CCFI

Cryptographically Enforced Control Flow Integrity

Konrad Weiss

December 5th 2016
Outline

1. Introduction
2. Design
3. Implementation
4. Evaluation
5. Conclusion
Introduction
The current problem with CFI

Compile time build information and checks during runtime

- Only few checks during runtime
Introduction
The current problem with CFI

Compile time build information and checks during runtime
- Only few checks during runtime
- Static analysis limits runtime checks
  - CFG
  - classification
Introduction
The current problem with CFI

Compile time build information and checks during runtime

- Only few checks during runtime
- Static analysis limits runtime checks
  - CFG
  - classification
- Tackle the symptoms and not the problem
  - React to known attacks
  - Control flow hijacking
Introduction
The current problem with CFI

Compile time build information and checks during runtime

- Only few checks during runtime
- Static analysis limits runtime checks
  - CFG
  - classification
- Tackle the symptoms and not the problem
  - React to known attacks
  - Control flow hijacking
- Create what the attacker needs
The **ultimate attacker** regarding CFI

- Read arbitrary areas of memory
- Overwrite all control flow elements
- User mode, not kernel mode
Compute a MAC for every control flow element
Introduction
The solution sketch

- Compute a MAC for every control flow element
- Use hardware supported AES
- Compute a MAC for every control flow element
- Use hardware supported AES
- Use compile and runtime information
Compute a MAC for every control flow element
Use hardware supported AES
Use compile and runtime information
On every legit store/load: compute/verify MAC
Introduction

The solution sketch

- Compute a MAC for every control flow element
- Use hardware supported AES
- Use compile and runtime information
- On every legit store/load: compute/verify MAC
- Keep the key secret from any attacker
Outline

1. Introduction
2. Design
3. Implementation
4. Evaluation
5. Conclusion
Design
AES-128 an key protection

- MAC = 1 AES-128 block
Design
AES-128 key protection

- MAC = 1 AES-128 block
- Key generated on program start

Assumptions:
1. Key is never leaked into memory by program code
2. Attacker has to break CCFI before leaking the key
3. Kernel does not leak the secret key
Design

AES-128 an key protection

- MAC = 1 AES-128 block
- Key generated on program start
- stored in registers reserved by CCFI compiler
Design
AES-128 an key protection

- MAC = 1 AES-128 block
- Key generated on program start
- stored in registers reserved by CCFI compiler

Assumptions
- (1) Key is never leaked into memory by program code
Design
AES-128 an key protection

- MAC = 1 AES-128 block
- Key generated on program start
- stored in registers reserved by CCFI compiler

Assumptions
(1) Key is never leaked into memory by program code
(2) Attacker has to break CCFI before leaking the key
Design

AES-128 key protection

- MAC = 1 AES-128 block
- Key generated on program start
- stored in registers reserved by CCFI compiler

Assumptions

1. Key is never leaked into memory by program code
2. Attacker has to break CCFI before leaking the key
3. Kernel does not leak the secret key
Design
What will be protected

All relevant control flow objects

- Function pointers
Design
What will be protected

All relevant control flow objects

- Function pointers
- Return addresses
Design
What will be protected

All relevant control flow objects

- Function pointers
- Return addresses
- Frame pointers
All relevant control flow objects

- Function pointers
- Return addresses
- Frame pointers
- vtable pointers
Introduction  Design  Implementation  Evaluation  Conclusion

Design
What will be protected

All relevant control flow objects
- Function pointers
- Return addresses
- Frame pointers
- vtable pointers
- Exception handlers
Design
MAC storage and computation

Pointer:
Design
MAC storage and computation

Pointer: 64 bit pointer
Design
MAC storage and computation

Pointer: 0x0000 (16-bits)  user-space pointer (48-bits)
Design
MAC storage and computation

Pointer: 0x0000 (16-bits) user-space pointer (48-bits)

Mac Input: 128 bit block
Design
MAC storage and computation

Pointer:  0x0000 (16-bits) user-space pointer (48-bits)

Mac Input:  pointer (48-bits)
Design

MAC storage and computation

Pointer:
- 0x0000 (16-bits)
- user-space pointer (48-bits)

Mac Input:
- pointer (48-bits)
- class (80-bits)
Design
Pointer classes

Mac Input: class (80-bits)

Design

Pointer classes

Mac Input: class (80-bits)
Design

Pointer classes

Introduction: Design

Implementation: Evaluation

Conclusion

Mac Input: class (80-bits)

\[
\text{class} :\begin{cases} 
\{0, \text{hash of type, address}\} \\
\{1, \text{frame address}\} \\
\{2, \text{method ptr., address}\} \\
\{3, \text{address}\}
\end{cases}
\]

- Function pointers
- Return addresses
- Method pointers
- Vtable pointers
Design

Pointer classes

Mac Input:

\[
\text{class} \ := \ \begin{cases}
\{0, \text{hash of type, address}\} \\
\{1, \text{frame address}\} \\
\{2, \text{method ptr., address}\} \\
\{3, \text{address}\}
\end{cases}
\]

- Domain separation

Function pointers
Return addresses
Method pointers
Vtable pointers
Design

Pointer classes

Mac Input: class (80-bits)

\[
\text{class} := \begin{cases} 
0, \text{hash of type, address} \\
1, \text{frame address} \\
2, \text{method ptr., address} \\
3, \text{address} 
\end{cases}
\]

- Domain separation
- Type based classification

Function pointers
- Return addresses
- Method pointers
- Vtable pointers

Konrad Weiss — CCFI
Design
Pointer classes

Mac Input: class (80-bits)

\[
\begin{align*}
\text{class} & := \\
& \begin{cases}
0, \text{hash of type, address} \\
1, \text{frame address} \\
2, \text{method ptr., address} \\
3, \text{address}
\end{cases}
\end{align*}
\]

- Function pointers
- Return addresses
- Method pointers
- Vtable pointers

- Domain separation
- Type based classification
- Replay protection
Memory address acts as a **nonce** for MAC
Memory address acts as a **nonce** for MAC

Those addresses are kept random
Design
Stack and Heap randomization

- Memory address acts as a **nonce** for MAC
- Those addresses are kept random
- 4 bits of randomness are injected
Design
Stack and Heap randomization

- Memory address acts as a **nonce** for MAC
- Those addresses are kept random
- 4 bits of randomness are injected
  - `alloca` to displace the new stack frame
Memory address acts as a **nonce** for MAC

Those addresses are kept random

4 bits of randomness are injected

- `alloca` to displace the new stack frame
- `malloc` adds the randomness in heap allocation
There are values that still have to be protected:

- Global offset table (GOT)
- Global destructors (.dtors)
There are values that still have to be protected:

- Global offset table (GOT)
- Global destructors (.dtors)

Protected by using RELRO:

reallocates GOT and .dtors and marks them read only
Implementation
Platform and Compiler

- Compiler build on top of Clang/LLVM
- supports x86_64 and (AMD64 SysV)
- Tested on FreeBSD
Implementation

ABI changes

<table>
<thead>
<tr>
<th>Registers</th>
<th>SysV ABI</th>
<th>CCFI ABI</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmm0-xmm3</td>
<td>Arguments</td>
<td>Arguments</td>
</tr>
<tr>
<td>xmm4</td>
<td>Arguments</td>
<td>Temporary</td>
</tr>
<tr>
<td>xmm5-xmm7</td>
<td>Arguments</td>
<td>Expanded Key</td>
</tr>
<tr>
<td>xmm8-xmm15</td>
<td>Temporary</td>
<td>Expanded Key</td>
</tr>
<tr>
<td>xmm16-xmm31</td>
<td>in AVX-512</td>
<td>in AVX-512</td>
</tr>
</tbody>
</table>

Table: XMM register changes in the CCFI ABI [1]

- AES computation uses an expanded Key
Implementation

ABI changes

<table>
<thead>
<tr>
<th>Registers</th>
<th>SysV ABI</th>
<th>CCFI ABI</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmm0-xmm3</td>
<td>Arguments</td>
<td>Arguments</td>
</tr>
<tr>
<td>xmm4</td>
<td>Arguments</td>
<td>Temporary</td>
</tr>
<tr>
<td><strong>xmm5-xmm7</strong></td>
<td>Arguments</td>
<td><strong>Expanded Key</strong></td>
</tr>
<tr>
<td><strong>xmm8-xmm15</strong></td>
<td>Temporary</td>
<td><strong>Expanded Key</strong></td>
</tr>
<tr>
<td>xmm16-xmm31</td>
<td>in AVX-512</td>
<td>in AVX-512</td>
</tr>
</tbody>
</table>

Table: XMM register changes in the CCFI ABI [1]

- AES computation uses an expanded Key
- 11 128-bit registers are necessary
Implementation

ABI changes

<table>
<thead>
<tr>
<th>Registers</th>
<th>SysV ABI</th>
<th>CCFI ABI</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmm0-xmm3</td>
<td>Arguments</td>
<td>Arguments</td>
</tr>
<tr>
<td>xmm4</td>
<td>Arguments</td>
<td>Temporary</td>
</tr>
<tr>
<td>xmm5-xmm7</td>
<td>Arguments</td>
<td>Expanded Key</td>
</tr>
<tr>
<td>xmm8-xmm15</td>
<td>Temporary</td>
<td>Expanded Key</td>
</tr>
<tr>
<td>xmm16-xmm31</td>
<td>in AVX-512</td>
<td>in AVX-512</td>
</tr>
</tbody>
</table>

Table: XMM register changes in the CCFI ABI [1]

- AES computation uses an expanded Key
- 11 128-bit registers are necessary
- Other instructions are blocked from using them
## Implementation

### ABI changes

<table>
<thead>
<tr>
<th>Registers</th>
<th>SysV ABI</th>
<th>CCFI ABI</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmm0-xmm3</td>
<td>Arguments</td>
<td>Arguments</td>
</tr>
<tr>
<td>xmm4</td>
<td>Arguments</td>
<td>Temporary</td>
</tr>
<tr>
<td>xmm5-xmm7</td>
<td>Arguments</td>
<td>Expanded Key</td>
</tr>
<tr>
<td>xmm8-xmm15</td>
<td>Temporary</td>
<td>Expanded Key</td>
</tr>
<tr>
<td><strong>xmm16-xmm31</strong></td>
<td><strong>in AVX-512</strong></td>
<td><strong>in AVX-512</strong></td>
</tr>
</tbody>
</table>

**Table:** XMM register changes in the CCFI ABI [1]

- AES computation uses an expanded Key
- 11 128-bit registers are necessary
- Other instructions are blocked from using them
- XMM registers will be doubled in the future
PP compiler pass identifies store/load instructions
Implementation

General operation

- PP compiler pass identifies store/load instructions
- SP compiler pass finds prologue and epilogue of calls
PP compiler pass identifies store/load instructions
SP compiler pass finds prologue and epilogue of calls
Adds necessary `macptr` and `checkptr`
Implementation
General operation

- PP compiler pass identifies store/load instructions
- SP compiler pass finds prologue and epilogue of calls
- Adds necessary `macptr` and `checkptr`
- `.got` and `.dtors` are reordered and protected
Implementation

General operation

- PP compiler pass identifies store/load instructions
- SP compiler pass finds prologue and epilogue of calls
- Adds necessary `macptr` and `checkptr`
- `.got` and `.dtors` are reordered and protected
- Memory randomization during allocation
Implementation

General operation

- PP compiler pass identifies store/load instructions
- SP compiler pass finds prologue and epilogue of calls
- Adds necessary `macptr` and `checkptr`
- `.got` and `.dtors` are reordered and protected
- Memory randomization during allocation
- Execution of `macptr` and `checkptr`
MACs are stored in a **Hash table**
MACs are stored in a **Hash table**

- loads and stores of FPs are identified
MACs are stored in a *Hash table*

- loads and stores of FPs are identified
- Find nested function pointers
  - recursively walk structures, arrays and vectors
MACs are stored in a **Hash table**
- loads and stores of FPs are identified
- Find nested function pointers
  - recursively walk structures, arrays and vectors
- `=` or `memcpy` of known FPs trigger MAC verification and recomputation
MACs are stored in a **Hash table**

- loads and stores of FPs are identified
- Find nested function pointers
  - recursively walk structures, arrays and vectors
- `=` or `memcpy` of known FPs trigger MAC verification and recomputation
- Generates constructors to MAC all global FPs on start up
MACs are stored in a Hash table
- loads and stores of FPs are identified
- Find nested function pointers
  - recursively walk structures, arrays and vectors
- = or memcpy of known FPs trigger MAC verification and recomputation
- Generates constructors to MAC all global FPs on start up
- Also check and verify structures from/to libc
MAC is stored in **local variable**
- MAC is stored in **local variable**
- Prologue generates and stores the MAC
- MAC is stored in **local variable**
- Prologue generates and stores the MAC
- Epilogue verifies the MAC
MAC is stored in **local variable**

- Prologue generates and stores the MAC
- Epilogue verifies the MAC
- On fails, ret address and frame pointer contain `0x00`
Some functions do not call other functions
Implementation
Stack leaf optimization

- Some functions do not call other functions
- Number of such leafs is big
- Some functions do not call other functions
- Number of such leaves is big
- Avoid expensive mac computation
 Implementation
 Stack leaf optimization

- Some functions do not call other functions
- Number of such leafs is big
- Avoid expensive mac computation
- before leaf call: store return address and frame pointer in XMM4
Some functions do not call other functions

Number of such leafs is big

Avoid expensive mac computation

Before leaf call: store return address and frame pointer in XMM4

Compared after call
Other elements can still influence control flow

- **Indirect function pointers**
  - e.g. struct pointers
  - cannot always be identified
  - manipulating struct pointers can beat address based classification
  - function signature type is still protected
  - Idea: Identify and MAC sensitive struct pointers
Other elements can still influence control flow

- **Indirect function pointers**
  - e.g. struct pointers
  - cannot always be identified
  - manipulating struct pointers can beat address based classification
  - function signature type is still protected
  - Idea: Identify and MAC sensitive struct pointers

- **Data flow Attacks**
  - CCFI could even protect against **Data flow Attacks**
  - e.g. variables for branches or index into switch statements
  - Annotate sensitive variables
  - Compiler pass then inserts necessary verification and checks
1 Introduction
2 Design
3 Implementation
4 Evaluation
5 Conclusion
Evaluation
Application compatibility

- Some code breaks MAC computation and verification
Some code breaks MAC computation and verification

- FP or structures containing FP are casted to `void*`
Some code breaks MAC computation and verification

- FP or structures containing FP are casted to `void*`
- then copied via `memcpy` or assignment =
Some code breaks MAC computation and verification

- FP or structures containing FP are casted to \texttt{void*}
- then copied via \texttt{memcpy} or assignment =
- Address in MAC does not match
Some code breaks MAC computation and verification

- FP or structures containing FP are casted to `void*`
- then copied via `memcpy` or assignment =
- Address in MAC does not match
- Mostly in custom array implementations
Some code breaks MAC computation and verification

- FP or structures containing FP are casted to `void*`
- then copied via `memcpy` or assignment `=`
- Address in MAC does not match
- Mostly in custom array implementations

- Manual changes to the binaries or code is necessary
Some code breaks MAC computation and verification

- FP or structures containing FP are casted to `void*`
- then copied via `memcpy` or assignment `=`
- Address in MAC does not match
- Mostly in custom array implementations

Manual changes to the binaries or code is necessary

- 21 libraries, 5 servers and SPEC CINT2006 were compiled
Some code breaks MAC computation and verification
- FP or structures containing FP are casted to `void*`
- then copied via `memcpy` or assignment =
- Address in MAC does not match
- Mostly in custom array implementations

Manual changes to the binaries or code is necessary
- 21 libraries, 5 servers and SPEC CINT2006 were compiled
- 1 line in nginx, 2 in libapr and a few in perlbench
**Evaluation**

MAC computation overhead (1)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Baseline</th>
<th>Ptr. Prot.</th>
<th>CCFI</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Func. call</em></td>
<td>7</td>
<td>-</td>
<td>70</td>
</tr>
<tr>
<td><em>Fptr. call</em></td>
<td>7</td>
<td>50</td>
<td>153</td>
</tr>
<tr>
<td><em>Mthd. call</em></td>
<td>8</td>
<td>53</td>
<td>156</td>
</tr>
<tr>
<td><em>Vptr. call</em></td>
<td>17</td>
<td>60</td>
<td>164</td>
</tr>
</tbody>
</table>

**Table:** Round-trip time of noop function in CPU cycles [1]
**Table:** Round-trip time of noop function in CPU cycles [1]
### Evaluation

MAC computation overhead (1)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Baseline</th>
<th>Ptr. Prot.</th>
<th>CCFI</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Func. call</em></td>
<td>7</td>
<td>-</td>
<td>70</td>
</tr>
<tr>
<td><em>Fptr. call</em></td>
<td>7</td>
<td>50</td>
<td>153</td>
</tr>
<tr>
<td><em>Mthd. call</em></td>
<td>8</td>
<td>53</td>
<td>156</td>
</tr>
<tr>
<td><em>Vptr. call</em></td>
<td>17</td>
<td>60</td>
<td>164</td>
</tr>
</tbody>
</table>

**Table:** Round-trip time of noop function in CPU cycles [1]
### Evaluation

MAC computation overhead (1)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Baseline</th>
<th>Ptr. Prot.</th>
<th>CCFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Func. call</td>
<td>7</td>
<td>-</td>
<td>70</td>
</tr>
<tr>
<td>Fptr. call</td>
<td>7</td>
<td>50</td>
<td>153</td>
</tr>
<tr>
<td>Mthd. call</td>
<td>8</td>
<td>53</td>
<td>156</td>
</tr>
<tr>
<td>Vptr. call</td>
<td>17</td>
<td>60</td>
<td>164</td>
</tr>
</tbody>
</table>

**Table:** Round-trip time of noop function in CPU cycles [1]
Evaluation

MAC computation overhead (2)

Figure: SPEC2006 results: unoptimized left and optimized right
**Evaluation**

**Server performance**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Baseline</th>
<th>CCFI</th>
<th>R. decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx (https)</td>
<td>207</td>
<td>128</td>
<td>38 %</td>
</tr>
<tr>
<td>Nginx</td>
<td>16482</td>
<td>14103</td>
<td>14.4 %</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>22714</td>
<td>18516</td>
<td>18.5 %</td>
</tr>
<tr>
<td>Apache</td>
<td>25305</td>
<td>24537</td>
<td>3 %</td>
</tr>
<tr>
<td>memcached</td>
<td>283403</td>
<td>276006</td>
<td>2.6 %</td>
</tr>
<tr>
<td>redis</td>
<td>107527</td>
<td>33496</td>
<td>17.7 %</td>
</tr>
</tbody>
</table>

**Table**: Web and cache server request throughput [1]
Evaluation
Server performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Baseline</th>
<th>CCFI</th>
<th>R. decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx (https)</td>
<td>207</td>
<td>128</td>
<td>38 %</td>
</tr>
<tr>
<td>Nginx</td>
<td>16482</td>
<td>14103</td>
<td>14.4 %</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>22714</td>
<td>18516</td>
<td>18.5 %</td>
</tr>
<tr>
<td>Apache</td>
<td>25305</td>
<td>24537</td>
<td>3 %</td>
</tr>
<tr>
<td>memcached</td>
<td>283403</td>
<td>276006</td>
<td>2.6 %</td>
</tr>
<tr>
<td>redis</td>
<td>107527</td>
<td>33496</td>
<td>17.7 %</td>
</tr>
</tbody>
</table>

Table: Web and cache server request throughput [1]
## Evaluation

### Server performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Baseline</th>
<th>CCFI</th>
<th>R. decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx (https)</td>
<td>207</td>
<td>128</td>
<td>38 %</td>
</tr>
<tr>
<td>Nginx</td>
<td>16482</td>
<td>14103</td>
<td>14.4 %</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>22714</td>
<td>18516</td>
<td>18.5 %</td>
</tr>
<tr>
<td>Apache</td>
<td>25305</td>
<td>24537</td>
<td>3 %</td>
</tr>
<tr>
<td>memcached</td>
<td>283403</td>
<td>276006</td>
<td>2.6 %</td>
</tr>
<tr>
<td>redis</td>
<td>107527</td>
<td>33496</td>
<td>17.7 %</td>
</tr>
</tbody>
</table>

**Table:** Web and cache server request throughput [1]
## Evaluation

### Server performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Baseline</th>
<th>CCFI</th>
<th>R. decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx (https)</td>
<td>207</td>
<td>128</td>
<td>38 %</td>
</tr>
<tr>
<td>Nginx</td>
<td>16482</td>
<td>14103</td>
<td>14.4 %</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>22714</td>
<td>18516</td>
<td>18.5 %</td>
</tr>
<tr>
<td>Apache</td>
<td>25305</td>
<td>24537</td>
<td>3 %</td>
</tr>
<tr>
<td>memcached</td>
<td>283403</td>
<td>276006</td>
<td>2.6 %</td>
</tr>
<tr>
<td>redis</td>
<td>107527</td>
<td>33496</td>
<td>17.7 %</td>
</tr>
</tbody>
</table>

**Table:** Web and cache server request throughput [1]
## Evaluation

Server performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Baseline</th>
<th>CCFI</th>
<th>R. decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx (https)</td>
<td>207</td>
<td>128</td>
<td>38 %</td>
</tr>
<tr>
<td>Nginx</td>
<td>16482</td>
<td>14103</td>
<td>14.4 %</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>22714</td>
<td>18516</td>
<td>18.5 %</td>
</tr>
<tr>
<td>Apache</td>
<td>25305</td>
<td>24537</td>
<td>3 %</td>
</tr>
<tr>
<td>memcached</td>
<td>283403</td>
<td>276006</td>
<td>2.6 %</td>
</tr>
<tr>
<td>redis</td>
<td>107527</td>
<td>33496</td>
<td>17.7 %</td>
</tr>
</tbody>
</table>

**Table**: Web and cache server request throughput [1]
Conclusions

- Attacks on writable control flow elements
Conclusion

Attack mitigation

- Attacks on writable control flow elements
- After complete memory leakage
Conclusion
Attack mitigation

- Attacks on writable control flow elements
- After complete memory leakage
- Even **COOP-attacks**
Conclusion

Attack mitigation

- Attacks on writable control flow elements
- After complete memory leakage
- Even **COOP-attacks**
- Replay of pointers/MACs only:
Conclusion

Attack mitigation

- Attacks on writable control flow elements
- After complete memory leakage
- Even **COOP-attacks**
- Replay of pointers/MACs only:
  - of the same type and address
Conclusion

Attack mitigation

- Attacks on writable control flow elements
- After complete memory leakage
- Even COOP-attacks
- Replay of pointers/MACs only:
  - of the same type and address
  - or after a class collision
Conclusions

- Attacks on writable control flow elements
- After complete memory leakage
- Even COOP-attacks
- Replay of pointers/MACs only:
  - of the same type and address
  - or after a class collision

But:
Conclusion

Attack mitigation

- Attacks on writable control flow elements
- After complete memory leakage
- Even **COOP-attacks**
- Replay of pointers/MACs only:
  - of the same type and address
  - or after a class collision

But:
- only of the user privileged mode
Conclusion
Attack mitigation

- Attacks on writable control flow elements
- After complete memory leakage
- Even **COOP-attacks**
- Replay of pointers/MACs only:
  - of the same type and address
  - or after a class collision

But:
- only of the user privileged mode
- not the kernel mode
## Performance numbers

- Avg runtime increase of **52\%**
- omnetpp with 5 times overhead was reduced to 3.5 due to optimization
- **3-18\%** reduction of server throughput
Performance numbers

- Avg runtime increase of **52%**
- omnetpp with 5 times overhead was reduced to 3.5 due to optimization
- **3-18 %** reduction of server throughput

Outlook

- Promising for network servers
- will profit from AVX-512 extension
- hardware supported checkptr/macptr could cover all of the overhead
Questions?