Practical Byte-Granular Memory Blacklisting using Califoms

Hiroshi Sasaki, Miguel A. Arroyo, Mohamed Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, Simha Sethumadhavan
Califorms

Hiroshi Sasaki, Miguel A. Arroyo, Mohamed Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, Simha Sethumadhavan
MEMORY SAFETY IS A SERIOUS PROBLEM!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.
MEMORY SAFETY IS A SERIOUS PROBLEM!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder
MEMORY SAFETY IS A SERIOUS PROBLEM!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.

The New York Times

WhatsApp Rushes to Fix Security Flaw Exposed in Hacking of Lawyer’s Phone

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder
IT’S EASY TO MAKE MISTAKES
IT’S EASY TO MAKE MISTAKES

SEGFAULT!
PREVALENCE OF MEMORY SAFETY VULNS

Memory Safety vs Non-Memory Safety CVEs

% of CVEs

0% 25% 50% 75% 100%


Patch Year

Microsoft Product CVEs

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019
PREVALENCE OF MEMORY SAFETY VULNS

Microsoft Product CVEs

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019

Google OSS-Fuzz bugs from 2016-2018.

ATTACKERS

MEMORY SAFETY
ATTACKERS PREFER MEMORY SAFETY VULNS

CVEs Exploited

% of CVEs

Patch Year

Microsoft Product Exploits

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019
RESEARCHERS TOO

MEMORY SAFETY
Current solutions aren’t practical

- Low Performance Overhead
- Low Implementation Complexity
- Minor or no software changes
CALIFORMS

- Low Overhead
- Robust Security
- Legacy Software Compatibility
- Easy to Implement
- 32-bit Compatible
MEMORY BLACKLISTING

A tripwire is a blacklisted location.

This is program data.
MEMORY BLACKLISTING

Accesses to this region trigger an exception!
MEMORY BLACKLISTING

Challenge

How to efficiently **identify** blacklisted locations?
MEMORY BLACKLISTING

Simple
1-bit per byte
MEMORY BLACKLISTING

Simple
1-bit per byte

12.5% memory overhead!

~200% runtime overhead!
 MEMORY BLACKLISTING

Califorms
1-bit per cache line
MEMORY BLACKLISTING

Califorms
1-bit per cache line

Metadata

0.2% memory overhead for 64B line!

~2-14% runtime overhead!
OUR TALK

● Califorms

● Benefits
  ○ Performance, Security

● Related work
  ○ State-of-the-art Memory Safety Mitigations

● Conclusion
CALIFORMS
MEMORY BLACKLISTING
CALIFORMS: CACHE LINE FORMATS

Our Metadata: Encoded within unused data.
CALIFORMS: CACHE LINE FORMATS

Our Metadata: Encoded within unused data.
CALIFORMS: CACHE LINE FORMATS

Our Metadata: Encoded within unused data.
CALIFORMS: CACHE LINE FORMATS

Our Metadata:  Encoded within unused data.

Blacklisted Location

Normal

12.5% Memory Overhead!
CALIFORMS: CACHE LINE FORMATS

Our Metadata: Encoded within unused data.

Blacklisted Location

Normal

Califorms

Header

A B C D E
CALIFORMS: CACHE LINE FORMATS

Our Metadata: Encoded within unused data.

Blacklisted Location

Normal

A B C D E

Califorms

Is Califomred?

Y

Header

A B C D E
CALIFORMS: CACHE LINE FORMATS

Our Metadata: Encoded within unused data.

Blacklisted Location

Normal

A B C D E

Is Califormed? Califorms
Y Header A B C D E

Normal

1 2 3 4 5 6 7 8

Is Califormed? Normal
N 1 2 3 4 5 6 7 8
CALIFORMS: FULL SYSTEM

- Microarchitecture
- Architecture Support
- Software
CALIFORMS: MICROARCHITECTURE

Diagram showing the flow of data between CPU, L1 Data, L2, and DRAM.
CALIFORMS: MICROARCHITECTURE
CALIFORMS: MICROARCHITECTURE

BitVector Califorms
No Latency Overhead
12.5% memory overhead
CALIFORMS: MICROARCHITECTURE

1-bit Califorms
1 Cycle or can be hidden
0.2% memory overhead
CALIFORMS: MICROARCHITECTURE

1-bit Califorms
Stored in ECC bits (if available)
CALIFORMS: MICROARCHITECTURE
CALIFORMS: MICROARCHITECTURE
CALIFORMS: MICROARCHITECTURE
CALIFORMS: MICROARCHITECTURE
CALIFORMS: MICROARCHITECTURE
CALIFORMS: MICROARCHITECTURE
CALIFORMS: MICROARCHITECTURE

Diagram showing the flow between CPU, L1 Data, L2, and DRAM.
CALIFORMS: FULL SYSTEM

- **Microarchitecture**
  - Cache controller.
  - L1/L2 Califorms converters.

- **Architecture Support**

- **Software**
CALIFORMS: FULL SYSTEM

- **Microarchitecture**
  - Cache controller.
  - L1/L2 Califorms converters.

- **Architecture Support**
  - A new **Blacklisting** instruction.

- **Software**
  - Compiler, memory allocator and OS extensions.
CALIFORMS: FULL SYSTEM

- **Microarchitecture**
  - Cache controller.
  - L1/L2 Califorms converters.
- **Architecture Support**
  - A new **Blacklisting** instruction.
- **Software**
  - Compiler, memory allocator and OS extensions.

For more details, please refer to our paper.
CALIFORMS: SUMMARY

● Has no false positives
  ○ Precise storage (0-64 blacklisted locations per cache line).
CALIFORMS: SUMMARY

- Has no false positives
  - Precise storage (0-64 blacklisted locations per cache line).
- Supports existing performance optimizations
  - Critical word first.
CALIFORMS: SUMMARY

- Has no false positives
  - Precise storage (0-64 blacklisted locations per cache line).
- Supports existing performance optimizations
  - Critical word first.
- Integrates into existing microarchitectures
  - Does NOT disturb coherency.
CALIFORMS
PERFORMANCE
CALIFORMS: PERFORMANCE OVERHEADS

- Hardware Overheads
- Blacklisting Overheads
CALIFORMS: PERFORMANCE OVERHEADS

- Hardware Overheads
- Blacklisting Overheads
CALIFORMS: PERFORMANCE OVERHEADS

- Hardware Overheads
- Blacklisting Overheads
CALIFORMS: INSERTION POLICIES

```c
struct A_opportunistic
{
    char c;
    char tripwire[3];
    int i;
    char buf[64];
    void (*fp)();
}
```

(1) Opportunistic

*Tripwire Insertion Policies*
CALIFORMS: INSERTION POLICIES

struct A_opportunistic {
    char c;
    char tripwire[3];
    int i;
    char buf[64];
    void (*fp)();
}

struct A_full {
    char tripwire[2];
    char c;
    char tripwire[1];
    int i;
    char tripwire[3];
    char buf[64];
    char tripwire[2];
    void (*fp)();
    char tripwire[1];
}

(1) Opportunistic

(2) Full

Tripwire Insertion Policies
CALIFORMS: INSERTION POLICIES

(1) Opportunistic

(2) Full

(3) Intelligent

Tripwire Insertion Policies
CALIFORMS
EVALUATION METHODOLOGY
CALIFORMS: EVALUATION METHODOLOGY

- Emulating the Blacklisting instruction
  - Inserting dummy stores to blacklisted bytes.
CALIFORMS: EVALUATION METHODOLOGY

- Emulating the Blacklisting instruction
  - Inserting dummy stores to blacklisted bytes.
- NO simulations
  - Taking results from a real Skylake-based machine.
CALIFORMS: EVALUATION METHODOLOGY

- Emulating the Blacklisting instruction
  - Inserting dummy stores to blacklisted bytes.
- NO simulations
  - Taking results from a real Skylake-based machine.
- Using SPEC2006 benchmarks with reference inputs
  - Running experiments to completion.
CALIFORMS: POLICIES OVERHEADS

Opportunistic (BLOC)

Slowdown

-10.0%
-0.0%
20.0%
30.0%
40.0%
50.0%

-astar  bzip2  gobmk  h264ref  hminer  ltm  libquantum  mcf  milc  namd  perlbench  povray  sjeng  soplex  sphinx3  xalancbmk

Slowdown

7.9%
20.0%
18.0%
16.0%
14.0%
12.0%
10.0%
8.0%
6.0%
4.0%
2.0%
0.0%

80.3%
CALIFORMS: POLICIES OVERHEADS
CALIFORMS: POLICIES OVERHEADS

![Graph showing slowdown comparison between Opportunistic (BLOC), Full (BLOC), and Intelligent (BLOC).]

- Opportunistic (BLOC)
- Full (BLOC)
- Intelligent (BLOC)

- Slowdown
- astar: 80.3%
- bzip2: 85.2%
- gobmk: 7.9%
- h264ref: 13.9%
- hmmmer: 1.5%
- ibm: 0.0%
- libquantum: 0.0%
- mcf: 0.0%
- milc: 0.0%
- namd: 0.0%
- perlbench: 0.0%
- povray: 0.0%
- sjeng: 0.0%
- soplex: 0.0%
- sphinx3: 0.0%
- xalancbmk: 0.0%

- AMean
- 7.9%
- 13.9%
- 1.5%
CALIFORMS: POLICIES OVERHEADS

(1) Opportunistic

(2) Full

(3) Intelligent

Provides the best performance-security tradeoff.

Tripwire Insertion Policies
CALIFORMS
SECURITY BENEFITS
CALIFORMS: SECURITY BENEFITS

- Blacklisted locations must be placed *unpredictably*. 
CALIFORMS: SECURITY BENEFITS

- Blacklisted locations must be placed *unpredictably*.
**CALIFORMS: SECURITY BENEFITS**

This is the **best case** for me. Only one object!

We insert **up to 7** tripwires between fields

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int pin;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CALIFORMS: SECURITY BENEFITS

Let me *guess* where the field is.

Field may be anywhere in this region

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int pin;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let me *guess* where the field is.

Field may be anywhere in this region

```
int pin;
```
CALIFORMS: SECURITY BENEFITS

```
int pin;
```

```
int creditcard;
```
CALIFORMS: SECURITY BENEFITS

The more I need to disclose the \textbf{harder} it is!

where $n$ is the number of fields to be disclosed.
CALIFORMS: SECURITY BENEFITS

Can I disable blacklisted memory?
Can I disable blacklisted memory?

They would first need to bypass Califoms.
MEMORY SCANNING ATTACK

WITH CALIFORMS
EXAMPLE: MEMORY SCANNING ATTACK

Memory

- int id;
- short age;
- char tripwireA[2];
- char username[28];
- char tripwireB[4];
- char password[12];

Unallocated

- char *Msg = “Wrong Password!”;

Attacker Controlled Pointer

- char *p;
EXAMPLE: MEMORY SCANNING ATTACK

Memory

<table>
<thead>
<tr>
<th>int id;</th>
</tr>
</thead>
<tbody>
<tr>
<td>short age;</td>
</tr>
<tr>
<td>char tripwireA[2];</td>
</tr>
<tr>
<td>char username[28];</td>
</tr>
<tr>
<td>char tripwireB[4];</td>
</tr>
<tr>
<td>char password[12];</td>
</tr>
</tbody>
</table>

Unallocated

char *p;

Attacker Controlled Pointer

Where can I find the password?

char *Msg = “Wrong Password!”;
EXAMPLE: MEMORY SCANNING ATTACK

These memory regions are blacklisted by Califoms!

char *p;

Attacker Controlled Pointer

char *Msg = "Wrong Password!";
**EXAMPLE:** MEMORY SCANNING ATTACK

```c
char *p;
Attacker Controlled Pointer
```

```
int id;
short age;
char tripwireA[2];
char username[28];
char tripwireB[4];
char password[12];
```

```
char *Msg = "Wrong Password!";
```

Oh no! I’ve triggered an **exception**!
EXAMPLE: MEMORY SCANNING ATTACK

```c
int id;
short age;
char tripwireA[2];
char username[28];
char tripwireB[4];
char password[12];

char *Msg = "Wrong Password!";
```

Califorms provides **intra-object protection!**
RELATED WORK
### RELATED WORK

<table>
<thead>
<tr>
<th>Technique</th>
<th>Program Memory Footprint</th>
<th>Performance Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base &amp; Bound</td>
<td>$\propto$ # of pointers</td>
<td>$\propto$ # of pointer dereferences</td>
</tr>
</tbody>
</table>
## RELATED WORK

<table>
<thead>
<tr>
<th>Technique</th>
<th>Program Memory Footprint</th>
<th>Performance Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base &amp; Bound</td>
<td>∝ # of pointers</td>
<td>∝ # of pointer dereferences</td>
</tr>
<tr>
<td>FAT Pointers</td>
<td>∝ # of pointers and physical mem.</td>
<td>∝ # of pointer operations</td>
</tr>
</tbody>
</table>
## RELATED WORK

<table>
<thead>
<tr>
<th>Technique</th>
<th>Program Memory Footprint</th>
<th>Performance Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base &amp; Bound</td>
<td>$\propto$ # of pointers</td>
<td>$\propto$ # of pointer dereferences</td>
</tr>
<tr>
<td>FAT Pointers</td>
<td>$\propto$ # of pointers and physical mem.</td>
<td>$\propto$ # of pointer operations</td>
</tr>
<tr>
<td>Califorms</td>
<td>$\propto$ # blacklisted locations</td>
<td>$\propto$ # of blacklisting instructions</td>
</tr>
</tbody>
</table>
CONCLUSION

● Californs can be applied to non 64-bit systems (e.g. IoT, CPS, etc).

● Californs’ blacklisting is an efficient solution to memory safety:
  ○ Is easy to implement.
  ○ Has low overheads.
  ○ Offers robust security.
CONCLUSION

- Califorms can be applied to non 64-bit systems (e.g IoT, CPS, etc).

- Califorms’ blacklisting is an efficient solution to memory safety:
  - Is easy to implement.
  - Has low overheads.
  - Offers robust security.

QUESTIONS?

Stop by during the poster session to chat!
2:50-4:00pm
BACKUPS
CALIFORMS: ENCODING SCHEMES

califorms-bitvector

califorms-sentinel
CALIFORMS: HARDWARE DIAGRAM
CALIFORMS: HARDWARE DIAGRAM

Blocks F (if 2-bit matches 00, 01, 10 or 11) and G (if 6-bit matches the sentinel) are constructed using logic comparators.
CALIFORMS: CONSERVATIVE ANALYSIS

Slowdown with additional one-cycle access latency for both L2 and L3 caches.
## CALIFORMS: HARDWARE PERFORMANCE

<table>
<thead>
<tr>
<th>L1 Califorms</th>
<th>Area (GE)</th>
<th>Delay (ns)</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Overheads</td>
<td>[+18.69%] 412,263.87</td>
<td>[+1.85%] 1.65</td>
<td>[+2.12%] 16.17</td>
</tr>
<tr>
<td>Fill Module</td>
<td>8,957.16</td>
<td>1.43</td>
<td>0.18</td>
</tr>
<tr>
<td>Spill Module</td>
<td>34,561.80</td>
<td>5.50</td>
<td>0.52</td>
</tr>
</tbody>
</table>
**CALIFORMS: OPPORTUNISTIC POLICY**

Normally Occurring Dead Bytes

\[
\text{Struct density} = \sum_i \frac{\text{#fields}\text{sizeof(field}_i\text{)}}{\text{sizeof(struct)}}
\]