The Feasibility of Memory Encryption and Authentication

Donald Owen, Jr.

Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78705, USA

FastPath 2013: April 21, 2013
Outline

Introduction
Motivation
Background
Solution Characterization
  Software
  Hardware
Results
Conclusion
Motivation

Digital data is increasingly put on mobile devices and remote servers. This data needs to be protected.

Security Problems
- Health records
- SSN
- Private communication (emails)
- Corporate documents
- Crypto keys
- ... more

Security Solutions?
- Disk encryption
- Strong passwords
- Hardened software

A major component left unprotected: DRAM.
Existing Protections Not Enough

Attacks abound

- Passive
  - Bus Sniffing*
- Active
  - Spoofing
  - Splicing
  - Replay
  - Cold boot*
Existing Protections Not Enough

Xbox™ hacked by Andrew “bunnie” Huang (MIT) using a bus sniffer to read a secret key / decryption code.

Image Huang [1]
Existing Protections Not Enough

The “cold boot” attack, and variants thereof, exploit DRAM remanence to extract data from RAM.

1. Interrupt power of a running system
2. Reboot into custom OS
3. Dump contents of DRAM to permanent storage
4. Mine dumped data for keys, files, fragments
5. Use recovered data to exploit, for example, encrypted disk
Outline

Introduction
Motivation
Background
Solution Characterization
  Software
  Hardware
Results
Conclusion
Memory Encryption & Authentication

What do we want?
C: Confidentiality
I: Integrity
A: Authentication

How do we get it?
Encryption
Hash/Tag/Signature
Tag/Signature

In short, encrypt and tag data to RAM; decrypt and authenticate data from RAM.
Memory Encryption & Authentication

- **Approach 1**: Encrypt and tag each cache line. Store tags on the chip. Verify on reading back from RAM.
  - Problem: Storage space.
- **Approach 2**: Use a tree structure! Store tags in DRAM with the root stored on the chip. Verify up the tree on reading back.
  - Problem: Speed.
  - Variants: Use a dedicated tag cache on the chip. Verify until you hit in the cache.
Memory Encryption & Authentication

Memory Space (M = 8)

- \( f \): authentication primitive
- Leaf: data node

Image Elbaz et al. [2]
Galois Counter Mode
Outline

Introduction
Motivation
Background
Solution Characterization
  Software
  Hardware
Results
Conclusion
Previous work assumed the existence of hardware adequate to the task of encryption, decryption, tagging, and verifying fast enough to meet performance demands. We evaluated how feasible those assumptions are under different implementation characteristics.

- **Software**
  - Pure C on x86
  - C + x86 Assembly
  - C + x86 Assembly + ISA Extensions

- **RTL on FPGA**
- **RTL on synthesized ASIC**
# Experimental Setup

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core i7 2620M</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Fedora Linux 17</td>
</tr>
<tr>
<td></td>
<td>GNU/Linux 3.7 x86_64</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.7.2</td>
</tr>
<tr>
<td>FPGA Synthesis</td>
<td>Xilinx ISE v. 14.3</td>
</tr>
<tr>
<td></td>
<td>Kintex 7-325T</td>
</tr>
<tr>
<td>ASIC Synthesis</td>
<td>Synopsys Design Vision v. E2010-12</td>
</tr>
<tr>
<td></td>
<td>FreePDK 45 nm Library</td>
</tr>
</tbody>
</table>
Pure C Implementation

MiBench has an AES (Rijndael) benchmark. We modified this benchmark to suit the implementation requirements.

Modifications

- Convert AES-CBC to AES-GCM.
- Convert File I/O to in-memory operations.
- Profile at cache line sizes
## Pure C Implementation

**Table**: Cycles per Byte Measurements for Pure C Implementation of AES-GCM

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Encrypt</th>
<th>Decrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>32B</td>
<td>52.2</td>
<td>74.2</td>
</tr>
<tr>
<td>64B</td>
<td>37.8</td>
<td>50.4</td>
</tr>
<tr>
<td>128B</td>
<td>35.6</td>
<td>39.8</td>
</tr>
<tr>
<td>256B</td>
<td>28.3</td>
<td>35.8</td>
</tr>
<tr>
<td>512B</td>
<td>25.4</td>
<td>33.0</td>
</tr>
</tbody>
</table>
C + Assembly

We can do better!

- The same code in the pure C implementation has optional Assembly routines.
- OpenSSL uses Assembly optimizations.
- Modern x86 processors have ISA extensions for AES and GCM.
# C + Assembly

Table: Cycles per Byte Measurements for Assembly-Optimized AES-GCM (64B Buffer)

<table>
<thead>
<tr>
<th>Method</th>
<th>Encrypt</th>
<th>Decrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure C</td>
<td>37.8</td>
<td>50.4</td>
</tr>
<tr>
<td>C + Opt.</td>
<td>22</td>
<td>30</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>~25</td>
<td>~25</td>
</tr>
<tr>
<td>ISA Extensions [4]</td>
<td>3.5</td>
<td>~3.5</td>
</tr>
</tbody>
</table>
Most previous work assumes the existence of hardware modules. We adapted an open-source AES-GCM module to be suitable for both FPGA and ASIC synthesis.
## RTL Module Characteristics - FPGA

### Table: Open-Source vs. Representative Commercial AES-GCM RTL Core on Kintex 7 FPGA

<table>
<thead>
<tr>
<th>Metric</th>
<th>Open Source</th>
<th>Commercial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup</td>
<td>19 clocks</td>
<td>0 clocks</td>
</tr>
<tr>
<td>16B Enc/Dec</td>
<td>22 clocks</td>
<td>12 clocks</td>
</tr>
<tr>
<td>16B Tag(Hash)</td>
<td>17 clocks</td>
<td>12 clocks</td>
</tr>
<tr>
<td>64B Cache Line + Tag</td>
<td>123 clocks</td>
<td>60 clocks</td>
</tr>
<tr>
<td>Freq. Max</td>
<td>212 MHz</td>
<td>256 MHz</td>
</tr>
<tr>
<td>Logic Slices</td>
<td>~800</td>
<td>~1000</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>
FPGA RTL Synthesis Results

Linear Fit: $193\text{mW/instance.}$
RTL Module Characteristics - ASIC

FreePDK 45 Implementation

- Freq Max: 250 MHz
- Area: 89k $\mu m^2$
- Power: 12 mW
ASIC RTL Synthesis Results

Linear Fit: $11.05mW/\text{instance}$. 
Outline

Introduction
Motivation
Background
Solution Characterization
  Software
  Hardware

Results
Conclusion
## Results - Summary

### Table: Summary of Different Implementation Methods

<table>
<thead>
<tr>
<th></th>
<th>ASIC</th>
<th>FPGA</th>
<th>x86 C</th>
<th>x86 Assembly</th>
<th>x86 ISA Ext.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock (Hz)</strong></td>
<td>225 M</td>
<td>200 M</td>
<td>2.7 G</td>
<td>2.7 G</td>
<td>2.7 G</td>
</tr>
<tr>
<td><strong>Cycles/Byte</strong></td>
<td>1.9</td>
<td>1.9</td>
<td>44</td>
<td>22</td>
<td>3.5</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>936.6 Mbps</td>
<td>882.5 Mbps</td>
<td>490.9 Mbps</td>
<td>981.8 Mbps</td>
<td>6.17 Gbps</td>
</tr>
<tr>
<td><strong>Typ. Power</strong></td>
<td>11.05 mW</td>
<td>192.9 mW</td>
<td>~35 W</td>
<td>~35 W</td>
<td>~35 W</td>
</tr>
<tr>
<td><strong>Typ. Area</strong></td>
<td>74.1 $k\mu$m²</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Mbps/mW</strong></td>
<td>84.7</td>
<td>4.57</td>
<td>1.40 * 10⁻²</td>
<td>2.81 * 10⁻²</td>
<td>1.76 * 10⁻¹</td>
</tr>
</tbody>
</table>
## Implementation Feasibility

**Table**: Peak Memory Bandwidth of Several Modern Systems

<table>
<thead>
<tr>
<th></th>
<th>Nexus 7</th>
<th>Nexus 10</th>
<th>iPhone 5</th>
<th>iPad 3</th>
<th>Intel i7</th>
<th>AMD FX</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW ((\frac{GB}{s}))</td>
<td>5.3</td>
<td>12.8</td>
<td>8.5</td>
<td>12.8</td>
<td>25.6</td>
<td>21</td>
</tr>
</tbody>
</table>
Implementation Feasibility

Table: Number of Instances to Meet Peak BW

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>C + Opt.</th>
<th>C + ISA</th>
<th>FPGA</th>
<th>ASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel i7</td>
<td>590</td>
<td>210</td>
<td>34</td>
<td>230</td>
<td>220</td>
</tr>
</tbody>
</table>

220 ASIC Modules $\approx 16 \text{ mm}^2$ at 45 nm.
220 ASIC Modules $\approx 2.4 \text{ W}$ at 45 nm.
Outline

Introduction
Motivation
Background
Solution Characterization
  Software
  Hardware
Results
Conclusion
What Have We Shown?

- Software solutions require too much power.
- Software solutions require too much area.
- Software solutions are too slow.
- FPGA solution may be useful for existing designs.
- ASIC solution may be feasible for implementation in a real system.
Thank you!

Questions?
References


