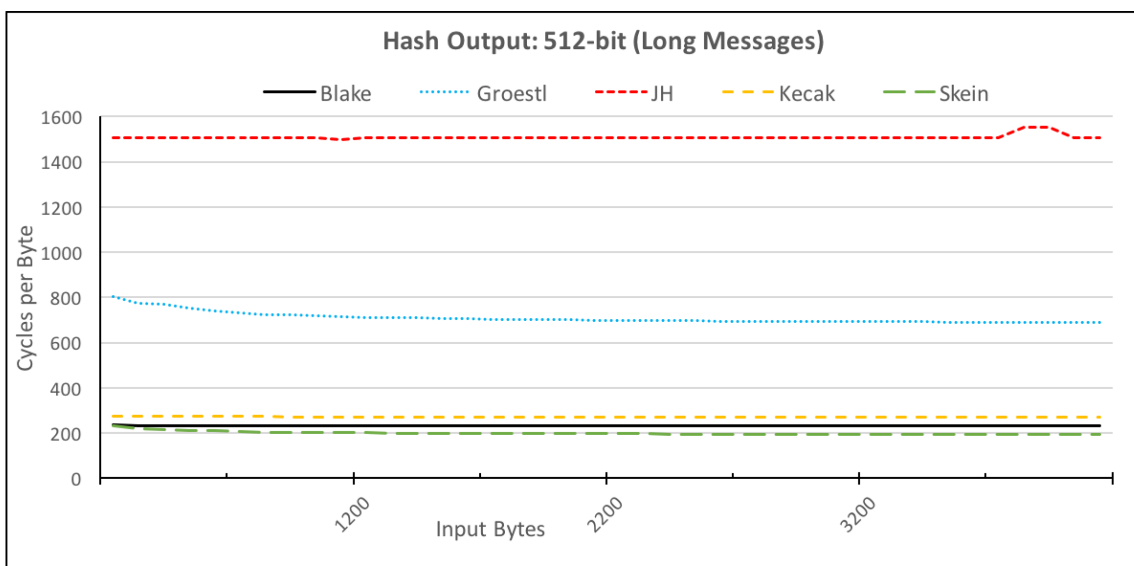


Figure 44. Results on ARM7TDMI for Long Messages (512-bit Message Digest)

B-III) Except Grøstl, reduction in CPB with increase in message size **is not prominently visible.**

Table 8. Mean, Median and Quartiles of CPBs Consumed by SHA-3 Finalists on ARM7TDMI

SHA-3 Algo.	Msg. Type	Short Msg. (input from 0 to 2040 bits)		Long Msg. (input from 2048 bits to 34304)	
	Hash Size	256	512	256	512
Skein	Average	297.1	297.0	199.6	199.6
	1st Quartile	365.0	365.2	200.6	200.6
	Median	275.3	274.1	196.5	196.5
	3rd Quartile	257.4	257.5	194.8	194.8
Keccak	Average	167.4	288.1	150.4	271.5
	1st Quartile	172.8	300.6	150.8	271.7
	Median	172.5	286.4	149.7	270.9
	3rd Quartile	161.1	280.9	149.3	270.6
Blake	Average	177.4	240.2	168.9	232.1
	1st Quartile	178.2	243.3	169.0	232.3
	Median	174.7	238.5	168.7	231.8
	3rd Quartile	173.5	237.2	168.5	231.5
Groestl	Average	633.9	949.8	499.9	709.3
	1st Quartile	637.4	1052.6	501.4	713.1
	Median	586.4	868.5	495.4	698.8
	3rd Quartile	568.4	865.5	492.9	692.6
JH	Average	1507.6	1507.7	1505.3	1505.8
	1st Quartile	1508.5	1508.6	1505.6	1505.7
	Median	1507.5	1507.6	1505.5	1505.5
	3rd Quartile	1506.9	1507.0	1505.4	1505.4

C) Recommendation for ARM7TDMI Based Devices

Close look at the results reflects that for 512-bit message digests Skein's performance is quite good whereas Keccak is seen to be performing better for 256-bit hash. Looking at the high security margins provided by 512-bit message digest, Skein is recommended. Keccak is the second best option to use for ARM7TDMI based devices particularly if the requirement is mainly for 256-bit message digest.

3.8 Concluding Remarks on Performance of SHA-3 Final Round Candidate Algorithms on ARM Architecture

This chapter attempts to provide an exhaustive detail of the rationale behind the selection of ARM architecture as Target platform for evaluation of SHA-3 final round candidate algorithms. Presenting the logic behind selection of specific processor series, performance results of algorithms on these processor series have been thoroughly discussed. After each processor series, recommendations are given on algorithms to be used on that specific processor series. A few other points that require special mention are discussed here in this denouement.

a) As all these processor series (Cortex-A, Cortex-M, and Classical Processor) are considerably related and derived from same ARM architecture so broad performance characteristics shown by the algorithms across these processor series are similar.

b) JH and Grøstl are among the bottom two performers irrespective of a) *Size of message digest*, b) *Type of Input message (long or short message)* c) *Processor series used*. So, the use of JH and Grøstl is not recommended for security applications that may require usage of hash functions on ARM architecture.

c) Skein, Blake, and Keccak have shown good performances on ARM architecture on the whole. The positions of the best performer and the No. 2/3 performer have been changing with change in *hash size* or *input message type* or *processor series used* for evaluation. Graphs shown in previous sections might reflect performance of all these to be very close and this is because of the fact that high CPB consumed by JH and Grøstl makes other graph lines look quite close. But if JH and Grøstl are taken out of discussion, then the comparison of all these three algorithms reflects considerable difference in their performances. For example, we have reproduced the results of Skein, Keccak, and Blake on Cortex-A8 for long messages producing 512-bit hash in the graph shown in Figure 45.

   The difference **in performance is up to 12.7%** and this is a considerable difference on the target machine (ARM Cortex series) which is generally characterized by low power consumption, small size, and limited processing resources.

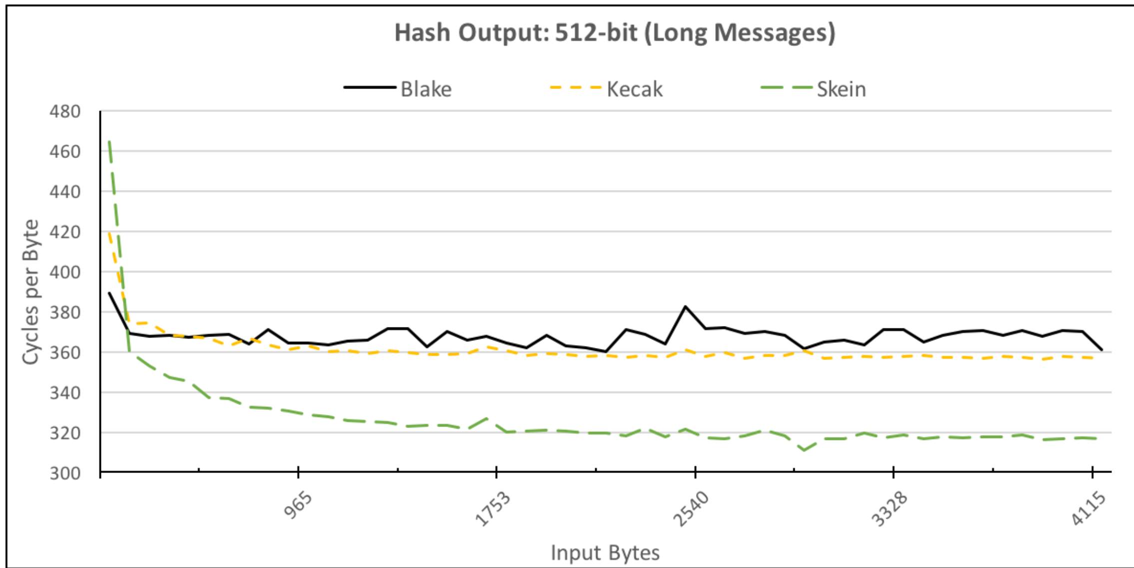


Figure 45. Performance of Blake, Keccak, and Skein for Long Messages on Cortex-A8 (512-bit Message Digest)

d) Before zeroing on to framing the final recommendation, the following points require to be highlighted:

   a. Out of all the three processor series that were discussed, Cortex-A based processor are the ones which fetch maximum applications requiring use of hash functions. So it becomes imperative for the algorithms to perform good on Cortex-A series.

   b. It is mainly the long messages that are more affected by algorithm's CPB performance. So while comparing and recommending algorithms, performance on long messages becomes very important.

   c. A Brute-force Collision attack on 256-bit hash requires 2^{128} operations (as per Birthday paradox). Of course this looks quite safe but the cryptanalysis attacks can bring it down considerably and the same had happened with multiple algorithms. As 512-bit hash gives much higher security margins so an algorithm that gives better performance on 512-bit hash becomes an important candidate for consideration.

e) Based on the above points, Skein appears to be a promising option for long messages out of the whole lot. For longer messages in general and 512-bit message digest in particular, Skein has shown good performance. Secondly, moving from 256-bit message digest to 512-message digest, the cost in terms of CPB does not increase in case of Skein. So the first recommendation of this study is to use Skein for 512-bit hash followed by Blake and Keccak for 256-bit hash. However, for

Short messages (e.g. in Password hashing, pseudo random number generator, and many such applications) Blake and Keccak are better alternatives.

The next chapter presents the design and development of a new cryptographic primitive - **M**odified **C**haCha **C**ore (MCC) that leads to the design of a new hash function which performs considerably well on ARM architecture as well as the Reference platform announced by NIST. This new hash function, as presented in the next chapter, may be used as a variant to Skein Hash family. Besides Skein, this new hash function may also be used in place of other SHA-3 final round candidate algorithms.

CHAPTER 4: DESIGNING AND DEVELOPMENT OF 'MODIFIED CHACHA CORE' - A CRYPTOGRAPHIC PRIMITIVE, LEADING TO THE DESIGN OF '*COCKTAIL*' - AN ARX BASED NEW HASH FUNCTION

"The most important part of design is finding all the issues to be resolved. The rest are details."

Sumeet Lanka

In fulfilment of the second objective of the thesis, this chapter presents a new ARX based hash function that has been designed using **M**odified **C**haCha **C**ore (MCC). In its initial sections is presented the design of MCC, the basic primitive that is used as a building block for construction of the new hash function named as ***Cocktail***. The subsequent part of the chapter details the specifications of ***Cocktail*** followed by ***Cocktail's*** design philosophy and rationale behind various design decisions. The last section presents the various ways of using ***Cocktail*** and its security aspects. As such the whole chapter is organized into multiple headings covering:

- Analysis of Quarter Rounds of Salsa and ChaCha Core and Proposal of an Alternative Design (MCC) for Maximizing Diffusion **(4.1)**
- Introduction to ***Cocktail*** and Description of Notations and Operations Used **(4.2)**
- Iterated Construction of ***Cocktail*** **(4.3)**
- Specifications of ***Cocktail-512*** **(4.4)**
- Specifications of ***Cocktail-1024*** **(4.5)**
- Complexity of ***Cocktail*** **(4.6)**
- ***Cocktail's*** Design Philosophy, Design Decisions, and its Rationale **(4.7)**
- Using ***Cocktail*** **(4.8)** and Its Security Aspects **(4.9)**

4.1 Analysis of Quarter Rounds of Salsa and ChaCha Core and Proposal of an Alternative Design (MCC) for Maximizing Diffusion

This section discusses the design of MCC – **M**odified **C**haCha **C**ore, the basic primitive, that has been used to build ***Cocktail***. MCC is an improvisation over Salsa [157] and ChaCha Core [158] that has been used for generating stream ciphers. The following few paragraphs carry a brief about stream ciphers, Salsa, and ChaCha Core. The later part of this section presents MCC and results of an experiment that reflects MCC is better than Salsa and ChaCha core.

Stream ciphers are symmetric key encryption methods in which encryption and decryption are done one symbol (character or bit) at a time. Stream ciphers generate pseudorandom key streams and each digit of plaintext is encrypted one at a time with digit of key stream. A prominent example of stream cipher is A5/1 cipher which has been used in GSM mobile phone standard for voice encryption. RC4 is another example which has been extensively used for encrypting internet traffic. Being typically smaller, stream ciphers execute faster than block ciphers and are traditionally considered more efficient than block ciphers. Efficient for software-optimized stream ciphers means they take fewer processor cycles to encrypt and efficient for hardware-optimized stream ciphers mean they take fewer gates and smaller chip area than a block cipher for encrypting data at the same data rate. However, modern block ciphers have been doing really well in software as well as in hardware. Example is AES [5] in software and PRESENT [159] in hardware.

About eSTREAM: ECRYPT - **E**uropean Network of Excellence in **Crypt**ology, a European research initiative launched on 1st February 2004, issued a call for steam cipher primitives in November 2004 under the Project eSTREAM [17]. The objective of eSTREAM was to identify new stream ciphers suitable for widespread adoption. Stream cipher proposals were solicited in one of the following two profiles - Profile 1: Stream ciphers for software applications with high throughput requirements. Profile 2: Stream ciphers for hardware applications with restricted resources such as limited storage, gate count, or power consumption.

The eSTREAM project was completed in April 2008. Selected Profile 1 algorithms were: HC-128, Rabbit, Salsa20/12 [157], and SOSEMANUK. The selected Profile 2 algorithms were: F-FCSR-Hv2, Grain v1, Mickey v2, and Trivium. Out of these, F-FCSR was originally in eSTREAM profile but was removed later on because of its cryptanalysis.

Salsa and ChaCha: MCC improvises one of the eSTREAM profile cipher Salsa20 (also known as Snuffle 2005) and its improvement ChaCha [158] (also known as Snuffle 2008) stream cipher. Salsa20/12 and Salsa20/8 are reduced round variants of Salsa20 encryption function (Stream Cipher) that make use of 12 and 8 rounds respectively in place of 20 rounds (10 Double rounds) of Salsa20. Salsa20 encryption functions (stream cipher) uses **Salsa20 core** that diffuses 4x4 matrix representing nonce, key and constants. ChaCha family is an improvement over Salsa family and has variants like ChaCha8, ChaCha12 and ChaCha20 corresponding to Salsa20/8, Salsa20/12 and Salsa20. Both Salsa core and ChaCha core are based on ARX operations (Arithmetic, Rotation with constant and XOR)

Salsa and ChaCha as stream ciphers: Salsa and ChaCha family arranges the data into a matrix of 4x4 elements where each element is a 32-bit word representing nonce/block number, key, and constants. Twenty rounds of Salsa20 core (or ChaCha20 core) diffuses these values and the resultant value is used to encrypt 64-byte plaintext by XORing plaintext with this hashed value (output of Salsa20/ChaCha20 core). This concept is expanded to generate encryption for longer plaintexts. The following explanation as given by Bernstein in Salsa 20 stream cipher [157] illustrates the working of Salsa as stream cipher. ChaCha20 stream cipher works in the similar fashion.

Let key (**k**) be of 32 bytes. Let '**v**' be an 8-byte sequence. Let '**m**' be an 'l'-byte sequence for some $l \in \{0,1,\ldots,2^{70}\}$. The Salsa20 encryption of '**m**' with nonce '**v**' under key **k**, denoted $Salsa20_k(\mathrm{v}) \oplus m$ is an '$\boldsymbol{l}$'-byte sequence. Normally '**k**' is a secret key; '**v**' is a nonce, i.e., a unique message number; '**m**' is a plaintext message; and $Salsa20_k(\mathrm{v}) \oplus m$ is a cipher text message. $Salsa20_k(\mathrm{v})$ is also generated as 2^{70} byte sequence represented as

$$Salsa20_k(v,o), Salsa20_k(v,1), Salsa20_k(v,2), \ldots, Salsa20_k(v,2^{64}-1)$$

Here '**i**', the second argument to $Salsa20_k(v,o), Salsa20_k(v,1)\ etc.$ is the unique 8-byte sequence $(i_0, i_1, i_2, i_3, i_4, i_5, i_6, i_7)$ such that $i = i_0 + 2^8 i_1 + 2^{16} i_2 + \ldots + 2^{56} i_7$. Combination of **k, v, i** is mapped to matrix of size 4x 4 (of 32 bits each) with some constants and Salsa20 core or ChaCha20 core (or reduced round version) is applied. The result is used as key that is XORed with plaintext to obtain encrypted value.

The approach in this study: **Salsa core** and **ChaCha core** (used to hash 4x4 matrix representing nonce, key, and constants) are analysed in this study to know how much

diffusion both bring. Twenty rounds of Salsa20 core or ChaCha20 core may be viewed as 10 Double rounds, where each Double round consists of 4 Column Quarter rounds and 4 Row (or Diagonal) Quarter rounds i.e. operations on each single column/row is termed as Quarter Round. The Quarter rounds of Salsa and ChaCha use same number of operations but differ considerably. Quarter rounds of both the functions involve 32-bit Additions, 32-bit XOR, and 32-bit constant Rotations. For both Salsa and ChaCha core, Bernstein [157] [158] has suggested specific rotation constants.

An experiment was conducted to analyse diffusion property of Quarter rounds of both the functions to see how the diffusion changes with change in rotation constants. This study also proposes an alternative design of Quarter round involving same number of operations but exhibiting better diffusion properties. Based on this newly constructed Quarter round, a new core, named **M**odified **C**haCha **C**ore (MCC), is designed which uses Column and Row rounds with changed parameter values for better diffusion.

The next subsection (4.1.1), introduces the Quarter rounds of the Salsa and ChaCha core and also explains the Column and Row (or Diagonal) rounds of both the cores. The subsequent subsection (4.1.2), introduces the new design MCC. Subsection 4.1.3, explains the experiment used to analyse the diffusion of all three candidate Quarter rounds (Salsa, ChaCha, and **M**odified **C**haCha). The results of the experiment are discussed in subsection 4.1.4.

4.1.1 Salsa and ChaCha Core

To explain both Salsa and ChaCha core, bottom up approach is followed. First the basic primitive Quarter round is introduced, followed by Column and Row/Diagonal round, and lastly Double round. All the operations are carried out on 4x4 matrix where each element is 32-bit word. Let's assume the input matrix **'x'** is:

$$\begin{matrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{matrix}$$

A) Salsa Core

The operations of Salsa core [157] is introduced below:

Quarter round of Salsa core takes four 32-bit words as input and mixes these words and generates four 32-bit words as output. If $x = (x_0, x_1, x_2, x_3)$is four word input to $QuarterRD_{Salsa}(x)$, then output (y_0, y_1, y_2, y_3) is defined as

$$y_1 = x_1 \oplus ((x_0 + x_3) \lll 7)$$
$$y_2 = x_2 \oplus ((y_1 + x_0) \lll 9)$$
$$y_3 = x_3 \oplus ((y_2 + y_1) \lll 13)$$
$$y_0 = x_0 \oplus ((y_3 + y_2) \lll 18)$$

Quarter round makes use of four 32-bit XORs, four 32-bit Additions and four 32-bit Rotations (left rotation i.e. towards higher bits) with constants. The output $y = QuarterRD_{Salsa}(x)$ is graphically represented in Figure 46.

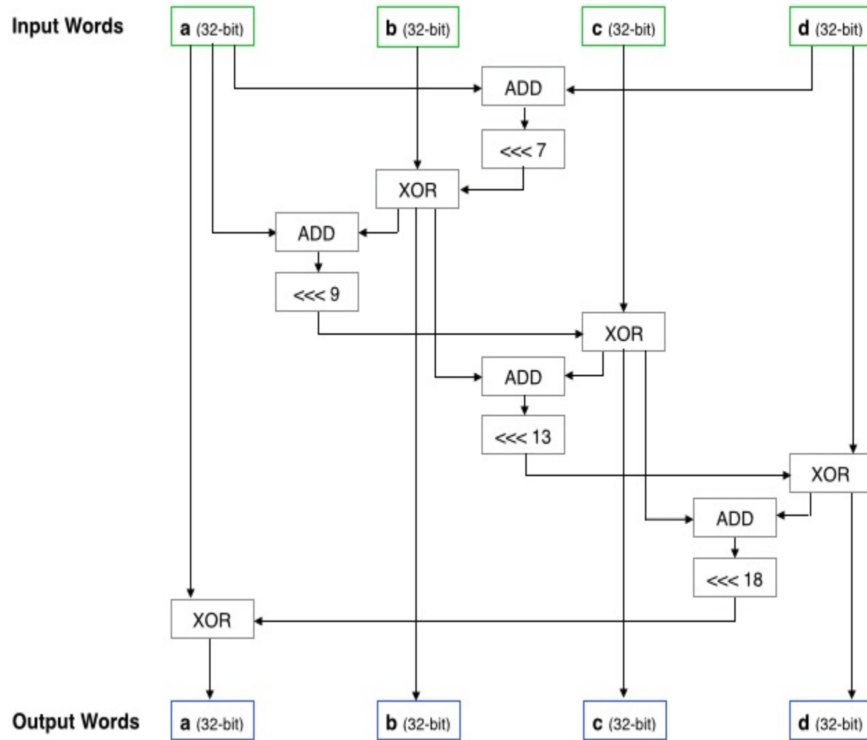


Figure 46. Quarter Round of Salsa Core

Column round takes 16 words as input and generates 16 words as output. In fact, Column round makes use of Quarter rounds. If $x = (x_0, x_1, x_2 \dots x_{13}, x_{14}, x_{15})$ is 16 words input to $ColumnRD_{Salsa}(x)$, then output $y = (y_0, y_1, y_2, \dots, y_{13}, y_{14}, y_{15})$ is defined as:

$$(y_0, y_4, y_8, y_{12}) = QuarterRD_{Salsa}(x_0, x_4, x_8, x_{12})$$
$$(y_5, y_9, y_{13}, y_1) = QuarterRD_{Salsa}(x_5, x_9, x_{13}, x_1)$$

$$(y_{10}, y_{14}, y_2, y_6) = QuarterRD_{Salsa}(x_{10}, x_{14}, x_2, x_6)$$
$$(y_{15}, y_3, y_7, y_{11}) = QuarterRD_{Salsa}(x_{15}, x_3, x_7, x_{11})$$

Row round also takes 16 words as input and generates 16 words as output. Just like Column round, Row round also makes use of four Quarter rounds. If $y = (y_0, y_1, y_2, \dots y_{13}, y_{14}, y_{15})$ is 16 words input to $RowRD_{Salsa}(y)$, then output $z = (z_0, z_1, z_2, \dots z_{13}, z_{14}, z_{15})$ is defined as:

$$(z_0, z_1, z_2, z_3) = QuarterRD_{Salsa}(y_0, y_1, y_2, y_3)$$
$$(z_5, z_6, z_7, z_4) = QuarterRD_{Salsa}(y_5, y_6, y_7, y_4)$$
$$(z_{10}, z_{11}, z_8, z_9) = QuarterRD_{Salsa}(y_{10}, y_{11}, y_8, y_9)$$
$$(z_{15}, z_{12}, z_{13}, z_{14}) = QuarterRD_{Salsa}(y_{15}, y_{12}, y_{13}, y_{14})$$

In Column (or Row) round, each Column (or Row) of the matrix is used to call Quarter round function but ordering of parameters passed to $QuarterRD_{Salsa}$ is different for each column (or row). For example for the first row, parameter passed to $QuarterRD_{Salsa}$ is (y_0, y_1, y_2, y_3) but for second row, the parameters are not passed in the same sequence i.e. rather than (y_4, y_5, y_6, y_7), parameters passed are (y_5, y_6, y_7, y_4) .

Double round takes 16 words $x = (x_0, x_1, x_2 \dots x_{13}, x_{14}, x_{15})$ as input and generates 16 words $z = (z_0, z_1, z_2, \dots z_{13}, z_{14}, z_{15})$ as output. In fact, Double round is a Column round followed by a Row round and is defined as:

$$z = DoubleRD_{Salsa}(x) = RowRD_{Salsa}(ColumnRD_{Salsa}(x))$$

Salsa20 core calls 10 Double rounds. Reduced round versions Salsa20/12 or Salsa20/8 call Double round 6 and 4 times respectively. eSTREAM [17] has Salsa20/12 cipher in its profile that uses Salsa20/12 core as detailed above.

B) ChaCha Core

ChaCha core [158], proposed by Bernstein, was an improvement over Salsa core to increase diffusion using the same number of operations. The details of ChaCha core are given below:

Quarter round of ChaCha, like Salsa's Quarter round, takes four 32-bit words as input and mixes these words and generates four 32-bit words as output. If $x = (x_0, x_1, x_2, x_3)$ is four word input to $QuarterRD_{ChaCha}(x)$, then output (y_0, y_1, y_2, y_3) is defined as:

$$u_0 = x_0 + x_1 \; ; u_3 = x_3 \oplus u_o \; ; \; u_3 = u_3 \lll 16 \; ;$$

$$u_2 = x_2 + u_3 ; u_1 = x_1 \oplus u_2 ; \; u_1 = u_1 \lll 12 ;$$
$$y_0 = u_0 + u_1 ; y_3 = u_3 \oplus y_o ; \; y_3 = y_3 \lll 8 ;$$
$$y_2 = u_2 + y_3 ; y_1 = u_1 \oplus y_2 ; \; y_1 = y_1 \lll 7 ;$$

The $y = QuarterRD_{ChaCha}(x)$ is graphically represented in Figure 47.

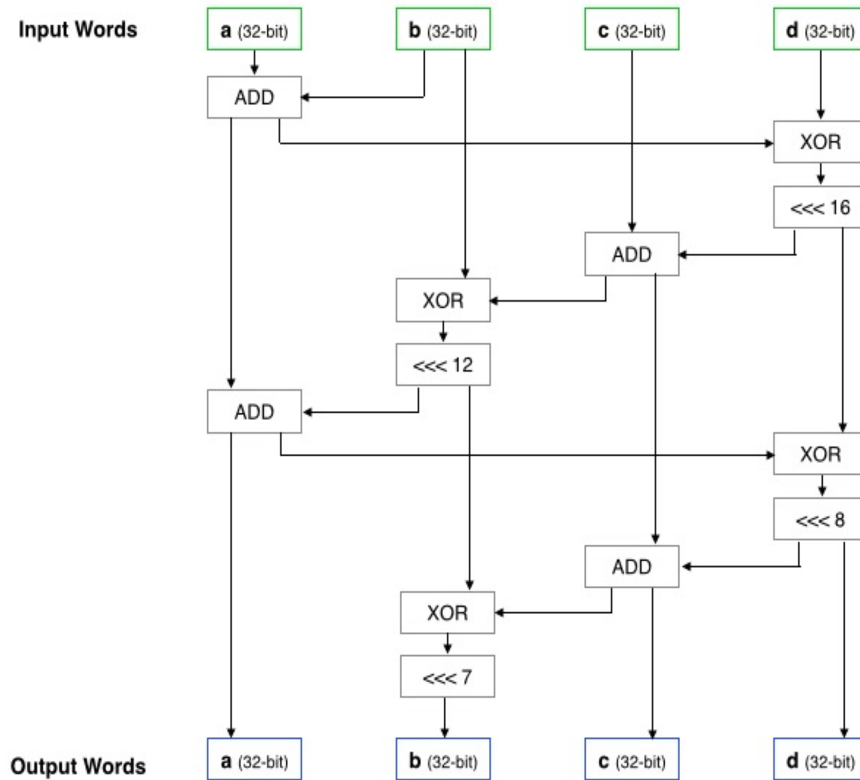


Figure 47. Quarter Round of ChaCha Core

A close look at Quarter rounds of both Salsa and ChaCha core reflects following major differences:

i. ChaCha Quarter round, unlike Salsa Quarter round, gives each input word a chance to affect each output word.

ii. ChaCha updates each input word twice. For example, initially x_0 is updated to u_0 and then updated again to y_0.

iii. Because of the above two reasons, diffusion by ChaCha Quarter round is much better than Salsa which is evident in the result of experiment discussed in subsequent sub-section.

iv. Rotation distances have been changed. Salsa uses 7, 9, 13, 18 as rotation constants while ChaCha uses 16, 12, 8, 7.

Column round of ChaCha core is a little different from Salsa core in terms of sequence of parameters passed to Quarter rounds. If $x = (x_0, x_1, x_2 \dots x_{13}, x_{14}, x_{15})$ is 16 words input to $ColumnRD_{ChaCha}(x)$, then output $y = (y_0, y_1, y_2, \dots, y_{13}, y_{14}, y_{15})$is defined as:

$$(y_0, y_4, y_8, y_{12}) = QuarterRD_{ChaCha}(x_0, x_4, x_8, x_{12})$$
$$(y_1, y_5, y_9, y_{13}) = QuarterRD_{ChaCha}(x_1, x_5, x_9, x_{13})$$
$$(y_2, y_6, y_{10}, y_{14}) = QuarterRD_{ChaCha}(x_2, x_6, x_{10}, x_{14})$$
$$(y_3, y_7, y_{11}, y_{15}) = QuarterRD_{ChaCha}(x_3, x_7, x_{11}, x_{15})$$

Diagonal round: In place of Row round, ChaCha makes use of Diagonal round that calls four Quarter rounds; one for each diagonal of input matrix. If $y = (y_0, y_1, y_2, \dots y_{13}, y_{14}, y_{15})$ is 16 words input to $DiagonalRD_{ChaCha}(y)$, then output $z = (z_0, z_1, z_2, \dots z_{13}, z_{14}, z_{15})$ is defined as:

$$(z_0, z_5, z_{10}, z_{15}) = QuarterRD_{ChaCha}(y_0, y_5, y_{10}, y_{15})$$
$$(z_1, z_6, z_{11}, z_{12}) = QuarterRD_{ChaCha}(y_1, y_6, y_{11}, y_{12})$$
$$(z_2, z_7, z_8, z_{13}) = QuarterRD_{ChaCha}(y_2, y_7, y_8, y_{13})$$
$$(z_3, z_4, z_9, z_{14}) = QuarterRD_{ChaCha}(y_3, y_4, y_9, y_{14})$$

Double round of ChaCha also takes 16 words $x = (x_0, x_1, x_2 \dots x_{13}, x_{14}, x_{15})$ as input and generates 16 words $z = (z_0, z_1, z_2, \dots z_{13}, z_{14}, z_{15})$ as output. Double round of ChaCha is Column round followed by a Diagonal round and is defined below:

$$z = DoubleRD_{ChaCha}(x) = DiagonalRD_{ChaCha}(ColumnRD_{ChaCha}(x))$$

ChaCha20 core calls 10 Double rounds. Reduced round versions ChaCha12 or ChaCha8 calls Double round 6 and 4 times respectively. One more difference in ChaCha and Salsa Core that has not been highlighted here is in the mapping of nonce/block number, key, and constants to initial matrix $'x'$.

4.1.2 Modified ChaCha Core (MCC)

A close look at the Quarter round of ChaCha core (refer Figure 47) reflects that the first and third words (word ‘a’ and ‘c’) always get updated with 32-bit addition operation, while second and fourth words (word ‘b’ and ‘d’) always get updated with an XOR operation followed by Rotation with a constant. In the proposed design (named MCC’s Quarter round), this symmetry is broken to create an alternative design as shown in Figure

48. In this proposed design, all four words are exposed to Addition, XOR, and Rotation operation. For example, word ‘b’ initially gets updated with a 32-bit addition operation and next with a 32-bit XOR and last by a 32-bit rotation operation. Similarly, all other words are updated with all three operations. The result of experiment, as discussed in next sub-section, reflect that this alternative design creates better diffusion than Quarter round of Salsa and ChaCha.

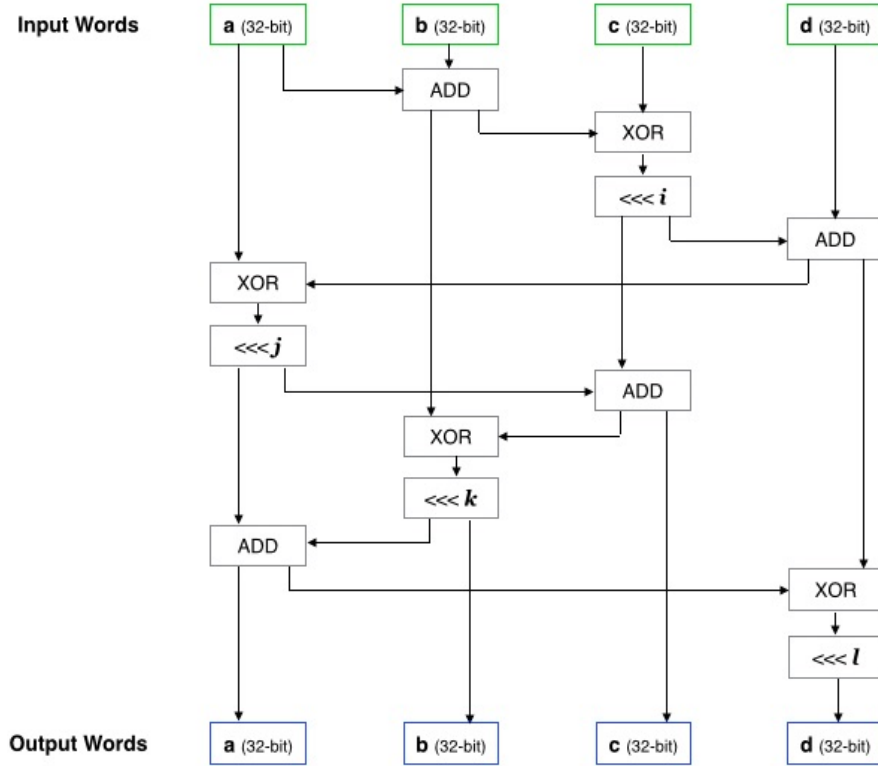


Figure 48. Quarter Round of MCC

The formal definition of Quarter round as well as Column and Row rounds of MCC is given below:

Quarter round of MCC: If $x = (x_0, x_1, x_2, x_3)$ is four word input to $QuarterRD_{MCC}(x)$, then output (y_0, y_1, y_2, y_3) is defined as:

$$u_1 = x_1 + x_0 \, ; u_2 = x_2 \oplus u_1 \, ; \; u_2 = u_2 \lll i \, ;$$
$$u_3 = x_3 + u_2 \, ; u_0 = x_0 \oplus u_3 \, ; \; u_0 = u_0 \lll j \, ;$$
$$y_2 = u_2 + u_0 \, ; y_1 = u_1 \oplus y_2 \, ; \; y_1 = y_1 \lll k \, ;$$
$$y_0 = u_0 + y_1 \, ; y_3 = u_3 \oplus y_0 \, ; \; y_3 = y_3 \lll l \, ;$$

This section does not specify any particular rotation constants. The discussion on rotation constants is done in the next section.

Column round of MCC operates in a fashion similar to Salsa's Column round. Even the parameters passed are in the same sequence. If $x = (x_0, x_1, x_2 \ldots x_{13}, x_{14}, x_{15})$ is 16 words input to $ColumnRD_{MCC}(x)$, then output $y = (y_0, y_1, y_2, \ldots, y_{13}, y_{14}, y_{15})$ is defined as:

$$(y_0, y_4, y_8, y_{12}) = QuarterRD_{MCC}(x_0, x_4, x_8, x_{12})$$
$$(y_5, y_9, y_{13}, y_1) = QuarterRD_{MCC}(x_5, x_9, x_{13}, x_1)$$
$$(y_{10}, y_{14}, y_2, y_6) = QuarterRD_{MCC}(x_{10}, x_{14}, x_2, x_6)$$
$$(y_{15}, y_3, y_7, y_{11}) = QuarterRD_{MCC}(x_{15}, x_3, x_7, x_{11})$$

Row round of MCC is different from both Salsa and ChaCha. In fact, ChaCha has diagonal round in place of Row round. If $y = (y_0, y_1, y_2, \ldots y_{13}, y_{14}, y_{15})$ is 16 words input to $RowRD_{MCC}(y)$, then output $z = (z_0, z_1, z_2, \ldots z_{13}, z_{14}, z_{15})$ is defined as

$$(z_1, z_2, z_3, z_0) = QuarterRD_{MCC}(y_1, y_2, y_3, y_0)$$
$$(z_6, z_7, z_4, z_5) = QuarterRD_{MCC}(y_6, y_7, y_4, y_5)$$
$$(z_{11}, z_8, z_9, z_{10}) = QuarterRD_{MCC}(y_{11}, y_8, y_9, y_{10})$$
$$(z_{12}, z_{13}, z_{14}, z_{15}) = QuarterRD_{MCC}(y_{12}, y_{13}, y_{14}, y_{15})$$

The reason for passing different sequence of parameters in Row and Column round is detailed in the sub-section '4.1.4 Results and Discussion'.

Double round of MCC takes 16 words $x = (x_0, x_1, x_2 \ldots x_{13}, x_{14}, x_{15})$ input and generates 16 words $z = (z_0, z_1, z_2, \ldots z_{13}, z_{14}, z_{15})$ output. Double round of MCC is Column round followed by a Row round and is defined below:

$$z = DoubleRD_{MCC}(x) = RowRD_{MCC}(ColumnRD_{MCC}(x))$$

Different number of rounds for MCC can be proposed as done for Salsa and ChaCha core like Salsa20 or ChaCha8. The number of rounds may be decided on the basis of diffusion factor required i.e. how many full diffusions are needed in a cryptographic primitive. MCC is able to generate one full diffusion in second Double round.

4.1.3 Experiment Used to Measure the Diffusion Property of Quarter Rounds

Diffusion, considered as one of the main properties of the operation of a cryptographic primitive, refers to the property of a function to quickly spread a small change in the input to maximum possible bits in the output. From the perspective of Quarter round functions, diffusion may be measured in terms of change in output words with a small change in input words. Higher diffusion means a better Quarter round function.

Quarter rounds of all three candidate designs have four rotation constants: { ***i, j, k, l*** }. For Salsa, Bernstein [157] has defined these rotation constants as {7, 9, 13, 18} and for ChaCha [158], it is defined as {16, 12, 8, 7}. For MCC, the exact value of rotation constants is determined based on the analysis of all possible diffusion matrices as detailed in this section.

To understand the diffusion property of all three candidate Quarter rounds (Salsa, ChaCha, and MCC), the matrix, as mentioned in Table 9, were calculated for all possible values of ***i, j, k, and l*** where ***i, j, k, and l*** varies from 0 to 31. This will generate 32*32*32*32 = 1,048,576 (more than one million) different matrices for each alternative design.

Table 9. Diffusion Matrix

OP / IP	a	b	c	d
a	D_{aa}	D_{ab}	D_{ac}	D_{ad}
b	D_{ba}	D_{bb}	D_{bc}	D_{bd}
c	D_{ca}	D_{cb}	D_{cc}	D_{cd}
d	D_{da}	D_{db}	D_{dc}	D_{dd}

Each cell like "D_{ab}" of the matrix (Table 9), refers average number of bits modified in word 'b', given a random one-bit difference in input word 'a'. So, the first row of the matrix reflects the change (average number of bits modified) in output words (a to d) with a one-bit difference in input word 'a'. Similarly, the second row reflects the change in output words with one-bit difference in input word 'b' and so do the third and the fourth row represent changes for input words 'c' and 'd' respectively.

For all three alternative designs, the following algorithm was executed to obtain diffusion matrices (as explained above) for all possible rotation constants.

Step 1: Take four words (a, b, c, d) of 32 bits each.
Step 2: Run the following steps (Step 3 to 9) for different values of *i, j, k, and l* ; each varying from 0 to 31 (so there will be four nested loops totalling to 1,048,576 iterations)
Step 3: Generate four 32-bit random values and assign these values to words a, b, c and d respectively.
Step 4: Run the Quarter round of the concerned design

$a', b', c', d' = QuarterRD(a, b, c, d)$ with rotation constant as *i, j, k, and l.*

[These will be called $QuarterRD_{Salsa}(a, b, c, d)$ or $QuarterRD_{ChaCha}(a, b, c, d)$ or $QuarterRD_{MCC}\ (a, b, c, d)$ depending on the design for which this algorithm is being called]

Step 5: Flip one bit of word 'a' randomly and keep all other words (b, c, and d) same and call the Quarter round functions of concerned design again.

$a'', b'', c'', d'' = QuarterRD\left(a_{flipped}, b, c, d\right)$ with rotation constant as *i, j, k, and l.*

Step 6: Compare a', b', c', d' with a'', b'', c'', d'' and find how many bits are different in each word and store it in one dimensional array D_a having four elements $[\ D_{aa}, D_{ab}, D_{ac}, D_{ad}\ ]$. D_{ab} represent change in word 'b' because of one bit flip in word 'a' i.e. difference in b' and b''. Similarly, D_{ac} represent the change in word 'c' i.e. difference in c' and c'' with one bit flip in word 'a'.

Example: If b' is 11100110111001101110011011100110 and b'' is 10111111111001101011111101111111 then it means change (No. of bits modified) in word 'b' because of change in one bit of word 'a' is 12. This will be stored in D_{ab}.

Step 7: Repeat step 5 and 6. But this time, rather than flipping one bit of 'a', flip one bit of word 'b' and keep all words same. This means, we will generate $a'', b'', c'', d'' = QuarterRD\left(a, b_{flipped}, c, d\right)$ with rotation constant as *i, j, k, l* and will accordingly calculate D_b having four elements $[\ D_{ba}, D_{bb}, D_{bc}, D_{bd}\ ]$.

Step 8: Similar to step 7, find D_c and D_d by flipping one bit of word 'c' and 'd' respectively. After all this we will have matrix 'D' with first row as $D_a = [\ D_{aa}, D_{ab}, D_{ac}, D_{ad}\ ]$; second row as $D_b = [\ D_{ba}, D_{bb}, D_{bc}, D_{bd}\ ]$; third row as $D_c = [\ D_{ca}, D_{cb}, D_{cc}, D_{cd}\ ]$ and fourth row as $D_d = [\ D_{da}, D_{db}, D_{dc}, D_{dd}\ ]$

Step 9: Repeat Step 3 to 8 one thousand times and find average of each element of D. After averaging that out, we will get diffusion matrix as mentioned in Table 9 for one set of rotation constants *i, j, k and l.*

Using the above algorithm, the obtained diffusion matrices for Salsa and ChaCha for the rotation constants {7, 9, 13, 18} and {16, 12, 8, 7} are given in Table 10 and Table 11 respectively.

Matrices in Table 10 and Table 11 clearly reflect that Quarter round of ChaCha performs better diffusion than Salsa. Salsa diffusion matrix makes it evident that change in word 'b' and 'c' does not bring much change in word 'b' and / or word 'c'.

Table 10. Diffusion Matrix of Salsa's Quarter Round

IP \ OP	a	b	c	d
a	12.205	1.955	4.316	6.430
b	7.255	1.0	1.946	4.330
c	4.333	0	1.0	1.958
d	8.784	1.958	2.468	5.649
Mean: 4.0992, Standard Deviation: 3.1887				

Table 11. Diffusion Matrix of ChaCha's Quarter Round

IP \ OP	a	b	c	d
a	4.047	11.285	9.331	5.989
b	5.987	13.418	10.825	7.794
c	2.47	6.781	4.803	2.47
d	2.399	8.528	6.751	3.399
Mean: 6.6424, Standard Deviation: 3.2731				

The above algorithm was executed for all three candidate Quarter round designs (Salsa, ChaCha, and MCC). For each design, 1,048,576 different diffusion matrices were obtained that corresponded to all 1,048,576 possible permutations of rotation constants *i, j, k, and l* (each varying from 0 to 31). For MCC, the experiment was a way to decide the exact rotation distances (constants) to be used. For Salsa and ChaCha, Bernstein [157] [158] had already prescribed the rotation constants. However, diffusion matrices for different rotation distances were still calculated with an objective to study whether there exist rotation distances that generate better diffusion than the prescribed rotation distances.

4.1.4 Results and Discussion

Each diffusion matrix contains 16 values, all representing diffusions in different words (a, b, c, and d). To evaluate all these diffusion matrices (that are more than 1 million matrices per design), the mean and standard deviation for each matrix were calculated and plotted with mean on 'x' axis and standard deviation on 'y' axis. For every single

diffusion matrix (that corresponded to a specific permutation of *i, j, k, and l)*, one point is plotted on the graph. So the graphs of Quarter round of Salsa (Figure 49), ChaCha (Figure 50) and MCC (Figure 51) represent 1,048,576 points each. The graphs for all three designs and their comparison is discussed in this subsection:

A) Results for Quarter Round of Salsa Core

As mentioned earlier, the graph for Quarter round of Salsa core is given in Figure 49. Some interesting observations from the results of Salsa's Quarter round are:

i. There are more than 45000 (4.3%) permutations of *i, j, k, and l*; that give slightly better results than the prescribed value of rotation constants i.e. mean is higher and standard deviation is lower than the mean and standard deviation obtained with prescribed rotation constants {7, 9, 13, 18}.

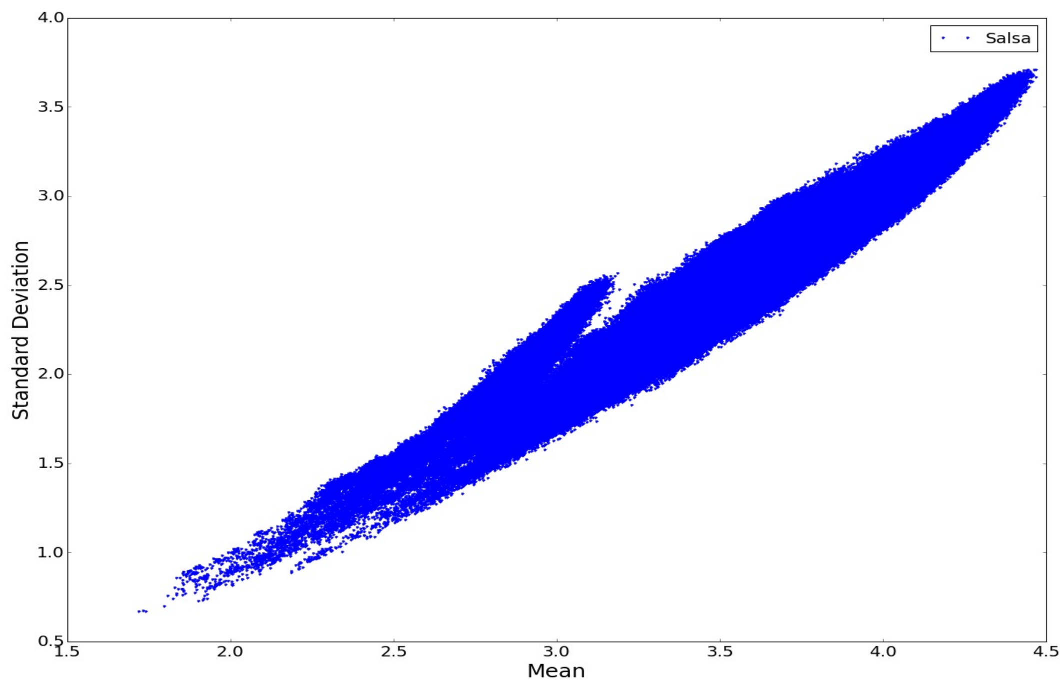


Figure 49. Mean and Standard Deviation of Diffusion Matrices of Salsa's Quarter Round

ii. The rotation constants that generate the diffusion matrix with the highest mean is {12, 21, 12, 3} i.e. use of these rotation constants will result in more diffusion than the diffusion obtained with rotation constants prescribed by Bernstein in [157]. However, the improvement in the mean is not considerable and is just slightly more than the mean obtained using prescribed rotation constants. The mean and standard deviation of diffusion matrix at {12, 21, 12, 3} are 4.2462 and 3.1821 respectively.

iii. The rotation constants that generate diffusion matrix with smallest standard deviation is {12, 18, 6, 8}. The mean and standard deviation for the diffusion matrix corresponding to these rotation constants are 4.1076 and 2.9392 respectively.

B) Results for Quarter Round of ChaCha Core

The graph for Quarter round of Salsa core is given in Figure 50. Some interesting observations that emerge from the results of ChaCha's Quarter round are given below:

i. For ChaCha Quarter round, Bernstein [158] changed the rotation constants from {7, 9, 13, 18} to {16, 12, 8, 7}. However, on examination of the diffusion matrices of ChaCha's Quarter round on both these sets, not much improvement could be found. For 1000 random values, the experiment reflected that for rotation constants {7, 9, 13, 18}, ChaCha's Quarter round created diffusion matrix with mean = **6.8377** and standard deviation = **3.2872,** and for rotation constants of {16, 12, 8, 7}, it generated diffusion matrix with mean = **6.6424** and standard deviation = **3.2731**.

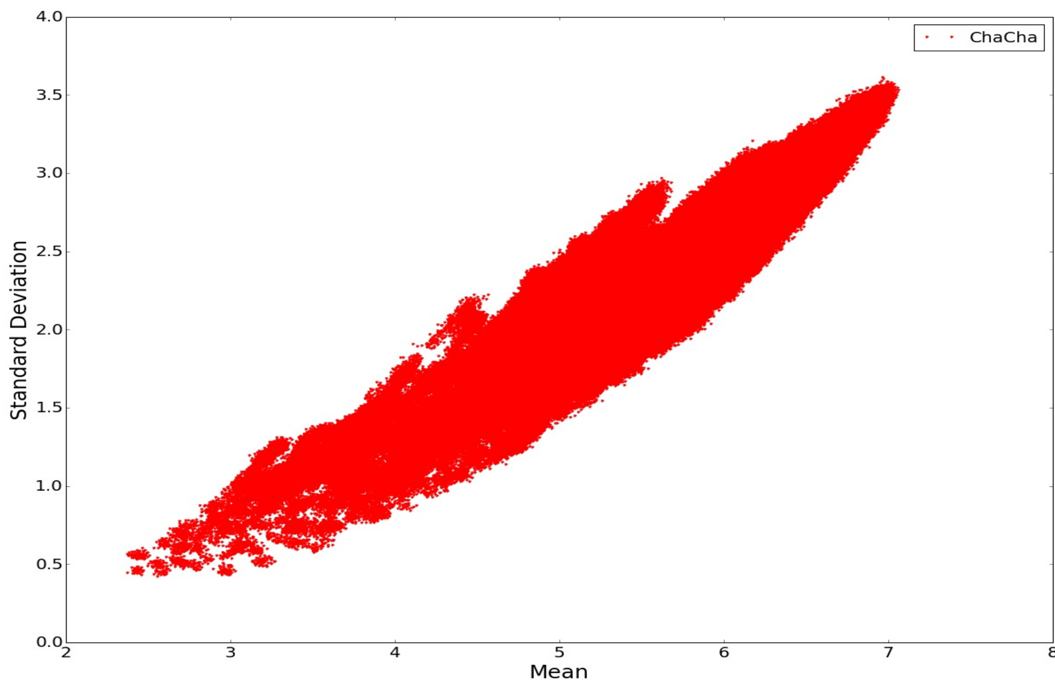


Figure 50. Mean and Standard Deviation of Diffusion Matrices of ChaCha's Quarter Round

ii. There are more than **58000** (5.5%) permutations *of i, j, k, and l;* that give better results than the prescribed value of rotation constants i.e. the mean is higher and

standard deviation is lower than the mean and standard deviation obtained with the prescribed rotation constants {16, 12, 8, 7}.

iii. The rotation constants that generate the diffusion matrix with the highest mean is **{14, 24, 19, 11}** i.e. if we use these rotation constants then we will have maximum diffusion. The mean and standard deviation of diffusion matrix at these rotation constants is **7.0599** and **3.5353** respectively.

iv. The rotation constants that generate diffusion matrix with the smallest standard deviation is **{5, 13, 6, 23}**. The mean and standard deviation for the diffusion matrix corresponding to these rotation constants are **6.6526** and **2.8851** respectively.

C) Results for Quarter Round of MCC

The graph for Quarter round of MCC core is given in Figure 51.

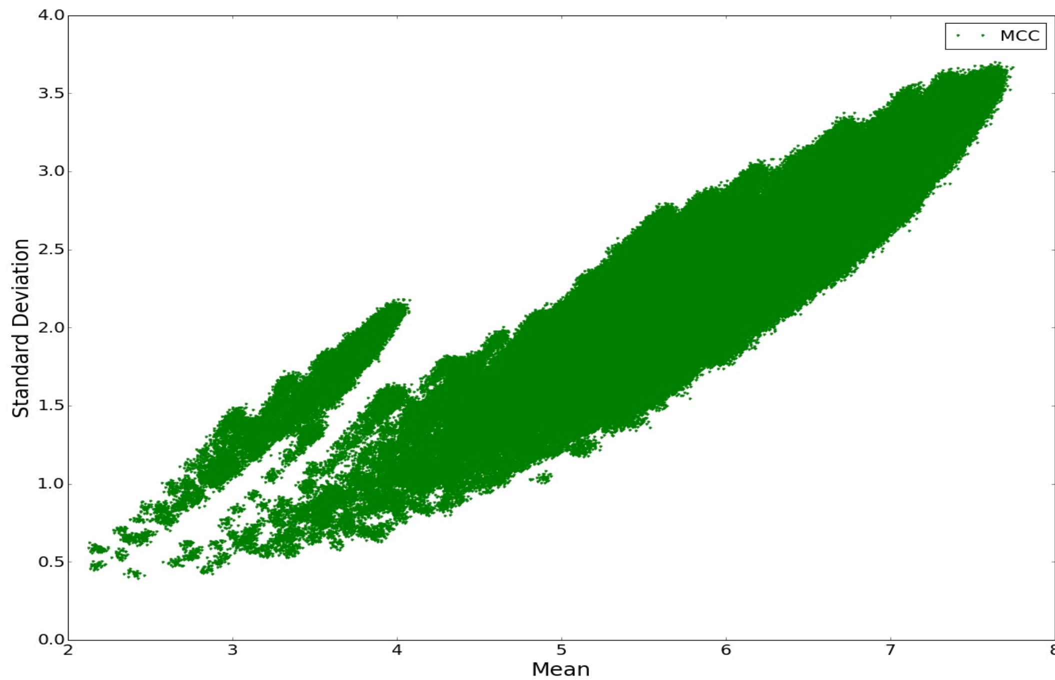


Figure 51. Mean and Standard Deviation of Diffusion Matrices of MCC's Quarter Round

Results of the experiment for MCC Quarter round yielded some important observations that led to the decision of **a) rotation constants to be used for MCC Quarter round** and **b) about the sequence of parameters to be passed in Column and Row round of MCC.** These are detailed below:

i. There are 12 sets of rotation constants that generate diffusion matrices with mean more than 7.70. These 12 top sets are listed in Table 12.

Table 12. Top 12 Sets of Rotation Constants with Mean ≥ 7.70

Sr. No.	**1st rotation constant (*i*)**	**2nd rotation constant (*j*)**	**3rd rotation constant (*k*)**	**4th rotation constant (*l*)**	**Mean of diffusion matrix**	**Std. dev. of diffusion matrix**
1.	20	17	24	9	7.743	3.669
2.	27	14	22	16	7.730	3.657
3.	4	16	8	19	7.728	3.607
4.	24	16	21	15	7.726	3.527
5.	**4**	**17**	**8**	**0**	**7.716**	**3.605**
6.	21	17	24	15	7.709	3.551
7.	4	17	9	19	7.708	3.652
8.	26	13	23	24	7.706	3.674
9.	4	17	8	16	7.705	3.637
10.	26	14	23	8	7.704	3.644
11.	19	23	15	1	7.704	3.587
12.	26	14	23	22	7.700	3.641

ii. Diffusion matrices of top 12 sets of rotation constants are quite similar to each other and any one of them can be picked for the best possible diffusion. However, set number 5 with rotation constants **{4, 17, 8, 0}** was picked for its specific property i.e. its 4th rotation constant is zero. Non-requirement of 4th rotation means reduction in execution cost as we have to do one operation lesser and still be able to generate the best possible diffusion.

iii. The diffusion matrix of MCC corresponding to rotation constants {4, 17, 8, 0} is given in Table 13.

Table 13. Diffusion Matrix of MCC's Quarter Round

IP \ OP	**a**	**b**	**c**	**D**
a	13.4	9.03	7.271	15.266
b	10.551	6.909	5.207	12.287
c	7.733	4.319	4.32	9.369
d	5.39	2.77	2.78	6.861

iv. The diffusion matrix shown in Table 13 diffuses output words 'a' and 'd' more than 'b' and 'c'. To be precise, word 'd' is diffused most and word 'c' is diffused the least. So in Column and Row rounds, parameters are passed in a different sequence to make sure that each row and each column have equal opportunities for diffusion. Figure 52 illustrates this concept. The element with highest diffusion is coloured red, the second in diffusion is coloured green, followed by the third and fourth in blue and yellow. For example, in Column round, the sequence of

parameters in first call to Quarter round is (x_0, x_4, x_8, x_{12}) and this sequence results in highest diffusion in x_{12} (element of fourth row, coloured red), followed by x_0 (the element of first row, coloured green) and then x_4 (element of second row, coloured blue) and least diffusion in x_8(element of third row, coloured yellow). So, in calling Quarter round for subsequent columns / rows, the intention is to create more diffusion in those elements which belong to rows that had lesser diffusion in previous calls. With this objective, sequence of parameters is decided in Column and Row round. It is evident from Figure 52 that after all four calls of Column round, each column and row has exactly one element that has been diffused maximum and similar behaviour is ascertained by different Row rounds. This sequence of parameters results in uniform diffusion.

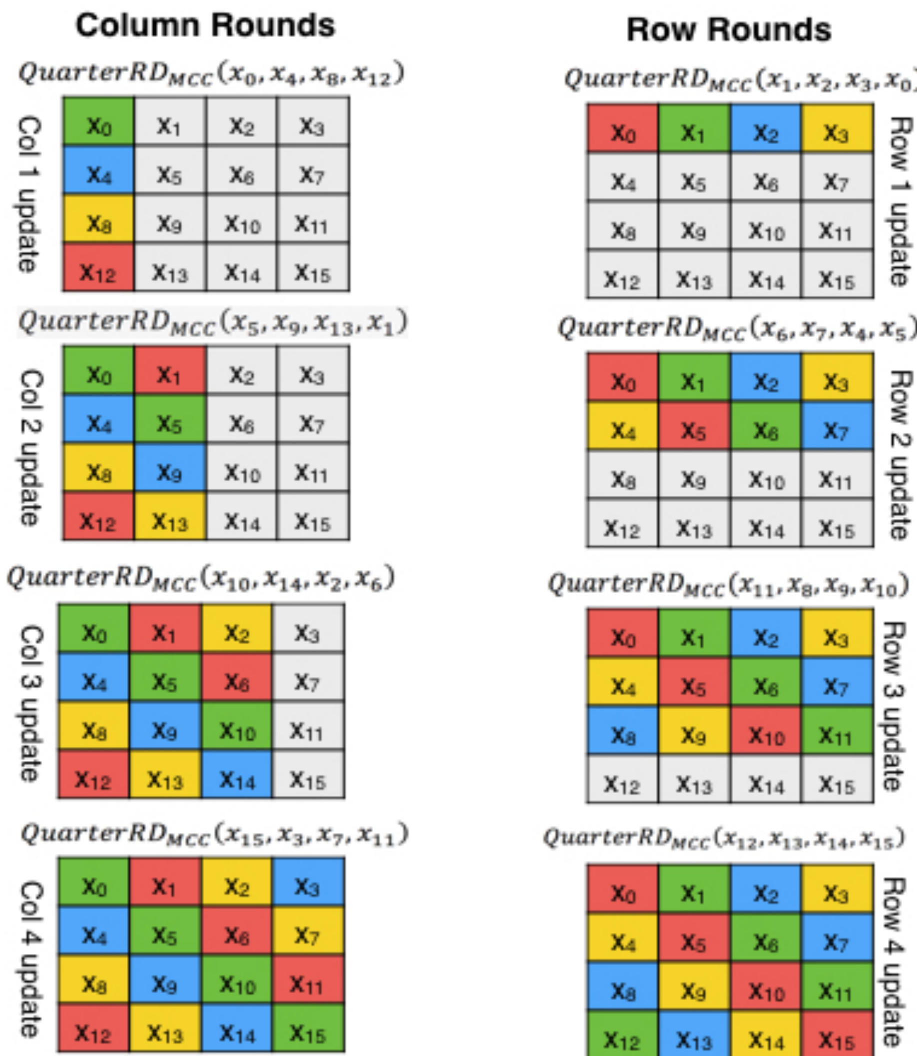


Figure 52. Sequence of Parameters Ensure Uniform Diffusion

D) Comparison of Quarter Round of Salsa Core, ChaCha Core, and MCC

The graph in Figure 53 gives better picture of relative performance of these three designs. Figure 54 is a zoomed version of this graph showing points having means ≥ 7.0. Important observations from result of all three algorithms are listed below:

i. **MCC's and ChaCha's Quarter round perform better than Salsa's Quarter round:** For majority of values of *i, j, k, and l;* Quarter round of ChaCha and MCC creates more diffusion than Salsa and the same is evident from Figure 53.

ii. **MCC's Quarter round outperforms ChaCha's Quarter round**: It is evident from Figure 53 and Figure 54 that Quarter round of MCC has better diffusion property than that of ChaCha. For considerably high number of permutations of *i, j, k and l*, MCC's Quarter round creates more diffusion than ChaCha's Quarter round. The experiment reflected that:

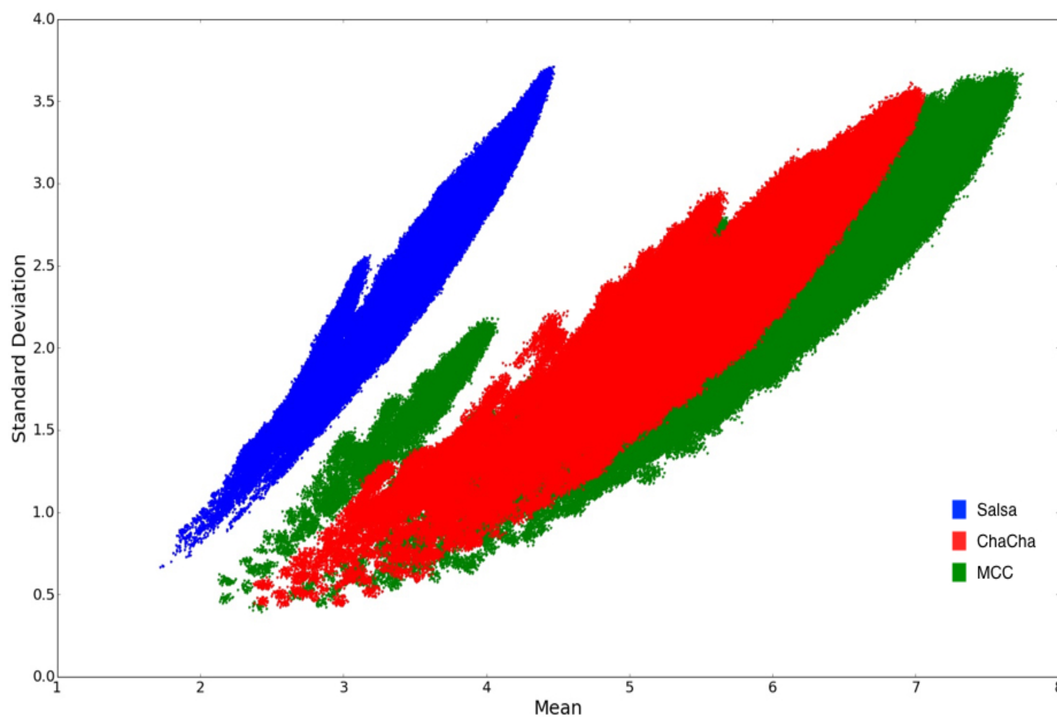


Figure 53. Comparison of Quarter Rounds of Salsa Core, ChaCha Core and MCC

- For at least **200 thousand permutations (about 21%)**, MCC's Quarter round have higher mean than the highest possible mean (7.0599) of ChaCha's Quarter round.

- Highest possible mean of ChaCha's diffusion matrix, was not achieved with permutation constants prescribed by Bernstein in [158]. If the prescribed permutation constants are considered, the resultant diffusion matrix has mean of 6.6424, and MCC's core gives at least **450 thousand permutations (about 44%)** that have higher mean than this.
- The similar trend is visible if we look at the standard deviation of diffusion matrices of both the designs. For more than **650 thousand permutations (about 64%)**, MCC Quarter round generates diffusion matrices, having standard deviation better (lesser) than the best possible diffusion matrix (having lowest standard deviation of 2.8851) of ChaCha's Quarter round. As discussed earlier, this diffusion matrix (with least standard deviation) is not observed at the rotation constants prescribed by the author (Bernstein in [158]). Using authors prescribed rotation constants, the diffusion matrix result in standard deviation of 3.2731. For about **90% permutations**, MCC's Quarter round results in diffusion matrix having lesser standard deviation than this.

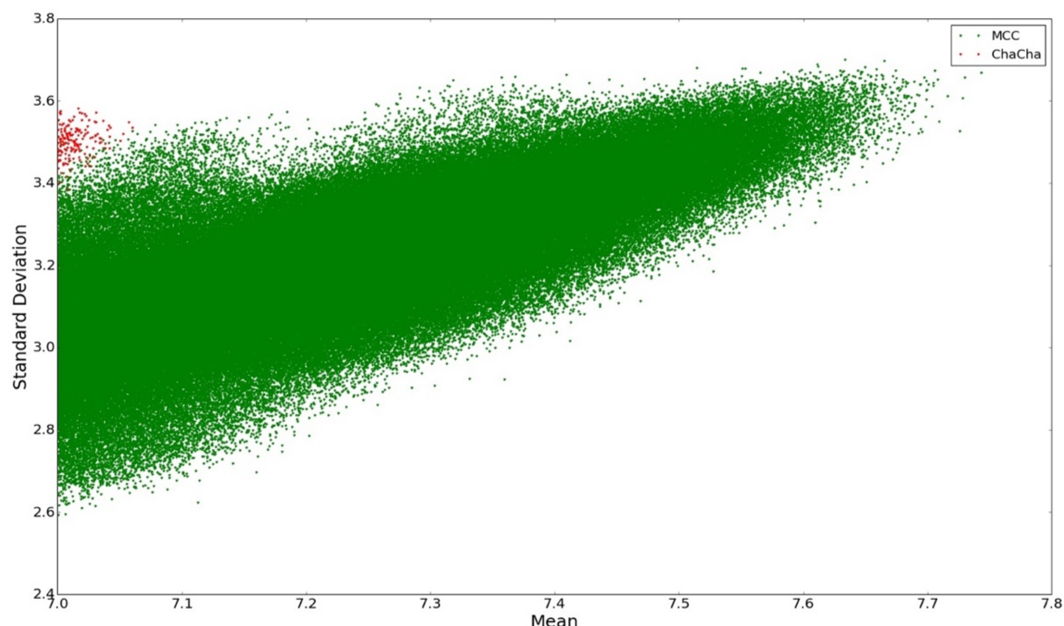


Figure 54. Zoomed Version of Comparison of Three Designs for Mean ≥ 7.0

- If we compare both mean and standard deviation, more than **340 thousand permutations (about 32%)** generate diffusion matrices, that have more mean as well as lesser standard deviation than the diffusion matrix generated by ChaCha's Quarter round at {16, 12, 8, 7} (rotation constants prescribed by the author).

- On average, more than 250 thousand permutations (about 22%) were found in MCC that produce diffusion matrices with mean greater than equal to 7.0 and only about 150 permutations (less than 0.02%) in ChaCha that produce mean greater than or equal 7.0. This is reflected in Figure 54.

Concluding Remarks about the Experiment: Analysis of Quarter round of Salsa and ChaCha core for all possible rotation distances reflect that there are considerably high number of alternative rotation distances that perform better than the rotation constants prescribed by Bernstein in [157] and [158]. Quarter round of MCC creates more diffusions than its counterpart and it does so in lesser operations. The Quarter round of MCC as described in Figure 48 is proposed with rotation constants of {4, 17, 8, 0} i.e. without fourth rotation. So the improved design uses four 32-bit additions, four 32-bit XORs and **three** 32-bit rotations against **four** rotations in ChaCha as well as Salsa's Quarter round. Even after having one operation lesser, MCC has more diffusion than ChaCha and Salsa. MCC's Quarter round, on an average, generate diffusion matrices with mean of 7.716 against 6.6424 of ChaCha and 4.0992 of Salsa i.e. on an average, gain of **16%** from ChaCha and **88%** from Salsa Quarter round. The MCC core, based on Quarter round as per Figure 48, calls multiple Double rounds and may be used to generate stream cipher, block ciphers or may be used to generate collision resistant compression function [160] for a cryptographic hash algorithm. Spritz [161] and Rumba [162] are examples of such an attempt where compression function or hash has been designed from basic structure used by stream cipher. This thesis has used MCC core for generating collision resistant compression function for cryptographic hash algorithm named ***Cocktail*.**

4.2 Introduction to *Cocktail* and Description of Notations and Operations Used

This chapter presents a new hash function named ***Cocktail*** which can act as a variant to Skein Hash family [9] . Besides Skein, it can be used in place of any existing hash function. ***Cocktail*** is *simple*, *flexible,* and *efficient* but still *secure*.

Cocktail is based on ARX i.e. Arithmetic, Rotation (with constants) and XOR operations and does not use any S Boxes lookups or Integer multiplications at all which makes ***Cocktail*** quite *simple* as well as *efficient*.

Cocktail is an iterated hash function and its compression function is based on **M**odified **C**haCha **C**ore (MCC). MCC is an alternative to Bernstein's Salsa [157] and ChaCha core [158]. Salsa was submitted in eSTREAM [17] project and is one of eSTREAM's profile algorithm. Its security has been intensively analysed. ChaCha and MCC are extensions of Salsa core and offer similar or higher level of security (because both offers more diffusion). Apart from the security aspect, MCC offers high level of parallelism that can be exploited for better performance. Designing compression function around MCC (which is an improvement over an existing primitive Salsa and ChaCha) instils great confidence in *security* and *performance* of ***Cocktail***.

Cocktail is a Wide Pipe iterated design [49] which is an improvement over Merkle Damgard construction [14] [45]. The internal hash state of ***Cocktail*** is at least double the size of the output (message digest) size. In fact, ***Cocktail*** also uses bits hashed so far in chain value and thus implements the feature of Biham and Dunkleman's HAIFA [51] construction also. Such wide pipe design, along with introduction of number of bits hashed so far, makes ***Cocktail*** more *secure* from multiple generic attacks like second pre-image attacks, internal collision, length extension and Joux Multicollisions.

Cocktail is *flexible* as it can output message digest of varying sizes. This study presents the two variants of ***Cocktail*** hash function: ***Cocktail-512*** and ***Cocktail-1024***. Akin to SHA-2, ***Cocktail*** has a 32-bit version ***(Cocktail-512)*** and a 64-bit version ***(Cocktail-1024).*** Table 14 illustrates ***Cocktail's*** usage.

*Table 14. **Cocktail's** Overview*

Algorithm to be used	**Word Size**	**Size of Message Block and Internal Hash State**	**Message Digest**	**Initial Values**	**Input Message Length**
Cocktail-512	32 bits	512	32 bits to 256 bits in steps of 32 bit each.	16 constants of 32 bit each	Less than 2^{64}
Cocktail-1024	64 bits	1024	320 bits to 512 bits in steps of 64 bit each.	16 constants of 64 bits each	Less than 2^{128}

4.2.1 Definitions, Symbols, Notations, Operations and Parameters Used

A) Terms

Bit	A binary digit having value 0 or 1
Byte	A group of 8 bits
Word	32 bits i.e. 4 bytes in case of ***Cocktail – 512*** and 64 bits i.e. 8 bytes in case of ***Cocktail – 1024***. Big Endian Notation has been used for converting bytes into words. For example, if Location 0, 1, 2, and 3 contains 0x12, 0x34, 0x56, and 0x78 respectively, then the 32-bit word will be 0x12345678 i.e. the byte stored at location 0 is considered as the most significant byte.

B) Symbols and Operations

$=$	Assignment operation. A variable assigned to other variable
$+$	Addition of two words. Addition will be modulo 2^{32} for ***Cocktail-512*** and module 2^{64} for ***Cocktail-1024***
$\oplus$	XOR Operation of two words
$\ggg \boldsymbol{r}$	Rotation of r bits towards right i.e. towards less significant bits
$\lll \boldsymbol{r}$	Rotation of r bits towards left i.e. towards more significant bits

C) Algorithm Parameters

$\boldsymbol{M}$	The Message to be hashed.
$\boldsymbol{L}$	Length of the message **M** in bits.
$\boldsymbol{Z}$	Number of zeroes appended to message during the padding step.
$\boldsymbol{l}$	Size of the Block or Internal Hash State in bits (512 for ***Cocktail*** -512 and 1024 for ***Cocktail***-1024).
$\boldsymbol{N}$	Total number of Message Blocks after padding.
$\boldsymbol{M^i}$	The $\boldsymbol{i^{th}}$ Message Block of size $\boldsymbol{l}$ bits. $\boldsymbol{M^0}$ represent the first Block and $\boldsymbol{M^{N-1}}$ **r**epresent the final message block.
$\boldsymbol{M_j^i}$	The $\boldsymbol{j^{th}}$ word of $\boldsymbol{i^{th}}$ Message Block. $\boldsymbol{M_0^i}$ represent the first and the left most word of message block $\boldsymbol{i}$.
$\boldsymbol{H^i}$	The $\boldsymbol{i^{th}}$ chain value (Internal Hash State) of size $\boldsymbol{l}$ bits. $\boldsymbol{H^0}$ represent the Initial Value also termed as **IV** (Initial Value) and $\boldsymbol{H^{N-1}}$ is the final state

value outputted from last compression function and is used to determine the message digest using output transformation.

$\boldsymbol{H_j^i}$ The $\boldsymbol{j^{th}}$ word of $\boldsymbol{i^{th}}$ internal hash state value.

$\boldsymbol{T_0^i}$ Lower order word representing number of bits hashed up to $\boldsymbol{i^{th}}$ message block (compression function)

$\boldsymbol{T_1^i}$ High order word representing number of bits hashed up to $\boldsymbol{i^{th}}$ message block (compression function)

$\boldsymbol{NR}$ Number of rounds used in each compression function.

$\boldsymbol{r}$ Used to signify the round number of compression function. $\boldsymbol{r}$ varies from 0 to $\boldsymbol{NR-1}$

$\boldsymbol{K^i}$ The Key Derivation Word (KDW) to be used in $\boldsymbol{i^{th}}$ compression function.

$\boldsymbol{K_j^i}$ The $\boldsymbol{j^{th}}$ word of KDW to be used in $\boldsymbol{i^{th}}$ compression function.

$\boldsymbol{SK_s^i}$ $\boldsymbol{s^{th}}$ Sub-Key used in r^{th}round of the i^{th}compression function, where $r = 2 * s$

$\boldsymbol{SK_{s,j}^i}$ The $\boldsymbol{j^{th}}$ word of $\boldsymbol{s^{th}}$Sub-Key of i^{th}compression function.

$\boldsymbol{H(M)}$ Message Digest i.e. Final Hash Value of Message $\boldsymbol{M}$

$\boldsymbol{n}$ Size of Message Digest required.

4.3 Iterated Construction of *Cocktail*

The whole Message $\boldsymbol{M}$ is padded and divided into $\boldsymbol{N}$ blocks of $\boldsymbol{l}$ bits each. An Initial Chain Value (IV) of $\boldsymbol{l}$ bits is also defined where $\boldsymbol{l}$ is of 512 bits for ***Cocktail-512*** and of 1024 bits for ***Cocktail-1024***. The compression function $\boldsymbol{f}$ takes three inputs: Message Block, Chain Value (Internal Hash State) and number of bits hashed until now (represented by $\boldsymbol{T}$ which will be of two words i.e 2 words of 32 bits each (totaling 64 bits) for ***Cocktail-512*** and 2 words of 64 bits (totalling 128 bits) each for ***Cocktail-1024***). Compression function generates $\boldsymbol{l}$ bits output. The iterated structure works as follow:

$$H^0 = IV$$

$$H^i = f\left(H^{i-1},\ M^{i-1},\ T^{i-1}\right) \oplus M^{i-1}$$

After the last block M^{N-1} is processed, the Final Hash value is computed using Output Transformation $\boldsymbol{O}$ as defined below:

$$H(M) = O\left(H^N\right)$$

$$O(x) = truncate_n(g(x) \oplus x)$$

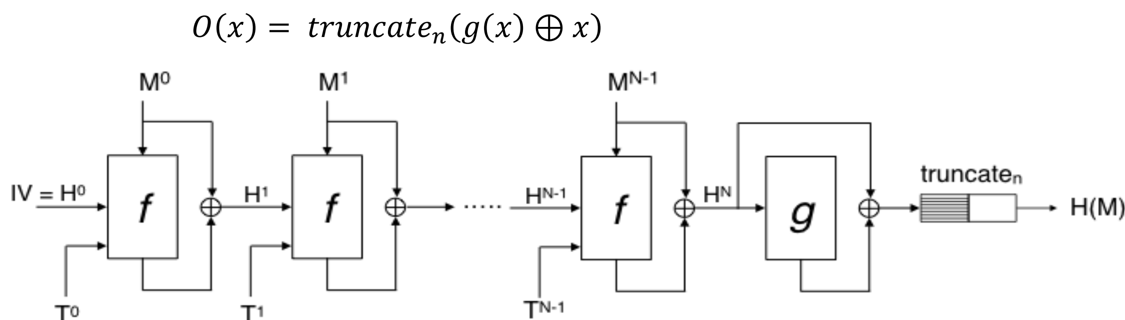


Figure 55. Iterative Structure of ***Cocktail*** *Hash Function*

The iterated structure of cocktail is a variant of Matyas – Mayer – Oseas [93] iteration mode and is represented graphically in Figure 55. The compression function $\boldsymbol{f}$ and output transformation $\boldsymbol{O}$ are detailed in subsequent sections.

4.4 Specifications of *Cocktail-512*

Hash function ***Cocktail-512*** operates on 32-bit words and can be used to generate message digest of any length from 32 bits to 256 bits in steps of 32 bit each (i.e. output of 32, 64, 96, 128, 160, 192, 224, and 256 bits are possible). Internal Hash State and Message Block Size is of 512 bits represented as 16 words of 32 bits each. This section illustrates Initial Values, padding technique used, Compression Function, and Output Transformation for ***Cocktail-512*** in detail.

4.4.1 Padding

Cocktail uses multi-rate padding. For a message M of length L, the following steps are adopted for padding in ***Cocktail-512***.

Step 1: Append Bit '1' to Message M.

Step 2: Append $Z = (-L - 66) \; mod \; 512$ zero ('0') bits i.e. enough zeroes so that message after insertion of bit 1 and zeroes is congruent to 447 $modulo$ 512.

Step 3: Append bit '0' followed by 64-bit representation of message length L.

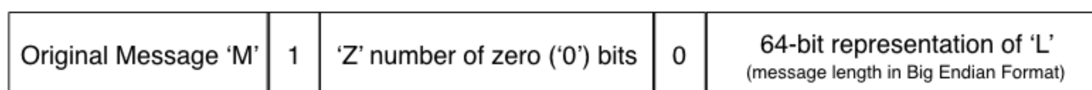


Figure 56. Message 'M' after Padding in ***Cocktail-512***

At least one bit and maximum of 512 bits are appended. After following the above steps, Message will always be multiple of 512 bits. The maximum length of the message that can be hashed with ***Cocktail-512*** is $(2^{64} - 1)$ bits.

4.4.2 Initial Values

For ***Cocktail-512***, 16 words of 32-bit each (i.e. 512 bits) are needed as IV (initial values). The 16 words of initial values are derived from first 16 prime numbers (i.e. 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53). Initial value is the fraction part of the square root of the corresponding prime numbers after converting them to binary and keeping only the first 32 bits. Appendix –V gives steps to generate these initial values.

H_0^0	=	6A09E667	H_1^0	=	BB67AE85
H_2^0	=	3C6EF372	H_3^0	=	A54FF53A
H_4^0	=	510E527F	H_5^0	=	9B05688C
H_6^0	=	1F83D9AB	H_7^0	=	5BE0CD19
H_8^0	=	CBBB9D5D	H_9^0	=	629A292A
H_{10}^0	=	9159015A	H_{11}^0	=	152FECD8
H_{12}^0	=	67332667	H_{13}^0	=	8EB44A87
H_{14}^0	=	DB0C2E0D	H_{15}^0	=	47B5481D

4.4.3 Compression Function of *Cocktail-512*

This section delineates the compression function of ***Cocktail-512*** wherein all details for Compression function called for i^{th} Message Block (M^i) is given. As evident from Figure 55, the compression function $\boldsymbol{f}$ of ***Cocktail-512*** has the following following inputs and outputs:

A) Inputs

- **Message Block** of 512 bits represented as 16 words of 32-bit each. For example the 16 words of message block $\boldsymbol{i}$ will be $M_0^i,\ M_1^i,\ \dots, M_{15}^i$.
- **Hash State** (Chain Value) of 512 bits represented as 16 words of 32-bit each. The 16 words are represented as $H_0^i,\ H_1^i,\ \dots, H_{15}^i$. For the first compression function, IV (Initial Values), as mentioned in '4.4.2 Initial Values', will be used and for subsequent compression functions, output of previous compression function acts as Hash State (Chain Value) input.
- Two words input representing the **message bits hashed so far** (excluding any padding bits). These words are represented as $T_0^i\ and\ T_1^i$. T_0^i is the lower order 32-bits and T_1^i is the high order 32-bits representing the number of bits hashed up to

i^{th} message block (without considering the padding data). Example: If the original message is of 330 bits, then after padding, Message will be of one block of 512 bits. In this case T_1^i will have zero value and T_0^i will represent 300 value.

B) Output

The compression function mixes the message block and hash state to give the hash state / chain value (H^{i+1}) as o/p which is used in next compression function $\boldsymbol{f}$ or output transformation O.

C) Internal State of Compression Function

Cocktail-512, for processing, arranges Internal State within the compression function and other date items (Message Block, Key etc.) in a matrix of 4x4 where each element of matrix is of 32 bits. The Internal state of Compression function is represented with the following matrix.

$$X = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{bmatrix}$$

Whenever Compression function is called, this Internal State matrix of compression function is initialized with the 16 words of Message Block for which compression function is called.

$$X = M^i$$

Various words of Message Block M^i may be represented using the following matrix:

$$M^i = \begin{bmatrix} M_0^i & M_1^i & M_2^i & M_3^i \\ M_4^i & M_5^i & M_6^i & M_7^i \\ M_8^i & M_9^i & M_{10}^i & M_{11}^i \\ M_{12}^i & M_{13}^i & M_{14}^i & M_{15}^i \end{bmatrix}$$

D) Rounds and Steps of Compression Function

Compression function of ***Cocktail-512*** (refer Figure 57) makes use of the following three steps:

- Add Sub-Key (Added in Alternate rounds)
- Four Column Quarter Rounds
- Four Row Quarter Rounds

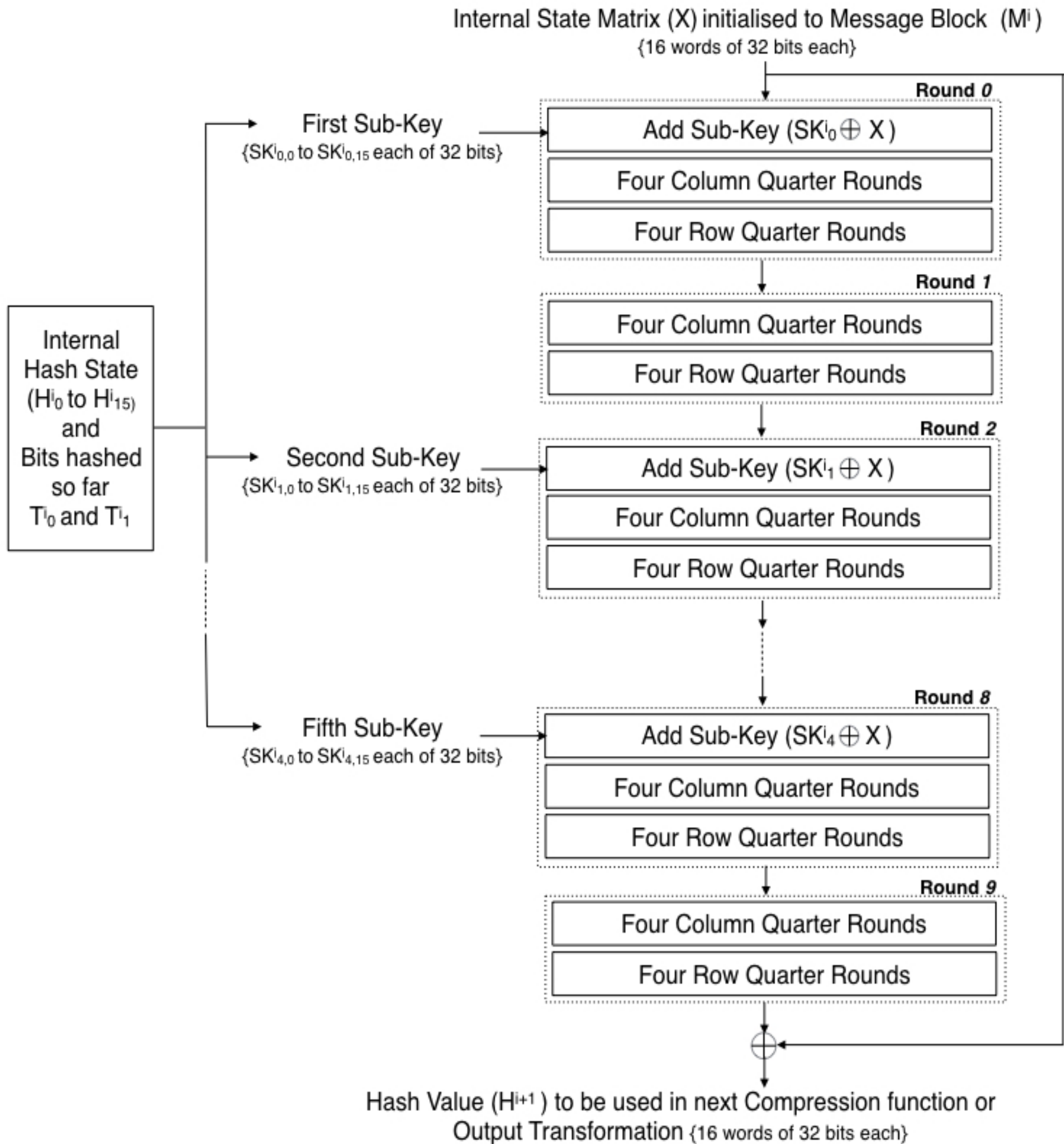


*Figure 57. Structure of 10 Round Compression Function **Cocktail***

The number of rounds (NR) is flexibility in the hands of the user and can be decided depending on the full diffusions required and the speed expectations. 10 rounds are proposed for ***Cocktail-512*** and the structure of the same is represented in Figure 57. Before explaining the steps to be followed in each round, key expansion method used to obtain Sub-Keys for alternate rounds of compression function, are explained.

E) Sub-Keys and Its Expansion

Each alternate round of the compression function makes use of 16 words of key. So for all NR rounds ($0\ to\ NR-1$) of compression function we need $16 * (NR/2)$ words

of sub-key (for 10 rounds, it will be 80 words). All these sub-key words are derived from Internal Hash State /chain value $(H_0^i, H_1^i, \ldots, H_{15}^i)$ and two words (T_0^i, T_1^i) representing number of bits hashed so far.

Key words to be used in s^{th} sub-key of ith compression function is termed as SK_s^i and is represented as:

$$SK_s^i = \begin{bmatrix} SK_{s,0}^i & SK_{s,1}^i & SK_{s,2}^i & SK_{s,3}^i \\ SK_{s,4}^i & SK_{s,5}^i & SK_{s,6}^i & SK_{s,7}^i \\ SK_{s,8}^i & SK_{s,9}^i & SK_{s,10}^i & SK_{s,11}^i \\ SK_{s,12}^i & SK_{s,13}^i & SK_{s,14}^i & SK_{s,15}^i \end{bmatrix}$$

s will vary from $0\ to\ (\frac{NR}{2} - 1)$ and s^{th} sub-key will be used in r^{th} round of compression function where $r = 2 * \text{s}$. For example first round $(r = 0)$ of compression function will make use of first sub-key $(s = 0)$ and then third $(r = 2)$ round of compression function will make use of second sub-key $(s = 1)$and so on.

To understand the process of obtaining sub-keys (SK_s^i) for alternate rounds of compression function, we define Key Derivation Words (KDWs) $K_0^i\ to\ K_{\left(16*\frac{NR}{2}-1\right)}^i$ (NR is total number of rounds) and these KDWs will be used to obtain round keys SK_s^i.

First 16 Key Derivation Words (KDWs) i.e. $K_0^i\ to\ K_{15}^i$ are obtained using following formulae

$$K_j^i = H_j^i, \qquad for\ j = 0\ to\ 15$$

The remaining KDWs i.e. $K_{16}^i\ to\ K_{(16*\frac{NR}{2}-1)}^i$ are obtained using the following formulae. The technique to obtain remaining KDWs is inspired from key expansion technique used by AES [5].

$$K_j^i = \begin{cases} W \oplus K_{j-16}^i, & for\ j\ \%\ 16 = 0 \\ K_{j-1}^i \oplus K_{j-16}^i, & Otherwise \end{cases}$$

where $W = (K_{j-1}^i \lll 8) \oplus RCon32(Rd\ \%\ 8)\ \ \{and\ Rd = j/16\}$

$RCon32(Rd)$ is a round constant dependent on sub-key number for which KDWs are being calculated and is given in Table 15.

*Table 15. Round Constants for Key Derivation Words(**Cocktail-512**)*

	$\boldsymbol{RCon32(Rd)}$		$\boldsymbol{RCon32(Rd)}$
$Rd = 1$	0x10000000	$Rd = 2$	0x20000000

$Rd = 3$	0x40000000	$Rd = 4$	0x80000000
$Rd = 5$	0x1B000000	$Rd = 6$	0x36000000
$Rd = 7$	0x6C000000	$Rd = 8$	0xD8000000

Figure 58 represents the process of computing KDWs.

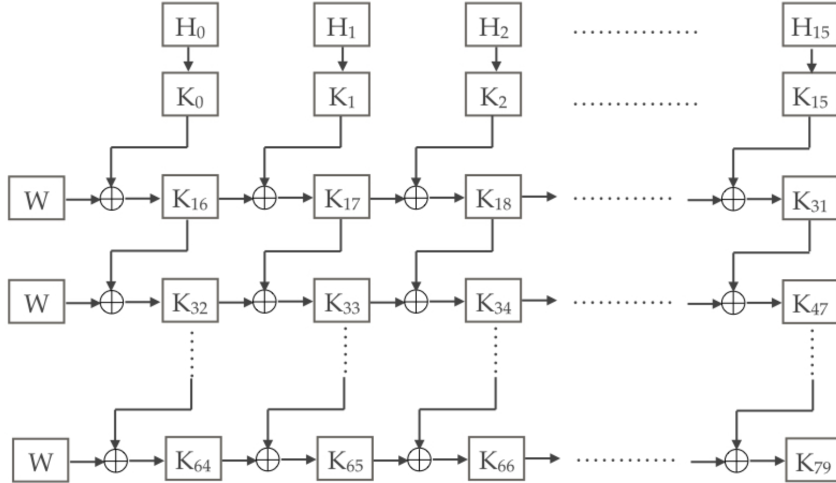


Figure 58. Process of Computing Key Derivation Words

Once the KDWs, have been computed, the round keys SK_s^i are computed as per following details:

First Sub-Key i.e. Sub-Key for first Round : SK_0^i : First sub-key i.e. sub-key for for the first round ($s = r = 0$) is defined as:

$$SK_{0,j}^i = \begin{cases} K_j^i + T_0^i, & for\ j = 0, 15 \\ K_j^i + T_1^i, & for\ j = 5 \\ K_j^i + T_2^i, & for\ j = 10 \\ K_j^i, & for\ j = 1\ to\ 4, 6\ to\ 9, 11\ to\ 14 \end{cases}$$

where $T_2^i = T_0^i \oplus T_1^i$

The same can be represented in matrix shape as below:

$$\begin{bmatrix} SK_{0,0}^i & SK_{0,1}^i & SK_{0,2}^i & SK_{0,3}^i \\ SK_{0,4}^i & SK_{0,5}^i & SK_{0,6}^i & SK_{0,7}^i \\ SK_{0,8}^i & SK_{0,9}^i & SK_{0,10}^i & SK_{0,11}^i \\ SK_{0,12}^i & SK_{0,13}^i & SK_{0,14}^i & SK_{0,15}^i \end{bmatrix} = \begin{bmatrix} K_0^i + T_0^i & K_1^i & K_2^i & K_3^i \\ K_4^i & K_5^i + T_1^i & K_6^i & K_7^i \\ K_8^i & K_9^i & K_{10}^i + T_2^i & K_{11}^i \\ K_{12}^i & K_{13}^i & K_{14}^i & K_{15}^i + T_0^i \end{bmatrix}$$

From above equation it is evident that ***T*** words (no. of bits hashed so far) affect each row and column of the Sub-key matrix.

Sub-key for Subsequent Rounds : SK_s^i : The sub-key for subsequent rounds ($s = 1\ to \frac{NR}{2} - 1$) is defined as:

$$SK_{s,j}^i = \begin{cases} K_{u+j}^i + T_{(s+0)\%3}^i, & for\ j = 0, 15 \\ K_{u+j}^i + T_{(s+1)\%3}^i, & for\ j = 5 \\ K_{u+j}^i + T_{(s+2)\%3}^i, & for\ j = 10 \\ K_{u+j}^i, & for\ j = 1\ to\ 4, 6\ to\ 9, 11\ to\ 14 \end{cases}$$

where $u = 16 * s$ and $s \in (1, 2, \ldots, \frac{NR}{2} - 1)$

The 16 words of sth sub-key used in r^{th} round (where $r = 2 * s$) may be represented as following matrix of 4x4.

$$SK_s^i = \begin{bmatrix} SK_{s,0}^i & SK_{s,1}^i & SK_{s,2}^i & SK_{s,3}^i \\ SK_{s,4}^i & SK_{s,5}^i & SK_{s,6}^i & SK_{s,7}^i \\ SK_{s,8}^i & SK_{s,9}^i & SK_{s,10}^i & SK_{s,11}^i \\ SK_{s,12}^i & SK_{s,13}^i & SK_{s,14}^i & SK_{s,15}^i \end{bmatrix}$$

$$= \begin{bmatrix} K_u^i + T_{(s+0)\%3}^i & K_{u+1}^i & K_{u+2}^i & K_{u+3}^i \\ K_{u+4}^i & K_{u+5}^i + T_{(s+1)\%3}^i & K_{u+6}^i & K_{u+7}^i \\ K_{u+8}^i & K_{u+9}^i & K_{u+10}^i + T_{(s+2)\%3}^i & K_{u+11}^i \\ K_{u+12}^i & K_{u+13}^i & K_{u+14}^i & K_{u+15} + T_{(s+0)\%3}^i \end{bmatrix}$$

where $u = 16 * s$ and $s \in (1, 2, \ldots, \frac{NR}{2} - 1)$

The key expansion method as explained above ensures that each row and column of all Sub-key matrices get some impact of no. of bits hashed so far (i.e. $T_0^i\ or\ T_1^i\ words$).

F) Add Sub-Key

'Add Sub-Key' step is not used in each round of the compression function. 'Add Sub-Key' step is called in alternate round i.e. in 0th, 2nd, 4th, 6th and 8th round of a 10 round compression function. 'Add Sub-Key' step, XOR the internal state matrix of Compression function with the 16 words of Sub-Key. The operation can be represented as: $X = X \oplus SK_s^i$

G) Column and Row Quarter Rounds

Four Column and Four Row Quarter Rounds are practically one Double round of **M**odified **C**haCha **C**ore (as defined in '4.1.2 Modified ChaCha Core (MCC)'). The Figure 59 represent the block diagram of one set of Four Column and Four Row Quarter

rounds. Column Quarter round call four Quarter rounds one for each column of the internal state and Row round does the similar operation by calling four Quarter rounds again but for each row of the internal state.

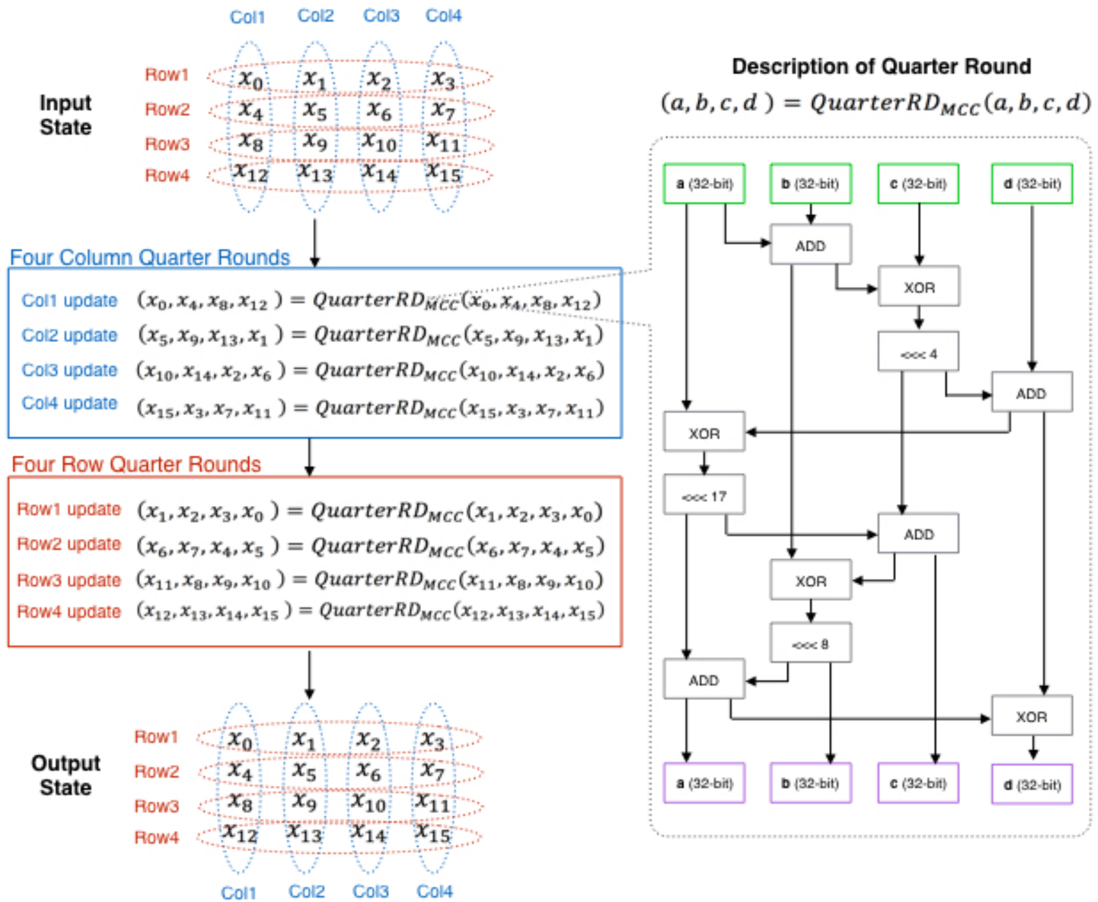


Figure 59. Working of Column and Row Round of ***Cocktail-512***

The logic behind sequence of parameters (elements of internal state) passed in each call to Quarter round was detailed in previous section under the head '4.1.4-C) Results for Quarter Round of MCC'. The specific sequence of parameters ensures uniform diffusion. In Table 16, we present the diffusion matrix generated by one set of Column and Row Quarter rounds (4 Column and 4 Row Quarter rounds). The values are rounded to closest integer. Majority of elements of internal state matrix gets diffusion close to ideal. The ideal value is 16 (i.e. out of 32 bits, 50% bits are modified by only one-bit change in the input).

As evident from the Table 16, just one set of Column and Row Quarter rounds, results in about 80% (79.6% to be precise) elements (words of internal state) getting diffusion close to 16 (above 10) and about 40% of these elements get ideal diffusion (diffusion of

16 bits). All this takes place in just one round of ***Cocktail*** and ***Cocktail*** has10 such rounds and 5 out of these will also have influx of 16 words of sub-key which will help in bringing confusion.

Table 16. Diffusion Matrix for One Double Round of MCC (Four Column and Four Row Quarter Rounds)

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
x_0	16	16	14	14	11	16	16	11	13	11	16	15	16	16	16	16
x_1	16	16	16	15	6	11	9	6	6	6	12	10	12	8	6	13
x_2	15	14	11	7	15	16	16	15	9	9	13	12	12	7	7	13
x_3	16	15	12	12	8	15	15	12	16	16	16	16	14	12	12	16
x_4	16	14	12	12	12	16	15	12	12	8	15	15	16	16	16	16
x_5	16	16	16	16	14	16	16	14	11	11	16	16	15	12	10	16
x_6	13	12	8	6	15	16	16	16	6	6	11	9	10	6	6	12
x_7	13	12	7	7	7	14	14	11	15	15	16	16	12	9	9	14
x_8	14	12	9	9	7	13	12	7	11	7	14	14	16	16	15	16
x_9	16	16	16	16	12	16	14	12	12	12	16	15	15	12	8	15
x_{10}	16	15	13	11	16	16	16	16	14	14	16	16	16	11	11	16
x_{11}	12	10	6	6	6	13	12	8	16	15	16	16	9	6	6	11
x_{12}	11	9	6	6	6	12	10	6	8	6	13	12	16	16	15	16
x_{13}	16	16	16	15	9	13	12	9	7	7	13	12	14	11	7	15
x_{14}	15	15	12	7	16	16	16	16	12	12	15	14	15	12	12	16
x_{15}	16	16	11	11	11	16	15	13	16	16	16	16	16	14	14	16

The algorithm to generate diffusion matrix is similar to the one discussed in '4.1.3 Experiment Used to Measure the Diffusion Property of Quarter Rounds'. Instead of four words (a, b, c, d), difference in sixteen words are analysed. Also in Step 4 and Step 5 of the prescribed algorithm; rather than calling $QuarterRD_{MCC}(a, b, c, d)$, $DoubleRD_{MCC}(x_0, x_1, .., x_{15})$ is called and accordingly diffusion matrix is generated for analysis.

H) Final XOR with Message Block

NR number of rounds (proposed $NR = 10$) of compression functions call Add Sub-Key ($NR/2$) times, Column and Row Rounds NR times, and updates the internal State Matrix X at each step and round. After NR rounds, the outcome of the compression

function (Hash State / Chain value for next compression function/output transformation) is obtained by using the following XOR operation.

$$H^{i+1} = X \oplus M^i$$

4.4.4 Output Transformation of *Cocktail-512*

The Output Transformation of ***Cocktail-512***, as defined below, is used to obtain the message digest from the final hash state / chain value (H^N).

$$H(M) = O(H^N)$$

$$O(x) = truncate_n(g(x) \oplus x)$$

$truncate_n(x)$ operation discards all but the trailing n bits of x. $g(x)$ calls one set of Column and Row Quarter rounds i.e. four Column and four Row Quarter rounds. The Output Transformation is illustrated in Figure 60.

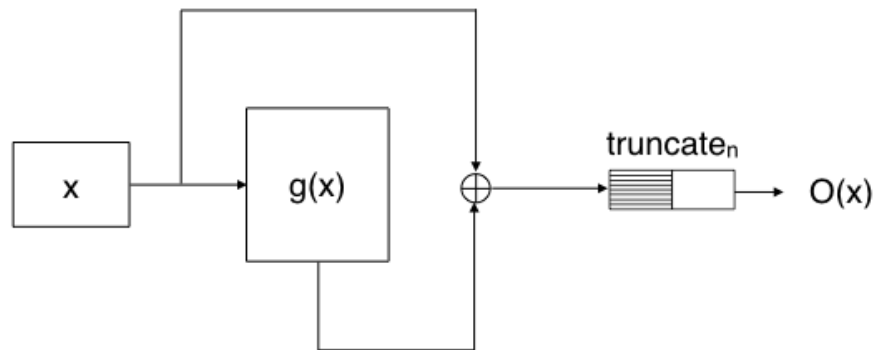


*Figure 60. The Output Transformation (O) of **Cocktail***

Output transformation helps in achieving variable size hash as described in Table 14.

4.5 Specifications of *Cocktail-1024*

Cocktail-1024 operates on 64-bit words and can be used to generate message digest of more than 256 bits and less than or equal to 512 bits. The possible sizes of hash values are: 320, 384, 448, and 512 bits. In ***Cocktail-1024***, Internal Hash State and Message Block is of 1024 bits represented as 16 words of 64 bits each. Specifications of ***Cocktail-1024*** are similar to ***Cocktail-512*** except the changes in size of word, initial values, padding technique, number of rounds, round constants for expanding keys and rotation constants used in column and row quarter round. All these changes are discussed here in this section.

4.5.1 Padding

As detailed earlier, ***Cocktail*** makes use of multi-rate padding. Padding in ***Cocktail-1024*** is almost similar to the one used in ***Cocktail-512***. For a message M of length L, we follow the following steps for padding in ***Cocktail-1024***:

Step 1: Append Bit '1' to Message 'M'

Step 2: Append $Z = (-L - 130) \; mod \; 1024$ zero ('0') bits i.e. enough zeroes so that message after insertion of bit 1 and zeroes is congruent to 895 modulo 1024.

Step 3: Append bit '0' followed by 128-bit representation of message length 'L'.

At least one bit and maximum of 1024 bits are appended. After following the above steps, Message will always be multiple of 1024 bits.

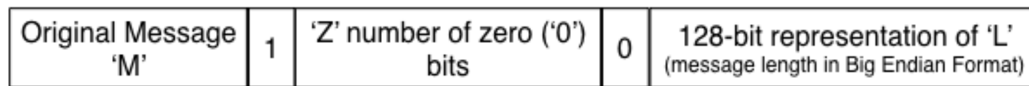


*Figure 61. Message 'M' after Padding in **Cocktail-1024***

The maximum length of the message that can be hashed with ***Cocktail-1024*** is $(2^{128} - 1)$ bits.

4.5.2 Initial Values

Cocktail-1024 also requires16 words of 64-bit each as Initial value. Like ***Cocktail-512***, the 16 words of initial values for ***Cocktail-1024*** are also derived from first 16 prime numbers (i.e. 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53). Initial value is the fraction part of the square root of the corresponding prime numbers after converting them to binary and keeping only the first 64 bits. The details are available in Appendix-V.

H_0^0 = 6A09E667F3BCC908 H_1^0 = BB67AE8584CAA73B

H_2^0 = 3C6EF372FE94F82B H_3^0 = A54FF53A5F1D36F1

H_4^0 = 510E527FADE682D1 H_5^0 = 9B05688C2B3E6C1F

H_6^0 = 1F83D9ABFB41BD6B H_7^0 = 5BE0CD19137E2179

H_8^0 = CBBB9D5DC1059ED8 H_9^0 = 629A292A367CD507

H_{10}^0 = 9159015A3070DD17 H_{11}^0 = 152FECD8F70E5939

H_{12}^0 = 67332667FFC00B31 H_{13}^0 = 8EB44A8768581511

H_{14}^0 = DB0C2E0D64F98FA7 H_{15}^0 = 47B5481DBEFA4FA4

4.5.3 Compression Function and Output Transformation of *Cocktail-1024*

The Input, Output, Internal state of compression function, Add Round Key, Column and Row Quarter rounds, Key expansion, and Output transformation for ***Cocktail-1024*** is same as that of ***Cocktail-512*** except the following changes:

i. All 32-bit words are replaced by 64-bit words. For example, all 16 words of Message Block M^i are 64-bit each making message block size equal to 1024 bits. Similarly, Key words, Internal state words, all are 64-bits wide.

ii. There is a minor change in the process of calculating KDWs. First 16 Key Derivation Words (KDWs) i.e. $K_0^i \; to \; K_{15}^i$ are obtained in the same fashion as in case of ***Cocktail-512*** using the following formulae:

$$K_j^i = H_j^i \,, \qquad for \; j = 0 \; to \; 15$$

For calculating the remaining KDWs i.e. $K_{16}^i \; to \; K_{(16*\frac{NR}{2}-1)}^i$, the process for calculating temporary variable W is changed a bit and the same is detailed below:

$$K_j^i = \begin{cases} W \oplus K_{j-16}^i, & for \; j \,\%\, 16 = 0 \\ K_{j-1}^i \oplus K_{j-16}^i, & Otherwise \end{cases}$$

where $W = (K_{j-1}^i \lll 16) \oplus RCon64\,(Rd \,\%\, 8) \;\; \{and \; Rd = j/16\}$

$RCon64(Rd)$ is a round constant dependent on rounds for which KDWs are being calculated and is given in Table 17.

*Table 17. Round Constants for Key Derivation Words (**Cocktail-1024**)*

	$\boldsymbol{RCon64\,(Rd)}$		$\boldsymbol{RCon64(Rd)}$
$Rd = 1$	0x1020000000000000	$Rd = 2$	0x2040000000000000
$Rd = 3$	0x4080000000000000	$Rd = 4$	0x801B000000000000
$Rd = 5$	0x1B36000000000000	$Rd = 6$	0x366C000000000000
$Rd = 7$	0x6CD8000000000000	$Rd = 8$	0xD801000000000000

iii. Number of rounds proposed for ***Cocktail-1024*** are 12. However, it is a tuneable parameter in the hands of user.

iv. Rotation constants used in Quarter round are changed from {4, 17, 8, 0} to {52, 41, 16, 0} for ***Cocktail-1024***. The rationale for this is given in subsequent section ‘4.7.6 Use of Modified ChaCha Core and Choice of Rotation Distances’.

4.6 Complexity of *Cocktail*

The number of operations used in ***Cocktail*** and its memory requirements are detailed here in this section. The calculations shown are of ***Cocktail-512,*** and those of ***Cocktail-1024*** may be computed accordingly.

4.6.1 No. of Operations Used

KDWs and Sub-Key calculation: The first 16 KDWs are same as Internal Hash State and calculation of remaining KDWs ($K_{16}^{i}\ to\ K_{(16*\frac{NR}{2}-1)}^{i}$) require 2 XOR and 1 Rotation operation if word is multiple of 16 and require only 1 XOR operation for all other words. Once KDWs are calculated, the calculation of sub-key words for each alternate round require only 4 Addition operations. So for 10-round compression function, 5 sub-keys are required and key expansion will require the following operations.

$$Operations\ by\ KDWs\ = 4 * (2\ XORs + 1\ Rotaion) +\ 4 * 15 * (1\ XOR)$$

$Opertions\ to\ compute\ Sub - Key = 5 *\ (4\ Additions) + 1\ XOR$ (for calculating T_2^i)

Add Sub-Key uses 16 XOR operations, one for each word of Internal State matrix. However, Add Sub-key is called on alternate rounds. So for total 10 rounds, Add Sub-Key is called 5 times. So total operations become $16 * \frac{10}{2} =\ 80\ XOR$ operations.

Column and Row Quarter Rounds in Compression Function: Each Quarter round makes use of 4 Additions, 4 XORs, and 3 Rotations. Thus 4 Column and 4 Row Quarter rounds make $8\ *\ (4\,Additions\ +\ 4\,XORs\ +\ 3\,Rotation)$ and for total of 10 rounds it becomes $10\ *\ 8\ *\ (4\,Additions\ +\ 4\,XORs\ +\ 3\,Rotation)$.

Final XOR with Message Block: Final XOR with message block involves $16\,XORs$.

Output Transformation: Output transformation consists of one set of Column and Row Quarter Rounds {that requires $8 * (4\,Additions\ +\ 4\,XORs\ +\ 3\,Rotation)$}, followed by XOR with chain value {that requires $16\,XORs$}, and finally computation of desired number of hash words {which involves $16\,Additions$}. So Output Transformation in total requires $48\,Additions\ +\ 48\,XORs\ +\ 24\,Rotation.$

Total Operations: So a 10 round *Cocktail-512* requires: $\mathbf{388}, 32 - bit\ additions$; $\mathbf{533}, 32 - bit\ XORs\ and\ \mathbf{268}\ Rotations$ summing up **1189 operations**. The overhead for initialization the hash structure, padding the message and

implementing the logic of algorithm is omitted here as their cost is negligible compared to that of compression function and output transformation.

4.6.2 Memory Requirement

Cocktail-512 requires 64 bytes and 32 bytes to store Initial Values and Round Constants (used for key expansion) respectively. In RAM, ***Cocktail-512*** requires memory for storing Internal State of compression function X, Message Block M, Sub-Keys SK, Hash State (Chain Value) H, T words storing message bits hashed so far, and few other integer / character variables for implementing logic of algorithm. The memory requirement for these data items is mentioned below:

$$Hash\ State\ H\ and\ Internal\ state\ X\ of\ Compression\ Fn:\ \mathbf{64\ bytes\ each}\ (\mathbf{i.e.\ 128\ bytes})$$

$$Mesage\ Block\ \mathbf{M}\ requires:\ \mathbf{64\ bytes}$$

$$Sub-Keys\ \mathbf{SK}\ requires:\ 64 * \frac{NR}{2}\ bytes\ i.e.\ \mathbf{320\ bytes}\ for\ 10\ rounds$$

$$\mathbf{T}\ containing\ message\ bits\ hashed\ so\ far\ requires:\ \mathbf{8\ bytes}$$

The memory required for sub-keys can be reduced to **70 bytes only** by computing it dynamically for each round rather than pre-computing in advance for all rounds. So **total memory requirement** for ***Cocktail-512*** is **96 bytes (for Constants) and 270 bytes for other data items in RAM**. In addition to this, about 50 – 100 bytes may be required for implementation of logic.

4.7 Design Philosophy, Design Decisions, and Its Rationale

4.7.1 Simplicity

The basic guiding principle has been to keep the design as simple as possible. Simplicity ensures that design is easy to understand and in turn easy to analyse. The easy analysis instils more confidence in the design. ***Cocktail*** uses simple operations **A**rithmetic, **R**otation and **X**OR) on words of 32 bit / 64 bits and makes use of long chain of these three simple operations to add confusion and diffusion.

4.7.2 ARX Based Design

Cocktail is based on **ARX** design and uses three simple operations: 32-bit **A**ddition, 32-bit-**X**OR, and constant distance 32-bit **R**otations. Addition and XOR break linearity and rotation diffuses the changes from high bits to lower bits. The decision to build

Cocktail around ARX operations is well thought of and is based on a lot of literature survey.

ARX based cryptographic primitives have been well analysed by the cryptographic community. The concept of ARX is quite old and even legacy cryptographic primitive like FEAL [163] used ARX based design way back in 1987. A lot of recent cryptographic primitives have also been designed around ARX operations. Examples are:

- **Lightweight Block Cipher**: Speck [164] recently launched by NSA (National Security Agency, USA) in June 2013.
- **Block Cipher** : RC5 [165], Threefish [9]
- **Stream Ciphers**: Salsa20 [157], ChaCha [158], HC-128 [166]
- **Hash Functions:** SHA-3 finalists - Blake [26], Skein [9] and many other SHA-3 first and second round candidate algorithms like Blue Midnight Wish, CubeHash, EDON-R etc.

Some other quite famous cryptographic primitives had their designs close to ARX but not exactly ARX. Examples are TEA [167], XTEA [168], XXTEA [169] block ciphers and all these three, make use of addition, XOR and left / right shift operations in place of rotations.

Mouha in [170] has given a detailed analysis of ARX based designs. The major advantage of using ARX based design is that they are relatively fast on PCs and cheap in hardware and software. Its implementation is quite compact and generally results in easy algorithm. ARX designs are also immune to timing attacks as they run in constant time. ARX designs are functionally complete and using these three simple operations one can simulate any circuit. Using these three operations, one can achieve security level achievable with any other set of operations.

Bernstein in [157] has given a lot of arguments in favour of using these simple operations over Integer multiplication and S-Box Substitution that are commonly used in design of various cryptographic primitives. These choices are discussed one by one hereunder.

A) Integer multiplication

Integer multiplication $modulo$ 2^{64} or 2^{32} is quite an efficient operation on some CPUs. Example of Cryptographic primitives that have used integer multiplication includes

IDEA [171], MESH [172]. In integer multiplication, output has a complicated relationship with inputs and thus mixes the words considerably well. In fact, multiplication mixes words more thoroughly than a few simple integer operations. However, integer multiplication suffers from following drawbacks:

a) Integer multiplication will generally require several cycles even on faster CPUs and on an average CPU, cycle requirement will be even more. Undoubtedly integer multiplication is becoming faster but is still not consistent across CPUs. Contrary to this, long chain of simple operations (ARX) are quite efficient and achieve desired mixing.
b) Secondly, integer multiplication may result in timing attacks. Bernstein [157] has given example of Motorola PowerPC 7450 (G4e) and 8641D where integer multiplication takes different cycles depending on second operands bit pattern. Such a timing leak may be exploited for timing attacks.

B) S-Boxes

S-Boxes have also been extensively used in design of cryptographic primitives. DES [88], AES [5], Whirlpool [8], Grøstl [24], Blowfish [173], and Twofish [174] are examples of cryptographic primitives based on S-Boxes. S-Box (Substitution Box) involves substitution of input bits with output bits. Input bits acts as an index and are used to lookup an S-Box (Table) and looked up value replaces the input bits. AES uses an S-Box consisting of 16 rows and 16 columns and each cell contains one byte. The input byte is used to locate the byte from S-Box that will substitute the input byte. Undoubtedly S-Box creates great diffusion/confusion and mingles input quite exhaustively which is difficult to do with simple integer operations. However, it suffers from the following drawbacks:

a) The size of S-Box can be detrimental. For example, in AES the size of S-Box will be around 2048 bytes and using this size of S-Box we are able to mix one byte at a time. On the other side, in ARX based constructions, generally we mix 32-bit data at a time i.e. four bytes are mixed at once. So if we do it using S-Boxes, then we need series of S-Box looks up or quite a big S-Box that may not fit into L1 Cache of certain CPUs.

b) Secondly, S-Box lookups can also be exploited for timing attacks. Bernstein has shown the same in [175].

C) Rotation over Shift:

Rotations create better diffusions than Shift operations when used with XOR. Generally, one rotation and XOR will yield the same level of diffusion as two shifts (one in the direction of rotation and other in opposite direction) will yield with two XORs. Example below illustrates the concept:

Example of Rotation with one XOR

$Given\ Data\ Item\ (say\ A\ of\ 8\ bits)$: 1 0 1 0 0 1 1 1

$Z = A <<< 3$: 0 0 1 1 1 1 0 1

$A \oplus Z$: 1 0 **0 1 1 0** 1 **0**

$Diffusion\ Achieved$: $5\ bits\ (red\ colored)$

Example of Shift with one XOR

$Given\ Data\ Item\ (say\ A\ of\ 8\ bits)$: 1 0 1 0 0 1 1 1

$Z = A \ll 3$: 0 0 1 1 1 0 0 0

$A \oplus Z$: 1 0 **0 1 1** 1 1 1

$Diffusion\ Achieved$: $3\ bits\ (red\ colored)$

Example of Two Shift with two XORs

$Given\ Data\ Item\ (say\ A\ of\ 8\ bits)$: 1 0 1 0 0 1 1 1

$Y1 = A \ll 3$: 0 0 1 1 1 0 0 0

$Y2 = A \gg (8-3)$: 0 0 0 0 0 1 0 1

$Z1 = A \oplus Y1$: 1 0 0 1 1 1 1 1

$Z2 = Z1 \oplus Y2$: 1 0 **0 1 1 0** 1 **0**

$Diffusion\ Achieved$: $5\ bits\ (red\ colored)\ equal\ to\ one\ rotation\ with\ XOR$

In addition to the reasons mentioned above, **one important reason** of using ARX over any other operations was the objective to design a hash function that performs better (faster) than Skein at the similar level of security. Skein is based on ARX operations and it would not have been possible to perform faster than Skein by using any other set of operations that are fully complete.

4.7.3 Need for Padding and Initial Value

Cocktail follows Merkle-Damgard's Iterated hash structure that make use of fixed length collision resistant compression function $f\colon \{0,1\}^a \ X \{0,1\}^b \rightarrow \{0,1\}^c$ to design variable length input collision resistant hash function $H\colon \{0,1\}^* \rightarrow \{0,1\}^n$. The details about MD Structure are given in Chapter 2- Review of Literature.

Adding message length (padding) and fixing IV (Initial Value) in MD structure is very important as without these, finding second pre-image or collision is trivial. The following two paragraphs, dwell on the rationale behind inserting length padding and fixing IV in ***Cocktail***.

Padding's importance: Consider a message M that is divided into N message blocks $M^0, M^1, .., M^{N-1}$ and is being processed by hash function $H(\,)$ that does not use message length padding. An adversary that wishes to find another message X such that $H(M) = H(X)$ need to randomly select X' until he gets $H(X')$ equal to any of the N chaining values computed during calculation of $H(M)$. When such a X' is found, the adversary can concatenate X' with remaining blocks of M starting from the given chaining value to get $X = X'||M^k \dots M^{N-1}$ such that $H(X) = H(M)$. Here k represents iteration of compression function at which we get chaining value equal to X'. This attack can be foiled if message length is appended at the end of the message as message length is expected to differ for X and M. Message padding also helps in thwarting fixed point attacks as insertion of fixed point pair will not match with existing message length.

IV's importance: Correcting block attack may be used to generate 2nd pre-image of an existing message M. To generate 2nd pre-image X of target message M, adversary choses one of the message block M^i, and replaces it with an alternative block X^i such that $f(H^i, M^i) = f(H^i, X^i)$. If all other blocks of M and X are same, the same hash result will be obtained and thus 2nd pre-image can be found. This attack is feasible, if the pre-images for compression function can be obtained with the computation starting from pre-specified chaining values. However, if we fix the Initial Value, this attack can be thwarted [50].

Cocktail's iterative structure also make use of MD strengthening i.e. have a fixed Initial Value (16 words of 32 or 64 bit each) and length of the message is built within the

padding technique. Having Fixed value and length padding overcomes the challenges as mentioned above.

4.7.4 Using Wide Pipe Design with HAIFA features

Certain pitfalls have been observed for Merkle Damgard Structure (MD structure). Length extension attacks also known as “Message extension” or “padding attacks” is one of the well known weaknesses of MD construction. Joux in [46] also studied generic multi-collision attack on iterated structure and showed that finding multi-collisions is not much harder than finding ordinary collisions. Dean [73] presented different techniques that uses fixed points to produce attack on complete hash functions even in the presence of Merkle-Damgard strengthening. Kesley and Schneier [74] also improved generic correcting block attack using the notion of expandable messages such that it bypasses the defense provided by MD strengthening. Kesley and Schneier [74] used multi-collision technique to produce an expandable message.

The Wide Pipe Iterated Design proposed by Lucks [49] and HAIFA structure proposed by Biham and Dunkleman [51] create more security against the generic attacks mentioned in previous paragraph. These two designs are introduced in Chapter 2.

Cocktail*'s** structure implements wide pipe design where internal pipe (chain value) is double the size of hash output. The Output Transformation O (as defined earlier under the head ‘Specification of ***Cocktail-512’) does the same purpose as g mentioned in Figure 6. Wide Pipe Design ***Cocktail*** also includes $\#bits$ i.e. number of bits hashed so far in the iterative structure and thus implements features of both Wide Pipe Design and HAIFA to overcome majority of generic attacks as mentioned earlier.

4.7.5 Support for 32-bit as well as 64-bit Architecture

Cocktail proposes set of two hash functions. ***Cocktail-512*** works on 32-bit words and ***Cocktail-1024*** works on 64-bit words. Undoubtedly, the 32-bit version will work faster on a 32-bit architecture and 64-bit version will give better performance on 64-bit machines. This philosophy of supporting two word sizes is akin to SHA-2 that also support both 32-bit and 64-bit versions. Such approach is always beneficial than supporting only one version. For example Skein [9] supports only 64-bit version and thus runs more efficiently on 64-bit architecture compared to 32-bit systems. Also NIST

Federal notice for SHA-3 competition [27], prescribed both 32-bit (x86) and 64-bit (x64) version of operating system as Reference platform.

Looking at the current industry scenario, down to the levels of processor and operating system, it becomes evident that though the industry is progressing towards 64-bit architecture, yet a considerable segment is still using 32-bit machines, operating systems or applications. In the PC and laptop segments, majority of processors being launched are of 64-bit that support both 32-bit and 64-bit operating systems and applications. But at the level of portable devices like smartphones, ARM (Advance RISC machines) is leading the market and its majority of 'Application' series processors are of 32-bit with exception of Cortex-A57 and Cortex A-53 processors that support both 32-bit and 64-bit processing [146]. An analysis of operating systems on PC / Laptops shows that the market share of 64-bit operating systems is increasing day by day (perhaps because majority of new machines are being shipped with updated copy of operating system which is generally 64-bit) but still a lot of machines use 32-bit operating systems. A close insight into Smartphone segment is also interesting. Android has considerable market share in smartphone segment. The entire Android stack, that is based on Linux, is in fact, 32 bit that is cross compiled usually over 32/64-bit host environment. Host environment is generally one of distributions of Linux. Google recommends Ubuntu for building and cross compiling Android. From last three versions of Android, i.e. Ice Cream Sandwich (Android 4.0) and Jelly Beans (Android 4.1) onwards, cross compilation with 64-bit host environment is being promoted.

From the foregoing discussion it is evident that though the trend is in the direction of 64-bit architectures but the current status demands provision of options that support both 32-bit and 64-bit versions. History also suggest in the same direction. When 32-bit processors were launched, transition from 16-bit windows to 32-bit windows took considerable time. In fact, it took more than 10 years (from 1985 to about 1995) to get a 32-bit windows operating system and even after launch of 32-bit windows, a few people continued to use 16-bit windows applications on older version of windows for considerable time. Undoubtedly in this transition from 32-bit to 64-bit, industry has learnt from previous experience and new OS (64-bit) are being released at the same time as new processors but still we do not have too many 64-bit applications. Already about 15 years have passed since the advent of the first 64-bit processors and the surge of 64-bit

operating systems over 32-bit OS is being witnessed in the last few years only. In spite of this, 32-bit applications are still prevailing and dominating the market.

Taking cognizance to all the points as discussed above, it was decided to design ***Cocktail*** that supports both 32-bit and 64-bit words.

4.7.6 Use of Modified ChaCha Core and Choice of Rotation Distances

Cocktail uses **M**odified **C**haCha **C**ore (MCC) which is an improvisation over Salsa and ChaCha core. The first section of this chapter, under the head '4.1 Analysis of Quarter Rounds of Salsa and ChaCha Core and Proposal of an Alternative Design (MCC) for Maximizing Diffusion' provides thorough details of MCC and also proves that MCC creates more diffusion than Salsa and ChaCha core. MCC's Quarter round, on an average generates diffusion matrices with mean of 7.716 against 6.6424 of ChaCha and 4.0992 of Salsa i.e. on an average, gain of **16%** from ChaCha and **88%** from Salsa Quarter round. More than 44% permutations of *i, j, k, and l* were found that result in diffusion matrices having mean greater than ChaCha's diffusion matrix. That section also gives the rationale behind choice of rotation constants.

However, all the details mentioned in that section are applicable for ***Cocktail-512*** *(*32-bit word size). ***Cocktail-1024*** deals with 64-bit word size and accordingly a similar experiment (based on algorithm specified in '4.1.3 Experiment Used to Measure the Diffusion Property of Quarter Rounds') was conducted for 64-bit word size also. For 64-bit word size, the performance of Quarter round of **M**odified **C**haCha **C**ore (MCC) for different values of rotation constants *(i, j, k, l)* is given in Figure 62. For generating this plot, more than **16 million (to be precise 16,777,216)** possible permutation of rotation *constants i, j, k and l* (each varying from 0 to 63) were evaluated.

There are more than 72 sets of rotation distances that generate diffusion matrix with mean greater than 8.45. The set {8, 34, 16, 37} results in diffusion matrix with highest mean (8.4942). However, this study opted for set {52, 41, 16, 0} with mean 8.4676. Both these sets of rotation constants generate comparable diffusion matrices and are quite efficient. In the first set {8, 34, 16, 37}, two rotation distances are byte aligned and in other set {52, 41, 16, 0} one rotation distance is zero (i.e. fourth rotation is not required) and one rotation distance is byte aligned. The rotation distances opted for were the second set i.e. **{52, 41, 16, 0}**.

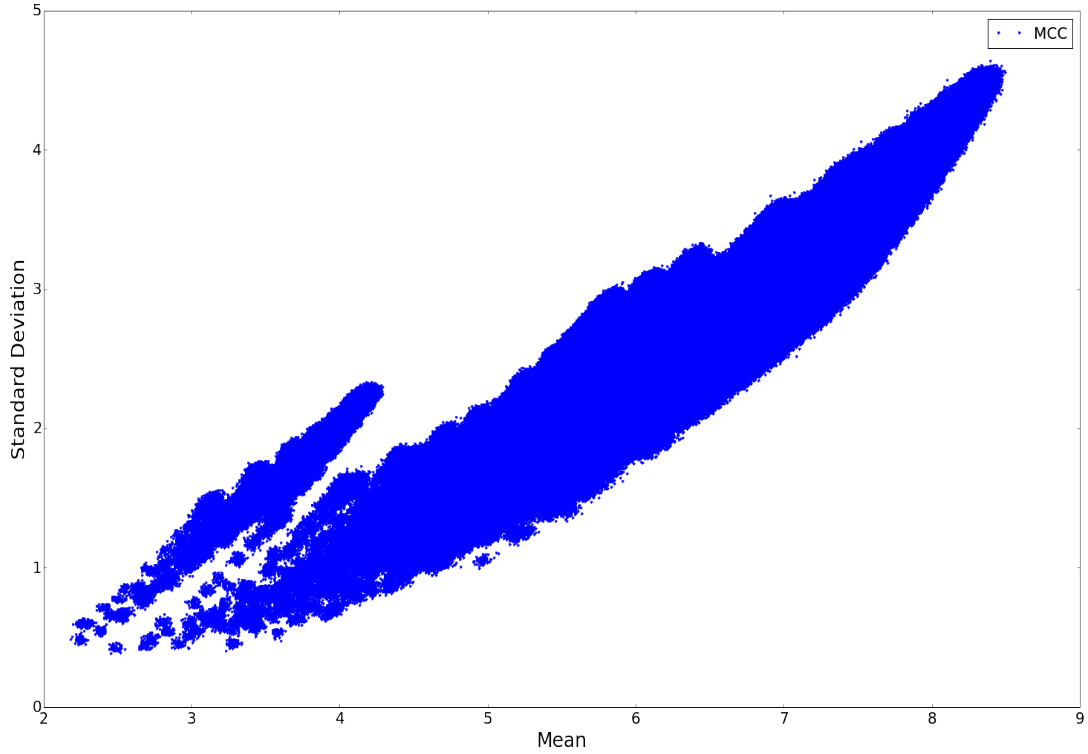


Figure 62. Mean and Standard Deviation of Diffusion Matrices of MCC's Quarter Round (64-bit)

There were 2 more set of rotation distances [{7, 34, 18, 0}, {54, 31, 45, 0}] among the top 72 sets where the fourth rotation constant was zero but these rotation distances were not selected as other three rotation distances of these sets were not byte aligned. Also the mean of diffusion matrices generated by these sets were lesser than the rotation constants finally opted for this study.

The diffusion matrix for Quarter round and Double round generated by rotation distances {52, 41, 16, 0} is given in Table 18 and Table 19 respectively.

Table 18. Diffusion Matrix for MCC's Quarter Round (64-bit version)

OP / IP	a	b	c	d
a	15.684	9.234	7.287	17.492
b	11.533	7.273	5.282	13.588
c	7.928	4.451	4.45	9.634
d	5.429	2.46	2.461	7.213

Table 19. Diffusion Matrix on One Double Round of MCC (64-bit version)

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
x_0	28	25	18	18	17	27	25	17	17	15	26	22	32	20	28	31
x_1	30	29	26	22	6	14	9	6	6	6	13	10	13	9	6	15
x_2	20	19	13	8	25	31	32	27	9	9	18	15	13	8	8	15
x_3	25	22	14	14	10	22	20	15	29	27	31	31	20	13	13	25
x_4	25	20	13	13	15	26	23	15	15	10	22	20	31	29	27	31
x_5	31	32	30	28	18	28	26	18	17	17	27	25	23	17	15	26
x_6	15	13	9	6	22	30	29	26	6	6	13	9	11	6	6	13
x_7	15	13	8	8	8	20	18	13	27	25	31	32	15	9	9	18
x_8	17	15	9	9	8	15	13	8	13	8	20	18	32	27	25	31
x_9	31	31	29	27	13	25	20	13	14	14	26	23	20	14	9	21
x_{10}	26	23	17	15	28	31	32	30	18	18	28	26	25	17	17	27
x_{11}	13	10	6	6	6	16	13	9	26	22	30	29	9	6	6	14
x_{12}	14	9	6	6	6	13	10	6	9	6	16	13	29	26	22	30
x_{13}	31	32	27	25	9	18	15	9	8	8	15	13	18	13	8	20
x_{14}	22	20	15	10	27	31	31	29	13	13	25	20	22	14	14	26
x_{15}	27	24	17	17	15	26	23	17	30	28	31	32	25	18	18	28

4.7.7 Rationale behind Sequence of Parameters Passed in Row and Column Rounds of MCC

The logic behind the sequence of parameters passed in Row and Column rounds, as detailed in '4.1.4-C) Results for Quarter Round of MCC' and Figure 52, is based on the level of diffusion that various output words experience in MCC's diffusion matrix. Diffusion matrix of 64-bit version (Table 18) of MCC′s Quarter round shows similar behaviour as that of 32-bit version (Table 13) i.e. word 'd' is diffused most, followed by word 'a', then word 'b' and at last word 'c'. So the logic for deciding sequence of parameters is applicable for 64-bit version of MCC also.

4.7.8 Number of Rounds

Number of rounds is dependent on how efficiently internal primitive structure of our compression function diffuses / mixes the bits of message block / internal state matrix

and thus achieve full diffusion. Full diffusion is the number of rounds to propagate a single-bit change to all bits.

For understanding how many rounds ***Cocktail's*** compression function consumes to achieve full diffusion, the Column and Row rounds of MCC should be pseudo-run. For explaining this point, 32-bit version of MCC is used here. Appendix-VI details the output of dry (pseudo) run and it is very much evident from the output that **one bit change in internal state affects all 512-bits of internal state within two rounds**. In fact, full diffusion is accomplished even before the completion of the second round. In Appendix-VI, Quarter round is represented as an eight step process. Row Quarter rounds of Round-2 accomplish full diffusion in 4th or 5th step of Quarter round.

The other way of analysing the diffusion property of compression function is based on diffusion matrices computed for MCC through the experiment detailed in '4.1.3 Experiment Used to Measure the Diffusion Property of Quarter Rounds'. Table 13 gives the diffusion property of basic primitive of compression function i.e. Quarter round which is used in Column and Row round. One-bit change in any of the four input words is spread to different bits of all four words by Quarter round. Certain words get more diffusion than others. For example, random change of one bit in input word 'a', on an average results in change of more than 13 bits of word 'a', 9 bits of word 'b', 7 bits of word 'c' and 15 bits of word 'd'. However, on an average, one-bit change in input words, results in more than 7-bit change in output words. When multiple call to Quarter round function are made by Column and Row rounds, then this diffusion gets multiplied. For example if one bit of word x_0 of internal state matrix X is changed, then first call to Quarter round by Column round (for x_0, x_4, x_8, x_{12}) will spread these changes in all $x_0, x_4, x_8, and\ x_{12}$words. After this call, on an average 13, 9, 7 and 15 bits of $x_0, x_4, x_8, and\ x_{12}$ are changed. Four subsequent calls to Quarter round function by Row rounds will spread these changes to all the remaining words of internal state matrix. In fact, mixing of bits will get multiplied. For second row x_6, x_7, x_4, x_5 ; input word x_4, is already affected by 7 bits and call to Quarter round will affect almost all bits of x_6, x_7, x_4, x_5. Looking at worst case (fourth row of Table 13) or using a dry run (as per Appendix-VI), it can easily be concluded that before completion of second round every bit of internal state matrix affects all other 512 bits of internal state matrix / message block.

Based on the properties of the basic primitive which result in full diffusion in second round, we decided to have **10 rounds in *Cocktail-512* to give at least 5** full diffusions in each compression function.

The **decision to have at least five full diffusions was based on** the attacks on previous hash functions. Insufficient diffusion has been exploited in the past to attack the hash functions. Attack on MD and SHA family of hash functions [68] [69] [78] [121] [176] [177] are examples where insufficient diffusion has been exploited. In SHA-1, full diffusion takes place after 30 steps (total steps - 80) and thus offer diffusion factor of 2.7. SHA-256 and SHA-512 have better diffusion factor. In the former, full diffusion takes place after 14 steps (total steps – 64) and in the later, full diffusion takes place after 18 steps (total steps – 80) so they offer diffusion factor of 4.6 and 4.4 respectively. It was decided to have full diffusions more than that of SHA-256 and SHA-512. In ***Cocktail***, full diffusion is achieved in the second round, even before it is fully completed (At least 30 percent operations of the second round are still pending when full diffusion is achieved). So with a target of at least five full diffusions, the following number of rounds are proposed for ***Cocktail***:

Cocktail-512 - 10 Rounds

Cocktail-1024 - 12 Rounds

The other argument that vindicates the proposal to have 10 rounds for ***Cocktail-512*** is based on performance of Salsa20, ChaCha, and **M**odified **C**haCha **C**ore. The following points present an alternative argument in favour of this decision:

a) Each round of ***Cocktail-512*** is equivalent to one Double round of Salsa20 or ChaCha or MCC. So 10 rounds ***Cocktail-512*** are equivalent to 20 rounds (10 Double rounds) of Salsa20 or ChaCha or MCC.
b) ECRYPT's eSTREAM project [17] chose a version of Salsa20 with 12 rounds (6 Double rounds) in its profile and is still being used as a profile algorithm. MCC′s 12 rounds (6 Double rounds) are at least as strong as 12 rounds of Salsa20. MCC gives much better diffusion than Salsa and thus offers much better security. So if we keep eSTREAM as a benchmark, only six rounds of ***Cocktail-512*** would have been

enough. But opting for a conservative choice of 10 rounds (equivalent to 20 rounds of MCC which are stronger than Salsa20's 20 rounds) was better for providing high security margins.

c) ***Cocktail's*** basic primitive MCC is an improvisation of ChaCha [158] which itself is an improved version of Salsa. Till date the best attack on ChaCha is reported by [178]. Aumasson et al. [178] attacked 256-bit ChaCha6 (6 round ChaCha in place of 20) with complexity of 2^{139} and ChaCha7 with complexity of 2^{248}. On the other hand, 128-bit ChaCha6 was attacked with complexity of 2^{107} and the author claimed that the attack failed to break 128-bit ChaCha7. The author also showed collision and pre-image attacks for simplified versions of Rumba 3 (three round Rumba), a compression function based on Salsa.

   The ***Cocktail's*** compression function has been generated using **M**odified **C**haCha **C**ore. Attacks on three rounds of Rumba should not affect the compression function of ***Cocktail*** as MCC is considerably different from Salsa and provide much better security. Similarly attacks on 6 rounds of ChaCha are not exactly on compression function and does not reflect any pre–image or collision attack on compression or hash function. And lastly ChaCha is not being used exactly.

d) Even for reference, if it is assumed that same attack is possible on compression function of ***Cocktail***, and somehow (hypothetical assumption) some sort of simplified variation of ***Cocktail*** is exploited up to 6 or lesser rounds out of 20 rounds. Even in such a hypothetical scenario, we have a safety factor of more than 3.3. It is worth noting that when AES [5] standardization process was underway, at that time attack on 6 of 10 rounds of AES were reported i.e. AES had a safety factor of about 1.7. Similarly during SHA-3 competition, when Skein [9] algorithm was being submitted, attack on 35 of 72 rounds of tweaked Threefish-512 (underlying block cipher for Skein hash function) was already reported i.e. Skein at the time of submission had a safety factor of little over 2.0. So ***Cocktail*** has much higher security factor to play with.

Cocktail-1024 has more rounds, as larger state size requires more rounds to spread the affect on all bits and achieve similar level of security.

We have kept number of rounds as a tuneable parameter. Current proposal of 10 rounds for ***Cocktail-512***, performs faster than all SHA-3 final round candidate algorithms. For

more conservative security margins, the number of rounds may be increased further. The experiment reflects that up to 14 rounds, ***Cocktail*** will still perform faster than majority of SHA-3 final round candidate algorithms. Secondly, the number of rounds will affect the throughput but not the memory requirements or gate requirements for implementation on Hardware.

4.7.9 Four Column Quarter Rounds followed by Four Row Quarter Rounds rather than Interlacing of Row and Column Quarter Rounds

Every round of ***Cocktail's*** compression function uses four Column Quarter rounds followed by four Row Quarter rounds. The other option is to have interlaced Column and Row Quarter rounds i.e. one Column Quarter round followed by a row Quarter round that is succeeded by another Column round and Row round and so on. Interlaced Column and Row Quarter round can result in faster diffusion in the first round compared to the other option of having four Column Quarter rounds followed by four Row Quarter rounds. For comparing these two options, a dry run of these options is done to cross check how change in one bit of word x_0 will affect other words.

Option-1: Four Column rounds followed by four Row rounds: Four Column Quarter rounds call the Quarter round function of MCC for each column one by one. Details are mentioned under the head '4.1.2 Modified ChaCha Core (MCC)'. The first call will spread the effect of one bit change in x_0 to all words of the first column i.e. to (x_0, x_4, x_8, x_{12}). The subsequent calls will not have any common words and thus will not spread any changes in x_0 to words in the second, third, and fourth columns. Now when the first Row Quarter round is called, the effect of change in x_0 is diffused to other words of the first round and similarly succesive Row Quarter rounds spread the effect to other rows. Table 20 reflects how diffusion spreads in this option. Red coloured words mean, the word has some impact of change in one bit of x_0.

Option – 2: Interlaced Column and Row Quarter rounds: The first Column Quarter round will spread the effect of one bit change in x_0 to all the first column words and subsequent Row round will immediately pass on the diffusion to words of first column and successive Column round will effect all words of second column. Table 21 reflect the spread of diffusion in this option. As evident from in Table 21, diffusion is faster in Option – 2 (Interlaced Column and Row Quarter round). Not only faster, the

diffusion is more. For example before calling second Row Quarter round, x_4 and x_5 words have already got some effect of change in one bit of x_0 word. However, the call to the second Row Quarter round will effect and diffuses these words further.

Table 20. Spread of Diffusion in Four Column Quarter Rounds followed by Four Row Quarter Rounds

First Column QR	Second Column QR	Third Column QR	Fourth Column QR
$\begin{bmatrix} \mathbf{x_0} & x_1 & x_2 & x_3 \\ \mathbf{x_4} & x_5 & x_6 & x_7 \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & x_1 & x_2 & x_3 \\ \mathbf{x_4} & x_5 & x_6 & x_7 \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & x_1 & x_2 & x_3 \\ \mathbf{x_4} & x_5 & x_6 & x_7 \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & x_1 & x_2 & x_3 \\ \mathbf{x_4} & x_5 & x_6 & x_7 \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$
First Row QR	**Second Row QR**	**Third Row QR**	**Fourth Row QR**
$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & x_5 & x_6 & x_7 \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & \mathbf{x_9} & \mathbf{x_{10}} & \mathbf{x_{11}} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & \mathbf{x_9} & \mathbf{x_{10}} & \mathbf{x_{11}} \\ \mathbf{x_{12}} & \mathbf{x_{13}} & \mathbf{x_{14}} & \mathbf{x_{15}} \end{bmatrix}$

Table 21. Spread of Diffusion in Interlaced Column and Row Quarter Rounds

First Column QR	First Row QR	Second Column QR	Second Row QR
$\begin{bmatrix} \mathbf{x_0} & x_1 & x_2 & x_3 \\ \mathbf{x_4} & x_5 & x_6 & x_7 \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & x_5 & x_6 & x_7 \\ \mathbf{x_8} & x_9 & x_{10} & x_{11} \\ \mathbf{x_{12}} & x_{13} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & x_6 & x_7 \\ \mathbf{x_8} & \mathbf{x_9} & x_{10} & x_{11} \\ \mathbf{x_{12}} & \mathbf{x_{13}} & x_{14} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & \mathbf{x_9} & x_{10} & x_{11} \\ \mathbf{x_{12}} & \mathbf{x_{13}} & x_{14} & x_{15} \end{bmatrix}$
Third Column QR	**Third Row QR**	**Fourth Column QR**	**Fourth Row QR**
$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & \mathbf{x_9} & \mathbf{x_{10}} & x_{11} \\ \mathbf{x_{12}} & \mathbf{x_{13}} & \mathbf{x_{14}} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & \mathbf{x_9} & \mathbf{x_{10}} & \mathbf{x_{11}} \\ \mathbf{x_{12}} & \mathbf{x_{13}} & \mathbf{x_{14}} & x_{15} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & \mathbf{x_9} & \mathbf{x_{10}} & \mathbf{x_{11}} \\ \mathbf{x_{12}} & \mathbf{x_{13}} & \mathbf{x_{14}} & \mathbf{x_{15}} \end{bmatrix}$	$\begin{bmatrix} \mathbf{x_0} & \mathbf{x_1} & \mathbf{x_2} & \mathbf{x_3} \\ \mathbf{x_4} & \mathbf{x_5} & \mathbf{x_6} & \mathbf{x_7} \\ \mathbf{x_8} & \mathbf{x_9} & \mathbf{x_{10}} & \mathbf{x_{11}} \\ \mathbf{x_{12}} & \mathbf{x_{13}} & \mathbf{x_{14}} & \mathbf{x_{15}} \end{bmatrix}$

Though option-2 seemed better, even then we opted for option–1 i.e. four Column Quarter rounds followed by four Row Quarter rounds because this gives chance to exploit parallelism and achieve better performance in terms of speed. All four Column rounds can be executed in parallel and similarly all four Row Quarter rounds can be executed in parallel after Column Quarter rounds. The speed gain achieved with parallelism can be close to 4 times wherever the diffusion speedup is not comparable to this speed gain.

4.7.10 Maximum Size of Input Message

Cocktail imposes a constraint on the length of the message (without padding) that it should be less than 2^{128} as number of bits reserved for storing size of message during the padding step in ***Cocktail-1024*** is 128 bits. This means that using ***Cocktail*** hash function

we can not hash message of more than 2^{128}. But this is not a problem as 2^{128} bits is big enough a size that generally the storage of a system can manage. The following example will further illustrate that message length constraint of ***Cocktail*** is not a serious problem at all.

Let us say we have a message of 2^{128} bits and we need to send it to someone over the network. If we try to transfer this data with a speed of 2^{64} bits per second then it will take years to send this much data. And a communication network that can transmit the data at a speed of 2^{64} bits per second is currently not available. The maximum speed of network that was recently reported is 255 Tbps (close to 2^{48} bits per second) [179].

4.7.11 Key Schedule / Key-Expansion

Generally, we find complex key schedules in various block ciphers and hash functions. Complex Key Schedules require considerable clock cycles to setup. For large blocks of message, it is not a big cost but when small size messages are to be encrypted or hashed and key is to be changed for every next block, then complex key schedules are sometime irritating. However, Complex key schedule may act as a motivation to have lesser rounds. Key Expansion of ***Cocktail*** is inspired from AES. It is considerably simple and requires fewer operations. ***Cocktail's*** key is not calculation intensive. Ten round compression function of ***Cocktail-512*** requires 68 XORs, 20 additions, and 8 rotations for generating key schedule. Sub-Keys do not repeat and can be implemented using a small looping construct and one branch instruction. Sub-keys do have impact of number of bits hashed so far and diagonal words of sub-key are added with number of bits hashed so far to make sure that each column and row round gets some impact of this value. The details about ***Cocktail's*** Key Schedule is mentioned under the heading 'Specification of ***Cocktail-512***'.

4.7.12 Matyas-Meyer-Oseas Iterative Structure

The work of Preneel et al. [91] referred to in section '2.5.1 - A) Single Block Length Construction', explains various ways of constructing hash functions from block ciphers. Preneel et al. showed that there are 64 options of creating iterative structures (Figure 10) and 12 out of these are secure (Table 1). The most commonly used schemes among these 12 listed secure schemes are Matyas-Meyer-Oseas [93], Miyaguchi-Preneel [94] [95],

and Davies–Meyer [96] [97]. These three commonly used schemes are given in brief as under:

Matyas-Meyer-Oseas scheme

$$H^i = E_{H^{i-1}}\left(M^i\right) \oplus M^i$$

Miyaguchi-Preneel Scheme

$$H^i = E_{H^{i-1}}\left(M^i\right) \oplus H^{i-1} \oplus M^i$$

Davies-Meyer Scheme

$$H^i = E_{M^i}\left(H^{i-1}\right) \oplus H^{i-1}$$

Cocktail uses Matyas-Meyer-Oseas scheme to construct iterative hash structure from compression function based on MCC. Selection of Matyas–Meyer–Oseas scheme over the other two schemes (listed above) is based on some personal convenience and the fact that the other schemes have no added advantage over Matyas-Meyer-Oseas scheme. However, Davies-Meyer scheme does suffer some drawbacks and a few of these have been discussed by Ferguson et al. in [9]. These drawbacks with reference to ***Cocktail*'s** iterative structure are highlighted here.

The main ways of attacking hash functions involve adversary to choose data input. Data input act as key in Davies-Meyer scheme and thus corresponds to the chosen-key attack on cipher. While in case of Matyas-Meyer-Oseas scheme (used in ***Cocktail's*** structure), it corresponds to chosen-plaintext attack. The cryptographic community has considerably high experience in safeguarding block ciphers against chosen-plaintext attacks, but have less experience in the field of chosen-key attacks. So it was decided to stay with what the world knows better.

The second argument that goes in favour of Matyas-Meyer-Oseas scheme is defence that both these schemes present against differential cryptanalysis. In ***Cocktail*** (based on Matyas-Meyer-Oseas scheme), difference in data block results in difference in each round of the compression function. However, with Davies-Meyer scheme, as data block is used as a key, so difference can be cancelled at the level of one sub-key and reintroduced at successive sub-key level. And this could be done repeatedly within one block also. This drawback makes differential attack considerably easy which is equivalent to giving free pass to differentials through some rounds of compression function.

Lastly, finding fixed point in a Davies Meyer structure is quite simple. For any message block M^i, finding value of H^i such that $H^i = E_{M^i}\left(H^i\right) \oplus H^i$ is easy as we just need to compute $H^i = E_{M^i}^{-1}(0)$.

The above arguments were the main rationale behind the decision of designing iterative structure based on Matyas-Meyer-Oseas Scheme.

4.7.13 Familiarity and Based on Intensively Analysed Constructs

Keeping primitives based on some structure that has been analysed thoroughly brings more confidence in its security. The second advantage is that the cryptographers and cryptographic software implementers will feel familiar with ***Cocktail***.

Cocktail makes use of matrix notation for representing internal hash state and other data constructs. Matrix operations have already proven to be really helpful in creating efficient diffusion and confusion and has been used in AES [5], Whirlpool [8], Grøstl [24] and many other cryptographic primitives.

Cocktail is based on MCC, an improvised version of ChaCha which itself is an improved version of Salsa. Salsa 20/12 (12 round version) is also a profile algorithm in ECRYPT's eSTREAM project [17] since April 2008. Both Salsa and ChaCha have been considerably analysed and not much weakness has been observed in them. Existing cryptanalysis on Salsa and ChaCha gives a fair confidence in Modified ChaCha core's security and compression function built on that.

4.7.14 Flexibility

One of the concerns of this study was to design a hash function that could generate varying size message digests. This concern provided a major rationale to have separate output transformation which can generate digests of varying sizes. Flexibility to have message digests of varying sizes increases the application area of hash functions.

Cocktail offers flexibility in terms of number of rounds also. User can increase or decrease the rounds. Though 10 rounds are recommended for ***Cocktail-512,*** the value can be changed depending on the performance and security margins that the user of hash function is looking for.

Also ***Cocktail*** can be implemented on different environments (hardware and software) varying from 8-bit microcontroller to large processors. As memory requirements given

under the head '4.6 Complexity of ***Cocktail***' are not huge, ***Cocktail*** can be implemented easily in resource constrained environments like Smart Cards. This study plans to implement ***Cocktail*** in FPGA, ASICs, 8 bit Microcontrollers, and other environments in the near future.

4.7.15 Speed

Another major concern, that needs a special mention, is performance of hash function in terms of execution speed. Though the objective of this study was to design an algorithm that performs better than Skein [9], yet the endeavours were to have performance better than other SHA-3 final round candidate algorithms also.

The decision to have an ARX based design was backed up by this objective of speed. ARX based designs offer a very good speed. The decision to follow 32-bit or 64-bit system also has a lot to do with speed expectations. The 32-bit and 64-bit systems found favour in this study in place of byte oriented approach used in AES [5] and many other cryptographic primitives. Byte oriented approach might have helped in achieving efficient diffusion and also help to allay the concerns about endian-ness of the system. But 32-bit or 64-bit systems help in achieving performance efficiency much better than byte oriented approach with just one caveat that full diffusion must take place; and in ***Cocktail*****,** full diffusion does take place efficiently. The performance degradation in case of byte oriented approach was evident from the results that have been obtained for various SHA-3 final round candidate algorithms while pursuing the first objective of this research. Performance of Grøstl (byte oriented design) has been found quite slow as compared to other algorithms.

4.8 Using *Cocktail*

Cocktail can be used at all places where hash functions are generally used for implementing various security objectives. Various applications of hash functions have been listed in Chapter 2 under the heading '2.1 Security Services of Cryptographic Hash Functions'. Therefore, this section gives a list of some important ways of using ***Cocktail*** for implementing various security applications including those mentioned in Chapter 2.

4.8.1 *Cocktail* as HMAC

HMAC (Hashed Message Authentication Code), as specified in NIST's standard FIPS PUB 198-1 [180], is a technique to achieve data integrity and message authentication. ***Cocktail*** may be used to implement HMAC that takes two input: Message M and Key K (shared between sender and receiver). HMAC can be obtained from ***Cocktail*** by computing the following:

$$HMAC_{Coctail}\ (K, M) = \ Cocktail[(K_0 \oplus opad)||Cocktail\{\ (K_0 \oplus ipad)||\ M)\}\ ]$$

Cocktail-512 or ***Cocktai-1024*** may be used depending on the requirement. If ***Cocktail-512*** is used, then the message M is divided into blocks of 512-bit each. K_0 is the modified key obtained after necessary pre-processing of 512-bit key. As defined in FIPS PUB 198-1 [180], $ipad$ and $opad$ are inner and outer pad respectively. There is no property of ***Cocktail*** that prevents its usage as HMAC. Figure 63 represents usage of HMAC for achieving message integrity and data origin authentication.

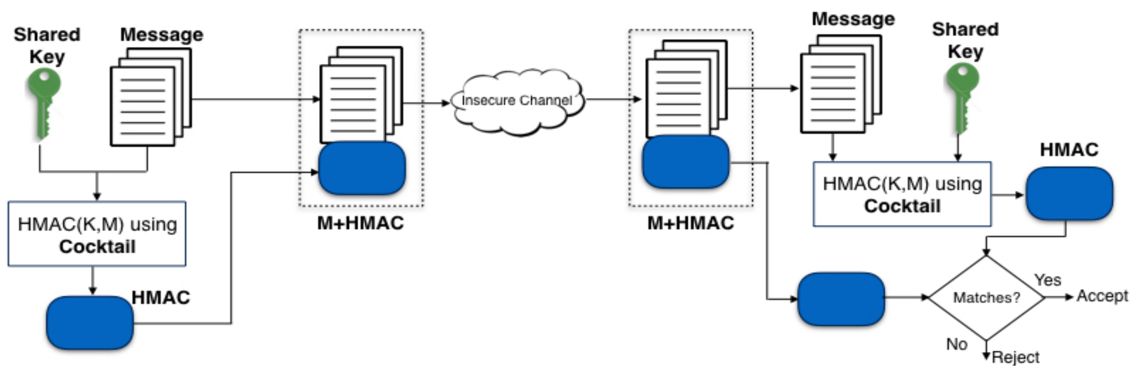


*Figure 63. HMAC with **Cocktail** for Achieving Message Integrity and Authentication*

4.8.2 *Cocktail's* Compression function as CMAC

CMAC is cipher based message authentication code and is recommended by NIST in [181]. CMAC may be used to generate MAC like HMAC but from block cipher. So rather than using ***Cocktail*** as a whole, we can use the compression function of ***Cocktail*** (without T words i.e. number of bits hashed so far) to generate CMAC.

For using ***Cocktail*** in CMAC, the shared key is used as Initial Value for the compression function. Figure 64 illustrates how CMAC can be generated from ***Cocktail's*** compression function. Structure is based on [181]. The k used in last block is obtained by multiplying o/p of compression function with $x\ or\ x^2$ depending on whether the message is padded or not.

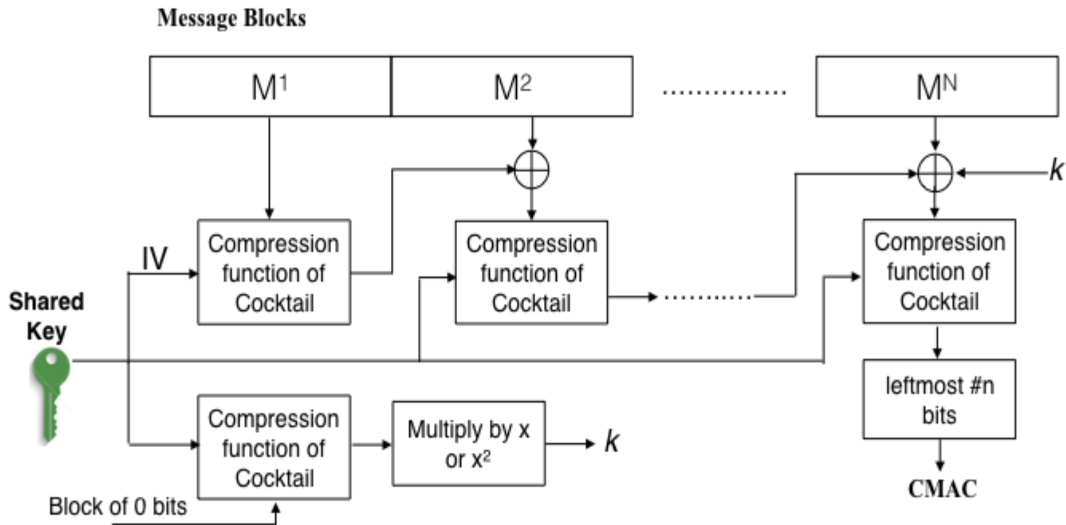


*Figure 64. CMAC using **Cocktail's** Compression Function*

4.8.3 *Cocktail* as PRF Family

Pseudo Random Function family (known as PRF family) is a collection of efficiently computable hash functions that can emulate random oracle. To generate a family of PRF from ***Cocktail***, we can use any one of the following techniques:

a) The $index$ used to select function can be concatenated to Message input M either before or after M. i.e. $Cocktail\ (M||index)$ or $Cocktail\ (index||M)$. So depending on index, we will have separate random output of message M.
b) The second way is to have Initial Value of ***Cocktail*** based PRFs dependent on $index$. Initial Value for various functions can be computed as $IV_i = f(IV, i)$. Where IV_i is the Initial Value for i^{th} pseudo random function.

4.8.4 *Cocktail* in Digital Signature

Cocktail may be used to implement Digital Signatures efficiently. Rather than signing message M, the digest $Cocktail(M)$ is signed with private key of sender. This makes signature quite efficient as encrypting a long message with asymmetric key is time consuming. $Signature = Encrypt_{Pvt_key_sender}(Cocktail(M))$. The Message M and $Signature$ is sent to receiver. On the other side, receiver can verify the authenticity and integrity by calculating $Cocktail(M) = Decrypt_{Pub_key_sender}(Signature)$ and digest again and then comparing both. The scheme is shown in Figure 4 and discussed in '2.1.2 Implementing Efficient Digital Signature'. The approach shown in Figure 4 is based on

RSA approach. Other digital signature schemes prescribed as FIPS 186-4 standard [182] may also be implemented using ***Cocktail*.**

4.8.5 *Cocktail* in Randomized Hashing Mode

Randomized hashing may be used to free the security of digital signature from relying on collision resistance of underlying cryptographic hash function. NIST, in its Special Publication SP 800-106 [183] , has recommended to randomize the message before using it in digital signature. ***Cocktail*** may be used in randomized mode and to do the same, the input message is randomized using a random value rv for every signature following the technique outlined in [57] or [183]. If we use technique specified in [183], random value rv between 40 bits to 1024 bits is generated. Then two other values - RV and len_{rv} - are computed. RV is obtained by repeating rv as many times as required and then appending the leftmost bits of rv to make it equal to message size whereas len_{rv} is 16-bit representation of length of random value. ***Cocktail*** in randomized hashing mode is represented as:

$$Cocktail_{random}\,(M, rv) = \,Cocktail\,(rv||M \oplus RV||len_{rv})$$

4.8.6 *Cocktail* for Password Based Authentication

Password is one of the oldest and most commonly used technique of entity authentication to access a system. If password is stored as plaintext in the password file (database) on a system, then then the administrator can access the same.

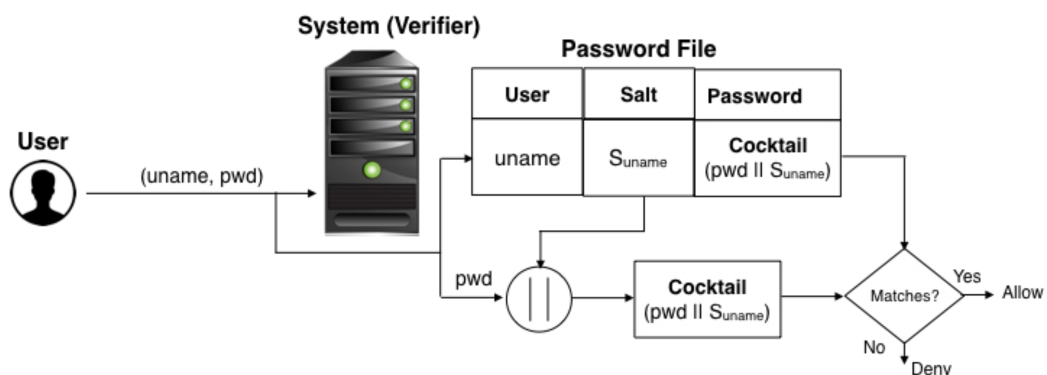


*Figure 65. Password Based Authentication Using **Cocktail***

However, if the hash result of the password is stored in password file then even verifier (administrator) can not have access to the same. Salting the password is better technique to make brute-force attack tougher even if length of the password is small. Figure 65 represents how ***Cocktail*** may be used for password based authentication.

4.8.7 *Cocktail* as Key Derivation Functions (KDF)

Key derivation functions use PRFs like cryptographic hash or cipher or HMAC to derive one or more secret keys (also known as derived keys) from a master key (password / passphrase) and a salt value. To generate these derived keys, KDFs iterate PRFs multiple times to add computational work which makes cracking of the derived key much more difficult. PBKDF2 – Password Based Key Derivation Function 2 is a KDF that is published as IETF's (Internet Engineering Task Force) RFC 2998 [184].

The ***Cocktail*** may be used in PBKDF2 using the following mechanism.

$$Key_{dervied} = PBKDF2\ (Cocktail, Password, Salt, iteration, len_{dervied_key})$$

$len_{dervied_key}$ is desired length of derived key and is multiple of output length of ***Cocktail*** function (n). Maximum value is limited to $(2^{32} - 1) * (n)$. $Key_{dervied}$ is computed using

$$Key_{dervied} = KD_1 \,||\, KD_2 || \dots || KD_{(len_{dervied_key})/n}$$

$$where\ KD_j = HMAC_{Cocktail}\ (Password, Salt, iteration, j).$$

Cocktail will be used in HMAC mode (section 4.8.1) and called multiple times. However, input to the same will change in every iteration.

$$HMAC_{Cocktail}\ (Password, Salt, iteration, j) = CT_OP_1 \oplus CT_OP_2 \oplus \dots \oplus CT_OP_{iteration})$$

where CT_OP_i is output of ***Cocktail*** in HMAC mode.

In the first call to ***Cocktail*** in HMAC mode, $Password$ is inputted as key and $Salt$ concatenated with j, encoded in 32-bit integer format, is inputted as message input. In the subsequent calls, output of the previous call act as message input and $Password$ continue to act as key. If output of multiple calls is named CT_OP. Then we have

$$CT_OP_1 = HMAC_{Cocktail}\ (Password, Salt \,||\, Integer_32bit(j))$$

$$CT_OP_2 = HMAC_{Cocktail}\ (Password, CT_OP_1)$$

:

$$CT_OP_{iteration} = HMAC_{Cocktail}\ (Password, CT_OP_{iteration-1})$$

Generally, number of $iterations$ is kept quite high. At the time of writing the PBKDF standard for the first time, the recommended value was 1000 and nowadays even iterations of more than 100000 are also used for password based authentication at server side.

4.8.8 As Stream and Block Cipher

Cocktail's basic primitive MCC may be used to generate efficient stream cipher on the same lines as Salsa and ChaCha Core are used. ***Cocktail's*** compression function (based on MCC) is a bijective function and thus can be used for encryption as well as decryption. ***Cocktail's*** compression function (without use of T words i.e. number of bits hashed so far) may be used to generate block ciphers processing 512-bit message blocks in any operation mode like ECB (Electronic Codebook) or CBC (Cipher Block Chaining) or CFB (Cipher Feedback). ***Cocktail*** may also be used with RKC (Random Key Chaining) mode [185], enlisted as one of the 14 recommended authenticated encryption mode by NIST, Computer Security Division, U.S. Department of Commerce [186] which is designed by the team consisting of present researcher. Whenever ***Cocktail's*** compression function is used as block cipher, decryption algorithm will process inverse of MCC′s Quarter round as shown below:

MCC's Quarter Round (for Encryption)	**Inverse of MCC's Quarter Round (for Decryption)**
Step 1: $b = b + a$;	**Step 1:** $d = d \oplus a$;
Step 2: $c = (c \oplus b) \lll 4$;	**Step 2:** $a = a - b$;
Step 3: $d = d + c$;	**Step 3:** $b = c \oplus (b \ggg 8)$;
Step 4: $a = (a \oplus d) \lll 17$;	**Step 4**: $c = c - a$;
Step 5: $c = c + a$;	**Step 5:** $a = d \oplus (a \ggg 17)$;
Step 6: $b = (b \oplus c) \lll 8$;	**Step 6**: $d = d - c$;
Step 7: $a = a + b$;	**Step 7:** $c = b \oplus (c \ggg 4)$;
Step 8: $d = d \oplus a$;	**Step 8:** $b = b - a$;

Also in decryption algorithm, row rounds will be processed before column rounds and sub-keys will be injected in reverse order.

4.8.9 Other Applications of *Cocktail* as Hash Function

In the preceding paragraphs, different modes of using ***Cocktail*** have been discussed. However, ***Cocktail*** may be used to implement various other security services that are based on any generic hash functions like Digital Time stamping, generating Pseudo Random Number Generators, identifying file or data, verifying file integrity on computer

system and the enormous gamut of email, IP, and web security protocols. All these are listed in '2.1 Security Services of Cryptographic Hash Functions' also.

4.9 Security Aspects of *Cocktail*

The design philosophy, as detailed in '4.7 Design Philosophy, Design Decisions, and Its Rationale', has already highlighted different design decisions that ensure security of ***Cocktail.*** This section touches upon various security aspects of ***Cocktail*** to substantiate that it does not succumb to important generic attacks.

4.9.1 Local Collisions

Cocktail is free from localized collision in a compression function. Local collision in a compression function exists when two distinct messages yield same internal state (within a compression function) after few rounds. ***Cocktail*** exhibits this property because of the fact that its round function is a Double round of MCC with an infusion of key and thus practically is a permutation of input message i.e. permutes input message using various Quarter rounds. So, for two distinct messages, with same initial state (IV and T words) the permutations and in turn output will be different. And this property will hold for any number of rounds.

4.9.2 Fixed Points

Fixed point is a chaining variable H_i such that $f(H_i, X_i) = H_i$. Once fixed point exist, the presence of message block X_i does not affect the message digest and pre-images can be computed by creating expandable messages (using arbitrary number of X_i messages) as detailed in [73] and [74].

Fixed points may be easily computed for compression functions constructed using 'Davies-Meyer' principle i.e. compression function constructed in the following manner:

$$f(H^{i-1}, M^i) = E_{M^i}(H^{i-1}) op\ H^{i-1} \quad \{op \text{ is XOR or ADD operation}\}$$

In such cases adversary can create a fixed point by selecting a message M^i and computing $H^{i-1} = E_{M^i}^{-1}(0)$. This H^{i-1} acts as fixed point as shown below:

$$H^i = f(H^{i-1}, M^i) = E_{M^i}(H^{i-1}) op\ H^{i-1} = 0\ op\ H^{i-1} = H^i$$

SHA-1, SHA-2 and many other hash functions are based on Davies Meyer scheme and hence fixed points can be easily found for these functions.

As ***Cocktail*** is based on Matyas-Meyer-Oseas scheme [93], so fixed point can not be easily computed as mentioned above. To find fixed point for ***Cocktail*** we need to find $M^{i-1} \oplus f_{Cocktail}(M^{i-1}, H^{i-1})$ that results in H^{i-1} and estimating this is not simple as in Davies-Meyer based structures. Secondly, Internal state of ***Cocktail*** is double the size of hash output which results in cost of constructing expandable message from fixed point quite huge ($2^{l/2} = 2^{n}$).

4.9.3 Simulates Random Oracle

A hash function is always desired to work as random oracle i.e. for a random input the output should also be random. In other words, we can say that any difference or relation between two inputs should not be statistically related to difference or relation of corresponding output values. ***Cocktail's*** compression function is based on MCC, a variant of Salsa and ChaCha core, and till date any non-randomness property of these constructs has not been identified. In [178], non-randomness (truncated differentials having 1-bit input and output difference) up to 3 steps of ChaCha was detected (equivalent to one and a half round of ***Cocktail*** if we hypothetically assume this attack is applicable on MCC also) but this was not visible in the fourth step. ***Cocktail*** inherit pseudo-randomness of ChaCha core. Infusion of round dependent key and input of T words (counter recording number of bits hashed so far) to compression function simulates each call of compression function unique and all this contributes in making ***Cocktail*** complicated enough to behave like random oracle.

4.9.4 Length Extension

Length extension (also known as padding or message extension) attack is a weakness of MD structure. Given $h = H(M)$, and an adversary computes $M' and\ h'$, such that $h' = H(M||M')$ even for unknown M (but for known length $|M|$). The attack uses $H(M)$ as an internal hash for computing $H(M||M')$.

Cocktail does not suffer from length extension attack. Input of T words (number of message bits hashed so far) in call to compression function for each message block prevents length extension attack as the last call to compression function becomes unique. This is illustrated by the following example. Let us assume we have a message M of 1040 bits. After padding, the message will be of three block $M^0, M^1\ and\ M^2$. T^0 will contain

512, T^1 will contain 1024, and T^2 will contain 1040. Final chain value (before output transformation) will be $H^3 = f_{Cocktail}(H^2, M^2, T^2)$ i.e. $H^3 = f_{Cocktail}(H^2, M^2, 1040)$. If adversary applies length extension attack and extend the message with another block M^3 (with desired padding) and tries to hash $M' = M^0||M^1||M^2||M^3$ by using chain value between M^2 and M^3 then adversary will receive $H^3 = f_{Cocktail}(H^2, M^2, 1040)$. But in fact for M' the correct value should be $H^3 = f_{Cocktail}(H^2, M^2, 1536)$. So length extension is prevented. The presence of output transformation also makes length extension impossible (even without T words) because $H(M)$ that the adversary uses is not H^3. It is $O(H^3)$ and it can not be used as intermediate chain value to compute $H(M')$.

4.9.5 Collision Multiplication

Aumasson et al. [26] have coined the term collision multiplication to refer to the ability of extending a given collision (M, M') to derive a number of other collisions. For example in MD structure let us assume we already have a collision (M, M') for a hash function H with $|M| = |M'|$. Hash function pads M and M' to get $\overline{M}$ and $\overline{M'}$. Now by choosing any message suffix s , we can have $N = \overline{M}||s$ and $N' = \overline{M'}||s$ which will also collide. So a single collision (M, M') can be used to generate multiple collisions for free. It is basically a type of length extension attack. The method as stated above is not easy to apply on ***Cocktail*** unless the message collision takes place before output transformation and finding collision will require $2^{l/2}$ time complexity (as per Birthday attack) which is equal to or greater than 2^n for all hash outputs and this is certainly more than brute-force attack. Also for the same reasons as mentioned in the previous point, ***Cocktail*** resists collision multiplication.

4.9.6 Joux Multicollisions

By k-multicollisions, we mean that there are k messages that all collide to same hash value. Joux in [46] presented an attack to find k-multicollisions for a hash H in time $O(log_2 k * 2^{\frac{n}{2}})$ instead of $\Omega(2^{n*\frac{d-1}{2}})$. The Joux multicollision is applicable to ***Cocktail*** also and for that matter applicable to all hash functions based on HAIFA or MD structure. The wide internal state used by ***Cocktail*** helps in resisting any practical concerns because of Joux multicollisions. As per Lucks [49], the complexity to find k-collisions in a wide pipe design is approximately of $log_2 k * 2^{\frac{l}{2}}$ where $l \geq 2 * n$ in case of ***Cocktail.*** That

makes time complexity of $log_2 k * 2^n$ for ***Cocktail*** which is comparable to a Brute-Force k-collision attack for which complexity is around $k!^{1/k} * 2^{k*(k-1)/k}$ [24].

4.9.7 Kelsey & Schneier and Faster Multicollisions

Kesley and Schneier in [74] highlighted another method of finding multi-collisions. The cost of finding k-collisions with this method is independent of k and is given by $3 * 2^{n/2}$, where n is size of hash output. The method works only when fixed points for compression function are easily computed. As fixed points for ***Cocktail*** cannot be easily computed so Kesley and Schneier's method of multicollisions is not applicable to ***Cocktail.*** Faster multicollisions suggested by Aumasson et al. in [187] is also not applicable to ***Cocktail*** because that technique also requires fixed points to be found in compression function and it is assumed that IV is chosen by the adversary.

4.9.8 Second Pre-image attack

Dean in [73] and Kelsey and Schneier in [74] presented generic attack on an n-bit hash function to generate second pre-image in much less than 2^n evaluations of the compression function. For an n-bit hash function based on n-bit compression function, these techniques generate second pre-image in 2^{n-k} evaluations of compression function if the first pre-image of 2^k message block is given. The large internal state of ***Cocktail*** makes this attack also not worth worrying about. For example, if the first pre-image of 2^{64} blocks is given then finding second pre-image using these methods will require 2^{l-64} which is considerably higher than 2^{n-k} .

4.9.9 Algebraic Attacks

Algebraic attacks express the cipher operation as system of equations, then substitute known data in some of the variables and then solve for key. The high complexity of ***Cocktail*** and absence of any such attack on Salsa and ChaCha core reflects the resistance of ***Cocktail*** against these attacks.

4.9.10 Side Channel Attacks

Cocktail process all input messages in the same way, independent of its content and thus is not prone to timing attacks or power analysis. Not using S-Boxes also help in preventing these attacks. Cache-Timing analysis of eSTREAM finalists was carried out

by [188] which suggests that Salsa and ChaCha are not vulnerable to this attacks. ***Cocktail's*** basic primitive MCC is based on ChaCha and Salsa and thus ***Cocktail*** inherit this property.

4.9.11 Few Other Important Concerns

In the last few points, ***Cocktail's*** resistance to major generic attacks have been discussed. A few other important factors highlighting security aspects that need special mention are: Using fixed IV, padding the length of the message, and possibility of differential cryptanalysis. The message padding and fixed initial value prevents trivial attacks like correcting block attacks which has already been discussed in the previous section under the head '4.7.3 Need for Padding and Initial Value'. Differential cryptanalysis uses relationship that may exist between differentials of input messages and corresponding output messages. ***Cocktail's*** each round is practically Double Round of MCC which is an improvisation of ChaCha and Salsa core's Double round for better diffusion. Differential analysis of ChaCha and Salsa core have been extensively analysed without any real success and this also reposes confidence in ***Cocktail***. Till date the best attack on ChaCha is reported by [178]. This attack is based on the existence of truncated differentials after three steps but after four steps no differentials were found. This means that if it is assumed hypothetically that such differentials can be found for MCC also, then it is equivalent to two rounds of ***Cocktail***. For 10 round ***Cocktail*** with infusion of sub-keys and counter words (representing number of bits hashed so far) these attacks are not worth worrying about. Most importantly, these differentials should not affect the compression function of ***Cocktail*** as MCC performs differently than ChaCha and offers better diffusion and thus better security also.

4.10 Concluding Remarks

The newly designed ***Cocktail*** is a simple, flexible, and efficient hash function that blends security with speed. Use of **M**odified **C**haCha **C**ore as a basic primitive ensures good speed and high diffusion properties and also gives scope of exploiting parallelism which can add to the performance extensively. ***Cocktail*** can be efficiently used in 32-bit as well 64-bit architecture. The various design choices for ***Cocktail*** make sure to prevent it from any generic attacks. In addition to a normal hash function, ***Cocktail*** can be used

in different other operating modes like HMAC, CMAC, Randomized hashing, Key derivation functions, and can also be used to build block and stream cipher.

CHAPTER 5: *COCKTAIL'S* PERFORMANCE COMPARISON WITH SKEIN AND OTHER SHA-3 FINALISTS

> *"Many things difficult to design prove easy to performance"*
>
> ***Samuel Johnson***

Second objective of the thesis is to design a new hash function that can act as a variant to Skein Hash family and perform better than Skein on Reference platform prescribed by NIST and Target platform chosen for first objective. In chapter 4, proposal for new hash function ***Cocktail*** was presented and this chapter presents the performance results of ***Cocktail*** both on Reference and Target platform. This chapter also carry the comparison of ***Cocktail's*** performance with Skein and other SHA-3 finalists.

The chapter is organized under the following headings:

- Performance Results of ***Cocktail*** **(5.1)**
- Comparison of ***Cocktail*** and **Skein (5.1)**
- Comparison of ***Cocktail*** and SHA-3 Finalists **(5.3)**

5.1 Performance Results of *Cocktail*

Cocktail may be implemented on different platforms varying from resource constrained devices like smart cards, 8 / 16 bit microcontrollers to large processors. Here in this section results of ***Cocktail*** are presented on x86 architecture (Reference platform [27] announced by NIST for SHA-3 competition) and on ARM architecture (Target platform selected for the first objective of this study).

5.1.1 Performance Results on Intel x86_64 and x86_32 Architecture

A) Hardware and Software Used

The machine used for performance results on **x86_64 architecture** had Intel Core i3 – 4150 CPU @ 3.50 GHz x 4 (Quad Core) processor that supports 64-bit instruction set architecture. The machine had 8 GB RAM with L1d and L1i Cache of 32K, L2 Cache of

256K and L3 Cache of 3072K. The operating system used was 64-bit version of Linux Ubuntu 14.04.1 (Kernel version: 3.13.0-32-generic).

For performance results on **x86_32 architecture,** Intel Core 2 Duo CPU E7500 @ 2.93 GHz x 2 (Dual Core) processor, that supports 32-bit instruction set architecture, was used. It had 2 GB RAM and L2 cache of 3MB. The machine operated on 32-bit version of Linux Ubuntu 14.04.1 (Kernel version 3.16.0-23-generic) operating system.

B) Methodology Used

All results were obtained by running tests on a platform where BIOS was optimized by removing every factor that could cause indeterminism. All power optimizations, Intel Hyper threading, core multiprocessing, frequency scaling, and turbo mode functionalities were turned off. Disabling core multiprocessing resulted in only one core available to the operating system.

To read TSC (Time Stamp Counter), RDTSC – Read Time Stamp Counter instruction supported by almost all x86 Intel Architecture systems was used. Using RDTSC before and after the code (to be profiled) gives the value of cycles consumed before and after the code and accordingly cycles consumed by the code can be calculated. However, there are multiple issues that needs to be addressed before using RDTSC instruction. These are:

- RDTSC instruction fills EAX and EDX register with value of TSC and change in these registers may result in segmentation fault. To avoid this, we should push the register before using RDTSC and pop after RDTSC usage. Other option is to use clobbered register in Inline assembly which acts as a message to the processor that this instruction is effecting EAX and EDX register. The second option was used in this study.
- Majority of Intel Architecture machines support 'out of order' execution which may break the temporal sequence of operations and might give incorrect result of cycle count because of execution of adjacent instructions between two calls to RDTSC, which may not be part of the code to be profiled. To overcome this issue, the solution lies in using some serializing instruction. CPUID instruction was used as serializing instruction and its usage before first call to RDTSC ensures execution of all instructions before RDTSC.

- Using combination of CPUID and RDTSC at the end of code (to be profiled) may add variance in results as highlighted by Paoloni in [189]. So, combination of RDTSCP and CPUID as suggested in [189] was used for x86_64 machine but not in x86_32 machine as that machine did not support RDTSCP instruction. The code used is discussed as an example in Appendix-II.
- To counter the effect of caches, warming up was done by calling the code a few times before profiling it.

Code written in 'C' was compiled using *"gcc – O3"* option i.e. the highest possible optimization level for execution speed (level 3). The Cycles per Byte (CPB) consumed by ***Cocktail*** are listed in Table 22. For computing CPB, the cycles after computation were divided by total bytes after padding. Averaging and overhead subtraction was done in the similar fashion as mentioned in '3.5.2 E) Averaging the Cycle Count and Subtracting the Overhead'.

*Table 22. Performance Results of **Cocktail** on Intel x86 Architecture*

Type of x86 Arch.	Input / Hash	1 byte	100 bytes	1000 bytes	5000 bytes
64-bit Arch.	**256 bit**	16.53	14.09	12.54	12.28
	512 bit	10.24	10.21	8.28	8.02
32-bit Arch.	**256 bit**	22.61	19.23	17.01	16.13
	512 bit	49.75	49.88	36.73	35.16

On **x86_64 architecture**, ***Cocktail,*** on an average, takes **8.7 Cycles Per Byte** to generate 512-bit hash and **13.0 Cycles per Byte** to generate 256-bit hash output. On **x86_32 architecture**, ***Cocktail,*** on an average, takes **39.2 Cycles Per Byte** to generate 512-bit hash and **17.6 Cycles per Byte** to generate 256-bit hash output.

5.1.2 Performance Results on ARM Architecture

The performance results of ***Cocktail*** on ARM architecture is given in Table 23. The presented results are for Cortex-A8, Cortex-M4, and ARM7TDMI processors.

Methodologies and tools used were same as those used for evaluation of SHA-3 final round candidate algorithms on this platform. On Cortex-A8, code was compiled using '*gcc –O3*' option.

*Table 23. Performance Results of **Cocktail** on ARM Architecture*

Type ARM Arch.	Input / Hash	1 byte	100 bytes	1000 bytes	5000 bytes
Cortex-A8	256 bit	45.38	37.20	30.33	29.47
	512 bit	103.45	103.15	91.99	89.44
Cortex-M4	256 bit	79.19	64.62	53.28	51.98
	512 bit	113.30	111.41	93.38	91.32
ARM7TDMI	256 bit	137.39	121.27	108.22	106.73
	512 bit	181.14	179.76	156.05	153.34

On **Cortex-A8**, ***Cocktail,*** on an average, takes **94.59 Cycles Per Byte** to generate 512-bit hash and **32.64 Cycles per Byte** to generate 256-bit hash output. On **Cortex-M4**, ***Cocktail,*** on an average, takes **97.81 Cycles Per Byte** to generate 512-bit hash and **57.37 Cycles per Byte** to generate 256-bit hash output. On **ARM7TDMI**, ***Cocktail,*** on an average, takes **161.86 Cycles Per Byte** to generate 512-bit hash and **113.2 Cycles per Byte** to generate 256-bit hash output.

5.1.3 Few Important Observations

- A close perusal of the results reflects that on all 32-bit machines (x86_32, Cortex-A8, Cortex-M4 and ARM7TDMI), 256-bit output takes lesser time than 512-bit output whereas on 64-bit machines (x86_64), 512-bit output is more efficient than 256-bit output.
- Just like any other hash algorithm, ***Cocktail's*** efficiency improves with increase in size of input message. The reason for this is that a few preliminary tasks are always required for implementation of the algorithm.
- ***Cocktail*** gives the best result on Intel x86 machines and it is quite obvious as the architecture is more powerful and dedicated for general purpose PCs as compared to ARM architecture that is targeted towards Mobile and Embedded systems.

The subsequent sections contain a comparison of ***Cocktail's*** performance with Skein and other SHA-3 final round candidate algorithms.

5.2 Comparison of *Cocktail* and Skein

Before discussing performance results of ***Cocktail*** and **Skein,** it is important to consider some important structural differences in both these algorithms that eventually affect their performances. The discussion on structural differences is followed by comparison of their performance results on Reference and Target platform.

5.2.1 Difference in Structure of *Cocktail* and Skein

Both ***Cocktail*** and **Skein** are based on Matyas-Meyer-Oseas structure [93] and use ARX (Arithmetic, Rotation and XOR) operations for hashing. The structure of ***Cocktail*** is detailed in Chapter 4 and introduction to **Skein** is given in Chapter 3 under the head '3.4 Introduction to SHA-3 Final Round Candidate Algorithms'. The important difference in the structure of these algorithms are:

a) ***Cocktail*** works on 64-bit as well as 32-bit word size while **Skein** works on 64-bit word size only. On a 64-bit machine, **Skein** might have minor advantage. However, supporting both the word sizes results in good performance of ***Cocktail*** on both 32-bit as well as 64-bit machines. The other important points that influenced the decision in favour of both 32-bit and 64-bit word size is listed in section '
b) 4.7.5 Support for 32-bit as well as 64-bit Architecture'.
c) **Skein** family consists of three different functions - **Skein-256**, **Skein-512**, and **Skein-1024** with internal state of 256, 512, and 1024 bits respectively - all operating on 64-bit word size. However, ***Cocktail*** family consists of ***Cocktail-512*** (32-bit) and ***Cocktail-1024*** (64-bit) with 512 and 1024-bit internal state only. ***Cocktail*** with 256-bit internal state has not been defined because of concerns shown by Lucks [49], and Biham and Dunkleman [51]. According to them, the maximum size of hash that can be generated with 256-bit internal state without any threat of inner collision (and thus length extension attack) is 128-bit. Based on Birthday paradox, 128-bit is too small a size for message digest as the brute-force collision attack will take approximately $1.18 * 2^{64}$ operations to break it and these operations are not a big challenge for machines available these days.

   Ferguson et al. [9] have dedicated **Skein-256** for constrained devices such as 8-bit Smart cards or other 16-bit devices. However, it will be really inefficient for an 8

or 16-bit devices to process an algorithm based on 64-bit words. ***Cocktail-256*** can also be designed for smaller constrained devices. By changing all the words from 32-bit to 16-bit and defining new constants (initial values, round constants, and rotation constants for MCC), ***Cocktail-256*** may be derived from ***Cocktail-512***. Such an algorithm is expected to be efficient than **Skein-256** on 8/16-bit constrained devices.

d) The padding technique used by **Skein** does not append message length at the end of message. Avoiding fixed point attacks is more challenging in this scenario. Insertion of fixed pairs add to message length and length padding can detect it. But without length padding, detection will be difficult. Similarly, attack discussed under the head '4.7.3 Need for Padding and Initial Value' will pose more challenges for **Skein**. On the other hand, ***Cocktail*** makes use of length padding to avoid such attacks.

e) **Skein** is based on tweakable block cipher Threefish and the basic primitive used by Threefish is a MIX function which is derived from Helix [190] and Phelix [191] ciphers. On the other hand, the basic primitive of ***Cocktail*** is **M**odified **C**haCha **C**ore, an improvisation of Salsa and ChaCha core used in stream ciphers.

f) Different MIX functions used in a specific round of **Skein** can be processed in parallel using Tree Hashing. However, compared to ***Cocktail*****,** the No. of operations that can be executed in parallel are lesser. Each MIX operation involves one addition, one rotation, and one XOR. A single round of **Skein-512** can execute four MIX operations in parallel. Contrary to this, ***Cocktail*** has considerable operations that can be executed in parallel to have good performance gains. All four Columns (and Row Quarter rounds) can be executed in parallel and each such round consists of (4 additions, 4 XORs and 3 rotations).

g) ***Cocktail*** is based on wide pipe design and uses the $\#bits$ hashed as one of input parameter to compression function. The wide internal state pipe and usage of $\#bits$ enhances security against Length Extension and Joux Multicollisions as stated in [49] and [51] . **Skein** on the other hand can use wide pipe and also uses tweak that can incorporate message size hashed so far. However, when **Skein-512** is used to generate 512-bit hash output, then its internal state is no longer wide which may create challenges in avoiding above mentioned attacks.

h) Diffusion is a desirable property of a hash function. For 512-bit internal state, **Skein's** compression function (Threefish Cipher) requires 10 rounds for one full diffusion (detailed in Appendix-VI). In comparison, ***Cocktail's*** compression function for the same internal state is able to generate full diffusion in the second round. In fact, full diffusion is achieved even before the second round is completed (More than 30 percent operations of second round are still pending before we achieve full diffusions). Appendix-VI details how full diffusion is achieved in ***Cocktail*** and **Skein**.

i) **Skein-512** works on diffusion factor of 7.2 (72 rounds and 10 rounds for one full diffusion) and ***Cocktail-512*** works on diffusion factor of 5.9 (10 rounds proposed and about 1.7 rounds for one full diffusion). Undoubtedly, **Skein's** diffusion factor is higher than ***Cocktail*** and to have same diffusion factor, ***Cocktail-512*** should be run with 12 rounds. The number of rounds have been kept as tuneable parameters and it is proposed to have 10 rounds based on few other rationales as discussed in '4.7.8 Number of Rounds'. Even with 12 rounds (to have diffusion factor of 7 or more), ***Cocktail*** executes faster than **Skein**.

j) Comparison of operations and memory used by both these algorithms is quite interesting and gives fairly good judgment about their performance characteristics. For ***Cocktail,*** both these details are explained under the head '4.6 Complexity of ***Cocktail***'.

Comparison of Operations:

- For 512-bit Internal state, **Skein** requires **1083 operations** involving **64-bit words** which on a 32-bit machine are equivalent to **2166 operations** on 32-bit words. (994 additions, 596 XORs and 576 rotations). In comparison, 10-round ***Cocktail*** for 512-bit internal state requires **1189 operations** involving **32-bit words** (388 additions, 533 XORs and 268 rotations). To have same diffusion factor as **Skein's** 72 rounds, a 12-round ***Cocktail-512*** requires **1387 operations** involving 32-bit words which are **36% lesser** than that of **Skein.**
- For 1024-bit internal state, **Skein** requires **2171 operations** involving **64-bit words (**871 additions, 660 XORs and 640 rotations). In comparison, 12 round ***Cocktail*** requires **1391 operations** (456 additions, 614 XORs and 317 rotations). If we increase the rounds from 12 to 14 (to have higher diffusion

factor), the No. of operations consumed by ***Cocktail*** will be **1585** i.e. **27% lesser** than **Skein**.

Comparison of Memory Requirement:

- For 512-bit internal state. ***Cocktail*** requires **96 bytes** for constants (64 bytes for Initial Value and 32 bytes for round constants used for key expansion). In comparison, **Skein** uses 32 bytes to store rotation constants for different rounds, 64 bytes for initial chaining values, 8 bytes for constant used in key derivation, and 8 bytes for permutation constants. All this sum up to **112 bytes**.
- In RAM, for 512-bit internal state ***Cocktail*** requires 270 bytes to store Hash State H, Internal State of Compression function, Sub-Keys (computed dynamically for each round), variable to store No. of bits hashed so far, and local copy of message block to be processed in each compression function. **Skein** also stores all these variables and the memory requirement for hash state, internal state of compression function, sub keys (computed dynamically), and for local copy of message block is same as ***Cocktail***. However, tweak words of **Skein** require 16 bytes compared to 8 bytes required to store input bits hashed so far in ***Cocktail***. In totality, the amount of RAM required is almost the same.

The above comparison reflects that both ***Cocktail*** and **Skein** can be easily implemented on memory constrained devices but from perspective of execution speed, ***Cocktail*** is expected to perform better.

k) Some of the important features of **Skein**, that need to be highlighted are mentioned below:

- **Skein** can generate variable output sizes (in byte increments). However, ***Cocktail's*** output is not as flexible as **Skein**. ***Cocktail*** generate few selected byte sizes as hash output (32, 64, 96, 128, 160, 192, 224, 256, 320, 384, 448, 512)
- As far as total number of operations are concerned, ***Cocktail*** undoubtedly outperforms **Skein**. However, Key Expansion of **Skein** is very efficient and can generate sub-keys much faster than ***Cocktail*.**

- **Skein** uses Tweak input in compression function that enables **Skein** to support multiple optional arguments which adds flexibility. For example, if 'Tree hashing' is to be done, then optional arguments can be setup accordingly.

5.2.2 Difference in Performance of *Cocktail* and Skein

This subsection presents the comparison of performance of ***Cocktail*** and **Skein** on Reference platform (x86 architecture) as prescribed by NIST in [27] and Target platform (ARM architecture) chosen under this study for evaluation of SHA-3 final round candidate algorithms. All the results are compared separately for 512-bit internal state (***Cocktail-512*** and **Skein-512** recommended for 256-bit hash output) and separately for 1024-bit Internal state ((***Cocktail-1024*** and **Skein-1024** recommended for 512-bit hash output)

A) Comparison on Reference Platform (Intel x86 Architecture)

Figure 66, Figure 67, Figure 68, and Figure 69 present the performance comparison of ***Cocktail*** and **Skein** on CISC based 64-bit and 32-bit Intel architecture commonly referred as x86_64 and x86_32 (IA32) architecture. The software and hardware tools used are same as mentioned in previous section '5.1.1 Performance Results on Intel x86_64 and x86_32 Architecture'. For comparison, **Skein's** reference submission as available on NIST SHA-3 competition website was used. The code for both ***Cocktail*** and **Skein** were written in 'C' language and compiled using *'gcc –O3'* i.e. highest level of optimization for execution speed. Clock cycles were computed by using RTDSCS, CPUID, and RDTSCP instructions as mentioned in previous section. All the other methodologies as mentioned in chapter 3 under the head '3.5.2 Methodology Used' were followed.

Some important observations from these graphs are listed below:

a) On Intel 64-bit machine (x86_64 architecture), ***Cocktail-512*** consumes **13 Cycles per Byte (CPB)** compared to **15.5 CPB** consumed by **Skein-512**. However, **Skein-512 performs better as size of input increases above 650-700 bytes**. The major reason behind comparatively better performance of **Skein** is the word size used by ***Cocktail-512***. ***Cocktail-512*** works on 32-bit word size and **Skein-512** works on 64-bit word size. On a 64-bit architecture, an algorithm processing 64-bit data will undoubtedly have an advantage. However, **Skein-1024** does not have the advantage over ***Cocktail-1024*** as both these algorithms use 64-bit word size

and inherent efficiency of ***Cocktail*** results in better performance. ***Cocktail-1024*** takes **8.68 CPB** compared to **22.73 CPBs of Skein**.

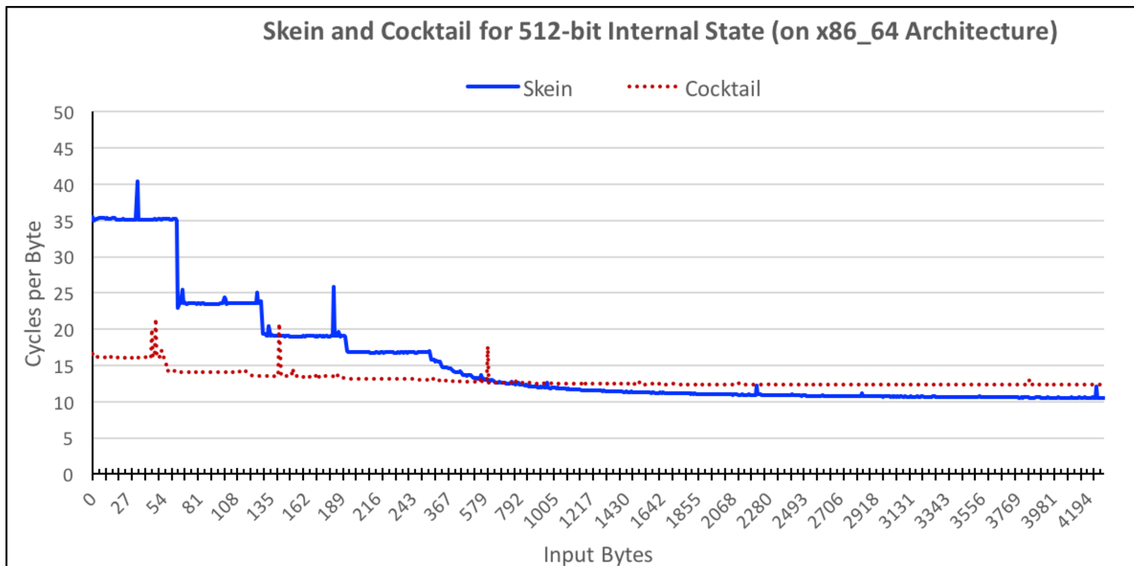


*Figure 66. Performance of Skein and **Cocktail** for 512-bit Internal State on Intel x86_64 Machine*

b) On Intel 32-bit machine (x86_32 / IA32 architecture), ***Cocktail-512*** performs much better than **Skein-512**. **Skein** does improve with increase in input size but on average the CPB consumed are 71.73 compared to 17.56 by ***Cocktail-512***. Similar scenario exists for 1024-bit internal state also. ***Cocktail-1024*** on an average consumes 39.2 CPB compared to 100.47 CPB by **Skein-1024.**

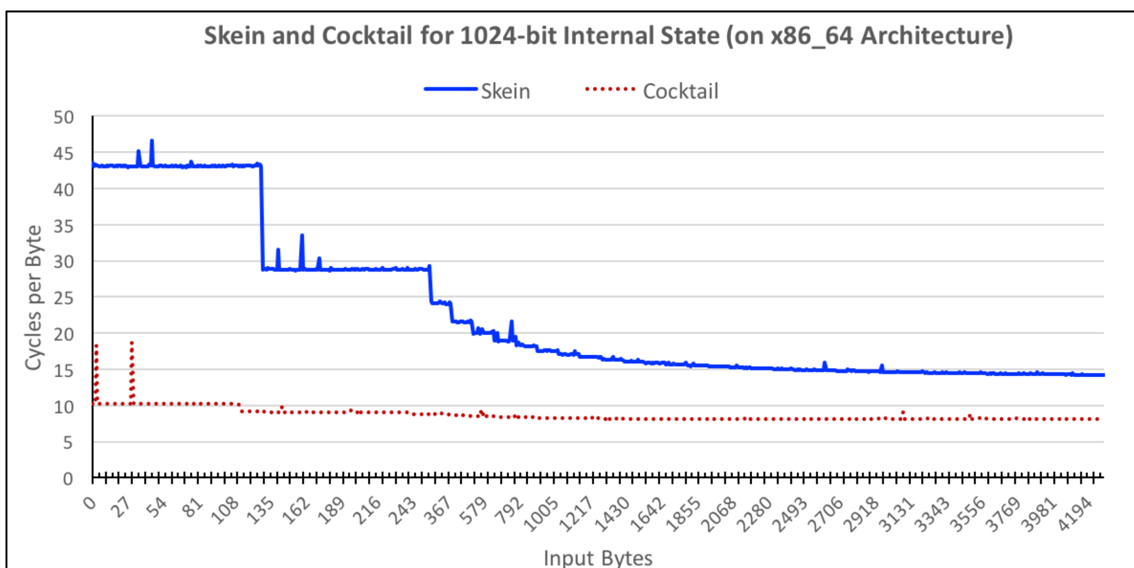


*Figure 67. Performance of Skein and **Cocktail** for 1024-bit Internal State on Intel x86_64 Machine*

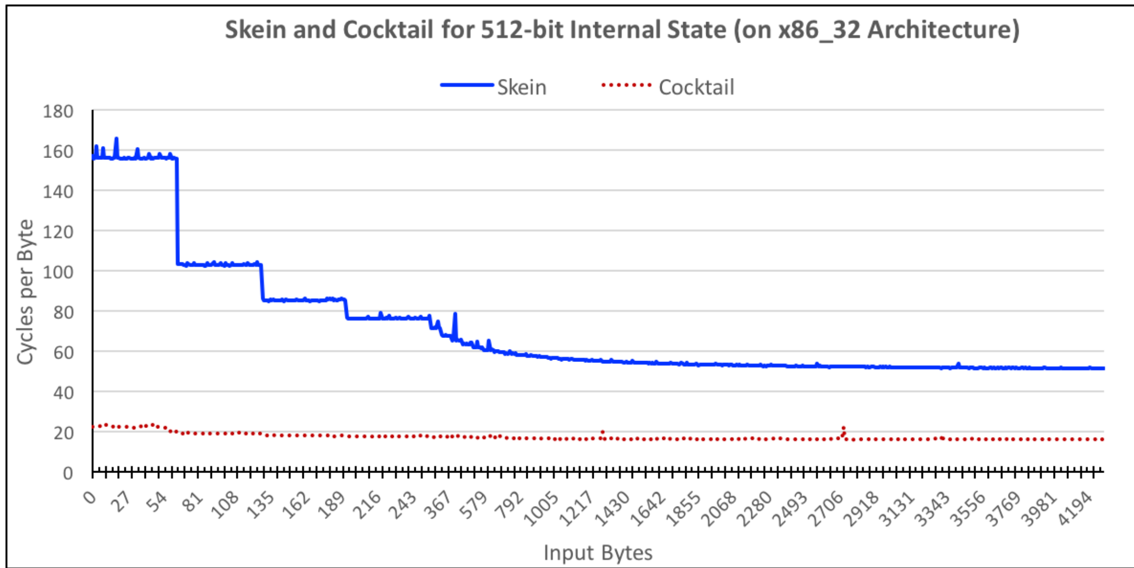


*Figure 68. Performance of Skein and **Cocktail** for 512-bit Internal State on Intel x86_32 Machine*

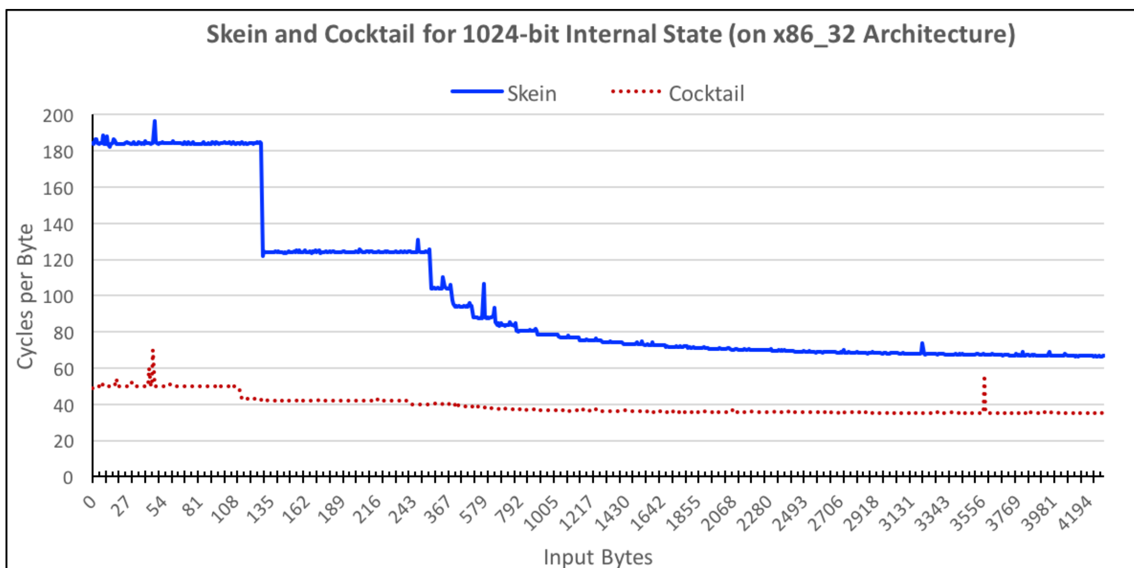


*Figure 69. Performance of Skein and **Cocktail** for 1024-bit Internal State on Intel x86_32 Machine*

B) Comparison on Target Platform (ARM Architecture)

The target architecture selected for evaluation of SHA-3 finalists was ARM Architecture of which Cortex-A8, Cortex-M4, and ARM7TDMI were used for evaluation (detailed in Chapter 3). In this subsection, the performance comparison of ***Cocktail*** and **Skein** on these processors is presented. Various tools, methodologies, and approach used for evaluation on the selected ARM processors were exactly the same as detailed in Chapter 3 under respective sections. For evaluation on Cortex-A8, codes of both algorithms were compiled with –O3 optimization. Figure 70, Figure 71 present results for

Cortex-A8; Figure 72, Figure 73 present results for Cortex-M4, and Figure 74, Figure 75 present results for ARM7TDMI. In all cases, ***Cocktail*** was found to perform better than **Skein.**

Some of important observations are highlighted below:

a) The average CPB consumed by ***Cocktail-512*** is 32.64, 57.37, and 113.2 for Cortex-A8, Cortex-M4 and ARM7TDMI respectively. In Comparison **Skein-512** consumes 270.86 CPB **(8 times more) for Cortex-A8**, 197.10 CPB **(3.4 times more) for Cortex-M4** and 238 CPB **(2 times more) for ARM7TDMI**.

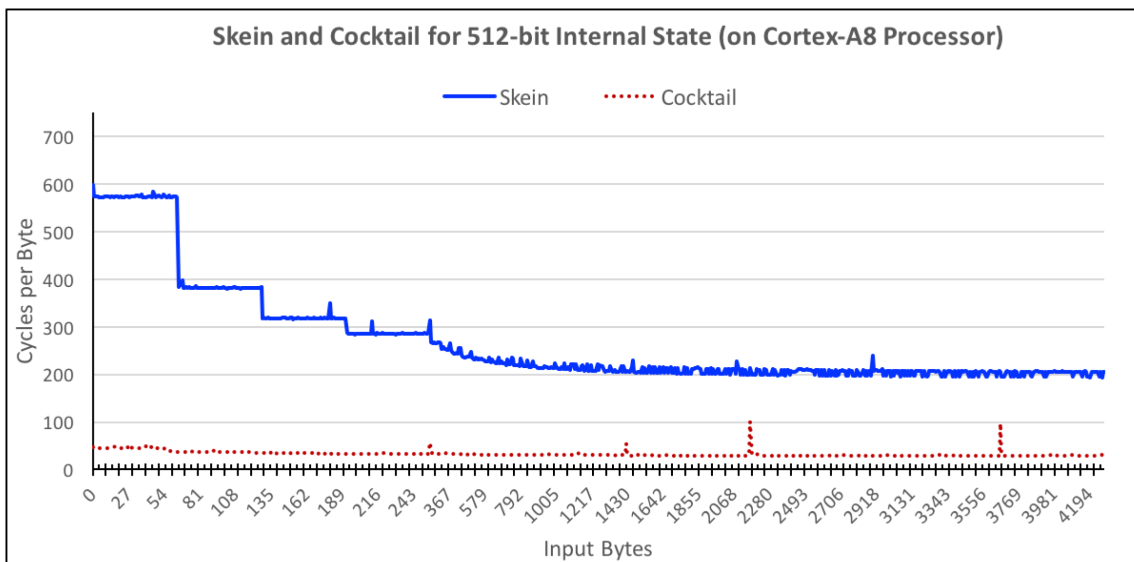


*Figure 70. Performance of Skein and **Cocktail** for 512-bit Internal State on Cortex-A8 Processor*

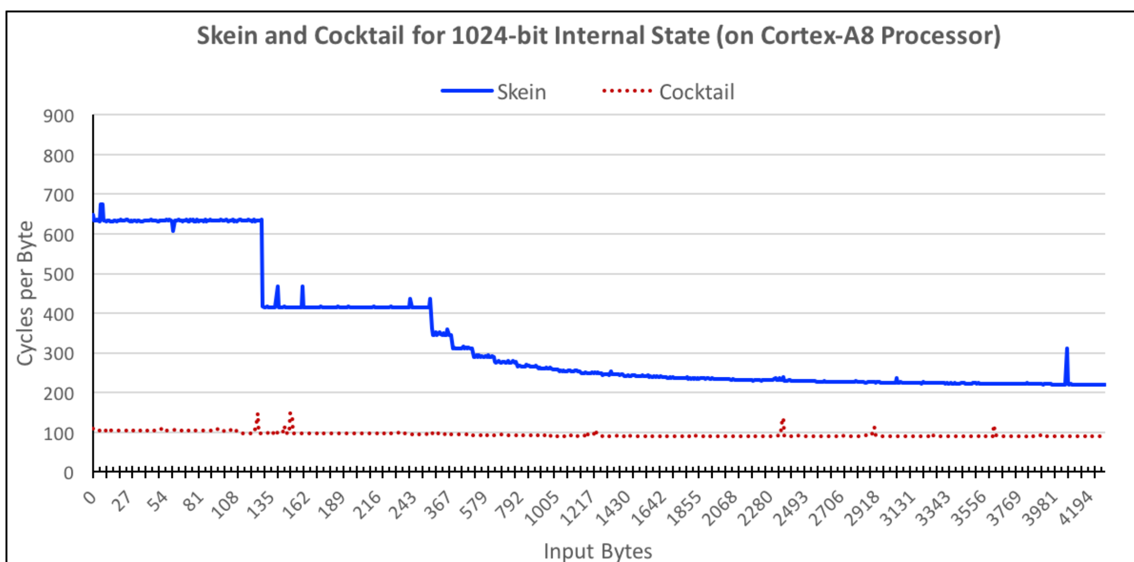


*Figure 71. Performance of Skein and **Cocktail** for 1024-bit Internal State on Cortex-A8 Processor*

b) For 1024-bit internal state also ***Cocktail*** performs better than **Skein**. CPBs consumed by ***Cocktail-1024*** are 94.59, 97.81, and 161.86 for Cortex-A8, Cortex-M4, and ARM7TDMI respectively. In comparison, **Skein-1024** consumes 337.22, 238.81, and 346.65 CPBs for Cortex-A8, Cortex-M4, and ARM7TDMI respectively.

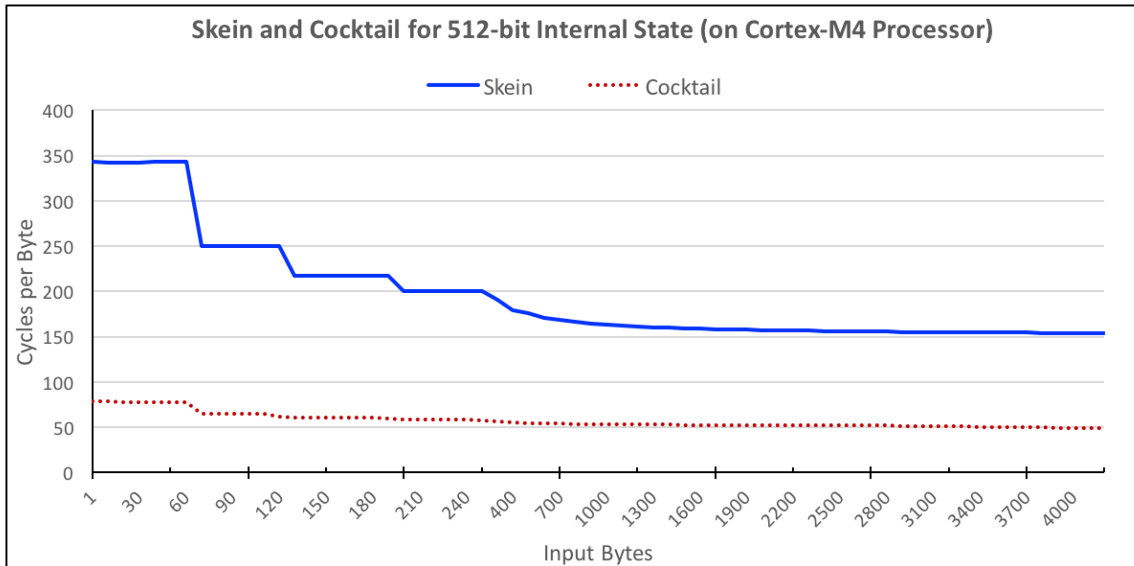


*Figure 72. Performance of Skein and **Cocktail** for 512-bit Internal State on Cortex-M4 Processor*

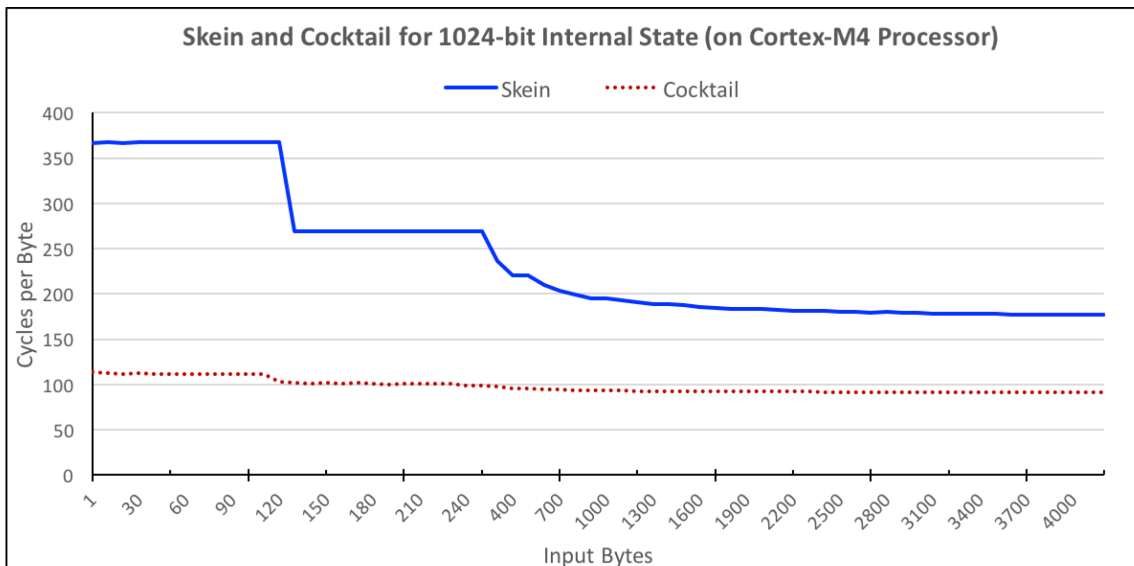


*Figure 73. Performance of Skein and **Cocktail** for 1024-bit Internal State on Cortex-M4 Processor*

c) For Cortex-A8 and x86 machines, considerable spikes in CPB values were observed because these machines are used with multiple other peripherals and run operating systems. Because of this, CPU is interrupted by external devices and

other software that causes considerable spikes. Such spikes are not visible in Cortex-M4.

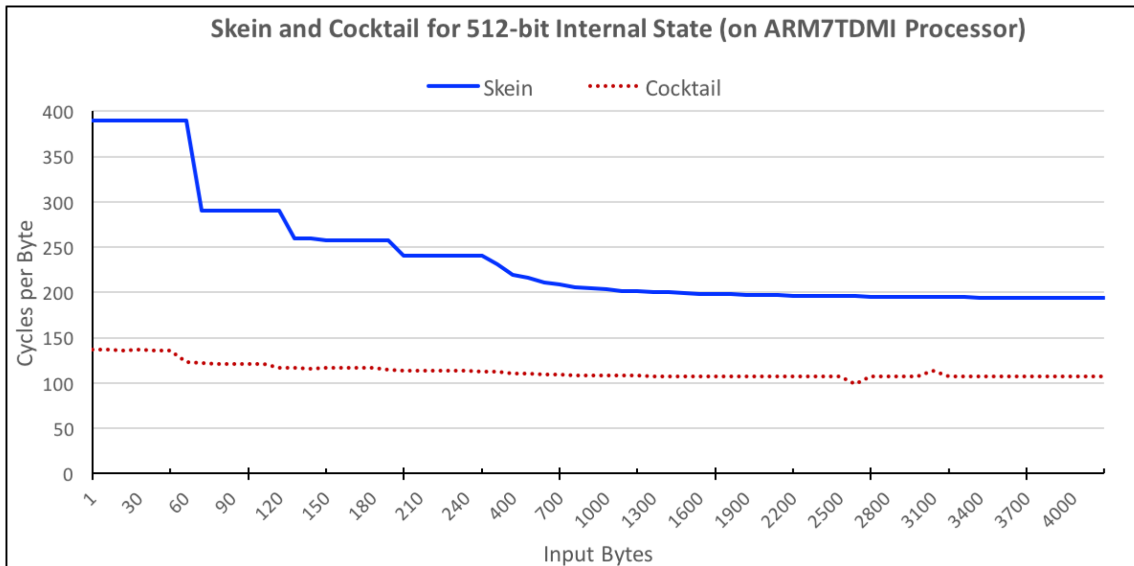


*Figure 74. Performance of Skein and **Cocktail** for 512-bit Internal State on ARM7TDMI Processor*

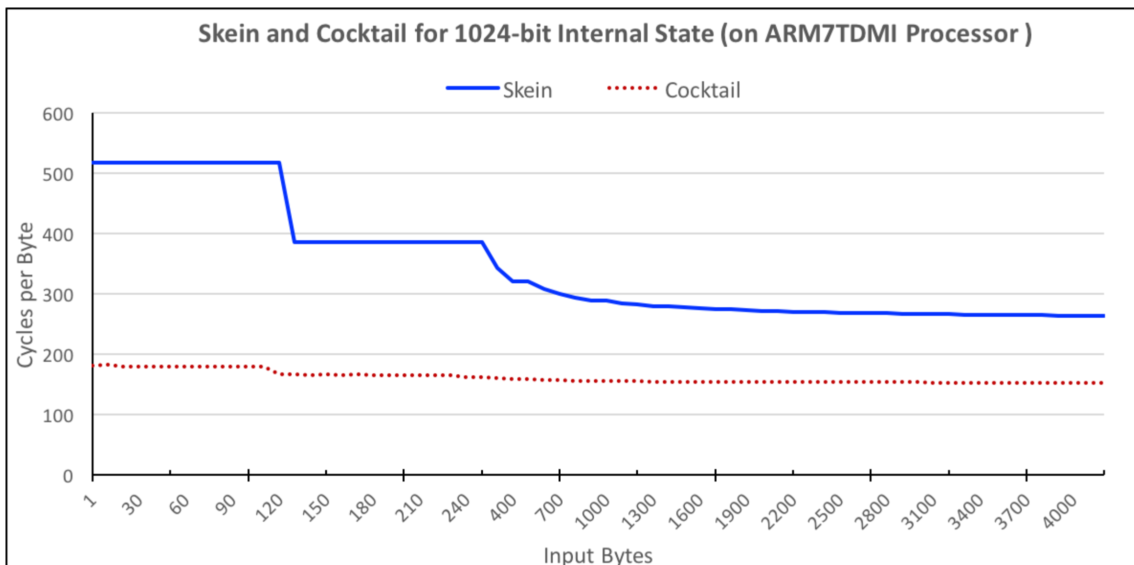


*Figure 75. Performance of Skein and **Cocktail** for 1024-bit Internal State on ARM7TDMI Processor*

d) In all 32-bit machines (x86 or ARM), consumption of CPBs increases considerably as ***Cocktail*** switches from 512-bit internal state (256-bit hash output) to 1024-bit internal state (512-bit hash output). For example, in x86 32-bit architecture this value goes up to 123%. In comparison, Skein performs better on this front and CPB increases by about 40% as we switch from 512-bit internal state to 1024-bit

internal state on x86_32 architectures. Same attribute of Skein was also noticed while comparing it with other SHA-3 finalists in Chapter 3.

e) Cycles per byte consumed by ***Cocktail*** and **Skein** decrease with increase in input size but this is more obvious in **Skein** as compared to ***Cocktail***.

C) Summary

As a whole ***Cocktail*** performs better than **Skein** on Intel x86 architecture as well as ARM architecture. The only scenario where **Skein** marginally performs better than ***Cocktail*** is for 512-bit internal state on a 64-bit machine. In all other cases, ***Cocktail*** **performs much better than Skein** as the consumption of Cycles per Byte is **at least 60% lesser in case of x86 machines** and in ARM architecture this improvement (reduction in CPB) is even better and goes up to **88% for 512-bit internal state on Cortex-A8 machine**.

The above comparison was done with 10 round ***Cocktail-512*** and 12 round ***Cocktail-1024*.** Number of rounds have been kept as tuneable parameter for ***Cocktail***. To keep its diffusion factor same as that of **Skein,** even if the number of rounds of ***Cocktail-512*** and ***Cocktail*-1024** are increased to 12 and 14 respectively, then also ***Cocktail*** performs better than **Skein.** This proves that better performance of ***Cocktail*** is not the result of lesser rounds but is because of its inherent structure. This is discussed in section '5.2.1 Difference in Structure of ***Cocktail*** and Skein' under the sub head 'Comparison of Operations' which reflect that in comparison to **Skein**, ***Cocktail*** even with extended rounds uses **36% lesser operations for 512 bit internal state** and **27% lesser operations for 1024-bit internal state**.

The graphs in Figure 76 and Figure 77 present comparison of performance of **Skein** with ***Cocktail*** when its number of rounds are increased to match diffusion factor of **Skein**. Extended ***Cocktail*** {written as ***Cocktail*** (ER)} in these figures, means rounds of ***Cocktail-512*** increased from 10 to 12; and of ***Cocktail-1024*** increased from 12 to 14. In these figures, ***Cocktail*** with recommended rounds is written as ***Cocktail*** (RR).

It is evident from the graphs that even with extended rounds (shown in grey colour) ***Cocktail*** continues to consume lesser Cycles Per Byte compared to those by **Skein**. Only in case of 64-bit Intel x86 architecture, **Skein-512** performs better than ***Cocktail-512***.

The reason behind better performance of **Skein-512** is the usage of 64-bit word size even for 512-bit hash. However, with 1024-bit internal state, ***Cocktail*** gets better.

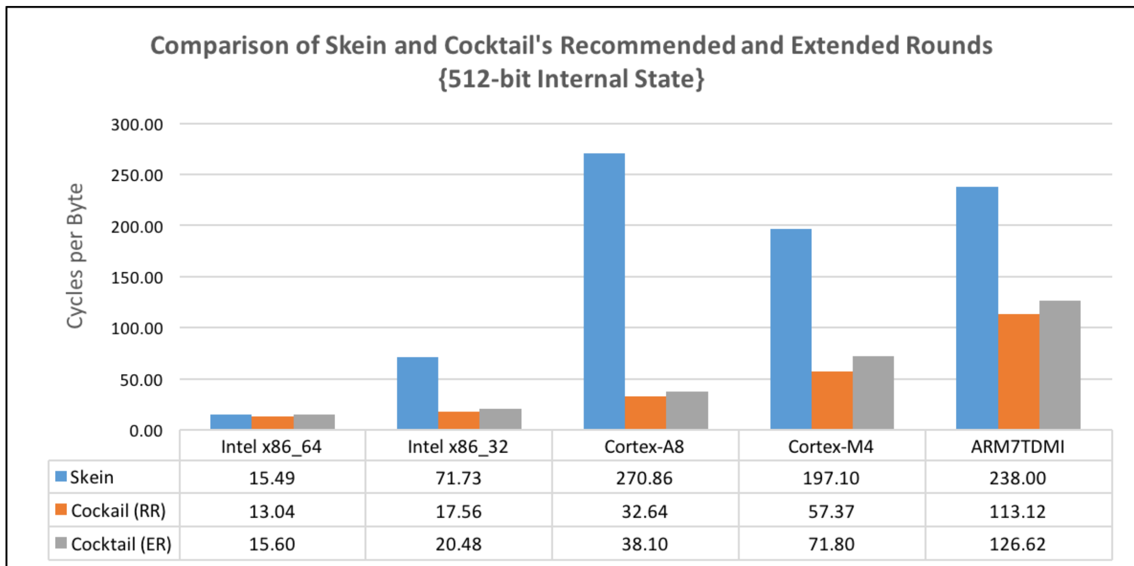


*Figure 76. Comparison of Skein and **Cocktail's** Recommended and Extended Rounds on Different Architectures (512-bit Internal State)*

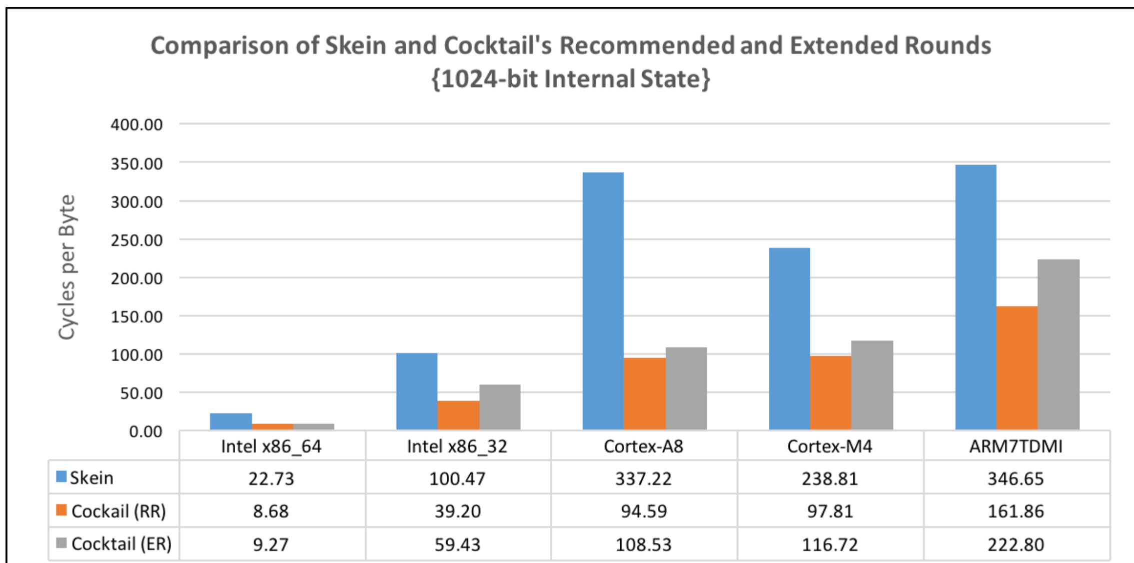


*Figure 77. Comparison of Skein and **Cocktail's** Recommended and Extended Rounds on Different Architectures (1024-bit Internal State)*

5.3 *Cocktail* and SHA-3 final Round Candidate Algorithms

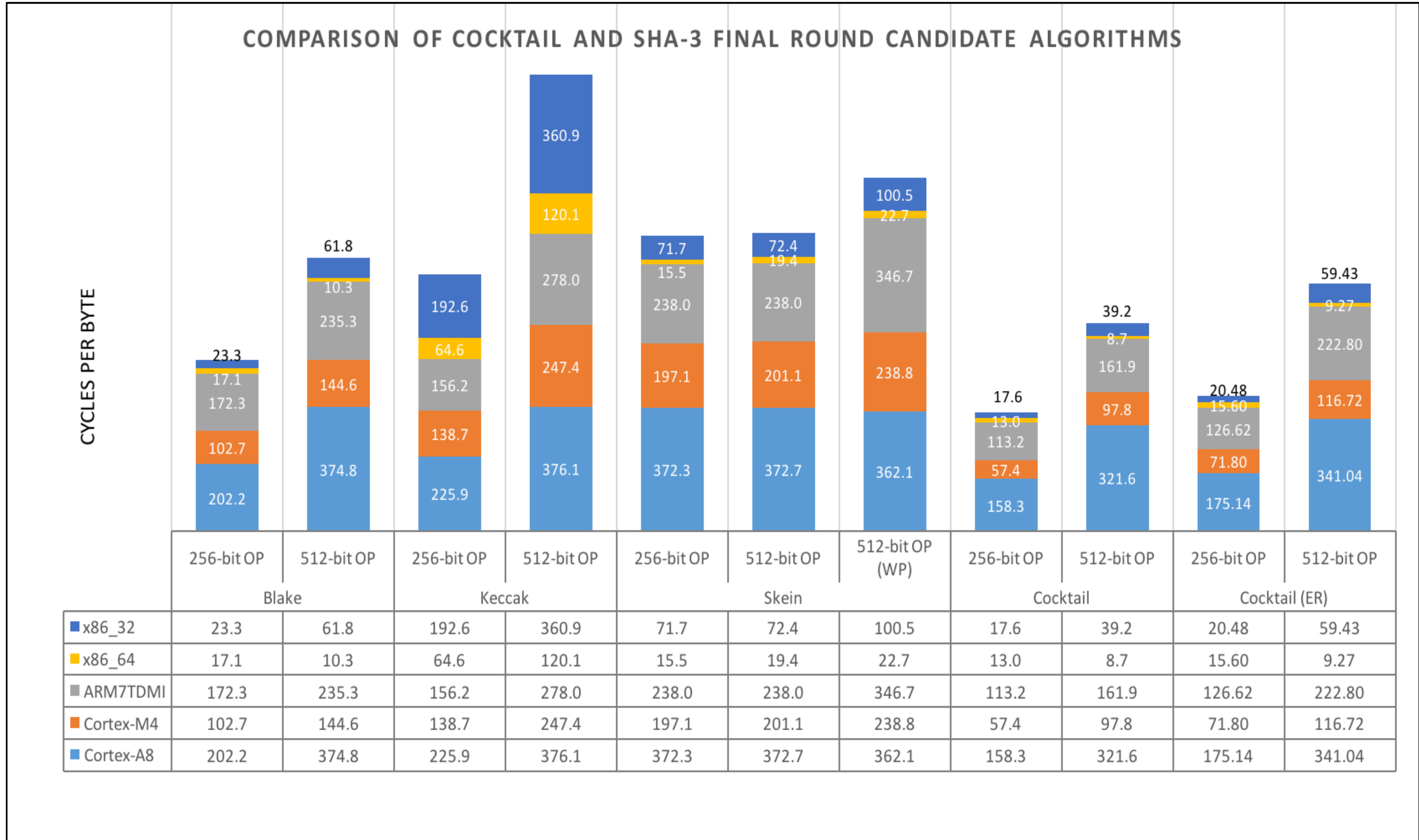
After comparing ***Cocktail*** with Skein in the previous section, this section is devoted to the comparison of ***Cocktail*** with all SHA-3 final round candidate algorithms on various platforms. The comparative values of Cycles per byte consumed by all algorithms on different architectures are presented in Figure 78. As the performance of all algorithms

have been discussed in detail (for varying input sizes) in Chapter 3, this section deals with comparison of average CPB consumed by various algorithms to give a summarized comparison on performance of ***Cocktail*** and all other algorithms. The methodologies and tools used for computing these values are same as mentioned in Chapter 3 for ARM architecture processors and in this chapter under the heading '5.2.2 - A) Comparison on Reference Platform (Intel x86 Architecture)' for Intel x86 architecture.

Figure 78 represents the results for 256-bit hash as well as 512-bit hash. For Skein, results are presented using author's [9] prime submission (Skein-512) as well as using wide pipe for 512-bit hash output {written as Skein (WP) in the figure}. For ***Cocktail***, the presented results are for recommended rounds as well as for the increased rounds {written as ***Cocktail*** (ER) in the figure}. The results of Grøstl and JH are not shown in Figure 78 as the CPBs consumed by these algorithms are too high and that affect the visibility of the CPBs consumed by other algorithms in the figure. These values are given separately in Figure 79.

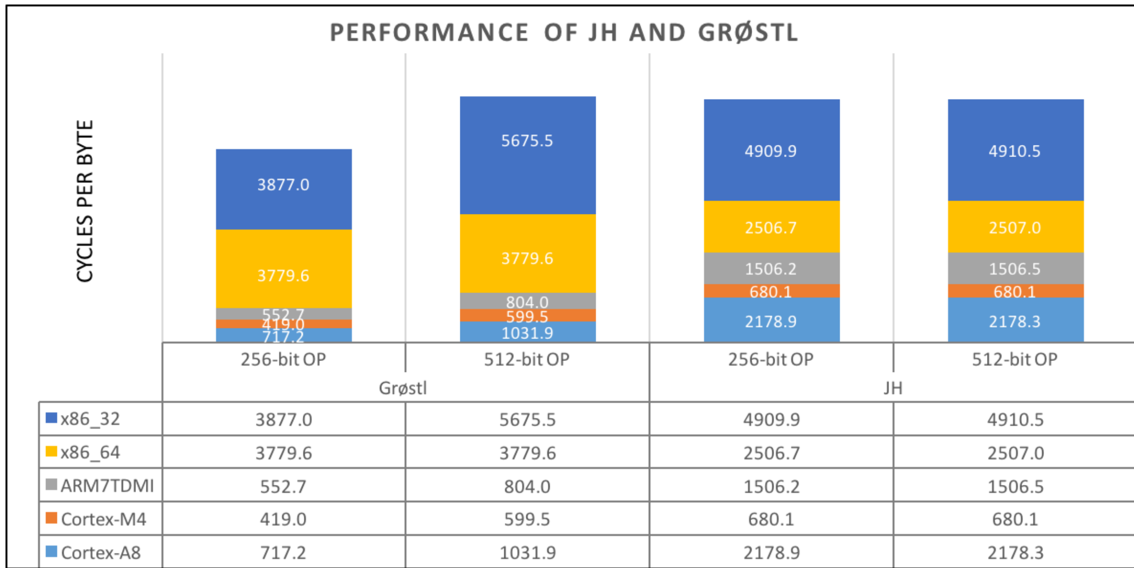
The following observations are drawn from the comparisons:

i. For all algorithms, on 32-bit architectures, as we move from 256-bit hash output to 512-bit hash output for better security margins, the cost in terms of CPB increases. This increase in CPB consumption as per architecture varies from 40% to 170% for Blake, for Keccak it varies from 70% to 90%, for ***Cocktail*** it varies from 40% to 100%, and for Grøstl it is around 40-50%. Only JH and Skein shows consistent performance irrespective of output size. JH otherwise takes considerably high CPB.
ii. For Skein, the rationale behind such performance is not using wide internal state for 512-bit hash output and is detailed in Chapter 3. Figure 79 also presents CPB consumed by Skein for 512-bit hash with wide internal state (i.e. 1024 bit internal state). Results reflect that with wide pipe, Skein does consume more CPB but the increase is not as high as for other algorithms. For Cortex-A8, it does not increase at all but for other 32-bit processors it varies from 20-40 percent.
iii. **On 64-bit x86 Machine: *Cocktail*** runs about **15% faster** than No. 2 performer (Skein and Blake) for 256 as well as 512-bit hash. An extended ***Cocktail*** (with increased round) continues to be the best for 512-bit hash. However, for 256-bit hash, it is at the second position, lagging marginally behind Skein.



	Blake		Keccak		Skein			Cocktail		Cocktail (ER)	
	256-bit OP	512-bit OP	256-bit OP	512-bit OP	256-bit OP	512-bit OP	512-bit OP (WP)	256-bit OP	512-bit OP	256-bit OP	512-bit OP
x86_32	23.3	61.8	192.6	360.9	71.7	72.4	100.5	17.6	39.2	20.48	59.43
x86_64	17.1	10.3	64.6	120.1	15.5	19.4	22.7	13.0	8.7	15.60	9.27
ARM7TDMI	172.3	235.3	156.2	278.0	238.0	238.0	346.7	113.2	161.9	126.62	222.80
Cortex-M4	102.7	144.6	138.7	247.4	197.1	201.1	238.8	57.4	97.8	71.80	116.72
Cortex-A8	202.2	374.8	225.9	376.1	372.3	372.7	362.1	158.3	321.6	175.14	341.04

*Figure 78. Comparison of **Cocktail** and All SHA-3 Final Round Candidate Algorithms*



	256-bit OP (Grøstl)	512-bit OP (Grøstl)	256-bit OP (JH)	512-bit OP (JH)
x86_32	3877.0	5675.5	4909.9	4910.5
x86_64	3779.6	3779.6	2506.7	2507.0
ARM7TDMI	552.7	804.0	1506.2	1506.5
Cortex-M4	419.0	599.5	680.1	680.1
Cortex-A8	717.2	1031.9	2178.9	2178.3

Figure 79. Performance of JH and Grøstl on Various Platforms

iv. **On 32-bit x86 Processor, *Cocktail*** runs about **25% faster** than No. 2 performer for 256-bit hash output and about **35% faster** than No. 2 performer for 512-bit hash output. Increasing number of rounds decreases the difference. With increased rounds, for 256-bit hash, it is still **13%** faster than No. 2 but in case of 512-bit hash it is marginally better by **4%**

v. **On ARM Cortex-A8 Processor, *Cocktail*** retains its best position in terms of speed. It is **fastest** and better than No. 2 position holder by **21% and 11%** for 256 and 512-bit hash respectively. Increase in rounds does not deter the position but difference reduces in 512-bit hash which is about 6%.

vi. **On ARM7TDMI and Cortex-M4** also, ***Cocktail*** comes out to be the fastest among all candidate algorithms. In case of Cortex-M4, it is faster than No. 2 performer by about **44%** and **32%** for 256 and 512-bit hash respectively, and on ARM7TDMI this difference is **27%** and **31%** for 256 and 512-bit hash.

5.4 Concluding Remarks

Our new designed hash function, ***Cocktail,*** is flexible and quite efficient. It performs faster than SHA-3 winner and other SHA-3 final round candidate algorithms on majority of platforms under discussion.

On 32-bit architecture (ARM as well as Intel), *Cocktail* is the fastest among all the candidate algorithms irrespective of whether we use ***Cocktail*** with recommended rounds or increased rounds (increased by 20% for 256-bit hash and 16.6% for 512-bit hash). For

256-bit hash output, the difference remains considerable even with increased rounds but for 512-bit hash, difference in speed is not very considerable with increased rounds.

On Intel 64-bit architecture, Skein's performance for 256-bit hash is quite competitive. With same diffusion factor (increased rounds of Cocktail), **it is faster than *Cocktail*** but for 512-bit hash ***Cocktail*** proves to be the best.

CHAPTER 6: CONCLUSION AND FUTURE ENHANCEMENTS

"An objective achieved makes culmination pleasant"

Anonymous

6.1 Conclusion

This thesis accomplishes two objectives; the first being performance analysis of SHA-3 final round candidate algorithms on a platform other than the 'Reference platform' announced by NIST; and the second is the designing of a new cryptographic hash function that performs better than Skein on both the Reference platform (announced by NIST) and Target platform (selected under the present study for performance evaluation of SHA-3 finalists).

The decision to opt for ARM architecture as Target platform for evaluation of SHA-3 final round candidate algorithms is supported by the rationale provided in Chapter 3. Briefly put, the decision to go with ARM architecture includes inter alia the recent surge in usage of mobile and portable devices while the 'Reference platform' did not cover Embedded and Mobile segment. The market dominance of ARM in Mobile and Embedded segment, supported by its technical capabilities was a major factor that also went into this decision. Cortex-A8 (ARM application series), Cortex-M4 (ARM Embedded series), and ARM7TDMI (classical processor) were picked for evaluation of algorithms under discussion. The performance parameter was Cycles per Byte consumed by an algorithm and results are presented for short and long messages separately.

For **evaluation on Cortex-A8**, OpenBoard-AM335x kit that features Texas Instruments Sitara™ ARM Cortex™ – A8 CPU and runs Linux kernel was utilized. Control Coprocessor CP15 was used to access cycle counts consumed by various algorithms. Methodology involved writing a Kernel module to initialize and access various coprocessor registers. **Evaluation on Cortex-M4** was also done on hardware using Stellaris® LM4F232 Evaluation Board (EK-LM4F232) from Texas Instruments. This machine is a bare machine and does not run any operating system. Code Composer

studio was used to code, debug, burn, and run the code on target machine. To measure cycle counts consumed by various algorithms, DWT's CYCCNT counter was utilized. **Evaluation on ARM7TDMI** was done using IAR Embedded Workbench (simulator) and cycle counts were computed using function profiler and timeline tool.

The results reflect that **JH and Grøstl are quite slow** compared to other three algorithms. Grøstl is at number 4 and JH is at the last position in terms of consumption of Cycles per Byte (CPB). On Cortex-A8, Grøstl consumes at least double (100% increase) and 1.8 times (80% increase) the CPB than No. 3 performer for long and short messages respectively. On same platform JH consumes at least 136% and 73% more CPB than Grøstl for long and short messages respectively. The same trend is visible on Cortex-M4 and ARM7TDMI also. Value of percentage increase in consumption of CPB by Grøstl (compared to No. 3 performer) varies from 47% to 118% and 113% to 220% for Cortex-M4 and ARM7TDMI respectively depending on *hash size* and *input message type* (long or short messages). CPB consumption by JH is 29% to 80% and 58% to 137% more than Grøstl on Cortex-M4 and ARM7TDMI respectively depending on the *hash size*, and *input message type*.

For all ARM processors and all hash output sizes, **Skein, Blake and Keccak have shown good performance**. **Position of the best performer or No. 2 or No. 3 performer changes** with change in *hash size* or *input message type* or *ARM processor* used for evaluation. **On Cortex-A8** and for short messages, Blake and Keccak perform faster than Skein but as message size increases and particularly for 512-bit hash, Skein outperforms the other three. **On Cortex-M4**, Blake stands out as the best but for long messages Skein improves considerably to generate 512-bit hash. **On ARM7TDMI**, Keccak performs better than the other two competitors for 256-bit hash but for 512-bit hash, Skein is the best for long messages whereas Keccak is marginally ahead for short messages.

For almost all algorithms, result of **224-bit and 384-bit hash match with 256-bit and 512-bit hash** respectively. **As input size increases, consumption of Cycles per Byte decreases** for almost all algorithms on all ARM architectures This trend is prominently visible in Skein and Grøstl.

Skein however exhibits an important characteristic that **CPB does not increase as we increase the size of message digest from 224/256 bits to 384/512 bits**. On ARM architecture as a whole, for long messages, this study recommends the use of Skein for

512-bit hash followed by Blake and Keccak for 256-bit hash. Whereas, for Short messages (e.g. in Password hashing, pseudo random number generator, and many such applications) Blake and Keccak are better alternatives.

To accomplish the second objective, a new primitive named **M**odified **C**haCha **C**ore (MCC) was designed that can be used to construct a stream cipher or block cipher or compression function of cryptographic hash function. MCC is an improvisation over Salsa and ChaCha core. An exhaustive experiment was conducted to study the diffusion property of Quarter round of our proposed primitive and its counterparts. The experiment studied all possible rotation distances that result in 1 million permutations for 32-bit word size and 16 million for 64-bit word size. This experiment reflected that MCC creates more diffusion than its counterparts. The Quarter round of MCC, on average, results in gain of 16% over that of ChaCha and 88% over that of Salsa. The performance of Salsa and ChaCha core for different rotation distances were also evaluated and we propose alternative constants (other than the one specified by the author) to have better diffusion. This study reflects that there are 45000 (4.3%) and 58000 (5.5%) set of rotation distances for Salsa and ChaCha respectively that give better results than prescribed value of rotation constants.

A new ARX based hash function named ***Cocktail*** that makes use of **M**odified **C**haCha **C**ore (MCC), to build its compression function and output transformation, is presented. ***Cocktail's*** iterative structure is a variant of Matyas–Mayer–Oseas iteration mode and uses number of message bits hashed so far as one of the input to compression function that makes every call to compression function unique. ***Cocktail*** family consists of two hash functions - ***Cocktail-512*** and ***Cocktail-1024*** - that work on 32-bit and 64-bit word size respectively and internal state is always double the output size. Each round of compression function is a Double round of MCC and sub-key is injected in alternate rounds. The key expansion is quite efficient and uses round dependent constants. ***Cocktail's*** compression function can generate full diffusion in the second round. The whole algorithm can be implemented in about 350 bytes of RAM and additional 96 bytes for constants. Number of rounds is taken as a tuneable parameter. However, to have diffusion factor of more than five, 10 rounds for ***Cocktail-512*** and 12 rounds for ***Cocktail-1024*** are recommended. Decision to have ARX based design makes it quite simple and efficient. Multi-rate padding, fixed initial value, and wide internal state helps

in thwarting multiple generic attacks. The decision to have four column rounds followed by four row rounds gives a lot of scope to exploit parallelism. Being based on familiar constructs that have been analysed considerably, helps in generating more confidence in ***Cocktail*** for its security and efficiency.

Discussion in the last section of Chapter 4 substantiates that ***Cocktail*** is free from any generic attack and no specific attack is anticipated in future. ***Cocktail*** can be used in multiple ways to achieve various security objectives like implementing efficient Digital signature, password hashing, verifying data integrity and authentication as HMAC/CMAC, Key derivation functions, and also to build block and stream cipher. On the performance front, ***Cocktail*** can generate 256-bit message digest with average speed of 13 Cycles per Byte and 512-bit message digest with average speed of 8.7 Cycles per byte on Intel x86_64 Architecture.

Though our attempt was to design a variant that could perform better than Skein family, the new designed hash function ***Cocktail*** outperforms Skein as well as all SHA-3 finalists including 'Keccak', the winner announced by NIST. The comparison with Skein and other SHA-3 final round candidate algorithms reflects that, **on 32-bit architecture (ARM as well Intel)**, ***Cocktail*** is fastest among all the candidate algorithms irrespective of whether ***Cocktail*** is used with recommended rounds or increased rounds (increased by 20% for 256-bit hash and about 17% for 512-bit hash). With recommended rounds, ***Cocktail*** is 11% to 45% faster than SHA-3 final round candidate algorithms depending on *hash size* and *processor architecture used*. With increased rounds, this margin reduces and varies between 4% to 23%.

On Intel 64-bit architecture, Skein's performance for 256-bit hash is quite competitive. With same diffusion factor (increased rounds of ***Cocktail***), Skein is marginally faster (15.49 CPB compared to 15.60 CPB) than ***Cocktail,*** but for 512-bit hash ***Cocktail*** is the best with 8.7 CPB (16% better than No. 2 performer).

6.2 Future Enhancements

Although the researcher has put in his best efforts on the present study, still cryptographic hash functions have wide scope for further research. Like a stepping stone, this research has enough scope for further enhancements. Thus, for future research and in

light of the results and conclusions of this study, the following future work may be carried out:

a) Extending ***Cocktail*** so that it can generate hash output of any size. Presently it generates output up to 256 bits in steps of 32 and from 256 to 512 bits in steps of 64.
b) Improving key-generation algorithm of ***Cocktail*** to reduce CPB consumption further.
c) Implementation of ***Cocktail*** on various platforms including FPGA, ASIC and 8-bit microcontroller and evaluation of its performance in comparison to other prevalent algorithms as done in [192] and [193] for FPGAs.
d) Conducting further experiments with MCC and analyse how it performs.
e) Designing stream and block cipher from existing or updated MCC core.
f) Broadening the sphere of cryptanalysis of ***Cocktail*** to open new doors to further improve and enhance it in emerging scenarios.

BIBLIOGRAPHY

[1] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons, 2007.

[2] D. Kahn, The Codebreakers, Weidenfeld and Nicolson, 1974.

[3] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory,* vol. 22, no. 5, pp. 644-654, November 1976.

[4] B. V. Rompay, Analysis and Design of Cryptographic Hash Functions, MAC Algorithms and Block Ciphers (Ph.D. Thesis), Lauven, Flanders: Katholieke Universiteit Leuven, 2004.

[5] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), "Advanced Encryption Standard (AES), Federation Information Processing Standards Publication (FIPS PUB) 197," NIST Computer Security Publications, 2001.

[6] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), "Secure Hash Standard, Federal Information Processing Standards Publication (FIPS PUB) 180-2," NIST Computer Security Publications, 2002.

[7] R. Sobti and G. Ganesan, "Cryptographic Hash Functions: A Review," *IJCSI International Journal of Computer Science,* vol. 9, pp. 461-479, 2012.

[8] P. S. Barreto and V. Rijmen, "The WHIRLPOOL Hashing Function," in *Final report of European project number IST-1999-12324, named New European Schemes for Signatures, Integrity, and Encryption(NESSIE)*, Springer-Verlag, 2004, pp. 563-572.

[9] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas and J. Walker, "The Skein Hash Function Family," 1 October 2010. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html. [Accessed 23 January 2011].

[10] R. Rivest, "The MD4 Message Digest Algorithm," in *Advances in Cryptology - CRYPTO'90 Proceedings*, vol. 537 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1991, pp. 303-311.

[11] R. Rivest, "The MD5 Message Digest Algorithm," Internet Engineering Task Force, April 1992. [Online]. Available: http://www.faqs.org/rfcs/rfc1321.html. [Accessed 18 July 2011].

[12] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), "Secure Hash Standard, Federation Information Processing Standards Publication (FIPS PUB) 180-4," NIST Computer Security Publications, 2015.

[13] R. C. Merkle, Secrecy, Authentication and Public Key Systems (Ph.D. Thesis), Stanford, California: Stanford University, 1979.

[14] R. C. Merkle, "One Way Hash Functions and DES," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, vol. 435 of the series Lecture Notes in Computer Science, Springer New York, 1990, pp. 428-446.

[15] W. Tuchman, "A Brief History of the Data Encryption Standard," in *Internet Besieged*, D. E. Denning and P. J. Denning, Eds., New York, NY: ACM Press/Addison-Wesley Publishing Co., 1998, pp. 275-280.

[16] Information Society Technology (IST) Programme of the European Commision, "NESSIE - New European Schemes for Signatures, Integrity, and Encryption," [Online]. Available: https://www.cosic.esat.kuleuven.be/nessie/. [Accessed 21 June 2012].

[17] ECRYPT - European Network of Excellence for Cryptology, Information Societies Technology (IST) Programme of European Commision, "The eSTREAM Project," 8 September 2008. [Online]. Available: http://www.ecrypt.eu.org/stream/project.html. [Accessed 11 December 2011].

[18] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), "The SHA-3 Cryptographic Hash Algorithm Competition," [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/. [Accessed 12 January 2010].

[19] G. Bertoni, J. Daemen, P. Michaël and G. V. Assche, "The Keccak reference," 14 January 2011. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html. [Accessed 22 January 2011].

[20] "Password Hashing Competition," [Online]. Available: https://password-hashing.net. [Accessed 15 February 2014].

[21] D. J. Bernstein, "CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness," 02 September 2015. [Online]. Available: http://competitions.cr.yp.to/caesar.html. [Accessed 04 Dec 2015].

[22] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), "Secure Hash Standard, Federation Information Processing Standards Publication (FIPS PUB) 180," NIST Computer Security Publications, 1993.

[23] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), "Secure Hash Standard, Federation Information Processing Standards Publication (FIPS PUB) 180-1," NIST Computer Security Publications, 1995.

[24] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlaffer and S. S. Thomsen, "Grøstl – a SHA-3 candidate," 2 March 2011. [Online].

Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html. [Accessed 12 May 2011].

[25] H. Wu, “The Hash Function JH,” 16 January 2011. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html. [Accessed 12 March 2011].

[26] J. P. Aumasson, L. Henzen, W. Meier and R. C.-W. Phan, “SHA-3 Proposal BLAKE,” 16 July 2010. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html. [Accessed 22 January 2011].

[27] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), “Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family,” 02 November 2007. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf. [Accessed 12 January 2010].

[28] D. Khovratovich, I. Nikoli´c and C. Rechberger, “Rotational Rebound Attacks on Reduced Skein,” *Cryptology ePrint Archive, Report 2010/538,* 2010.

[29] D. Khovratovich and I. Nikolić, “Rotational Cryptanalysis of ARX,” in *Fast Software Encryption : 17th International Workshop, FSE 2010*, vol. 6147 of the series Lecture Notes in Computer Science, Seoul, Springer Berlin Heidelberg, 2010, pp. 333-346.

[30] Q. Dang and T. Polk, “SHA-3 for Internet Protocols,” in *IETF-83 Proceedings*, 2012.

[31] G. Tsudik, “Message authentication with one-way hash functions,” *ACM SIGCOMM Computer Communication Review,* vol. 22, no. 5, pp. 29-38, 1992.

[32] M. Bellare, R. Canetti and H. Krawczyk, “Keying Hash Functions for Message Authentication,” in *Advances in Cryptology — CRYPTO ’96, 16th Annual International Cryptology Conference Proceedings*, vol. 1109 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 1996, pp. 1-15.

[33] R. L. Rivest, A. Shamir and L. Adleman, “A Method for Obtaining Digital Signatures and Public-key Cryptosystems,” *Communications of the ACM,* vol. 21, no. 2, pp. 120-126, 1978.

[34] S. Singh, The Code Book : The Evolution of Secrecy from Mary, Queen of Scots to Quantum Cryptography, New York: Doubleday, 1999.

[35] W. Stallings, Cryptography and Network Security Principles and Practices, Prentice Hall, 2005.

[36] A. Biryukov, D. Dinu and D. Khovratovich, “Argon2: The Memory-hard Function for Password Hashing and Other Applications,” 1 October 2015. [Online]. Available: https://password-hashing.net/argon2-specs.pdf. [Accessed 01 December 2015].

[37] G. Hatzivasilis, I. Papaefstathiou and C. Manifavas, “Password Hashing Competition - Survey and Benchmark,” *Cryptology ePrint Archive, Report 2015/265,* 2015.

[38] S. Haber and S. W. Stornetta, “How to Time-stamp a Digital Document,” *Journal of Cryptology,* vol. 3, no. 2, pp. 99-111, January 1991.

[39] M. Bellare, R. Canetti and H. Krawczyk, “Pseudorandom Functions Revisited: The Cascade Construction and Its Concrete Security,” in *Proceedings of 37th Annual Symposium on Foundations of Computer Science, 1996.*, IEEE, 1996, pp. 514-523.

[40] I. Haitner, D. Harnik and O. Reingold, “Efficient Pseudorandom Generators from Exponentially Hard One-Way Functions,” in *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006 Proceedings, Part-II*, vol. 4052 of the series Lecture Notes in Computer Science, Venice, Springer Berlin Heidelberg, 2006, pp. 228-239.

[41] S. M. Matyas, A. V. Le and D. Abraham, “A Key-management Scheme Based on Control Vectors,” *IBM Systems Journal,* vol. 30, no. 2, p. 175, 1991.

[42] H. Handschuh and D. Naccache, “SHACAL-2,” in *Final report of European project number IST-1999-12324, named New European Schemes for Signatures, Integrity, and Encryption(NESSIE)*, Springer-Verlag, 2004, pp. 555-559.

[43] D. Armstrong, “An Introduction to File Integrity Checking on Unix Systems,” GIAC practical repository, SANS Institute, 2003. [Online]. Available: http://www.giac.org/paper/gcux/188/introduction-file-integrity-checking-unix-systems/104739.

[44] X. Lai and J. L. Massey, “Hash Functions Based on Block Ciphers,” in *Advances in Cryptology — EUROCRYPT’ 92, Workshop on the Theory and Application of Cryptographic Techniques, 1992 Proceedings*, vol. 658 of the series Lecture Notes in Computer Science, Balatonfüred, Springer Berlin Heidelberg, 1993, pp. 55-70.

[45] I. B. Damgård, “A Design Principle for Hash Functions,” in *Advances in Cryptology — CRYPTO’ 89 Proceedings*, vol. 435 of the series Lecture Notes in Computer Science, Springer New York, 1990, pp. 416-427.

[46] A. Joux, “Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions,” in *Advances in Cryptology – CRYPTO 2004, 24th Annual International Cryptology Conference, Proceedings*, vol. 3152 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 2004, pp. 306-316.

[47] J. Kelsey and T. Kohno, “Herding Hash Functions and the Nostradamus Attack,” in *Advances in Cryptology - EUROCRYPT 2006, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 4004 of the series Lecture Notes in Computer Science, St. Petersburg, Springer Berlin Heidelberg, 2006, pp. 183-200.

[48] Y. Dodis, T. Ristenpart and T. Shrimpton, "Salvaging Merkle-Damgård for Practical Applications," in *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 5479 of the series Lecture Notes in Computer Science, Cologne, Springer Berlin Heidelberg, 2009, pp. 371-388.

[49] S. Lucks, "Design Principles for Iterated Hash Functions," *IACR Cryptology ePrint Archive, Report 2004/253,* p. 253, 29 September 2004.

[50] P. Gauravaram, Cryptographic Hash Functions: Cryptanalysis, Design and Applications (Ph.D. Thesis), Brisbane, Queensland: Queensland University of Technology, 2007.

[51] E. Biham and O. Dunkleman, "A Framework for Iterative Hash Functions - HAIFA," *Cryptology ePrint Archive, Report 2007/278,* 2006.

[52] N. Nandi and S. Paul, "Speeding Up The Wide-pipe: Secure and Fast Hashing," *Cryptology ePrint Archive, Report 2010/193,* 2010.

[53] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, "On the Indifferentiability of the Sponge Construction," in *Advances in Cryptology – EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques,Proceedings* , vol. 965 of the series Lecture Notes in Computer Science, Istanbul, Springer Berlin Heidelberg, 2008, pp. 181-197.

[54] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, "Sponge Functions," in *ECRYPT hash workshop*, vol. 2007, 2007.

[55] G. Bertoni, D. Joan, M. Peeters and G. Van Assche, "Cryptographic Sponges," [Online]. Available: http://sponge.noekeon.org. [Accessed 23 January 2011].

[56] M. Bellare and T. Ristenpart, "Multi-Property-Preserving Hash Domain Extension and the EMD Transform," in *Advances in Cryptology – ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Proceedings*, vol. 4284 of the series Lecture Notes in Computer Science, Shanghai, Springer Berlin Heidelberg, 2006, pp. 299-314.

[57] S. Halevi and H. Krawczyk, "Strengthening Digital Signatures Via Randomized Hashing," in *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Proceedings*, vol. 4117 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 2006, pp. 41-59.

[58] E. Andreeva, G. Neven, B. Preneel and T. Shrimpton, "Seven-properties-preserving Iterated Hashing: The RMC construction," *ECRYPT document STVL4-KUL15-RMC-1.0, private communications,* 2006.

[59] E. Andreeva, G. Neven, B. Preneel1 and T. Shrimpton, "Seven-Property-Preserving Iterated Hashing: ROX," *Cryptology ePrint Archive, Report 2007/176,* 2007.

[60] P. Rogaway and T. Shrimpton, "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance," in *Fast Software Encryption, 11th International Workshop, FSE 2004, Revised Papers*, vol. 3017 of the series Lecture Notes in Computer Science, New Delhi, Springer Berlin Heidelberg, 2004, pp. 371-388.

[61] B. A. Forouzan and D. Mukhopadhyay, Cryptography and Network Security, Tata McGraw Hill Education Private Limited, 2nd Edition.

[62] A. F. Webster and S. E. Tavares, "On the Design of S-Boxes," in *Advances in Cryptology — CRYPTO '85 Proceedings*, vol. 218 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1986, pp. 523-534.

[63] I. Mironov, "Hash functions: Theory, attacks, and applications," *Microsoft Research, Silicon Valley Campus,* pp. 1-22, 14 November 2005.

[64] B. d. Boer and A. Bosselaers, "Collisions for the Compression Function of MD5," in *Advances in Cryptology — EUROCRYPT '93, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 765 of the series Lecture Notes in Computer Science, Lofthus, Springer Berlin Heidelberg, 1994, pp. 293-304.

[65] L. Knudsen, Block Ciphers: Analysis, Design and Applications (Ph.D. Thesis), Aarhus: Aarhus University, 1994.

[66] O. Mikle, "Practical Attacks on Digital Signatures Using MD5 Message Digest," *Cryptology ePrint Archive, Report 2004/356,* 2004.

[67] H. Dobbertin, "Cryptanalysis of MD5 compress," *German Information Security Agency,* 1996.

[68] X. Wang, D. Feng, X. Lai and H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD," *Cryptology ePrint Archive, Report 2004/199,* 2004.

[69] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," in *Advances in Cryptology – EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 3494 of the series Lecture Notes in Computer Science, Aarhus, Springer Berlin Heidelberg, 2005, pp. 19-35.

[70] M. Bellare and T. Kohno, "Hash Function Balance and Its Impact on Birthday Attacks," in *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 3027 of the series Lecture Notes in Computer Science , Interlaken, Springer Berlin Heidelberg, 2004, pp. 401-418.

[71] B. Kaliski and M. Robshaw, "Message Authentication with MD5," *CryptoBytes (RSA Labs Technical Newsletter),* vol. 1, no. 1, 1995.

[72] T. Duong and J. Rizzo, “Flickr's API Signature Forgery Vulnerability,” 2009. [Online]. Available: http://netifera.com/research/flickr_api_signature_forgery.pdf. [Accessed 12 April 2010].

[73] R. D. Dean, Formal Aspects of Mobile Code Security (Ph.D. Thesis), Princeton: Princeton University, 1999.

[74] J. Kelsey and B. Schneier, “Second Preimages on n-bit Hash Functions for Much Less than 2^n Work,” in *Advances in Cryptology – EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 3494 of the series Lecture Notes in Computer Science, Aarhus, Springer Berlin Heidelberg, 2005, pp. 474-490.

[75] B. Preneel, “Cryptographic Primitives for Information Authentication - State of the Art,” in *State of the Art in Applied Cryptography - Course on Computer Security and Industrial Cryptography*, B. Preneel and V. Rijmen, Eds., Leuven, Springer, 1997.

[76] V. G. Bard, “The Fixed-Point Attack,” in *Algebraic Cryptanalysis*, Springer, 2009, pp. 17-28.

[77] S. Bakhtiari, R. Safavi-Naini and J. Pieprzyk, “Cryptographic Hash Functions: A Survey,” *Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Australia,* 1995.

[78] X. Wang, X. Lai, D. Feng and H. Chen , “Cryptanalysis of the Hash Functions MD4 and RIPEMD,” in *Advances in Cryptology – EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 3494 of the series Lecture Notes in Computer Science, Aarhus, Springer Berlin Heidelberg, 2005, pp. 1-18.

[79] X. Wang, H. Yu and Y. L. Yin, “Efficient Collision Search Attacks on SHA-0,” in *Advances in Cryptology – CRYPTO 2005, 25th Annual International Cryptology Conference, Proceedings*, vol. 3621 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 2005, pp. 1-16.

[80] X. Wang, Y. L. Yin and H. Yu, “Finding Collisions in the Full SHA-1,” in *Advances in Cryptology – CRYPTO 2005, 25th Annual International Cryptology Conference, Proceedings*, vol. 3621 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, pp. 17-36.

[81] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet and W. Jalby, “Collisions of SHA-0 and Reduced SHA-1,” in *Advances in Cryptology – EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 3494 of the series Lecture Notes in Computer Science, Aarhus, Springer Berlin Heidelberg, 2005, pp. 36-57.

[82] E. Biham and A. Shamir, “Differential Cryptanalysis of DES-like Cryptosystems,” in *Advances in Cryptology-CRYPT0’ 90 Proceedings*, vol. 537 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1991, pp. 2-21.

[83] E. Biham and A. Shamir, “Differential Cryptanalysis of FEAL and N-Hash,” in *Advances in Cryptology — EUROCRYPT ’91, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 547 of the series Lecture Notes in Computer Science, Brighton, Springer Berlin Heidelberg, 1991, pp. 1-16.

[84] E. Biham and A. Shamir, “Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer,” in *Advances in Cryptology — CRYPTO ’91 Proceedings*, vol. 576 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1992, pp. 156-171.

[85] M. Matsui, “Linear Cryptanalysis methods for DES Cipher,” in *Advances in Cryptology — EUROCRYPT ’93, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 765 of the series Lecture Notes in Computer Science, Lofthus, Springer Berlin Heidelberg, 1994, pp. 386-397.

[86] H. M. Heys, “A Tutorial on Linear and Differential Cryptanalysis,” *Cryptologia,* vol. 26, no. 3, pp. 189-221, 2002.

[87] S. Miyaguchi, K. Ohta and M. Iwata, “Confirmation that some Hash Functions are not Collisions Free,” in *Advances in Cryptology — EUROCRYPT ’90, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 473 of the series Lecture Notes in Computer Science, Aarhus, Springer Berlin Heidelberg, 1991, pp. 326-343.

[88] US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), “Data Encryption Standard (DES), Federation Information Processing Standards Publication (FIPS PUB) 46-3,” 1999.

[89] B. Preneel, R. Govaerts and J. Vandewalle, “Differential cryptanalysis of hash functions based on block ciphers,” in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ACM, 1993, pp. 183-188.

[90] V. Rijmen and B. Preneel, “Improved characteristics for differential cryptanalysis of hash functions based on block ciphers,” in *Fast Software Encryption, Second International Workshop, Proceedings*, vol. 1008 of the series Lecture Notes in Computer Science, Leuven, Springer Berlin Heidelberg, 1995, pp. 242-248.

[91] B. Preneel, R. Govaerts and J. Vandewalle, “Hash Functions Based on Block Ciphers: A Synthetic Approach,” in *Advances in Cryptology — CRYPTO’ 93, 13th Annual International Cryptology Conference, Proceedings*, vol. 773 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 1994, pp. 368-378.

[92] J. Black, P. Rogaway and T. Shrimpton, “Black-box Analysis of the Block-cipher-based Hash Function Constructions from PGV,” in *Advances in Cryptology — CRYPTO 2002, 22nd Annual International Cryptology Conference, Proceedings*, vol. 2442 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 2002, pp. 320-335.

[93] S. M. Matyas, C. H. Meyer and J. Oseas, “Generating Strong One-way Functions with Cryptographic Algorithm,” *IBM Technical Disclosure Bulletin,* vol. 27, no. 10A, pp. 5658-5659, 1985.

[94] S. Miyaguchi, M. Iwata and K. Ohta, “New 128-bit Hash Function,” in *4th International Joint Workshop on Computer Communications, Proceedings* , Tokyo, 1989, pp. 279-288.

[95] B. Preneel, R. Govaerts and J. Vandewalle, “Cryptographically Secure Hash Functions: An Overview,” ESAT Internal Report, KU Leuven, 1989.

[96] J.-J. Quisquater and M. Girault, “2n-Bit Hash-Functions Using n-Bit Symmetric Block Cipher Algorithms,” in *Advances in Cryptology — EUROCRYPT ’89, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 434 of the series Lecture Notes in Computer Science, Houthalen, Springer Berlin Heidelberg, 1990, pp. 102-109.

[97] R. S. Winternitz, “Producing a One-Way Hash Function from DES,” in *Advances in Cryptology, Proceedings of Crypto 83*, Springer US, 1984, pp. 203-207.

[98] B. O. Brachtl, D. Coppersmith, M. M. Hyden, S. M. Matyas Jr, C. H. Meyer, J. Oseas, S. Pilpel and M. Schilling, “Data Authentication Using Modification Detection Codes Based on a Public One Way Encryption Function”. Patent U.S. Patent No. 4,908,861, 13 March 1990.

[99] C. H. Meyer and M. Schilling, “Secure Program Load with Manipulation Detection Code,” in *Proceedings Securicom*, vol. 88, 1988, pp. 111-130.

[100] X. Lai, On the Design and Security of Block Ciphers (Dissertation - Doctor of Technical Sciences), Zurich: SWISS FEDERAL INSTITUTE OF TECHNOLOGY, 1992.

[101] B. Preneel, A. Bosselaers, R. Govaerts and J. Vandewalle, “Collision-Free Hash Functions Based on Block Cipher Algorithms,” in *Proceedings of the 1989 Carnahan Conference on Security Technology*, 1989.

[102] W. Hohl, X. Lai, T. Meier and C. Waldvogel, “Security of Iterated Hash Functions Based on Block Ciphers,” in *Advances in Cryptology — CRYPTO’ 93, 13th Annual International Cryptology Conference, Proceedings*, vol. 773 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 1994, pp. 379-390.

[103] Center for Information Protection and Special Communications of the Federal Security Service of the Russian Federation, *GOST R 34.11-2012, Information technology. Cryptographic Data Security. Hashing function,* Federal Agency on Technical Regulating and Metrology, 2012.

[104] ISO N179, *AR Fingerprint Function,* ISO-IEC/JTC1/SC27/WG2, International Organization for Standardization, 1992.

[105] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL), "Secure Hash Standard, Federation Information Processing Standards Publication (FIPS PUB) 180-3," NIST Computer Security Publications, 2008.

[106] H. Dobbertin, A. Bosselaers and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD," in *Fast Software Encryption, Third International Workshop, Proceedings*, vol. 1039 of the series Lecture Notes in Computer Science, Cambridge, Springer Berlin Heidelberg, pp. 71-82.

[107] Y. Zheng, J. Pieprzyk and J. Seberry, "HAVAL — A one-way hashing algorithm with variable length of output (extended abstract)," in *Advances in Cryptology — AUSCRYPT '92, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, Gold Coast, Queensland: Springer Berlin Heidelberg, 1993, pp. 81-104.

[108] R. C. Merkle, "A Fast Software One-way Hash Function," *Journal of Cryptology,* vol. 3, no. 1, pp. 43-58, 1990.

[109] R. Anderson and E. Biham, "Tiger — A Fast New Hash Function," in *Fast Software Encryption, Third International Workshop, Proceedings*, vol. 1039 of the series Lecture Notes in Computer Science, Cambridge, Springer Berlin Heidelberg, pp. 89-97.

[110] E. Biham, "New techniques for Cryptanalysis of hash functions and improved attacks on Snefru," in *Fast Software Encryption, 15th International Workshop, FSE 2008, Revised Selected Papers*, vol. 5086 of the series Lecture Notes in Computer Science, Lausanne, Springer Berlin Heidelberg, 2008, pp. 444-461.

[111] P. Camion and J. Patarin , "The Knapsack Hash Function proposed at Crypto'89 can be broken," in *Advances in Cryptology — EUROCRYPT '91, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 547 of the series Lecture Notes in Computer Science, Brighton, Springer Berlin Heidelberg, 1991, pp. 39-53.

[112] A. Joux and L. Granboulan, "A Practical Attack against Knapsack based hash functions," in *Advances in Cryptology — EUROCRYPT'94, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 950 of the series

Lecture Notes in Computer ScienceS, Perugia, Springer Berlin Heidelberg, 1995, pp. 58-66.

[113] S. Wolfram, "Cryptology with Cellular Automata," in *Advances in Cryptology — CRYPTO '85 Proceedings*, vol. 218 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1986, pp. 429-432.

[114] J. Daemen, R. Govaerts and J. Vandewalle, "A framework for the design of one-way hash functions including cryptanalysis of Damgård's one-way function based on a cellular automaton," in *Advances in Cryptology — ASIACRYPT '91, International Conference on the Theory and Application of Cryptology, Proceedings*, vol. 739 of the series Lecture Notes in Computer Science, Fujiyosida , Springer Berlin Heidelberg, 1993, pp. 82-96.

[115] C. P. Schnorr, "FFT-Hash II, Efficient Cryptographic Hashing," in *Advances in Cryptology — EUROCRYPT' 92, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 658 of the series Lecture Notes in Computer Science, Balatonfüred, Springer Berling Heidelberg, 1993, pp. 45-54.

[116] C. P. Schnorr and S. Vaudenay , "Parallel FFT-hashing," in *Fast Software Encryption, Cambridge Security Workshop, Proceedings*, vol. 809 of the series Lecture Notes in Computer Science, Cambridge, Springer Berlin Heidelberg, 1994, pp. 149-156.

[117] J. Daemen, A. Bosselaers, R. Govaerts and J. Vandewalle, "Collisions for Schnorr's hash function FFT-Hash presented at Crypto '91," in *Advances in Cryptology — ASIACRYPT '91, International Conference on the Theory and Application of Cryptology, Proceedings*, vol. 739 of the series Lecture Notes in Computer Science, Fujiyosida, Springer Berlin Heidelberg, 1993, pp. 477-480.

[118] S. Vaudenay, "FFT-Hash II is not yet Collision Free," in *Advances in Cryptology — CRYPTO' 92, 12th Annual International Cryptology Conference, Proceedings*, vol. 740 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 1993, pp. 587-593.

[119] F. Chabaud and A. Joux, "Differential collisions in SHA-0," in *Advances in Cryptology — CRYPTO '98, 18th Annual International Cryptology Conference, Proceedings*, vol. 1462 of the series Lecture Notes in Computer Science, Santa Barbara, California: Springer Berlin Heidelberg, 1998, pp. 56-71.

[120] J. Kesley, "SHA3: Past, Present, and Future," in *Workshop on Cryptographic Hardware and Embedded Systems CHES 2013 (Invited Talk)*, Santa Barbara, USA, 2013.

[121] E. Biham and R. Chen, "Near-Collisions of SHA-0," in *Advances in Cryptology – CRYPTO 2004, 24th Annual International Cryptology Conference, Proceedings* , vol. 3152 of the series Lecture Notes in Computer Science , Santa Barbara, California: Springer Berlin Heidelberg, 2004, pp. 290-305.

[122] W. E. Burr, "Cryptographic Hash Standards: Where do we go from here?," *Security & Privacy, IEEE,* vol. 4, no. 2, pp. 88-91, 2006.

[123] J. J. Hoch and A. Shamir, "Breaking the ICE – Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions," in *Fast Software Encryption, 13th International Workshop, FSE 2006, Revised Selected Papers*, vol. 4047 of the series Lecture Notes in Computer Science, Graz, Springer Berlin Heidelberg, 2006, pp. 179-194.

[124] Y. Sasaki, L. Wang and K. Aoki, "Preimage Attacks on 41-Step SHA-256 and 46-Step SHA-512," *Cryptology ePrint Archive, Report 2009/479,* 2009.

[125] Q. Dang, "SHA-3 Update," in *IETF 86*, 2013.

[126] B. Burr, "SHA3 Where we've been, where we're Going (update to RSA 2013 talk)," in *DIMACS Workshop on Current Trends in Cryptography*, 2013.

[127] Q. Dang, "NIST Draft FIPS 202: SHA-3 Permutation- Based Hash Standard-Status," in *IETF 87*, 2013.

[128] US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, DRAFT FIPS PUB 202," 2014.

[129] "NIST policy on Hash functions," [Online]. Available: http://csrc.nist.gov/groups/ST/hash/policy.html. [Accessed August 2015].

[130] ICT Data and Statistics Division, Telecommunication Development Bureau, International Telecommunication Union (ITU), "ICT Facts and Figures - The World in 2015," May 2015. [Online]. Available: https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf. [Accessed 20 June 2015].

[131] M. Meeker, "Internet Trends 2015 - Code Conference," 27 May 2015. [Online]. Available: http://www.kpcb.com/internet-trends. [Accessed 20 June 2015].

[132] R. Murtagh, "Mobile Now Exceeds PC: The Biggest Shift Since the Internet Began," 8 July 2014. [Online]. Available: http://searchenginewatch.com/sew/opinion/2353616/mobile-now-exceeds-pc-the-biggest-shift-since-the-internet-began. [Accessed 10 July 2014].

[133] D. Bosomworth, "Mobile Marketing Statistics 2015," 22 July 2015. [Online]. Available: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/. [Accessed 10 August 2015].

[134] S. Byrne, "Survey results: Devices we use daily in 2014," CNET, 15 September 2014. [Online]. Available: http://www.cnet.com/au/news/survey-results-devices-we-use-daily-in-2014/. [Accessed 20 September 2014].

[135] S. Byrne, "Survey results: Laptops, tablets and phones - together or apart?," CNET, 16 September 2014. [Online]. Available: http://www.cnet.com/au/news/survey-results-laptops-tablets-and-phones/. [Accessed 20 September 2014].

[136] M. Velayanikal, “How Flipkart hopes to shut out rivals by going app-only in India,” Tech In Asia, 13 October 2015. [Online]. Available: https://www.techinasia.com/flipkart-hopes-shut-rivals-apponly-india/. [Accessed 14 October 2015].

[137] NDTV Gadget, “Flipkart Partners With Google to Launch App-Like Mobile Website,” Gadget 360 (An NDTV Venture), 10 November 2015. [Online]. Available: http://gadgets.ndtv.com/apps/news/flipkart-lite-app-like-mobile-website-launched-with-google-763542. [Accessed 11 November 2015].

[138] Times of India Tech, “Snapdeal takes on Flipkart Lite with Snap-lite,” Times of India, 13 November 2015. [Online]. Available: http://timesofindia.indiatimes.com/tech/tech-news/Snapdeal-takes-on-Flipkart-Lite-with-Snap-lite/articleshow/49766314.cms. [Accessed 13 November 2015].

[139] ARM Holdings, “ARM Holdings plc Results Centre,” [Online]. Available: http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-presentations. [Accessed 10 November 2015].

[140] T. Robinson, “Celebrating 50 Billion shipped ARM-powered Chips,” 12 February 2014. [Online]. Available: https://community.arm.com/community/news/blog/2014/02/12/celebrating-50-billion-shipped-arm-powered-chips. [Accessed 14 July 2014].

[141] ARM Holdings, “ARM Holdings plc Annual Report 2013: Strategic Report,” [Online]. Available: http://financialreports.arm.com/pdfs/ARM_FullReport_2013.pdf. [Accessed 12 July 2015].

[142] S. Murry, “ARM’s Reach: 50 Billion Chip Milestone,” 3 March 2014. [Online]. Available: http://www.broadcom.com/blog/chip-design/arms-reach-50-billion-chip-milestone-video/. [Accessed 14 July 2014].

[143] S. ByraM, “An Interview with Steve Furber,” *Communications of the ACM,* vol. 54, no. 5, 2011.

[144] S. E. Kady, M. Khater and M. Alhafnawi, “MIPS, ARM and SPARC- an Architecture Comparison,” in *Proceedings of the World Congress on Engineering*, 2014.

[145] A. N. Sloss, D. Symes and C. Wright, ARM Systems Developer's Guide, Designing and Optimizing System Software, Morgan Kaufmann Publishers, 2014.

[146] “ARM Processors,” [Online]. Available: http://www.arm.com/products/processors/index.php. [Accessed 10 April 2012].

[147] R. Sobti and G. Ganesan, “Performance Comparison of Keccak, Skein, Grøstl, Blake and JH: SHA-3 Final Round Candidate Algorithms on ARM Cortex A8 Processor,” *International Journal of Security and Its Applications,* vol. 9, no. 12, 2015.

[148] R. Gupta, “ARM Cortex: The force that drives mobile devices,” The Mobile Indian, 26 April 2013. [Online]. Available: http://www.themobileindian.com/news/11825_ARM-Cortex-The-force-that-drives-mobile-devices. [Accessed 12 August 2015].

[149] ARM Limited, “Cortex A8 Technical Reference Manual (Revision r3p2),” ARM Limited, Cambridge, 2010.

[150] “OpenBoard-AM335x,” PHYTEC Embedded Pvt. Ltd., [Online]. Available: http://www.phytec.in/products/sbc/openboard-am335x.html. [Accessed 20 May 2014].

[151] PHYTEC Embedded Pvt. Ltd., “OpenBoard-AM3359 Software Development kit for Linux,” January 2013. [Online]. Available: http://www.phytec.in/manuals/OpenBoard-AM335x_SDK.pdf. [Accessed 20 May 2014].

[152] K. Yaghmour, Building Embedded Linux Systems, O'Reilly Media Inc..

[153] US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), “ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissi,” 11 February 2008. [Online]. Available: csrc.nist.gov/groups/ST/hash/documents/SHA3-C-API.pdf. [Accessed 21 July 2010].

[154] US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), “Description of Known Answer Test (KAT) and Monte Carlo Test (MCT) for SHA-3 Candidate Algorithm Submissions,” 20 February 2008. [Online]. Available: csrc.nist.gov/groups/ST/hash/documents/SHA3-KATMCT1.pdf. [Accessed 21 July 2010].

[155] Texas Instruments, “Stellaris® LM4F232 Evaluation Board User Manual,” 14 September 2012. [Online]. Available: http://www.ti.com/lit/ug/spmu272/spmu272.pdf. [Accessed 12 November 2012].

[156] Technical Training Organization, Texas Instruments, “Getting Started with StellarisWare® and the ARM® CortexTM-M4F Workshop: Student Guide and Lab Manual,” July 2012. [Online]. Available: http://software-dl.ti.com/trainingTTO/trainingTTO_public_sw/GSW-M4F-StellarisWare/M4F_Workbook.pdf. [Accessed 01 September 2012].

[157] D. J. Bernstein, “The Salsa20 Family of Stream Ciphers,” in *New Stream Cipher Designs, The eSTREAM Finalists*, vol. 4986 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 84-97.

[158] D. J. Bernstein, “ChaCha, a variant of Salsa20,” 28 January 2008. [Online]. Available: http://cr.yp.to/chacha/chacha-20080128.pdf. [Accessed 22 June 2014].

[159] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robsha, Y. Seurin and C. Vikkelsoe, “PRESENT: An Ultra-Lightweight Block Cipher,” in *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International*

Workshop, Vienna, Austria, September 10-13, 2007. Proceedings, vol. 4727, Springer Berlin Heidelberg, 2007, pp. 450-466.

[160] Y. Nakano, J. Kurihara, S. Kiyomoto and T. Tanaka, "Stream Cipher-Based Hash Function and Its Security," in *e-Business and Telecommunications, 7th International Joint Conference, ICETE 2010, Revised Selected Papers*, vol. 222 of the series Communications in Computer and Information Science, Athens, Springer Berlin Heidelberg, 2010, pp. 188-202.

[161] R. R. Rivest and J. C. N. Schuldt, "Spritz - A Spongy RC4-Like Stream Cipher and Hash Function," in *Presented at CRYPTO 2014 Rump Session*, 2014.

[162] D. J. Bernstein, "The Rumba20 compression function," 2007. [Online]. Available: http://cr.yp.to/rumba20.html. [Accessed 12 January 2015].

[163] A. Shimizu and S. Miyaguchi, "Fast Data Encipherment Algorithm FEAL," in *Advances in Cryptology — EUROCRYPT' 87, Workshop on the Theory and Application of Cryptographic Techniques, Proceedings*, vol. 304 of the series Lecture Notes in Computer Science, Amsterdam, Springer Berlin Heidelberg, 1988, pp. 267-278.

[164] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks and L. Wingers, "The SIMON and SPECK Families of Lightweight Block Ciphers," *Cryptology ePrint Archive, Report 2013/404,* 2013.

[165] R. L. Rivest, "The RC5 Encryption Algorithm," in *Fast Software Encryption, Second International Workshop, Proceedings*, vol. 1008 of the series Lecture Notes in Computer Science, Leuven, Springer Berlin Heidelberg, 1995, pp. 86-96.

[166] H. Wu, "The Stream Cipher HC-128," in *New Stream Cipher Designs, The eSTREAM Finalists*, vol. 4986 of the series Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 39-47.

[167] D. J. Wheeler and R. M. Needham , "TEA, A Tiny Encryption Algorithm," in *Fast Software Encryption, Second International Workshop, Proceedings*, vol. 1008 of the series Lecture Notes in Computer Science, Leuven, Springer Berlin Heidelberg, 1995, pp. 363-366.

[168] R. M. Needham and D. J. Wheeler , "Tea Extensions," Computer Laboratory, University of Cambridge, 1997.

[169] D. J. Wheeler and R. M. Needham, "Correction to XTEA," Computer Laboratory, Cambridge University, 1998.

[170] N. Mouha, "ARX-based Cryptography," 3 June 2011. [Online]. Available: https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/nicky_mouha_arx-slides.pdf. [Accessed 18 August 2014].

[171] J. Massey and X. Lai, *International Data Encryption Algorithm (IDEA),* 1991.

[172] J. Nakahara Jr., V. Rijmen, B. Preneel and J. Vandewalle, “The MESH Block Ciphers,” in *Information Security Applications, 4th International Workshop, WISA 2003, Revised Papers*, vol. 2908 of the series Lecture Notes in Computer Science, Jeju Island, Springer Berlin Heidelberg, 2004, pp. 458-473.

[173] B. Schneier , “Description of a New Variable-length Key, 64-bit Block Cipher (Blowfish),” in *Fast Software Encryption, Cambridge Security Workshop, Proceedings*, vol. 809 of the series Lecture Notes in Computer Science, Cambridge, Springer Berlin Heidelberg, 1994, pp. 191-204.

[174] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson, “The Twofish Encryption Algorithm: A 128-bit Block Cipher,” John Wiley & Sons, Inc., 1999.

[175] D. J. Bernstein, “Cache-timing attacks on AES,” 14 04 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. [Accessed 20 May 2015].

[176] M. Stevens, “Fast Collision Attack on MD5,” *Cryptology ePrint Archive, Report 2006/104. ,* 2006.

[177] V. Klíma, “Finding MD5 Collisions – a Toy For a Notebook,” *Cryptology ePrint Archive, Report 2005/075,* 5 March 2005.

[178] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier and C. Rechberger, “New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba,” in *Fast Software Encryption, 15th International Workshop, FSE 2008, Revised Selected Papers*, vol. 5086 of the series Lecture Notes in Computer Science, Lausanne, : Springer Berlin Heidelberg, 2008, pp. 470-488.

[179] S. Anthony, “255Tbps: World’s fastest network could carry all of the internet’s traffic on a single fiber,” 27 October 2014. [Online]. Available: http://www.extremetech.com/extreme/192929-255tbps-worlds-fastest-network-could-carry-all-the-internet-traffic-single-fiber. [Accessed 17 January 2015].

[180] US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), “The Keyed-Hash Message Authentication Code (HMAC) , Federation Information Processing Standards Publication (FIPS PUB) 198-1,” 2008.

[181] US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), “Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, NIST Special Publication 800-38B,” 2005.

[182] US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), “Digital Signature Standard (DSS), Federation Information Processing Standards Publication (FIPS PUB) 186-4,” 2013.

[183] Q. Dang, “NIST Special Publication 800-106 Randomized Hashing for Digital Signatures,” 2009.

[184] B. Kaliski, “PKCS #5: Password-Based Cryptography Specification (Version 2.0),” Internet Engineering Task Force, September 2000. [Online]. Available: https://tools.ietf.org/html/rfc2898. [Accessed 12 August 2015].

[185] P. Kaushal, R. Sobti and G. Ganesan, “Random Key Chaining (RKC): AES Mode of Operation,” *International Journal of Applied Information Systems,* vol. 1, no. 5, pp. 39-45, February 2012.

[186] National Institute of Standards and Technology, “Modes Development - Proposed Modes,” [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/BCM/modes_development.html. [Accessed 12 January 2015].

[187] J.-P. Aumasson, “Faster Multicollisions,” in *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India, Proceedings*, vol. 5365 of the series Lecture Notes in Computer Science, Kharagpur, Springer Berlin Heidelberg, 2008, pp. 67-77.

[188] E. Zenner, “A Cache Timing Analysis of HC-256,” in *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Revised Selected Papers*, vol. 5381 of the series Lecture Notes in Computer Science, Sackville, New Brunswick: Springer Berlin Heidelberg, 2009, pp. 199-213.

[189] G. Paoloni, “How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures,” September 2010. [Online]. Available: http://www.intel.in/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. [Accessed 12 May 2015].

[190] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks and T. Kohno, “Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive,” in *Fast Software Encryption, 10th International Workshop, FSE 2003*, vol. 2887 of the series Lecture Notes in Computer Science, Lund, Springer Berlin Heidelberg, 2003, pp. 330-346.

[191] D. Whiting, B. Schneier, S. Lucks and F. Muller, “Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive,” [Online]. Available: https://www.schneier.com/cryptography/archives/2005/01/phelix.html. [Accessed 12 June 2013]

[192] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid and M. U. Sharif, "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs," *Cryptology ePrint Archive 2012/368,* 2012.

[193] J.-P. Kaps, P. Yalla, K. K. Surapathi, B. Habib, S. Vadlamudi, S. Gurung and J. Pham, "Lightweight Implementations of SHA-3 Candidates on FPGAs," in *Progress in Cryptology - INDOCRYPT 2011, 12th International Conference on*

Cryptology in India Proceedings, vol. 7107 of the series Lecture Notes in Computer Science, Chennai, Springer Berlin Heidelberg, 2011, pp. 270-289.

LIST OF PUBLICATIONS

[1] R. Sobti and G. Ganesan, "Cryptographic Hash Functions: A Review," *IJCSI International Journal of Computer Science,* vol. 9, pp. 461-479, 2012.

[2] R. Sobti, A. Bagga and G. Ganesan, "Security of Online Social Networks," in *International Conference on Control, Communication, Computer & Mechanical Engineering*, New Delhi, 2012.

[3] P. Kaushal, R. Sobti and G. Ganesan, "Random Key Chaining (RKC): AES Mode of Operation," *International Journal of Applied Information Systems,* vol. 1, no. 5, pp. 39-45, February 2012.

US department of Commerce, National Institute of Tech (NIST), Information Technology Lab (ITL), "Modes Development - Proposed Modes," [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/BCM/modes_development.html. [Accessed 12 January 2015].

[4] S. Chauhan, R. Sobti and S. Anand, "Cryptanalysis of SHA-3 Candidates - A Survey," *Research Journal of Information Technology,* vol. 5, no. 2, pp. 149-159, 2013.

[5] R. Sobti, G. Ganesan and S. Anand, "Performance comparison of Grøestl, JH and BLAKE – SHA-3 Final Round Candidate Algorithms on ARM Cortex M3 Processor," in *2012 International Conference on Computing Sciences*, 2012. ***{Scopus Indexed}***

[6] G. Singh and R. Sobti, "SHA-3 Blake Finalist on Hardware Architecture of ARM Cortex A8 Processor," *International Journal of Computer Applications,* vol. 123, no. 13, pp. 22-27, August 2015.

[7] R. Sobti and G. Ganesan, "Performance Comparison of Keccak, Skein, Grøstl, Blake and JH: SHA-3 Final Round Candidate Algorithms on ARM Cortex A8 Processor," *International Journal of Security and Its Applications,* vol. 9, no. 12, pp. 353-370, December 2015. ***{Scopus Indexed}***

[8] R. Sobti and G. Ganesan, "Analysis of Quarter Rounds of Salsa and ChaCha Core and Proposal of an Alternative Design to Maximise Diffusion," *Indian Journal of Science and Technology,* vol. 9, no. 3, 2016 ***{Scopus Indexed}***

[9] R. Sobti and G. Ganesan, "Performance Evaluation of SHA-3 Final Round Candidate Algorithms on ARM Cortex-M4 Processor," *International Journal of Information Security and Privacy.* (**Accepted for Publication**) ***{Scopus Indexed}***

APPENDIX I

GUIDELINES TO SETUP AND USE MINICOM

1.1 Installing USB to Rs232 Driver for MCP2200 Converter Chip

The driver for MCP220 USB-to-Serial device are generally preloaded in the Linux Kernel. It might be present either in CDC-Communication Device Class or USB Class depending on whether MCP2200 is based on CDC interface or USB interface. The following steps may be used to install drivers in both the cases. For this study, CDC interface was used.

i. **For CDC class**, one the following command is used to cross check loadable module:

   lsmod | grep cdc or
   dmesg | grep ttyACM

ii. **For USB class**, one the following command is used to cross check loadable module:

   lsmod | grep USB or
   dmesg | grep ttyUSB

iii. Cross check the ***/dev*** folder using the following command

   *ls /dev/ttyACM** (for CDC class) or
   *ls /dev/ttyUSB** (for USB class)

   One should be able to locate something like **/dev/ttyACMx** or **/dev/ttyUSBx**

iv. Map /**ttyACMx (or /ttyUSBx)** to a serial port. To map, the following command is used:

   ln -sf /dev/ttyACMx /dev/ttyS0 or
   ln -sf /dev/ttyUSBx /dev/ttyS0

   Note: Certain Minicom software versions may allow us to use **ttyACMx** directly while configuring Minicom as mentioned in the next section.

1.2 Setting up Minicom for the First Time

The following steps should be followed for setting up Minicom for the first time.

i. Install Minicom using the following command.

sudo apt-get install minicom

ii. Start Minicom by executing the following command:

sudo minicom –s.

After this, follow the following steps.

iii. Select option 'Serial Port Setup' from the menu and press 'Enter'.

```
+-----[configuration]------+
| Filenames and paths      |
| File transfer protocols  |
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
+--------------------------+
```

iv. A new screen showing current settings will be displayed. Change the values to make settings as shown in the following figure. The settings can be changed by pressing alphabets given on the left. For example, to change the baud rate, press 'E' and when the cursor blinks, enter the desired value.

```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/ttyS0                                 |
| B - Lockfile Location     : /var/lock                                  |
| C -   Callin Program      :                                            |
| D -  Callout Program      :                                            |
| E -    Bps/Par/Bits       : 115200 8N1                                 |
| F - Hardware Flow Control : No                                         |
| G - Software Flow Control : No                                         |
|                                                                        |
|    Change which setting?                                               |
+-----------------------------------------------------------------------+
        | Screen and keyboard      |
        | Save setup as dfl        |
        | Save setup as..          |
        | Exit                     |
        | Exit from Minicom        |
        +--------------------------+
```

v. After making all changes, press 'Enter' and following screen will be visible.

```
                  +----------------------+
                  |                      |
+-----[configur|  Configuration saved   |
| Filenames and|                        |
| File transfer+----------------------+
| Serial port setup          |
| Modem and dialing          |
| Screen and keyboard        |
| Save setup as dfl          |
| Save setup as..            |
| Exit                       |
| Exit from Minicom          |
+----------------------------+
```

vi. Then select 'Exit'. It will take us to Minicom serial terminal.

1.3 Using Minicom for Accessing Target Machine

i. To open Minicom, use the following command and enter password

sudo minicom

ii. Certain initialization activities will take place and display version number, port name etc.

```
Welcome to minicom 2.5

OPTIONS: I18n
Compiled on May  2 2011, 00:39:27.
Port /dev/ttyACM0

Press CTRL-A Z for help on special keys
```

iii. Switch on the target board (OpenBoard-AM335x). Minicom will start emulating the board and will display the Linux booting process on target board.

```
starting pid 811, tty '/dev/console': '/sbin/getty -L 115200 ttyO0 vt10'

OSELAS(R)-phyCORE-AM335x-PD12.1.1 / phyCORE-AM335x-PD12.1.1
ptxdist-2012.03.0/2013-10-14T15:10:18+0530

phyCORE-AM335x login:
```

iv. Login with root and we can start using ***ls*** or any other command to see the files transferred.

APPENDIX II

INLINE ASSEMBLY AND KERNEL MODULE USED FOR ACESSING CP15 REGISTERS

This appendix gives a brief of Inline assembly and also presents the loadable kernel module written for this study to access USEREN and INTENC registers and in turn access all other registers of CP15 coprocessor necessary for reading cycle counters.

2.1 Inline Assembly

Set of assembly instructions written as inline function are called inline assembly. Using keyword ***asm,*** assembly instructions can be mixed with C program. Inline assembly is used in this study to access assembly instructions like RTDSC and CPUID for x86 machine, and MCR and MRC instructions for ARM architecture. Assembly language has two flavours - Intel style and AT&T style. As GCC uses AT&T style, so this study has used the same. Few important points for using inline assembly in AT&T style are:

i. While accessing CPU registers through inline assembly register, name should be prefixed with ***%*** i.e. for accessing ***eax*** register, use ***%eax.***

ii. Assembly code can be written as:

```
asm (“Assembly Instructions”);  OR __asm__(“Assembly instructions”);
```

iii. Multiple assembly instructions can be written using semicolon. For example, the following instructions add 5 and 10 and store result in ***eax*** register.

```
asm (“movl $5, %eax;” “movl $10, %ebx;” “addl %ebx, %eax;”);
```

iv. Extended Assembly can be written using colon after the assembly code.

```
asm (“Assembly Instructions”
                : output operand
                : input operands
                : clobbered registers);
```

v. Output and input operands help to work on variables. For example, in the following instruction ***%0*** reflect that we have an operand and ***%%*** is used to distinguish register from operand. Contents of ***eax*** register are copied to ***value*** variable.

```
asm ("movl %%eax, %0;" : "=r" (value) );
```

vi. We can have output as well as input operands. The following is an example of a code, where both output/input operands and clobbered register are used. If output operands are missing, then two colons **::** will be used before input operand.

```
asm ("movl %1, %%eax;" "movl %%eax, %0;"
                    :    "=r"(value)
                    :    "=r"(ivar)
                    :    "%eax"  );
```

The above code will copy content of ***ivar*** to ***value*** variable. The ***%eax*** is defined as clobbered register which tells CPU that content in this register will change.

The concepts discussed above have been extensively used for writing inline assembly for this study. The following is an example of code used to access Time Stamp Counter using RDTSC, CPUID, and RDTSCP assembly instructions as mentioned in Chapter 5.

```
static word64 startreading (void)
{
 unsigned int cycles_high, cycles_low;

 preempt_disable(); /* Disable CPU preemption. Works in Supervisor mode only*/
 raw_local_irq_save(flags); /*Disable hard Interrupts Works in Supervisor mode only*/

 asm volatile ("CPUID\n\t"
            "RDTSC\n\t"
            "mov %%edx, %0\n\t"
            "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
             "%rax", "%rbx", "%rcx", "%rdx");

   return (((word64)cycles_low) | (((word64)cycles_high) << 32));
}

static word64 endreading(void)
{
 unsigned int cycles_high, cycles_low;

 asm volatile("RDTSCP\n\t"
           "mov %%edx, %0\n\t"
           "mov %%eax, %1\n\t"
          "CPUID\n\t": "=r" (cycles_high), "=r" (cycles_low):: "%rax",
          "%rbx", "%rcx", "%rdx");

 raw_local_irq_restore(flags); /*Enable hard interrupts on our CPU*/
 preempt_enable(); /*Enable preemption*/
```

```
    return (((word64)cycles_low) | (((word64)cycles_high) << 32));
}
```

2.2 Loadable Kernel Module Used for Accessing CP15 Registers

Loadable kernel modules are the code that can be loaded and unloaded from the kernel as per requirement. It is required whenever some specific code is to be executed in supervisor / privilege mode. A kernel module should have an ***__init*** and ***__exit*** module. The instructions written in ***__init*** module are executed when we load the kernel module using ***insmod*** command whereas instructions written in ***__exit*** module are executed whenever the kernel module is unloaded using ***rmmod*** command.

The loadable kernel module written for this study to access USEREN and INTENC register of Coprocessor CP15 is given below:

```
// File Name : accessCP15.c
//Module name : accessCP15.ko

#include <linux/module.h>   // To include all kernel modules
#include <linux/kernel.h>   // To include KERN_INFO
#include <linux/init.h>     // To include __init and __exit macros
MODULE_AUTHOR("Rajeev Sobti");
MODULE_DESCRIPTION("Writing USEREN and INTENC register of CP15");

static int __init start_code (void)  //can have any name in place of start_code
{
      /* Enable user-mode access to the all performance counters */
      asm volatile  ("MCR p15, 0, %0, c9, c14, 0 \n" :: "r"(1));

      /* Disable counter overflow interrupts */
      asm volatile  ("MCR p15, 0, %0, c9, c14, 2 \n" :: "r"(0x8000000f));

      printk(KERN_INFO "Necessary Action Taken\n");
      return 0;    // Non-zero return means that the module couldn't be loaded.
}

static void __exit cleanup(void)  //can have any name in place of cleanup
{
   printk(KERN_INFO "Exiting Module.\n");
}

module_init(start_code);  // name given above to be used
module_exit(cleanup);  // name given above to be use
```

The ***makefile*** used to ***make*** the above kernel module is given below:

```
obj-m += accessCP15.o // name of source code file followed by .o
```

```
all:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

APPENDIX III

USING CODE COMPOSER STUDIO

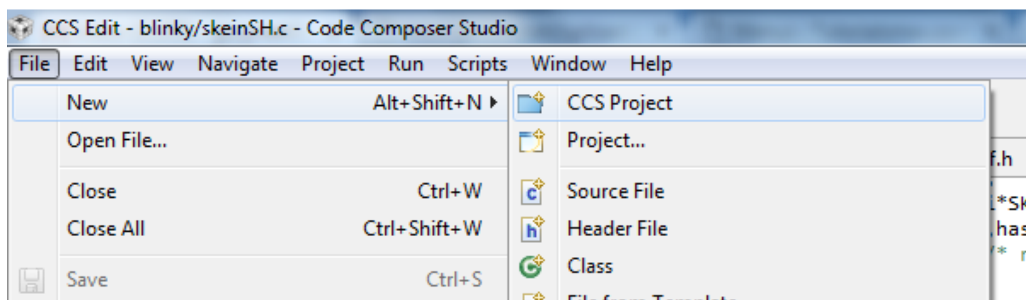
This appendix gives a brief introduction on the use of Code Composer Studio (CCS), an Integrated Development Environment from Texas Instruments, that is used to code, compile, debug, and burn the code on TI processors. This study used CCS for executing and profiling the candidate algorithms on Stellaris® LM4F232 Evaluation Board.

CCS may be downloaded from http://processors.wiki.ti.com/index.php/Download_CCS. Version 5.2.1.00018 was used for this study. The installation process is detailed in [156]. Rather than installing the complete feature set, 'custom' installation was used and Stellaris Cortex M MCU processor architecture was selected for installation. It should be ensured that JTAG emulator support is installed.

In addition to CCS, ICDI (In-Circuit Debug Interface), that supports programming and debugging of the onboard microcontroller, is also required. Installation of StellarisWare is also useful as one can get access to sample projects to understand working on Stellaris Board. In addition to the above mentioned software, drivers of board are also installed on the host machine. The details of downloading and installing all software and drivers are available in [156]. Once all software and drivers are installed, the following steps are required to be followed for accessing CYCCNT counter of Data Watchpoint and Trace unit of Cortex-M4.

3.1 Creating New Project

i. Create a new Project by following File → New →CCS Project



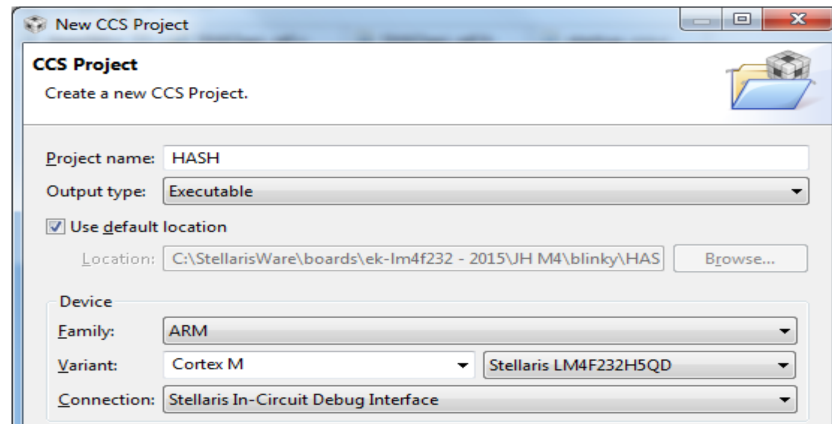
ii. For the new CCS project, certain setup information needs to be provided. The following information was added against different fields.

Device Family - ARM

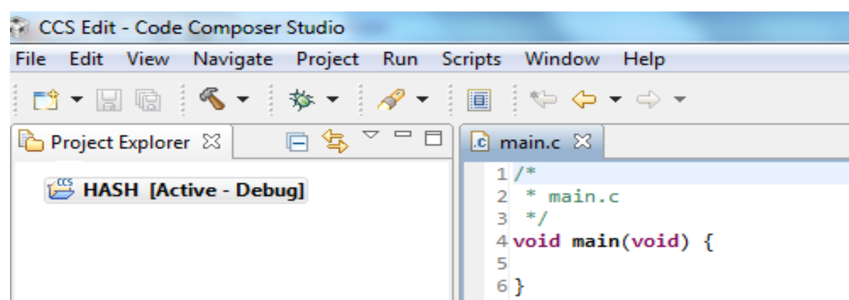
Device Variant - Cortex

Chip Name - Stellaris LM4F232H5QD

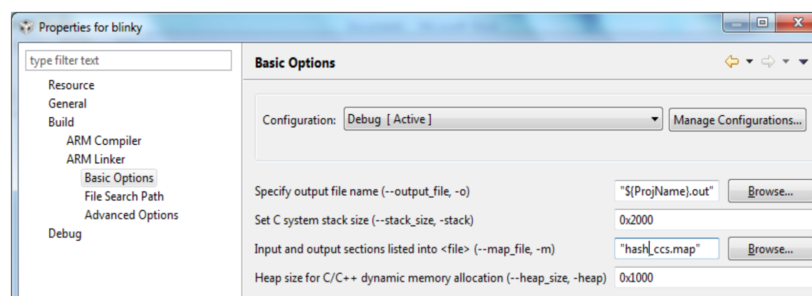
Connection for Debugging and Interfacing with Stellaris Board – Stellaris ICDI



iii. Set Project as active project to start writing the code.



iv. Heap and Stack size need to be setup depending on the type and size of data being used in the project. The same can be set by modifying properties of Project (e.g. HASH shown in the figure). Properties can be accessed by right clicking the project name from Project Explorer window. After that, from the ARM Linker → Basic Options, the heap and stack size can be set as shown in the following figure.



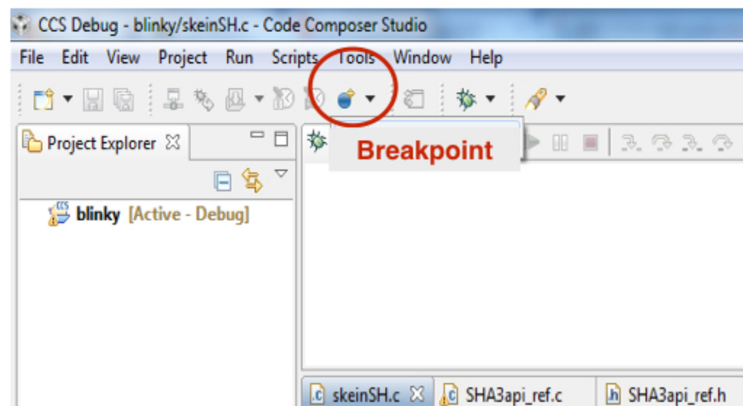
v. Now the written code can be run on target machine by connecting **Target Board** to Host and then pressing Debug Button or the same may be done from Project menu.

3.2 Using Breakpoints to Access DWT's CYCCNT Events

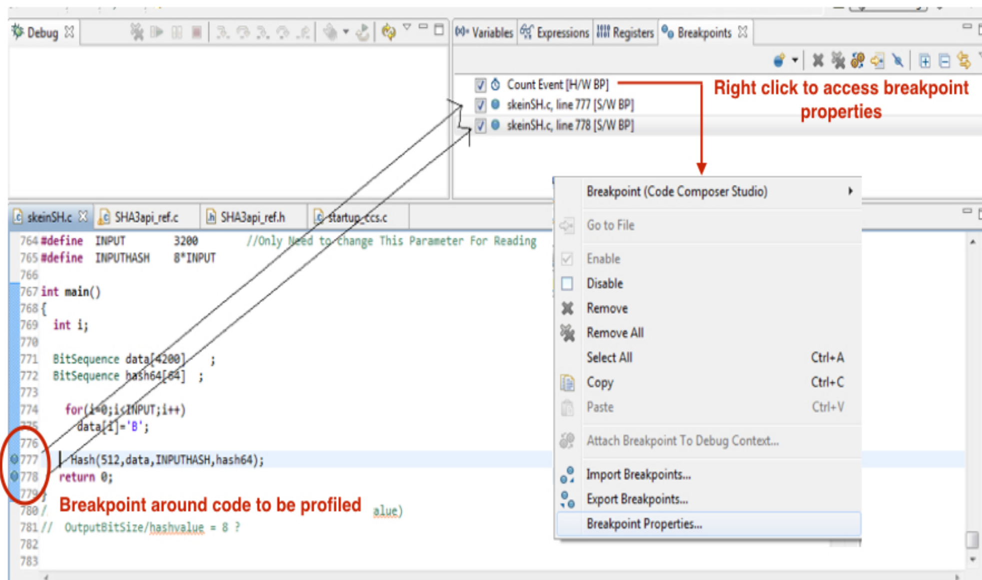
CCS provides 'Count Event' feature that can be used to count various counters provided by DWT (Data Watchpoint and Trace Unit of Cortex-M4). For this study, the important counter to access was CYCCNT (Clock Cycle Counter). The steps as mentioned below should be followed to count cycles consumed by a specific code.

i. The target board must be connected to the host machine.

ii. Put the breakpoints around the code that we want to profile. Breakpoint can be set by clicking on breakpoint icon as shown in the following figure.

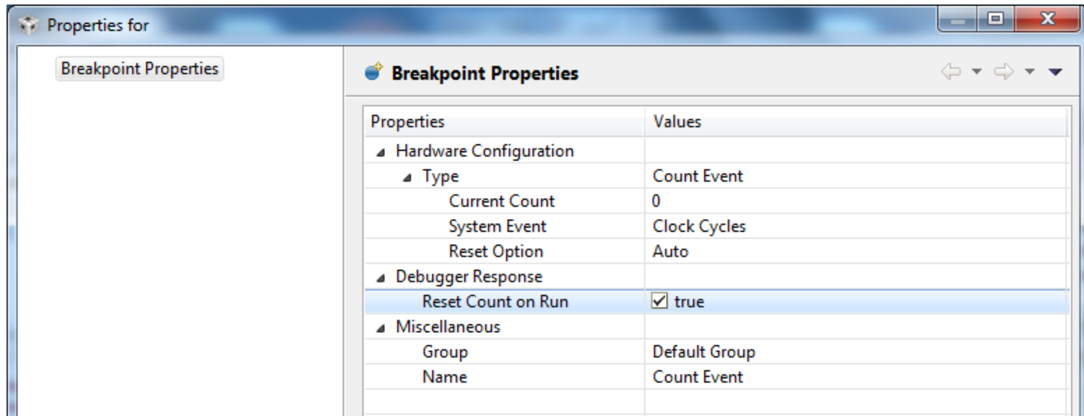


iii. The following figure presents breakpoints put around call to *Hash( )* function. The breakpoint properties need to be set and to access the same, right click Count Event (H/W BP).

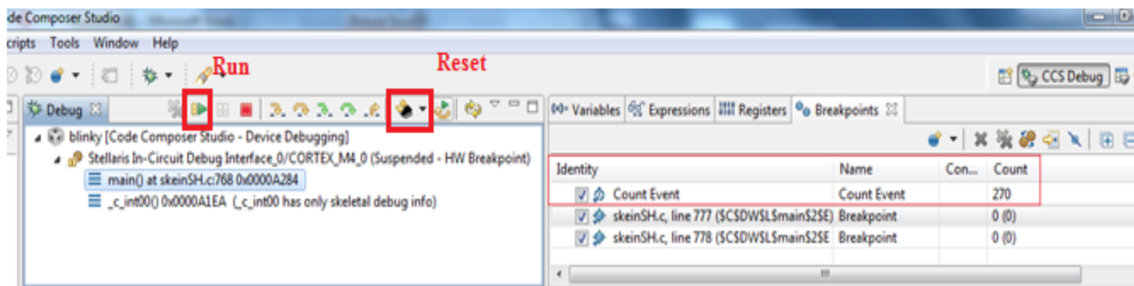


iv. The following settings need to be changed in ‘Breakpoint Properties’ interface.

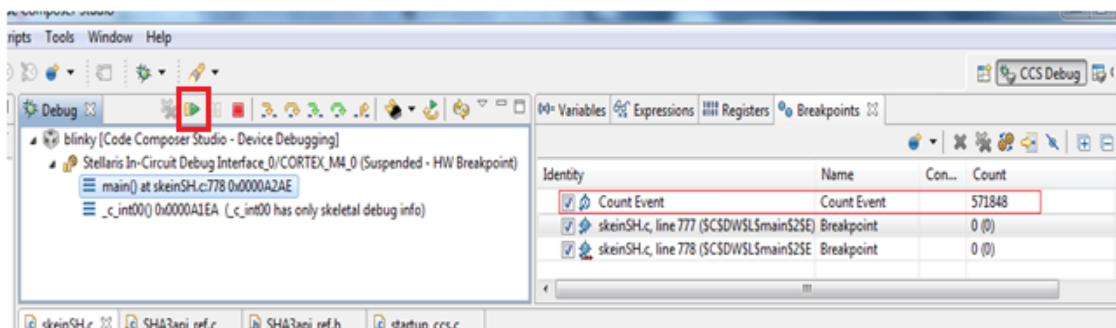
- Set ‘Type’ to ‘Count Event’ and ‘System Event’ to ‘Clock Cycles’.
- Set ‘Reset Count on Run’ to ‘True’



v. After all these settings, the code can be ‘Reset’ and ‘Run’. Execution will halt at the first breakpoint.



vi. Press ‘Run’ again and now program will halt at the second breakpoint. The ‘Count Event’ will display the cycles consumed by the code put within the breakpoints.

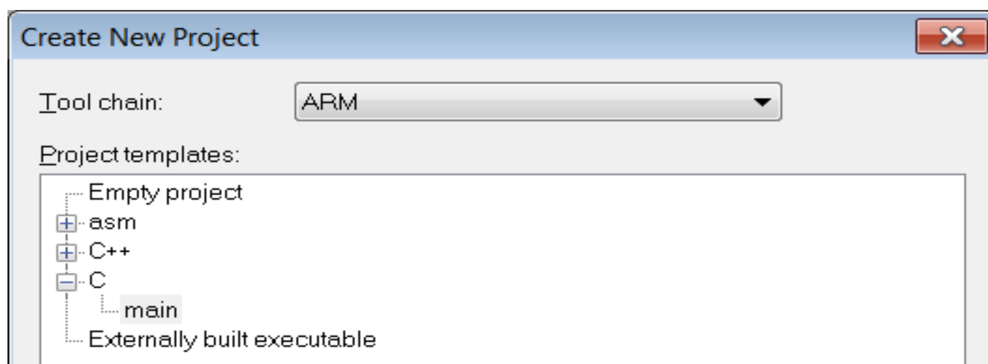


APPENDIX IV

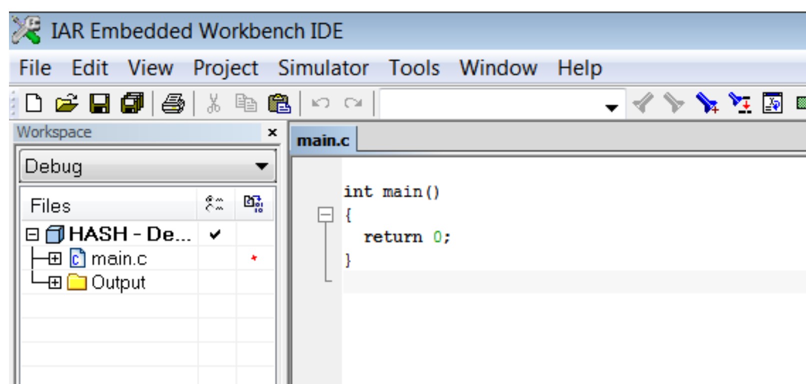
USING IAR EMBEDDED WORKBENCH

This appendix gives a brief introduction on the use of IAR Embedded Workbench. IAR Embedded Workbench is a leading C/C++ compiler, debugger, and simulator tool suite from IAR Systems for 8, 16, and 32-bit Microcontroller units. The 'Kickstart edition' (trial version) can be downloaded from IAR website. For this study, version 7.40.5 was used. After installing the time limited 'Kickstart edition', the following steps can be used to create a project and use Function Profiler for knowing cycle count consumed by different functions/subroutines of the code.

i. Create a new project from the Project menu. 'Create New Project' interface will ask for 'Tool chain' and 'Project template' to be used. Select ARM as Tool chain and C→main as 'Project template'.

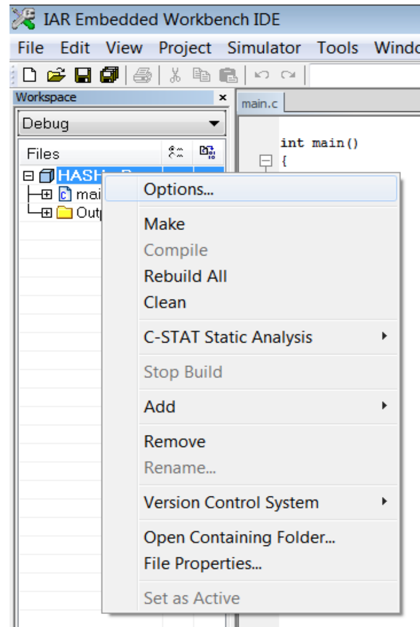


ii. The following screen will appear. Write code that is to be profiled.



iii. Existing header and source files can be added from the 'Project' menu.

iv. Before compiling and profiling the code, certain settings need to be done. The settings can be done through the 'Option' interface and the same can be accessed by right clicking the project name as shown in the following figure.

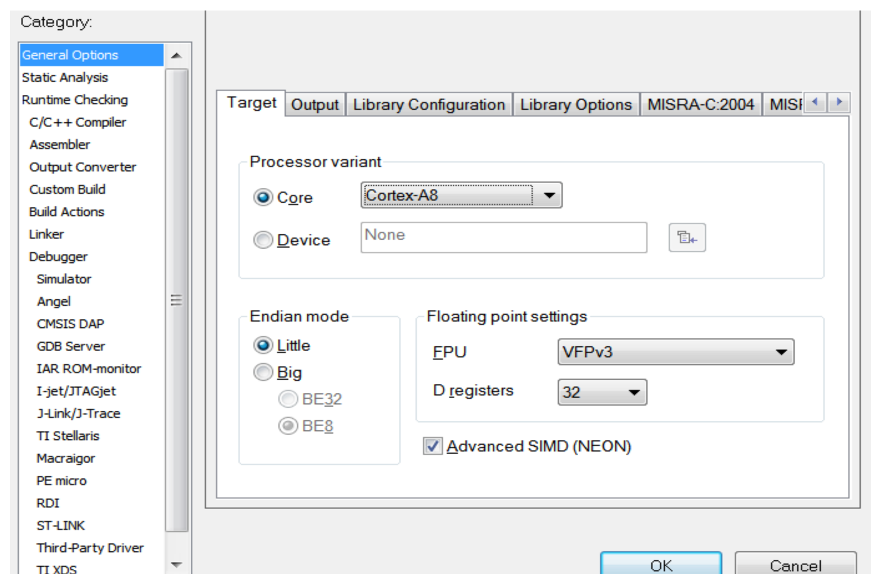


v. The major settings that need to be selected are in 'General Options' and 'Debugger' category.

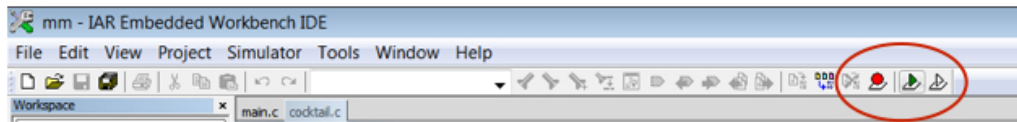
a. In 'General Option' category, under 'Target' tab, select the Core. (ARM7TDMI was selected for this study)

b. Few other settings like Endian mode, Floating Point Settings etc. can also be done.

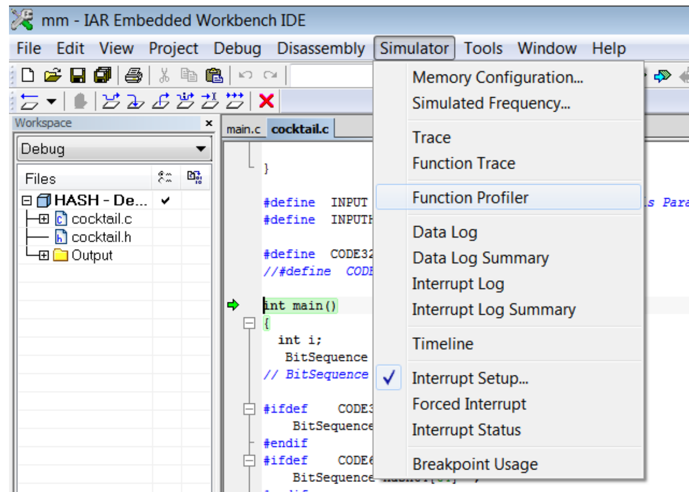
c. Under 'Debugger' category, select driver as 'Simulator'.



vi. After doing necessary settings, the code can be downloaded and debugged from the ‘Project’ menu or by using shortcut icons as shown in the following figure.

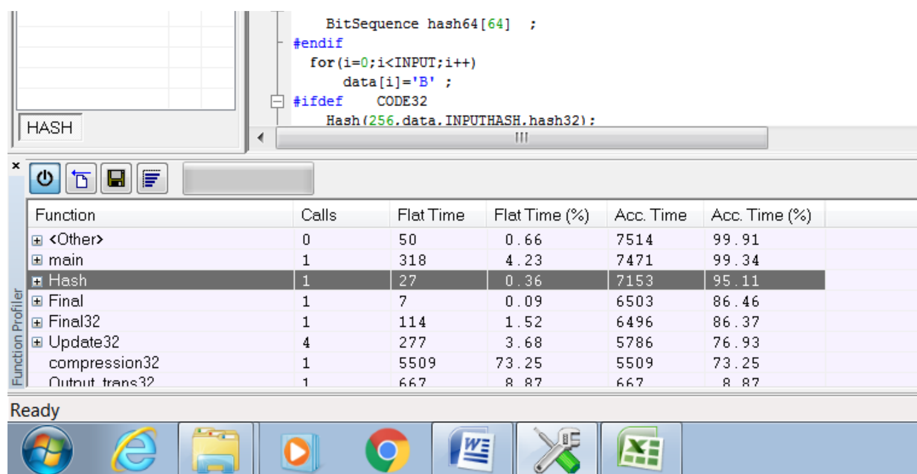


vii. The Function Profiler can be accessed from ‘Simulator’ menu.



viii. After all the settings are done and Function Profile is switched on, the code can be run by using shortcut icons or from the ‘Project’ menu.

ix. The cycles consumed by different functions will be shown under Function Profiler. The value under ‘Accumulated Time’ reflects the cycles consumed by the function written on the left as well as any other function called by this function.



APPENDIX V

PROCESS OF GENERATING INITIAL VALUES FOR *COCKTAIL*

This appendix presents the steps to generate Initial Values used by ***Cocktail***. ***Cocktail-512*** uses 16 words of 32-bit each and ***Cocktail-1024*** uses 16 words of 64-bit each as Initial Values. The 16 words of Initial Values for ***Cocktail-512*** are derived from first 16 prime numbers i.e. 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53. The following table represents steps for first Initial Value i.e. Initial Value corresponding to prime number 2.

Steps	Input	Output
Step 1: Find the square root of the prime number	2	1.414213562373095
Step 2: Convert the result to binary form	1.414213562373095	1.01101010000010011110011001100111111110011101111001100010110000100001…
Step 3: Keep only first 32 bits of fractional part	1.01101010000010011110011001100111111110011101111001100010110000100001…	01101010000010011110011001100111
Step 4: Convert to Hexadecimal notation	01101010000010011110011001100111	**0x6A09E667** (First Initial value H^0)

Similarly, Initial Values corresponding to other prime numbers can be obtained. For ***Cocktail-1024,*** process is same except step 3. In step 3, rather than first 32 bits of fractional part, we keep 64 bits and then convert it to Hexadecimal. For example, output of step 3 for first Initial Value (corresponding to prime number 2), in case of ***Cocktail-1024*** will be: 0110101000001001111001100110011111111001110111100110001011000010 and its hexadecimal conversion is **0x6A09E667F3BCC908**

APPENDIX VI

FULL DIFFUSION IN COCKTAIL AND SKEIN

This appendix shows the result of pseudo running the rounds of ***Cocktail*** and **Skein** to understand how many rounds of compression function of these algorithms take to achieve full diffusion. Full diffusion is number of rounds to propagate a single-bit change to all the bits.

6.1 Full Diffusion in *Cocktail*

To understand how many rounds ***Cocktail's*** compression function consumes to achieve full diffusion, the Column and Row rounds of MCC are pseudo run. To put forward this point, 32-bit version of MCC is used.

The internal state of Cocktail-512 is represented as following matrix where each element is a 32-bit word.

$$\begin{matrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{matrix}$$

This appendix will demonstrate how change in any one bit affects all 512 bits of internal state in different Column and Row Quarter rounds. For the purpose of this appendix, different bits of 32-bit word x_i are numbered from 1 to 32 as shown below.

Quarter round of MCC can be represented as

$QuarterRD_{MCC}\ (a, b, c, d)$

{

Step 1: $b\ = b + a;$

Step 2: $c = c \oplus b; c = c \lll 4$

Step 3: $d = d + c$

Step 4: $a = a \oplus d$**;** $a = a \lll 17$

Step 5: $c = c + a$

Step 6: $b = b \oplus c$**;** $b = b \lll 8$

Step 7: $a = a + b;$

Step 8: $d = d \oplus a$**;**

}

Let's assume bit No. 32 of x_0 is changed. Next few tables present how change in this bit propagates to other bits of internal state.

6.1.1 ROUND 1: (Column Quarter Rounds)

There will be four Column Quarter rounds i.e. four calls to Quarter round of MCC, one for each column.

A) 1st Column Quarter Round

1st Column Quarter round means call to MCC's Quarter round for 1st Column i.e. $QuarterRD_{MCC}(x_0, x_4, x_8, x_{12})$. The changes that take place after this call are given in the following table.

Parameters = (x_0, x_4, x_8, x_{12})	**Bits Positions affected**			
STEPS	x_0 **= a**	$x_4 = b$	x_8 **= c**	$x_{12} = d$
Step 1: $b = b + a;$		32		
Step 2: $c = c \oplus b; c = c \lll 4$			4	
Step 3: $d = d + c$				4
Step 4: $a = a \oplus d$**;** $a = a \lll 17$	21			
Step 5: $c = c + a$			21,4	
Step 6: $b = b \oplus c$**;** $b = b \lll 8$		29,12, 8		

Step 7: $a = a + b$;	29,21,12,8			
Step 8: $d = d \oplus a$;				29,21,12,8,4

B) 2nd, 3rd, and 4th Column Quarter Rounds

These calls will not propagate change in x_0 to any other words of internal state as 2nd, 3rd, and 4th Column Quarter rounds do not involve any word which has been impacted by 1st Column Quarter round (i.e. x_0, x_4, x_8, x_{12}) .

6.1.2 ROUND 1: (Row Quarter Rounds)

There will be four Row Quarter rounds i.e. four calls to Quarter round of MCC, one for each row.

A) 1st Row Quarter Round

1st Row Quarter round means call to MCC's Quarter round for 1st Row **i.e.** $QuarterRD_{MCC}(x_1, x_2, x_3, x_0)$. The call results in changes as per the following table. x_0 is the only word from previous affected words that is appearing in this call and x_0 will affect other words of first row.

Parameters = (x_1, x_2, x_3, x_0)	**Bits Positions affected**			
STEPS	x_1 = a	$x_2 = b$	x_3 = c	$x_0 = d$
Step 1: $b = b + a$;		--		
Step 2: $c = c \oplus b$; $c = c \lll 4$			--	
Step 3: $d = d + c$				29,21,12,8
Step 4: $a = a \oplus d$; $a = a \lll 17$	29,25,14,6			
Step 5: $c = c + a$			29,25,14,6	
Step 6: $b = b \oplus c$; $b = b \lll 8$		22,14,5,1		
Step 7: $a = a + b$;	29,25,22,14,6,5,1			
Step 8: $d = d \oplus a$;				29,25,22,21,14,12,8,6,5,1

B) 2nd Row Quarter Round

2nd Row Quarter round means $QuarterRD_{MCC}(x_6, x_7, x_4, x_5)$. In this call, x_4 is the only already affected word and thus x_4 will propagate the changes.

Parameters = (x_6, x_7, x_4, x_5)	**Bits Positions affected**			
STEPS	x_6 **= a**	$x_7 = b$	$x_4 = c$	$x_5 = d$
Step 1: $b = b + a$;		--		
Step 2: $c = c \oplus b$; $c = c \lll 4$			16,12,1	
Step 3: $d = d + c$				16,12,1
Step 4: $a = a \oplus d$**;** $a = a \lll 17$	29,18,1			
Step 5: $c = c + a$			29,18,16,12,1	
Step 6: $b = b \oplus c$**;** $b = b \lll 8$		26,24,20,9,5		
Step 7: $a = a + b$;	29,26,24,20,18,9,5,1			
Step 8: $d = d \oplus a$**;**				29,26,24,20,18,16,12,9,5,1

C) 3nd Row Quarter Round

3rd Row Quarter round means $QuarterRD_{MCC}(x_{11}, x_8, x_9, x_{10})$. In this call, x_8 is the only already affected word and thus x_8 will propagate the changes.

Parameters = $(x_{11}, x_8, x_9, x_{10})$	**Bits Positions affected**			
STEPS	x_{11} **= a**	$x_8 = b$	$x_9 = c$	$x_{10} = d$
Step 1: $b = b + a$;		21,4		
Step 2: $c = c \oplus b$; $c = c \lll 4$			25,8	
Step 3: $d = d + c$				25,8
Step 4: $a = a \oplus d$**;** $a = a \lll 17$	25,10			
Step 5: $c = c + a$			25,10,8	

Step 6: $b = b \oplus c$; $b = b \lll 8$		29,18,16,12,1		
Step 7: $a = a + b$;	29,25,18,16,12,10,1			
Step 8: $d = d \oplus a$;				29,25,18,16,12,10,8,1

D) 4th Row Quarter Round

4th Row Quarter round means i.e. $QuarterRD_{MCC}(x_{12}, x_{13}, x_{14}, x_{15})$. In this call, x_{12} is the only already affected word and thus x_{12} will propagate the changes.

Parameters = $(x_{12}, x_{13}, x_{14}, x_{15})$	**Bits Positions affected**			
STEPS	x_{12} = a	$x_{13} = b$	$x_{14} = c$	$x_{15} = d$
Step 1: $b = b + a$;	29,21,12,8,4	29,21,12,8,4		
Step 2: $c = c \oplus b$; $c = c \lll 4$			25,16,12,8,1	
Step 3: $d = d + c$				25,16,12,8,1
Step 4: $a = a \oplus d$; $a = a \lll 17$	29,25,21,18, 14,10,6,1			
Step 5: $c = c + a$			29,25,21,18,16,14,12,10,8,6,1	
Step 6: $b = b \oplus c$; $b = b \lll 8$		29,26,24,22,20,18,16,14,12,9 5,1		
Step 7: $a = a + b$;	29,26,25,24,22,21,20,18,16,14,12,10,9, 6,5,1			
Step 8: $d = d \oplus a$;				29,26,25,24,22,21,20,18,16,14,12,10,9, 8,6,5,1

After round1, each word has considerable bits already affected by one bit change in $\boldsymbol{x_0}$. The summary of affected bits of each word, after 1st round, is given below:

$\boldsymbol{x_0}$:29,25,22,21,14,12,8,6,5,1 $\quad$ $\boldsymbol{x_1}$:29,25,22,14,6,5,1 $\quad$ $\boldsymbol{x_2}$:22,14,5,1

$\boldsymbol{x_3}$:29,25,14,6 $\quad$ $\boldsymbol{x_4}$:29,18,16,12,1 $\quad$ $\boldsymbol{x_5}$:29,26,24,20,18,16,12,9,5,1

x_6 :29,26,24,20,18,9,5,1 x_7 :26,24,20,9,5 x_8 :29,18,16,12,1

x_9 :25,10,8 x_{10} :29,25,18,16,12,10,8,1 x_{11} :29,25,18,16,12,10,1

x_{12} : 29,26,25,24,22,21,20,18,16,14,12,10,9,6,5,1 x_{13} :29,26,24,22,20,18,16,14,12,9 5,1

x_{14} :29,25,21,18,16,14,12,10,8,6,1 x_{15} :29,26,25,24,22,21,20,18,16,14,12,10,9, 8,6,5,1

6.1.3 ROUND 2: (Column Quarter Rounds)

The round 2 will have multiplying effect as already all words have considerable number of bits affected by single bit change in x_0.

A) 1st Column Quarter Round

1st Column Quarter round mean $QuarterRD_{MCC}(x_0, x_4, x_8, x_{12})$ and this results in changes as per the following table.

Parameters = (x_0, x_4, x_8, x_{12})	**Bits Positions affected**			
STEPS	x_0 = **a**	$x_4 = b$	x_8 = **c**	$x_{12} = d$
BEFORE START	29,25,22,21,14,12,8,6,5,1	29,18,16,12,1	29,18,16,12,1	29,26,25,24,22,21,20,18,16, 14,12,10,9,6,5,1
Step 1: $b = b + a$;		29,25,22,21,18,16,14,12,8,6 ,5,1		
Step 2: $c = c \oplus b; c = c \lll 4$			29,26,25,22,20,18,16,12,10, 9,5,1	
Step 3: $d = d + c$				29,26,25,24,22,21,20,18,16, 14,12,10,9,6,5,1
Step 4: $a = a \oplus d$; $a = a \lll 17$	31,29,27,26,25,23,22,18, 14,11,10,9,7,6,5,3,1			
Step 5: $c = c + a$			31,29,27,26,25,23,22,20,18, 16,14,12,11,10,9,7,6,5,3,1	

Step 6: $b = b \oplus c; b = b \lll 8$		31,30,29,28,26,24,22,20,19,18,17,16,15,14,13,11,9,7,5,3,2,1		
Step 7: $a = a + b$;	31,30,29,28,27,26,25,24,23,22,20,19,18,17,16,15,14,13,11,10,9,7,6,5,3,2,1			
Step 8: $d = d \oplus a$;				31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,7,6,5,3,2,1

B) 2nd Column Quarter Round

2nd Column Quarter round means $QuarterRD_{MCC}(x_5, x_9, x_{13}, x_1)$ and this results in changes as per the following table.

Parameters = (x_5, x_9, x_{13}, x_1)	**Bits Positions affected**			
STEPS	$\boldsymbol{x_5}$ **= a**	$\boldsymbol{x_9 = b}$	$\boldsymbol{x_{13}}$ **= c**	$\boldsymbol{x_1 = d}$
BEFORE START	29,26,24,20,18,16,12,9,5,1	25,10,8	29,26,24,22,20,18,16,14,12,9 5,1	29,25,22,14,6,5,1
Step 1: $b = b + a$;		29,26,25,24,20,18,16,12,10,9,8,5,1		
Step 2: $c = c \oplus b; c = c \lll 4$			30,29,28,26,24,22,20,18,16,14,13,12,9,5,1	
Step 3: $d = d + c$				30,29,28,26,25,24,22,20,18,16,14,13,12,9,6,5,1
Step 4: $a = a \oplus d; a = a \lll 17$	31,30,29,26,23,22,18,15,14,13,11,10,9,7,5,3,1			

Step 5: $c = c + a$			31,30,29,28,26,24,23,22,20,18,16,15,14,13,12,11,10,9,7,5,3,1	
Step 6: $b = b \oplus c$; $b = b \lll 8$		32,31,30,28,26,24,23,22,21,20,19,18,17,16,15,13,11,9,7,6,5,4,2,1		
Step 7: $a = a + b$;	32,31,30,29,28,26,24,23,22,21,20,19,18,17,16,15,14,13,11,10,9,7,6,5,4,3,2,1			
Step 8: $d = d \oplus a$;				32,31,30,29,28,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,7,6,5,4,3,2,1

C) 3rd Column Quarter Round

3rd Column Quarter round means $QuarterRD_{MCC}(x_{10}, x_{14}, x_2, x_6)$ and this results in changes as per the following table.

Parameters = $(x_{10}, x_{14}, x_2, x_6)$	Bits Positions affected			
STEPS	x_{10} = a	$x_{14} = b$	x_2 = c	$x_6 = d$
BEFORE START	29,25,18,16,12,10,8,1	29,25,21,18,16,14,12,10,8,6,1	22,14,5,1	29,26,24,20,18,9,5,1
Step 1: $b = b + a$;		29,25,21,18,16,14,12,10,8,6,1		
Step 2: $c = c \oplus b$; $c = c \lll 4$			29,26,25,22,20,18,16,14,12,10,9,5,1	
Step 3: $d = d + c$				29,26,25,24,22,20,18,16,14,12,10,9,5,1

Step 4: $a = a \oplus d; a = a \lll 17$	31,29,27,26,25,22,18, 14,11,10,9,7,5,3,1			
Step 5: $c = c + a$			31,29,27,26,25,22,20,18, 16,14,12,11,10,9,7,5,3,1	
Step 6: $b = b \oplus c; b = b \lll 8$		30,29,28,26,24,22,20,19,18, 17,16,15,14,13,11,9, 7,5,3,2,1		
Step 7: $a = a + b;$	31,30,29,28,27,26,25,24,22, 20,19,18,17,16,15,14,13,11, 10,9,7,5,3,2,1			
Step 8: $d = d \oplus a;$				31,30,29,28,27,26,25,24,22, 20,19,18,17,16,15,14,13,12, 11,10,9,7,5,3,2,1

D) 4th Column Quarter Round

4th Column Quarter round means $QuarterRD_{MCC}(x_{15}, x_3, x_7, x_{11})$ and this results in changes as per the following table.

Parameters = $(x_{15}, x_3, x_7, x_{11})$	**Bits Positions affected**			
STEPS	x_{15} = a	$x_3 = b$	x_7 = c	$x_{11} = d$
BEFORE START	29,26,25,24,22,21,20,18,16, 14,12,10,9, 8,6,5,1	29,25,14,6	26,24,20,9,5	29,25,18,16,12,10,1
Step 1: $b = b + a;$		29,26,25,24,22,21,20,18,16, 14,12,10,9, 8,6,5,1		
Step 2: $c = c \oplus b; c = c \lll 4$			30,29,28,26,25,24,22,20,18, 16,14,13,12,10,9,5,1	

Step 3: $d = d + c$				30,29,28,26,25,24,22,20,18, 16,14,13,12,10,9,5,1
Step 4: $a = a \oplus d$**;** $a = a \lll 17$	31,30,29,27,26,25,23,22,18, 15,14,13,11,10,9,7,6,5,3,1			
Step 5: $c = c + a$			31,30,29,28,27,26,25,24,23, 22,20,18,16,15,14,13,12,11, 10,9,7,6,5,3,1	
Step 6: $b = b \oplus c$**;** $b = b \lll 8$		32,31,30,29,28,26,24,23,22, 21,20,19,18,17,16,15,14,13, 11,9, 7,6,5,4,3,2,1		
Step 7: $a = a + b$;	32,31,30,29,28,27,26,25,24, 23,22,21,20,19,18,17,16,15, 14,13,11,10,9, 7,6,5,4,3,2,1			
Step 8: $d = d \oplus a$**;**				32,31,30,29,28,27,26,25,24, 23,22,21,20,19,18,17,16,15, 14,13,12,11,10,9, 7,6,5,4,3,2,1

After all four Column rounds of Round 2, each word has almost all bits already affected by one bit change in $\boldsymbol{x_0}$. The summary of affected bits of each word, after all four Column rounds of Round 2, is given below:

$\boldsymbol{x_0}$: All – {4,8,12,21,32} $\boldsymbol{x_1}$: All – {8,27}

$\boldsymbol{x_2}$:All – {2,4,6,8,13,15,17,19,21,23,24,28,30,32} $\boldsymbol{x_3}$: All – {8,10,12,25,27}

$\boldsymbol{x_4}$: All – {4,6,8,10,12,21,23,25,27,32} $\boldsymbol{x_5}$: All – {8,12,25,27}

$\boldsymbol{x_6}$: All – {4,6,8,21,23,32} $\boldsymbol{x_7}$: All – {2,4,8,17,19,21,32}

$\boldsymbol{x_8}$: All – {2,4,8,13,15,17,19,21,24,28,30,32} $\boldsymbol{x_9}$: All – {3,8,10,12,14,25,27,29}

x_{10} : All – {4,6,8,12,21,23,32}

x_{11} : All – {8}

x_{12} : All – {4,8,32}

x_{13} : All – {2,4,6,8,17,19,21,25,27,32}

x_{14} : All – {4,6,8,10,12,21,23,25,27,31,32}

x_{15} : All – {8, 12}

6.1.4 ROUND 2: (Row Quarter Rounds)

A) 1st Row Quarter Round

1st Row Quarter round means $QuarterRD_{MCC}(x_1, x_2, x_3, x_0)$ and this results in changes as per following table:

Parameters = (x_1, x_2, x_3, x_0)	**Bits Positions affected**			
STEPS	x_1 = a	$x_2 = b$	x_3 = c	$x_0 = d$
BEFORE START	All – {8,27}	31,29,27,26,25,22,20,18, 16,14,12,11,10,9,7,5,3,1	All – {8,10,12,25,27}	All – {4,8,12,21,32}
Step 1: $b = b + a$;		All – {8}		
Step 2: $c = c \oplus b$; $c = c \lll 4$			All – {12}	
Step 3: $d = d + c$				All – {12}
Step 4: $a = a \oplus d$; $a = a \lll 17$	All			
Step 5: $c = c + a$			All	
Step 6: $b = b \oplus c$; $b = b \lll 8$		All		
Step 7: $a = a + b$;	All			
Step 8: $d = d \oplus a$;				All

B) 2nd Row Quarter Round

2nd Row Quarter rounds means $QuarterRD_{MCC}(x_6, x_7, x_4, x_5)$ and this results in changes as per following table:

Parameters = (x_6, x_7, x_4, x_5)	**Bits Positions affected**			
STEPS	x_6 = a	$x_7 = b$	x_4 = c	$x_5 = d$

BEFORE START	All – {4,6,8,21,23,32}	All – {2,4,8,17,19,21,32}	31,30,29,28,26,24,22,20,19,18,17,16,15,14,13,11,9,7,5,3,2,1	All – {8,12,25,27}
Step 1: $b = b + a$;		All – {4,8,21,32}		
Step 2: $c = c \oplus b$; $c = c \lll 4$			All –{4,8,12,25}	
Step 3: $d = d + c$				All – {8,12,25}
Step 4: $a = a \oplus d$; $a = a \lll 17$	All-{25}			
Step 5: $c = c + a$			All-{25}	
Step 6: $b = b \oplus c$; $b = b \lll 8$		All		
Step 7: $a = a + b$;	All			
Step 8: $d = d \oplus a$;				All

C) 3rd Row Quarter Round

3rd Row Quarter round means $QuarterRD_{MCC}(x_{11}, x_8, x_9, x_{10})$ and this results in changes as per following table:

Parameters = $(x_{11}, x_8, x_9, x_{10})$	**Bits Positions affected**			
STEPS	x_{11} = a	$x_8 = b$	x_9 = c	$x_{10} = d$
BEFORE START	All – {8}	All – {2,4,8,13,15,17,19,21,24,28,30,32}	All – {3,8,10,12,14,25,27,29}	All – {4,6,8,12,21,23,32}
Step 1: $b = b + a$;		All – {8}		
Step 2: $c = c \oplus b$; $c = c \lll 4$			All –{12}	
Step 3: $d = d + c$				All – {12}
Step 4: $a = a \oplus d$; $a = a \lll 17$	All			
Step 5: $c = c + a$			All	
Step 6: $b = b \oplus c$; $b = b \lll 8$		All		

Step 7: $a = a + b$;	All			
Step 8: $d = d \oplus a$**;**				All

D) 4th Row Quarter Round

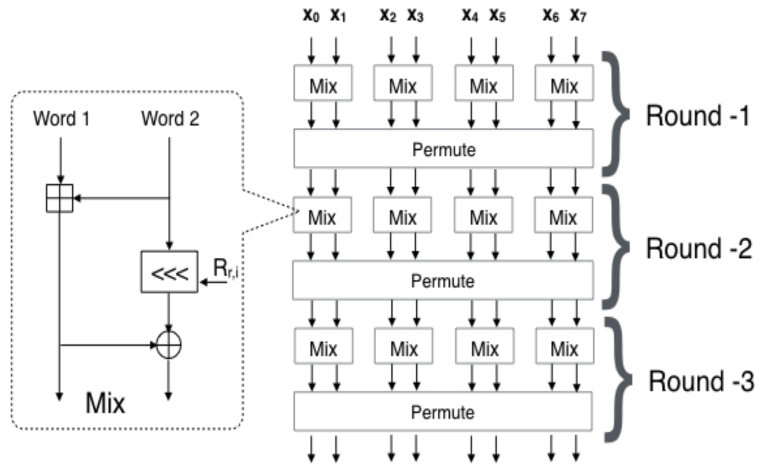
4th Row Quarter round means $QuarterRD_{MCC}(x_{12}, x_{13}, x_{14}, x_{15})$ and this results in changes as per following table:

Parameters = $(x_{12}, x_{13}, x_{14}, x_{15})$	**Bits Positions affected**			
STEPS	x_{12} **= a**	$x_{13} = b$	x_{14} **= c**	$x_{15} = d$
BEFORE START	All – {4,8,32}	All – {2,4,6,8,17,19,21,25, 27,32}	All – {4,6,8,10,12,21,23,25, 27,31,32}	All – {8, 12}
Step 1: $b = b + a$;		All – {4,8,32}		
Step 2: $c = c \oplus b; c = c \lll 4$			All –{4,8,12}	
Step 3: $d = d + c$				All – {8,12}
Step 4: $a = a \oplus d$**;** $a = a \lll 17$	All-{25}			
Step 5: $c = c + a$			All	
Step 6: $b = b \oplus c$**;** $b = b \lll 8$		All		
Step 7: $a = a + b$;	All			
Step 8: $d = d \oplus a$**;**				All

6.2 Full Diffusion in Skein

The diffusion in Skein is shown with an example of Skein-512 i.e. Skein with internal state of 512 bits. Skein works with 64-bit word size. The figure as shown in section 3.4.2 is reproduced below to understand the diffusion process in Skein. The figure represents three of the 72 rounds of Skein-512 (without sub-key) and also its basic primitive structure named Mix function. The 512-bit internal state is arranged as eight 64-bit words. Each round uses four Mix functions and one permutation. The 64-bit word is numbered 1 to 64 with the

most significant bit numbered as 64 and the least significant bit as 1. Let us assume that bit No. 64 of the first word (left most word in the figure) is changed. In next few paragraphs, we will see how change in 64th bit of x_0 affects all other 512 bits.



Permute Function

Input Word	0	1	2	3	4	5	6	7
Output Word	2	1	4	7	6	5	0	3

Round Constants

Round No.	1	2	3	4	5	6	7	8
Mix-1	46	33	17	44	39	13	25	8
Mix-2	36	27	49	9	30	50	29	35
Mix-3	19	14	36	54	34	10	39	56
Mix-4	37	42	39	56	24	17	43	22

In Round-1, first Mix function (operating on x_0 and x_1) affects 18th bit of second word (x_1) because rotation by 46 has shifted 18th bit of x_1 to 64th position. All other MIX functions don't involve x_0 and therefore do not affect any other word. The '*Permute'* function of Round-1, keeps this effect to 18th bit of x_1 only as x_1 is not permuted with any other word. **After Round-1,** only **one bit is affected** (18th bit of x_1) by change in 64th bit of x_0.

Position of all words will change as per *permute* function for example x_6 will be the first word for round 2 in place of x_0. If we name the new sequence of words (for second round) as x_i^j (j *is round number*), then for the second round we will have $x_0^2 = x_6$, $x_1^2 = x_1$, $x_2^2 = x_0$, $x_3^2 = x_7$, $x_4^2 = x_2$, $x_5^2 = x_5$, $x_6^2 = x_4$, $x_7^2 = x_3$

In Round-2, 18th bit of x_1^2 (x_1) affects 18th bit of x_0^2 (x_6) through Mix-1 and x_1^2 also gets rotated with rotation value of 33 and thus 18th bit of x_1^2 becomes 51st bit. So, after Mix function of Round-2, we have 18th bit of x_0^2 and 51st bit of x_1^2 already affected by change in 64th bit of x_0.

Permute function of Round-2 , shifts x_0^2 to x_2^3 and keep x_1^2 to x_1^3 which means **after Round 2**, we will have 18th bit of x_2^3 and 51st bit of x_1^3 affected by change. **(2 bits affected)**

Position of all words will change again as per *permute* function Just like the above mentioned two words (of third round), all other words will also change as per the following:

$x_0^3 = x_6^2$ (= x_4), $x_1^3 = x_1^2 (= x_1)$, $x_2^3 = x_0^2$ (= x_6), $x_3^3 = x_7^2$ (= x_3), $x_4^3 = x_2^2$ (=x_0), $x_5^3 = x_5^2$ (= x_5), $x_6^3 = x_4^2 (= x_2)$, $x_7^3 = x_3^2 (= x_7)$

In Round-3, effect will be distributed further by x_1^3 (through Mix-1) and x_2^3 (through Mix-2). **Mix-1** will result in affecting 51st bit of x_0^3 and 4th bit of x_1^3 which, after permutation, becomes 51st bit of x_2^4 and 4th bit of x_1^4. **Mix-2** will result in affecting 18th bit of x_2^3 and 33rd bit of x_3^3 (rotation by 49 has brought 33rd bit of x_3^3 to position No. 18) which after permutation has shifted to 18th bit of x_4^4 and 33rd bit of x_7^4. So in Round 3, effect has multiplied and we have **four bits affected** (4th bit of x_1^4, 51st bit of x_2^4, 18th bit of x_4^4 and 33rd bit of x_7^4) by one bit change in x_0.

In **Round-4,** these four bits - 4th bit of x_1^4, 51st bit of x_2^4, 18th bit of x_4^4, and 33rd bit of x_7^4 - will multiply the effect by making use of Mix-1, Mix-2, Mix-3, and Mix-4 respectively. Each bit will affect two more bits in this round. For example, effect of 4th bit of x_1^4 will spread to 4th bit of x_2^5 and 48th bit of x_1^5 through Mix-1 and Permute. In this fashion, after Round-4, we will have **eight bits affected**

In **Round-5**, effect of 8 bits will be distributed to 16 bits and in Sixth round, it will spread further to 32 bits. Going in the same way, after 10 round all 512 bits be affected by change in 64th bit of x_0.

APPENDIX VII

TEST VECTOS OF COCKTAIL

Test Vectors of ***Cocktail-512*** and ***Cocktail-1024,*** for few sample inputs, are given here in Hexadecimal notation.

Cocktail-512 (256-bit hash)

Message :

FF

Hash Output:

59 13 ED 2C 57 CD C1 50 9A AB 09 9F
40 2C BC 2F FF 37 07 AF FA A9 9E 9A
C4 80 D4 51 66 3D D9 9A

Message :

FE

Hash Output:

3A 32 8B 64 39 04 41 2D 5C 42 47 C9
AD AE 63 E6 E3 45 60 4B AE 7A F0 3D
CC 16 01 56 93 BF 10 1B

Message :

AE CB B0 27 59 F7 43 3D 6F CB 06 96
3C 74 06 1C D8 3B 5B 3F FA 6F 13 C6

Hash Output:

DB D8 7A E5 D6 C1 58 7F 62 CF AD 87
46 F4 F0 EF A5 AD 08 47 A5 A1 A4 20
5C CF 44 01 3E E3 55 56

Message :

2F DA 31 1D BB A2 73 21 C5 32 95 10
FA E6 94 8F 03 21 0B 76 D4 3E 74 48
D1 68 9A 06 38 77 B6 D1 4C 4F 6D 0E
AA 96 C1 50 05 13 71 F7 DD 8A 41 19
F7 DA 5C 48 3C C3 E6 72 3C 01 FB 7D

Hash Output:

48 68 C1 62 FD B4 AD 8F 3B 62 5E C6
79 20 D2 74 46 76 53 EC CA A1 A6 67
4A AE B7 92 7D 8B 2B 53

Message :

FF (repeated 200 times)

Hash Output:

40	8D	CB	0B	74	37	B1	E9	C1	FA	C8	80
74	C9	2F	02	58	4B	71	8F	D1	1E	7A	4D
31	81	63	71	BE	9C	75	A5				

Message :

00 (repeated 200 times)

Hash Output:

EA	F5	C9	E6	DD	65	AC	D1	7B	38	2A	6B
C6	BF	A0	65	7E	81	CE	AF	C2	51	7B	41
8A	32	94	21	22	51	5E	90				

Cocktail-1024 (512-bit hash)

Message :

FF

Hash Output:

3D	97	11	E9	0C	2D	54	B8	2E	8A	A8	3B
51	8A	E4	02	57	09	F4	D2	49	A9	AD	9C
F0	A9	8F	71	CF	49	48	5F	99	86	27	8D
79	23	B3	89	94	65	31	15	B9	9E	43	DF
6D	F0	21	ED	DB	A2	37	70	AB	BD	77	02
53	63	30	57								

Message :

FE

Hash Output:

FF	F9	26	E3	FC	C5	DE	53	E3	59	59	D8
A4	98	A2	9F	0C	B0	A9	C1	7B	27	8F	6A
7E	20	1C	06	DA	24	27	B1	C0	FD	39	EB
DF	9E	BC	A1	88	5F	E7	C9	1B	3B	5F	FF
4A	63	BC	75	87	89	75	15	07	FD	64	E9
B2	D6	56	8D								

Message :

AE	CB	B0	27	59	F7	43	3D	6F	CB	06	96
3C	74	06	1C	D8	3B	5B	3F	FA	6F	13	C6

Hash Output:

FA	D2	63	80	2A	D8	5A	0D	13	B9	EF	4B

AA 2E 56 2C A4 23 2D D5 F5 46 10 D8
51 DE 54 40 FD AF 68 8C 19 29 1A 06
65 FE 09 86 F0 D6 46 82 00 9C 8E 18
AF A6 DA AE 1B 25 F0 01 CD 70 4F 05
67 79 75 83

Message :

2F DA 31 1D BB A2 73 21 C5 32 95 10
FA E6 94 8F 03 21 0B 76 D4 3E 74 48
D1 68 9A 06 38 77 B6 D1 4C 4F 6D 0E
AA 96 C1 50 05 13 71 F7 DD 8A 41 19
F7 DA 5C 48 3C C3 E6 72 3C 01 FB 7D

Hash Output:

80 98 46 B6 54 C2 A0 78 8A 32 B0 41
D1 28 87 0A 1D D4 B5 13 79 BB 47 D5
87 FE 49 19 68 1B FA A7 68 D8 B1 79
E4 42 B2 E8 B6 E5 BE DB F7 66 2F AA
58 D0 5E C6 2C 7F 50 56 4F 72 90 8D
22 85 46 A1

Message :

FF (repeated 200 times)

Hash Output:

44 B8 7C 77 CB 13 65 FA 2F 6B 3B 91
3B 37 0D 26 2C 61 86 B8 FE 01 DE B8
5D 58 56 F6 23 E0 39 64 FC DB 4B 1A
0E C4 62 19 2B C1 86 44 DE E7 26 3C
15 1A FE DC AF 01 ED 5F E6 53 A9 71
CE A9 0F 66

Message :

00 (repeated 200 times)

Hash Output:

5F 0C 25 B0 96 72 B0 D0 67 B6 6F 13
32 65 64 63 5B AF 7B 10 A1 48 EE AE
BA 34 F3 EC 27 8C 8C 04 59 57 9C 71
21 71 79 47 AE AC 3E 44 7D 63 75 D1
99 B2 4F 4D 02 2C D4 B3 9D 8E C1 EB
95 63 E0 71