HashTag

Hash-based Integrity Protection for Tagged Architectures

Lukas Lamster  Martin Unterguggenberger  David Schrammel  Stefan Mangard
August 10, 2023

IAIK – Graz University of Technology
# Overview

- **Memory Tagging**
  - Hardware-enforced security
  - Mitigate memory safety issues
  - Introduces performance and memory overhead

- **DRAM Integrity Protection**
  - Detect and correct errors in data
  - Low memory overhead
  - Widely used in server systems

- **Our Contribution:**
  - We combine integrity protection and memory tagging
  - We perform a case study for existing memory tagging schemes
  - We reduce the performance overhead by an average factor of 20
DRAM Structure

- CPU
- Memory Controller
- DIMM
- Bank
- Control Bus
- Data Bus
- Sense Amplifiers
- Bank Rows
- Bank Columns
- Bit Line
- Word Line
- Cells **leak charge**
- Leakage **not constant**
  - Temperature, Radiation, ...
- Larger structures **influence multiple bits**
- **Common vs. uncommon** faults

<table>
<thead>
<tr>
<th>Failure Mode</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>1</td>
<td>Single faulty bit</td>
</tr>
<tr>
<td>F2</td>
<td>8</td>
<td>Single stuck pin</td>
</tr>
<tr>
<td>F3S</td>
<td>up to 56</td>
<td>Multiple stuck pins in a single chip</td>
</tr>
<tr>
<td>F3M</td>
<td>up to 56</td>
<td>Multiple stuck pins in multiple chips</td>
</tr>
<tr>
<td>F4</td>
<td>up to 64</td>
<td>Broken chip (all pins stuck)</td>
</tr>
<tr>
<td>F5S</td>
<td>up to 57</td>
<td>F3S + transient fault</td>
</tr>
<tr>
<td>F5M</td>
<td>up to 57</td>
<td>F3M + transient fault</td>
</tr>
</tbody>
</table>

Note: We only consider **naturally occurring** faults
- Use **Error Correcting Codes** (ECC)
- Add **redundancy** in additional chip
- Store **linear checksum** on write
- Verify on load
- Bus width **increases**

\[
R = f(D_0, D_1, \ldots, D_7)
\]
- Use Error Correcting Codes (ECC)
- Add redundancy in additional chip
- Store linear checksum on write
- Verify on load
- Bus width increases
- Limited error detection
  - Bounded by hamming distance
- Potential miscorrection
  - In case of large errors

\[ R = f(D_0, D_1, \ldots, D_7) \]
Memory Tagging

- Store **metadata** for each allocation
- Check metadata on access
- Enforce **tagging policies**
- Provide **memory safety**
- Implement **domain isolation**

```cpp
char* ptrA = new char[32];  // ✓
char* ptrB = new char[16];  // ✗
```

```
ptrA[2]
ptrA[32]
```
Memory Tagging

- Store **metadata** for each allocation
- Check metadata on access
- Enforce **tagging policies**
- Provide **memory safety**
- Implement **domain isolation**
- Additional storage overhead
- Increased memory pressure

```cpp
char* ptrA = new char[32];
char* ptrB = new char[16];
```

ptrA[2] ✓✓
ptrA[32] ✓✓

ptrA[2] ✓✓
ptrA[32] ✓✗
Can we combine tagging and integrity protection?

? “Steal” bits from linear code?

Additional Chip

64 Bit

8 Bit

Lukas Lamster — IAIK – Graz University of Technology
Can we combine tagging and integrity protection?

- "Steal" bits from linear code?
- Weakens error detection
- Weakens error correction
Can we combine tagging and integrity protection?

- "Steal" bits from linear code?
- Weakens error detection
- Weakens error correction
- Implicitly encode tags?
Can we combine tagging and integrity protection?

- “Steal” bits from linear code?
- Weakens error detection
- Weakens error correction
- Implicitly encode tags?
- Cannot read tags, aliasing possible
- Tag size is limited
Can we combine tagging and integrity protection?

? “Steal” bits from linear code?

? Weakens error detection

? Weakens error correction

? Implicitly encode tags?

? Cannot read tags, aliasing possible

? Tag size is limited

Takeaways:
Can we combine tagging and integrity protection?

- “Steal” bits from linear code?
  - Weakens error detection
  - Weakens error correction
- Implicitly encode tags?
  - Cannot read tags, aliasing possible
  - Tag size is limited

Takeaways:

- Explicitly store tag
Can we combine tagging and integrity protection?

- “Steal” bits from linear code?
  - Weakens error detection
  - Weakens error correction
- Implicitly encode tags?
  - Cannot read tags, aliasing possible
  - Tag size is limited

Takeaways:

- **Explicitly** store tag
- Use **non-linear** function
Idea:

- Replace linear checksum with **hash**
  
  Not necessarily a cryptographic hash

- Compute on **cache line granularity**

- **Truncate output** to accommodate tag

\[ H(\text{Cache Line, } \text{tag}) \]

\[
\begin{align*}
D_0 & \quad D_1 & \quad D_2 & \quad \ldots & \quad D_7
\end{align*}
\]

From CPU

\[
\begin{align*}
\text{fingerprint} & \quad \text{tag}
\end{align*}
\]
Idea:

- Replace linear checksum with **hash**
  - Not necessarily a cryptographic hash
- Compute on **cache line granularity**
- **Truncate output** to accommodate tag
- Tags are readable
Our Design

Idea:

- Replace linear checksum with hash
  Not necessarily a cryptographic hash
- Compute on cache line granularity
- Truncate output to accommodate tag

Tags are readable

Errors are detectable
We want to **detect** and **correct** errors
What properties does $\mathcal{H}$ need to have?
We want to **detect** and **correct** errors

What properties does $\mathcal{H}$ need to have?

- $P(\mathcal{H}(D) = \mathcal{H}(D'))$ should be low
We want to **detect** and **correct** errors

What properties does $\mathcal{H}$ need to have?

- $P(\mathcal{H}(D) = \mathcal{H}(D'))$ should be low
- $\mathcal{H}$ should be quick to compute
We want to **detect** and **correct** errors.

What properties does $\mathcal{H}$ need to have?

- $P(\mathcal{H}(D) = \mathcal{H}(D'))$ should be low
- $\mathcal{H}$ should be quick to compute

⚠️ Not cryptographically secure!

⚠️ Slightly slower

Note: Any function with strong diffusion is suitable if no cryptographic security is required.
We want to **detect** and **correct** errors.

What properties does $\mathcal{H}$ need to have?

- $P(\mathcal{H}(D) = \mathcal{H}(D'))$ should be low
- $\mathcal{H}$ should be quick to compute

⚠️ Not cryptographically secure!

⚠️ Slightly slower

Note: Any function with strong diffusion is suitable if no cryptographic security is required.
• Iterate over possible errors $e$
• Compute $H(D' + e)$
• Matching hash $\rightarrow$ error corrected
• Strong complexity growth
- Iterate over possible errors $e$
- Compute $H(D' + e)$
- Matching hash $\rightarrow$ error corrected
- Strong complexity growth
- Decrease complexity?
• Iterate over possible errors $e$
• Compute $\mathcal{H}(D' + e)$
• Matching hash $\rightarrow$ error corrected
• Strong complexity growth
• Decrease complexity?
  • Add **parity bits**
• Iterate over possible errors $e$
• Compute $\mathcal{H}(D' + e)$
• Matching hash $\rightarrow$ error corrected
• Strong complexity growth
• Decrease complexity?
  • Add parity bits
  • Consider error patterns
### Case Study: Tagged Memory Architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tag Size</th>
<th>Granularity</th>
<th>Bit Distribution</th>
<th>Faulty Bits</th>
<th>Correctable Failure Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIFT [6], HDFI [5]</td>
<td>1 bit</td>
<td>8 bytes</td>
<td>8 + 48 + 8</td>
<td>✓ ✓ ✓ ✓ ✓ (8,3)</td>
<td>2 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Shakti-t [4], M-Machine [3]</td>
<td>4 bits</td>
<td>64 bytes</td>
<td>8 + 52 + 4</td>
<td>✓ ✓ ✓ ✓ ✓ (8,3)</td>
<td>3 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>SPARC ADI [1]</td>
<td>1 bit</td>
<td>64 bytes</td>
<td>8 + 52 + 4</td>
<td>✓ ✓ ✓ ✓ ✓ (8,3)</td>
<td>3 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CHERI 128 [9, 8]</td>
<td>1 bit</td>
<td>16 bytes</td>
<td>8 + 52 + 4</td>
<td>✓ ✓ ✓ ✓ ✓ (8,3)</td>
<td>3 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CHERI 256 [9, 8]</td>
<td>1 bit</td>
<td>32 bytes</td>
<td>8 + 54 + 2</td>
<td>✓ ✓ ✓ ✓ ✓ (8,3)</td>
<td>3 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>ARM MTE [2]</td>
<td>4 bits</td>
<td>16 bytes</td>
<td>8 + 40 + 16</td>
<td>✓ ✓ ✓ ✓ ✓ (8,3)</td>
<td>2 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>lowRISC [7]</td>
<td>4 bits</td>
<td>8 bytes</td>
<td>8 + 24 + 32</td>
<td>✓ ✓ ✓ ✓ ✓ (8,3)</td>
<td>3 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model A</td>
<td>32 bits</td>
<td>64 bytes</td>
<td>1 + 31 + 32</td>
<td>✓ ✓ ✓ ✓ ✓ (8,2)</td>
<td>2 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model B</td>
<td>46 bits</td>
<td>64 bytes</td>
<td>1 + 17 + 46</td>
<td>✓ ✓ ✓ ✓ ✓ (4,2)</td>
<td>2 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model C</td>
<td>51 bits</td>
<td>64 bytes</td>
<td>1 + 12 + 51</td>
<td>✓ ✓ ✓ ✓ ✓ (8,4)</td>
<td>3 ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>No Tagging, SEC-DED</td>
<td>-</td>
<td>- 64</td>
<td>-</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>No Tagging, Chipkill</td>
<td>-</td>
<td>- 64</td>
<td>-</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

A wide range of TMAs can be implemented, Many error patterns are correctable. Combining tagging with hash-based integrity protection is feasible.
## Case Study: Tagged Memory Architectures

A wide range of TMAs can be implemented. Many error patterns are correctable. Combining tagging with hash-based integrity protection is feasible.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tag Size</th>
<th>Granularity</th>
<th>Bit Distribution</th>
<th>Faulty Bits in Hash (d)</th>
<th>Correctable Failure Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIFT [6], HDFI [5]</td>
<td>1 bit</td>
<td>8 bytes</td>
<td>8 + 48 + 8</td>
<td>✓ ✓ ✓ ✓</td>
<td>3 ✓ (8,3) 2</td>
</tr>
<tr>
<td>Shakti-t [4], M-Machine [3]</td>
<td>4 bits</td>
<td>64 bytes</td>
<td>8 + 52 + 4</td>
<td>✓ ✓ ✓ ✓</td>
<td>3 ✓ (8,3) 3</td>
</tr>
<tr>
<td>SPARC ADI [1]</td>
<td>1 bit</td>
<td>16 bytes</td>
<td>8 + 52 + 4</td>
<td>✓ ✓ ✓ ✓</td>
<td>3 ✓ (8,3) 3</td>
</tr>
<tr>
<td>CHERI 256 [9, 8]</td>
<td>1 bit</td>
<td>32 bytes</td>
<td>8 + 54 + 2</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ (8,4) 3</td>
</tr>
<tr>
<td>ARM MTE [2]</td>
<td>4 bits</td>
<td>16 bytes</td>
<td>8 + 40 + 16</td>
<td>✓ ✓ ✓ (8,3) ✓</td>
<td>2 ✓ (4,2) (8,2) 2</td>
</tr>
<tr>
<td>lowRISC [7]</td>
<td>4 bits</td>
<td>8 bytes</td>
<td>8 + 24 + 32</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model A</td>
<td>32 bits</td>
<td>64 bytes</td>
<td>1 + 31 + 32</td>
<td>✓ ✓ ✓ (8,2) (4,2) 2</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>Model B</td>
<td>46 bits</td>
<td>64 bytes</td>
<td>1 + 17 + 46</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model C</td>
<td>51 bits</td>
<td>64 bytes</td>
<td>1 + 12 + 51</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>No Tagging, SEC-DED</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>No Tagging, Chipkill</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓ ✓ ✓ (✓) ✓ ✓ ✓ ✓</td>
<td></td>
</tr>
</tbody>
</table>

Lukas Lamster — IAIK – Graz University of Technology
## Case Study: Tagged Memory Architectures

A wide range of TMAs can be implemented

Many error patterns are correctable

Combining tagging with hash-based integrity protection is feasible

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tag Size</th>
<th>Granularity</th>
<th>Bit Distribution</th>
<th>Faulty Bits in Hash (d)</th>
<th>Correctable Failure Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Parity Hash Tag</td>
<td>F1</td>
<td>F2 F3S(8,4) F3S(4,4) F3M F5S(4,4) F5S(8,..) F5M</td>
</tr>
<tr>
<td>DIFT [6], HDFI [5]</td>
<td>1 bit</td>
<td>8 bytes</td>
<td>8 + 48 + 8</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Shakti-t [4], M-Machine [3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPARC ADI [1]</td>
<td>4 bits</td>
<td>64 bytes</td>
<td>8 + 52 + 4</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CHERI 128 [9, 8]</td>
<td>1 bit</td>
<td>16 bytes</td>
<td>8 + 52 + 4</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CHERI 256 [9, 8]</td>
<td>1 bit</td>
<td>32 bytes</td>
<td>8 + 54 + 2</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>ARM MTE [2]</td>
<td>4 bits</td>
<td>16 bytes</td>
<td>8 + 40 + 16</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>lowRISC [7]</td>
<td>4 bits</td>
<td>8 bytes</td>
<td>8 + 24 + 32</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model A</td>
<td>32 bits</td>
<td>64 bytes</td>
<td>1 +31 +32</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model B</td>
<td>46 bits</td>
<td>64 bytes</td>
<td>1 +17 +46</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Model C</td>
<td>51 bits</td>
<td>64 bytes</td>
<td>1 +12 +51</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>No Tagging, SEC-DED</td>
<td>-</td>
<td>- 64 bytes</td>
<td>- - -</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>No Tagging, Chipkill</td>
<td>-</td>
<td>- 64 bytes</td>
<td>- - -</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>
- Integrity is verified on each read
- Computing the hash takes time
- This impacts the system performance
- But how big is the impact?
Integrity is verified on each read
Computing the hash takes time
This impacts the system performance
But how big is the impact?

Model system in gem5

Benchmark and measure overhead
Performance Analysis

Runtime Overhead in %

- **SPEEDY**
- **QARMA**
- **Separate Tags**

---

**SPEC Geomean**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SPEEDY</th>
<th>QARMA</th>
<th>Separate Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>0.0%</td>
<td>0.3%</td>
<td>0.2%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.4%</td>
<td>1.3%</td>
<td>0.2%</td>
</tr>
<tr>
<td>mcf</td>
<td>2.2%</td>
<td>2.0%</td>
<td>1.3%</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>5.2%</td>
<td>5.8%</td>
<td>1.3%</td>
</tr>
<tr>
<td>lbm</td>
<td>2.5%</td>
<td>2.7%</td>
<td>1.3%</td>
</tr>
<tr>
<td>omnetpp</td>
<td>0.3%</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>18.5%</td>
<td>2.7%</td>
<td>0.1%</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>2.8%</td>
<td>2.6%</td>
<td>1.3%</td>
</tr>
<tr>
<td>imagick</td>
<td>0.1%</td>
<td>0.7%</td>
<td>1.3%</td>
</tr>
<tr>
<td>leela</td>
<td>1.3%</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>exchange2</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>xz</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

**Low overall overhead**

**Much faster than regular TMA**

Lukas Lamster — IAIK – Graz University of Technology
Performance Analysis

Low overall overhead
Much faster than regular TMA

Lukas Lamster — IAIK – Graz University of Technology
# Conclusion

**Hash-based Integrity Protection and Memory Tagging**
- Replace ECC with truncated hash
- Co-locate tags in additional chip
- Eliminate storage and tag fetch overheads
- Still offer error detection and correction

**Future Work**
- Consider DDR5 on-chip ECC
- Adapt to different granularities and burst sizes
- Consider alternative hash functions
# HashTag

Hash-based Integrity Protection for Tagged Architectures

Lukas Lamster    Martin Unterguggenberger    David Schrammel    Stefan Mangard
August 10, 2023

IAIK – Graz University of Technology
References


