SoK: Shining Light on Shadow Stacks

Nathan Burow, Xinpeng Zhang, Mathias Payer
Control-Flow Hijacking (CFH)

- Microsoft: 70% of bugs are memory corruptions
- Control and Data Planes are interleaved
- Memory corruption $\rightarrow$ Control-Flow Hijacking
Control-Flow Hijacking (CFH)

• Microsoft: 70% of bugs are memory corruptions

• Control and Data Planes are interleaved

• Memory corruption → Control-Flow Hijacking

Data  Code Pointer
Forward Edge

- Function pointers; virtual calls
- Control-Flow Integrity (CFI) – statically calculates target sets
Forward Edge

• Function pointers; virtual calls

• Control-Flow Integrity (CFI) – statically calculates target sets
Forward Edge

• Function pointers; virtual calls

• Control-Flow Integrity (CFI) – statically calculates target sets

\[
\text{fptr()} \rightarrow \rightarrow \rightarrow
\]
Backward Edge

• Return Instructions
• Does CFI style analysis work?
Backward Edge

- Return Instructions
- Does CFI style analysis work?
Backward Edge

• Return Instructions
• Does CFI style analysis work?

NO
Backward Edge

- CFI style target sets include every call site for the function
- Target sets are too large to provide meaningful protection

Security requires integrity for return addresses!
CFH Mitigation Today

• Seminal CFI paper by Abadi et. al. called for shadow stack
• See Burow et al CSUR 2017[1]
• Deployed versions by Microsoft / Google only cover forward edge

No equally strong defense for backward edge!

Shadow Stacks

• Separate return addresses from data plane

• Provide integrity protection for return addresses

• Performant and highly compatible

Need to deploy Shadow Stack with CFI!
Control-Flow Hijacking Illustrated

Program Stack

Return Address

Stack Canary

Pointer

Array
Control-Flow Hijacking Illustrated

Program Stack

- Return Address
- Stack Canary
- Pointer
- Array

The illustration shows a stack with various elements, including a return address, stack canary, pointer, and array. This diagram is used to illustrate how control-flow hijacking can be performed by manipulating the stack.
Control-Flow Hijacking Illustrated

Program Stack

- Return Address
- Stack Canary

Pointer

Array
Control-Flow Hijacking Illustrated

Program Stack

- Return Address
- Stack Canary

Pointer

Array
Control-Flow Hijacking Illustrated

Program Stack

- Return Address
- Stack Canary

Pointer

Array
Control-Flow Hijacking Illustrated
What is a Shadow Stack?

Program Stack

- foo()
  - Return Address
  - 
  - Return Address

bar()

Shadow Stack

- Return Address
  - 
  - Return Address
Shadow Stack Defense

![Diagram showing the Shadow Stack Defense mechanism. The program stack contains ROP Pointer and Stack Canary. The shadow RA is empty.]
Shadow Stack Defense

Program Stack

- ROP Pointer
- Stack Canary

Shadow Stack

- Shadow RA

Pointer

Array
Shadow Stack Defense

Program Stack

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROP Pointer</td>
<td>Stack Canary</td>
</tr>
</tbody>
</table>

Shadow Stack

<table>
<thead>
<tr>
<th>Shadow RA</th>
</tr>
</thead>
</table>
Advantages of Shadow Stacks

• Know at runtime what function you were called from

• Dynamic defense – does **NOT** rely on static analysis

• Separates code and data planes for backward edges

**Fully precise backward edge protection!**
Shadow Stack Design Space

Direct Mapping [1]

Indirect Mapping [2],[3]

Recommended Shadow Stack

- Indirect mapping
- Use a general purpose register for shadow stack pointer

Optimal performance and high compatibility!
Recommended Mapping

• Indirect Mapping

• As performant as direct mapping

• Minimizes memory overhead

Fastest mapping has lowest memory overhead!
Recommended Encoding

• Use general purpose (GP) register for shadow stack pointer
• Does not increase register pressure
• Significant optimization for shadow stacks

Dedicating a register to the shadow stack pointer is an effective optimization!
Compatibility of Recommended Shadow Stack

• Threading: fully supported. GP registers are thread local

• Stack Unwinding: provide instrumented setjmp / longjmp

• Unprotected Code: save and restore shadow stack pointer

Support all applications and incremental deployment!
Intra-Process Memory Isolation

• Shadow Stack splits code and data planes

• Enables integrity enforcement by isolating return addresses

Shadow Stacks enable code pointer integrity for return addresses!
Intra-Process Memory Isolation

- Software based randomization defense are defeasible
- Intel MPX uses bounds checks for isolation, moderate performance
- Intel MPK changes permissions of pages, slow performance

None of these are fully satisfactory. Tagged architectures are a promising new approach.
### SPEC CPU2006 Performance Evaluation

<table>
<thead>
<tr>
<th>Shadow Stack</th>
<th>Geometric Mean</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>5.78%</td>
<td>38.68%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Recommended</td>
<td>3.65%</td>
<td>9.70%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
## SPEC CPU2006 Performance Evaluation

<table>
<thead>
<tr>
<th>Shadow Stack</th>
<th>Geometric Mean</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>5.78%</td>
<td>38.68%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Recommended</td>
<td>3.65%</td>
<td>9.70%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
## SPEC CPU2006 – Integrity Enforcement

<table>
<thead>
<tr>
<th>Integrity Scheme</th>
<th>Geometric Mean</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomization</td>
<td>4.31%</td>
<td>13.68%</td>
<td>0.00%</td>
</tr>
<tr>
<td>MPX</td>
<td>12.12%</td>
<td>33.02%</td>
<td>2.47%</td>
</tr>
<tr>
<td>MPK</td>
<td>61.18%</td>
<td>419.81%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
Conclusion

- Stack remains vulnerable to code reuse attacks
- Need to separate return addresses from data plane
- Recommend a compact, register based shadow stack for deployment

Shadow Stacks + CFI = practical CFH mitigation

https://github.com/HexHive/ShadowStack