Execution Domain Transition:
Binary and LLVM IR can run in conjunction

Jaeyong Ko, Sangrok Lee, Jieun Lee, Jaewoo Shim
| 1. Background                        |
| 2. LLI Interpreter mode             |
| 3. Binary Rewriting                 |
| 4. Our idea                         |
| 5. Domain Transition                |
| 6. Experiment Result                |
| 7. Demo                             |
• Multi-CPU, Single Host Analysis Platform
  • To analyze Multi-arch Linux malware
    • The source code of Mirai opened 6 years ago
    • The variations of Mirai are still attacking IoT systems today
  • To prepare execution environment of each CPU architecture

*Mirai Source Code ➔ compile ➔ attack
Multi-CPU arch. systems

• No Source code, Only Binary
  • Most Analysts has not knowledge about every CPU architecture
  • There are huge cost to build all virtual machine environment
  • So, we will analyze binaries using CPU-independent IR

※ https://github.com/jgamblin/Mirai-Source-Code
Background: Binary Lifting

**LLVM IR from a source code**

```c
int64_t bar(){
    int64_t var = 1;
    return var;
}

int64_t foo(){
    int64_t foovar = bar()+1;
    return foovar;
}

int64_t doo(){
    int64_t doovar = foo()+1;
    return doovar;
}

int64_t car(){
    int64_t carvar = doo()+1;
    return carvar;
}
```

```llvm
define i64 @doo() #0 {
    entry:
        %doovar = alloca i64, align 8
        %call = call i64 @foo()
        %add = add nsw i64 %call, 1
        store i64 %add, i64* %doovar, align 8
        %0 = load i64, i64* %doovar, align 8
        ret i64 %0
}
```

```c
int64_t bar(){
    int64_t var = 1;
    return var;
}

int64_t foo(){
    int64_t foovar = bar()+1;
    return foovar;
}

int64_t doo(){
    int64_t doovar = foo()+1;
    return doovar;
}

int64_t car(){
    int64_t carvar = doo()+1;
    return carvar;
}
```

```llvm
define i64 @car() #0 {
    entry:
        %carvar = alloca i64, align 8
        %call = call i64 @doo()
        %add = add nsw i64 %call, 1
        store i64 %add, i64* %carvar, align 8
        %0 = load i64, i64* %carvar, align 8
        ret i64 %0
}
```
Background: Binary Lifting

- Lifted IR and Binary comparison

```llvm
bb.car:
%10 = load i64, i64* @rbp
%11 = load i64, i64* @rsp
%12 = sub i64 %11, 8
%13 = inttoptr i64 %12 to i64*
store volatile i64 %10, i64* %13
store i64 %12, i64* @rsp
%14 = load i64, i64* @rsp
store i64 %14, i64* @rbp
%15 = load i64, i64* @rsp
%16 = sub i64 %15, 8
%17 = inttoptr i64 %16 to i64*
store volatile i64 4197433, i64* %17
store i64 %16, i64* @rsp
store volatile i64 4197408, i64* pc
call void @doo_0x400c20_lifted(i64 4197433)
br label %bb.car.0x9
```

```assembly
_\texttt{test} \_\texttt{\_\_\_test\.linked.\_\_ll (Lifted IR)}

\texttt{0x400c20 <doo>}: push %rbp
\texttt{0x400c21 <doo+1>}: mov %rsp,%rbp
\texttt{0x400c24 <doo+4>}: callq 0x400c10 <foo>
\texttt{0x400c29 <doo+9>}: add $0x1,%eax
\texttt{0x400c2c <doo+12>}: pop %rbp
\texttt{0x400c2d <doo+13>}: retq

_\texttt{0x400c30 <car>}: push %rbp
\texttt{0x400c31 <car+1>}: mov %rsp,%rbp
\texttt{0x400c34 <car+4>}: callq 0x400c20 <doo>
```

Binary LLVM IR from Binary
Background: Virtual CPU, Virtual Stack

- Tiny Virtual Machine in LLVM IR for supporting Multi-CPU Arch.
  - Store the state to virtual CPU and Stack, neither real CPU and Stack.
  - We developed the LLI variation for Multi-CPU Binaries

※ A Cross Debugger for Multi-Architecture Binaries(euro-llvm2020)

Machine code's implicit & explicit expression

ex) x64

```c
mov %fs:0x28, %rax
```

lifting

```c
%22 = load i64, i64* @fs
%23 = add i64 40, %22
%24 = inttoptr i64 %23 to i64*
%25 = load volatile i64, i64* %24
store i64 %25, i64* @rax
```
LLI Interpreter Mode: Drawbacks

• Extremely slow.. Why?

Lifted IR contains all sorts of information about execution such as CPU state variation and side effects etc.

LLVM IR

4 machine codes
→ 19 lines

40 bits
4 machine codes
→ 19 lines

LLVM Interpreter

bb.main:
; 0x400cbf
%15 = load i64, i64* @rbp
%16 = load i64, i64* @rsp
%17 = sub i64 %16, 8
%18 = inttoptr i64 %17 to i64*
store volatile i64 %15, i64* %18
store i64 %17, i64* @rsp

; 0x400cc0
%19 = load i64, i64* @rsp
store i64 %19, i64* @rbp

; 0x400cc3
%20 = load i64, i64* @rsp
%21 = sub i64 %20, 48
store i32 17, i32* @cc_op
store i64 %21, i64* @cc_dst
store i64 48, i64* @cc_src
store i64 %21, i64* @rsp

; 0x400cc7
%22 = load i64, i64* @reg_288_64
%23 = add i64 40, %22
%24 = inttoptr i64 %23 to i64*
%25 = load volatile i64, i64* %24
store i64 %25, i64* @rax
LLI Interpreter Mode: Drawbacks

- Extremely slow. Why?

To emulate 1 line of assembly code from original binary, from **tens to thousands** assembly lines should be executed on host.

Accumulated overhead results badly on running time.
LLI Interpreter Mode: Running Time

- **Running time comparison**

### Original
- **false**: 0.01
- **true**: 0.01
- **nice**: 0.01

### Binary
- **false**: 0.461719
- **true**: 0.448955
- **nice**: 0.741142

---

**Groups**

**Group A**: 0 ≤ running time < 1 (sec)
- **logname**: 0.02
- **date**: 0.02
- **hostid**: 0.01
- **groups**: 0.01
- **id**: 0.01

**Group B**: 1 ≤ running time < 15 (sec)
- **logname**: 1.028615
- **date**: 2.847946
- **hostid**: 2.737518
- **groups**: 4.219185
- **id**: 13.41932

**Group C**: 15 ≤ running time (sec)
- **ls**: 4.4871
- **vdir**: 152.6706
- **df**: 394.0128


- **LLI Execution Demo**

```
@ubuntu:/Downloads/webserver-master$ ./exec.sh arm ls
Target ARCH : arm
Target Binary : ls
cmd: ./webserver -elf ./bin/arm/ls.arm.translated ./bin/arm/ls.arm.linked.ll
log: note: Load the bitcode..
IR Parsing Time 17.456044sec
lli: note: Ready
```

- **Default Execution (No debug info)**

```
@ubuntu:/Downloads/webserver-master$ ./exec.sh arm ls
Target ARCH : arm
Target Binary : ls
cmd: ./webserver -elf ./bin/arm/ls.arm.translated ./bin/arm/ls.arm.linked.ll
log: note: Load the bitcode..
IR Parsing Time 17.271865sec
lli: note: Ready
(lli-master) info flag printcall true [3]
```

- **Print function call flow**
**Binary Rewriting**

- **Rewritten Binary Architecture**

  %22 = load i64, i64* @fs
  %23 = add i64 40, %22
  %24 = inttoptr i64 %23 to i64*
  %25 = load volatile i64, i64* %24
  store i64 %25, i64* @rax

  **Rewritten Binary**

  **Original Binary**

  [Data]

  @fs @rax

  [Data]

  **Rewritten Code**

  **Virtual CPU**

  **Virtual Stack**

  [Data]

  **Rewritten Binary**

  **Rewritten Code**

  **Real CPU**

  **Real Stack**

  **Host Platform**

  compile

  load
  inttoptr
  store
  add
  load
Binary Rewriting: Running Time

- Running time comparison

### Group A
- **FALSE**
- **TRUE**
- **nice**

### Group B
- logname: 1.028615
- date: 2.847946
- hostid: 2.737518
- groups: 4.219185
- id: 13.41932

### Group C
- ls: 44.871
- vdir: 152.6706
- df: 394.0128

### LLI
- FALSE: 0.461719
- TRUE: 0.448955
- nice: 0.741142

### Rewritten Binary
- FALSE: 0.016
- TRUE: 0.036
- nice: 0.04

- logname: 0.038
- date: 0.046
- hostid: 0.048
- groups: 0.04
- id: 0.049

- ls: 0.182
- vdir: 0.244
- df: 0.226
Our idea

- Binary and IR(LLI) can run in conjunction
  - Can achieve both execution performance and analysis efficiency

Shared Data Storage

Analysis
Optimization
Symbol Recovery
Decompilation

Execution Speed
Portability

Interpreting
Compile

State
State

'Same'

LLVM IR
Domain Transition: LLI & Binary Execution

**Target Execution**
- Updating
- LLI (target)
- Interpreting

**Host Execution**
- LLVM IR
- Single Host Platform
  - Real Stack
  - Real CPU

**Virtual Memory**
- [Stack] (Host-execution)
- External Libraries
- [heap] Parsed IR Module

**LLI**
- Virtual CPU & Stack (Target-execution)
- Rewritten Binary

**Single LLI process maps**

**STATE**
- Virtual Storage
  - Stack
  - CPU

**Compile**

**Share**
• Two closed worlds (IR vs Binary)
  • The IR and Binary domain execute for each in closed world.
Domain Transition: IR World & Binary World

- Two closed worlds (IR vs Binary)
  - Can take benefits of the other closed world using domain transition.

```
LLVM IR
1. start()
2. main_lifted()
   _Exit()
   Foo()
   Bar()

Binary
1. start()
2. main_lifted()
   Car()
   Doo()
   Foo()
   Bar()

Closed World
Analysis
Optimization
Symbol Recovery

Closed World
Running Time
Portability
```
# Sharing the Data storage for the virtual state

1. Collects symbol Information from the rewritten binary

```
Symbol Table
SymName   Addr
SymName   Addr
SymName   Addr
...
```
# Sharing the Data storage for the virtual state

1. Collects symbol information from the rewritten binary

   Symbol Table
   - SymName ➔ Addr
   - Virtual CPU ➔ Addr
   - SymName ➔ Addr

   in ExecutionEngine::EmitGlobals()

2. Finds virtual CPU address of the rewritten binary and stores it in Execution Engine of the LLI
# Sharing the Data storage for the virtual state

1. Collects symbol information from the rewritten binary

   Symbol Table
   
<table>
<thead>
<tr>
<th>SymName</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual CPU</td>
<td>Addr</td>
</tr>
<tr>
<td>SymName</td>
<td>Addr</td>
</tr>
</tbody>
</table>

2. Finds virtual CPU address of the rewritten binary and stores it in Execution Engine of the LLI

3. Updates virtual CPU address in the LLI

   Using `ExecutionEngine::updateGlobalMapping()`
• # Change program counter

1. Compares FuncName in llvm::Interpreter::callFunction()
   if (F-)getName() == "main_lifted"
Domain Transition: Implementation details

• # Change program counter

1. Compares FuncName
   in llvm::Interpreter::callFunction()
   if (F->getName() == "main_lifted")

2. Finds Addr
   Mapping Table
   FuncName -> Addr
   main_lifted -> Addr
   FuncName -> Addr
Domain Transition: Implementation details

- # Change program counter

```c
#define runTransitionFunction(FuncAddr){
    Store rsp, rbp
    (*FuncAddr)();
    Restore rsp, rbp
}
```

1. Compares FuncName in llvm::Interpreter::callFunction()
   if (F) getName() == "main_lifted"

2. Finds Addr
   - Mapping Table
     - FuncName -> Addr
     - main_lifted -> Addr

3. Calls FuncAddr
Domain Transition: Implementation details

1. Compares FuncName
   *in llvm::Interpreter::callFunction()*
   
   if (F->getName() == "main_lifted")

2. Finds Addr

3. Calls FuncAddr

4. Executes in rewritten binary code

```cpp
#define runTransitionFunction(FuncAddr)
{  
  Store rsp, rbp
  (*FuncAddr)();
  Restore rsp, rbp
}
```
Domain Transition: Implementation details

• # Change program counter

1. Compares FuncName
   in llvm::Interpreter::callFunction()
   if (Func->getName() == "main_lifted")

2. Finds Addr
   Mapping Table
   FuncName → Addr
   main_lifted → Addr
   FuncName → Addr

#define runTransitionFunction(FuncAddr){
  Store rsp, rbp
  (*FuncAddr)();
  Restore rsp, rbp
}

3. Calls FuncAddr

4. Executes in rewritten binary code

5. Returns to the LLI code when it encounters the ret instruction
Domain Transition: Implementation details

• # Change program counter

(Initialize) The Global Variable of binary to the Ili's trampoline address
• # Change program counter

(Initialize) The Global Variable of binary to the lli’s trampoline address

1. Dives on the lli trampoline when it encounters function start address on binary.
Domain Transition: Implementation details

• # Change program counter

  (Initialize) The GlobalVariable of binary to the lli’s trampoline address

  1. Dives on the lli trampoline when it encounters function start address on binary.

  2. Gets a value in virtual pc register
Domain Transition : Implementation details

• # Change program counter

(Initialize) The GlobalVariable of binary to the lli’s trampoline address

1. Dives on the lli trampoline when it encounters function start address on binary.

2. Gets a value in virtual pc register

3. Finds a FuncName

Mapping Table

<table>
<thead>
<tr>
<th>FuncName</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc_lifited</td>
<td>Addr</td>
</tr>
<tr>
<td>FuncName</td>
<td>Addr</td>
</tr>
</tbody>
</table>
### Domain Transition: Implementation details

- **# Change program counter**

1. Dives on the lli trampoline when it encounters function start address on binary.
2. Gets a value in virtual pc register
3. Finds a `FuncName`
4. Setups a `ExecutionContext`

```
struct ExecutionContext {
    Function *CurFunction;
    BasicBlock *CurBB;
    BasicBlock::iterator CurInst;
    CallSite Caller;
}
```

(Initialize) The `GlobalVariable` of binary to the lli’s trampoline address

```
size = ECStack.size()
ECStack.size() +
```

Mapping Table

<table>
<thead>
<tr>
<th>FuncName</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc_lifted</td>
<td>Addr</td>
</tr>
</tbody>
</table>
Domain Transition: Implementation details

- # Change program counter

1. Dives on the lli trampoline when it encounters function start address on binary.
2. Gets a value in virtual pc register
3. Finds a FuncName
4. Setups a ExecutionContext

```
struct ExecutionContext {
    Function  *CurFunction;
    BasicBlock  *CurBB;
    BasicBlock::iterator CurInst;
    CallSite  Caller;
}
```
5. Executes lli

```
while (size < ECstack.size())
    size = ECStack.size() ++
```

(Initialize) The GlobalVariable of binary to the lli’s trampoline address

Mapping Table
- FuncName — Addr
  - malloc_lifted — Addr

ECStack
- ExecutionContext
- ExecutionContext
- ExecutionContext

Domain Transition: Implementation details

- # Change program counter

1. Dives on the lli trampoline when it encounters function start address on binary.
2. Gets a value in virtual pc register
3. Finds a FuncName
4. Sets up a ExecutionContext
   ```
   struct ExecutionContext {
     Function *CurFunction;
     BasicBlock *CurBB;
     BasicBlock::iterator CurInst;
     CallSite Caller;
   }
   ```
5. Executes lli
6. Returns to binary

(Initialize) The GlobalVariable of binary to the lli's trampoline address

Mapping Table
- FuncName - Addr
  - malloc_lifted - Addr
### Experiment Result

#### Running time comparison

**Group A**

<table>
<thead>
<tr>
<th>FALSE</th>
<th>TRUE</th>
<th>nice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.434187</td>
<td>0.431787</td>
<td>0.509698</td>
</tr>
</tbody>
</table>

**Group B**

<table>
<thead>
<tr>
<th>FALSE</th>
<th>TRUE</th>
<th>nice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.656174</td>
<td>0.516916</td>
<td>0.543351</td>
</tr>
</tbody>
</table>

**Group C**

<table>
<thead>
<tr>
<th>FALSE</th>
<th>TRUE</th>
<th>nice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.573</td>
<td>0.661431</td>
<td>0.76253</td>
</tr>
</tbody>
</table>

The results of conducted domain transition on LLI:
**Experiment Result**

### Running time comparison

- **Result**
  - Rate of Change = \( \frac{T_L - T_B}{T_L} \times 100 \)
  - Percentage = \( \frac{T_L}{T_B} \)

  ※ \( T_L \): Only 1li execution, no domain transition
  ※ \( T_B \): Domain transition for “main_lifited” function

- Maximum 99.82% reduction on execution time
- Achieved outstanding performance improvement especially on programs with frequent file searching and looping
- Conducted Domain Transition for lifted main function

<table>
<thead>
<tr>
<th>logname</th>
<th>Rate of Change</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>3.82%</td>
<td>x1.06</td>
</tr>
<tr>
<td>true</td>
<td>5.96%</td>
<td>x1.25</td>
</tr>
<tr>
<td>nice</td>
<td>31.23%</td>
<td>x1.45</td>
</tr>
</tbody>
</table>

| date     | 81.07%         | x5.28      |
| hostid   | 81.12%         | x5.3       |
| groups   | 87.40%         | x7.94      |
| id       | 95.95%         | x24.7      |
| ls       | 98.72%         | x78.31     |
| vdir     | 99.57%         | x230.82    |
| df       | 99.82%         | x542.53    |
DEMO
Before Transitioning Execution Domain

```
Command: ./webserver-elf ./bin/arm/ls.arm translated ./bin/arm/ls.arm
log: note: Load the bitcode..
IR Parsing Time 17.456044sec
lli: note: Ready
(lli-master) functions main [1]
  Functions input : main
    __uClIBC main 0x4bb7c_lifted = 0x3ea6e0
    main 0x1178c_lifted = 0x86ed0
    __libc_start_main@GLIBC_2.2.5 = 0x0
    setdomainname@GLIBC_2.2.5 = 0x0
    main = 0x472ad0
(lli-master) tp main 0x1178c_lifted IMI on [3]
```

After Transitioning Execution Domain

```
Command: ./exec.sh arm ls
Target ARCH : arm
Target Binary : ls
cmd: ./webserver-elf ./bin/arm/ls.arm translated ./bin/arm/ls.arm
log: note: Load the bitcode..
IR Parsing Time 17.454475sec
lli: note: Ready
(lli-master) functions main [1]
  Functions input : main
    __uClIBC main 0x4bb7c_lifted = 0x3ea6e0
    main 0x1178c_lifted = 0x86ed0
    __libc_start_main@GLIBC_2.2.5 = 0x0
    setdomainname@GLIBC_2.2.5 = 0x0
    main = 0x472ad0
(lli-master) tp main 0x1178c_lifted IMI on [3]
```
k@ubuntu:~/Downloads/webserver-master$ ./exec.sh arm ls
Target ARCH : arm
Target Binary : ls
cmd: ./webserver -elf ./bin/arm/ls.arm.translated ./bin/arm/ls.arm.linked.ll
log: note: Load the bitcode..
IR Parsing Time 17.367615sec
ll: note: Ready

k@ubuntu:~/Downloads/webserver-master$ ./test q
"Quit":"All processes died."}k@ubuntu:~/Downloads/webserver-master$
Thank you