Hyperchaining Optimizations for an LLVM-Based Binary Translator on x86-64 and RISC-V Platforms

Jyun-Kai Lai and Wuu Yang
Department of Computer Science,
National Yang Ming Chiao Tung University, Taiwan
Outline

- Background
- Hyperchaining
- Experiments
- Conclusion
Outline

- Background
- Hyperchaining
- Experiments
- Conclusion
Binary translation

- Translate from guest (input) binary to host (output) binary and the translated program runs on a host machine
- Rabbit - an LLVM-based binary translator developed by our lab
  - Our work is based on Rabbit and translate from RISC-V binary to x86-64
Architecture

- **Frontend (Guest → LLVM IR)**
  - Translate from guest binary to LLVM IR basic blocks
  - Incorporate one or more instructions into a LLVM IR basic block depend on SBT or DBT

- **Backend (LLVM IR → Host)**
  - Translate from LLVM IR basic blocks to binary chunks
Translation

- **Static Binary Translation (SBT)**
  - Translate code blocks before run-time

- **Dynamic Binary Translation (DBT)**
  - Translate code blocks in run-time

- **Hybrid Binary Translation (HBT; adopted by Rabbit)**
  - SBT + DBT
**Translated code block**

- Translation phase processes the client’s instruction and translated into many code blocks, which we call **Translated Code Blocks** (TCBs)

- You may get many guest-to-host mapping relationship in the translation phase
Translation request: 0x12000

Run-time dispatcher

Target Program Counter (TPC)

Source Program Counter (SPC)

next destination

forward

Transfer control

address-mapping table

0x12000 \rightarrow 0x2000000

CPU Context

a5 = 0x12000

guest-binary

...
Requested SPC

Run-time Library

Address-Mapping Table

TPC

Just-in-time Compiler

 translate SPC if lookup fails

look up address

Add new entry <SPC, TPC>
In address-mapping table

Possible Case 1: the source block has been translated (from SBT or DBT)
Possible Case 2: the source block has never been translated
Run-time Library

Address-Mapping Table

Just-in-time Compiler

Possible Case 1: the source block has been translated (from SBT or DBT)
Possible Case 2: the source block has never been translated
Run-time Library

Address-Mapping Table

Just-in-time Compiler

Translates SPC if lookup fails

Add new entry <SPC, TPC>
In address-mapping table

Possible Case 1: the source block has been translated (from SBT or DBT)

Possible Case 2: the source block has never been translated

Jyun-Kai Lai
Requested SPC

Run-time Library

Address-Mapping Table

Just-in-time Compiler

TPC

Possible Case 1: the source block has been translated (from SBT or DBT)
Possible Case 2: the source block has never been translated

translate SPC if lookup fails

look up address

Add new entry <SPC, TPC>
In address-mapping table

Jyun-Kai Lai
Instruction Terminator

- Every basic block ends with an instruction terminator
- Can be classified into two groups
  - Direct Branches
  - Indirect Branches

```
...                ...
instruction terminator
jr a5
```
Direct Branches

- Conditional branches, unconditional branches, direct function calls
- Single destination
- It is possible to know the destination in compilation time

Target $\leftarrow \$PC + \text{offset} \ (\text{PC-relative addressing})$

Indirect Branches

- May have more than one destination
- The destination may be stored in a register or memory
- It is difficult to predict the destinations without context information
Optimizations in Rabbit

- Direct Branch Link Optimization
  - In compilation time
- Direct Function Return Optimization
  - In compilation time
- Block Chaining Optimization
  - In run-time
Direct Branch Link

**Steps**

1. Analyze LLVM basic blocks which end with direct branches
2. Try to find the SPC of successor block as much as possible and modify the former block to directly jump to the next one in compilation time

**Mitigate run-time overhead and improve performance**
Direct Function Return

- In the normal situation, the return address of a function call should be the address of next instruction.

- This optimization exploits that fact to tell the translator the TPC of return address in compilation time.
  - No need to lookup return address in run-time.
Guest-binary

```assembly
<foo>:
  ...
  ld s0,8(sp)
  addi sp,sp,16
  ret

  Block 1

  ...

  ...  
  addi s0,sp,16
  addi s0,sp,16
  jal ra,<foo>

  Block 2

  α+4 : addi s0,sp,16
  α+0 : jal ra,<foo>

  Block 3

  α+4 : li a5,0
  ...

  Block 3

  ...
```

CPU Context

<table>
<thead>
<tr>
<th>register</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ra</td>
<td>α+4 → 0x4028000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Address-Mapping Table

<table>
<thead>
<tr>
<th>SPC</th>
<th>TPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>α+4</td>
<td>0x4028000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Replaced by TPC

TCB (Block 2)

Code to write TCP of return address to CPU Context

Jyun-Kai Lai
Guest-binary

```assembly
<foo>:
... 
ld  s0,8(sp)
addi sp,sp,16
ret
```

TCB (Block 1)

```
<table>
<thead>
<tr>
<th>CPU Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>ra</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```

Read ra to go to successor block

TCB (Block 3)

```
0x4028000:
```

 степени Кай Ли, Jyun-Кай Лай
Direct Function Return

- Caveat: We assume the link register will not be modified in the subroutine.
- Otherwise, the program will crash because of corrupt link register.
Block Chaining Optimization

- Basically, block chaining optimization does the same thing as direct branch link optimization does but in run-time.

- Adopt **self-modifying code**
  - Modify its own instructions in run-time
  - Patch the destination into the code
Block Chaining Optimization

TCB1 -> TCB1’s Stub

- Dummy instructions
- Call address dispatching
- Jump to successor block

TCB2 (successor)

Run-time library

First run

Tail call
Block Chaining Optimization

First run

TCB1

TCB1's Stub

Dummy instructions

Call address dispatching

Jump to successor block

SPC

Run-time library

TCB2 (successor)

tail call
Block Chaining Optimization

First run

TCB1

TCB1’s Stub

Patched code (jump to TCB2)

Call address dispatching

Jump to successor block

TCB2 (successor)

Run-time library

tail call
Block Chaining Optimization

First run

TCB1

TCB1’s Stub

Patched code (jump to TCB2)

Call address dispatching

Jump to successor block

TCB2 (successor)

tail call

TPC

Run-time library

Jyun-Kai Lai
Block Chaining Optimization

Later runs

TCB1 → TCB1’s Stub

Patched code (jump to TCB2)

Call address dispatching

Jump to successor block

Run-time library

TCB2 (successor)

tail call
Block Chaining Optimization

- Limitations
  - Only support chaining dtblocks to dtblocks
    - That means this optimization can’t handle (stblocks → stblocks; stblocks → dtblocks and dtblocks → stblocks) chaining
  - Tail call optimization is involved that means we have an extra jump instruction. That is not efficient
  - Cannot handle indirect branch chaining
  - This optimization only supports old translator we had made before. Rabbit cannot use this optimization
Outline

- Background
- Hyperchaining
- Experiments
- Conclusion
Hyperchaining

- We develop **Hyperchaining** optimization, an improved version of block chaining optimization
- Be able to chain both stblocks and dtblocks
- Be able to handle indirect branches
- Design to work well with HBT and Rabbit
Hyperchaining

- Depend on the implementation, hyperchaining optimization can be classified into two groups
  - Platform-independent (indep) hyperchaining
  - Platform-dependent (dep) hyperchaining


Indep-Hyperchaining

- In indep-hyperchaining, we design a subroutine, **direct-branch chaining handler**, to chain blocks for direct branches.
- Implemented in LLVM IR level.
- This subroutine supports some functionalities of run-time dispatcher:
  - Dynamic binary translation
  - Address lookup
Indep-Hyperchaining

Steps

1. TCB loads address from a dedicated memory, \textit{TransferDestination}, for every TCB
   - \textit{TransferDestination} is initialized with the address of \textit{Intermediary}
   - \textit{Intermediary} calls direct-branch chaining handler to chain blocks and update \textit{TransferDestination} with TPC
First run

TCB1 (direct branch)

Load target from TransferDestination and jump

&intermediary

TransferDestination

&Intermediary

TCB1’s intermediary

ICPP-EMS 2021
Intermediary

Call direct-branch chaining handler

Load destination from TransferDestination

Jump

direct-branch chaining handler

&Intermediary \rightarrow TPC
Second run

TCB1 (direct branch)

Load target from TransferDestination and jump

&TCB2

TCB1’s intermediary

&TCB2

TCB2
Indep-Hyperchaining For Indirect Branches

- For indirect branch, we have similar subroutine, *indirect-branch chaining handler*, to chain blocks with indirect branches
- Implemented in LLVM IR level
Indep-Hyperchaining For Indirect Branches

- Steps
  - TCB calls indirect-branch chaining handler and pass SPC to handler
  - Chaining handler lookups SPC in **Jump Destination Cache** first. If nothing is found, do address lookup to acquire corresponding TPC and store address translation pair entry <SPC, TPC> to that cache
TCB1 (indirect-branch)

Call indirect-branch chaining handler

jump

---

TCB2 (successor block)'s SPC and &JumpDestinationCache

indirect-branch chaining handler

lookup SPC & insert entry if lookup fails

TCB1’s Jump Destination Cache

<table>
<thead>
<tr>
<th>SPC</th>
<th>TPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCB2’s SPC</td>
<td>TCB2’s TPC</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Jyun-Kai Lai
Call indirect-branch chaining handler

TCB1's Jump Destination Cache

<table>
<thead>
<tr>
<th>SPC</th>
<th>TPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCB2's SPC</td>
<td>TCB2's TPC</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

TCB2's TPC

indirect-branch chaining handler

TCB1 (indirect-branch)
Dep-Hyperchaining For Direct Branches

- The reasons why we develop **dep-hyperchaining** (platform-dependent hyperchaining)
  - We observe some redundant operations in Indep-hyperchaining
    - For direct branches, TCBs need to read the same TPC of successor’s block every time the block executes
    - For indirect branches, one possible case is single destination indirect branch which makes jump destination cache inefficient
  - The design of **dep-hyperchaining** is based on the **indep-hyperchaining**
Indep-hyperchaining for direct branches

Load target from TransferDestination and jump

&TCB2

&TCB2

How to simplify? Block chain optimization is a good idea but Rabbit doesn’t support that.
Indep-hyperchaining for direct branches

TransferDestination

&TCB2

Patch?

&TCB2

TCB1 (direct branch) --> TCB2

How to simplify? Block chain optimization is a good idea but Rabbit doesn’t support that. What if we directly patch to the TCB?
Challenges & Solutions

- Where is the instructions of load memory operations in TCBs
  - We need a anchor point to tell the run-time library the address of that instructions in TCBs
- How to patch the code. Is it practicable?
  - We need to analyze possible generated code patterns
Load Effective Address

- **LEA**
  - `lea` permits the program counter as its source operand
  - rip minus the offset 7 (i.e. the length of `lea`) to get the address of `lea` instruction itself

- Store the address into the globally accessible **CPU context**

```
REX.W 8D ModRM F9 FF FF FF  lea r64, QWORD PTR [rip - 0x7]
```
Store Address Into CPU Context

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcodes</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 05 F9 FF FF FF</td>
<td>lea rax, QWORD PTR [rip-7]</td>
<td></td>
</tr>
<tr>
<td>48 8B 0C 24</td>
<td>mov rcx, QWORD PTR [rsp]</td>
<td></td>
</tr>
<tr>
<td>48 89 41 0A</td>
<td>mov QWORD PTR [rcx+10], rax</td>
<td></td>
</tr>
</tbody>
</table>

CPU Context

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x10</td>
<td>anchor point</td>
</tr>
</tbody>
</table>
Load TransferDestination’s Address

| 48 B8 08 60 EA F7 FF 7F 00 00 | movabs rax, 0x7ffff7ea6008 |
| 48 8B 00                      | mov rax, QWORD PTR [rax] |

TCB (direct branch)

Load target from TransferDestination and jump
Every block contains only CPU context as a parameter to make code block be able to read and write simulated registers through CPU context.

| 48 89 CF | mov rdi, rcx |
| FF E0   | jmp rax    |
The aforementioned instruction sequences constitute code pattern 1 in our definition.

<table>
<thead>
<tr>
<th>48 8D 05 F9 FF FF FF</th>
<th>lea rax, QWORD PTR [rip-7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8B 0C 24</td>
<td>mov rcx, QWORD PTR [rsp]</td>
</tr>
<tr>
<td>48 89 41 0A</td>
<td>mov QWORD PTR [rcx+10], rax</td>
</tr>
<tr>
<td>48 B8 08 60 EA F7 FF FF 00 00</td>
<td>movabs rax, 0x7fffffff7ea60008</td>
</tr>
<tr>
<td>48 8B 00</td>
<td>mov rax, QWORD PTR [rax]</td>
</tr>
<tr>
<td>48 89 CF</td>
<td>mov rdi, rcx</td>
</tr>
<tr>
<td>FF E0</td>
<td>jmp rax</td>
</tr>
</tbody>
</table>
Then, How To Patch

- The intuitive way is patching the last instruction "jmp r64"

- Unfortunately, it is impracticable
  - Patch out-of-bound
  - Conditional branch block
Jump Patch

- A minimal jump patch is a direct-jump patch
- "JMP rel32" takes 5 bytes
- "JMP rel8" is useless here because tiny displacement is rare to appear. We don’t consider such case

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EB cb</td>
<td>JMP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short, RIP = RIP + 8-bit displacement sign extended to 64 bits</td>
</tr>
<tr>
<td>E9 cw</td>
<td>JMP rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>E9 cd</td>
<td>JMP rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64 bits</td>
</tr>
</tbody>
</table>

From Intel® 64 and IA-32 Architectures Software Developer’s Manual
A minimal patch, "jmp rel32", takes 5 bytes. However, last instruction "jmp r64" only provides 2 bytes space to patch. 

Patch 3 more bytes!!

\[
\begin{array}{ccc}
FF & E0 & jmp \text{ rax} \\
\end{array}
\]

\[
\begin{array}{ccc}
E9 & XX & jmp \text{ rel32} \\
XX & XX & XX \\
\end{array}
\]
Conditional Branch Blocks

- “taken” and “not taken” blocks are contiguous in the run-time memory
  - There are no reserved bytes between two TCBs
  - Patch out-of-bound problem implies “not taken” block get overwritten at the first few bytes and becomes corrupt
Patch

- For such concerns, we don’t patch last instruction

- We pick “movabs rax, 0x7fffff7ea6008” as a proper instruction to patch
  - movabs offers 10 bytes space that is quite enough
# Patched TCB for code pattern 1

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 05 F9 FF FF FF FF</td>
<td>lea rax, QWORD PTR [rip-7]</td>
</tr>
<tr>
<td>48 8B 0C 24</td>
<td>mov rcx, QWORD PTR [rsp]</td>
</tr>
<tr>
<td>48 89 41 0A</td>
<td>mov QWORD PTR [rcx+10], rax</td>
</tr>
<tr>
<td>48 B8 08 60 EA F7 FF 7F 00 00</td>
<td>movabs rax, 0x7fffff7ea6008</td>
</tr>
<tr>
<td>48 8B 00</td>
<td>mov rax, QWORD PTR [rax]</td>
</tr>
<tr>
<td>48 89 CF</td>
<td>mov rdi, rcx</td>
</tr>
<tr>
<td>FF E0</td>
<td>jmp rax</td>
</tr>
</tbody>
</table>

| 48 8D 05 F9 FF FF FF FF                                               | lea rax, QWORD PTR [rip-7]                                            |
| 48 8B 0C 24                                                           | mov rcx, QWORD PTR [rsp]                                              |
| 48 89 41 0A                                                           | mov QWORD PTR [rcx+10], rax                                           |
| 48 89 CF                                                              | mov rdi, rcx                                                         |
| E9 XX XX XX XX XX                                                     | jmp <TPC> (successor block)                                           |
| 00 00                                                                 | ??? (don’t care; unreachable)                                         |
| 48 8B 00                                                               | mov rax, QWORD PTR [rax]                                              |
| 48 89 CF                                                              | mov rdi, rcx                                                         |
| FF E0                                                                 | jmp rax                                                               |
Indirect-Branch Patch

- If the displacement is too large to be encoded as a direct-jump patch, we use indirect-branch patch

```
movabs rax, <TPC>
jmp rax
```
# Patched TCB For Code Pattern 1

## Before

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 05 F9 FF FF FF</td>
<td>lea rax, QWORD PTR [rip-7]</td>
</tr>
<tr>
<td>48 8B 0C 24</td>
<td>mov rcx, QWORD PTR [rsp]</td>
</tr>
<tr>
<td>48 89 41 0A</td>
<td>mov QWORD PTR [rcx+10], rax</td>
</tr>
<tr>
<td>48 B8 08 60 EA F7 FF 7F 00 00</td>
<td>movabs rax, 0x7fffff7ea6008</td>
</tr>
<tr>
<td>48 8B 00</td>
<td>mov rax, QWORD PTR [rax]</td>
</tr>
<tr>
<td>48 89 CF</td>
<td>mov rdi, rcx</td>
</tr>
<tr>
<td>FF E0</td>
<td>jmp rax</td>
</tr>
</tbody>
</table>

## After

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 05 F9 FF FF FF</td>
<td>lea rax, QWORD PTR [rip-7]</td>
<td></td>
</tr>
<tr>
<td>48 8B 0C 24</td>
<td>mov rcx, QWORD PTR [rsp]</td>
<td></td>
</tr>
<tr>
<td>48 89 41 0A</td>
<td>mov QWORD PTR [rcx+10], rax</td>
<td></td>
</tr>
<tr>
<td>48 B8 XX XX XX XX XX XX XX</td>
<td>movabs rax, &lt;TPC&gt; (replaced by successor block)</td>
<td>nop DWORD PTR [rax](disable dereference)</td>
</tr>
<tr>
<td>0F 1F 00</td>
<td>nop DWORD PTR [rax](disable dereference)</td>
<td></td>
</tr>
<tr>
<td>48 89 CF</td>
<td>mov rdi, rcx</td>
<td></td>
</tr>
<tr>
<td>FF E0</td>
<td>jmp rax</td>
<td></td>
</tr>
</tbody>
</table>
Code Pattern 2

- Similar to code pattern 1 but has less instructions
- rdi still holds &CPUContext

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea rax, QWORD PTR [rip-7]</td>
<td>48 8D 05 F9 FF FF FF</td>
</tr>
<tr>
<td>mov QWORD PTR [rcx+10], rax</td>
<td>48 89 41 0A</td>
</tr>
<tr>
<td>movabs rax, 0x7fffff7ea6008</td>
<td>48 B8 08 60 EA F7 FF 7F 00 00</td>
</tr>
<tr>
<td>mov rax, QWORD PTR [rax]</td>
<td>48 8B 00</td>
</tr>
<tr>
<td>jmp rax</td>
<td>FF E0</td>
</tr>
</tbody>
</table>
# Patched TCB For Code Pattern 2

**Before**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 05 F9 FF FF FF FF</td>
<td>lea rax, QWORD PTR [rip-7]</td>
</tr>
<tr>
<td>48 89 41 0A</td>
<td>mov QWORD PTR [rcx+10], rax</td>
</tr>
<tr>
<td>48 B8 08 60 EA F7 FF 7F 00 00</td>
<td>movabs rax, 0x7fffffff7ea6008</td>
</tr>
<tr>
<td>48 8B 00</td>
<td>mov rax, QWORD PTR [rax]</td>
</tr>
<tr>
<td>FF E0</td>
<td>jmp rax</td>
</tr>
</tbody>
</table>

**After**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E9 XX XX XX XX XX</td>
<td>jmp &lt;TPC&gt; (successor block)</td>
</tr>
<tr>
<td>FF FF</td>
<td>??</td>
</tr>
<tr>
<td>48 89 41 0A</td>
<td>mov QWORD PTR [rcx+10], rax</td>
</tr>
<tr>
<td>48 B8 08 60 EA F7 FF 7F 00 00</td>
<td>movabs rax, 0x7fffffff7ea6008</td>
</tr>
<tr>
<td>48 8B 00</td>
<td>mov rax, QWORD PTR [rax]</td>
</tr>
<tr>
<td>FF E0</td>
<td>jmp rax</td>
</tr>
</tbody>
</table>
Dep-Hyperchaining For Indirect Branches

- According to our study of SPEC CPU 2006 CINT benchmarks:
  - 56% (avg.) of the indirect branches in the source binary are function-return instructions.
  - 72.88% (avg.) of the indirect branches which are not function return have only a single jump destination.
    - Jump-destination cache is unnecessary in single destination case.
SPEC CPU CINT 2006 (the percentage of indirect branches that have only one destination. Function returns are exclude in this calculation).
Design

- The code design need to consider situations for both single destination and multiple destinations
- Change the code behavior dynamically
  - Transform single destination $\rightarrow$ multiple destinations by patching code
Solutions

- The code design need to consider situations for both single destination and multiple destinations
  - Single destination: Code Cave
A **code cave** is one or several consecutive instructions in which some fields are intentionally filled with dummy values in the beginning.

Later these fields are patched with data related to the indirect branch, such as SPC, TPC, etc.
TCB1 (indirect-branch)

Code Cave

Call indirect-branch chaining handler

jump

Code Cave (initial state)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 15 00 00 00 00</td>
<td>lea rdx, QWORD PTR [rip]</td>
</tr>
<tr>
<td>48 81 FE FF FF FF 7F</td>
<td>cmp rsi, 0x7fffffff</td>
</tr>
<tr>
<td>48 0F 44 FF</td>
<td>cmove rdi, rdi</td>
</tr>
<tr>
<td>0F 1F 80 00 02 00 00</td>
<td>nop dword ptr [rax+0x200]</td>
</tr>
<tr>
<td>0F 1F 80 00 02 00 00</td>
<td>nop dword ptr [rax+0x200]</td>
</tr>
</tbody>
</table>
ICPP-EMS 2021

TCB1 (indirect-branch)

Call indirect-branch chaining handler

Jump

Patched Code Cave (direct jump patch; \(\alpha\)-patch)

<table>
<thead>
<tr>
<th>Code Cave</th>
<th>Patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 15 00 00 00 00</td>
<td>lea rdx, QWORD PTR [rip]</td>
</tr>
<tr>
<td>48 81 FE 00 35 01 00</td>
<td>cmp rsi, 0x13500</td>
</tr>
<tr>
<td>49 0F 44 FE</td>
<td>cmov rdi, r14</td>
</tr>
<tr>
<td>0F 84 XX XX XX XX</td>
<td>je 0x2000000</td>
</tr>
<tr>
<td>0F 1F 84 00 00 02 00 00</td>
<td>nop DWORD PTR [rax+rax*1+0x200]</td>
</tr>
</tbody>
</table>

SPC = 0x13500

indirect-branch chaining handler

mov rdi, r14
jmp rax

Analyze

From Intel® 64 and IA-32 Architectures Software Developer’s Manual

Jyun-Kai Lai
### TCB1 (indirect-branch)

**Code Cave**

**Call indirect-branch chaining handler**

**Jump**

**Patched Code Cave (indirect jump patch; β-patch)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 15 00 00 00 00 00</td>
<td>lea rdx, QWORD PTR [rip]</td>
</tr>
<tr>
<td>48 81 FE 00 35 01 00</td>
<td>cmp rsi, 0x13500</td>
</tr>
<tr>
<td>49 0F 44 FE</td>
<td>cmov rdi, r14</td>
</tr>
<tr>
<td>75 0C</td>
<td>jne failure</td>
</tr>
<tr>
<td>48 B8 00 00 00 02 00 00 00 00</td>
<td>movabs rax, 0x2000000</td>
</tr>
<tr>
<td>FF E0</td>
<td>jmp rax</td>
</tr>
</tbody>
</table>

**SPC = 0x13500**

**Indirect-branch chaining handler**

**mov rdi, r14**

**jmp rax**

**Analyze**

**Patch**

**75 cb**

**JNE rel8**

---

*From Intel® 64 and IA-32 Architectures Software Developer's Manual*

---

Jyun-Kai Lai

ICPP-EMS 2021
Solutions

- The code design need to consider situations for both single destination and multiple destinations
  - Single destination: **Code Cave**
  - Multiple destinations: **Dynamic Address Mapping Table**
- Change the code behavior dynamically
  - Repatch the code cave
**TCB1 (indirect-branch)**

- Call indirect-branch chaining handler
- Jump

---

**Patched Code Cave (direct jump patch; \( \alpha \)-patch)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 15 00 00 00 00</td>
<td><code>lea rdx, QWORD PTR [rip]</code></td>
</tr>
<tr>
<td>48 81 FE 00 35 01 00</td>
<td><code>cmp rsi, 0x13500</code></td>
</tr>
<tr>
<td>49 0F 44 FE</td>
<td><code>cmov rdi, r14</code></td>
</tr>
<tr>
<td>0F 84 XX XX XX XX</td>
<td><code>je 0x2000000</code></td>
</tr>
<tr>
<td>0F 1F 84 00 00 02 00 00</td>
<td><code>nop DWORD PTR [rax+rax*1+0x200]</code></td>
</tr>
</tbody>
</table>

---

**FAIL**
We find that this indirect branch has more than one destination.
Rabbit allocates a block of memory for dynamic address mapping table

TCB1’s dynamic address mapping table

<table>
<thead>
<tr>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 15 00 00 00 00</td>
<td>lea rdx, QWORD PTR [rip]</td>
</tr>
<tr>
<td>E9 XX XX XX XX</td>
<td>jmp &lt;DAMT&gt;</td>
</tr>
<tr>
<td>91 00</td>
<td>??</td>
</tr>
<tr>
<td>49 0F 44 FE</td>
<td>cmove rdi, r14</td>
</tr>
<tr>
<td>0F 84 XX XX XX</td>
<td>je 0x2000000</td>
</tr>
<tr>
<td>0F 1F 84 00 00 02 00</td>
<td>nop DWORD PTR [rax+rax*1+0x200]</td>
</tr>
</tbody>
</table>

Jyun-Kai Lai

ICPP-EMS 2021
TCB1 (indirect-branch)

Code Cave

Call indirect-branch chaining handler

jump

TCB1’s dynamic address mapping table

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 8D 15 00 00 00 00</td>
<td>lea rdx, QWORD PTR [rip]</td>
</tr>
<tr>
<td>E9 XX XX XX XX</td>
<td>jmp &lt;DAMT&gt;</td>
</tr>
<tr>
<td>01 00</td>
<td>??</td>
</tr>
<tr>
<td>49 0F 44 FE</td>
<td>cmp rdi, r14</td>
</tr>
<tr>
<td>0F 84 XX XX XX XX</td>
<td>je 0x2000000</td>
</tr>
<tr>
<td>0F 1F 84 00 00 02 00 00</td>
<td>nop DWORD PTR [rax+rax*1+0x200]</td>
</tr>
</tbody>
</table>

lookup_fail_in_DAMT:
Dynamic Address Mapping Table

DAMT:
cmp rsi, 0x10000
je tag1
jmp <code cave>

tag1:
mov rdi, r14
movabs rax, 0x7ffff0000000
jmp rax

<table>
<thead>
<tr>
<th>Visited Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPC</td>
</tr>
<tr>
<td>0x10000</td>
</tr>
</tbody>
</table>

Jyun-Kai Lai
Dynamic Address Mapping Table

DAMT:
cmp rsi, 0x10000
je tag1
cmp rsi, 0x11000
je tag2
jmp <code cave>

tag1:
mov rdi, r14
movabs rax, 0x7ffff0000000
jmp rax
tag2:
mov rdi, r14
movabs rax, 0x7ffff1000000
jmp rax

<table>
<thead>
<tr>
<th>SPC</th>
<th>TPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000</td>
<td>0x7ffffff0000000</td>
</tr>
<tr>
<td>0x11000</td>
<td>0x7ffffff1000000</td>
</tr>
</tbody>
</table>
Outline

- Background
- Hyperchaining
- Experiments
- Conclusion
Experimental Setup

- CPU: Intel Core i7-7700 3.60GHz
- OS: Ubuntu 18.04.3 LTS with Linux kernel 4.15.0
- RAM: 16GB
- LLVM: 8.0.1
- Benchmark: SPEC CPU 2006 CINT
SPEC CPU CINT 2006 (platform-dependent hyperchaining for direct branches)
SPEC CPU CINT 2006 (Memory usage for indirect branch hyperchaining)

Memory usage (MB) for various SPEC CPU 2006 benchmarks, comparing platform-independent and platform-dependent hyperchaining.
Outline

- Background
- Hyperchaining
- Experiments
- Conclusion
Conclusion

- This paper proposes platform-independent hyperchaining and platform-dependent hyperchaining on the x86-64 platform.

- The experimental results show that platform-dependent hyperchaining can reach 1.08x and 1.05x speedup compared to platform-independent hyperchaining.
Conclusion

- Memory usage of platform-dependent hyperchaining is $1.0089x$ and $1.019x$ times that of platform-independent hyperchaining for direct branches and indirect branches, respectively.

- It has small memory overhead to adopt platform-dependent approach.
Future Work

- The approach of platform-dependent hyperchaining for indirect branches implies potential opportunities to be improved on performance
  - Schedule the first section of the SPC section to reduce and mitigate the miss penalty in run-time
  - Change the mechanism of searching in DAMT (Hash table ?)
  - Lack of a policy to disable platform-dependent hyperchaining when the dynamic address mapping table is full
Credits

- The hack font - authors of Source Foundry, which is licensed under MIT
  https://github.com/source-foundry/Hack
THANK YOU FOR YOUR ATTENTION