PARALLEL HASHING
ALGORITHMS FOR SECURITY
AND FORENSIC APPLICATIONS

THESIS

Submitted
in fulfilment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

By
Neha Kishore
University Regd. No: PHDENG10024

Supervised by
Dr. Bhanu Kapoor,
Professor, Chitkara University, Himachal Pradesh

December, 2014

Department of Computer Science & Engineering
CHITKARA UNIVERSITY, HIMUDA EDUCATIONAL HUB,
SOLAN, HIMACHAL PRADESH-174103
CHITKARA UNIVERSITY, HIMACHAL PRADESH

DECLARATION BY THE STUDENT

I hereby certify that the work which is being presented in this thesis entitled “Parallel Hashing Algorithms for Security and Forensic Applications” is for fulfillment of the requirement for the award of Degree of Doctor of Philosophy submitted in the Department of Computer Science & Engineering, Chitkara University, Barotiwala, Solan, Himachal Pradesh is an authentic record of my own work carried out under the supervision of Dr. Bhanu Kapoor.

The work has not formed the basis for the award of any other degree or diploma, in this or any other Institution or University. In keeping with the ethical practice in reporting scientific information, due acknowledgements have been made wherever the findings of others have been cited.

(Signature)

(Neha Kishore)
This is to certify that the thesis entitled “Parallel Hashing Algorithms for Security and Forensic Applications” submitted by Neha Kishore, Regd. No. PHDENG10024 to the Chitkara University, Barotiwala, Solan, Himachal Pradesh in fulfillment for the award of the degree of Doctor of Philosophy is a bonafide record of research work carried out by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institution or University for the award of any degree or diploma.

(Signature)
Dr. Bhanu Kapoor,
Professor, Chitkara University,
Himachal Pradesh, India
ACKNOWLEDGEMENT

First and foremost, I would like to thank God, for giving me the inner strength and passion that drives me and helps me keep going on in my life. This research could not have been accomplished without his blessings. I hope and pray that this work is not only accepted by him but also that he continues to guide me throughout this life and in my afterlife.

This thesis could never have been completed without the guidance, help and support of my advisor Dr. Bhanu Kapoor. I would like to express my whole hearted appreciation and gratitude towards him for his guidance, support and the invaluable discussions we had during my Ph.D. studies. I was very fortunate to have been supervised by him. I would also like to thank Dr. Srinivas Sistla Murthy, Senior Scientist Computer Forensics Department, for his valuable advices and discussions during the research work.

Many thanks to my research colleagues Ms. Disha Handa, Ms. Sapna Saxena and Ms. Tanu Sharma for their invaluable assistance whenever I needed. A special thanks to Ms. Harpreet Kaur from Electronics and Communication Department for their advices and support and especially for their willingness to help. A heartfelt thanks to Ms. Aarti Joshi and her team for helping me on refining this document in terms of language accuracy.

I also wish to express my deepest, evergreen gratitude to my parents, who have helped, guided and supported me throughout my life. My deepest gratitude also goes to my wonderful brothers, sisters and their families, Mr. Piyush Garg, Ms. Kriti Kishore, Mr. Ajay Goyal, Ms. Archana Gupta and my little niece Bhavya Goyal for their prayers and support, especially during the difficult times of the PhD journey. A special thanks to my little sister Ms. Prachi Kishore for her valuable unconditional support and help in this endeavor.

Last but not the least, I would like to thank the management of Chitkara University, a special thanks to Dr. Ashok Chitkara, Chancellor, Dr. Madhu Chitkara, Pro-Chancellor, Brig. (Dr.) R.S. Grewal, Vice Chancellor, Dr. Varinder Kanwar,
Registrar and Dr. Sudhir Mahajan, Dean Research and Development and his team, Dr. Rajnish Sharma, Dean Academics, Dr. Shaily Jain, HoD CSE Department and all the internal and external examiners for their valuable time and their expert guidance for various progress seminars, presentations, suggestions, feedback and the approvals.

Once, again I would like to extend my deep gratitude to everyone who has helped me in transforming my dream and turning it into a reality.

Neha Kishore
LIST OF PUBLICATIONS

Published/Presented


Accepted


Communicated


• Kishore N & Kapoor B, Energy Efficient Parallel Hash. Paper communicated to *WSEAS Transactions on Communications* (ISSN: 1109-2742).
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANS</td>
<td>American National Standard</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CART</td>
<td>Computer Analysis and Response Team</td>
</tr>
<tr>
<td>CC-NUMA</td>
<td>Cache Coherent NUMA</td>
</tr>
<tr>
<td>CHF</td>
<td>Cryptographic Hash Functions</td>
</tr>
<tr>
<td>CR</td>
<td>Collision Resistance</td>
</tr>
<tr>
<td>CRHF</td>
<td>Collision Resistant Hash Functions</td>
</tr>
<tr>
<td>CTPH</td>
<td>Context Triggered Piecewise Hashing</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DCFL</td>
<td>Defense Computer Forensics Laboratory</td>
</tr>
<tr>
<td>DF</td>
<td>Digital Forensics</td>
</tr>
<tr>
<td>DFS</td>
<td>Digital Forensics Specialist</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>DSS</td>
<td>Digital Signature Standard</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic Curve Digital Signature Algorithm</td>
</tr>
<tr>
<td>EITRH</td>
<td>Enveloped Inverted Tree Recursive Hashing</td>
</tr>
<tr>
<td>EMD</td>
<td>Enveloped Merkle-Damg˚ard</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GCC</td>
<td>GNUs Compiler Collection</td>
</tr>
<tr>
<td>GNU MP</td>
<td>GNUs multi precision library</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
</tbody>
</table>
GPGPU  General Purpose Graphics Processing Unit
HAIFA  HAsh Iterative FrAmework
HPC    High Performance Computing
ITRH   Inverted Tree Recursive Hashing
MAC    Message Authentication Codes
MD     Merkle–Damgård Construction
MD5    Message Digest 5
MDC    Manipulation Detection Codes
MIC    Message Integrity Check
MIMD   Multiple Instruction Stream and Multiple Data Stream
MISD   Multiple Instruction Stream Single Data Stream
MPMD   Multiple Program Multiple Data
NIJ    National Institute of Justice
NIST   National Institute of Standards and Technology
NSRL   National Software Reference Library
NUMA   Non Uniform Memory Access
OWHF   One-Way Hash Functions
OpenMP Open Multiprocessing
PE     Processing Element
PIR    Pre-image Resistance
PKCS   Public Key Cryptography Standard
PPR    Partial Pre-image Resistance
PRO-Pr Pseudo Random Oracle Preservation
RDS    Reference Data Set
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC</td>
<td>Request for Comments</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest Shamir Adleman</td>
</tr>
<tr>
<td>RSHA</td>
<td>Revised Secure Hash Algorithm</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Stream Multiple Data Stream</td>
</tr>
<tr>
<td>SISD</td>
<td>Single Instruction Stream Single Data Stream</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessor Architecture</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Socket Layer</td>
</tr>
<tr>
<td>UOWHF</td>
<td>Universal One Way Hash Functions</td>
</tr>
<tr>
<td>UMA</td>
<td>Uniform Memory Access</td>
</tr>
<tr>
<td>WCR</td>
<td>Weak Collision Resistance</td>
</tr>
<tr>
<td>WYSIWYS</td>
<td>What You See Is What You Sign</td>
</tr>
</tbody>
</table>
## NOTATIONS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>Domain</td>
</tr>
<tr>
<td>$R$</td>
<td>Range</td>
</tr>
<tr>
<td>${0, 1}^n$</td>
<td>Set of all binary strings of length $n$</td>
</tr>
<tr>
<td>${0, 1}^*$</td>
<td>Set of all binary strings, including the empty string</td>
</tr>
<tr>
<td>$M$</td>
<td>Message</td>
</tr>
<tr>
<td>$</td>
<td>M</td>
</tr>
<tr>
<td>$M_i$</td>
<td>$i^{th}$ message block</td>
</tr>
<tr>
<td>$k$</td>
<td>Number of blocks</td>
</tr>
<tr>
<td>$l$</td>
<td>Block size</td>
</tr>
<tr>
<td>$n$</td>
<td>Hash size</td>
</tr>
<tr>
<td>$IV_1(H_0)$</td>
<td>Initial Vector 1</td>
</tr>
<tr>
<td>$IV_2(H_{00})$</td>
<td>Initial Vector 2</td>
</tr>
<tr>
<td>$H_i$</td>
<td>Hash Value at level $i$</td>
</tr>
<tr>
<td>$f()$</td>
<td>Compression function</td>
</tr>
<tr>
<td>$h_t$</td>
<td>Height of the tree</td>
</tr>
<tr>
<td>$h_n$</td>
<td>Final hash size in a tree</td>
</tr>
<tr>
<td>$|$</td>
<td>Append</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>XOR operation</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>Standard asymptotic lower bound</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>Standard asymptotic upper bound</td>
</tr>
<tr>
<td>$H(M(i))$</td>
<td>Hash value of message $M(i)$</td>
</tr>
<tr>
<td>$\text{Cmplx}(H(M(i)))$</td>
<td>Complexity for finding $H(M(i))$</td>
</tr>
<tr>
<td>$\text{Adv}_H^{\text{prop}}$</td>
<td>Adversarial advantage of adversary $A$ against function $H$</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>$T_s$</td>
<td>Wall clock time by sequential algorithm</td>
</tr>
<tr>
<td>$T_p$</td>
<td>Wall clock time by parallel algorithm</td>
</tr>
<tr>
<td>$T_o$</td>
<td>Total parallel overhead</td>
</tr>
<tr>
<td>$S$</td>
<td>Speedup</td>
</tr>
<tr>
<td>$E$</td>
<td>Efficiency</td>
</tr>
<tr>
<td>$X$</td>
<td>Number of times speedup</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of processors</td>
</tr>
<tr>
<td>$L_{\text{min}}$</td>
<td>Minimum number of bits changed</td>
</tr>
<tr>
<td>$L_{\text{max}}$</td>
<td>Maximum number of bits changed</td>
</tr>
<tr>
<td>$L_i$</td>
<td>Mean of number of bits changed</td>
</tr>
<tr>
<td>$Pr$</td>
<td>Mean of Changed Probability</td>
</tr>
<tr>
<td>$\sigma_L$</td>
<td>Standard Deviation of the bit numbers changed</td>
</tr>
<tr>
<td>$\sigma_{Pr}$</td>
<td>Standard Deviation Probability</td>
</tr>
<tr>
<td>$m$</td>
<td>In RSA, bit length of $m$ is considered to be the key size; used as modulus.</td>
</tr>
<tr>
<td>$(m,pr)$</td>
<td>An RSA private key, where $m$ is the modulus, and $pr$ is the private signature exponent.</td>
</tr>
<tr>
<td>$(m,pu)$</td>
<td>An RSA public key, where $m$ is the modulus, and $pu$ is the public verification exponent.</td>
</tr>
<tr>
<td>Table No.</td>
<td>Title</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Table 1.1</td>
<td>Attacks made on SHA-0</td>
</tr>
<tr>
<td>Table 1.2</td>
<td>Attacks made on SHA-1</td>
</tr>
<tr>
<td>Table 1.3</td>
<td>Attacks made on SHA-2 Variants</td>
</tr>
<tr>
<td>Table 1.4</td>
<td>Comparison of SHA Variants</td>
</tr>
<tr>
<td>Table 2.1</td>
<td>Main Characteristics of the Machine Used</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Execution Time by RSHA-1 (ITRH) and SHA-1 on AMD FX Bulldozer Machine</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Time Consumption on Multiple Cores by ITRH</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>EITRH Variants</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Execution Time Taken by SHA-1 and RSHA-1 (EITRH)</td>
</tr>
<tr>
<td>Table 3.5</td>
<td>Execution Results of RSHA-1 (EITRH) on Different Number of Cores</td>
</tr>
<tr>
<td>Table 3.6</td>
<td>Efficiency of RSHA-1 as a Function of Problem Size and Number of Processing Elements</td>
</tr>
<tr>
<td>Table 3.7</td>
<td>Comparison of RSHA-1 with Standard Serial Hashing Algorithms</td>
</tr>
<tr>
<td>Table 3.8</td>
<td>Comparison of RSHA-1 with Other Tree Hashing Based Algorithms</td>
</tr>
<tr>
<td>Table 3.9</td>
<td>Comparison of RSHA-1 with Other Parallel Hashing Algorithms</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Statistics of Number of Changed Bit $L_i$</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Execution Time Taken by SHA-1 and RSHA-1 on AMD Bulldozer Machine</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Experimental Results of RSHA-1 (EITRH) on Different Number of Cores with Different File Sizes</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Execution Time Taken by SHA-1 and RSHA-1 on Compressed Data</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Performance of RSHA-1 on Multiple Cores</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>Estimated Runtime for a Sample Use Case</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>Execution Results of Signature Generation for Parallel Hashing Algorithm</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Execution Results of Signature Verification and Validation for Parallel Hashing Algorithm</td>
</tr>
</tbody>
</table>
Table 6.3 Execution Results of Signature Generation for Parallel Hashing Algorithm on Multiple Cores .................................................................137
Table 6.4 Execution Results of Signature Verification and Validation for Parallel Hashing Algorithm on Multiple Cores .................................................................137
Table 6.5 Execution Results of Signature Generation for Parallel RSA and RSHA-1 .................................................................139
Table 6.6 Execution Results of Signature Verification and Validation for Parallel RSA and RSHA-1 .................................................................139
Table 6.7 Execution Results of Signature Generation for Parallel RSA and RSHA-1 on Multiple Cores .................................................................140
Table 6.8 Execution Results of Signature Verification and Validation for Parallel RSA and RSHA-1 on Multiple Cores .................................................................140
Table 7.1 Phases of Application Processing and Performance Factors ............146
Table 7.2 Execution Time SHA-1 and RSHA-1 on CPU, GPU .........................148
Table 7.3 Execution Time SHA-1 and Optimized RSHA-1 on CPU, GPU ........153
Table 8.1 Test Environment Configuration .......................................................162
Table 8.2 Energy and Throughput Performance on Platform 1 .......................163
Table 8.3 Energy and Throughput Performance on Platform 2 .......................164
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Title</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 1.1</td>
<td>Cryptographic Hash Function</td>
<td>5</td>
</tr>
<tr>
<td>Fig. 1.2</td>
<td>Classification of Cryptographic Hash Functions</td>
<td>6</td>
</tr>
<tr>
<td>Fig. 1.3</td>
<td>Security Properties of OWHFs and CRHFs</td>
<td>7</td>
</tr>
<tr>
<td>Fig. 1.4</td>
<td>Compression Function of Merkle-Damg˚ard Construction</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 1.5</td>
<td>Domain Extender of Merkle Damgard Construction</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 1.6</td>
<td>NMAC Construction</td>
<td>12</td>
</tr>
<tr>
<td>Fig. 1.7</td>
<td>HMAC Construction</td>
<td>12</td>
</tr>
<tr>
<td>Fig. 1.8</td>
<td>EMD Construction</td>
<td>13</td>
</tr>
<tr>
<td>Fig. 1.9</td>
<td>HAIFA Construction</td>
<td>14</td>
</tr>
<tr>
<td>Fig. 1.10</td>
<td>Sponge Construction</td>
<td>15</td>
</tr>
<tr>
<td>Fig. 1.11</td>
<td>One Iteration of Compression Function of SHA-1</td>
<td>19</td>
</tr>
<tr>
<td>Fig. 1.12</td>
<td>One Iteration of Compression Function of SHA-2 Family</td>
<td>22</td>
</tr>
<tr>
<td>Fig. 2.1</td>
<td>Flynn’s Taxonomy</td>
<td>41</td>
</tr>
<tr>
<td>Fig. 3.1</td>
<td>ITRH Framework</td>
<td>58</td>
</tr>
<tr>
<td>Fig. 3.2</td>
<td>Three Step ITRH Framework</td>
<td>58</td>
</tr>
<tr>
<td>Fig. 3.3</td>
<td>Execution Time by SHA-1 and RSHA-1 (ITRH) on Small Files</td>
<td>62</td>
</tr>
<tr>
<td>Fig. 3.4</td>
<td>Execution Time by SHA-1 and RSHA-1 (ITRH) on Large Files</td>
<td>63</td>
</tr>
<tr>
<td>Fig. 3.5</td>
<td>Execution Time on Multiple Cores by ITRH</td>
<td>64</td>
</tr>
<tr>
<td>Fig. 3.6</td>
<td>Speedup of RSHA-1 with Increasing Number of Cores</td>
<td>64</td>
</tr>
<tr>
<td>Fig. 3.7</td>
<td>EITRH Framework</td>
<td>66</td>
</tr>
<tr>
<td>Fig. 3.8</td>
<td>Speedup versus File Size with Fixed Number of Processors</td>
<td>75</td>
</tr>
<tr>
<td>Fig. 3.9</td>
<td>Speedup versus Number of Processing Elements</td>
<td>77</td>
</tr>
<tr>
<td>Fig. 3.10</td>
<td>Efficiency as a Function of b and p</td>
<td>77</td>
</tr>
<tr>
<td>Fig. 3.11</td>
<td>Performance of RSHA-1 (EITRH) over SHA-1 for Small File Sizes</td>
<td>82</td>
</tr>
<tr>
<td>Fig. 3.12</td>
<td>Performance of RSHA-1(EITRH) over SHA-1 for Large File Sizes</td>
<td>82</td>
</tr>
<tr>
<td>Fig. 3.13</td>
<td>Performance of RSHA-1 with the Increasing Number of Cores</td>
<td>84</td>
</tr>
<tr>
<td>Fig. 3.14</td>
<td>Speedup Gain with the Increase in Number of Processors</td>
<td>85</td>
</tr>
<tr>
<td>Fig. 3.15</td>
<td>Variation in Efficiency in RSHA-1 as File Size Increases with Fixed Number of Processors (8)</td>
<td>86</td>
</tr>
</tbody>
</table>
Fig. 3.16 Variation in Efficiency in RSHA-1 as Number of Processing Elements Increases for a Fixed Problem Size (512 MB) ................................................................. 86
Fig. 4.1 (a) Pre Image Collision Resistance, (b) Weak Collision Resistance, (c) Strong Collision Resistance ................................................................. 92
Fig. 4.2 Distribution of Changed Bit Li ................................................................. 97
Fig. 5.1 Digital Forensic Investigation Model ......................................................... 103
Fig. 5.2 Digital Forensic Investigation Process ......................................................... 104
Fig. 5.3 Binary Hash Values of Text File under Different Conditions ................. 112
Fig. 5.4 Google Bird Image - I11 ........................................................................ 114
Fig. 5.5 Mona Lisa ................................................................................................ 114
Fig. 5.6 Transformed Images – I12, I13, I14 ......................................................... 114
Fig. 5.7 Binary Hash Values of bmp File under Different Conditions ........... 115
Fig. 5.8 Embedded Image .................................................................................. 116
Fig. 5.9 Transformed Images I22, I23 ................................................................. 116
Fig. 5.10 Transformed Images I24, I25, I26 ......................................................... 116
Fig. 5.11 Binary Hash Values of jpeg File under Different Conditions .......... 117
Fig. 5.12 Performance of RSHA-1 and SHA-1 for Different File Sizes .......... 119
Fig. 5.13 Run Time Efficiency of RSHA-1 (EITRH) on Different Number of Cores with Different File Sizes ................................................................. 119
Fig. 5.14 Performance of RSHA-1 and SHA-1 on Compressed Data ............ 121
Fig. 6.1 RSA Digital Signature Generation and Verification Process ........ 128
Fig. 6.2 Comparative Analysis of Signature Generation and Verification Process on Parallel Digital Signature Algorithm ................................................. 141
Fig. 7.1 Grid of Thread Blocks ........................................................................ 145
Fig. 7.2 Grid, Block and Thread Distribution in Kernel ................................. 147
Fig. 7.3 Performance of SHA-1 and RSHA-1 on CPU, GPU ....................... 149
Fig. 7.4 NVIDIA’s Visual Profiler Host to Device Copy Time Utilization .... 149
Fig. 7.5 NVIDIA Visual Profiler Host to Device Copy Details ................. 149
Fig. 7.6 Page-locked Memory Usage Shown in NVIDIA Visual Profiler .... 151
Fig. 7.7 GPU Utilization ................................................................................. 152
Fig. 7.8 Performance of SHA-1 and Optimized RSHA-1 on CPU, GPU .... 153
Fig. 8.1 Power Usage Window of Joulemeter Tool ........................................ 160
Fig. 8.2 Joulemeter Calibration Window ................................................................. 161
Fig. 8.3 Energy Consumption Results by Joulemeter in Excel Sheet for an Application .................................................................................................................. 163
Fig. 8.4 Comparison of Energy and Throughput Performance by RSHA-1 on Platform 1 .................................................................................................................. 166
Fig. 8.5 Comparison of Energy and Throughput Performance by RSHA-1 on Platform 2 .................................................................................................................. 166
<table>
<thead>
<tr>
<th>Algorithm No.</th>
<th>Title</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 3.1</td>
<td>ITRH Algorithm</td>
<td>59</td>
</tr>
<tr>
<td>Algorithm 3.2</td>
<td>EITRH Algorithm</td>
<td>67</td>
</tr>
<tr>
<td>Algorithm 6.1</td>
<td>RSA Digital Signature Generation with Parallel Hashing</td>
<td>131</td>
</tr>
<tr>
<td>Algorithm 6.2</td>
<td>RSA Digital Signature Verification and Validation with Parallel Hashing</td>
<td>132</td>
</tr>
<tr>
<td>Algorithm 6.3</td>
<td>Parallel RSA Digital Signature Generation</td>
<td>133</td>
</tr>
<tr>
<td>Algorithm 6.4</td>
<td>Parallel RSA Digital Signature Verification and Validation</td>
<td>134</td>
</tr>
</tbody>
</table>
# CONTENTS

Declaration by the Student ........................................... ii  
Certificate by the Supervisor ........................................ iii  
Acknowledgement ......................................................... iv  
List of Publications .................................................. vi  
Abbreviations .............................................................. viii  
Notations ................................................................. xi  
List of Tables .............................................................. xiii  
List of Figures ............................................................. xv  
List of Algorithms ...................................................... xviii  
Contents ................................................................. xix  
Abstract ................................................................. xxiii  

Chapter 1 Introduction .................................................. 2  
  1.1 Background .......................................................... 3  
  1.2 Cryptographic Hash Functions ................................... 4  
    1.2.1 CHF Security Notions ......................................... 6  
  1.3 CHF Designs ........................................................ 8  
    1.3.1 Iterative Hash Functions .................................... 9  
    1.3.2 Tree-based Hash Constructions ............................ 15  
  1.4 Popular Cryptographic Hash Algorithms ......................... 16  
    1.4.1 MD5 ........................................................... 16  
    1.4.2 SHA-0 ........................................................ 17  
    1.4.3 SHA-1 ........................................................ 18  
    1.4.4 SHA-2 ........................................................ 22  
    1.4.5 SHA-3 ........................................................ 24  
  1.5 Current Issues in Hashing Algorithms .......................... 27  
  1.6 Motivation for Proposed Research .............................. 27  
  1.7 Literature Review ................................................. 29  
  1.8 Outline of the Dissertation ..................................... 34  

Chapter 2 Basic Concepts, Tools and Techniques .................. 38  
  2.1 Introduction to Parallel Processing ........................... 38  
    2.1.1 Parallel Programming Terms ................................ 39  
  2.2 Concepts and Techniques ......................................... 41  
    2.2.1 Computer System Categorization .......................... 41
2.2.2 Decomposition Techniques ......................................................... 43
2.2.3 Mapping Technique for Load Balancing ........................................ 44
2.2.4 Parallel Computer Memory Architectures ...................................... 45
2.2.5 Parallel Algorithm Model .......................................................... 46
2.2.6 Data Dependencies .................................................................... 48

2.3 Tools Used .................................................................................... 48
2.3.1 OpenMP API ............................................................................. 50
2.3.2 Gprof ....................................................................................... 50
2.3.3 MinGW .................................................................................... 51
2.3.4 Code Blocks ............................................................................. 51
2.3.5 Joulemeter ............................................................................... 51

2.4 Assessment Metrics ....................................................................... 51
2.4.1 Speedup .................................................................................. 52
2.4.2 Efficiency ............................................................................... 52
2.4.3 Security Factors ....................................................................... 52
2.4.4 Energy Efficiency ................................................................. 53

Conclusion .......................................................................................... 53

Chapter 3 Parallel Cryptographic Hashing Transformation ........................................ 55
3.1 Introduction ................................................................................. 55
3.2 Inverted Tree Recursive Hashing (ITRH) Transformation ...................... 56
  3.2.1 ITRH Framework ................................................................. 57
  3.2.2 RSHA-1 Algorithm- Variant of ITRH ....................................... 59
  3.2.3 Experimental Results .......................................................... 61
  3.2.4 Discussions ........................................................................... 65
3.3 Enveloped Inverted Tree Recursive Hashing (EITRH) Transformation ....... 65
  3.3.1 Framework Description ......................................................... 66
  3.3.2 Parallelization Techniques .................................................... 70
  3.3.3 EITRH Variants ................................................................... 71
  3.3.4 Performance Benchmarks ..................................................... 71
  3.3.5 Experimental Results .......................................................... 80

Conclusion .......................................................................................... 89

Chapter 4 Security Analysis of EITRH ...................................................... 91
4.1 Introduction ................................................................................. 91
4.2 Security Analysis ........................................................................... 92
  4.2.1 Pre-image Resistance (PIR) ................................................... 92
  4.2.2 Weak Collision Resistance (WCR) ......................................... 93
4.2.3 Collision Resistance (CR) .......................................................94
4.2.4 Non-correlation Resistance (Confusion and Diffusion Analysis) ...95
4.2.5 Partial Pre-image Resistance (PPR) ........................................97
4.2.6 Pseudorandom Oracle Preservation (PRO-Pr) ..........................97

Conclusion .............................................................................98

Chapter 5 Forensic Application of EITRH ..................................100
5.1 Introduction .........................................................................100
5.2 Digital Forensics ................................................................102
   5.2.1 Digital Forensic Investigation Process .................................104
   5.2.2 Hashing in Digital Forensic ...............................................106
5.3 Current Approaches ............................................................108
5.4 Parallelized Framework for File Hashing .................................109
5.5 Performance Analysis ..........................................................110
   5.5.1 Randomness Test ..............................................................110
   5.5.2 Performance Test ..............................................................118

Conclusion .............................................................................123

Chapter 6 EITRH in Digital Signatures ....................................125
6.1 Introduction .........................................................................125
6.2 Digital Signatures ................................................................126
   6.2.1 RSA Digital Signatures ....................................................128
6.3 Parallel Digital Signature ......................................................130
   6.3.1 RSA Digital Signature with Parallel Hashing .......................131
   6.3.2 RSA Digital Signature with Parallel RSA and Parallel Hashing ...132
6.4 Experimental Results ............................................................135
   6.4.1 Results Analysis for RSA Digital Signature with Parallel Hashing ................................................135
   6.4.2 Results Analysis for RSA Digital Signature with Parallel RSA and Parallel Hashing ................138

Conclusion .............................................................................141

Chapter 7 EITRH on GPUs ....................................................143
7.1 Introduction to GPU ..............................................................143
   7.1.1 Compute Unified Device Architecture .................................144
   7.1.2 GPU Processing Phases ....................................................145
7.2 Implementation Details ..........................................................146
7.3 Experimental Results .............................................................148
   7.3.1 Performance Optimization ...............................................150
<table>
<thead>
<tr>
<th>Conclusion</th>
<th>153</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 8 Energy Analysis of EITRH</td>
<td>156</td>
</tr>
<tr>
<td>8.1 Introduction</td>
<td>156</td>
</tr>
<tr>
<td>8.2 Energy Efficiency</td>
<td>157</td>
</tr>
<tr>
<td>8.3 Test Environment</td>
<td>158</td>
</tr>
<tr>
<td>8.3.1 Joulemeter</td>
<td>159</td>
</tr>
<tr>
<td>8.3.2 Energy Conversions</td>
<td>161</td>
</tr>
<tr>
<td>8.4 Experimental Results</td>
<td>162</td>
</tr>
<tr>
<td>8.5 Performance Impact</td>
<td>165</td>
</tr>
<tr>
<td>Conclusion</td>
<td>167</td>
</tr>
<tr>
<td>Chapter 9 Conclusions and Future Work</td>
<td>169</td>
</tr>
<tr>
<td>9.1 Thesis Contributions</td>
<td>169</td>
</tr>
<tr>
<td>9.2 Conclusions of this Thesis</td>
<td>171</td>
</tr>
<tr>
<td>9.3 Future Research Directions</td>
<td>173</td>
</tr>
<tr>
<td>References</td>
<td>177</td>
</tr>
<tr>
<td>Appendix A</td>
<td>187</td>
</tr>
<tr>
<td>Appendix B</td>
<td>188</td>
</tr>
<tr>
<td>Appendix C</td>
<td>189</td>
</tr>
<tr>
<td>Appendix D</td>
<td>190</td>
</tr>
<tr>
<td>Appendix E</td>
<td>191</td>
</tr>
</tbody>
</table>
ABSTRACT

Modern computing is now using parallelism to achieve higher performance as opposed to scaling frequency in the past. Security algorithms are compute intensive and can benefit significantly in their performance from parallel implementations. Parallel computing can also be used for more energy-efficient implementation of security algorithm making them more usable in areas like mobile computing, personal computing, and cloud computing.

Cryptographic hashing algorithms are considered to be one of the important requirements for the digital data security and digital forensics. Many secure hashing algorithms are available but most of them are serial in nature. Not much work has been done to parallelize these algorithms to exploit parallel computing resources that are available in modern computing devices.

The design of new secure and fast hashing algorithms is important for many applications in today’s information-based world. In this thesis, we broadly explore the analysis and design of key security primitives, cryptographic hash functions and digital signatures, used for assuring authentication and data integrity. A novel hashing transformation has been proposed and the security of the proposed algorithms have been analyzed using some well-known statistical tests in the field. Several analyses and computer simulations are performed to verify their characteristics, confirming the satisfaction of the characteristics and conditions of the cryptographic hashing algorithms. In addition to this, tests have been also performed in some of the application areas of hash functions such as the digital signatures and the digital forensic applications.

We have also compared the performance of algorithms based on proposed transformation with the well-known hashing algorithms. Our comparison of results show that the proposed algorithms have a better performance than other existing algorithms. Apart from the speedup gained due to parallel implementation, the energy efficiency of the algorithms has also been measured. Energy-efficient parallel algorithms make them suitable for their use in the handheld devices. The
proposed algorithms in this thesis have a high potential for their adoption in the area of secure and forensic applications.
CHAPTER 1
INTRODUCTION
Chapter 1

INTRODUCTION

The development and usage of computers have been taking place at a remarkable pace since the advent of the computer. The number of computers and its users have grown exponentially over the years. Information/data no longer resides on paper, whether it’s a government organization or private sector or personal information. There is a huge amount of information on the internet and it travels through the networks from one place to another.

Before the arrival of data processing tools, the security of vital information of an organization was primarily provided by physical means like lockers, signatures, and safe boxes. But now, with the development of a number of computerized data processing tools, it has become necessary to deploy automated tools in order to protect the files and all other sorts of information stored on computer or internet. The security of digital information is maintained while it’s static and dynamic. For static security, the information stored on computers must be secure, and network security protects data during their transmission to provide security in the dynamic context.

The security of digital information is not just a service but is a collection of various services provided altogether. These services include: authentication, access control, data confidentiality, non-repudiation, and data integrity. A system has to ensure all or at least one of these depending upon the requirement of the security for a particular system. For example, in addition to the encryption of the data, we may also need authentication and data integrity checks for most situations of data in motion. There has been tremendous growth in the development of cryptographic hash algorithms to ensure authentication and data integrity services as part of ensuring information security.

For ensuring data integrity, SHA-1 and MD5 (William and Stallings, 2006) are the common algorithms which are being used for over a decade in various applications and protocols. Digital Signature, password protection, digital forensics, SSL protocol,
micropayment are some of the applications in which these algorithms are used. There have been several efficient advances in these algorithms to speed up the overall process and to secure it from attacks. This chapter presents an overview of cryptographic hash algorithms and their software and hardware implementations to achieve the goals of improved security and performance gains.

1.1 Background

The transition from physical data storage to digital data storage has led to changing requirements in its security as well. Some of the automated information systems are being used to replace the traditional security measures associated with physical data like signatures, keys, dates, code words etc. They provide these security measures (preserving the integrity, availability and confidentiality) in a technical form to all the information system resources like hardware, software and firmware. The main objective is to provide the security to data processing systems and to information transfers of an organization which are likely to face security attacks. The key security services in a digital information systems include:

- **Authentication** – to assure that communicating entity is the one who has claimed
- **Access Control** – to prevent the unauthorized use of the resources
- **Data Confidentiality** – to protect information from unauthorized disclosure
- **Data Integrity** – to assure that message received is same as sent by an authorized body
- **Non-Repudiation** – to protect against denial by any of the parties in a communication
- **Availability** – to assure resource accessibility/usability

These security services can be achieved by using one or more of the security mechanisms. Security mechanisms are features designed to identify, avert, or recuperate from a security attack. Cryptographic techniques are elements that
underlie many of these security mechanisms. Cryptography is an art of disguising a message, hiding information such that it is not readable by any unauthorized party. According to Menezes et al. (2010), three broad areas of study in cryptography have been defined: symmetric cryptosystems, asymmetric cryptosystems, and keyless cryptosystems. Symmetric cryptosystems use a single private key to convert a plain text to cipher text (i.e. in disguised form) whereas asymmetric cryptosystems uses set of two keys, public and private, for the encryption and the decryption processes, respectively. Both of these primitives provide secrecy as a service. Cryptographic Hash Functions (CHFs) act as symmetric primitives when using keyed hash functions and as keyless primitives when using keyless hash functions. Ideally the problem of message authentication, message integrity, and confirming the identity of the sender in ecommerce applications is perhaps more important than concealment. Message authentication is more concerned towards: protecting the reliability of a message, validating identity of initiator, and non-repudiation of an originator and CHFs ensure these services.

1.2 Cryptographic Hash Functions

Cryptographic Hash functions (CHFs) (Schneier, 2007) are one of the building blocks for many protocols and are commonly known as compression functions, message digest, fingerprint, cryptographic checksum, Message Integrity Check (MIC), and Manipulation Detection Codes (MDC).

CHF (informally), is a function that takes a unit of data or a message as an input and returns a fixed-size hash or a unique code, known as Message Digest (as shown in Fig. 1.1). More precisely, a hash function $H$ maps bit-strings of arbitrary length from a domain $D$ to strings of fixed length ($n$) in range $R$ with $H: D \rightarrow R$ and $|D| > |R|$. It is considered, relatively easy to compute a hash value $h$ for a given message $M$ through the use of CHF. Any accidental or intentional change to the data can lead to a complete change in the hash value which is useful in ensuring data integrity. These hash functions can be broadly classified into two classes: keyless hash functions and keyed hash functions.
**Definition 1.1:** Let a set of bit strings be denoted as $\{0,1\}^*$ and a message $M \in \{0,1\}^*$, then a *keyless hash function* accepts a message $M$ and generate $H: \{0,1\}^* \rightarrow \{0,1\}^n$, a unique value or fingerprint where $n$ is a positive integer for that message.

**Definition 1.2:** Let a set of bit strings be denoted as $\{0,1\}^*$ and a message $M \in \{0,1\}^*$, then a *keyed hash function* accepts a message $M$, fixed length key $k$ and generates $H_k: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$, a unique value or finger-print where $n$ is a positive integer for that message.

In addition to the above mentioned classification of hash functions, a functional classification of hash functions is also given by Menezes et al.(2010) in which hash functions are further divided as:

1. Manipulation Detection Codes (MDCs): They are also known as modification detection codes or message integrity checks with the goal of assuring data integrity. MDCs are a subclass of keyless hash functions and are further classified as *one-way hash functions* (OWHFs) and *collision resistant hash functions* (CRHFs) (formally defined in Section 1.2.1).

2. Message Authentication Codes (MACs): They are a subclass of keyed hash functions with the goal of assuring the source of message along with data integrity. MACs are MDCs with a secret key as an extra input with the message.
Figure 1.2 illustrates the simplified classification of Cryptographic Hash Functions. Some of the common security applications of CHFs include digital signatures, message authentication codes (MACs), fingerprinting, forensic applications, SSL protocol, and checksums to detect any accidental data corruption etc.

1.2.1 CHF Security Notions

A CHF is expected to preserve the few basic universal security properties to make it compatible for cryptographic operations. To be an ideal CHF, an algorithm has to satisfy the following significant properties (William and Stallings, 2006):

1. **Pre-image Resistance**: Given a hash value $h$ where $h \in \{0, 1\}^n$, it is practically infeasible to generate input message $M$ such that $H(M) = h$.
2. **Weak Collision Resistance**: Given a message $M \in \{0, 1\}^*$, it is hard to find another message, $M' \in \{0, 1\}^*$, such that $H(M) = H(M')$ and $M \neq M'$.
3. **Collision Resistance**: It is practically infeasible to find two messages, $M, M' \in \{0, 1\}^*$ and $M \neq M'$, such that $H(M) = H(M')$. 


In the literature, the pre-image resistance property is also referred as one-wayness, weak collision resistance as second pre-image resistance and collision resistance is also referred as strong collision resistance (Menezes et al., 2010) or collision freeness (Damgård, 1990). These properties of CHF’s work as security conditions or a check on them as violation of these lead to drastic consequences on the output of a hash function. As if changing any bit or bits in input message: $M \rightarrow M'$ the hash values $H(M)$ and $H(M')$ will affect at least half the hash value bits with a 50% probability of each bit in $H(M)$ being changed in $H(M')$. This effect in hash value is termed as an Avalanche Effect.

On the basis of these security properties, classification of MDCs can be defined as:

**Definition 1.3:** A One-Way Hash Function (OWHF) is a hash function as per definition 1.1 with additional properties of pre-image resistance and weak collision resistance.

**Definition 1.4:** A Collision Resistant Hash Function (CRHF) is a hash function as per definition 1.1 with additional properties of weak collision resistance and collision resistance.

Figure 1.3 illustrates the security requirements of OWHFs and CRHFs. Further work in thesis is based on CRHFs.
Apart from these properties, there are few other desirable properties which can be used to assess the security of CHFs also known as certificational properties (Menezes et al., 2010). Certificational properties can be application specific and are not necessary. These properties are:

1. **Near Collision Resistance:** It is practically infeasible to find two messages $M, M' \in \{0, 1\}^*$, such that $M \neq M'$ and $H(M) \oplus H(M') = \Delta$ for some small difference $\Delta$.

2. **Pseudo-randomness:** The hash $h$ must be deterministic and random for the input passed.

3. **Non-correlation:** The input bits of a message $M$ should not be statistically correlated to the output bits of $H(M)$.

4. **Partial Pre-image Resistance:** Given a hash result $h$, it is computationally difficult to recover any part of the message.

### 1.3 CHF Designs

Importance of CHFs was first realized by Diffie and Hellman (1976) when they invented Public Key Cryptography. Thereafter many new CHF constructions came into existence due to the inherent weaknesses revealed in popular hash functions by the cryptanalysts. One of the simplest hash functions available is the bit-by-bit exclusive-OR (XOR) of every block of the message. The other way could be to perform a one-bit circular shift, or rotation, on the hash code once each block has been processed. Although this procedure gives a good measure of data integrity but ideally it doesn’t provide any information security when the encrypted hash value is used with a simple plaintext message.

Section 1.3.1 and 1.3.2 discusses the popular designs prevalent for generating the hash codes. Apart from these there are many more constructions, but with the rapidly growing literature it is nearly impossible to be more exhaustive. So, the absence of any particular construction in the sections below does not imply that we disfavor it.
1.3.1 Iterative Hash Functions

The most popular conventional way to generate a hash code is to divide the message into multiple blocks and then processing the blocks iteratively and methodically. This iterative sequential hashing approach is still most widely used approach in the presence of parallel processors. Many variants of the approach are prevalent and few popular iterative constructions are discussed in the following sub-sections.

1.3.1.a Merkle-Damgård Construction

Ralph Merkle (Merkle, 1990) and Ivan Damgård (Damgård, 1990) independently proposed the most popular hash function construction where Damgård's construction was keyed and complemented with security proofs while Merkle's was keyless. Although it was claimed that Rabin (Rabin, 1978) has previously proposed the similar construction in 1978, Merkle and Damgård proved that the hash function is collision resistant if its underlying compression function is collision resistant.

In the Merkle-Damgård (MD) construction, a message is divided into multiple blocks of same size and is padded if required. To strengthen the construction, Merkle-Damgård strengthening is used by appending the length of the original message into the last message block to make it collision resistant. This MD strengthening technique was firstly created by Lai and Massey in (Lai and Massey, 1993), although it was already proposed by Merkle (Merkle, 1990) and Damgård (Damgård, 1990). After that the fixed length output compression function $f$ is repeated iteratively according to the domain extender as defined below:

- Compression function: a function $f$ which associates the fixed-length input to a fixed-length output i.e. $f: \{0, 1\}^{l+n} \rightarrow \{0, 1\}^n$, where $l$ is the length of a data block and $f$ maps $l + n$ bits to $n$ bit constant initial vector ($IV$) as shown in Fig. 1.4 and Eq. (1.1).
Domain extender: a generic process that uses the compression function $H$ with fixed-length input iteratively and transforms into a hash function which can handle arbitrary length of input. $H^f: \{0, 1\}^{i+n} \rightarrow \{0, 1\}^n$ associated with compression function. Iterate the compression function until all the blocks of $n$ bits have been hashed as shown in Fig. 1.5 and Eq. (1.2).

There is a long list of CHFs based on MD construction, including SNEFRU, N-hash, MD2, MD4, MD5, SHA family (William and Stallings, 2006). As of now, the most widely used MD construction based CHF has been the Secure Hash Algorithm (SHA) and the MD family. The National Institute of Standards and Technology (NIST)\(^1\) publishes a family of CHFs, the Secure Hash Algorithm as a U.S. Federal Information Processing Standard (FIPS) publication. This family includes number of CHFs, with the advancements done to the function according to the requirements.

\(^1\) National Institute of Standards and Technology, http://www.nist.gov/
These algorithms have gathered an unprecedented interest of its researchers in recent years, as clearly evident by the growing numbers. With advances in hardware and software technologies, attacks are made more feasible on them. The Section 1.4 discusses the SHA’s available and most likely to be carried out in the near future: with SHA-0 in Section 1.4.2, SHA-1 in Section 1.4.3, SHA-2: SHA-256, SHA-378, SHA-512 in Section 1.4.4 and SHA-3 in Section 1.4.5 followed by the attacks on them in their respective sections.

1.3.1.b Variants of Merkle-Damgard Construction

With the advances in cryptanalysis, weaknesses had been reported in popular hash functions (Wang et al., 2004, Wang et al., 2005a, Wang et al., 2005b) and had drove the interest of research community towards new constructions. Most of the new constructions are still iterative and are modified variants of the MD construction. The main motive of the new constructions is always to provide efficiency and prevention from the previous attacks. This section presents few of the modified versions of the MD construction irrespective of any order.

Wide and Double Pipe

Lucks in (Lucks, 2004, Lucks, 2005) proposed a wide/double pipe construction where security of the hash function is improved by increasing the size of intermediate hash states i.e. intermediate hash values is $w$ bits long while the final hash value is $n$ bits long where $w > n$. A second compression function is used to compress $w$ bits to final $n$ bit hash value $g: \{0, 1\}^w \rightarrow \{0, 1\}^n$. As the final hash value is truncated in wide/double pipe construction, so the attacker has to guess the discarded bits to append an extension which helps thwart extension attacks and other generic attacks. But the modification gives rise to an apparent drawback of inefficiency as the intermediate variable has increased, the compression function now has larger input/output while keeping the hashing rate constant.
Prefix Free, Chop, NMAC and HMAC Constructions

Coron et al. presented several constructions in (Coron et al., 2005) as immediate fixes to the MD construction. In the Prefix free construction, the padding algorithm is modified by prepending or appending the length of the complete message to every message block rather than the last block as in MD construction. But this construction couldn’t work well with streaming applications where the length of the message is not known beforehand and efficiency is also degraded by wasting the bits to represent the length of the message in every block. In the Chop Construction, the non-trivial number of bits from the final hash value are removed. However, with this modification, the issue of in-differentiability is resolved but security bounds of the hash function are lowered down.

![Fig. 1.6 NMAC Construction](image)

The NMAC construction has an independent extra function g which is applied to the hash result of the last block, while in HMAC construction, an extra compression function call is added to increase the level of security. Figure 1.6 and 1.7 illustrates Coron’s NMAC and HMAC constructions respectively.

![Fig. 1.7 HMAC Construction](image)

However, later in (Bellare and Ristenpart, 2006) it was shown that these constructions are not collision resistant.
**Enveloped Merkle-Damgård (EMD)**

A multi-property preserving construction called Enveloped Merkle-Damgård construction was proposed by Bellare and Ristenpart in (Bellare and Ristenpart, 2006). The main aim of the construction was to have a hash function scheme which can preserve multiple properties: collision resistance, pseudorandom function, and pseudorandom oracle-ness at the same time if the underlying compression function possesses these properties. This was curtailed from the fact that they were able to prove the constructions presented in (Coron et al., 2005) by Coron et al. do not preserve collision resistance while still being indistinguishable from the pseudorandom oracle-ness aspect. Figure 1.8 illustrates the EMD construction in which two initial vectors are being used, second one for the last block along with the length of the message to strengthen the collision resistance. According to Bellare and Ristenpart (2006), lack of MD strengthening was a major flaw in (Coron et al., 2005), this would have made it multi-property preserving.

**Hash Iterative Framework (HAIFA)**

A modified Merkle-Damgård construction was proposed by Dunkelman and Biham, named HAsh Iterative FrAmework (HAIFA) in (Dunkelman and Biham, 2006). The modification in this construction includes introduction of two extra input parameters: salt (is set to 0 if only one hash function is needed, otherwise used as a key to create families of hash functions), and the number of bits hashed so far. Figure 1.9 illustrates the HAIFA construction in which input to every compression function call becomes unique and thwarts many of the...
generic attacks against the plain MD construction. The main drawback of this framework lies in its inefficiency, as the compression function has more input parameters to process.

1.3.1.c Sponge Construction

Different from MD Construction design principles, Sponge Construction (Bertoni et al., 2007) is a simple iterated construction for building a function with variable length input and variable output length based on fixed length transformation or permutation $f$ operating on a fixed number of $b$ bits called width, and a sponge complaint padding rule pad. The sponge construction as illustrated in Fig. 1.10, operates on bitrate $r$ and capacity $c$ on a state $b = r + c$ bits which produces transformation/permutation of $b$. Values of $r$ and $c$ gives tradeoff in speed and security, higher bitrates give fast but less secure cryptographic functions.

Sponge hashing basically proceeds in two phases: absorbing phase and squeezing phase. In absorbing phase, the message is divided into $r$ bit blocks after padding if required and is XORed with first $r$ bits of $b$ (where initially all the bits of state are initialized to zero), interleaved with function $f$ and iterated until all the blocks have finished. In squeezing phase, $r$ parts of the state are returned after every iteration as output blocks after transformation/permutation by $f$. As the construction supports variable length output, the choice of the length of final hash value depends upon the user which further determines the number of returned blocks in the squeezing phase which need to be returned.
Keccak (Bertoni et al., 2013) is the latest CHF based on Sponge Construction selected as SHA-3 in the competition organized by NIST (discussed in Section 1.4.5).

1.3.2 Tree-based Hash Constructions

Tree-based constructions are considered to be one of the most parallelizable class of constructions that are suitable for multi-core platforms. They are appropriate for applications where a large amount of data needs to be hashed. Tree based hashing was firstly proposed by Merkle with the name Strengthened Merkle Tree (Merkle, 1980), and it was a narrow-pipe keyless domain extender. Strengthened Merkle Tree is commonly used in many MD based constructions. Later, Damgård (1990) and Andreeva et al. (2007) proved that the tree hash preserves collision resistance. After this advancement, some optimized versions of the hash functions came into existence, proposed by Sarkar and Schellenberg (Sarkar and Schellenberg, 2001) and then by Pal and Sarkar (Pal and Sarkar, 2003).

Universal One Way Hash Functions (UOWHFs) built by Bellare and Rogaway (Bellare and Rogaway, 1997) are also based on tree hashing and are narrow-pipe keyed hash functions. These hash functions have proven to be suitable for
many applications even though they are weaker than collision-resistant hash functions. After that much work has been done on UOWHFs which include (Lee et al., 2003) and (Sarkar, 2004).

Bellare and Micciancio had proposed a “randomize and combine” model in (Bellare and Micciancio, 1997) known as XOR-tree Hash Function. It is also narrow-pipe keyed hash function in which the message is split into blocks, randomized individually and finally combined by an XOR operation. It is known to have been broken by a linear algebra attack for long messages (Manuel and Sendrier, 2007). Building of an incremental hash model was the main motive behind its construction but was considered parallelizable because of the independence of individual blocks in randomization process. The latest hash algorithms called the SHA-3, Skein (Ferguson et al.), and MD6 (Rivest et al.) are all based on the tree-hashing model alongside the traditional iterative model (discussed in Section 1.7).

1.4 Popular Cryptographic Hash Algorithms

An extensive survey has been done on the popular cryptographic hashing algorithms and cryptanalysis performed on them. The following sub-sections cover the most prevalent sequential hash functions and improvements done on them along with the attacks performed on them.

1.4.1 MD5

Message Digest 5 is a hashing algorithm which generates a hash of 128-bit and was mainly used in digital signatures. The output of the MD5 algorithm is used to authenticate the owner of a private key and to check the integrity of files. The MD5 algorithm breaks a file into 512 bit input blocks and each block goes through a series of functions to produce a unique 128 bit hash value for that file. It was created with a property that a single bit of change in file completely alters the hash. Furthermore, it was believed that since the key size of 128 bits has 3.4*1038 possible combinations, the chance of randomly finding two files that
produce the same hash value should be computationally impossible (Schneier, 2007). However, Wang et al. in (Wang and Yu, 2005, Hawkes et al., 2004) proved that finding collisions for MD5 can be quite easy and further leading to extensive research interests in new secure hash functions. Later these theoretical attacks were also proven in practice with collisions in (Klima, 2006, Stevens et al., 2009).

1.4.2 SHA-0

In 1993, NIST developed the Secure Hash Algorithm (SHA) and published it as the FIPS 180 publication of NIST. The design of the algorithm is closely based on the MD4 hash function. It was initially built for use with the Digital Signature Standard (DSS) but due to an undisclosed flaw, it was withdrawn after two years and was replaced by a revised version and its name was changed to SHA-1. There was just one instruction more in SHA-1 than in SHA-0 but there were no reasons to prefer the initial version and so SHA-1 came into existence.

1.4.2.a Attacks on SHA-0

An attack on SHA-0 was firstly presented at CRYPTO 98 (Chabaud and Joux, 1998), in which authors proved that collisions of the hash values can be found with complexity $2^{61}$, which is much fewer than $2^{80}$ for a perfect CHF of the same size.

In year 2004 again, Biham and Chen (2004) found two different messages that hash to closely the same value with 142 out of the 160 bits are equal, a near-collision for SHA-0. Then, on 12 August 2004, Joux et al. (2004) announced a collision for the full SHA-0 algorithm. The chances of the collision was 1 in $2^{51}$ and a time of about 80,000 CPU hours was taken on a supercomputer with 256 Itanium dual processors by generalizing the Chabaud and Joux attack.

Later an attack with a chance of 1 in $2^{40}$, better than the previous one was announced by Wang et al. (2004) on MD5, SHA-0, and other hash functions on
17 August 2004, at the CRYPTO 2004 Rump Sessions. Again in February 2005, an attack of $2^{39}$ complexity was proclaimed by Wang et al. (2005b) in SHA-0.

Table 1.1 shows in brief the various attacks befallen on SHA-0 algorithm by different researchers in the corresponding years with their respective complexities.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Attacker/Publisher</th>
<th>Outcome</th>
<th>Year</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Florent Chabaud and Antoine Joux</td>
<td>Collisions with complexity $2^{61}$</td>
<td>1998</td>
<td>Differential collisions in SHA-0</td>
</tr>
<tr>
<td>2.</td>
<td>Biham and Chen</td>
<td>Full Collisions of 65 round and collision with 142 bits equal</td>
<td>2004</td>
<td>Near-collisions of SHA-0</td>
</tr>
<tr>
<td>3.</td>
<td>Joux, Carribault, Lemuet, and Jalby</td>
<td>Collisions with complexity $2^{51}$</td>
<td>2004</td>
<td>Collision in SHA-0</td>
</tr>
<tr>
<td>4.</td>
<td>Wang, Feng, Lai, and Yu</td>
<td>Collisions with complexity $2^{40}$</td>
<td>2004</td>
<td>Collisions for hash functions MD4</td>
</tr>
<tr>
<td>5.</td>
<td>X. Wang, Y. Lisa Yin, and H. Yu</td>
<td>Collisions with complexity $2^{39}$</td>
<td>2005</td>
<td>Efficient collision search attacks on SHA-0</td>
</tr>
</tbody>
</table>

After so many collision attacks with progressively reducing complexity, SHA-0, MD4, and MD5 were considered to be insecure for further use in authentication purposes.

### 1.4.3 SHA-1

SHA-1[RFC3174] was designed by the National Security Agency (NSA) and published by NIST (NIST, 2002) as FIPS the 180-1 publication in 1995. It is also standardized as a dedicated CHF in ISO/IEC 10118. The design of the algorithm was again based on the MD4 algorithm with some nods from MD5, which is almost the same as MD4. The compression function of SHA-1 is based on a block cipher and its domain extender on Merkle-Damgård construction. The maximum message/file size for the algorithm is $2^{69}$-1 bits and produces a message digest/fingerprint of 160 bits.
The compression function takes the block of size 512 bits as an input which is further subdivided into sixteen 32-bit blocks. The number of rounds for the updates of the internal state are 80 in SHA-1. The SHA family is now over to “big-endian”. The single round of the compression function in the algorithm is shown in Fig.1.11, which transforms the five 32-bit variables to form the final hash value.

1.4.3.a Attacks on SHA-1

After the attacks were found on SHA-0, experts suggested that the usage of SHA-1 in forthcoming cryptosystems should be given a second thought. The results of CRYPTO 2004 insisted NIST to announce the use of SHA-2 variants and phase out the use of SHA-1 by 2010.

Then after the announcement of SHA-1, an attack was published (Rijmen and Oswald, 2005) on a reduced version of SHA-1 that had 53 out of 80 rounds, which could basically find collisions with a computational effort of less than $2^{80}$ operations.
Wang et al. (2005a) announced another attack on the full version of SHA-1 in February 2005. In the attack collisions can be found requiring $< 2^{69}$ operations. Whereas a brute-force search would require $2^{80}$ operations to find a collision.

On behalf of X. Wang, A. Yao, and F. Yao, an announcement was made on lowering of the complexity to $2^{63}$ that is required for finding a collision in SHA-1 on 17th August 2005, at the CRYPTO 2005 rump session. Later in December 2007 this announcement and its results were explained in detail (Cochran, 2007).

A significant theoretical attack was presented (De Canniere and Rechberger, 2006) at ASIACRYPT2006. They proposed a two-block collision for 64-round found by using un-optimized methods with $2^{35}$ compression function evaluations. Grechnikov further extended their attack to 73 rounds (out of 80) in 2010 in response to catch a collision in the full 80 rounds of the hash function.

In 2008, Stéphane Manuel announced an attack of hash collisions with a projected theoretical complexity of $2^{51}$ to $2^{57}$ operations (Manuel, 2011) but later when he found that the that local collision paths were not in reality autonomous he just withdrew the claims made.

In (McDonald et al., 2009), the authors claimed a hash collision attack with complexity of $2^{52}$ at the Rump session of Eurocrypt 2009. But later on authors discovered the incorrect estimate and withdrew that paper as well.

In November 2010, Marc Stevens also claimed a completely working near-collision attack against full SHA-1 with a projected complexity equivalent to $2^{57.5}$ SHA-1 compressions. He developed a project HashClash by making use of CPU power from cloud servers to break a single hash value of the SHA-1 algorithm.
Table 1.2 shows in brief the various attacks befallen on SHA-1 algorithm by different researchers including the corresponding years along with their respective complexities.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Attacker/Publisher</th>
<th>Outcome</th>
<th>Year</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Rijmen and Oswald</td>
<td>Collisions possible for 53 rounds instead of 80</td>
<td>2005</td>
<td>Update on SHA-1</td>
</tr>
<tr>
<td>2.</td>
<td>Xiaoyun Wang, Yiqun Lisa Yin and H. Yu</td>
<td>Collisions with complexity $&lt; 2^{69}$ operations</td>
<td>2005</td>
<td>Finding collisions in the full SHA-1</td>
</tr>
<tr>
<td>5.</td>
<td>Manuel, Stéphane</td>
<td>Already known attack</td>
<td>2008-2011</td>
<td>Classification and generation of disturbance vectors for collision attacks against SHA-1</td>
</tr>
<tr>
<td>6.</td>
<td>Cameron McDonald, Philip Hawkes and Josef Pieprzyk</td>
<td>Paper withdrawn, estimate was incorrect</td>
<td>2009</td>
<td>Differential Path for SHA-1 with complexity $O(2^{52})$</td>
</tr>
<tr>
<td>7.</td>
<td>Marc Stevens</td>
<td>Complexity equivalent to $2^{57.5}$</td>
<td>2010</td>
<td></td>
</tr>
</tbody>
</table>

These attacks have pushed the transition of cryptographic world to newer and stronger versions of SHA. But still the algorithm is used in a wide variety of applications including Digital Signatures, TLS, SSL, SSH and PGP.
1.4.4 SHA-2

Three new revised versions of SHA were added into SHA family by NIST in August 2002 as FIPS 180-2, known to be SHA-256, SHA-384, and SHA-512 with the respective hash value lengths of 256, 384, and 512 bits. Later in 2008, FIP PUB 180-3 was issued as a revised document, which added SHA-224 [RFC3874] into the family. Collectively, these algorithms are recognized as SHA2.

The new versions bear the same underlying resemblance of structure, modular arithmetic and logical binary operations as that of SHA-1 without sharing its weaknesses. The algorithms SHA-256 and SHA-512 caries the same basic design with the difference that SHA-256 operates on eight 32-bit words, while SHA-512 operates on eight 64-bit words designed especially for the 64-bit processor. SHA-384 is designed with a slight modification in SHA-512, which is composite of different initial value of the chaining variable and reduction in the hash code to 384 bits. SHA-224 is again a trimmed version of SHA-256 algorithm with a different initial value.
These hash functions are targeted to provide 112-bit level of security. Apart from the hash size and initial value the four new functions differs from SHA-1 in the process of deriving sub-blocks from a block of a message. Figure 1.12 shows a single round of compression function of the SHA-2 family variant.

1.4.4.a Attacks on SHA-2

SHA-2 family has also faced cryptographic attacks partly resulting from the SHA-3 competition which provoked the researchers/attackers to work on the analysis of SHA-2 variants. But as of now, only collision attacks found are with the practical complexity and yet none of the other attacks on the hash functions are found using complete set of rounds. Some of these attacks are listed in Table 1.3 along with the year of attack, method, and complexity for collision.

In Cryptology-INDOCRYPT 2008, Sanadhya, Somitra Kumar, and Palash Sarkar presented a deterministic collision in 24/64 rounds with $2^{28.5}$ complexity of SHA-256 and in 24/80 rounds with $2^{32.5}$ complexity of SHA-512 (Sanadhya and Sarkar, 2008).

Then in ASIACRYPT 2009, "Preimages for step-reduced SHA-2" (Aoki et al., 2009) was presented which discusses the about meet-in-the-middle attack on SHA-256 and SHA-512 with different complexities.


Pseudo collision differential attack was presented in "Higher-Order Differential Attack on Reduced SHA-256" (Lamberger and Mendel, 2011) on SHA-256 with $2^{178}$ and $2^{46}$ complexity.

In 2011, Khovratovich, Dmitry, Christian Rechberger, and Alexandra Savelieva also presented various attacks on SHA-256 and SHA-512 in their paper (Khovratovich et al., 2012).
Table 1.3 Attacks made on SHA-2 Variants

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Paper Description</th>
<th>Year</th>
<th>Attack Method</th>
<th>Variant of SHA-2</th>
<th>Collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>New Collision attacks Against Up To 24-step SHA-2</td>
<td>2008</td>
<td>Deterministic Collision</td>
<td>SHA-256</td>
<td>In 24/64 rounds with $2^{28.5}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-512</td>
<td>In 24/80 rounds with $2^{32.5}$ complexity</td>
</tr>
<tr>
<td>2.</td>
<td>Preimages for step-reduced SHA-2</td>
<td>2009</td>
<td>Pre-image, Meet-in-the-middle</td>
<td>SHA-256</td>
<td>In 42/64 rounds with $2^{291.7}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-512</td>
<td>In 43/64 rounds with $2^{294.9}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-256</td>
<td>In 42/80 rounds with $2^{302.3}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-512</td>
<td>In 46/80 rounds with $2^{311.5}$ complexity</td>
</tr>
<tr>
<td>3.</td>
<td>Advanced meet-in-the-middle preimage attacks</td>
<td>2010</td>
<td>Pre-image, Meet-in-the-middle</td>
<td>SHA-256</td>
<td>In 42/64 rounds with $2^{248.4}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-512</td>
<td>In 42/80 rounds with $2^{494.6}$ complexity</td>
</tr>
<tr>
<td>4.</td>
<td>Higher-Order Differential Attack on Reduced SHA-256</td>
<td>2011</td>
<td>Pseudo Collision, Differential</td>
<td>SHA-256</td>
<td>In 46/64 rounds with $2^{2178}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In 46/64 rounds with $2^{46}$ complexity</td>
</tr>
<tr>
<td>5.</td>
<td>Bicliques for Pre-images: Attacks on Skein-512 and the SHA-2 family</td>
<td>2011</td>
<td>Pre-image, Biclique</td>
<td>SHA-256</td>
<td>In 45/64 rounds with $2^{555.3}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-512</td>
<td>In 50/80 rounds with $2^{511.5}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-256</td>
<td>In 52/64 rounds with $2^{555}$ complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-512</td>
<td>In 57/80 rounds with $2^{511}$ complexity</td>
</tr>
</tbody>
</table>

1.4.5 SHA-3

With the motivation from collision attacks on commonly used hash algorithms MD4, MD5, SHA-0 and theoretically attacked SHA-1, NIST announced a public competition in the Federal Register in November 2007 to develop a new hashing algorithm called SHA-3. SHA-3 is not meant to be directly linked with SHA-2 family or to replace it but will be preserving some of the properties of
SHA-2. There were 64 submissions for the competition in October 2008, 51 candidates were accepted for the first round, 14 semi-finalists were selected in 2009, 5 finalists were selected- BLAKE (Aumasson et al., 2008), Grøstl (Gauravaram et al., 2008), JH (Bhattacharyya et al., 2010), Keccak (Bertoni et al., 2013) and Skein (Andreeva et al., 2012) in December 2010. On October 2, 2012, Keccak was announced as the winner of the competition by NIST (Boutin, 2012).

Keccak was designed by Guido Bertoni, Joan Daemen and Gilles Van Assche of STMicroelectronics and Michaël Peeters of NXP. It has been found that Keccak has better performance in hardware implementations than the competitors and predecessors. It has an elegant design with the ability to execute effectively on different computing devices.

The algorithm uses the sponge construction (Bertoni et al., 2007) (discussed in Section 1.3.1.c) rather than the more famous Merkle-Damgård construction. For SHA-3 competition, authors have proposed the largest permutation size of 1600 in the algorithm named as Keccak-f[1600]. The message blocks are firstly XORed into a subset of state of a 5×5 array of 64-bit and then permuted as a whole. In each permutation there is an iteration of a simple round function including operations like bitwise XOR, AND, NOT and rotations. The efficiency in hardware performance of Keccak can be checked by the work done by Gürkaynak et al.(2012), Gaj et al.(2012), Latif et al.(2012), Kavun et al.(2012), Kaps et al.(2012) and Jungk(2012) presented at the Third SHA-3 Candidate Conference. It also gives better software performance than SHA-2 on modern multicore processors with 128-bit and 256-bit of hashing at 4.8 and 5.9 cycles/byte, respectively on a single AMD FX-8120, Bulldozer @3.1GHz core and 5.4 and 6.9 cycles/byte for the same size on a single Intel Xeon E3-1225, Sandy Bridge @3.1GHz core. There are good counter-measures like quadratic round functions and no table look-ups in the keyed Keccak to protect against power analysis attacks, cache-timing attacks and other variant attacks.

As of April 2014, a separate SHA-3 standard has been updated by NIST as Draft FIPS Publication 202 and the contents are yet to be finalized for the standard.
A complete comparison of SHA functions and their variants with respect to the hash size, message size, number of rounds, operations, security and performance has been provided in Table 1.4 for the reader’s reference.

The table shows standard CHF’s at a glance that are prevalent these days. Conventional CHF’s are based on MD construction (except the SHA-3 which is based on Sponge Construction) which makes the hash generation process serial.

Table 1.4 Comparison of SHA Variants

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SHA-0</th>
<th>SHA-1</th>
<th>SHA-2</th>
<th>SHA-3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>SHA-224</td>
<td>SHA-384</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SHA-256</td>
<td>SHA-512</td>
</tr>
<tr>
<td></td>
<td>SHA-3-224</td>
<td>SHA-3-256</td>
<td>SHA-3-384</td>
<td>SHA-3-512</td>
</tr>
<tr>
<td>Hash size (in bits)</td>
<td>160</td>
<td>160</td>
<td>224</td>
<td>384</td>
</tr>
<tr>
<td>Internal state size (no. of variables * size in bits)</td>
<td>(5×32)</td>
<td>(5×32)</td>
<td>(8×32)</td>
<td>(8×64)</td>
</tr>
<tr>
<td>Block size (in bits)</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Max message size</td>
<td>$2^{64} - 1$</td>
<td>$2^{64} - 1$</td>
<td>$2^{64} - 1$</td>
<td>$2^{128} - 1$</td>
</tr>
<tr>
<td>No. of Rounds</td>
<td>80</td>
<td>80</td>
<td>64</td>
<td>80</td>
</tr>
<tr>
<td>Operations</td>
<td>add mod 23 2, and, or, xor, rot</td>
<td>add mod 232, and, or, xor, rot</td>
<td>add mod 232, and, or, xor, shr, rot</td>
<td>add mod 26 4, and, or, xor, shr, rot</td>
</tr>
<tr>
<td>Security</td>
<td>&lt;80 (collisions found)</td>
<td>&lt;80 (theoretical attack in $2^{61}$)</td>
<td>112</td>
<td>192</td>
</tr>
<tr>
<td>Performance (MB/s)</td>
<td>-</td>
<td>192</td>
<td>139</td>
<td>154</td>
</tr>
</tbody>
</table>
1.5 Current Issues in Hashing Algorithms

Wang et al. proved attacks on MD5 and SHA-1 (Wang et al., 2005a), since then researchers have been striving hard to provide hashing algorithms which are secure enough to be used in the financial industry. A lot of work is being done on the security of these algorithms. According to the research and reports regarding the security evaluation for cryptographic algorithms, it is challenging to ensure the security if the algorithms are used for a long time period, of 10 to 15 years. But due to the advanced progress in cryptanalysis techniques, improvements in computing power, hardware and ever-growing amount of digital information, secure algorithms are no longer sufficient. Secure, fast and efficient hashing techniques are also required.

In CHF, performance and especially speed is very crucial. Hashing functions have been proven quite fast, but many of them do not make optimum utilization of modern architectures, as a result many CPU cycles get wasted. The main cause of wasted CPU cycles is due to the serial nature of the CHFs currently in use.

High energy consumption is another major issue affecting modern computing systems. Energy efficient hashing algorithms may prove to be a new benchmark for use in the high-end servers and data centers where cooling cost is very high. Moreover, it can extend battery life in mobile and embedded systems as well as in devices which have a limited battery life.

1.6 Motivation for Proposed Research

Information has now become a sort of asset for a business or an individual. As a result, information asset security, along with integrity and authenticity is an important concern due to the growing connectivity amid computer users. Regardless of the number of security algorithms proposed, the hazards between security and performance demands ongoing research towards its improvement (Dongara and Vijaykumar, 2003).
CHFs have moved their roots deep into financial industry, digital forensics, network protocols, password management and other information authentication and integrity demanding application areas. With the increasing size of information, secure processing of the data takes a much longer time than non-secured processing. This is due to the combination of different cryptographic operations, which include symmetric, asymmetric encryption operations and hash functions. Therefore, achieving high performance in cryptographic processing has increasingly become important.

For high-end servers and data centers where security is of utmost importance, increasing integration and power dissipation consequently increases on-chip temperature which in turn increases the cooling cost of these systems. Cost effective cooling techniques are applied but they limit the performance of computing systems. Even expensive cooling techniques do not match the growing power demands of these systems (Weste and Harris, 2005). Power consumption and subsequently heat generation by the computers has become a important concern in recent years, parallel computing has emerged to reach high performance targets while also maintaining acceptable power characteristics.

Moreover, in handheld devices which have a limited battery life, security algorithms eat up battery very quickly. Therefore, parallel computing can be used for more energy efficient implementation of security algorithms making them more usable in areas like mobile computing, data centers, mainstream computers, and high-end servers, etc.

“Without parallel hardware, various components of a computer system cannot operate at high speed” (Comer, 2005).

Present computers can exploit parallelism to achieve high performance, however lack of parallelism can obstruct achieving fast response times. As hashing algorithms are more compute intensive, so parallelizing them will lead to a fast response time and will also help to reduce the power consumption which is one of the most critical aspect these days.
However, as researchers are paying more attention on the secure and efficient implementation of hash functions, very less work has been done in their parallelization. It is observed that there is a great need of significant performance gain in huge data transfers without conceding the security when the size of digital information is increasing exponentially and availability of hardware resources quadratic-ally. This issue can only be dealt with by using the power of multiple cores available in current computing devices. By making the process of hashing even faster on modern processors can open up the doors to possibly more new uses, and make current uses more convenient.

The thesis presents a new approach to generate hash code in parallel to increase the efficiency and to speed up the hash generation process.

### 1.7 Literature Review

Traditionally CHFs were iterative indeed but with the hasty advent of parallel systems, parallelizable CHF’s are also increasing rapidly. This section covers in nut shell the parallelization of CHF’s both at hardware and software level. As such, there is not much work in the parallelization of CHF’s but SHA-3 competition has drew everybody’s attention towards this. Researchers are striving hard to parallelize the hashing process and make optimum use of the power of multi-core processors that are common today. In this section we have tried to discuss few of the efforts made to parallelize CHFs both at the hardware and the software levels.

In the paper (Bosselaers et al., 1996), presented that with the arrival of the Pentium processors parallelization became possible for Intel-based computer systems. In return, cryptographic hash functions - MD4, MD5, and SHA-1 also became faster on the 32-bit processors. As the implementations of these algorithms were easily able to exploit the power of Pentium processors with the performance gain of approximately 60 percent as compared to execution on non-parallel architectures. They have also shown that 10 percent of running time performance penalty is sustained by non-cached data and on the endianness conversion.
In contrast to the above claims, in the paper (Bosselaers et al., 1997) “SHA: A Design for Parallel Architectures?” the authors discussed that the implementation of MD4 based CHFs RIPEMD-128, RIPEMD-160, and SHA-1 contains more software level parallelism. They estimated that the parallelism observed in SHA-1 was a design principle and realizing it will require a 7-way multiple-issue architecture. They have also shown that as the organization of RIPEMD-160 is in two independent lines, future architectures could easily achieve its software parallelism.

In (Nakajima and Matsui, 2002), authors presented an exhaustive software performance analysis of CHFs MD5, RIPEMD-128 -160, SHA-1, SHA-256, SHA-512 and Whirlpool on a Pentium III processor. In order to optimize the speed of 32-bit oriented hash functions, they have used pipeline scheduling and MMX registers for processing few of the message blocks in parallel. For 64-bit algorithms- SHA-512 and Whirlpool, they had utilized the 64-bit MMX instructions to maximize the performance. A complete analysis has been provided, which was first for SHA-512 and Whirlpool for its own.

In 2004, Praveen S.S. Gauravaram, William L. Millan, and Lauren J. May proposed a new cryptographic algorithm CRUSH (Gauravaram et al., 2004). In contrast to standard Merkle Damgard construction based algorithms which could prove to be easier in the hands of cryptanalysts, the proposed algorithm was based on iterated halving (IH) to ensure security and efficiency. The authors have claimed to achieve a secure CHF with equivalent rate of 1, when the internal F-function of IH is instantiated with a half-complexity block cipher. According to them, 120 Mbps of speed was achieved with an initial un-optimized implementation of the algorithm.

In 2006, in (Hong-Qiang and Chang-yun, 2006) a hardware implementation of SHA 512 was proposed. The implementation was integrated by using VHDL and synthesized and routed in an FPGA device which showed high speed performance. Another implementation on FPGA device was given in (Khalil et al., 2008), in which a digital signature security scheme has been discussed by
using a public-key crypto System-on-Chip (SoC), having SHA-2 hash core in combination with a 2048-bit RSA co-processor. The crypto SoC was implemented on an Altera Nios II Stratix FPGA-based prototyping system running on a 50 MHz system clock, which showed a throughput of 644 Mbits/sec for the SHA-512 hardware core.

The hardware optimization techniques such as pipelining and unrolling were used in (McEvoy et al., 2006) to present a new VLSI architecture for the SHA-256 and SHA-512 hash functions. The processors were developed for implementation on the FPGAs and the results were analyzed and compared with other FPGA-based implementations. The aim of changing hardware implementation was to improve CHF’s performance but these techniques were weak in exploiting the parallelism in them. So it was felt that there is a great need of secure and fine granularity of parallelism for CHF.

Another way of improving the performance was adopted by the use of GPUs (Graphic Processing Unit). In 2009, an implementation of MD5-RC4 encryption was given using NVIDIA GPU cards in (Li et al., 2009, Hu et al., 2009). A performance gain of about 3-5X was measured on GeForce 9800GTX card. In (Hu et al., 2009), Hu implemented a parallel MD5 on CUDA-enabled GPU based on task stream or task block.

Liu et al. (2009) proposed a parallel digital signature method using parallelizing SHA based on content chunk for improving de-duplication storage system performance and by storage pipelining. The granularity of parallelism proposed by both Hu and Liu is coarse despite of both had restrained to exploit parallelism of CHF among intra-stream or intra-task while inter-message is also performed in serial.

In 2009, Du proposed a block cipher based fine granularity of parallelism for CHF (Du et al., 2009). Theoretical analysis and computer simulation are given to prove the security and performance of the proposed algorithm. The author has claimed it to be good choice for e-commerce applications, but unfortunately
his reduction method cannot ensure the security for CHF when the number of message blocks varies.

Hashem Mohammed et al. have proposed a parallel algorithm for improving Security Performance in SSL Bulk Data Transfer in (ALAILAROS, 2007). They have proposed a framework in which encryption of the information and calculation of its MAC is done in parallel. When the calculation of MAC is over then it is also encrypted. The algorithm was simulated on two different processors with one processor performing the MAC calculation and the other one encrypting the data, simultaneously.

Yantao Li et al. proposed and analyzed a chaotic based parallel hash algorithm framework with changeable parameters in paper (Li et al., 2011). The main key features of the proposed algorithm were its parallel processing mode and message expansion. Firstly the algorithm converts the expanded message blocks into the respective ASCII codes and then in order to generate intermediate hash values, iterates the chaotic asymmetric tent map and then the chaotic piecewise linear map, uninterruptedly, with the dynamically obtained changeable parameters from the position index of the respective message blocks, generate decimal fractions, rounds the decimal fractions to integers, and finally cascades the integers. XOR operation is performed to produce the final Hash value of length 128-bit. The authors have claimed, good statistical properties, collision resistance, and security against meet-in-the-middle attacks through theoretical analysis and computer simulations for the algorithm.

Blake (Aumasson et al., 2008) one of the SHA-3 finalist, was based on Dan Bernstein's ChaCha stream cipher. In the algorithm before each ChaCha round, an input block after permutation is XORed with some round constants and added. There were two variants of this algorithm, a 32-bit BLAKE-256 and a BLAKE-224 with output hash sizes of 256 and 224 bits and a 64-bit BLAKE-512 and BLAKE-384 with output hash sizes of 512 and 384 bits respectively. Some collision attacks were also presented in (Vidali et al., 2010) on the BLOKE and BRAKE versions of BLAKE.
Then later, BLAKE2 was presented, as an improved version of the BLAKE algorithm. The authors claim to have highest security like SHA-3 and speed like that of MD5 on 64-bit systems with at least 33% less RAM than SHA-2 or SHA-3 on low-end systems. The algorithm is based on the same concept as that of ChaCha of its predecessor BLAKE. There are also two variants of the BLAKE2 algorithm: BLAKE2b (BLAKE2) for 64-bit platforms producing hash of any size ranging 1 to 64 bytes and BLAKE2s for 8 to 32 bit platforms producing hash of any size ranging 1 to 32 bytes. The algorithms give increased performance on parallel systems with capability of keyed hashing, hashing with a salt, updatable or incremental tree-hashing, or any combination thereof. According to Aumasson et al. (2013) BLAKE2 provides up to 890 MB/s on a single Intel Xeon E3-1225, Sandy Bridge @3.1GHz core, and up to 559 MB/s on a single AMD FX-8120, Bulldozer @3.1GHz core.

Grøstl (Gauravaram et al., 2008) was also one of the SHA-3 competition finalists. The authors of the algorithm define it as an iterated hash function with its compression function constructed from two fixed large distinct permutations. The components of the algorithm are based on block cipher AES algorithm, as the S-box used and diffusion layers construction are similar to that of AES leading to strong confusion and diffusion in the algorithm. The effect of well-known generic attacks has been made difficult by its wide pipe construction in which the size of output is significantly smaller than the size of internal state. The authors had claimed to have good performance of Grøstl on varied range of platforms with counter-measures against the side-channel attacks.

The MD6 Message-Digest Algorithm was also one of the contestants of the SHA-3 competition and was designed by Rivest et al. (2008). The algorithm uses a Merkle tree-like structure in order to enable parallel processing while computation of hashes for very long inputs. The authors had claimed on an Intel Core 2 Duo, a performance of 28 cycles/byte for MD6-256 and verifiable resistance against differential cryptanalysis at the time of its submission. But later, it was found that the claims made regarding MD6’s resistance to differential attacks were for the submitted version and not for a faster reduced-
round version. So Rivest posted a comment on July 1, 2009, at NIST that MD6 is not however ready to be a candidate for SHA-3 due to the lack of the proofs on attack resistance. Then after, in September 2011, a paper (Heilman, 2011) was posted on MD6 website supporting MD6 with faster reduced-round versions which are resistant to differential attacks. But unfortunately MD6 was out of the competition by that time.

In IACR Cryptology, (Atighehchi et al., 2010) had proposed a parallel hash algorithm based on Skein hashing which was one of the candidates of SHA-3 competition organized by NIST. Their preliminary work presents the parallel implementation and associated performance evaluation of available Skein algorithm. To parallelize Skein they have used the tree hash mode in which they have created one virtual thread for each node of the tree which provides a generic method for fine grain maximal parallelism. But none of these were able to qualify as SHA-3 due to speed and security reasons.

Keccak is another parallel hashing algorithm accepted as new SHA-3 algorithm (discussed in Section 1.4.5).

1.8 Outline of the Dissertation

This thesis consists of three main parts (excluding appendix A, B, C, D and E). Part I comprises Chapters 1 and 2, which provides general background information on the state of art of cryptographic hash functions and parallel programming concepts. Part II comprises Chapter 3, 4, 7 and 8 where we introduce the EITRH transformation, analyze its security, and check its performance and energy efficiency. Part III comprises Chapters 5 and 6, which provides transformation’s adaptability in various application areas. This section summarizes the contents of all the chapters of the Thesis.

- Chapter 2: The chapter gives an overview of the basic concept of parallel programming along with the tools and techniques used to conduct the experiments for the research work. The server setup done is common for all sorts of experiments and is discussed in detail in the chapter. All the software and hardware related specifications are provided in it.
• Chapter 3: The chapter presents details of our new proposed parallel cryptographic hashing framework based on Tree hashing. It covers design and analysis of two versions of the fast and secure hash function framework, ITRH (Inverted Tree Recursive Hashing) and EITRH (Enveloped Inverted Tree Recursive Hashing). EITRH framework has an enhanced level of security as opposed to ITRH. Both the frameworks are described in detail along with the experimental analysis. Apart from this, the complexity calculation and performance measurement of one of the variants of EITRH is also given.

• Chapter 4: The chapter discusses the security analysis of the new proposed framework so as to qualify it as a CHF. Analysis is done on the basis of essential properties required to be satisfied by a CHF and few more.

• Chapter 5: The chapter covers in detail the digital forensic and application of cryptographic hash functions in digital forensics. It includes the current approaches used for disk imaging in forensics along with its drawbacks. The performance of one of the variants of EITRH along with randomness test is also discussed.

• Chapter 6: The chapter discusses digital signatures as another application of hash functions. Firstly, the process of digital signing has been explained followed by the process of parallel digital signing along with RSA algorithm. Then after the performance measure of the algorithm is done using one of the variant of EITRH along with standard RSA digital signature algorithm with varied key sizes.

• Chapter 7: The chapter covers the performance of newly proposed hash function EITRH on GPU cards. GPU has proved to be a fast, promising parallel computing technology which can make use of thousands of cores to achieve speedup. Along with OpenMP implementation, CUDA implementation of EITRH is analyzed on Nvidia graphics card.

• Chapter 8: The final objective of the research work, which include energy efficiency of the proposed algorithm is discussed in this chapter. A complete
discussion on the energy benefits along with the conversions and result analysis is provided in it. The impact of the performance gained over conventional systems is also given in the chapter.

- Chapter 9: The thesis concludes in this chapter with a quick summary and conclusion of the research work done followed by a brief discussion about several possible extensions to the work presented in this thesis.
CHAPTER 2
BASIC CONCEPTS, TOOLS AND TECHNIQUES
Chapter 2

BASIC CONCEPTS, TOOLS AND TECHNIQUES

Advances in the processors and manufacturing technology as predicted by the Moore’s Law (Moore, 1965) has led to increased performance of modern hardware and is still continuing with the availability of multiple cores executing in parallel. Various computing algorithms have been parallelized with the availability of multiple processing cores and massively parallel GPUs (Alcantara et al., 2009). However, the hashing algorithms are characteristically sequential in nature and are not able to take advantage of parallel processors available these days. The dissertation research represents an effort to parallelize the hashing algorithms and make the optimum utilization of current computers architecture.

The solution to the problem mentioned in Chapter 1, regarding time consumption by the cryptographic hashing algorithms, could be to parallelize them. Therefore, a study of parallel processing concepts is done to develop new parallel hashing algorithms that can use the power of multi-core processors available. This chapter discusses in detail the basic concepts, tools and techniques used for overcoming the problem. Further, Sections 2.1 and 2.2 discusses the parallel processing concepts and techniques necessarily required for parallelization. Performance of hashing algorithms developed in terms of speedup, efficiency, cost optimality and scalability has been measured using the tools and techniques discussed in Sections 2.3 and 2.4 respectively.

2.1 Introduction to Parallel Processing

Parallel processing (Kumar et al., 1994) is a promising technology in which multiple computational resources are used to achieve on-line dynamic security evaluation, real time simulation of power systems, or to solve complex computational problems. For quite a few decades, now parallelism has been used in the domain of High Performance Computing (HPC) for the faster and
more efficient execution of the instructions. PC clusters are attracting more and more attention for parallel processing as clustering allows to solve large problems in less time along with the advantages of low cost, upgradability, scalability and green house computing (energy efficiency).

In parallel computing, large complex problems are split into independent parts so that each processing element can execute its part simultaneously with others and then recombine to form the final outcome. It thereby improves the response time and increases throughput by making optimal use of the available computing resources. The computing resources can be diverse and can include a single computer with multiple processors or multiple networked computers or specialized hardware, or any combination.

Parallelization of a problem involves number of tasks including identifying share of work that can be performed concurrently, decomposition technique, mapping method for load balancing, type of parallel computer required, etc. For a parallel program to be efficient, the problem needs to be decomposed (divide) properly depending upon the type of parallelism required. Different levels of parallelism includes instruction level, data level and task level parallelism which can be figured out in a sequential problem. Further sections of this chapter discusses some of the key concepts and techniques which serve as the basis for parallelization methodology.

2.1.1 Parallel Programming Terms

Tasks

Tasks are units of computation defined by a programmer to divide the main problem into subparts. Parallelism is introduced into a problem to reduce the execution time by simultaneously executing multiple tasks. These tasks can be of varied size depending upon the problem.
Granularity

Granularity is a qualitative measure which specifies the amount of computation involved in a process. It is determined by the number and size of tasks into which a problem is decomposed. Granularity can be fine-grained or coarse-grained. A decomposition where relatively large size of small tasks are assigned is fine-grained granularity whereas the decomposition where relatively small size of large tasks are assigned is coarse-grained granularity. The type of granularity a parallel program contains is determined by its task decomposition techniques.

Scalability

Scalability in a parallel program refers to the ability to exhibit a proportionate increase in parallel speedup corresponding to the addition of more resources. Hardware, algorithm, overhead and characteristics of the specific problem are the key factors that contribute to scalability.

Synchronization

Synchronization is the organization of parallel tasks in real time in order to avoid overhead and incorrect outcomes. A task may have to wait to proceed further until another task(s) reaches a particular point, if a synchronization point has been marked within a program. It can therefore cause an increase in the parallel application's wall clock execution time. Synchronization is one of the most crucial problems in developing shared memory based parallel software. Barrier and Lock/Semaphore are two broad synchronization types which are commonly used.

Barrier commonly infers that all sub tasks are processing to accomplish a larger task where each task executes its work until it reaches the barrier. When all the tasks reach the barrier, they are synchronized.

Lock / semaphore can comprise any number of tasks. Ideally, it is used to protect access to global data or a section of code. Only one task at a time can use the lock/semaphore flag.
2.2 Concepts and Techniques

Parallelization of a problem or an algorithm involves numerous tasks to be completed before the actual implementation. In this process, the first task is to identify the most difficult part, identify the portion of a problem that can be performed concurrently. Once it is recognized then various parallelization techniques are applied which includes dependency detection, decomposition, mapping techniques for load balancing, synchronization etc. In this section, we have briefly discussed these techniques which are required for parallelization of a problem.

2.2.1 Computer System Categorization

In 1966, Michael Flynn gave the classification of computers based on dimensions of instructions and data streams rather than machine architecture. The multiplicity of instructions and data streams classifies the computer organization into four categories. Figure 2.1 illustrates all the classifications where each “PE” is a processing element (CPU).

![Flynn's Taxonomy Diagram](image)

Fig. 2.1 Flynn’s Taxonomy
2.2.1.a Single Instruction and Single Data Stream (SISD)

SISD is a conventional sequential computer in which a single instruction (I) from memory is fetched by a single processing element (PE) and processes a single data stream (D) as input during one clock cycle. No parallelism is exploited either in instruction or data streams.

Examples of SISD architecture includes old mainframes, minicomputers and workstations with uniprocessors.

2.2.1.b Single Instruction and Multiple Data Stream (SIMD)

In SIMD architecture, multiple processing elements execute the same instruction set on different data elements during one clock cycle i.e. a single instruction set is executed for multiple data items on multiple processing elements synchronously (lockstep execution). SIMD based computers are considered to be most natural form of parallel computers.

Examples of SIMD architecture includes Graphics processor units (GPUs), IBM 900, AMD and Intel’s multicore processors.

2.2.1.c Multiple Instruction and Single Data Stream (MISD)

In MISD architecture, each processing element operates on different instructions but on the same data independently during one clock cycle. The architecture of single data streams executing on multiple processors is rarely applied.

Examples of MISD architecture includes real time systems for fault-tolerance, space shuttle flight control systems.

2.2.1.d Multiple Instruction and Multiple Data Stream (MIMD)

MIMD is a parallel computer where each processing element operates on different instructions, on different data streams independently during one clock cycle. The processors work on its own instructions, on its own data. Tasks executed by processors are not lock-stepped and runs asynchronously. In real sense, MIMD systems are recognized as parallel systems.
Examples of MIMD architecture includes current supercomputers, networked parallel computer clusters, grids, multi-processor SMP computers, all multi-core PCs.

2.2.2 Decomposition Techniques

In order to solve a problem in parallel, the fundamental step is to split the computations into set of tasks for concurrent execution known as decomposition of the problem. There is no fixed set of decomposition techniques and no technique can guarantee to be best for a particular problem. Further sub-sections of this section briefly discuss some of the decomposition techniques.

2.2.2.a Recursive Decomposition

In recursive decomposition, a problem is solved by decomposing it into set of independent sub-problems. Thereafter each sub-problem is solved recursively converting into smaller sub-problems by the combination of their results following the divide-and-conquer technique. Concurrency can be naturally achieved as different sub-problems can be solved concurrently.

2.2.2.b Data Decomposition

Data decomposition is commonly preferred in the algorithms where large data sets are involved. It is a two-step process where, firstly the data on which computations are performed is partitioned (data may be input data/ output data/ both input and output or intermediate data, depending upon the possibility) and secondly the partitioned data is assigned to tasks for parallel execution. Usually these tasks are similar for different data partitions.

2.2.2.c Exploratory Decomposition

Problems where computations correspond to a search of a space for solutions requires exploratory decomposition. In this decomposition, the search space is partitioned into smaller parts and thereafter each part is searched concurrently.
to find the desired solution. As soon as the overall solution is found, unfinished tasks are terminated. This decomposition technique is used for a specific class of problems.

2.2.2.d Speculative Decomposition

In speculative decomposition, output of current computation decides the next suitable branch of the programming flow from the available decision branches. It is suitable in programs where, if one task is performing the computation and its output is used in deciding the next computation, then other tasks can concurrently start computations for the next phase.

2.2.2.e Hybrid Decomposition

A computation can be structured into multiple phases and sometimes it becomes essential to apply different decomposition techniques in different phases. Use of multiple decomposition techniques forms a hybrid decomposition. Selection of the techniques totally depends upon program’s computation.

2.2.3 Mapping Technique for Load Balancing

After the decomposition of computations into different tasks, the next step involved is to map the tasks onto processes. The objective is to engage all the cores, so that all the tasks are completed in minimum elapsed time. This process of mapping often takes care of two key source overheads: time spent in inter-process communication and processes left idle. Uneven distribution of load can lead some processes to finish before others or some may be waiting for others to finish because of dependency. Load balancing is the solution to this problem in which, approximately equal amount of work is distributed among all the cores so that all cores or processing units are kept busy with the least inter-process communication cost.

Proper load balancing can only be done by proper mapping of tasks onto processes. On the basis of parallel programming paradigm, characteristics of tasks and their interaction, two broad mapping techniques have been defined: static and dynamic mapping.
2.2.3.a Static Mapping

In this mapping technique, tasks are distributed among processes prior to the execution of an algorithm. It is often used in combination with decomposition technique based on data partitioning due to its static nature. Few of the static mapping techniques include mapping based on *data partitioning*, *task partitioning* and *hierarchical partitioning*.

2.2.3.b Dynamic Mapping

In this mapping technique, tasks are distributed among processes during the execution of an algorithm. It is employed often when tasks size are unknown as static mapping can result in a highly imbalanced distribution of work among processes. The principal reason for its usage is balancing the workload among processes, and hence dynamic mapping is often referred as dynamic load-balancing. Dynamic mapping techniques include *centralized* and *distributed*.

2.2.4 Parallel Computer Memory Architectures

Parallel computers can have shared memory architecture or distributed memory architecture. The type of memory architecture of a computer also classifies the type of parallel computer.

2.2.4.a Shared Memory Architecture

In shared memory architecture, all processors of a parallel computer accesses common memory as a global address space. Multiple cores or processors can work individually but share the same memory resources. Therefore, these type of systems are also known as *tightly coupled systems*. In these systems, changes made by any one processor in a memory location are visible to all other processors. They are usually preferred in systems where high speed real time processing is required. Based on the memory access times, shared memory systems can be further categorized as *UMA* and *NUMA*. 
**Uniform Memory Access (UMA):** In UMA, all processors uniformly accesses the main memory with equivalent bandwidth and latency. Symmetric Multiprocessor (SMP) machines are new representation of UMA these days and are preferred in time sharing applications. Sometimes, they are also known as CC-UMA (Cache Coherent UMA) in which if one processor updates a position in shared memory, it is known to all other processors.

**Non-Uniform Memory Access (NUMA):** NUMA is a combination of two or more SMPs where local memories from global shared memory are connected with every processor and can directly access each other’s memory. The access time is not same for all the processors and if cache coherency is also maintained, then it becomes CC-NUMA (Cache Coherent NUMA).

### 2.2.4.b Distributed Memory Architecture

In this architecture, systems do not share a global memory, as sharing leads to memory conflicts and in return slows down the execution of the system. Each processor has its own private or local memory and therefore changes made by one processor in its local memory does not affect the other’s memory. Hence, these type of systems are also known as *loosely coupled systems*. These systems require a communication network to connect inter-processor memory and whenever there is a requirement to access data of another processor, the programmer has to explicitly define the data communication.

### 2.2.5 Parallel Algorithm Model

Structuring of a parallel algorithm is done by an algorithm model in which selection of decomposition, mapping technique along with an application of appropriate strategy to minimize interactions is done. Defining model, forms the base of parallelization process for any problem or algorithm. These models also exist to perform mapping between hardware and memory architectures. Although, the models are not specific to any particular type of memory or machine architecture, but the choice of programming model for parallel implementation is based on the architecture of the algorithm. Few of the
commonly used parallel programming models include: \textit{Shared Memory Model, Distributed Memory / Message Passing Model, Data Parallel Model, Hybrid Model, Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD)}.

\textbf{2.2.5.a Shared Memory Model}

In a shared memory model, tasks share a common global address space in which they read/write asynchronously. Access to the shared memory is controlled by using different mechanisms like locks or semaphores. Data locality is the biggest problem faced, while using Shared memory model.

\textbf{2.2.5.b Threads Model}

In a threads model, a single heavy weight process is disintegrated into multiple light weight processes, which are called threads. Afterwards, these threads are allowed to execute concurrently in a shared memory area and can communicate with each other through global memory.

\textbf{2.2.5.c Distributed Memory Model}

This kind of model is designed especially for the devices connected through a network. In this model, set of tasks are executed in distributed memory area which can be part of any other device present on the network. The tasks in the model communicate with each other by passing messages between themselves and therefore are known as Message Passing Model.

\textbf{2.2.5.d Data Parallel Model}

In a data parallel model, tasks are mapped onto the processes to perform on similar instruction sets but with different data sets. Set of tasks mutually works on the same data structure/set however each task work on a different portion of same data structure. They can be implemented on both the shared memory systems, where all tasks have access to the data structure through global
memory as well as on the distributed memory systems where the data structure is divided and exist as small portions in the local memory of each task. The main advantage of a data parallel model is the increase in degree of parallelism with the increase in the size of data set.

2.2.5.6 Hybrid Model

A hybrid model is a combination of more than one programming model. For example, threads can be generated in a distributed memory architecture to utilize the memory available with other machines.

2.2.6 Data Dependencies

Dependencies are one of the primary inhibitors to parallelism, so dependence detection plays a vital role in parallel programming. A data dependence results from multiple use of the same location(s) in storage by different tasks and a flow dependence occurs when an input of the instruction depends on the output of a previous instruction. Any type of dependence present in code which hinders the parallelism should be removed carefully and if it is not possible, code is considered to be non-parallelizable.

All the parallelization techniques discussed above have been used to effectively parallelize the hashing algorithm and produce good results. Chapter 3 incorporates the parallelization techniques used in the proposed parallel framework. Next, Section 2.3 discusses in brief the experimental setup done along with various tools used in order to access the outcomes of the research work.

2.3 Tools Used

Assessment of the new proposed parallel framework is done by implementing the algorithms on a test environment on a server with the following hardware configuration:
Table 2.1 Main Characteristics of the Machine Used

<table>
<thead>
<tr>
<th>Processor</th>
<th>RAM</th>
<th>Core Freq. (MHz)</th>
<th>Computing Freq. (MHz)</th>
<th>Memory Freq. (MHz)</th>
<th>Temp (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD FX(tm)-8320 Eight-Core Processor X 8, 3.5 GHz</td>
<td>4GB DDR2</td>
<td>125</td>
<td>3500</td>
<td>1333</td>
<td>+ 44.0</td>
</tr>
</tbody>
</table>

The desktop environment is an AMD FX(tm)-8320 Eight-Core Processor X 8 with support up to 256 KB L1 Cache, 8 MB L2 Cache and 8 MB L3 Cache running at 3.5 GHz.

The software configuration of the server comprises of a dual boot system with a Linux operating system and a Windows operating system. The two different software configurations are:

**Setup I:** For measuring speedup

- **Operating System:** Ubuntu Linux 11.04 x86_64
- **Compiler:** gcc-4.5.4 x86_64
- **API:** OpenMP 3.1
- **Profiler:** gprof

**Setup II:** For measuring energy efficiency

- **Operating System:** Windows 7
- **Framework:** dot net service pack 1
- **Compiler:** GCC v4.8.1 with MinGW64
- **Editor:** Code Blocks
- **Tool:** Joule meter

Setup I has been used to measure the execution time, speedup and efficiency of the proposed algorithm whereas Setup II measures the energy efficiency of the algorithm developed. The need for using Setup II is because of the Microsoft’s Joule meter tool which only runs on a Windows 7 operating system and is very efficient in measuring power consumption.
The experiments conducted in Chapter 3, 4, 5 and 6 are executed on Setup I and the experiments done in Chapter 8 are executed on Setup II. Specification of tools used in both the setups are described in brief:

### 2.3.1 OpenMP API

OpenMP (Dagum and Menon, 1998, Chapman et al., 2008) is a shared-memory API to explicitly direct multi-threaded, shared memory parallelism and is appropriate for implementation on a broad range of SMP architectures. The API is basically a code that can be added to a sequential program written in Fortran, C, or C++ to describe how the tasks need to be shared among different threads that will run on different cores and to order the access to shared data as required. It is directive based and accomplish parallelism exclusively through the use of threads.

OpenMP uses *fork-join model* for parallel programs which means only a single thread (master thread) starts its execution and as soon as it encounters the parallel region it distributes the tasks among the team of other threads (slave threads) depending upon the constructs and data in the region. At the end of a parallel region, all threads terminate after the completion of their respective tasks and only the master thread continues execution until next parallel region is encountered in the program.

The reason for using OpenMP is its strong emphasis on structured parallel programming. It is comparatively simple to use, as the burden of chalking out the details of the parallel program is up to the compiler. Moreover, OpenMP has a major advantage of being widely adopted and therefore its applications can run on many different platforms.

### 2.3.2 Gprof

Gprof (Graham et al., 1982) is a profiler that collects and arranges statistics of programs. It generates a “gmon.out” data file containing all details of the program. The details may contain information such as the number of functions (which are computationally intensive in an algorithm), number of functions
calling other functions and how many times and where the loop dependencies are. It provides many other options to get details about the program as well. This tool has been used for profiling while conducting the experiments.

2.3.3 MinGW

MinGW (Minimalist GNU for Windows) (Peters et al., 2010) is a free and open source application development environment for intensive Microsoft Windows applications. It includes a port of the GNU Compiler Collection (GCC), GNU Binutils for Windows (assembler, linker, archive manager), and miscellaneous utilities. It can execute on either the native MS-Windows platform or can be cross-hosted on GNU/Linux. Most languages supported by GCC like C, C++, Fortran, etc. are also supported by the MinGW port. The GCC runtime libraries are such as lib stdc++ for C++, libg fortran for Fortran, etc. are used in it.

2.3.4 Code Blocks

Code::Blocks is a free C, C++ and Fortran IDE created for compiling and running multiple programming languages according to the current needs. It is designed to be extensible and fully configurable. The IDE is used to implement the proposed algorithm on a Windows operating system.

2.3.5 Joulemeter

Joulemeter (Goraczko et al., 2011) is a tool provided by Microsoft to view power consumption by a system or an application. This tool has been used to check the energy efficiency of the algorithm with that of existing standard algorithms. The complete specification and usage techniques of the tool has been discussed in detail in Chapter 8.

2.4 Assessment Metrics

In order to access the proposed research work, a few assessment metrics were laid down under which the authenticity, validity and effectiveness of research work was judged. Further sub-sections describe these assessment metrics applied.
2.4.1 Speedup

The speedup (Kumar et al., 1994) is measured to analyze the performance gain by parallelizing a problem over a sequential implementation. It is the ratio of time taken by a single processing element to solve a problem, to the time required to solve the same problem on a parallel system with $p$ identical processing cores.

Speedup can be defined as,

$$ S = \frac{T_s}{T_p} $$

where, $T_s$ is the wall clock time taken by a sequential algorithm and $T_p$ is the wall clock time taken by a parallel algorithm of the same problem.

The speedup has been used as one of the metrics to access the proposed framework.

2.4.2 Efficiency

Efficiency (Kumar et al., 1994) is a measure of portion of time for which the processing element is usefully engaged. It is defined as the ratio of speedup pertaining to the number of processing elements. Generally, efficiency is found to be between zero and one depending upon the efficacy with which the processing elements are utilized.

Efficiency can be defined as,

$$ E = \frac{S}{p} $$

2.4.3 Security Factors

Hashing algorithms without security measures are of no use. Authenticity and integrity by hashing algorithms can only be ensured if they are secure and fulfil the minimum security requirements to be a CHF. Therefore, a check on security factors has also been taken as an assessment factor for the proposed framework. Chapter 4 discusses about these factors in detail.
2.4.4 Energy Efficiency

The main objective for the development of multi-core processors was to have performance scalability along with energy scalability. High energy consumption is another major issue affecting modern computing systems. Energy efficient hashing algorithms can prove to be a new benchmark for use in high-end servers and data centers where cooling cost is very high and in mobile and embedded systems where battery life is limited. Chapter 8 discusses this in detail.

Conclusion

In this introductory chapter we presented the basic concepts and principles of parallel programming. The focus of this chapter is on parallel programming concepts and techniques. Computer system categorization, decomposition techniques, mapping techniques, parallel memory architectures, parallel algorithm models and data dependencies are explained in detail. The chapter also includes system hardware and software configuration specifications on which test experiments are executed. Finally, assessment metrics based on which assessment of the new framework has been done is discussed. This chapter will prove useful in applying various parallel programming concepts in later chapters.
CHAPTER 3
PARALLEL CRYPTOGRAPHIC HASHING TRANSFORMATION
Chapter 3

PARALLEL CRYPTOGRAPHIC HASHING TRANSFORMATION

As discussed in Chapter 1, almost all of the prevailing hashing algorithms are based on Merkle–Damgård construction which is iterative and serial in nature. This feature of the algorithms make digest generation a time consuming process for large files (file size in terabytes) and reduces its performance. However, if the process is parallelized and the machine is operated at low power states, then energy-efficient speedup can be achieved.

In this chapter, we propose a novel parallel hash function transformation based on recursive tree hashing. Hereafter, we will refer to the proposed hash function transformation as ITRH and EITRH. The transformation consists of three main sub-parts: message expansion, parallel reduction and hash value generation. The algorithms based on the transformation can be implemented in parallel mode to provide a fast hashing scheme. We further propose several EITRH class based algorithms and proved them to be pre-image, second pre-image and collision resistant. The comparisons between ITRH and EITRH based algorithms and other standard hash functions are also presented to check the performance gained.

3.1 Introduction

In the last decade, various CHFs have been proposed based on different transformations. However, some of them were proved to be insecure or of slow speed (refer Chapter 1 for details). But the main motive for developing new hash functions was to construct a secure, fast and efficient parallel hash function. These days computer systems come with multi-core processors, therefore designing a parallel hash function will help to exploit these processors more efficiently and effectively.
Apart from this, CHFs were designed to be keyless, where a variable-length message is accepted and a fixed-length digest is returned. However, significant weaknesses were reported in some of the popular keyless hash functions which motivated the researchers to start working on keyed hash functions, for more rigorous security influences. It turns out that converting an existing keyless hash function into a keyed one is non-trivial. After the extensive study of serial and parallel hash constructions in Chapter 1, it was observed that the tree based hashing could prove to be more promising than others. Therefore, in this chapter we propose a novel parallel hash function transformation based on recursive tree hashing that can potentially solve both the problems.

The rest of the chapter is organized as follows: Section 3.2 describes details of the proposed recursive tree based hash function followed by its framework and analysis in its subsequent sub-sections. Section 3.3 brings an enhanced secure version of the proposed framework along with a discussion on its description and design issues in Sections 3.3.1 and 3.3.2, its variants in Sections 3.3.3 followed by performance benchmarks and experimental results in Sections 3.3.4 and 3.3.5 respectively.

### 3.2 Inverted Tree Recursive Hashing (ITRH) Transformation

The new proposed framework is based on Inverted Tree Recursive Hashing (ITRH) with the same capabilities of compression and ease of computation as that of standard CHF. But in contrast to the standard Merkle–Damgård (MD) CHF model, the proposed framework removes the sequential dependencies of one block’s output to another (refer Eq. (1.1) and Eq. (1.2)) and makes it viable for parallel computation of the digest. This in return improves the run time efficiency of the algorithm irrespective of its computational complexity.

The general framework and design of the parallel hash function are illustrated in Fig. 3.1 and 3.2, respectively. The following sub-sections of this section explains the proposed hash function in detail.
3.2.1 ITRH Framework

The construction of ITRH transformation is based on recursive tree hashing as compared to the already existing tree hashing constructions (Rivest et al., 2008, Andreeva et al., 2012). It is basically designed to break the input data dependencies of one block of data to that of another. The main design of the framework is to divide the input data into blocks of equal size and to apply compression function on each block of data individually. In the next phase, the hash generated by each block is further combined to form a new set of input data. Thereafter, it is recursively rehashed until a final digest of size n bits (generic) is generated.

As shown in Fig. 3.1, the framework includes different levels where level 0 represents the main input message in equally sized blocks and level n represents the final digest generated for the message. The number of intermediate levels 1, 2, ..., n-1 created are directly proportional to the message size. All these levels are comprised of intermediate digests generated by each block. In order to strengthen the transformation (Merkle-Damg˚ard strengthening), length of the complete data at that level is appended in the last block of each level. This type of transformation makes parallel computation of the digest feasible while preserving its security and efficacy.

Figure 3.2 shows the three step foundation of a hash generation process of ITRH and Algorithm 3.1 explains the compression function and domain extender of the proposed framework. In ITRH, message $M$, where $M \in \{0, 1\}^*$ is first divided into $k$ blocks of size $l$ along with padding if required. In the transformation, the bit padding mode used is defined in RFC1321 and is employed in many cryptographic algorithms. Thereafter, each block is executed in parallel on multi-core processors with the same $IV (H_0)$ recursively until a hash code of n-bit is achieved.
Fig. 3.1 ITRH Framework

Fig. 3.2 Three Step ITRH Framework
The next Section 3.2.2 explains the algorithm based on the ITRH framework in detail along with experimental results.

### Algorithm 3.1 ITRH Algorithm

**Procedure:** Domain Extender  
**Model:** Data Parallel Model with P processors  
**Input:** Message M, n, IV  
**Output:** Hash Code  
**Given:** Compression function  
\[ f : \{0, 1\}^{l+n} \times \{0, 1\}^n \rightarrow \{0, 1\}^n; \]  
\[ n \text{-bit constant IV}. \]

**Declare:** Number_of_Cores, Block_Size, K  
Set Number_Of_Cores = P  
Break M into l-bit blocks \( M_1, \ldots, M_k \), padding if necessary;  
Let \( M_{k+1} \) be encoding of \( |M| \);  
Declare No_Of_Iterations := K / Block_Size  
Set \( H_0 := IV \);  
**Parbegin**  
For ALL \( i : [1, k + 1] \) in ASYNC  
Set \( H_i := f(H_0, M_i) \);  
EndFor  
Append \( H_i \) (1 < \( i \) < \( k + 1 \)) to form \( H \) such that  
\[ H := H_1||H_2|| \ldots ||H_{k+1} \]

**Paren**  
if \( |H| > n \)  
Domain Extender (message \( H, n \))  
**return** \( H \)

### 3.2.2 RSHA-1 Algorithm- Variant of ITRH

Based on ITRH framework, RSHA-1 algorithm (Kishore and Kapoor, 2014a) was developed which had mixed properties of SHA-1 algorithm and ITRH. Its detailed description is provided in Section 3.2.2.a.

**STEP 1-2: PADDING:** The message \( M \) is partitioned into multiple blocks of size 512 bits, \( M_i \) \( (0 < i <= |M| \mod 512) \) and is padded with a single one
followed by a necessary number of zeroes for the final block to have 448 bits. Even if the message is of the desired length, padding is still done.

STEP 3: LENGTH STRENGTHENING: The length of the original message is calculated and appended at the end of the message as a block of 64 bits to complete the last block as 512 bits.

STEP 4: Five 32-bit register variables $a, b, c, d,$ and $e$, represented as $H0$ are initialized to some specific constants. Further, this $H0$ becomes Initial Vector ($IV$) for each block $M(i)$ to cut out the dependences among hash values ($H_i$).

After that, for each 512-bit message block the compression function is executed simultaneously on multiple threads on a computer with multi-core processors in a thread pool model.

STEP 5: COMPRESSION FUNCTION: The compression function $f()$ executes in parallel for each block with message block $M_0$ and $H_0$ as inputs.

In the compression function ($f()$), the message block is firstly transformed from sixteen 32-bit words $M_0$ to $M_{15}$ to eighty 32-bit words ($W_0$ to $W_{79}$) by using the following algorithm:

$$W[i] = M_i,$$  \hspace{1cm} \text{for } i = 0 \text{ to } 15

$$W[i] = (W[i] - 3 \oplus W[i] - 8 \oplus W[i] - 14 \oplus W[i] - 16) \ll << 1,$$  \hspace{1cm} \text{for } i = 16 \text{ to } 79

The main loop has four rounds of 20 operations each. In each operation, a nonlinear function on three of $a, b, c, d,$ and $e,$ is performed and later shifting and adding is done.

Calculate $f()$ and the four constants $k$ (based on round number) for $i = 0 \text{ to } 79$.

$$e = d$$

$$d = c$$
\[ c = b \ll 30 \]

\[ b = a \]

\[ a = a \ll 5 + f() + e + k + W[i] \]

Add computed \( a, b, c, d \) and \( e \) to the original five 32 bit variables respectively to form the hash code of that block.

STEP 6: Hash values obtained from each block are concatenated according to the block number and stored using a pointer variable. These values become the new message \( M_i \) for the compression function and the whole process is executed recursively to produce an output hash value of 160-bits (as illustrated in Fig. 3.2).

In RSHA-1 each message block has the same initial vector \( H_0 \) to generate hash value. So the calculation of \( H(M(i)) \) for each \( M(i) \) is fully parallelized without dependences, and then reducing \( H_i \) using \( H(H_i) \) as similar to \( H(M(i)) \) to produce an output hash value. With the parallelization of the algorithm an extra overhead of dynamic memory requirement has been increased for storing concatenated hash values as compared to the traditional SHA-1 which requires only 160 bits for storing intermediate hashes. Therefore, de-allocation of memory is consistently required to overcome the overhead.

### 3.2.3 Experimental Results

In order to access the framework, tests were performed to check the performance gained over the existing sequential one. There are different ways to measure the time taken by an algorithm in the Linux operating system, one of which is to use the Linux time command which shows real, user and sys time. As our transformation improves the efficacy of complete hash generation process, so gettimeofday() function of Linux is used to measure the execution time of the algorithm.

Setup I (as discussed in Chapter 2, Section 2.3) was chosen as a test environment to conduct the experiments. The execution time taken by standard
SHA-1 algorithm and RSHA-1 was measured on files taken from the t5 corpus (Roussev, 2011) which is a sample of different types of files derived from the GovDocs corpus. The tests were executed on text files ranging from 2 MB to 1GB in size number of times. The average results obtained by invoking 8-cores of the machine are listed in Table 3.1.

Table 3.1 Execution Time by RSHA-1 (ITRH) and SHA-1 on AMD FX Bulldozer Machine

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>SHA-1</th>
<th>RSHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2MB</td>
<td>0.0285</td>
<td>0.01268</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4MB</td>
<td>0.05475</td>
<td>0.02314</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8MB</td>
<td>0.10744</td>
<td>0.03441</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16MB</td>
<td>0.21619</td>
<td>0.06658</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>32MB</td>
<td>0.42478</td>
<td>0.13172</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>64MB</td>
<td>0.84093</td>
<td>0.26237</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>128MB</td>
<td>1.70663</td>
<td>0.83703</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>256MB</td>
<td>3.38537</td>
<td>1.05623</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>512MB</td>
<td>7.10087</td>
<td>2.69129</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1GB</td>
<td>13.64857</td>
<td>4.21023</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.3 Execution Time by SHA-1 and RSHA-1 (ITRH) on Small Files
Figures 3.3 and 3.4 depicts the speedup gained in terms of reduced execution time for small as well as large files respectively. Both the images show a consistent gain in speedup as the file size increases. An improvement of approximately 3.5 folds is witnessed when compared with sequential SHA-1 algorithm.

An observation from the results can be made that the new framework can give good results for producing hash code for heavy files as well. Further to discuss the influence of different number of cores, our test also includes multiple runs of different size text files on different number of cores of the machine. Table 3.2 shows the results of these tests conducted on dual core, quad core, hexa core and octa core processors.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec) by RSHA-1 on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2 Core</td>
</tr>
<tr>
<td>1</td>
<td>32MB</td>
<td>0.46171</td>
</tr>
<tr>
<td>2</td>
<td>64MB</td>
<td>0.97682</td>
</tr>
<tr>
<td>3</td>
<td>128MB</td>
<td>1.9325</td>
</tr>
<tr>
<td>4</td>
<td>256MB</td>
<td>3.94609</td>
</tr>
<tr>
<td>5</td>
<td>512MB</td>
<td>7.76</td>
</tr>
<tr>
<td>6</td>
<td>1GB</td>
<td>15.63747</td>
</tr>
</tbody>
</table>
Fig. 3.5 Execution Time on Multiple Cores by ITRH

Fig. 3.6 Speedup of RSHA-1 with Increasing Number of Cores
Figure 3.5 demonstrates the performance of RSHA-1 on different number of cores and different file sizes. It can be deduced from the results that execution time of the algorithm decreases with the increase in number of processing elements. Increase in the speedup with the increase in processing units can be easily seen in Fig. 3.6. As the number of cores increases for a fixed file size, the framework shows a linear speedup.

3.2.4 Discussions

In Section 3.2.3, experimental results showed a speedup of approximately 3.5X when using the proposed framework for generating the hash, but only speedup is not enough. A complete security analysis of the framework is required. On the basis of essential CHF properties, security analysis was done using a computer simulator. The details of the security analysis for the new transformation is given in Chapter 4. During the analysis, it was found that the proposed transformation can be prone to various attacks. By removing the chain dependencies in SHA-1 algorithm, the security of SHA-1 has also been hindered. Therefore, an extra measure of security is required to make the proposed framework attack resistant. So in order to overcome the attacks, an enhanced version of the ITRH framework was proposed which is discussed in Section 3.3 in detail.

3.3 Enveloped Inverted Tree Recursive Hashing (EITRH)

Transformation

After the security analysis of ITRH transformation, Enveloped Inverted Tree Recursive Hashing (EITRH) transformation was proposed with an enhanced security level. The design of the model is based on ITRH only with an inclusion of property of Enveloped Merkle-Damg˚ard (EMD) construction (discussed in Chapter 1).
The sub-sections of this section describe the transformation in detail. Section 3.3.1 presents a general idea of EITRH framework along with its parallelization issues in Section 3.3.2. In Section 3.3.3, we have discussed EITRH variants followed by its performance benchmarks, experimental results and comparative analysis in Sections 3.3.4 and 3.3.5 respectively.

### 3.3.1 Framework Description

EITRH has an inverted tree structure which is composed of three recursive steps: message expansion, parallel reduction and hash value generation. The base value of the recursive procedure is the original message $M$’s length and the terminating point is the size $h_n$ of final hash value, where $h_n$ is dependent on the variant of EITRH used. The structure of the proposed transformation is described in Algorithm 3.2 and depicted in Fig. 3.7.

![Fig. 3.7 EITRH Framework](image-url)
**Procedure**: Domain Extender

**Model**: Data Parallel Model with $P$ processors

**Input**: Message $M, n, IV_1, IV_2$

**Output**: Hash Code

**Given**: Compression function
- $f: \{0, 1\}^{l+n} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$;
- $n$-bit constant $IV$.

**Declare**: Number_of_Cores, Block_Size, $K$
- Set Number_Of_Cores = $P$
- Break $M$ into $l$-bit blocks $M_1, \ldots, M_k$, padding if necessary;
- Let $M_{k+1}$ be encoding of $|M|$;
- Declare No_Of_Iterations := $K / \text{Block$_\text{Size}$}$
- Set $H_0 := IV_1$, $H_{00} := IV_2$

**Parbegin**

For ALL $i: [1, k + 1]$ in ASYNC

- If ($i == k + 1$)
  - Set $H_i := f(H_{00}, M_i)$
- else
  - Set $H_i := f(H_0, M_i)$

endif

**EndFor**

Append $H_i$ ($1 < i < k + 1$) to form $H$ such that

$H := H_1||H_2|| \ldots ||H_{k+1}$

**Parend**

if $|H| > n$

- Domain Extender (message $H, n$)

return $H$

Algorithm 3.2 EITRH Algorithm

**Step1: Message Expansion**

In order to improve sensitivity in the hash code generation process, message expansion is done at each level of the tree. It is a significant step, in which the message $M_i$ at level $h_i$ where $i >= 0$ and $i < h_t$ (height of the tree) is divided into $k$ blocks of size $l$ ($l$, determined by variant of EITRH) where $k = (|M| mod l) + 1$. The last block ($k^{th}$ block) is padded with the length of
corresponding message $M_i$ along with 1 followed by 0’s such that the length of block becomes the multiple of $l$.

$$M_i = M_i \parallel 10 \ldots 0 \parallel |M_i|,$$  
for each level $h_i$.

Thereafter, each block is executed in parallel on multi-core processor with two different initial vectors: $IV_1(H_0)$ of size $h_n$ for $k - 1$ blocks and $IV_2(H_{00})$ of size $h_n$ for the $k^{th}$ block.

**Step2: Parallel Reduction**

The parallel reduction procedure is same for each block of $M_i$, as it has been made independent of its preceding blocks. For each block the reduction is done in parallel, using the same compression function ($f()$) and same $IV_1(H_0)$ for $k - 1$ blocks and $IV_2(H_{00})$ for the $k^{th}$ block. Once all the blocks are done with the calculation of hash values (acting as internal state hashes), parallel reduction is done. The corresponding hash values generated for each block of $M_i$ are concatenated to form $H_i$ and becomes $M$ for next level of tree. The steps 1 and 2 are repeated recursively until a collective hash code of $h_n$ bits is achieved as $H_i$.

$$h(i) = \begin{cases} 
  h(M_i, H_0), & 0 < i \leq k - 1 \\
  h(M_i, H_{00}), & i = k
\end{cases}$$

The data dependency in the algorithm has been removed by passing $IV_1(H_0)$ to each block separately. Moreover, the level of security has also been improved by introducing $IV_2(H_{00})$ for the last block to avoid multi-collision attacks.

**Step3: Hash Generation**

After all the blocks of level $h_{t-1}$ have been processed, the final hash value is generated by $h = H(M_{h_{t-1}}, H_{00})$.

The steps 1 and 2 are recursive in nature and step 3 generates the final hash code. The $h_t$ of the tree totally depends on the file size and EITRH variant used. Formally, EITRH framework can be defined as:
**Definition 3.1:** Let a set of bit strings be denoted as \( \{0, 1\}^* \), a message \( M \in \{0, 1\}^* \) and \( k \) be the number of blocks of size \( l \), then a *keyless Enveloped Inverted Tree Recursive hash function* accepts a message \( M \in \mathcal{M} \) and generates \( H: \{0,1\}^* \to \{0,1\}^n \), a unique value or fingerprint where \( n \) is hash size,

\[
h(i) = \begin{cases} h(M_i, H_0), & 0 < i \leq k - 1 \\ h(M_i, H_{00}), & i = k \end{cases}
\]

and \( H_i = h_1 \parallel h_2 \ldots \parallel h_{k-1} \parallel h_k, M \to H_i \) until \( H_i \to \{0, 1\}^n \).

EITRH transformation can work as both keyless and keyed hash function, if \( H_{00} \) in definition 3.1 is replaced with a fixed length key \( \mathcal{K} \) as defined in definition 3.2. In this approach, a keyless hash function is perfectly transformed into a keyed variant by introducing an extra component \( \mathcal{K} \) in place of \( H_{00} \) accompanying the compression function to handle the key separately.

**Definition 3.2:** Let a set of bit strings be denoted as \( \{0, 1\}^* \), a message \( M \in \{0, 1\}^* \) and \( k \) be the number of blocks of size \( l \), then a *keyed Enveloped Inverted Tree Recursive hash function* accepts a message \( M \in \mathcal{M} \), fixed length key \( \mathcal{K} \in \mathcal{K} \) and generates \( H_k : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^n \), a unique value or fingerprint where \( n \) is hash size,

\[
h(i) = \begin{cases} h(M_i, H_0), & 0 < i \leq k - 1 \\ h(M_i, \mathcal{K}), & i = k \end{cases}
\]

and \( H_i = h_1 \parallel h_2 \ldots \parallel h_{k-1} \parallel h_k, M \to H_i \) until \( H_i \to \{0, 1\}^n \).

The difference in the framework of EITRH and Inverted Tree Recursive Hashing (ITRH) given in Section 3.2 is the usage of an extra initial vector \( IV_2 \) required for the last block of each level of the tree. This additional \( IV \) makes the transformation keyed as well as adding on to the security of the construction by making the construction multi-collision resistant, partial pre-image resistant and non-correlative along with other CHF essential properties. The details of these additional security measures are provided in Chapter 4.
3.3.2 Parallelization Techniques

In order to parallelize the proposed framework, various parallel programming techniques as described in Chapter 2 are used and are discussed below –

The first step involved in the parallelization of an algorithm is decomposition of the data or instructions. This decomposition depends upon the parallel model chosen. In the proposed framework, MIMD (Multiple Instruction Multiple Data) model is chosen in which both data and recursive decomposition is done and divide-and-conquer technique is applied.

The number and size of the tasks according to which the problem is decomposed in the research work makes it fine-coarse grained decomposition in which a high number of large sized tasks are performed. The task is divided into large number of chunks with each chunk performing its own computation. The maximum degree of concurrency will depend upon the machine used along with the support of number of threads executing in parallel.

The next step was to map the tasks onto processes with the idea of taking care of two key source overheads: time spent in inter-process communication and processes left idle. In order to alleviate the load-misbalancing and process idling problems, Static Mapping has been done with block-cyclic distribution. The central idea behind such type of distribution was to partition the input set into many more blocks than the number of available processes and process in a round robin fashion, as there is less locality of interaction involved in the algorithm worked upon.

The transformation is based on the Data-parallel model in which the tasks are semi-statically mapped onto the processes. Each task allocated performs similar operations on different data sets. The algorithm is implemented in shared-address-space by making the optimum use of shared and global memory available. This model increases the degree of parallelism as the size of the input increases making it feasible to implement the code effectively for large data sets.
The proposed approach is scalable in nature i.e. able to adapt itself to the increasing number of processing resources and acts dynamically. The proofs for its scalability are provided in Sections 3.3.4 and 3.3.5.

3.3.3 EITRH Variants

EITRH transformation can support hash functions with five different internal state sizes: 160, 224, 256, 384 and 512 bits. The variants proposed are RSHA-1, RSHA-224, RSHA-256, RSHA-384, and RSHA-512. These EITRH hash functions are based on standard SHA family variants with enhanced security and support for parallelization.

Experiments in all the chapters of this thesis have been conducted on the RSHA-1 variant of EITRH. Other variants can be used in substitution for the respective standard SHA family variants as shown in Table 3.3. Description of RSHA-1 algorithm based on ITRH transformation is provided in Section 3.2.2, an addition of $IV_2$ is done for the last block of each iteration to be EITRH. The other versions are quite similar to it.

Table 3.3 EITRH Variants

<table>
<thead>
<tr>
<th>Replace</th>
<th>With</th>
<th>Hash Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>RSHA-1</td>
<td>160</td>
</tr>
<tr>
<td>SHA-224</td>
<td>RSHA-224</td>
<td>224</td>
</tr>
<tr>
<td>SHA-256</td>
<td>RSHA-256</td>
<td>256</td>
</tr>
<tr>
<td>SHA-384</td>
<td>RSHA-384</td>
<td>384</td>
</tr>
<tr>
<td>SHA-512</td>
<td>RSHA-512</td>
<td>512</td>
</tr>
</tbody>
</table>

3.3.4 Performance Benchmarks

In this section, we will discuss about the performance measures of the parallel program developed (Kishore and Kapoor, 2014a) using some analytical tools. The scalability of a parallel program can be evaluated on the basis of a few metrics including complexity, speedup, efficiency, cost optimality and iso-efficiency. The sub-sections of this section discusses about all these metrics for an EITRH class hashing algorithm in detail.
3.3.4.a Algorithm Complexity

Before calculating the speedup and efficiency gains of the proposed framework it is necessary to measure the complexity of an algorithm. Different amount of time results can be found for an algorithm on the same inputs depending on factors like processor speed, instruction set, memory bandwidth, etc. Therefore, estimating efficiency of an algorithm asymptotically is another way without depending upon the implementation details. These theoretical estimates can provide an insight into reasonable directions in the search for an efficient algorithm.

So, in order to calculate the complexity of the proposed transformation we have made a few assumptions.

Let the complexity of generating hash value $H(M(i))$ for each message blocks is $\text{Cmplx}(H(M(i)))$, and the number of original blocks be $b$, than the computational complexity for standard CHF model can be given by:

$$b \cdot \text{Cmplx}(H(M(i)))$$  \hfill (3.1)

Therefore, the first phase of EITRH on $p$ processors, where $p \geq 1$ has the complexity,

$$\frac{b}{p} \cdot \text{Cmplx}(H(M(i)))$$  \hfill (3.2)

For the second phase of EITRH that is re-hash, number of times $H(M(i))$ will be called depends upon the original number of message blocks. After the first reduction, length of the message left will be $x/y$ of the length of the last message where, $x$ is the size of hash to be generated and $y$ is the block size. Both the sizes $x, y$ depend upon the variant of EITRH chosen. So, in order to achieve the hash value of $x$ bit, $\lceil \log_{y/x} b \rceil$ recursive reductions will be performed and the number of times $H(M(i))$ will be called can be given as:

$$n_{call} = \left\lfloor \frac{x}{y} \cdot \frac{b}{p} \right\rfloor + \left\lfloor \frac{x}{y}^2 \cdot \frac{b}{p} \right\rfloor + \cdots + \left\lfloor \frac{x}{y}^z \cdot \frac{b}{p} \right\rfloor$$
\[ n_{\text{call}} = \sum_{i=1}^{z} \left\lfloor \left( \frac{x^i}{y} \right) \cdot \frac{b}{p} \right\rfloor \]

where, \( z = \left\lfloor \log_y x \right\rfloor \) and \( \frac{x}{y} \equiv \text{hashsize} / \text{blocksize} \) (3.3)

Now complexity for the whole reduction process can be given by:

\[ O(rdc) = n_{\text{call}} \cdot \text{Cmplx} \left( H(M(i)) \right) \] (3.4)

From Eq. (3.2) and Eq. (3.4), the overall computational complexity \( f(rdc) \) of EITRH transformation, is given as:

\[ f(rdc) = \frac{b}{p} \cdot \text{Cmplx}(H(M(i))) + n_{\text{call}} \cdot \text{Cmplx} \left( H(M(i)) \right) \]

\[ = \left( \frac{b}{p} + n_{\text{call}} \right) \cdot \text{Cmplx} \left( H(M(i)) \right) \] (3.5)

Therefore from Eq. (3.1) and Eq. (3.5), although the EITRH transformation has more computations than that of standard Merkle-Damg˚ard CHF model but they have the same computational complexity.

### 3.3.4.b Speedup

As mentioned in Section 3.3.4.a, the EITRH framework has more computations than that of the standard CHF model, but the use of multi-core processors can overcome this burden with performance gains. In order to measure the performance gained by the new framework based algorithm over standard CHF, speedup needs to be calculated. It can give the quantitative measure of performance gain by capturing the relative benefit of running a program in parallel. This section provides theoretical calculation of the speedup gained and the related practical proofs are made in the corresponding clamming chapters.

To measure the speedup, it is assumed that the time taken by standard CHF model to calculate hash value of each message block = \( T(M(i)) \)

(We will measure time \( T(M(i)) \) as the number of elementary steps (for each block), provided each such step takes constant time.)
and hence for $b$ message blocks, time $= b \cdot T(M(i))$

Therefore, the sequential time can be given as:

$$T_S = b \cdot T(M(i))$$

$$T_S = \Theta(b) \quad (3.6)$$

In EITRH, each level in Fig. 3.1 is comprised of generating the hash of sub blocks, concatenating hashes, and recursively generating its hash. Therefore, the time complexity of EITRH on $p$ processors, where $p \geq 1$ can be calculated as:

Time taken to generate hash of $i^{th}$ block $= T(M(i))$

Time taken for concatenation (constant) $= t_c$

Number of recursive calls to sub-blocks (from Eq.(3.3)):

$$n_{call} = \sum_{i=1}^{z} \left\lceil \left( \frac{x}{y} \right)^i \cdot \frac{b}{p} \right\rceil$$

where, $z = \lceil \log_{y/x} b \rceil$ and $\frac{x}{y} \equiv \frac{\text{hashsize}}{\text{blocksize}}$

Time taken for reduction and recursive calls $= n_{call} \cdot T(M(i))$

So, parallel time can be given as:

$$T_p = \frac{b}{p} \cdot T(M(i)) + n_{call} \cdot T(M(i)) + t_c$$

$$= T(M(i)) \left( \frac{b}{p} + n_{call} \right)$$

$$= \Theta \left( \frac{b}{p} + n_{call} \right) \quad (3.7)$$

Based on above results, total parallel overhead can be computed as:

$$T_o = p \cdot T_p - T_s \quad (3.8)$$
\[ p \cdot T(M(i)) \left( \frac{b}{p} + n_{call} \right) - b \cdot T(M(i)) = p \cdot n_{call} \cdot T(M(i)) = \Theta(p \cdot n_{call}) \] (3.9)

Now from Eq. (3.6) and (3.7), Speedup can be given as:

\[ S(b) = \frac{T_s}{T_p} \]

\[ = \frac{b}{\left( \frac{b}{p} + n_{call} \right)} \]

\[ = \frac{b}{b \left( \frac{1}{p} + \frac{n_{call}}{b} \right)} \]

\[ = \frac{1}{\left( \frac{1}{p} + \frac{n_{call}}{b} \right)} \]

\[ = \Theta \left( \frac{p}{1 + \frac{p}{b} n_{call}} \right) \] (3.10)

Fig. 3.8 Speedup versus File Size with Fixed Number of Processors
The speedup $S(b)$ (from Eq. (3.10)) is a function of both problem size $b$ and the number of processing cores $p$, so $S(b)$ grows sub-linearly with respect to $b$. Therefore, it can be observed that speedup tends to remain approximately constant if the problem size is increased and the number of processing cores are kept constant, as shown in Fig. 3.8. But the speedup increases, if the problem size and number of processing cores are increased proportionally (Fig. 3.9).

3.3.4.c Efficiency

Efficiency gives the measure of fraction of time for which the processor is usefully employed and is defined in $1/b \leq E(b) \leq 1$. Efficiency of the EITRH variant on $p$ processors can be calculated as:

$$E(b) = \frac{\text{Speedup}}{\text{no. of processors}}$$

$$= \frac{T_s}{p.T_p}$$

This expression can be further rewritten in the form of parallel overhead using Eq. (3.8) and (3.9):

$$E(b) = \frac{1}{1 + \frac{T_o}{T_s}}$$

$$\therefore E(b) = \Theta \left(\frac{1}{1 + \frac{p.n_{\text{call}}}{b}}\right) \quad (3.11)$$

Generally, $b$ is much greater than $p$ in CHF, so $(p.n_{\text{call}})/b > 1$ and efficiency cannot be expected to be one, due to the parallel overhead involved. Therefore, as the size of data and number of processors increases, the performance shown by the algorithm also shows better results than that of standard CHF. The total overhead function $T_o$ (from Eq. (3.8)) is a function of both problem size $b$ and the number of processing elements $p$, so $T_o$ grows sub-linearly with respect to $b$. Therefore, it can be observed that efficiency increases if the problem size is increased by keeping the number of processing cores constant. Further Eqs. (3.10) and (3.11) can be used to calculate the speedup and efficiency for any pair of $b$ and $p$. Figure 3.9 shows the speedup versus number of processing
elements for a few different values of $b$ and $p$ with the corresponding dropping efficiencies in Fig. 3.10. As a consequence of Amdahl’s law, the speedup tends to saturate and efficiency drops with the increasing number of processors in the results.
3.3.4.1 Cost Optimality

The cost of solving a problem can be referred to as the sum of execution time each core spends for solving a problem. It is a product of parallel execution time and the number of cores or processing elements employed for that problem. If a single core is used for solving a problem then its cost is calculated as execution time of the fastest known serial algorithm. For a cost optimal parallel system, the cost of solving a problem on a parallel system has the same asymptotic growth as a function of the input size as the fastest known sequential algorithm on a single processing core (Kumar et al., 1994).

Cost of finding hash value using EITRH framework is:

\[ T_{cost} = p \cdot T_p \]

\[ = p \cdot \left( \frac{b}{p} + n_{call} \right) \]

\[ = \Theta(b + p \cdot n_{call}) \]  

(3.11)

From Eq. (3.11), as long as \( b = \Omega(p \cdot n_{call}) \) the cost is \( \Theta(b) \), which is same as serial cost as given in Eq. (3.6). Therefore the parallel system can be considered as cost optimal when \( b = \Omega(p \cdot n_{call}) \).

3.3.4.e Minimum Execution Time

One’s interest can also be to find the minimum possible execution time for a parallel program, apart from the speedup it can provide. As the number of processing elements for a given problem size increases, either the parallel run time continues to decrease and asymptotically reaches a minimum value, or starts rising after achieving a minimum value. The minimum parallel execution time \( T_{p_{min}} \) for a given problem size can be determined by:

\[ \frac{d}{dp} T_p = 0 \]

\[ \frac{d}{dp} T_p = \frac{d}{dp} \left( \frac{b}{p} + n_{call} \right) \]

\[ = \frac{b}{p^2} (-c) \]
where \( c \) is constant

\[
\therefore b = p^2 \Rightarrow p = \sqrt{b}
\]

Substituting \( p = \sqrt{b} \) in Eq. (3.7), we get

\[
T_p^{\text{min}} = 2\sqrt{b} \tag{3.12}
\]

Equation 3.12 shows that, \( 2\sqrt{b} \) is the minimum parallel execution time for EITRH based algorithms, with \( \sqrt{b} \) as the minimum number of processing elements required.

### 3.3.4.f Scalability

A parallel system is referred to as scalable if it has the ability to maintain its efficiency at a fixed value by concurrently increasing the number of cores and size of the problem (Kumar et al., 1994). An iso-efficiency function can be used to determine the way for a parallel system to maintain a constant efficiency and hence to increase the speedup in proportion to the number of processing elements. The iso-efficiency function from (Kumar et al., 1994) can be defined as:

\[
\text{problem size}(W) = \frac{E}{(1 - E)} \cdot T_o(T_s, p)
\]

The parallel overhead function of EITRH for \( b \) blocks and \( n \) processing elements as given in Eq. (3.8) is approximately \( p.n_{call} \). Therefore,

\[
W = \frac{E}{(1 - E)} \cdot p.n_{call} \tag{3.13}
\]

hence, the asymptotic iso-efficiency function for this parallel framework is \( \theta(p.n_{call}) \). Therefore, if the number of processing elements are increased by a factor of \( p'/p \) (where \( p' \) is the increased number of processors) then the problem size (\( b \)) also must be increased by a factor of \( (p'.n_{call})/(p.n_{call}) \) to increase the speedup by a factor of \( p'/p \).
The results of the above stated metrics infers the EITRH framework to be efficient, cost optimal and scalable for different number of processing elements and problem sizes. Further, Section 3.3.5 proves the claims made in Section 3.3.4 by practical implementation of the framework with different scenarios.

3.3.5 Experimental Results

After the theoretical performance analysis of the proposed framework this section provides practical experimental results. In order to test the relative execution speed of EITRH construction (described in Section 3.3.1), the execution results of proposed hash function algorithm RSHA-1 are compared with the commonly used SHA-1 hash function. The sub-sections of this section covers the overall run time efficiency of the proposed algorithm with its effect on use of multiple cores and is finally compared with other existing hashing algorithms.

All the tests are executed on text files taken from the t5 corpus test suite containing 4457 different types of files. The unzipped size of these files is 1.78 GB from which files of size $2^x$ MB ($2 \leq x \leq 8$) were taken as test cases. The tests were compiled using the same compiler and configuration settings and gettimeofday() function of Linux was used to measure the execution time. For the optimality, compiler flags –g0 to disable debugging and –O3 to enable third level of optimization were used. The framework’s memory requirement is controlled by grouping all digest buffers into one linear buffer. The OpenMP implementation of the Algorithm 3.2 is given in Appendix C for the reader’s reference.

3.3.5.a Overall Runtime Efficiency

This section demonstrates the improvement of using parallel proposed hashing algorithm over standard sequential hashing algorithm- SHA-1. Both of these hash function algorithms generate the same length of hash value (160-bit). The algorithms during experiments were implemented with the same level of optimization and their execution time on a server with Setup I (described in Chapter 2, Section 2.3) using 8 cores with a thread pool model was recorded.
The thread pool model is chosen, as in terms of execution time one thread per core implementation takes longer time due to the overhead of the thread scheduler and poor memory management. Creating a lot of threads, more than the number of processors will be highly scalable. But for a single use it will be quite costly as it will triple the heap memory usage in comparison to the sequential version and will not be very effective. It is also slower when compared to the sequential version, due to excessive synchronization required between threads. So, in order to optimize the parallel implementation, a thread pool is created. This model disables the underfeeding or idling of a processor and gives optimum results. There are a limited number of threads, with less memory, although still high when compared to the sequential version. On the other hand, the execution time for this implementation is less than one fourth of the sequential version. The results of the execution are listed in Table 3.4.

RSHA-1 has shown significant performance in comparison to the standard sequential algorithm. The results show that in the case of small file sizes ranging from 2MB to 32MB, the speedup achieved is growing linearly that is from 2.8 to 6.34 but for larger file sizes speedup remains approximately constant with 85\% of improvement when executed on an eight core processor (as shown in Section 3.3.4.b). Figures 3.11 and 3.12 depicts the reduction in execution time for the RSHA-1 algorithm for both small and large file sizes in comparison with the standard SHA-1 algorithm.

Table 3.4 Execution Time Taken by SHA-1 and RSHA-1 (EITRH)

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA-1</td>
<td>RSHA-1</td>
</tr>
<tr>
<td>1</td>
<td>2MB</td>
<td>0.01750</td>
<td>0.00611</td>
</tr>
<tr>
<td>2</td>
<td>4MB</td>
<td>0.03275</td>
<td>0.01087</td>
</tr>
<tr>
<td>3</td>
<td>8MB</td>
<td>0.06744</td>
<td>0.01751</td>
</tr>
<tr>
<td>4</td>
<td>16MB</td>
<td>0.12619</td>
<td>0.02851</td>
</tr>
<tr>
<td>5</td>
<td>32MB</td>
<td>0.26092</td>
<td>0.04113</td>
</tr>
<tr>
<td>6</td>
<td>64MB</td>
<td>0.50228</td>
<td>0.07813</td>
</tr>
<tr>
<td>7</td>
<td>128MB</td>
<td>0.95009</td>
<td>0.14249</td>
</tr>
<tr>
<td>8</td>
<td>256MB</td>
<td>1.91486</td>
<td>0.27564</td>
</tr>
</tbody>
</table>
Fig. 3.11 Performance of RSHA-1 (EITRH) over SHA-1 for Small File Sizes

Fig. 3.12 Performance of RSHA-1 (EITRH) over SHA-1 for Large File Sizes
### 3.3.5.b Impact of Multiple Cores

This section discusses the influence of using the different number of cores for the RSHA-1 algorithm with different set of input data. Table 3.5 lists the results of execution time, speedup and efficiency for 1, 2, 4, 6 and 8 number of cores for the algorithm executed on Setup I (as discussed in Chapter 2, Section 2.3) by enabling/disabling the number of cores available on the processor. Depending upon the number of cores, the number of threads are decided. It can be inferred from the table that the execution time for an input file is the function of, the number of cores employed to complete the task. For each file, adding a pair of processing cores halves the execution time.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>SHA-1</th>
<th>RSHA-1 on different number of cores</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2MB</td>
<td>0.0175</td>
<td>0.01241</td>
<td>0.01003</td>
<td>0.00919</td>
<td>0.00758</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.41053</td>
<td>0.872383</td>
<td>0.476061</td>
<td>0.384785</td>
</tr>
<tr>
<td>2</td>
<td>4MB</td>
<td>0.03275</td>
<td>0.0232</td>
<td>0.01763</td>
<td>0.01438</td>
<td>0.01087</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.411638</td>
<td>0.928815</td>
<td>0.569367</td>
<td>0.502147</td>
</tr>
<tr>
<td>3</td>
<td>8MB</td>
<td>0.06744</td>
<td>0.04634</td>
<td>0.03428</td>
<td>0.02544</td>
<td>0.02055</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.45533</td>
<td>0.983664</td>
<td>0.662736</td>
<td>0.546959</td>
</tr>
<tr>
<td>4</td>
<td>16MB</td>
<td>0.12619</td>
<td>0.08623</td>
<td>0.06504</td>
<td>0.03543</td>
<td>0.02851</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.463412</td>
<td>0.970095</td>
<td>0.890418</td>
<td>0.737694</td>
</tr>
<tr>
<td>5</td>
<td>32MB</td>
<td>0.26092</td>
<td>0.17038</td>
<td>0.12519</td>
<td>0.06669</td>
<td>0.04987</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.5314</td>
<td>1.042096</td>
<td>0.978108</td>
<td>0.872001</td>
</tr>
<tr>
<td>6</td>
<td>64MB</td>
<td>0.50228</td>
<td>0.33893</td>
<td>0.24408</td>
<td>0.12958</td>
<td>0.09459</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.481958</td>
<td>1.028925</td>
<td>0.969054</td>
<td>0.885013</td>
</tr>
<tr>
<td>7</td>
<td>128MB</td>
<td>0.95009</td>
<td>0.67417</td>
<td>0.49203</td>
<td>0.25421</td>
<td>0.18018</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.409274</td>
<td>0.96548</td>
<td>0.934355</td>
<td>0.878834</td>
</tr>
<tr>
<td>8</td>
<td>256MB</td>
<td>1.91486</td>
<td>1.34524</td>
<td>0.97652</td>
<td>0.49931</td>
<td>0.34769</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.423434</td>
<td>0.980451</td>
<td>0.958753</td>
<td>0.917896</td>
</tr>
</tbody>
</table>

Table 3.5 Execution Results of RSHA-1 (EITRH) on Different Number of Cores
Figure 3.13 illustrates the impact of using multiple cores. It can be concluded from the figure that for large file sizes as the number of cores increases it shows a significant performance gain but for small sized files increasing cores doesn’t show a greater impact on the execution time and thus does not impact efficiency. File size and number of processors are directly correlated as the size of file and number of processors increases, the execution time decreases. For the RSHA-1 algorithm improvement of 60% to 85% can be observed by increasing the number of processing cores.

![Fig. 3.13 Performance of RSHA-1 with the Increasing Number of Cores](image)

As discussed in Section 3.3.4 the speedup gain on the basis of theoretical results was linear, so authenticity of the results was verified by the practical implementation of the algorithm. Figure 3.14 shows the speedup gain for a particular file size with respect to the increasing number of processors. It is observed that as the number of processing cores are increasing, the speedup is increasing almost linearly. But, as a consequence of Amdahl’s Law there will be a saturation point and speedup tends to be constant and efficiency starts dropping.
Although, efficiency can be kept constant by increasing both the size of problem and number of processing cores simultaneously. For the RSHA-1 algorithm in Table 3.6, the efficiency of finding hash value of 2MB of file on 2 processors is 0.88. If the number of processing cores is increased to 4 and the problem size is scaled to 16MB, the efficiency remains 0.88. Increasing $p$ to 8 and $n$ to 1GB results in same efficiency. Therefore, the RSHA-1 algorithm can be considered scalable, as it is able to maintain the efficiency at a fixed value by increasing problem size and number of processing cores simultaneously.

Table 3.6 Efficiency of RSHA-1 as a Function of Problem Size and Number of Processing Elements

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Number of Processing Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$p = 1$</td>
</tr>
<tr>
<td>2MB</td>
<td>1.0</td>
</tr>
<tr>
<td>16MB</td>
<td>1.0</td>
</tr>
<tr>
<td>128MB</td>
<td>1.0</td>
</tr>
<tr>
<td>1GB</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figures 3.15 and 3.16 shows the variation in efficiency as file size increases with the fixed number of processors i.e. 8 and as the number of processing
elements increases for a fixed problem size i.e. 512 MB. It is deduced that the efficiency of the parallel algorithm increases, if the file size is increased while keeping number of processing elements constant and decreases vice-versa.

![Graph 1](image1)

**Fig. 3.15** Variation in Efficiency in RSHA-1 as File Size Increases with Fixed Number of Processors (8)

![Graph 2](image2)

**Fig. 3.16** Variation in Efficiency in RSHA-1 as Number of Processing Elements Increases for a Fixed Problem Size (512 MB)
3.3.5.c Comparative Analysis

After the performance measures of the RSHA-1 algorithm, the results of the EITRH variant RSHA-1 are also analyzed with other existing hashing algorithms. Table 3.7 shows comparison results of RSHA-1 with SHA-1, SHA-256 and SHA-512 algorithms in terms of MB/s and cycles per byte on an AMD FX(tm)-8120 Bulldozer; 4 x 3100 MHz processor. The MB/s performance is extrapolated from the CPU clock speed. The results for comparison of these three algorithms were taken from SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives) cryptographic benchmarking software (Bernstein and Lange, 2009). It is a toolkit developed by the VAMPIRE lab in order to measure the performance of cryptographic softwares including the performance of hash functions, symmetric stream ciphers, asymmetric encryption systems, public-key signature systems, and public-key secret-sharing systems. It has records for fifty odd wide varieties of systems for each hash function, ensuring direct comparability of all. The experiments are conducted on a single core of the processor so as to make them comparable.

Table 3.7 Comparison of RSHA-1 with Standard Serial Hashing Algorithms

<table>
<thead>
<tr>
<th>CPU Architecture</th>
<th>Frequency</th>
<th>Algorithm</th>
<th>Cycles/byte</th>
<th>MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD FX(tm)-8120</td>
<td>3.1 GHz</td>
<td>SHA-1</td>
<td>6.07</td>
<td>487.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA-256</td>
<td>21.72</td>
<td>136.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA-512</td>
<td>13.74</td>
<td>215.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSHA-1</td>
<td>15.93</td>
<td>185.55</td>
</tr>
</tbody>
</table>

The results show that RSHA-1 outperforms SHA-256 but not SHA-1 and SHA-512 in terms of cycles per byte and megabytes per second due to its parallel nature. The implementation of the algorithm on a single core leads to an overhead for RSHA-1 as the code is meant for multiple cores but still the performance is acceptable as compared with the existing sequential hashing algorithms. Moreover, today’s computer architectures are not longer single
core, a basic computer system comes with a dual core processor. So, it can be concluded that RSHA-1 will have a higher acceptance rate in present and coming computer generations.

Further, comparisons are also performed on other hashing algorithms based on tree hashing (SHA-3 candidates) like MD6 (Rivest et al., 2008) and Skein (Lucks et al., 2008) discussed in Chapter 1. Table 3.8 shows the result for RSHA-1, MD6 (on CPU and GPU) and Skein-512 in terms of cycle per byte and MB/s. In this RSHA-1 has shown better results than MD5 and Skein when executed on a 8-core machine and on large data set (1GB in size).

Table 3.8 Comparison of RSHA-1 with Other Tree Hashing Based Algorithms

<table>
<thead>
<tr>
<th>CPU Architecture</th>
<th>Frequency</th>
<th>Algorithm</th>
<th>Cycles/byte</th>
<th>MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD FX(tm)-8320</td>
<td>3.1 GHz</td>
<td>RSHA-1</td>
<td>2.81</td>
<td>1052.5</td>
</tr>
<tr>
<td>AMD Barcelona</td>
<td>2.2 GHz</td>
<td>MD6</td>
<td>3.89</td>
<td>539.8</td>
</tr>
<tr>
<td>AMD Barcelona with 8800GT GPU Card</td>
<td>2.2 GHz</td>
<td>MD6</td>
<td>3.44</td>
<td>610</td>
</tr>
<tr>
<td>Intel Core 2 Duo</td>
<td>3.1 GHz</td>
<td>Skein-512</td>
<td>6.5</td>
<td>454</td>
</tr>
</tbody>
</table>

Table 3.9 shows a comparison of EITRH with other parallel hashing techniques, HAIFA based Blake (Aumasson et al., 2008), Permutation based Grøstl (Gauravaram et al., 2008) and Sponge based Keccak (Bertoni et al., 2013) constructions (discussed in Chapter 1). The results for these algorithms are also taken from SUPERCOP cryptographic benchmarking software (Bernstein and Lange, 2009) on an AMD FX(tm)-8120 Bulldozer; 4 x 3100 MHz processor and are analyzed for performance. The results show that the RSHA-1 algorithm based on EITRH construction again performs better than other parallel hashing algorithms for long messages in terms of cycle per byte and MB/s.

Table 3.9 Comparison of RSHA-1 with Other Parallel Hashing Algorithms

<table>
<thead>
<tr>
<th>CPU Architecture</th>
<th>Frequency</th>
<th>Algorithm</th>
<th>Technique</th>
<th>Cycles/byte</th>
<th>MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD FX(tm)-8120</td>
<td>3.1 GHz</td>
<td>RSHA-1</td>
<td>EITRH Construction</td>
<td>5.27</td>
<td>561.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Grøstl-256</td>
<td>Permutation Based</td>
<td>13.29</td>
<td>222.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blake 512</td>
<td>HAIFA Construction</td>
<td>6.89</td>
<td>429.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Keccak</td>
<td>Sponge Construction</td>
<td>10.77</td>
<td>274.50</td>
</tr>
</tbody>
</table>
Conclusion

In this chapter, we proposed a fast parallel recursive tree-based transformation called EITRH. Tree hashing along with EMD construction has been used in designing EITRH functions: message expansion, parallel reduction and hash value generation. EITRH is proved to be a flexible hash transformation that generates different lengths of hash codes using different variants: RSHA-1, RSHA-224, RSHA-256, RSHA-384, and RSHA-512.

In order to analyze the performance of the new framework, it has been tested and analyzed for few performance benchmarks including speedup, efficiency, cost optimality and scalability. We have compared the execution results obtained by using OpenMP API for parallelization, with a well-known hash function (SHA-1) for different file sizes and number of processing elements; the comparison results showed that RSHA-1 performs better than existing hash functions which are commonly used. Speedup of approximately 6.5X has been observed with 625.24 MB/s of speed and efficiency of 0.88 when executed on an AMD Bulldozer FX machine. Along with this, RSHA-1 has also shown better performance than other existing parallel hash-based schemes from the literature. These results show that the proposed algorithm has high potential for adoption in various cryptographic applications.
CHAPTER 4
SECURITY ANALYSIS OF EITRH
Chapter 4

SECURITY ANALYSIS OF EITRH

Security is of great relevance for the CHFs. For digitally signing a document or proving authenticity of a digital document, hashed data can be used as an evidence in the court of law. Therefore, a new hashing scheme would by no means be even considered for adoption if not backed by a credible security analysis. This chapter provides security analysis of the proposed framework that satisfies the essential properties required for a CHF. Apart from the essential properties, additional properties have been proven for its acceptance in multiple applications.

4.1 Introduction

The proposed transformation has been developed to be used in a wide range of applications including but not limited to digital signatures, key derivation, pseudorandom number generation, and stream cipher usage. It can also support personalized and randomized hashing. When supplied with a secret key, EITRH can be used for message authentication and also used as a pseudorandom function. Security of CHFs can be evaluated on the basis of many criteria and is usually application dependent, where hash functions are expected to preserve a number of properties. However, if a particular hash function \( H \) fails to preserve a particular property \( Z \) this does not immediately infer that the hash function is broken in practice. It just means that \( H \) cannot be “securely” used in applications where property \( Z \) is required.

Further, Section 4.2 discusses in detail some of the security measures of the proposed transformation. In sub-sections of Section 4.2, security proofs for generic properties of CHFs to claim its usage in various applications is provided.
4.2 Security Analysis

Security of any CHF mainly depends on the satisfaction of three main properties: Pre-image resistance, Second Pre-image resistance and Collision resistance. Towards the goal of building strong, multi-purpose hash functions, satisfaction of ideal properties is not sufficient. A new transformation that preserves multiple properties, known as multi-property preserving domain extension transformation, is required. The EITRH transformation is built with the motive of such type of transformation which can preserve multiple properties.

We will present Pre-image resistance, weak collision resistance, collision resistance, partial pre-image resistance, non-correlation resistance, and pseudo random oracle preservation properties of the EITRH transformation.

![Diagram](image)

Fig. 4.1 (a) Pre Image Collision Resistance, (b) Weak Collision Resistance, (c) Strong Collision Resistance

4.2.1 Pre-image Resistance (PIR)

CHFs are considered to be computationally non-invertible which means, if a hash code $H(M)$ is generated for a message $M$, it is considered to be computationally infeasible for an adversary ($A$) to retrieve the original message ($M$) (illustrated in Fig. 4.1(a)). The pre-image resistant property also known as
*one-way* assures the non-reproducibility of the original message. The advantage of an adversary \((A)\) for finding a collision in a CHF can be defined as:

\[
Adv_H^{nir[m]}(A) = \Pr[M \leftarrow \{0,1\}^n; X \leftarrow H(M); M'' \leftarrow A(X); H(M'') = X;]
\]

For a CHF to be Pre-image resistant, brute force attacks are considered to be the best attacks. In a brute force attack, random values for a message \(M''\) are taken and tried until a collision is found. The level of effort required to find a collision is proportional to \(2^n\), for an \(n\)-bit hash value. On average, an attacker needs to try \(2^n - 1\) different values of \(M''\) in order to generate the same hash code \(h\). The attack does not depend on any specific algorithm but depends only on the bit length of the hash value. So, it is considered that the greater the hash code length the more secure the hash function is, as the complexity of finding collision also increases with the increase in the hash size. Therefore, a variant of EITRH can be chosen effectively depending upon the strength of security required to make the brute force attack more difficult. Moreover, finding a pre-image in EITRH for an attacker is far more difficult than finding it in Merkle-Damgård hash functions. The recursive nature of EITRH and the hash of unique hashes makes it computationally infeasible to backtrack the input message.

### 4.2.2 Weak Collision Resistance (WCR)

For a CHF to be weak collision resistant or second pre-image resistant, it should be computationally infeasible for an adversary \((A)\) to find two different messages \(M\) and \(M''\) which generate the same hash values from CHF. That is, to find \(M, M''\); such that \(H(M) = H(M'')\) but \(M \neq M''\) (illustrated in Fig. 4.1(b)). Briefly,

\[
Adv_H^{wcr[m]}(A) = \Pr[M \leftarrow \{0,1\}^n; M'' \leftarrow A(M); M\neq M'' \land H(M) = H(M'');]
\]
This property thwarts the falsification of the message in case an encrypted hash code is used. The level of effort required to find a collision is proportional to $2^n$, for an $n$-bit hash value. In Section 4.2.4 we will prove that, in EITRH transformation, a single bit of change in the message/input leads to more than 50% change in the hash value. Therefore, finding two different messages with the same hash value will be computationally infeasible, as $2^n$ possibilities are required for making a collision. With increase in the size of $n$, the level of effort increases exponentially.

4.2.3 Collision Resistance (CR)

For a CHF to be strong collision resistant, it should be computationally infeasible for an adversary ($A$) with given CHF $H$ and message $M$ to find another message $M''$ where $M \neq M''$ and $H(M) = H(M'')$ (illustrated in Fig. 4.1(c)). Briefly,

$$Adv_{H}^{cr}[m](A) = \Pr[(M, M'') \leftarrow A: M \neq M'' \land H(M) = H(M'')];$$

This type of attack involves much less effort than a pre-image or second pre-image attack. Rogaway in (Rogaway, 2006) has given an important caution that these notions of collision resistance and second pre-image resistance are only defined for keyed hash functions, but for keyless hash functions, they act as non-collision resistant. He observed that, there always exists an efficient algorithm that can find another message $M''$ such that $H(M) = H(M'')$ due to the pigeonhole principle. However, it is hard for a human to find such an efficient collision finding algorithm. Therefore, the claim that a CHF is collision resistant doesn’t mean that there is no way to find a collision, rather it is practically infeasible for a human to find. The level of effort required to find a collision is proportional to $2^{n/2}$, for an $n$-bit hash value.

Consider two message inputs $M, M''$ (where, $M \neq M''$) of same lengths for any instance of EITRH, and assume that the difference $\Delta$ in the message inputs is in one half of the inputs. Now, collision can occur only in one half of the
intermediate outputs generated at any stage before involving the second half in the next stage leading to partial collision. Here two cases can be considered: collision in the intermediate hash or collision in the data proceeding to the next stage. Since the function is recursive in nature, so a collision in one half of the intermediate hash output ensures a difference of Δ in the other half of the intermediate hash output. Therefore, collision is unlikely to persist in the succeeding stages, as in these stages, the subsequent input to the non-linear function \( f() \) along with the sub-key values will be different from the one in previous stages.

### 4.2.4 Non-correlation Resistance (Confusion and Diffusion Analysis)

CHF’s inputs and outputs should not be statistically correlated; like encryption methods, CHFs also necessitate to diffuse the influence of the complete input message into the hash value space. The avalanche effect proves to be a desirable property for CHF to be an ideal hash function. According to it, the correlation between the message and corresponding hash value bits must be complex and any change in the message should lead to a drastic (probability of at least 50%) change in the hash value.

EITRH supports the avalanche effect and any change in the original message leads to more than fifty percent change in the hash code generated. To prove this, a test of confusion and diffusion is performed on a simulator with a random text file from t5-corpus whose hash value is calculated and stored. Thereafter, a random bit is selected in the file and toggled to generate a new hash value in binary format. Now, a comparison between two binary hash values is performed and the number of bits changed at the same location \( L_i \) are counted. The test is repeated \( N \) times on the algorithm (where \( N = 128, 256, 512, or 1024 \)) and the corresponding distribution of changed bit number \( L_i \) is calculated as shown in Fig. 4.2. The statistics used for the algorithm are defined as follows:
Minimum number of bits changed: \( L_{\text{min}} = \min(L_i^N) \)  \( (4.1) \)

Maximum number of bits changed: \( L_{\text{max}} = \max(L_i^N) \)  \( (4.2) \)

Mean of number of bits changed: \( L = \frac{1}{N} \sum_{i=1}^{N} L_i \)  \( (4.3) \)

Mean Changed Probability: \( Pr = \left( \frac{L}{\text{HashSize}} \right) \times 100\% \)  \( (4.4) \)

Standard Deviation of bit number changed:

\[
\sigma_L = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (L_i - \bar{L})^2} \quad (4.5)
\]

Standard Deviation:

\[
\sigma_{Pr} = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} \left( \frac{L_i}{\text{HashSize}} - Pr \right)^2} \times 100\% \quad (4.6)
\]

Here, \( N \) is the total number of statistics, \( L_i \) is number of bits changed in the \( i^{th} \) test, and \( \sigma_{Pr}, \sigma_L \) indicate the stability of confusion and diffusion.

The statistics \( L_{\text{min}}, L_{\text{max}}, L_i, Pr, \sigma_L, \sigma_{Pr} \) obtained using the Eqs. (4.1), (4.2), (4.3), (4.4), (4.5) and (4.6) respectively are shown in Table 4.1. Analysis of the table shows that the mean values obtained for \( L_i, Pr \) are 83.94 and 52.46% respectively, which are very close to ideal values of 64 bit (where the hash code size is 160 bits), 50% as desired and \( \sigma_L, \sigma_{Pr} \) indicates the stable capability of confusion and diffusion of the algorithm. The results show that the proposed algorithm is resistant to statistical attacks.

Table 4.1 Statistics of Number of Changed Bit \( L_i \)

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Number of times test taken (N)</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_i )</td>
<td>128 256 512 1024</td>
<td>83.94263</td>
</tr>
<tr>
<td>( Pr% )</td>
<td>51.78223 52.88086 52.27783 52.91565</td>
<td>52.46414</td>
</tr>
<tr>
<td>( \sigma_{Pr}% )</td>
<td>6.071309 6.209049 6.191711 6.039089</td>
<td>6.127789</td>
</tr>
<tr>
<td>( L_{\text{min}} )</td>
<td>63 58 60 61</td>
<td>60.5</td>
</tr>
<tr>
<td>( L_{\text{max}} )</td>
<td>100 98 103 115</td>
<td>104</td>
</tr>
</tbody>
</table>
4.2.5 Partial Pre-image Resistance (PPR)

Partial pre-image resistance or local one-wayness, is the difficulty in retrieving part of an original message from its digest as opposed to retrieving the entire message, even if a portion of the message is already known. The EITRH class of hash functions satisfies this property as well. Given that the message digest of a message is hashed using the EITRH-based hash function, it is difficult for an attacker to generate a part of the original message. The final hash code generated by EITRH is a hash of \( n \) unique hashes where \( n \) is the number of intermediate hashes at each level and thus, can never lead to a part of original message.

4.2.6 Pseudorandom Oracle Preservation (PRO-Pr)

EITRH transformation is based on tree hashing and EMD transformation (as discussed in Chapter 3). Like EMD, EITRH also uses two initial vectors, second one for the last block of each level. This input is sufficient to cause the last application of random oracle to behave independently with high probability and makes it PRO-Pr by standard model proof, as EMD is itself PRO-Pr (Bellare and Ristenpart, 2006).
Conclusion

Several computer simulations and theoretical analysis on EITRH class hash functions showed that the proposed algorithm satisfies the characteristics and conditions required for CHFs such as, collision resistance, high bit confusion and diffusion, uniform distribution, flexibility, and fast speed. With the security analysis, it can be concluded that the EITRH transformation based algorithm satisfies the essential properties of CHF sufficiently to make it useful for authentication purposes and has high-potential for adoption in security applications.

EITRH is confirmed as a good candidate for texts/images authentication with high flexibility, confusion and diffusion, and fast speed. Apart from these claims, a few randomness tests are also performed on the transformation to check the sensitivity of hash values generated by any instance of EITRH and prove its feasibility for digital forensic applications. The specifications and results of these tests are provided in Chapter 5, Section 5.5.1.
CHAPTER 5
FORENSIC APPLICATION
OF EITRH
Chapter 5

FORENSIC APPLICATION OF EITRH

CHFs are being widely used by digital forensic tools to ensure the authentication and integrity of the device investigated. However, CHFs that are currently used are serial in nature and with the growing capacity of storage media it becomes a time consuming process when working with large data sets. This chapter discusses in detail the use of the EITRH class parallel hashing algorithm in the digest generation process and also compares the execution results with the existing sequential methods. The improvement of using the framework while comparing the results with the original SHA-1 hash algorithm has also been demonstrated.

5.1 Introduction

In our day-to-day lives, the amount of digital data is increasing at an exponential rate with the prevalent use of digital devices. The information is stored on different computing systems including desktops, laptops, smartphones, tablets, PDA’s, etc. Production of these devices has already crossed the ten digit figure with smartphones alone crossing the one billion mark and the number of devices increases every year. Along with the increase in numbers the connectivity of digital devices is also increasing.

With the popularity and growth of information technology digital crime has also increased at a tremendous rate. Cybercrimes including DoS attacks (Denial of Service), intellectual property thefts, child pornography, frauds, extortions, cyber stalking, spam mailing, identity theft, and hacking have become common. All these crimes need to be investigated and end up in the court of law in the presence of proper evidence.

In a typical crime investigation which involves computing and communication devices, the use of digital forensics tools has become necessary. With the advancements in the technology there have been changes in criminal laws periodically. On the basis of these laws, many commercial and open source
digital forensic tools (Reith et al., 2002) have been developed as an aid for the investigators and analysts.

Despite the presence of so many forensic investigation tools the increasing size of digital devices has made the job of investigators cumbersome. According to (Rowlingson, 2004, Rogers et al., 2006), seizing, imaging, and searching approach of the investigation is no longer feasible for large-scale examinations. Widely used investigation tools EnCase and FTK also don’t support any special feature to handle large data sets (usually in terabytes).

Researchers are working to come up with a solution to the problem of handling huge amounts of data. The techniques proposed by (Carrier, 2003, Beebe and Clark, 2005, Richard III and Roussev, 2006, Rogers et al., 2006, Abraham, 2006, Beebe and Dietrich, 2007, Liebrock et al., 2007, Roussev et al., 2009, Beebe, 2009, Polliitt, 2010, Tanner and Dampier, 2010, Bhat et al., 2010), are speeding up one or the other phases involved in digital investigation process (discussed in Section 5.2.1). However, all these techniques focus on filtering and searching factors of the process but not on authentication of the digital evidence. Hash codes are used to ensure authentication and integrity of the data stored on suspected digital devices and forensic processing during acquisition, verification, and examination phases of the investigation.

The process of hash generation also needs to be speeded up with the increasing size of data. This issue has been solved in the chapter with the parallelization of the hash code generation process during the acquisition phase of the evidence.

In Section 5.2 we have discussed the concept of digital forensics in brief followed by the investigation process and role of hashing in digital forensics in its subsequent sub-sections. Section 5.3 covers the current approaches currently used and Section 5.4 discusses the application of EITRH transformation in digital forensic tools. The experimental results and performance analysis of the new approach are given in Section 5.5.
5.2 **Digital Forensics**

Digital Forensics (DF) (Noblett et al., 2000, Palmer, 2001b, Kishore et al., 2014) is another branch of forensic science which is developed as a response to the increasing number of crimes which involve digital information either in planning or committing a crime. DF focuses upon the recovery, investigation and analysis of material found in digital devices, networks, social networking sites etc. Investigation of all these devices that store digital data can provide various digital evidence against crimes like cyber harassment, child pornography, planning of a murder or a robbery, burglary of data and information from the computer system, hacking, website defacement etc.

Digital forensics originated in the late 1970s and early 1980s where the Comprehensive Crime Control act was enacted in 1984. After the passage of the Computer Fraud and Abuse Act §18 U.S.C §1030 (1986) by the United State Congress, computer hacking became a crime and opened the doors to new advances in research and development in the field of digital forensics. With the change in technology the FBI and other law enforcement agencies and research groups like Scientific Working Group on Digital Evidence (SWGDE), the National Institute of Justice (NIJ), the Technical Working Group on Digital Evidence (TWGDE), and the Computer Analysis and Response Team (CART) are developing laws, programs, applications, and standardized approach to examine evidence present in digital form (Noblett et al., 2000).

DF can be formally defined as:

**Definition 5.1:** “The use of scientifically derived and proven methods toward the preservation, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations.” (Palmer, 2001b)
Figure 5.1 shows a typical digital forensic investigation model. DF aims to do structured investigation and then discover what happened exactly on digital system and who was responsible for it. This is accomplished by collecting, identifying and validating the digital information so that it could be used for the reconstruction of past events. As per some of the estimates, about 95% of criminals leave evidence which can be captured and analyzed through proper computer forensic procedure. The forensics relies on a set of tools and techniques that can be applied to suspects, victims, and bystanders.

The job of a Digital Forensic Specialist (DFS) is to scrutinize all the digital sources that are available and which are believed to be involved in the crime; generate a summary report comprising the contents of crime sources on the basis of investigation and analysis performed. The DFS follows certain methodology to ensure integrity, authenticity, reproducibility, and non-interference for the report to be acceptable in the court.
5.2.1 Digital Forensic Investigation Process

In a digital forensic investigation process a scientific investigation methodology is followed to scrutinize digital evidence. Eoghan Casey in (Casey, 2011) has defined the process as a number of steps involved in the investigation from incident alert to the investigation report submission. Due to the advances in technology and consecutively growing size and diversity in digital crimes, there have been tremendous changes in the digital investigation process over time. Many models and frameworks have been found in the literature (Lee et al., 2001, Kruse II and Heiser, 2001, Palmer, 2001a, Investigation and America, 2001, Reith et al., 2002, Kizza and Kizza, Baryamureeba and Tushabe, 2004, Standards et al., 2004, Ciardhuáin, 2004, Carrier and Spafford, 2004, Cohen, 2009, Yen et al., 2009). However, there is no as such fixed standard process to be followed.

Fig. 5.2 Digital Forensic Investigation Process

Figure 5.2 depicts a generic model that covers preservation of evidence from any modification, detection of the data concerned, recovery of the deleted data, and analysis of the data and its documentation followed by an expert consultation.
5.2.1.a Identification

Identification involves recognizing digital devices as evidence from possible recognized metrics by the experienced investigators.

5.2.1.b Preservation and Acquisition

Preservation is freezing the evidence to safeguard it from any activities that can damage the digital evidence or the information contained in it. It may include isolating the system, avoiding rebooting or running any application on it for preserving the physical and digital state of the evidence.

After the acquisition and preservation of the evidences, the next very first step is to save them in the same state as acquired so that they can be analyzed later. An image and hash is generated for the files and data by using standard forensic tools to use it as evidence later on.

5.2.1.c Examination and Analysis

Log files, files, directory contents, events log, data and information are processed forensically by using manual and automatic tools. It also involves extraction of registry information, access of information hidden or deleted without hindering the integrity of data.

After the examination, scrutiny of the evidence is performed in the analysis phase. This is done by using legally justifiable tools and techniques to obtain useful information with respect to the crime. The significance of the evidence is determined and evaluated to aid in evidence presentation.

5.2.1.d Decision and Reporting

On the basis of examination and scrutiny of evidences, decision is made and a written report is generated. The report includes a detailed outline of the examination process along with the record of data collected and conclusions. The report is thereafter presented in front of the Court in support of the case.


5.2.2 Hashing in Digital Forensic

Digital evidence is considered as circumstantial and hearsay in many cases (Cohen, 2009, Casey, 2011). The reason for such consideration is due to the lack of eyewitnesses who can actually confirm a fact. In a digital investigation, the investigator discovers the traces of the actual event, establishes a connection and gives its explanation. Since the investigator has not seen the actual event and cannot reproduce the actions, therefore the evidence is sometimes considered as hearsay and is treated null and void (Schwikkard and Van der Merwe, 2009).

In order to make findings of digital evidence acceptable in the Court of Law, authentication of the evidence is essential. In the paper (Hershensohn and Block, 2005), the legal necessities that need to be satisfied for authenticity are given. According to the article, the content of the data must not be changed; the data worked upon should be a duplicate of original source; and accurate metadata should be presented.

For all this, authenticity of a digital evidence is ensured by generating a hash value of the content of data and verifying it for the image file at later stages while presenting (Carrier, 2002, Ballou, 2010). Hashing process not only maintains the authenticity but also helps in maintaining integrity of the evidence for both dead and live analysis.

After the preservation of the digital device suspected to be evidence, the next step before the examination phase is to generate an image of the data stored on that device. A duplicate copy of the data is then scrutinized rather than the original source. In order to authenticate the duplicate copy to be examined, hash code of the original source and its duplicate copy is calculated and preserved to be presented in front of the Court later during the trial. Matching of both the hash codes generated proves that no unwanted alteration or tampering has been made to the data on the digital evidence and is authenticated.
The process of forensic investigation is accomplished with the use of various tools and software makes use of CHF to create the code. SHA-1 and MD5 algorithms are widely used to generate hash code of a file for ensuring integrity and authenticity of digital evidence, like the tools EnCase, HashKeeper, FTK makes use of MD5 and the tools TSK and Helix makes use of both the algorithms. However, due to the security issues found in MD5 (Schneier, 2005, Wang and Yu, 2005, Hawkes et al., 2004), and SHA-1 their continuation for use has become questionable.

Apart from ensuring authenticity and integrity during acquisition, hash functions are also used to identify known files: known-to-be-good and known-to-be-bad files and to efficiently identify large sets of files in the form of bloom filters (Carrier, 2003, Roussev, 2009). Known-to-be-good or bad files are precompiled files stored in databases and are verified manually. The databases that maintain an exhaustive list of each file entry containing digests of software, common operating system files, applications, etc. include Reference Data Set (RDS)² of National Software Reference Library (NSRL) (Biham and Chen, 2004) maintained by NIST and HashKeeper database of National Drug Intelligence Center. Investigators make use of these databases extensively to filter out the noisy data. The technique of simple hashing, fuzzy hashing (Roussev, 2009) is used to do the reduction.

However, as the memory size of storage media is increasing the problems for investigators (technical as well as legal) are also increasing. The size of HashKeeper and RDS databases is also increasing rapidly with the growing size of digital information leading to low throughput. Time constraints and increasing storage capacity have put a sword on the investigator’s neck.

²http://www.nsrl.nist.gov
5.3 Current Approaches

In order to overcome the problem of growing storage capacity and hence the forensic investigation distributed forensics has become a viable alternative. The power of multi-core processing systems can be used to meet the time-sensitive requirements of the digital forensics process. Currently a few techniques have been applied and proposed to be used in DF.

Van den Hengel et al. (2008) had proposed a system in which distributed computing and image processing has been used together to quickly extract relevant pieces of data for the investigator. The system has been proved by Roussev et al. (2009) with the use of the MPI MapReduce model.

Liebrock et al. (2007) had proposed a system to handle the process of imaging by the use of parallel computing and visual analytics. Richard III and Roussev (2006) proposed another distributed environment in which the load is spread over multiple cores of the system by utilizing more CPU cycles and processing data in RAM. However, the approach is not cost efficient as it experiences more hardware and power related costs.

AccessData has provided a feature of support of multi-core processing for the password recovery process in its Forensic Toolkit (FTK) product. However, the kit uses the power of multiple machines in parallel just for this process and other digital forensic processes like imaging, indexing, and searching are still not taking benefit of it.

As the size of digital data exceeds terabytes and time consumption and security thwarts are also increasing, there is a need to overcome the drawback of current CHFs available. Other than CHFs, byte-wise approximate matching algorithms (Breitinger et al., 2014, Breitinger and Roussev, 2014) have also been proposed which are still under analysis for acceptance as a substitute to CHFs in DF.
All of the approaches discussed above had used parallelism in different forensic investigation process phases other than in the digest generation phase. Section 5.4 presents a new approach for generating a signature or hash code of a file in a parallel mode by making the use of power of multi-core processors to speed up the digest generation process and compliment the distributed forensic technique.

5.4 Parallelized Framework for File Hashing

The concept of black-listing and white-listing was introduced in order to match the list of hash codes of black/white listed files with the suspected system’s files and to directly detect them. However, this concept fails when an alteration is made in the suspected file to remove it from the list of blacklisted files. Anti-blacklisting has become easier and has also made the job of a DFS cumbersome. Therefore, new ideas are required to tighten and speed up the forensic investigation process.

Parallelizing the image generation process is one of the solutions to the problem and moreover, makes investigating each file easier. The parallel framework, EITRH proposed in Section 3.3 can also be used in the process of digest generation for forensic investigation. The proposed framework can reduce the digest generation time by making the use of multiple cores of the devices and can help DFS to investigate the files thoroughly. Currently, the framework is designed to generate the digest of a file to maintain the integrity, but it can also be later converted to similarity preserving hash algorithm with its own hash tables.

With the same motive, the framework was implemented on devices like hard disk and pen drives containing different type and size of files. The results of implementation are provided in Section 5.5.
5.5 Performance Analysis

In order to analyze EITRH transformation in DF application, the performance and randomness of EITRH variant RSHA-1 with that of standard SHA-1 is checked in this section. Assessment of the new framework is done by implementing the algorithm with a test environment on a server with a configuration of Setup I as provided in Chapter 2, Section 2.3.

Two different types of tests were performed to check:

- Randomness of the transformation
- Performance of the transformation

Section 5.5.1 shows the randomness and sensitivity test for both, text and image files, followed by performance analysis in Section 5.5.2. All the test case files are of different sizes ranging from few KB to GBs. The correctness of the results is also measured by calculating the hash code of a particular file by storing it on a hard disk, pen drive, and on an external hard disk. Further, experiments are conducted on compressed folders to analyze the run time efficiency of the algorithm and are also estimated for a computer system on the basis of results obtained.

5.5.1 Randomness Test

In order to test the sensitivity and randomness of the hash values generated by EITRH class hash functions, a simulator was created which could make the changes in original file and track the changes in hash code generated by tampered file. The target was to check the avalanche effect of the framework and its adaptability in forensic investigations. The tests were performed on text as well as on image files.

5.5.1.a Tests on Text File

The first test was performed on a text file with a message “A cryptographic hash function is a hash function which is considered practically impossible to
invert, that is, to recreate the input data from its hash value alone. These one-way hash functions have been called "the workhorses of modern cryptography". The input data is often called the message, and the hash value is often called the message digest or simply the digest.” Few conditional changes were done in the file to get fifteen different hash simulation results, see Appendix A for the results. The changes done in the file are shown as C1 to C15:

C1: Original file
C2: Lowercase the first character
C3: Convert “hash” to “cash”
C4: Add a blank space at the end
C5: Swap first message block with the second
C6: Add a digit 1 at end
C7: Change the first character “A” in the original message into “B”.
C8: Remove full stop from the end of the statement.
C9: Remove the blank space between "A" and "cryptography".
C10: Remove the comma after the special character “.
C11: Remove “,” from the text
C12: Calculate the hash value with \( K_0 = 0xC1059ED8, 0x367CD507, 0x3070DD17, 0xF70E5939, 0xFFC00B31 \)
C13: Change the secret key to \( K_0 = 0x68581511, 0x64F98FA7, 0xBEFA4FA4, 0x9B05688C, 0x1F83D9AB \)
C14: Replace one with “1”
C15: Blank Message
Fig. 5.3 Binary Hash Values of Text File under Different Conditions
Figure 5.3 shows the hash codes received in binary form after the conditional changes done in the file via simulator. The results $C_{12}$ and $C_{13}$ are for the keyed version of RSHA-1, where $H_{00}$ is replaced with new $K_0$ (key values) and $C_{15}$ for a blank message. All other results involve slight changes in the text. It is observed that, even a single bit of change in the message leads to a variation with more than fifty percent changes in the hash code generated. The same set of experiment was conducted on several text files and the results clearly justify the claims made regarding the sensitivity of the framework.

5.5.1.b Tests on an Image

Steganography tools are commonly used to embed images with some code or secret message so as to disguise them from the Black-listing test of DFs. The Least Significant Bit (LSB) is used for embedding something other than color information in the image to make it undetectable by the human eye. Due to this, it is possible to encode one letter of ASCII text for every three pixels in a bitmap image.

The same technique was used to test the randomness in the hash code for an image file. For this purpose, two different image files were taken, one with sparse information (in bmp format) and other with dense information (in jpeg format) as shown in Fig. 5.4 and Fig. 5.5 respectively. An online steganography tool\(^3\) was used to embed some code into both the images. After that, hash code of the original images (Fig. 5.4 and Fig. 5.5) and transformed images (Fig. 5.6, 5.9 and Fig. 5.10) were calculated to get the hash simulation results, see Appendix B for the results. The changes done in the image $I_{11}$ include:

$I_{11}$: Original Image

$I_{12}$: Embedded with code – “google it is”

$I_{13}$: Embedded with code - “google it is” and key 1

$I_{14}$: Only Key 1

\(^3\) http://devfarm.it/steganography/
Fig. 5.4 Google Bird Image - I11

Fig. 5.5 Mona Lisa

Fig. 5.6 Transformed Images – $I_{12}, I_{13}, I_{14}$
The difference in the hash codes generated for $I_1$ to $I_4$ set of images is shown in Fig. 5.7. It is observed from the results, that the change in the LSB for each pixel leads to a complete change in the hash code.

The next set of simulations was performed on image $I_{21}$, with following conditional changes:

$I_{21}$: Original Image

$I_{22}$: Embedded with text – “My research work”

$I_{23}$: Embedded with text – “My research work” and key 24

$I_{24}$: Embedded with a text file

$I_{25}$: Embedded with another image file (Fig. 5.8)

$I_{26}$: Embedded with another image file (Fig. 5.8) and key 3
Fig. 5.8 Embedded Image

Fig. 5.9 Transformed Images $I_{22}, I_{23}$

Fig. 5.10 Transformed Images $I_{24}, I_{25}, I_{26}$
Figure 5.9 and Fig. 5.10 shows the transformed images after the manipulations $I_{22}$ to $I_{26}$ done $I_{21}$. The difference in the hash codes generated for $I_{21}$ to $I_{26}$ set of images is shown in Fig. 5.11. It is again observed that the change in single pixel of the image leads to a complete change in the original hash code.

The results of randomness tests performed on text and image files show that the proposed framework is sensitive for the inputs provided, whether in text form or in pixel form. Therefore, any kind of alteration in the input file gives different hash code. It is also observed that, the hash codes generated for each simulation provide totally random results, irrespective of original results. Hence, the framework qualifies the randomness and sensitivity preservation feature to make it suitable for acceptance in DF.
5.5.2 Performance Test

After the randomness test, the next requirement in forensic investigation is the performance analysis of the proposed framework. Tests are performed to check the speedup in the process of generating image of a file/folder/disk for the forensic process. The files used as test cases are random of varied size taken from the t5 Corpus Suite comprised of text, image and compressed folders. These files are stored on hard disk, pen-drive and the hash code results on both the devices are compared for its correctness and efficiency. All the tests are compiled using the same compiler and configuration settings. For the optimality, the compiler flags –g0 to disable debugging and –O3 to enable third level of optimization were used. The gettimeofday() function of Linux has been used in order to measure the execution time.

5.5.2.a Performance of Parallel Vs Serial

The first performance test is done on text files with execution time of the RSHA-1 algorithm compared with that of standard SHA-1. Files of sizes 512MB, 1GB, 2GB, and 4GB are taken for the purpose of analysis. Table 5.1 shows the execution results of both the algorithms on Setup I (discussed in Chapter 2, Section 2.3). A constant speedup of approximately 7X and 85% of improvement can be observed for the file size $2^x$ MB ($9 \leq x \leq 12$) in column 5 of Table 5.2. Consistency in speedup is observed due to the fixed number of processors (in this experiment it is 8), if the number of processors is increased or decreased, variation in speedup can also be observed (as proved in Section 3.3.5.b). Figure 5.12 illustrates the effect of using the EITRH class based algorithm RSHA-1 on the execution time.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA-1</td>
<td>RSHA-1</td>
</tr>
<tr>
<td>1</td>
<td>512MB</td>
<td>3.91528</td>
<td>0.55522</td>
</tr>
<tr>
<td>2</td>
<td>1GB</td>
<td>6.89237</td>
<td>0.97293</td>
</tr>
<tr>
<td>3</td>
<td>2GB</td>
<td>13.72953</td>
<td>1.93942</td>
</tr>
<tr>
<td>4</td>
<td>4GB</td>
<td>26.1999</td>
<td>3.6988</td>
</tr>
</tbody>
</table>
Fig. 5.12 Performance of RSHA-1 and SHA-1 for Different File Sizes

Fig. 5.13 Run Time Efficiency of RSHA-1 (EITRH) on Different Number of Cores with Different File Sizes
Table 5.2 Experimental Results of RSHA-1 (EITRH) on Different Number of Cores with Different File Sizes

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>RSHA-1 on different number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>512MB</td>
<td>3.91528</td>
<td>2.78188</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.407422</td>
<td>2.004167</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>1.407422</td>
<td>1.002083</td>
</tr>
<tr>
<td>2</td>
<td>1GB</td>
<td>6.89237</td>
<td>5.02887</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.37056</td>
<td>1.968669</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>1.37056</td>
<td>0.984335</td>
</tr>
<tr>
<td>3</td>
<td>2GB</td>
<td>13.72953</td>
<td>10.14322</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.353567</td>
<td>1.965112</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>1.353567</td>
<td>0.982556</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.295165</td>
<td>1.837059</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>1.295165</td>
<td>0.91853</td>
</tr>
</tbody>
</table>

From Table 5.2 and Fig. 5.13 it is observed that, as the number of processing elements are increased the execution time decreases for each file size. In an environment with eight processing cores, approximately 7X of speedup has been achieved as compared to 1.5X on single cored execution along with good efficiency when compared with time taken by standard SHA-1.

5.5.2.b Performance on Compressed Data

In order to make the algorithm feasible for forensics applications, the experiments are also performed on compressed files of data 2GB, 6GB, 8GB, 16GB, and 20GB on the same dual quad core AMD Bulldozer machine used in previous experiments. Readings of execution time for both SHA-1 and RSHA-1 for different file sizes on 8 cores are shown in Table 5.3. Table 5.4 shows the results of execution of RSHA-1 on different processing cores along with the gain in speedup and efficiency.
Table 5.3 Execution Time Taken by SHA-1 and RSHA-1 on Compressed Data

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA-1</td>
<td>RSHA-1</td>
</tr>
<tr>
<td>1</td>
<td>2GB</td>
<td>0.49378</td>
<td>0.09268</td>
</tr>
<tr>
<td>2</td>
<td>6GB</td>
<td>4.86723</td>
<td>0.9074</td>
</tr>
<tr>
<td>3</td>
<td>8GB</td>
<td>5.06786</td>
<td>0.92659</td>
</tr>
<tr>
<td>4</td>
<td>16GB</td>
<td>8.19474</td>
<td>1.48378</td>
</tr>
<tr>
<td>5</td>
<td>20GB</td>
<td>15.99908</td>
<td>2.8581</td>
</tr>
</tbody>
</table>

The test results demonstrate the improvement of using RSHA-1 compared with standard SHA-1 for larger file sizes in Fig. 5.14. A speedup of approximately 5X has been observed for parallel execution of the program with data in compressed form.
### Table 5.4 Performance of RSHA-1 on Multiple Cores

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>1-Core</th>
<th>2-Core</th>
<th>4-Core</th>
<th>6-Core</th>
<th>8-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2GB</td>
<td>0.49378</td>
<td>0.41921</td>
<td>0.30376</td>
<td>0.15874</td>
<td>0.11477</td>
<td>0.09268</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.17788</td>
<td>1.62556</td>
<td>3.11062</td>
<td>4.30234</td>
<td>5.32779</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Efficiency</td>
<td>1.17788</td>
<td>0.81278</td>
<td>0.77765</td>
<td>0.71705</td>
<td>0.66597</td>
</tr>
<tr>
<td>2</td>
<td>6GB</td>
<td>4.86723</td>
<td>3.20755</td>
<td>2.32752</td>
<td>1.18793</td>
<td>0.8098</td>
<td>0.9074</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.51742</td>
<td>2.09116</td>
<td>4.09723</td>
<td>6.01041</td>
<td>5.36393</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Efficiency</td>
<td>1.51742</td>
<td>1.04558</td>
<td>1.02430</td>
<td>1.00173</td>
<td>0.67049</td>
</tr>
<tr>
<td>3</td>
<td>8GB</td>
<td>5.06786</td>
<td>3.9894</td>
<td>2.80742</td>
<td>1.41825</td>
<td>0.95515</td>
<td>0.92659</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.27033</td>
<td>1.80516</td>
<td>3.57331</td>
<td>5.30582</td>
<td>5.46936</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Efficiency</td>
<td>1.27033</td>
<td>0.90258</td>
<td>0.89333</td>
<td>0.88430</td>
<td>0.68367</td>
</tr>
<tr>
<td>4</td>
<td>16GB</td>
<td>8.19474</td>
<td>7.93936</td>
<td>5.66733</td>
<td>2.79672</td>
<td>1.90733</td>
<td>1.48378</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.03216</td>
<td>1.44596</td>
<td>2.93012</td>
<td>4.29644</td>
<td>5.52288</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Efficiency</td>
<td>1.03216</td>
<td>0.72298</td>
<td>0.73253</td>
<td>0.71607</td>
<td>0.69036</td>
</tr>
<tr>
<td>5</td>
<td>20GB</td>
<td>15.9990</td>
<td>15.1581</td>
<td>10.5924</td>
<td>5.53135</td>
<td>3.6348</td>
<td>2.8581</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.05548</td>
<td>1.51042</td>
<td>2.89243</td>
<td>4.40164</td>
<td>5.59780</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Efficiency</td>
<td>1.05548</td>
<td>0.75521</td>
<td>0.72310</td>
<td>0.73360</td>
<td>0.69972</td>
</tr>
</tbody>
</table>

#### 5.5.2.c Impact on a Forensic Investigation

Based on above findings, we then analyzed the performance of EITRH transformation in real world forensic scenarios. But right now, the biggest constraint for not using EITRH directly for larger file sizes is its buffer size limitation. Therefore, we took our own two work stations with different data size and used the results from Table 5.1 to do a projection. The results calculated are given in Table 5.1. The estimated results for data size 98 GB and 150.25 GB are calculated by multiplying new data size with the results of 1GB given in Table 5.1 for both SHA-1 and RSHA-1. Since the algorithm works on the data in a block-by-block fashion, this is a reasonable assumption.

### Table 5.5 Estimated Runtime for a Sample Use Case

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Data Size</th>
<th>SHA-1</th>
<th>RSHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu 14.4</td>
<td>98 GB</td>
<td>11min 25sec</td>
<td>1min 58 sec</td>
</tr>
<tr>
<td>Windows 8</td>
<td>150.25 GB</td>
<td>17min 25 sec</td>
<td>2min 43 sec</td>
</tr>
</tbody>
</table>
Hence, from the projected results, it can be observed that RSHA-1 can outperform in real world scenarios as well. These results are for maintaining integrity and authentication in the forensic investigation but for similarity preservation EITRH can prove to be more useful.

**Conclusion**

In this chapter we related the application of hash functions from Chapter 3 to forensic authentication. We presented randomness tests and performance tests which proved that RSHA-1 can give good results when producing hash code for large files making it appropriate for data similarity detection, data integrity, and authentication. The speedup and efficiency of the algorithm has also been measured and an improvement of approximately 85% has been observed as compared with existing hashing algorithms. This has proved EITRH to be a good option for use in forensic applications during the process of evidence acquisition and is authentic to present in front of the Jury or Court of Law. It is an open problem to embed the algorithm in a forensic investigation tool for our scheme of Section 5.4 in order to construct an authentic tool.

The future course of action will include experimenting the code on even larger files typically a hard disk of size in tera-bytes to fully utilize it in the digital forensics context. We expect the algorithm to work very well there given the reasonable computational complexity of the algorithm.
CHAPTER 6
EITRH IN DIGITAL SIGNATURES
Chapter 6

EITRH IN DIGITAL SIGNATURES

Cryptographic hash functions are used in a range of applications including the digital signature. Digital signing of a document uses hash functions with a purpose of maintaining the authenticity. This chapter discusses the use of hash functions in generating digital signatures followed by the use of EITRH algorithm in parallel generation of the signatures. The proposed digital signature algorithm comprises of two techniques including parallel RSA as well as the previously discussed RSHA-1 algorithm.

6.1 Introduction

Digital Signature Standard (DSS) was first published in 1994 by NIST as FIPS PUB 186 (Mehuron, 1994) to provide a means of guaranteeing authenticity and integrity in the digital world. The main motive of developing digital signatures was to detect illegal alterations in data and to validate identity of the signatory. In addition, assurance of nonrepudiation can also be achieved through them. This property restricts the signatory from denying the signatures and allows the recipient to use it as an evidence to prove to the third party that the signature was really generated by the sender only. FIPS 186-4 (Gallagher and Kerry, 2013) is the latest revision of DSS published by NIST with some minor changes. Due to the ever growing technology advancements, updates to the standard are still going on.

The standard is envisioned for use in e-mail, e-funds transfer, e-data interchange, software distribution, data storage, and many other applications that require assurance of integrity of electronic documents and authenticity of the sender. It uses complex mathematical operations to encrypt and decrypt the “signatures” to make them impossible to forge.
In Section 6.2, we will discuss in brief about the functioning of digital signatures and propose a new parallel scheme for generating signatures in Section 6.3, followed by its experimental results in Section 6.4. The application of EITRH transformation in digital signatures is discussed in this chapter to demonstrate the power of parallelization on multi-core processors and is also analyzed for performance improvements over sequential implementations.

### 6.2 Digital Signatures

A digital signature is used to validate the authenticity and integrity of a digital message or a file. The signatures helps in uniquely identifying the signatory and also ensures non-repudiation. A valid digital signature ensures a receiver that the message was indeed created by a verified sender only. It is represented in a computer as a string of bits and can be generated on both stored and transmitted data.

Digital signature generation process makes use of asymmetric cryptography in which the signatory possesses a pair of private and public keys. Public keys are made available to the public and private keys are kept secret by the sender. Thereafter, the sender uses a FIPS approved cryptographic hash function to obtain a message digest and then applies a private key on it to generate a digital signature.

The generated signature is then provided to the intended receiver along with the data. The receiver verifies the signature by using the sender’s public key along with the same hash function which was used to generate the signature. These signatures can be verified by anyone by employing the signatory’s public key but can only be generated by the user who possesses the private key. In this form the scheme ensures, authentication, non-repudiation, and data integrity of the digital data. Digital signatures follows the rule of WYSIWYS (What You See Is What You Sign).
A digital signature scheme can more formally be defined as given in Definition 6.1.

**Definition 6.1:** A digital signature scheme is typically a three step process composed of the following three polynomial time algorithms:

- **Key pair generation algorithm** \( (kp) \): For an input \( 1^u \) (where \( u \) is a security parameter), the algorithm \( kp \) outputs a pair of public and private keys \( (pr, pu) \) for the encryption and decryption purpose. Algorithm \( kp \) is probabilistic.
- **Signing algorithm** \( (Sig) \): Given a message \( M \), and a public and private key pair \( (pr, pu) \), the algorithm \( Sig \) produces a signature \( \alpha \).
- **Verification algorithm** \( (Ver) \): Given a signature \( \alpha \), a message \( M \) and a public key \( pu \), the algorithm \( Ver \) either accepts the signatory's claim to authenticity (if \( \alpha \) is a valid signature of \( M \) with respect to \( pu \)) or rejects.

DSS defines methods for digital signature generation that can be used for the security of digital data, and for the verification and validation of those signatures. Primarily three techniques are approved to generate digital signatures (Gallagher and Kerry, 2013):

1. Digital Signature Algorithm (DSA): specified in FIPS PUB 186-4 Standard.

In order to execute our research work, the RSA digital signature algorithm has been used. The reason for using RSA digital signature algorithm and not DSA is not specific but can be correlated to lots of criticism that exist against the DSA. Moreover, DSA’s signature verification process is slower than that of RSA’s, which makes it less acceptable (Schneier, 2007).
6.2.1 RSA Digital Signatures

The RSA digital signature algorithm has been approved by FIPS 186-4 for the use of its implementations in the digital signature generation process. The algorithm is widely used by many of the certifying authorities to generate/verify the signatures. The detailed description of the process of digital signing and digital verification is illustrated in Fig. 6.1.

Fig. 6.1 RSA Digital Signature Generation and Verification Process

An RSA digital signature scheme is again a composition of three algorithms: key generation algorithm, signature generation algorithm and signature verification algorithm. The key generation algorithm and signature generation algorithm are used by the signing authority to generate the pair of keys and digitally sign the document, whereas signature verification algorithm is used by the receiver to verify and validate the signatures. The Sub-sections 6.2.1.a, 6.2.1.b and 6.2.1.c briefly explain these algorithms, respectively.
6.2.1.a Key Generation

The signatory first of all uses key generation algorithm to generate a pair of keys consisting of a private and a public key. Private key is retained by the signatory and is used to compute a digital signature, whereas public key is shared with the recipients to verify the digital signature. The key generation process of RSA algorithm involves following steps:

1. Choose two large random prime integers: \( q \) and \( r \).
2. Calculate \( m = q \times r \), which is used as a modulus for the modular reduction part.
3. Calculate \( \phi(m) = (q - 1) \times (r - 1) \).
4. Choose an integer \( pu \), such that: \( \text{GCD}(pu, \phi(m)) = 1 \) and \( 1 < pu < \phi(m) \).
5. Calculate \( pr \), such that: \( \text{mod}(pr \times pu, \phi(m)) \equiv 1 \) and \( 1 < pr < \phi(m) \).

An RSA private key consists of a modulus \( m \) and a private exponent \( pr \). Thus, RSA private key is a pair of values \( (m, pr) \) and is used to generate digital signatures. The corresponding RSA public key consists of the same modulus \( m \) and a public key exponent \( pu \). Thus, the RSA public key is a pair of values \( (m, pu) \) and is used to verify and validate the digital signatures.

6.2.1.b Digital Signature Generation

For a given message \( M \), the digital signatures are generated by the signatory by producing the Message Digest (\( MD \)) for a message \( (M) \) using SHA-1 algorithm (as specified in FIPS 180).

Then, the private key \( (m, pr) \) generated by RSA algorithm is used to compute the signature:

\[
S = (MD)^{pr} \mod m
\]

The message \( (M) \) along with signature \( (S) \) is sent to the receiver.
6.2.1.c Digital Signature Verification and Validation

The receiver receives the message along with the signatures. The next step is to verify the signatures to ensure the authenticity and integrity of the message received. The receiver uses the sender’s public key \((m, pu)\) generated by RSA to extract the hash value from the message received:

\[
V = (S)^{pu} \mod m
\]

The \(V\) obtained from the algorithm is the hash value 1 generated by the sender. The hash value 2 is generated by applying SHA-1 algorithm on the message received. Both the hash values 1 and 2 are compared; if the values are identical, then the signature is considered to be valid otherwise it is treated as invalid.

In RSA digital signature scheme, the security of the signatures increases with the increase in key size which is directly proportional to the execution time. The size of the data or message to be signed digitally is increasing exponentially. Signature generation/verification time is also directly affected with the increase in the size of data. Therefore, in order to overcome the time constraints in the digital signature generation/verification process, we have proposed parallelization of the whole process in Section 6.3 and discussed the outcomes in Section 6.4.

6.3 Parallel Digital Signature

As discussed in Section 6.2.1, the RSA digital signature algorithm uses both the RSA and the hashing algorithm together to generate the digital signatures. Therefore, in order to parallelize the digital signature process, two approaches are proposed:

1. RSA with parallel hashing
2. Parallel RSA with parallel hashing
6.3.1 RSA Digital Signature with Parallel Hashing

In this section, the hashing process in RSA digital signature algorithm (discussed in Section 6.2.1) is tried to parallelize by making use of EITRH transformation based algorithm RSHA-1 (discussed in Chapter 3). The RSHA-1 algorithm has been used for generating message digest in signing and verification algorithms. Algorithms 6.1 and 6.2 show the digital signature generation and verification processes, respectively, by making use of RSHA-1 algorithm for generating the hash code in parallel.

Algorithm 6.1 shows the digital signature generation process in which only the hash code generation part has been parallelized. In Algorithm 6.2, validation of the signatures is also done by generating the hash code of the message received in parallel. In order to check the influence of parallelization, the algorithm is implemented in OpenMP and checked for performance gain. The experimental results of the implementation are provided in Section 6.4.1 of this chapter.

**Procedure:** Signature_Generation

**Model:** Data Parallel Model with P processors [where P=2, 4, 6, 8]

**Input:** Message, private_key, modulus

**Output:** Digitally Signed Message

**Declare:** N, Number_of_Cores, MD, Vector Str, Result := 1

- Set Number_Of_Cores = P
- Read Message to Vector Str
- Set N := Length of Str
- Declare No_Of_Iterations := N / Block_Size

**Parbegin**
- Apply RSHA-1 to Vector Str and Generate MD

**Parend**

// Generate Signature by applying private key on message

**For ALL** i: [1, private_key]
- Set Result := Result * MD

**EndFor**

Set Signature := Result mod Modulus

Send Signature || Message to receiver

Algorithm 6.1 RSA Digital Signature Generation with Parallel Hashing
**Procedure:** Signature Verification

**Model:** Data Parallel Model with P processors [where P=2, 4, 6, 8]

**Input:** Digitally Signed Message, public_key, modulus

**Output:** Authenticated Message

**Declare:** N, Number_of_Cores, MD1, MD2 Vector Str, Result := 1

- Set Number_Of_Cores = P
- Extract Message to Vector Str and Signature to Sig_G
- Set N := Length of Str
- Declare No_Of_Iterations := N / Block_Size

**Parbegin**

- Apply RSHA-1 to Vector Str and Generate MD1

**Parend**

// Generate Message Digest by applying public key on Sig_G

**For ALL** i: [1, public_key]
- Set Result := Result * Sig_G

**EndFor**

- Set MD2 := Result mod Modulus
- Compare both the signatures
  - if MD1 equals MD2
    - Message Accepted
  - else
    - Message Rejected

**End if**

Algorithm 6.2 RSA Digital Signature Verification and Validation with Parallel Hashing

### 6.3.2 RSA Digital Signature with Parallel RSA and Parallel Hashing

After parallelizing the hashing process in RSA digital signature scheme, it was observed that the benefit of parallelization can be taken for the complete process. Therefore, for the complete parallel digital signature generation or verification process, two different algorithms were used:

- **RSHA-1** (Kishore and Kapoor, 2014a): to generate hash code
- **PRSA** (Saxena et al., 2013, Saxena and Kapoor, 2014): parallel RSA to perform digital signing as well as signature verification process
In the parallel RSA (PRSA) algorithm, the authors have proposed to use modular exponentiation technique to generate a pair of keys, private key ($pr$) and a public key ($pu$). They have shown the benefit of using the technique with GNU's GMP library for fast integer calculation and OpenMP for parallelization. Therefore, making the use of benefit provided both RSHA-1 and PRSA algorithm were used together to parallelize the whole process. The process of digital signature generation and verification and validation in parallel are shown in Algorithm 6.3 and 6.4 respectively.

**Procedure: Signature Generation**

**Model:** Data Parallel Model with P processors [where P=2, 4, 6, 8]

**Input:** Message, private_key, modulus

**Output:** Digitally Signed Message

**Declare:** N, Number_of_Cores, MD, Vector Str, Result := 1

Set Number_Of_Cores = P
Read Message to Vector Str
Set N := Length of Str
Declare No_Of_Iterations := N / Block_Size

Parbegin

Apply RSHA-1 to Vector Str and Generate MD

Parend

// Generate Signature by applying private key on message
N= private_key /Number_Of_Cores

Parbegin

if private_key mod Number_of_Threads := 0

For ALL i: [1, private_key] IN SYNC

For ALL j: [1, N]

Set Result := Result * MD

EndFor

EndFor

else

Set N := N - 1

For ALL i: [1, private_key] IN SYNC

For ALL j: [1, N]

Set Result := Result * MD

EndFor

EndFor

Set Result := Result * MD;
end if

Set Signature := Result mod Modulus

Parend

Send Signature || Message to receiver

Algorithm 6.3 Parallel RSA Digital Signature Generation
**Procedure:** Signature Verification

**Model:** Data Parallel Model with P processors [where \( P = 2, 4, 6, 8 \)]

**Input:** Digitally Signed Message, public_key, modulus

**Output:** Authenticated Message

**Declare:** N, Number_of_Cores, MD1, MD2 Vector Str, Result := 1

Set Number_Of_Cores = P

Extract Message to Vector Str and Signature to Sig_G

Set N := Length of Str

Declare No_Of_Iterations := N / Block_Size

**Parbegin**

Apply RSHA-1 to Vector Str and Generate MD1

**Parend**

// Generate Message Digest by applying public key on Sig_G

\( N = \text{private_key} / \text{Number_Of_Cores} \)

**Parbegin**

if public_key mod Number_of_Threads := 0

**For ALL** i: \([1, \text{public_key}]\) IN SYNC

**For ALL** j: \([1, \text{N}]\)

Set Result := Result * Sig_G

**EndFor**

**EndFor**

else

Set N := N - 1

**For ALL** i: \([1, \text{public_key}]\) IN SYNC

**For ALL** j: \([1, \text{N}]\)

Set Result := Result * Sig_G

**EndFor**

**EndFor**

Set Result := Result * Sig_G;

end if

Set MD2 := Result mod Modulus

**Parend**

Compare both the signatures

if MD1 equals MD2

Message Accepted

else

Message Rejected

End if

Algorithm 6.4 Parallel RSA Digital Signature Verification and Validation
Algorithm 6.3 shows the digital signature generation process in which both the hash code generation part and its encryption for signature generation has been parallelized. In Algorithm 6.4, validation of the signatures is also done by decrypting and then generating the hash code of the message received in parallel. In order to check the influence of parallelization, the algorithm is implemented in OpenMP and checked for performance gain. The experimental results of the implementation are provided in Section 6.4.2 of this chapter.

6.4 Experimental Results

In order to analyze the performance gained by the algorithms discussed in Sections 6.3.1 and 6.3.2, the experiments were performed on Setup I (as described in Chapter 2, Section 2.3) in the presence of GNU’s MP Library on Linux environment. GNU’s MP Library is used for modular exponentiation in RSA algorithm along with the use of OpenMP API for parallelization. Sections 6.4.1 and 6.4.2 shows the experimental results of the developed algorithms on a developed test case.

The performance of the algorithms is analyzed in terms of the time spent during key generation, signature generation, and signature verification routines of digital signing and verification and validation process using `gettimeofday()` function of Linux. The experiments were executed a number of times and the average time was taken out for the final results.

6.4.1 Results Analysis for RSA Digital Signature with Parallel Hashing

The test cases included experiments with random files on the developed parallel algorithms. Different files of size 256MB, 512MB, 1GB, 2GB and 4GB were used and the process of signature generation and signature verification and validation was performed on them. The motive of taking different size files was to check the performance of algorithm on small to large sized files. The key size was set to 2048 bits (as specified by NIST) for all the experiments and
execution time for key generation, signature generation and signature verification and validation was measured. For the key size 2048 bits, key generation time was observed to be approximately 0.1319 seconds for all the experiments irrespective of the file size and algorithm used (serial or parallel).

The results of OpenMP implementation of Algorithm 6.1 and 6.2 in the form of performance gains are given in Table 6.1 and 6.2 for the purpose of analysis. Table 6.1 lists the execution time of signature generation process for both serial and parallel code. The signature generation time includes the time for generating message digest and encryption of the digest generated with signer’s private key. Table 6.2 lists the execution time of signature verification and validation process for both serial and parallel code. The signature verification and validation time includes the time for extracting the message and message digest from the message received by decrypting it with signer’s public key. Further it includes time for the generation of hash digest of the message extracted and comparing it with the digest extracted for the purpose of verification of the signatures.

It can be inferred from both the tables that parallelizing the hash generation process can give good results in the form of reduced execution time for digital signature scheme. A speedup of approximately 4X to 6X has been observed in the case of digital signature generation and approximately 5X to 6X has been observed in digital signature verification and validation process for different file sizes.

Table 6.1 Execution Results of Signature Generation for Parallel Hashing Algorithm

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Serial Code</td>
</tr>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.78467</td>
</tr>
<tr>
<td>2</td>
<td>512MB</td>
<td>3.39826</td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.36049</td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.85867</td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.74468</td>
</tr>
</tbody>
</table>
Table 6.2 Execution Results of Signature Verification and Validation for Parallel Hashing Algorithm

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Serial Code</strong></td>
</tr>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.57292</td>
</tr>
<tr>
<td>2</td>
<td>512MB</td>
<td>3.1857</td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.14847</td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.65558</td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.03254</td>
</tr>
</tbody>
</table>

Table 6.3 Execution Results of Signature Generation for Parallel Hashing Algorithm on Multiple Cores

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Serial</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.78467</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.535795</td>
</tr>
<tr>
<td>2</td>
<td>512MB</td>
<td>3.39826</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.627784</td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.36049</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.645329</td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.85867</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.876573</td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.74468</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.700747</td>
</tr>
</tbody>
</table>

Table 6.4 Execution Results of Signature Verification and Validation for Parallel Hashing Algorithm on Multiple Cores

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Serial</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.57292</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.695834</td>
</tr>
<tr>
<td>2</td>
<td>512MB</td>
<td>3.1857</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.704312</td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.14847</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.690506</td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.65558</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.90696</td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.03254</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.677202</td>
</tr>
</tbody>
</table>
Tables 6.3 and 6.4 lists the results of execution time and speedup for signature generation and verification algorithms respectively. The codes are executed on 1, 2, 4, 6, and 8 number of cores on Setup I (as discussed in Chapter 2, Section 2.3) by enabling/disabling the number of cores available on the processor. Depending upon the number of cores, the number of threads are decided. It can be inferred from the tables that the execution time for an input file is the function of number of cores employed to complete the task. For each file, adding a pair of processing cores approximately halves the execution time and thus increases the speedup.

### 6.4.2 Results Analysis for RSA Digital Signature with Parallel RSA and Parallel Hashing

As discussed in Section 6.3.2, both the algorithms (RSHA-1 and RSA) are implemented in parallel so as to gain the maximum performance as compared with normal digital signature scheme. Same test case was considered as used in Section 6.4.1 for the experiments. The algorithm was implemented in C with OpenMP API directives for parallelization. In parallelization of complete process of digital signature generation and verification, GNUs multiple precision arithmetic library (GMP) is used. As, GMP is considered to be faster big-numbers library than other libraries, therefore it can fasten up the calculations for parallel computation of modular exponentiation and modular reduction. The OpenMP implementation of the Algorithm 6.3 and 6.4 is given in Appendix D for the reader’s reference and its implementation results in Table 6.5 and 6.6 for the purpose of analysis.

Table 6.5 lists the execution time of signature generation process for both serial and parallel code executed for different file sizes. The signature generation time includes the time for hash digest generation of the message and encryption of the digest generated with signer’s private key. Table 6.6 lists the execution time of signature verification and validation process for both serial and parallel code for different file sizes. The signature verification and validation time includes
the time for extracting the message and hash digest from the message received by decrypting it with signer’s public key. It also includes the time for generation of hash digest of the message extracted and comparing it with the digest extracted for the verification of the signatures.

It can be inferred from both the tables that parallelizing the whole process gives better results in the form of reduced execution time for digital signature scheme. A speedup of approximately 6X to 6.5X has been observed in the case of digital signature generation as well as digital signature verification and validation process for different file sizes.

Table 6.5 Execution Results of Signature Generation for Parallel RSA and RSHA-1

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>Serial Code</th>
<th>Parallel Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.78467</td>
<td>0.30728</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>512 MB</td>
<td>3.39826</td>
<td>0.5524</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.36049</td>
<td>1.00198</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.85867</td>
<td>2.06017</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.74468</td>
<td>3.62737</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6 Execution Results of Signature Verification and Validation for Parallel RSA and RSHA-1

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>Serial Code</th>
<th>Parallel Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.57292</td>
<td>0.25878</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>512 MB</td>
<td>3.1857</td>
<td>0.504625</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.14847</td>
<td>0.96525</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.65558</td>
<td>1.93926</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.03254</td>
<td>3.63154</td>
<td></td>
</tr>
</tbody>
</table>

Tables 6.7 and 6.8 list the results of execution time and speedup for signature generation and verification algorithms respectively on multiple processing cores. It can be inferred from the tables that by parallelizing the whole process and executing the code on a number of cores makes a big difference. By adding a pair of processing cores increases the speedup by a factor of approximately 1.5X.
Table 6.7 Execution Results of Signature Generation for Parallel RSA and RSHA-1 on Multiple Cores

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>Serial</th>
<th>Number of processing cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.78467</td>
<td>1.08153</td>
<td>0.5505</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.650135</td>
<td>3.241907</td>
<td>4.927714</td>
</tr>
<tr>
<td>2</td>
<td>512MB</td>
<td>3.39826</td>
<td>1.9899</td>
<td>0.99883</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.707754</td>
<td>3.402241</td>
<td>5.066434</td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.36049</td>
<td>3.76985</td>
<td>1.88787</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.6872</td>
<td>3.367512</td>
<td>4.975157</td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.85867</td>
<td>7.72255</td>
<td>3.76329</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.665081</td>
<td>3.416869</td>
<td>5.144373</td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.74468</td>
<td>7.58372</td>
<td>7.0021</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>3.131007</td>
<td>3.39108</td>
<td>4.965169</td>
</tr>
</tbody>
</table>

Table 6.8 Execution Results of Signature Verification and Validation for Parallel RSA and RSHA-1 on Multiple Cores

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>Serial</th>
<th>Number of processing cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>256MB</td>
<td>1.57292</td>
<td>0.93929</td>
<td>0.48623</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.674584</td>
<td>3.23493</td>
<td>4.897316</td>
</tr>
<tr>
<td>2</td>
<td>512MB</td>
<td>3.1857</td>
<td>1.86733</td>
<td>0.93446</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.706019</td>
<td>3.409135</td>
<td>5.058112</td>
</tr>
<tr>
<td>3</td>
<td>1GB</td>
<td>6.14847</td>
<td>3.64739</td>
<td>1.82721</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.685718</td>
<td>3.36495</td>
<td>4.968581</td>
</tr>
<tr>
<td>4</td>
<td>2GB</td>
<td>12.65558</td>
<td>7.4927</td>
<td>3.70751</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.689055</td>
<td>3.413499</td>
<td>5.147453</td>
</tr>
<tr>
<td>5</td>
<td>4GB</td>
<td>23.03254</td>
<td>7.70397</td>
<td>6.97661</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>2.989698</td>
<td>3.301394</td>
<td>4.85815</td>
</tr>
</tbody>
</table>

Figure 6.2 shows a comparative analysis of digital signing and verification process in the form of serial, parallel hash generation and overall parallel algorithm. The impact of parallelization can be seen in the form of reduced execution time on multi-core processors. It is observed from the above results that for 2048 bits key size, parallelization of signing and verifying signatures in RSA results in a significant speedup for the whole process.
Conclusion

In this chapter we looked at the applications of hash functions to the task of digital signing. The chapter presents a way of speeding up the process of digital signature generation and verification for large message sizes. Files of large size which needs to ensure security in the form of authentication, non-repudiation, and integrity can be sent by appending digital signatures created using the EITRH algorithm for a much improved performance of the process. We presented two algorithms for RSA digital signature scheme: one with parallel hash and another with both parallel hash and parallel RSA.

Experimental results show that RSA digital signature algorithm based on EITRH framework gives 6X to 6.5X of speedup on multi-core processors for large file sizes. As expected, even better results are observed when parallelizing the complete process rather than just the hashing part.
CHAPTER 7
EITRH ON GPUS
Chapter 7

EITRH ON GPUS

In Chapter 3, we have proved the efficacy of EITRH transformation in terms of its performance, efficiency and cost optimality on multi-core CPUs. Graphics Processing Units (GPUs) offer a great architecture to take advantage of data parallelism very effectively. In this chapter, we implement and discuss the impact of executing RSHA-1 algorithm on GPUs. Our results are based on primary research and are not specific to implementation technicalities in GPU. The main motive of the implementation of the code was to study the effect of parallel RSHA-1 working on the GPUs.

7.1 Introduction to GPU

nVIDIA’s Graphics Processing Unit (GPU) (Nickolls and Dally, 2010) accelerated computing is a graphics and video acceleration technology based on parallel computing. It uses a GPU together with a CPU to accelerate scientific, engineering, and enterprise applications. In year 2007, nVIDIA introduced GPUs, which have now become power and energy-efficient computer mechanism in government labs, universities, enterprises, and small- and-medium businesses around the world.

They have driven away the voracious demand of highly parallel, multi-threaded, high-definition 3D graphics by providing tremendous computational horsepower and high memory bandwidth. Unprecedented performance has been observed in GPU accelerated computing applications. In these applications, compute intensive parallel portions of the application are executed on the GPU, while the remaining code runs on the CPU. This type of task allocation procedure gives the application a great performance gains. GPU has a SIMD architecture, which is best suited for data parallel applications. The details regarding GPUs can be found at (Lindholm et al., 2008, Nickolls and Dally, 2010).
Hashing algorithms are also compute intensive and involve lots of arithmetic operations in generating hash code for files. The serial nature of prevalent hashing algorithms restricts its proper execution on GPUs. As RSHA-1 is a parallel hashing algorithm, therefore it can be implemented on such a GPU enabled machine. Next, we discuss the use of CUDA software development on GPUs followed by the details of the implementation of RSHA-1 algorithm. The experimental results from this implementation are discussed in Section 7.3.

7.1.1 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) (Sanders and Kandrot, 2010) is a software development kit created by nVIDIA to use their graphic devices for graphical as well as non-graphical purposes. Its strength is based on the divide and conquer technique, in which lots of threads are created for parallel execution over all the cores of a device. After performing the required calculations in each thread the result is unified.

A GPU is built around a vector of Streaming Multiprocessors (SMs). CUDA arranges a parallel computation using the generalization of threads, blocks and grids. These multiple blocks are organized into a one-dimensional or two-dimensional grid of thread blocks that execute independently from each other, as illustrated in Fig. 7.1.

The CUDA programming model allows programmers to use C as a high-level programming language. It supports various languages, APIs, and directive based approaches, such as FORTRAN, C++, DirectCompute, OpenACC. GPU memory architecture supports a small (64KB in the Tesla C2075) but fast shared memory, along with four other memory modules: global, register, texture, and constant. All the memories have different size, restricted access policies and access latencies.
7.1.2 GPU Processing Phases

The performance of an application on a GPU depends upon various phases it has to go through. An application has to pass through five main phases: pre-processing, host-to-device data transfer, processing, device-to-host results transfer, and post-processing (here host refers to CPU and device refers to GPU). Table 7.1 describes these phases and its functionality.
Table 7.1 Phases of Application Processing and Performance Factors

<table>
<thead>
<tr>
<th>Phase No.</th>
<th>Phase</th>
<th>Sub-phases</th>
<th>Operations performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pre-</td>
<td>Device initialization (T\textsubscript{D,Init})</td>
<td>Device initialization</td>
</tr>
<tr>
<td></td>
<td>processing</td>
<td>Memory allocation (T\textsubscript{Mem,Alloc})</td>
<td>Host and the device memory allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-processing (T\textsubscript{Pr,Proc})</td>
<td>Data pre-processing on the Host</td>
</tr>
<tr>
<td>2</td>
<td>Transfer</td>
<td>Data transfer to GPU (T\textsubscript{H,D})</td>
<td>Data transfer from host memory to Device’s global memory</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>Data transfer to shared memory (T\textsubscript{G,S})</td>
<td>Data transfer from global memory to shared memories of the Device.</td>
</tr>
<tr>
<td></td>
<td>Processing</td>
<td>Processing (T\textsubscript{D,Proc})</td>
<td>Application ‘kernel’ processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data transfer to device global memory (T\textsubscript{S,G})</td>
<td>Result transfer from shared memory to global memory</td>
</tr>
<tr>
<td>4</td>
<td>Transfer</td>
<td>Output data transfer (T\textsubscript{G,H})</td>
<td>Transfer the results to the host’s memory.</td>
</tr>
<tr>
<td>5</td>
<td>Post-</td>
<td>Post-processing (T\textsubscript{Po,Proc})</td>
<td>Post processing on host resource de-allocation</td>
</tr>
<tr>
<td></td>
<td>processing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In applications with data parallelism, the processing steps are commonly repeated several times until entire input data is processed. For each iteration, data blocks are copied from global memory to each multiprocessor’s shared memory and are processed by the kernel. Once the whole data is processed, the results are then transferred back to the global memory. Therefore, the execution time of a data parallel application can be modeled as:

\[
T_{\text{Total}} = T_{\text{D,Init}} + T_{\text{Mem,Alloc}} + T_{\text{Pr,Proc}} + T_{\text{H,D}} + T_{\text{Po,Proc}}
\]

\[
= T_{\text{D,Init}} + T_{\text{Mem,Alloc}} + T_{\text{Pr,Proc}} + T_{\text{H,D}} + (D_{\text{Size}/S_{\text{Size}} \times P}) \times (T_{\text{G,S}} + T_{\text{D,Proc}} + T_{\text{S,G}}) + T_{\text{G,H}} + T_{\text{Po,Proc}}
\]

where D\_Size is the size of application’s input data set, P is the number of multiprocessors, and S\_Size is the size of shared memory of multiprocessors.

### 7.2 Implementation Details

After receiving promising results of EITRH transformation on multicore CPUs, the next target is to test its performance on GPUs. As GPUs are performing well in many of the applications, so the target was to check the adaptability of
proposed transformations in GPUs. For this purpose, the code of EITRH based algorithm RSHA-1 was converted in CUDA and implemented on a GPU (Kishore et al., 2013, Kishore and Kapoor, 2014b). The implementation was mere conversion of OpenMP directive-based code to CUDA programming paradigm without any extra optimization.

In this implementation, the code was divided into three functions: main(), calc() and innerHash(). Main() function simply accepts the input, calls calc() function for processing and outputs the hash code. In the calc() function, device variables are initialized along with the number of threads and blocks depending on the input size and copies the host data to device variables.

The function further launches the kernel innerHash() with the command:

innerHash<<<dimGrid,dimBlock>>>(device_sarray,device_hash);

Hash code for each block is generated in parallel in the kernel for dimGrid number of blocks and dimBlock number of threads and is called recursively until desired length of hash code is generated. The final results generated are copied back from device variables to host variables with the command:

cudaMemcpy(hash,device_hash,SIZE,cudaMemcpyDeviceToHost);

<table>
<thead>
<tr>
<th>Method</th>
<th>#Calls</th>
<th>grid size</th>
<th>thread block size</th>
<th>registers per thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[1992 1 1]</td>
<td>[256 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[623 1 1]</td>
<td>[256 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[195 1 1]</td>
<td>[256 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[61 1 1]</td>
<td>[256 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[19 1 1]</td>
<td>[256 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[6 1 1]</td>
<td>[256 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[2 1 1]</td>
<td>[256 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[1 1 1]</td>
<td>[168 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[1 1 1]</td>
<td>[60 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[1 1 1]</td>
<td>[28 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[1 1 1]</td>
<td>[18 1 1]</td>
<td>18</td>
</tr>
<tr>
<td>innerHash</td>
<td>1</td>
<td>[1 1 1]</td>
<td>[2 1 1]</td>
<td>18</td>
</tr>
</tbody>
</table>

Fig. 7.2 Grid, Block and Thread Distribution in Kernel
Figure 7.2 shows the blocks per grid and threads per block distribution in nVIDIA Visual Profiler (Profiler, 2011) for executing 8MB of file. The experimental results obtained on executing the code are provided in Section 7.3.

### 7.3 Experimental Results

In order to assess the proposed algorithm for CHF applications, the code was executed on a GPU machine with the following configuration:

- **Processor**: Intel Core 2 Duo CPU E7500@2.93GHz X 2
- **RAM**: 3.0 GB
- **Kernel**: Ubuntu Linux 12.04
- **Platform**: gcc 4.5.4
- **GPU**: Tesla C2075
- **API**: CUDA 5.5

The thread size and block size in the code were decided on the basis of the file size chosen. The experiment was executed on random files of size 512KB, 1MB, 2MB, 4MB and 8MB. Table 7.2 shows the execution time of SHA-1, RSHA-1 on CPU and RSHA-1 on GPU in seconds for different files. It was observed from the results that simple implementation of code on CUDA when compared with implementation on OpenMP API and with standard SHA-1 (on the same CPU) could not provide good performance gain.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA-1</td>
</tr>
<tr>
<td>1</td>
<td>512 KB</td>
<td>0.01426</td>
</tr>
<tr>
<td>2</td>
<td>1 MB</td>
<td>0.02829</td>
</tr>
<tr>
<td>3</td>
<td>2 MB</td>
<td>0.05836</td>
</tr>
<tr>
<td>4</td>
<td>3 MB</td>
<td>0.08543</td>
</tr>
<tr>
<td>5</td>
<td>4 MB</td>
<td>0.11312</td>
</tr>
<tr>
<td>6</td>
<td>8 MB</td>
<td>0.21336</td>
</tr>
</tbody>
</table>
Figure 7.3 shows that for file sizes 512 KB, 1 MB and 2 MB, implementation on GPU had rather increased the execution time. For 3MB file size, performance on CPU and GPU is almost same, but for 4 MB and 8 MB file execution time in GPU has decreased. The reason for the non-performance of GPU implementation for smaller file size was analyzed by using nVIDIA’s Visual profiler as shown in Fig. 7.4 and Fig. 7.5.

![Graph showing execution time for different file sizes on CPU and GPU](image1)

**Fig. 7.3 Performance of SHA-1 and RSHA-1 on CPU, GPU**

![Screenshot of NVIDIA Visual Profiler](image2)

**Fig. 7.4 NVIDIA’s Visual Profiler Host to Device Copy Time Utilization**

<table>
<thead>
<tr>
<th>GPU Timestamp (us)</th>
<th>Method</th>
<th>GPU Time (us)</th>
<th>CPU Time (us)</th>
<th>grid size</th>
<th>thread block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>memcpyHtoAsync</td>
<td>23141.5</td>
<td>23428</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>innerHash</td>
<td>332352</td>
<td>332370</td>
<td>[1992 1 1]</td>
<td>[256 1 1]</td>
</tr>
<tr>
<td>3</td>
<td>memcpyHtoAsync</td>
<td>3845.18</td>
<td>3856.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>innerHash</td>
<td>104139</td>
<td>104194</td>
<td>[623 1 1]</td>
<td>[256 1 1]</td>
</tr>
<tr>
<td>5</td>
<td>memcpyHtoAsync</td>
<td>1203.74</td>
<td>1212.74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>innerHash</td>
<td>32792.8</td>
<td>32801.8</td>
<td>[195 1 1]</td>
<td>[256 1 1]</td>
</tr>
<tr>
<td>7</td>
<td>memcpyHtoAsync</td>
<td>377.44</td>
<td>384.44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>innerHash</td>
<td>1031.19</td>
<td>10318.9</td>
<td>[61 1 1]</td>
<td>[256 1 1]</td>
</tr>
<tr>
<td>9</td>
<td>memcpyHtoAsync</td>
<td>119552</td>
<td>124552</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>innerHash</td>
<td>3210.78</td>
<td>3216.78</td>
<td>[19 1 1]</td>
<td>[256 1 1]</td>
</tr>
</tbody>
</table>

**Fig. 7.5 NVIDIA Visual Profiler Host to Device Copy Details**
It was observed that frequency of memory copies from Host to Device and Device to Host was the main overhead of the code. Therefore, few optimization techniques were used including pinned memory and shared memory to optimize the code and save memory flaws.

7.3.1 Performance Optimization

Performance of an application can be optimized by three basic strategies in CUDA: maximize parallel portion to achieve maximum exploitation; optimize instruction set execution to achieve maximum instruction throughput; optimize memory requirement to achieve maximum memory throughput.

Based on the above results, it was observed that the reason for under-performance of the code is improper memory accesses and continuous copy of data from host to device and vice versa. Therefore in order to fix it, the code was optimized by using the feature of pinned memory and shared memory of CUDA to achieve gain in performance.

7.3.1.a Pinned Memory

The developers need to struggle to minimize data transfers between the host and the device in an application. CUDA runtime environment provides high performance for data transfers between host and device by using page-locked (also known as pinned) host memory functions as opposed to regular pageable host memory allocated by malloc() (Sanders and Kandrot, 2010).

In addition, mapped page-locked memory eliminates the need of device memory allocation and copy of data between device and host memory. Each time the kernel accesses the mapped memory, data transfers are performed implicitly. Performance improvements can be achieved by using mapped pagelocked memory instead of explicit copies between device and host memory. The cudaHostAlloc() and cudaFreeHost() functions are used to allocate and free page-locked host memory.

The same concept has been used in the code to optimize it for data copies (shown in Fig. 7.6) with the following commands:

cudaSetDeviceFlags(cudaDeviceMapHost);
cudaHostAlloc((void**)hash,SIZE,cudaHostAllocMapped);
cudaHostGetDevicePointer((void **)&device_hash,void *)hash,0);
Fig. 7.6 Page-locked Memory Usage Shown in NVIDIA Visual Profiler
7.3.1.b Shared Memory

Shared memory (Sanders and Kandrot, 2010) is an on-chip memory which has much higher bandwidth and much lower latency than local or global memory. In order to achieve high bandwidth, shared memory is divided into equal sized memory segments known as banks. These banks can be accessed simultaneously. However, it is important to know how memory addresses map to memory banks in an application to minimize the bank conflicts and schedule the memory requests.

Shared memory and pinned memory optimization techniques reduced the memory copy overhead of the code and increased the GPU utilization. Figure 7.7 shows the maximum time utilization by the innerHash() function rather than memcpyHtoDasync after optimization. The CUDA implementation of the Algorithm is given in Appendix E for the reader’s reference and its implementation results in Table 7.3 for the purpose of analysis.

![Graph showing GPU Utilization](image)

Fig. 7.7 GPU Utilization

Table 7.3 shows the execution results after using pinned memory and shared memory concepts. The speedup achieved over sequential SHA-1 is approximately 5.5X and 4.5X over RSHA-1 on CPU (Intel Core 2 Duo) (Kishore and Kapoor, 2015). It is observed from Fig. 7.8 that optimized GPU code has outperformed than both of the algorithms for large files as well on a dual core system.
### Table 7.3 Execution Time SHA-1 and Optimized RSHA-1 on CPU, GPU

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>File Size</th>
<th>Execution Time (in sec)</th>
<th>SHA-1</th>
<th>RSHA-1 on CPU</th>
<th>Optimized RSHA-1 on GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>512 KB</td>
<td></td>
<td>0.01426</td>
<td>0.01307</td>
<td>0.0029</td>
</tr>
<tr>
<td>2</td>
<td>1 MB</td>
<td></td>
<td>0.02829</td>
<td>0.02476</td>
<td>0.0055</td>
</tr>
<tr>
<td>3</td>
<td>2 MB</td>
<td></td>
<td>0.05836</td>
<td>0.05064</td>
<td>0.01125</td>
</tr>
<tr>
<td>4</td>
<td>3 MB</td>
<td></td>
<td>0.08543</td>
<td>0.07175</td>
<td>0.01594</td>
</tr>
<tr>
<td>5</td>
<td>4 MB</td>
<td></td>
<td>0.11312</td>
<td>0.09544</td>
<td>0.02121</td>
</tr>
<tr>
<td>6</td>
<td>8 MB</td>
<td></td>
<td>0.21336</td>
<td>0.17754</td>
<td>0.03945</td>
</tr>
</tbody>
</table>

Fig. 7.8 Performance of SHA-1 and Optimized RSHA-1 on CPU, GPU

### Conclusion

In this chapter we looked at the implementation of proposed hash structure in GPUs. We studied the GPU architecture and implemented RSHA-1 in CUDA. If we compare the results of RSHA-1 CPU performance on AMD Fx8320 Bulldozer machine (please refer to Chapter 3, Section 3.3.5) and RSHA-1 GPU performance on Intel Core 2 Duo E7500 machine, GPU is still lacking behind. One of the reasons observed is bank conflicts in case of using shared memory.
and other is the excessive use of pinned memory. Consuming too much page-locked memory, reduces overall system performance due to the reduction in the amount of physical memory available to the operating system for paging. Therefore, more optimization techniques are required to achieve better performance. The future research work here would include optimization of GPU implementation of RSHA-1 and fully exploit thousands of cores available on GPUs.
CHAPTER 8

ENERGY ANALYSIS OF EITRH
Chapter 8

ENERGY ANALYSIS OF EITRH

The next objective of our research work is to analyze the algorithm developed for its energy efficiency as well as for making it more applicable to portable devices such as smartphones and tablets. Longer battery life is a critical requirement for today’s performance demanding devices. Poorly written applications can be a drain for phone’s battery. In this chapter, we have analyzed the energy consumption by EITRH transformation based algorithm during its execution on varied file sizes so as to prove it energy efficient and suitable for many hand held devices. The comparisons between the serial implementation of conventional hashing algorithm and the parallel EITRH based algorithm is presented. Our results confirm that the proposed hash function is much more energy-efficient than the conventional algorithms.

8.1 Introduction

Nowadays, CHFs are used in almost every modern security application, especially in protocols, digital signatures, mobile devices and digital forensics for achieving message authentication and integrity protection. Handheld devices use hashing to maintain the security of the data stored in them or while transmitting the data. However, a constant use of cryptographic algorithms results in significantly reduced lifetime for battery powered devices. As, special care should be taken while downloading the files from internet, utilizing security algorithms which are fast enough may consume more energy and drain device’s battery faster. Therefore, people these days prefer fast but energy-efficient applications on their portable devices to save on the energy consumption of these devices.

In Chapter 3, we provided a new fast means of generating hash values in the form of EITRH transformation. We proved that EITRH based algorithms gives
better performance than existing hashing algorithms on multi-core processors. Now, the next motive is to check energy efficiency of the algorithm that was developed. In Section 8.2, we discuss the some of the concepts related to energy consumption followed by a discussion on the test environment, experimental results obtained, and the power impact of higher performance in Sections 8.3, 8.4 and 8.5 respectively.

8.2 Energy Efficiency

Energy consumption is becoming a critical concern, both in mobile and computing sectors these days. Its optimization can be made at various levels of abstraction: at the computer architecture level, at the system-layer level, and at the application-layer level (Kansal and Zhao, 2008). Since the development and the use of multi-core processors in our devices of various kinds, the demands for their performance as well their energy-efficiency is also increasing. The main objective for the development of multi-core processors was to have performance scalability along with better energy management in these devices.

It is important to note that, we can always find difference between the performance and the energy scalability of multi-core processors. Energy scalability is not same as performance scalability. This is mainly due to two important factors (Korthikanti and Agha, 2009):

- Power and frequency have a nonlinear relationship between them for a given multi-core processor. Typically, power consumed by a core is proportional to the square of the operating voltage and directly proportional to its frequency of operation. There is linear dependency between supply voltage requirement of a device and the frequency it needs to operate at. This makes the overall dependency have a cubic relationship with the operating voltage.
- Execution of parallel algorithms typically involves communication and computation. The power and performance characteristics of communication and computation in an algorithm may vary.
In the case of sequential algorithms, both the execution time and the energy consumption are proportional to the total number of operations. Therefore, for a fixed performance target in sequential algorithms, the amount of energy required is also fixed. However, the frequency of a core can be decreased to reduce the energy consumption but at the cost of increased computation time.

In the case of parallel algorithms, the defined relationship of execution time and energy consumption, for a given number of processing cores and the operating frequency of each core, can be more involved (Chandrakasan et al., 1992). In (Mani and Jee, 2007), it is shown that, each core in a multicore processor can be executed at lower frequency by dividing the power rather than giving it to a single core. The authors have proved that, increasing clock frequency by 20% to a single core gives a 13% performance growth, but requires approximately 73% greater power. On the other hand, if clock frequency is decreased by 20%, power reduction is up to 49% but leads to only 13% performance loss. Thereafter, adding another core in a single core design, it results in 73% performance gain with 20% reduced clock frequency, while using the same power as of single core processor at maximum frequency. This relation between power and frequency can be effectively used in multi-core processors to increase the number of cores for improved performance without increasing the power usage.

On the basis of these facts, EITRH transformation based RSHA-1 algorithm is also analyzed for its energy efficiency on multi-core processors. The results of the experiments conducted are discussed in succeeding sections.

### 8.3 Test Environment

We measure the energy consumption of our algorithms running on a machine with the characteristics provided as Setup II in Chapter 2, Section 2.3. Many instruction level power estimation tools have been proposed to estimate the power consumption by a particular application. We have chosen Microsoft’s
Joulemeter as a tool to estimate the power consumption by hashing algorithms. Setup II was chosen for measuring energy efficiency as, the tool Joulemeter runs only on Windows 7 operating system. Therefore, Code::Blocks and MinGW were also used for compiling and executing the binaries in Windows operating system.

8.3.1 Joulemeter

Joulemeter (Goraczko et al., 2011) is a software tool by Microsoft to view the power consumption of a computer. It estimates the energy usage of a virtual machine, computer, or an application by measuring various hardware resources involved like CPU, disk, memory, monitor, etc. and then converting the estimated usage of resource to its actual power consumption based on the tools automatically learned realistic power models. It gives an opportunity to test an application’s energy consumption under various scenarios and operating conditions. The results provided by the tool are being used to improvise the power consumption costs for high-end data centers, virtualized power budgeting, pc energy optimizations, and mobile battery management.

8.3.1.a Using Joulemeter

Joulemeter estimates the power consumption through a process called calibration. Calibration develops a power model which relates the computer resource utilization and hardware power states like processor utilization, processor frequency, screen brightness, monitor on/off state, disk utilization to the power drawn.

The tool can be used to measure the energy consumed by the computer as a whole or for a particular application. The data can be recorded every second in the form of a .csv file. On laptops calibration can be performed without any external power meter. For desktops, a Watts UP PRO power meter is used and if not available, approximate power data can be monitored by doing manual entry.
Figure 8.1 shows a screenshot of the power metering interface exposed by Joulemeter. The interface of the tool comes with three tabs: Calibration, Power usage and about. In the power usage tab, power usage of component is shown like CPU, monitor, disk and base. Application’s CPU power consumption per second can also be recorded by entering the name of executing application and initiating the program. There are preset settings for the components which can be changed through the calibration tab by selecting the model type or by doing manual entry as shown in Fig. 8.2.

![Joulemeter Power Metering Interface](image)

**Fig. 8.1 Power Usage Window of Joulemeter Tool**
Base (Idle) Power in the calibration window represents the least power consumed by the computer when turned on or when no programs are running. Here, the Processor Peak Power - high frequency is the power consumed at its highest frequency and low frequency is the power consumed at its lowest (P-state) frequency. Monitor power represents the power consumption by the monitor.

8.3.2 Energy Conversions

Joulemeter provides the total energy used for the entire run of the application by adding up the power values (in Watts) for the duration of execution. Since
each resulted value is the power use over one second, therefore its sum is the 
energy used in Joules. Joules can be converted to other units such as Watts using 
the following conversion factor:

\[
Watt = Joules / Second
\]

## 8.4 Experimental Results

In order to measure the power consumption of the proposed algorithm, two test 
environments including a laptop and a dual quad core machine were taken with 
the configuration given in Table 8.1.

<table>
<thead>
<tr>
<th>Component</th>
<th>Laptop (Platform 1)</th>
<th>Desktop (Platform 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core 2 Duo</td>
<td>AMD FX (tm) - 8320 Eight - Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Processor</td>
</tr>
<tr>
<td>Frequency</td>
<td>1.40 GHz</td>
<td>3.50 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>2.00 GB</td>
<td>8.00 GB</td>
</tr>
<tr>
<td>System Type</td>
<td>32 Bit Operating System</td>
<td>64 Bit Operating System</td>
</tr>
<tr>
<td>Environment</td>
<td>Windows 7</td>
<td>Windows 7</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC v4.8.1 with MinGW64</td>
<td>GCC v4.8.1 with MinGW64</td>
</tr>
<tr>
<td>Editor</td>
<td>Code::Blocks</td>
<td>Code::Blocks</td>
</tr>
<tr>
<td>Tool</td>
<td>Joulemeter 1.2</td>
<td>Joulemeter 1.2</td>
</tr>
</tbody>
</table>

The two test environments were used to check the energy efficiency of the 
proposed algorithm on both desktop servers and hand held devices. Experiments were conducted on both the machines to measure energy 
consumption for which few assumptions were made. We assume that all cores 
are homogeneous and operate at the same frequency, the cores that are idle 
consume minimal power and communication time between the cores is 
constant. We validate the assumption by making a note that the time spent for 
transmitting and receiving a message is high as compared to the time consumed 
to route the message between the cores. Figure 8.2 shows one of the results of 
the energy consumption by the algorithm using Joulemeter as a tool.
For algorithms SHA-1 and RSHA-1, recordings on both the platforms specified in Table 8.1 were done for 512 MB and 1 GB of file. Table 8.2 summarizes the energy cost and throughput performance of the algorithms in the form of µJ/B and MB/s on Platform 1. The readings recorded are for the by default operating frequency, monitor and idle power settings in the tool. It can be observed from the table that although RSHA-1 outperforms SHA-1 in the form of performance in hashing bytes per second but however consumes more energy. Here the processor used is dual core, therefore RSHA-1 takes less time for execution but both of its cores consume more joules altogether.

<table>
<thead>
<tr>
<th>File Size</th>
<th>µJ/B</th>
<th>MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SHA-1</td>
<td>RSHA-1</td>
</tr>
<tr>
<td>512MB</td>
<td>0.550195</td>
<td>0.817383</td>
</tr>
<tr>
<td>1GB</td>
<td>0.5979</td>
<td>0.8212</td>
</tr>
</tbody>
</table>

Similarly, Table 8.3 shows the energy consumption and throughput performance of both the algorithms on Platform 2. Platform 2 is a high speed AMD Bulldozer server with eight operating cores. High throughput
performance from RSHA-1 can be observed with each processing element
drawing its own energy. It is observed from both the tables that RSHA-1
algorithm can give good results in terms of performance on parallel platforms
but is not power saver. It is trivial to see that the energy consumed by the
parallel algorithm increases with the number of cores. Therefore, there is
somehow need to control the energy consumption of the parallel algorithm by
still achieving speedup on multi-core processors.

Table 8.3 Energy and Throughput Performance on Platform 2

<table>
<thead>
<tr>
<th>File Size</th>
<th>µJ/B</th>
<th>MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SHA-1</td>
<td>RSHA-1</td>
</tr>
<tr>
<td>512MB</td>
<td>0.055469</td>
<td>0.11582</td>
</tr>
<tr>
<td></td>
<td>25.6</td>
<td>51.2</td>
</tr>
<tr>
<td>1GB</td>
<td>0.0588</td>
<td>0.1194</td>
</tr>
<tr>
<td></td>
<td>45.455</td>
<td>71.42857</td>
</tr>
</tbody>
</table>

We know that, energy equals the amount of power dissipated over time.
Therefore, by increasing the degree of parallelism increases the energy
consumption, but at the same time decreases the computation time on a multi-
core architecture. The trade-off depends on the overall structure of the
architecture, since certain elements may have constant size.

In relation to this, it was found that number of researchers are working on
dynamic power management through some software controlled techniques in
multicore processors. The idea is to use some controlling methods for runtime
power performance adaptation. For this two such ways have been proposed in
(Korthikanti et al., 2011): dynamic concurrency throttling which enables the
change of level of concurrency at runtime, and another is dynamic voltage and
frequency scaling (DVFS) which changes how much power is used.

In order to reduce the power consumption or the amount of heat produced by
the chip, DVFS (also known as CPU throttling) (Nowka et al., 2002) is used to
adjust the frequency of a microprocessor automatically. This technique is
commonly used in hand-held battery operated devices like laptops, mobiles,
PDAs, etc., having limited energy. Many high-end servers also use this in order
to decrease energy and cooling costs.
Energy consumption of any compute device depends on the voltage and frequency supplied to it. Therefore, lowering the frequency in a compute device will lead to a more than linear decrease in its energy consumption. A linear increase in processing elements could compensate the loss of performance in terms of execution time, with the same energy consumption by scaling down the frequency supply.

In order to reduce energy consumption by the RSHA-1 algorithm, DFS was performed on Joulemeter results through the calibration of power model to scale down the frequency to the values specified by the vendor. Section 8.5 shows the impact of DFS on RSHA-1 for both the platforms.

8.5 Performance Impact

In DVFS technique, basically the number of instructions a processor can issue in a given amount of time are reduced, which in return reduces the performance. The statement is based upon the physical law for power consumption that:

\[
\frac{\text{Energy}}{\text{time}} \propto \text{voltage}^2 \cdot \text{frequency} \tag{8.1}
\]

The results obtained in Table 8.2 and Table 8.3 are based on predefined system settings. As per the system specifications, Platform 1 can operate from 1400 MHz to 1200 MHz and Platform 2 can operate from 3500 MHz to 1400 MHz, with a corresponding core supply voltage of 1.5V to 0.95V. Therefore the results were again measured on Joulemeter by calibrating system on its low power states.

Figure 8.4 shows the post calibration results for 512 MB of file hashed on Platform 1. It is observed from the results that if we execute the parallel algorithm on dual cores at high frequency then the energy consumed is near to double, but if we calibrate the frequency of the system to its specified low power states then the energy consumption decreases 10% with a slight increase in execution time.
Fig. 8.4 Comparison of Energy and Throughput Performance by RSHA-1 on Platform 1

Fig. 8.5 Comparison of Energy and Throughput Performance by RSHA-1 on Platform 2
Similarly, Fig. 8.5 shows a comparison of the quantity of data that can be hashed in a second of time and with a Joule of energy. The execution results are for hashing 512 MB of file on Platform 2. Our motive for the experiments is to have low energy consumption with high throughput performance. It is observed from the experimental results received that for serial SHA-1, energy consumed for hashing a file is low but throughput is also low. Unlike SHA-1, parallel RSHA-1 consumes high energy and also provides high throughput performance. Therefore, the system was calibrated to its low power states to operate it on low frequency for RSHA-1 algorithm to have a fair comparison at the same performance levels. The results have shown approximately 30% of less energy consumption. It was inferred that reducing the frequency and increasing the number of processing cores shows a quadratic relationship in which energy consumption can be reduced. Moreover, if we assume that the required speedup is one (1) then, reducing frequency can prove the RSHA-1 algorithm to be much more energy-efficient on multi-core processors operating at their low power states.

**Conclusion**

In this chapter, the energy-efficiency of the proposed algorithm has been proven for hand held devices as well as desktop servers. Experiments were been carried out using Microsoft’s Joulemeter tool to measure the energy consumed by the proposed parallel hashing algorithm and compared with the sequential implementation. The results generated show that if we execute the parallel algorithm on multiple cores at high frequency then the energy consumed is almost double but if we decrease the frequency of the system then energy consumption decreases significantly with the increase in execution time. This feature can be used to make the RSHA-1 algorithm suitable for handheld digital devices as most of these devices use multi-core processors now.

Our analysis shows that the energy consumed by a parallel program can be reduced in a polynomial fashion by determining the lower frequency operation, without losing much performance, for an application for a given number of cores.
CHAPTER 9
CONCLUSIONS AND FUTURE WORK
Chapter 9

CONCLUSIONS AND FUTURE WORK

In this chapter, the thesis concludes with contributions of the research and brief discussions about how our work can be extended. A summary of our work in this thesis is given in the chapter which can be classified into seven parts: cryptanalysis of existing hash functions; designing new tree-based hash function transformation EITRH; designing algorithm based on proposed framework; cryptanalysis of the EITRH framework and its application in different areas; performance analysis; and energy efficiency analysis. In Section 9.1, thesis contributions are elaborated along with conclusions in Section 9.2, while avenues for further research in this area are explored in Section 9.3.

9.1 Thesis Contributions

Over recent years, the use of digital devices has increased tremendously. Digitization of information and transmission of data over network has led to a revolution in the computing world. Electronic commerce has become a part and parcel of one’s life by providing services like online shopping, order-tracking, online payments and net banking. Security of all the digital information has become a necessary concern for the success of the system. In all of these applications and processes, cryptographic algorithms are playing a vital role.

Cryptographic hash functions are being widely used for providing authentication and data integrity. But with the increase in the digital information, traditional CHFs can degrade the overall performance of the system. Therefore, designing new parallel CHFs is required for digital world to guarantee authentication and data integrity with speed.

With the digitization of the information, digital crime is also increasing rapidly. Construction of fast and secure security primitive is essential for the use in
digital forensic tools. In this thesis, we explored comprehensively the analysis and design of hash functions. The main purpose of this thesis is to design new, fast and secure parallel hash function for security and forensic applications. In addition, we have studied and analyzed the security, energy efficiency and application in digital signatures of the proposed parallel hashing algorithm.

We provided a thorough discussion on the state of the art in CHF’s design framework. We have also discussed the SHA family hash functions, how they fell prey to various generic attacks and what all modifications were proposed to fix it along with the state of the art in parallel hash functions.

In this thesis, we proposed a fast parallel recursive tree-based transformation called the EITRH. Tree hashing along with EMD construction has been used in designing EITRH functions. In order to analyze the performance of the new framework, it has been tested and analyzed for few performance benchmarks for metrics including speedup, efficiency, cost optimality, and scalability. The comparison results showed that RSHA-1 outperforms existing hash functions commonly used today. Speedup of approximately 6.5X has been observed with 625.24 MB/s of bandwidth and an efficiency of 0.88 when executed on an AMD Bulldozer FX machine. Along with this, RSHA-1 has also shown better performance than the existing parallel hash-based schemes from the literature.

Several computer simulations and theoretical analysis of EITRH class hash functions is done to satisfy characteristics and conditions required for CHFs such as, collision resistance, high bit confusion and diffusion, flexibility, and fast speed. It can be concluded from the results obtained that EITRH transformation-based algorithm satisfies the essential properties of CHF to make it useful for authentication purposes and has high-potential for adoption in security applications.

Randomness test and performance tests in Chapter 5, proved EITRH to be a good option for use in forensic applications during the process of evidence acquisition and can be used for presentation of evidence analysis in the courts.
Further experiments were also done to prove the benefit of using EITRH algorithm in digital signatures for large file sizes to ensure security in the form of authentication, non-repudiation, and integrity.

We have also implemented proposed algorithm on nVIDIA GPUs as they are known for their ability to accelerate common applications. The speedup achieved on the GPU implementation is 5.5X, which is less than 6.5X achieved when executing the OpenMP code on the multi-core AMD CPU. It was observed that hashing large amount of data on GPUs still require more fine-grained parallelization for their best performance.

The proposed framework uses more instructions than the existing sequential ones. More instructions consume more frequency cycles. Moreover, the proposed algorithm is executed on multi-core architecture for better performance which in turn consumes more energy. Experiments in Chapter 8 have been performed using Microsoft’s tool Joulemeter to measure energy consumed by the proposed hashing algorithm and the traditional ones. The results generated shows that if we execute the parallel algorithm on eight cores at high frequency then the energy consumed is almost double but if we decrease the voltage and frequency of the system then energy consumption decreases significantly at the same performance level. This feature can be used to leverage the advances of algorithm on battery operated handheld digital devices, as most of these devices, such as smartphones and tablets, use multi-core processors.

9.2 Conclusions of This Thesis

In this thesis, we studied existing cryptographic hash functions, their designs, security properties and notions. It was observed from the study that most of the prevalent hash functions like MD5, SHA-1, SHA-2, etc. are based on Merkle-Damgård construction and are, by default, serial in nature. Researchers have started working on developing new hash constructions. In order to exploit the modern computer architecture properly, the need of parallel hash constructions was observed.
As per the topic of this thesis, research went into two directions: developing parallel hash functions and leveraging its advances in various applications. In Chapter 3, we have provided a novel parallel hashing framework to generate a family of hash functions. The improvements of 85% in performance for parallel implementation of proposed algorithm over existing one confirms the claim that the algorithm outperforms the existing hash algorithms in terms of speedup and efficiency on a multicore architecture. However, on a single core, the framework could degrade the performance.

It is also observed that as the size of data and number of processors increases, the performance shown by the algorithm also shows better results than that of standard CHF and efficiency increases if the problem size is increased when we keep the number of processing elements constant. In contrast, the speedup tends to saturate and efficiency drops with the increasing number of processors for a fixed problem size. Therefore, it is concluded that the number of processing cores, problem size, and performance are directly proportional to each other. Increasing the number of processing cores may not work every time especially for small file sizes.

RSHA-1 algorithm is proved to be faster than SHA-1 but has more memory requirements than SHA-1 due to its rehashing nature. Therefore, applications requiring higher speedup can prefer RSHA-1. However, in case of small file sizes or in password verification process, RSHA-1 can degrade the performance as it is meant for higher performance on large data sets.

Digital forensics require data similarity detection, data integrity, and authentication services and therefore use the benefits of the new framework. Message digest generation is considered to be a necessary task in the investigation process. In Chapter 5, the results obtained for randomness test and performance test confirms the benefit of using proposed algorithm in forensic tools. The increasing size of digital information demands a fast and secure image generating algorithm.
In Chapter 6, RSHA-1 has also proved its supremacy in digital signature generation and verification process. The process can completely take the benefit of today’s multi-core architecture by parallelizing both digest generation and signature generation process. The performance of RSHA-1 is only checked along with RSA algorithm, further experiments can be performed with other signature generation algorithms.

In any given application, there is an interdependence between three quantities: workload, performance, and energy. Performance can be improved by dividing the workload on number of processors and hence increasing the energy consumption. Each processing core draws its own energy and therefore increasing the number of processing cores increases the energy consumption while increasing performance. It was inferred from Chapter 8 that energy consumption can be reduced on multi-core architectures by using the Dynamic Frequency Scaling technique. The concept can be effectively used in hand-held devices to reduce the energy consumption of these algorithm running at the same performance level.

9.3 Future Research Directions

While closing this chapter, some of the possibilities for future work directions are summarized as follows:

- **Design and analysis of parallel hash functions**

This dissertation presented a transformation for creating a parallel CHF from the blend of conventional and modern constructions. More efficient methods are needed to create a viable parallel CHF on the basis of proposed transformation. The most obvious avenue for creating a more efficient parallel CHF is by introducing parallelism in its compression function as well. Improved variants of EITRH can be proposed for more efficient computation of digests while leveraging its advances to computing world.
• **Design and analysis of keyed hash functions**

As discussed in Chapter 1, CHFs are mainly keyless and it is difficult to convert a keyless hash function to a keyed hash function. Therefore, the analysis and design of keyed CHF based on EITRH transformation is a promising route for designing secure and fast algorithms for high-security applications.

• **Cryptanalysis of EITRH based hash algorithms**

In Chapter 4, security analysis of EITRH transformation has been given including pre-image resistance, weak collision resistance, collision resistance, non-correlation resistance, and partial pre-image resistance. All of the security of the algorithm related results presented in this work were carried out under the assumption that particular underlying primitive is ideal. However, a more regressive cryptanalysis of proposed transformation based algorithms can be conducted with a goal to examine its resistance to existing and new attacks.

• **Adopting EITRH in digital forensics**

In Chapter 5, result analysis has been done on the maximum file size of 4GB and compressed 20GB. The preferred direction of research would be to perform experiments on larger files, typically a hard disk of size in terabytes so as to fully encounter proposed algorithm’s usage in digital forensic application. In order to construct an authentic, fast forensic investigation tool, it has been left as an open ended problem to embed the proposed algorithm in the available tools. Further research can also be done to transform the algorithm from digest generation to a similarity detection algorithm as well.

• **EITRH as a pseudo random number generator**

A possible continuation of the research could go in direction of extending usage of EITRH based hash algorithms as pseudo-random number generators. The proposed algorithm can also be tested with NIST statistical test suite to qualify
as a pseudo-random number generator. This can be further used to provide high security for many cryptography systems such as digital signature, secret key generation for the DES algorithm, and as a prime number generator in the RSA algorithm.

- **EITRH on hand-held devices**

Finally, more work can be done to check the energy efficiency of the proposed algorithm at hardware level. In Chapter 8, the results provided are based on the readings done by software tool Joulemeter. The implementation of the proposed algorithms on smart-phones and tablets can be used to prove its energy efficiency in real usage context.
REFERENCES


Gallagher P & Kerry C (2013) FIPS PUB 186-4: Digital Signature Standard, DSS. NIST.


Weste NH & Harris DM (2005) CMOS VLSI design: a circuits and systems perspective, Pearson Education India.


APPENDIX A

The corresponding hexadecimal hash codes generated from the simulator are as follows:

C1: 1B BD 90 15 52 7 EE 45 1D F0 D B0 B4 4C A2 82 39 39 C2 CE
C2: 60 40 D1 4F 6F 2D 42 CA E5 4E 72 D7 3A C1 9 88 43 8E 49
C3: 7D 78 78 37 78 A1 C4 8B 4 4D A5 A 64 75 69 6A EE 72 50 53
C4: 68 BF FA 1C 13 1 3E 63 C0 B4 40 D C1 BD 15 B9 BE 66 81 E9
C5: 97 B 2B FE 28 FD AA 8F A1 C8 EC 1C F4 2A B0 3C CD 3C 7E 56
C6: 32 16 33 E5 47 E9 3B 94 6E B9 38 65 CB D1 FE 64 32 A 4F CF
C7: 6 D2 9C 11 FB 9B 99 A1 E3 E4 6F 14 27 46 95 67 44 9A 85 77
C8: 5C 2C CC 79 3F 55 C8 9C 51 F3 8E 16 79 B6 41 5 A8 D7 E0 62 BF
C9: D 7 64 2B 9A 93 D 9D 3B 72 A4 33 B0 D9 61 53 D2 1E 2B BF DC
C10: D9 70 18 A8 78 5D 24 21 6F B3 FA B2 28 C4 40 24 65 D0 10 E1
C11: DA E0 FA 5A 5 7A F9 E 7E 25 A 79 5C 4D 73 38 EF 97 6F 12
C12: AC CC 3B 98 3A D9 F3 72 C6 18 54 82 54 14 35 D8 B6 67 95 B8
C13: D7 82 D8 9F 21 5A 4D D 6C 1C B0 4E AE D3 D5 8D 9B CA 5A 77
C14: E2 A1 C2 7D 70 F8 B1 CA 24 9A 80 D5 F 49 BB F9 B8 79 B3 4
C15: 30 1D 95 53 5A AE 59 D8 FC AD A6 D8 68 14 35 4C 5 3F F6 EBF
APPENDIX B

The corresponding hexadecimal hash codes generated from the simulator are as follows:

I11:   FA2BDA1583287E97DB793BD2D3B565333B9C55E96
I12:   9B72AA6B5145CE1EC8D181AF66C20B9CC2AA569
I13:   69CA25126D361F6A18159D4B483F96228D49FBF
I14:   05D6D2EE8CB6A2556B6B536FBEA2DE5CD8084
I21:   D7A7D2E98645B6C0DAC696613CFE4AAE6FBE7D0
I22:   5AAD1A050A5BACADFE48D9D379FE8D88EF75A28
I23:   456E37FCA69B5B3CBBF796CE48EB52FFB429
I24:   C46CC12528E727216D709D9269154482B3DC9712
I25:   EC7EC391E5BF5279D0C4F5FE4E935853F0C188D8
I26:   B2985F25A96F24D97553D94689959D2C6C41253A
void innerHash(unsigned char* sarray, unsigned char* hash)
{
    // HASH GENERATION OF EACH BLOCK
}

void calc(const unsigned char* sarray, const long int bytelength)
{
    // VARIABLE INTIALIZATION
    // DYNAMIC MEMORY ALLOCATION TO HASH
    // THREAD SIZE GENERATION
    omp_set_num_threads(omp_get_num_procs() * 4);
    #pragma omp parallel default(none)
        shared(sarray, hash, noi, n, SIZE) /
    private(currentBlock, endCurrentBlock, index, w)
    {
        #pragma omp for schedule(guided, 8) nowait
        for (i = 0; i < noi; i++) // FOR EACH BLOCK ITERATION
        {
            currentBlock = i * 64;
            endCurrentBlock = currentBlock + 64;
            index = (i * 20) + 20;
            for (int roundPos = 0; currentBlock < endCurrentBlock; currentBlock += 4)
            {
                // GENERATE W FOR EACH BLOCK
            }
            innerHash(w, hash, index, 1);
            #pragma omp flush(hash)
        } // END OF FOR LOOP
    } // END OF PRAGMA

    // Handle last and not full 64 byte block if existing.
    // RECURSIVE CALL TO calc()
}

int main(int argc, char* argv[])
{
    // Read the file in src
    calc(src, bytelength);
    // print hash
    return 0;
}
APPENDIX D

/* -------------- SIGNATURE GENERATION --------------*/

//Generate pair of RSA keys (d= private key, e= public key)

//Generate hash code of the message in Parallel using RSHA-1 Algorithm

//Generating Signature in Parallel
#pragma omp parallel for schedule (guided, iter) \
shared(plain, cipher) private (x) \default (shared)
    for(x=0;x<length;x++)
    {
        mpz_powm(cipher[x], plain[x], d, n);
    }

/* -------------- SIGNATURE VERIFICATION --------------*/

//Extract message and digest from the message received

//Apply public key to decrypt the digest received
#pragma omp parallel for schedule (guided, iter) \
shared(aplain, cipher) private (x) \default (shared)
    for(x=0;x<length;x++)
    {
        mpz_powm(aplain[x], cipher[x], e, n);
    }

//Generate hash code of the message in Parallel using RSHA-1 Algorithm

//Verify the signatures by comparing both the hash codes
__global__ void innerHash(unsigned char* sarray, unsigned char* hash)
{
    int index= threadIdx.x + blockIdx.x * blockDim.x;
    // HASH GENERATION OF EACH BLOCK IN THE KERNEL
}

void calc(const void* src, const int bytelength)
{
    int numThreadsPerBlock = 256;
    int numBlocks=1;

    //extra cuda variables for copying dat to device
    unsigned char* device_sarray;
    cudaMalloc((void**)&device_sarray ,bytelength);

    // NUMBER OF THREADS AND BLOCKS ALLOCATION
    // copying sarrary to device
    cudaMemcpyAsync(device_sarray,sarray,bytelength,
                     cudaMemcpyHostToDevice);

    // Launching Kernel
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);

    innerHash<<<dimGrid,dimBlock>>>(device_sarray,device_hsh;
    cudaMemcpyAsync(device_sarray,sarray,bytelength,
                     cudaMemcpyHostToDevice);

    cudaThreadSynchronize();

    // check if kernel execution generated an error
    checkCUDAError("kernel execution");

    /* HANDLING THE LAST BLOCK OF 64 BYTES ON HOST   */
    cudaFree(device_sarray);
    // RECURSIVE CALL TO calc()
}

int main(int argc, char* argv[])
{
    cudaDeviceProp deviceProp;
    #if CUDART_VERSION < 2020
    #error "This version does not support mapped memory!\\n"
    #endif

    //...
// Get properties & verify device 0 supports mapped memory

cudaGetDeviceProperties(&deviceProp, 0);
checkCUDAError("cudaGetDeviceProperties");

if(!deviceProp.canMapHostMemory) {
    fprintf(stderr, "Device %d cannot map host mem!\n", 0);
    exit(EXIT_FAILURE);
}

// set the device flags for mapping host memory
cudaSetDeviceFlags(cudaDeviceMapHost);
checkCUDAError("cudaSetDeviceFlags");

// allocate mapped arrays
cudaHostAlloc((void **)&hash, SIZE, cudaHostAllocMapped);
checkCUDAError("cudaHostAllocMapped");

// Get the device pointers to the mapped memory
cudaHostGetDevicePointer((void **)&device_hash,
       (void *)hash, 0);
checkCUDAError("cudaHostGetDevicePointer");

// Calling calc()
// Read the file in src
calc(src, bytelength);

cudaFreeHost(hash);
return 0;
}