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

Jaeyong Ko, Sangrok Lee, Jieun Lee, Jaewoo Shim

Table of contents

1. Background
2. LLI Interpreter mode
3. Binary Rewriting
4. Our idea
5. Domain Transition
6. Experiment Result
7. Demo

Background

- **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 ⟫ intel, ARM, MIPS Technologies, PowerPC, AMD ⟫ 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**

※ clang -emit-llvm -fno-discard-value-names -S test.c -o test.ll

_test.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;
}
```

Code Symbols

Source Code

_test.ll

```
...
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
}

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
}
```

Code Symbols

LLVM IR from Source Code

Background : Binary Lifting

- **Lifted IR and Binary comparison**

_test

```
...
 0x400c20 <doo>:     push   %rbp
 0x400c21 <doo+1>:mov    %rsp,%rbp
 0x400c24 <doo+4>:callq  0x400c10 <foo>
 0x400c29 <doo+9>:add    $0x1,%eax
 0x400c2c <doo+12>:          pop    %rbp
 0x400c2d <doo+13>:          retq
...
 0x400c30 <car>:     push   %rbp
 0x400c31 <car+1>: mov    %rsp,%rbp
 0x400c34 <car+4>: callq  0x400c20 <doo>
...
```

Binary

_test.linked.ll (Lifted IR)

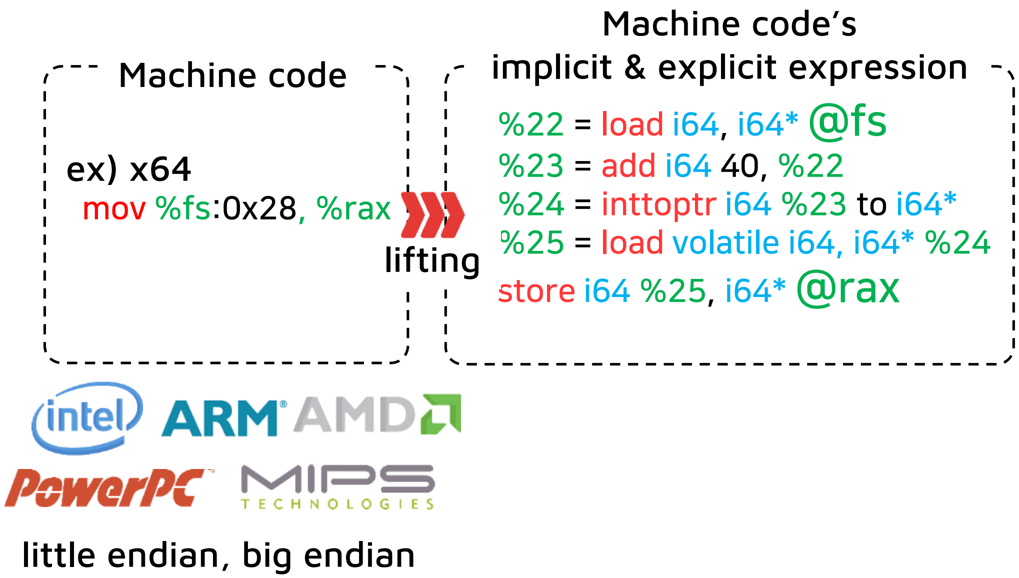
```
bb.car:
 %10 = load i64, i64* @rbp
 %11 = load i64, i64* @rsp
 %12 = sub i64 %11, 8
 %13 = inttoptr i64 %12 to i64*
 store volatile i64 %10, i64* %13
 store i64 %12, i64* @rsp
 %14 = load i64, i64* @rsp
 store i64 %14, i64* @rbp
 %15 = load i64, i64* @rsp
 %16 = sub i64 %15, 8
 %17 = inttoptr i64 %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
...
```

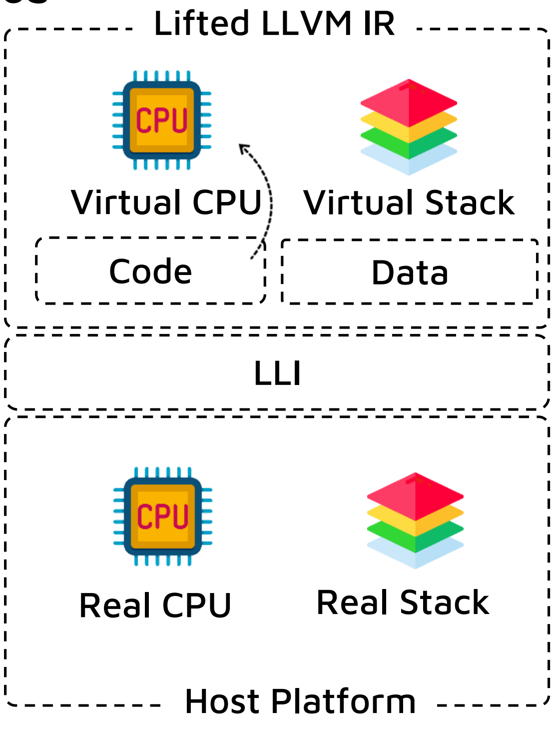
LLVM IR from Binary

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

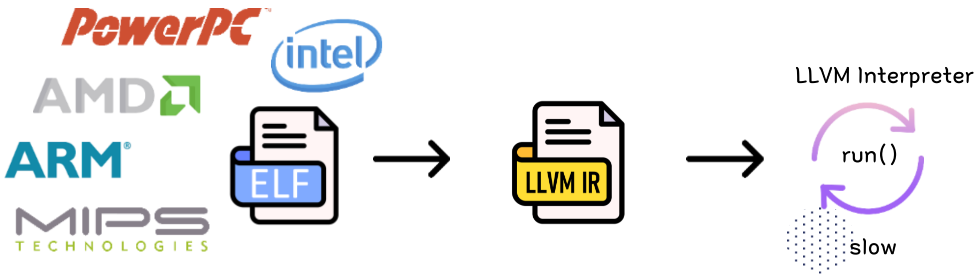
※ A Cross Debugger for Multi-Architecture Binaries(euro-llvm2020)





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.

```
Breakpoint 2, 0x0000000000400cbf in main ()
1: x/20i $rip
=> 0x400cbf <main>: push   %rbp
   0x400cc0 <main+1>:    mov    %rsp,%rbp
   0x400cc3 <main+4>:    sub    $0x30,%rsp
   0x400cc7 <main+8>:    mov    %fs:0x28,%rax
   0x400cd0 <main+17>:   mov    %rax,-0x8(%rbp)
   0x400cd4 <main+21>:   xor    %eax,%eax
   0x400cd6 <main+23>:   movabs $0x31646165726854,%rax
   0x400ce0 <main+33>:   mov    %rax,-0x18(%rbp)
   0x400ce4 <main+37>:   movabs $0x32646165726854,%rax
   0x400cee <main+47>:   mov    %rax,-0x10(%rbp)
   0x400cf2 <main+51>:   movq   $0x0,-0x20(%rbp)
   0x400cfa <main+59>:   lea    -0x18(%rbp),%rdx
```

4 machine codes
→19 lines

```
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?**

```
bb.main:
%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
 %19 = load i64, i64* @rsp
 store i64 %19, i64* @rbp

 %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

 %22 = load i64, i64* @reg_288_64
 %23 = add i64 40, %22
```

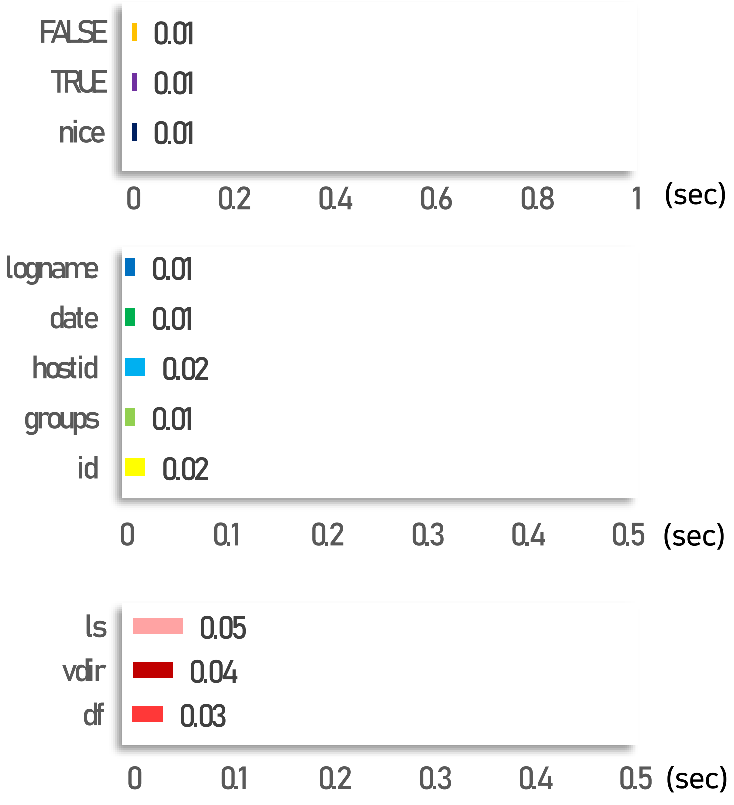
```
void Interpreter::visitLoadInst(LoadInst &I) {
  ExecutionContext &SF = ECStack.back();
  GenericValue SRC = getOperandValue(I.getPointerOperand(), SF);
  GenericValue *Ptr = (GenericValue*)GVTOP(SRC);
  GenericValue Result;
  LoadValueFromMemory(Result, Ptr, I.getType());
  SetValue(&I, Result, SF);
  if (I.isVolatile() && PrintVolatile)
    dbgs() << "Volatile load " << I;
}
```

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

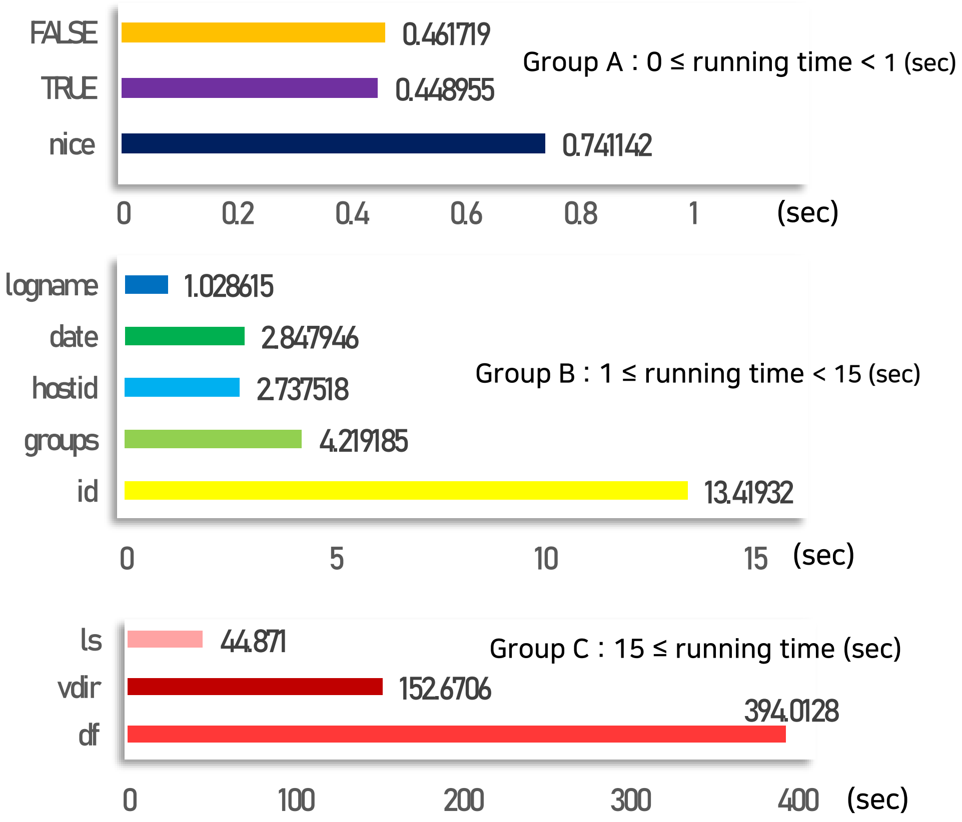
Accumulated overhead results badly on running time

LLI Interpreter Mode : Running Time

- **Running time comparison**



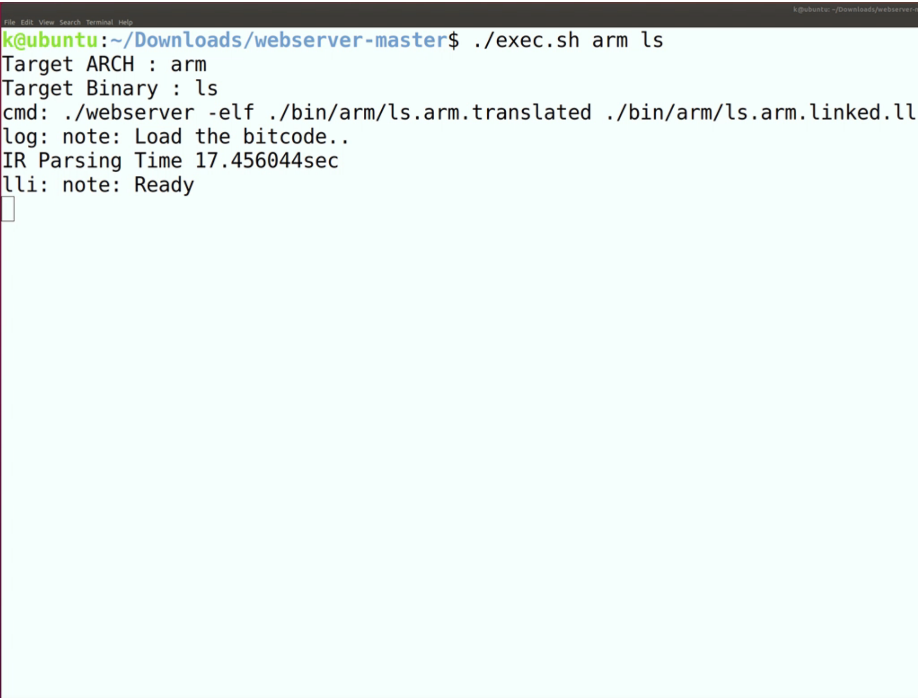
Original Binary



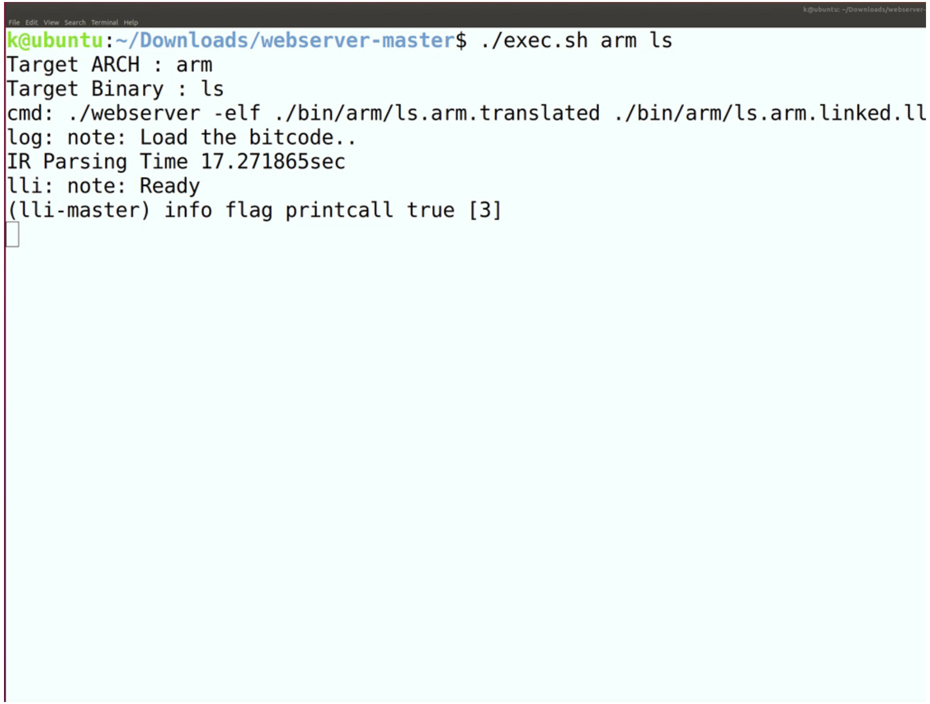
LLI

DEMO

- **LLI Execution Demo**



Default Execution(No debug info)

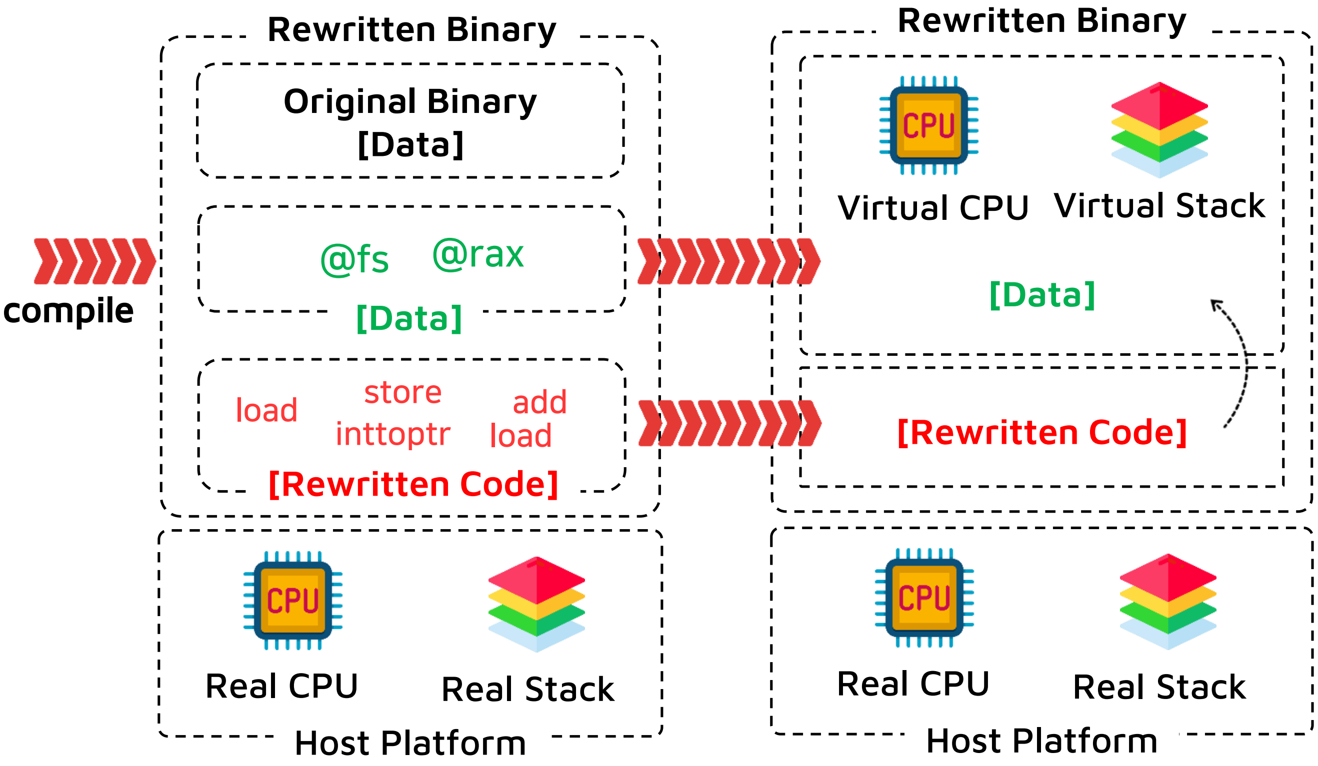


Print function call flow

Binary Rewriting

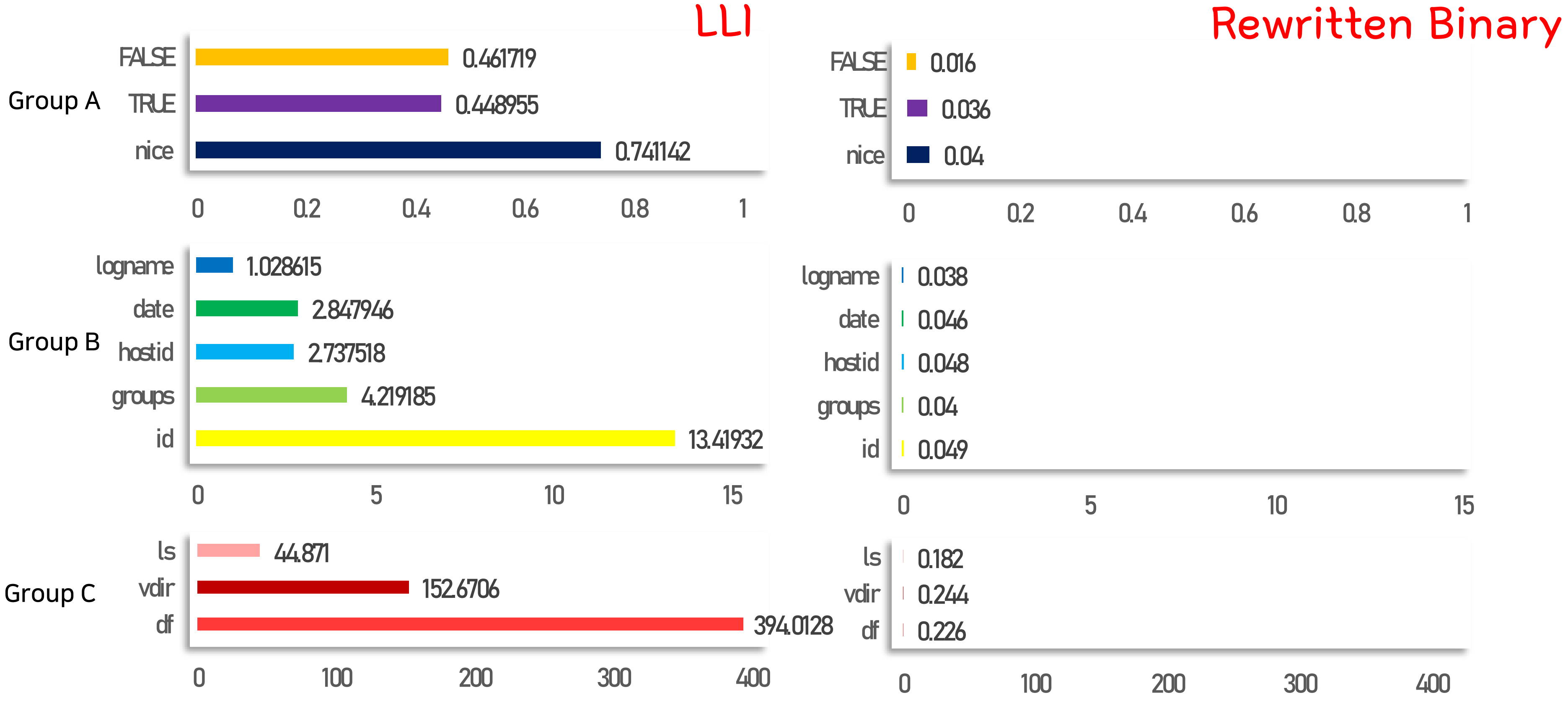
- **Rewritten Binary Architecture**

```
%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
```



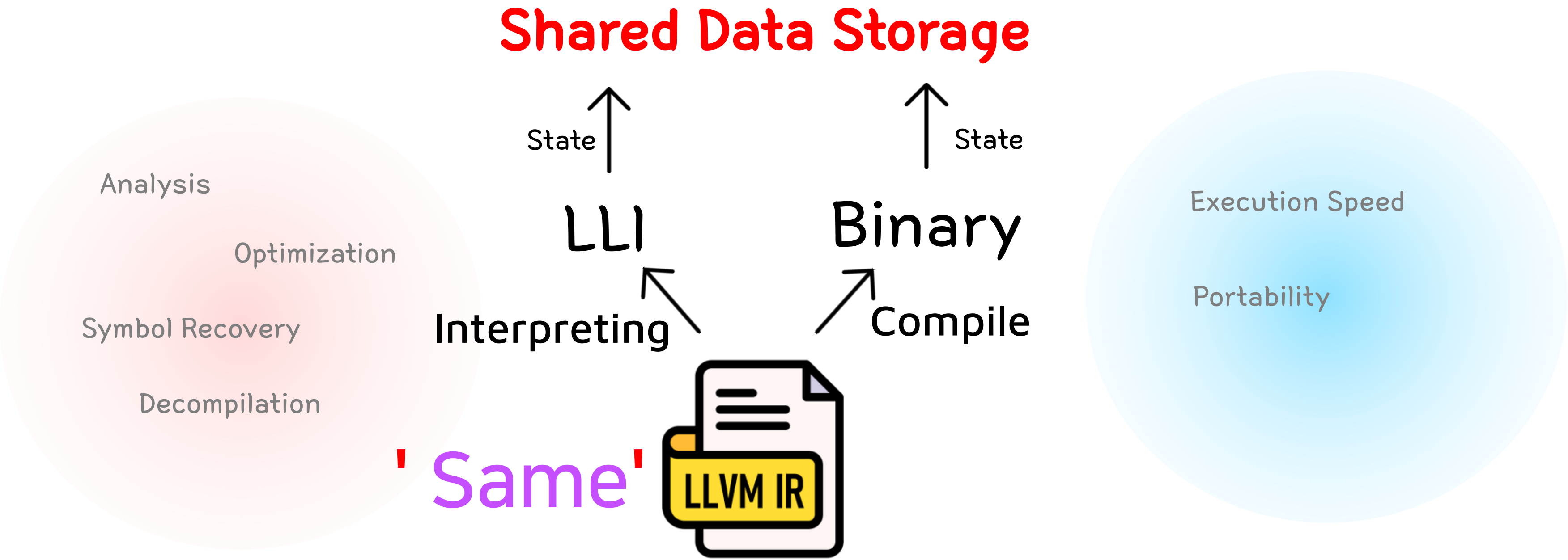
Binary Rewriting : Running Time

- **Running time comparison**

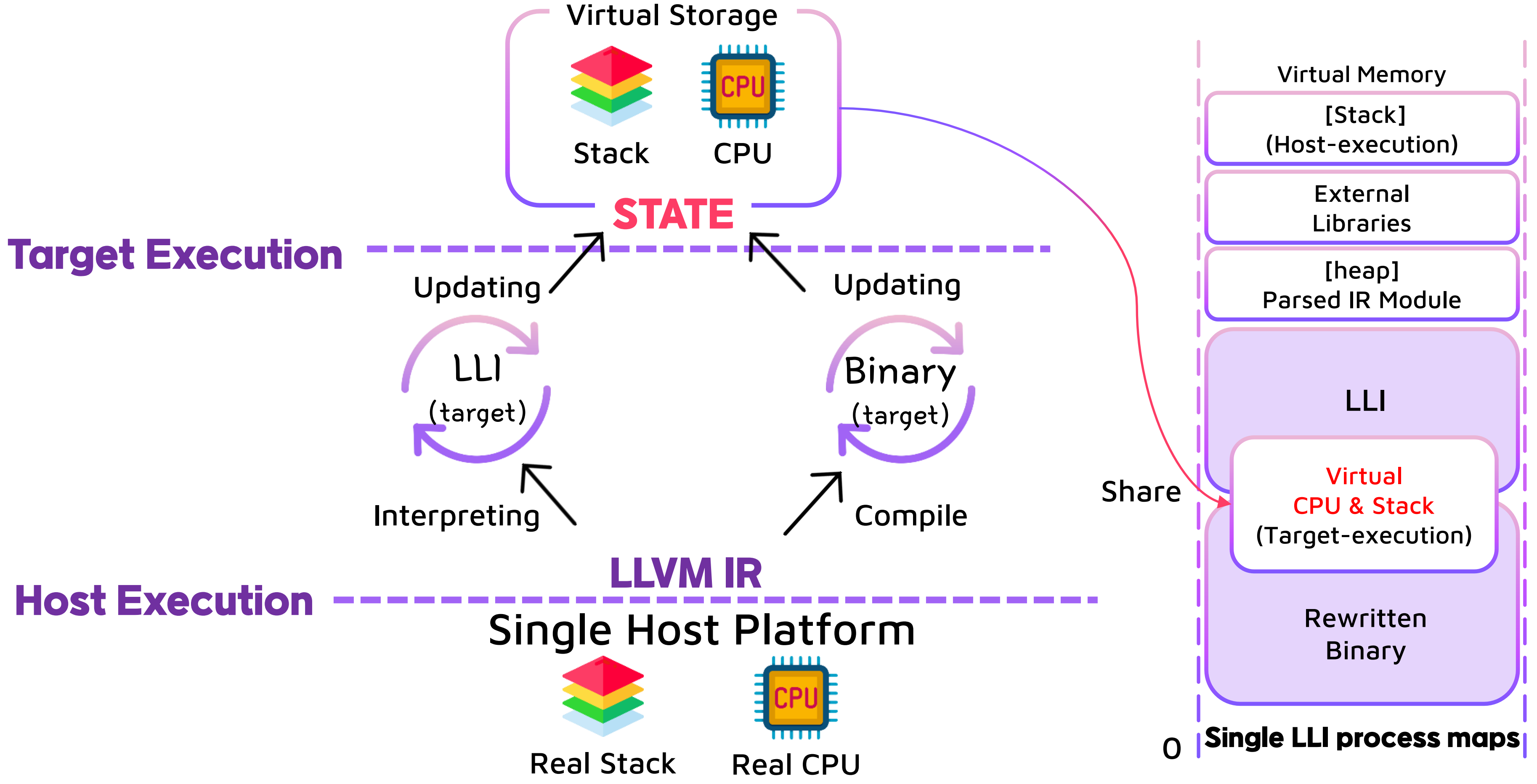


Our idea

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

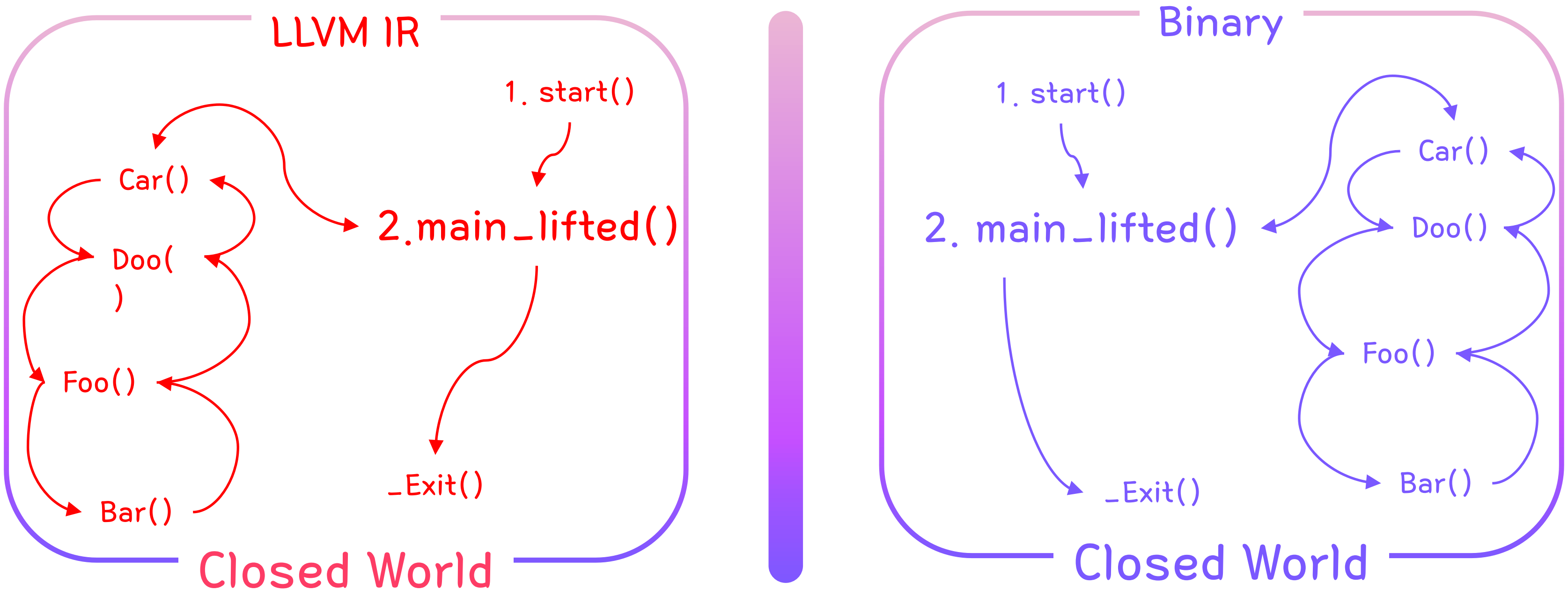


Domain Transition : LLI & Binary Execution



Domain Transition : IR World & Binary World

- **Two closed worlds (IR vs Binary)**
  - The IR and Binary domain execute for each in closed world.



Analysis

Optimization

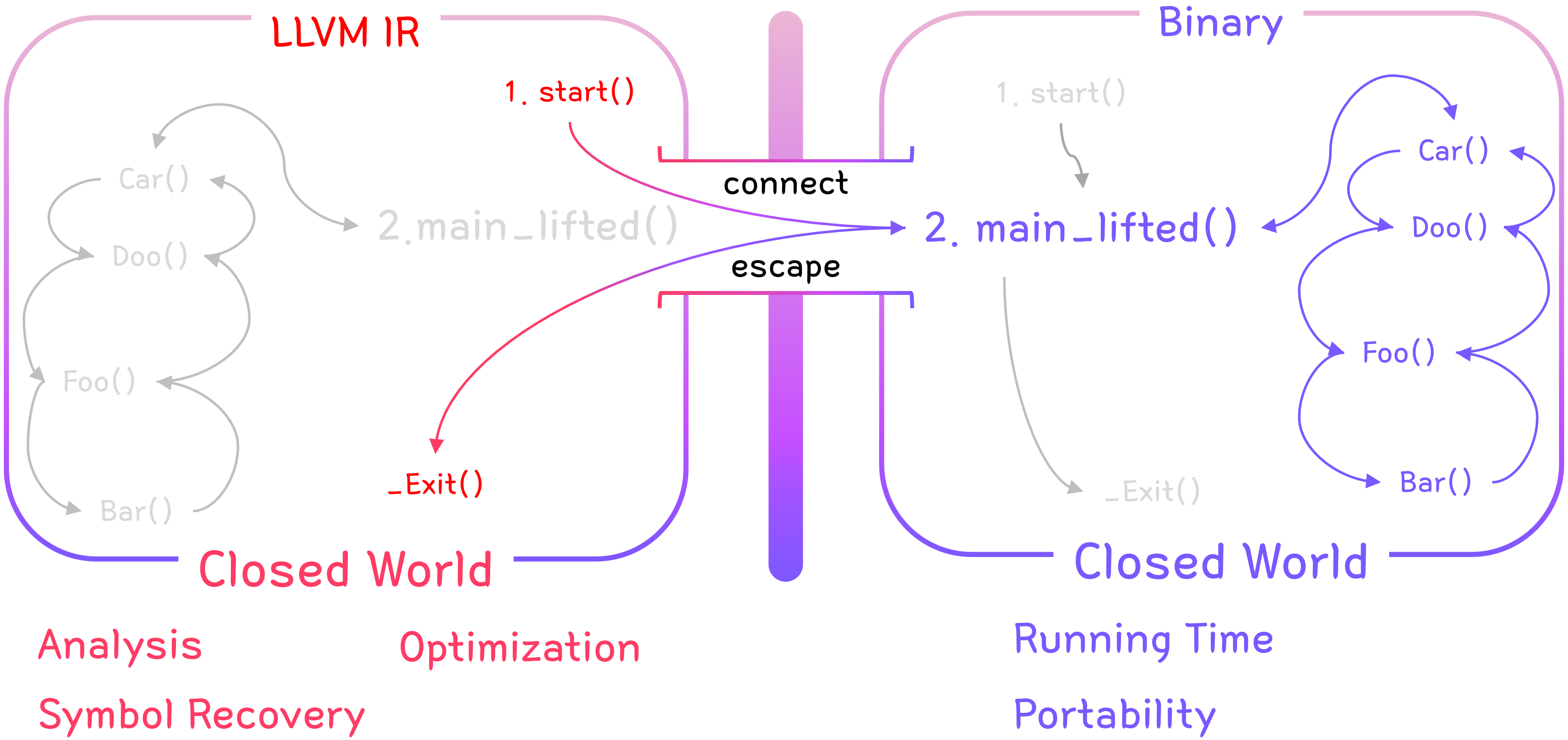
Symbol Recovery

Running Time

Portability

Domain Transition : IR World & Binary World

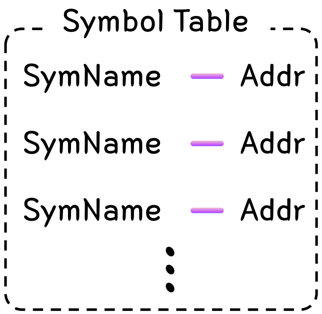
- **Two closed worlds (IR vs Binary)**
  - Can take benefits of the other closed world using domain transition.

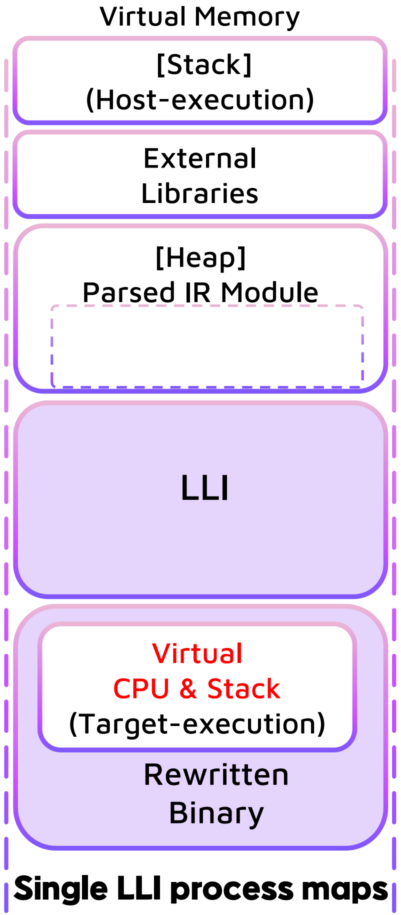


Domain Transition : Implementation details

- **# Sharing the Data storage for the virtual state**

1. Collects symbol Information from the rewritten binary

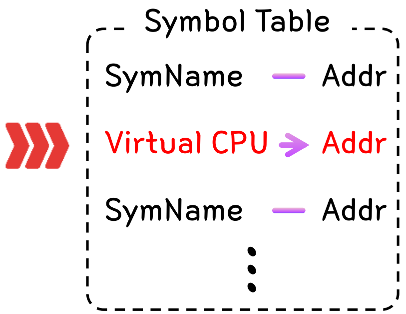




Domain Transition : Implementation details

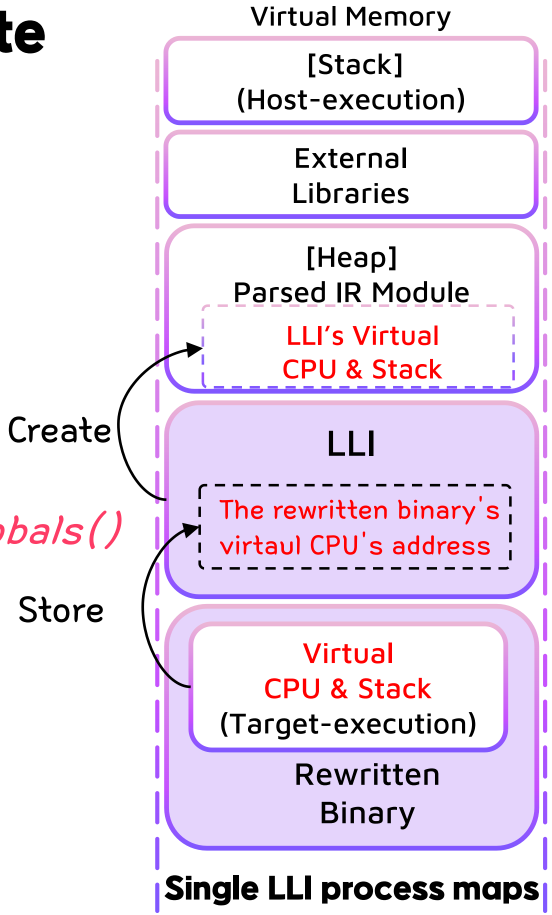
- **# Sharing the Data storage for the virtual state**

1. Collects symbol Information from the rewritten binary



in ExecutionEngine::EmitGlobals()

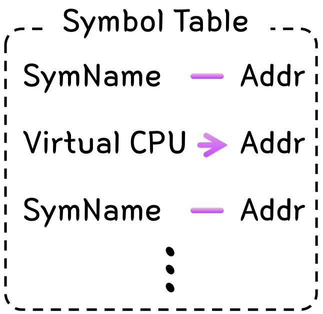
2. Finds virtual CPU address of the rewritten binary and Stores it in Execution Engine of the LLI



Domain Transition : Implementation details

- **# Sharing the Data storage for the virtual state**

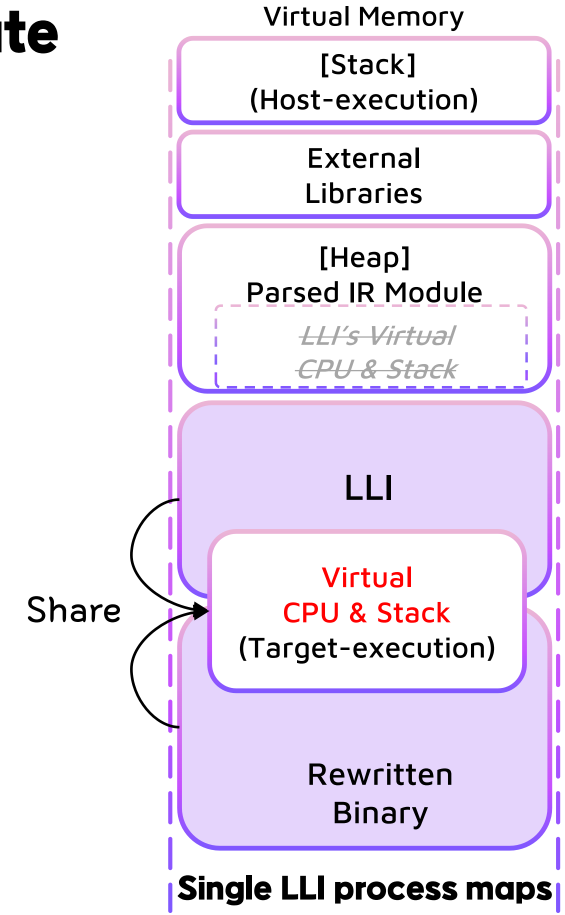
1. Collects symbol Information from the rewritten binary



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

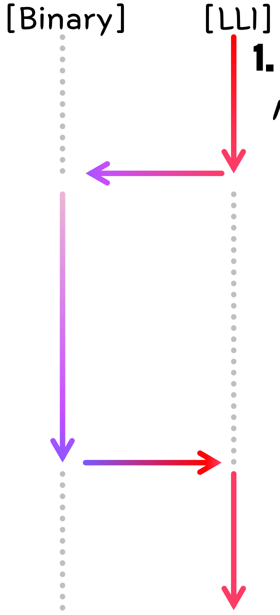
3. Updates virtual CPU address in the LLI

Using ExecutionEngine::updateGlobalMapping()



Domain Transition : Implementation details

- **# Change program counter**



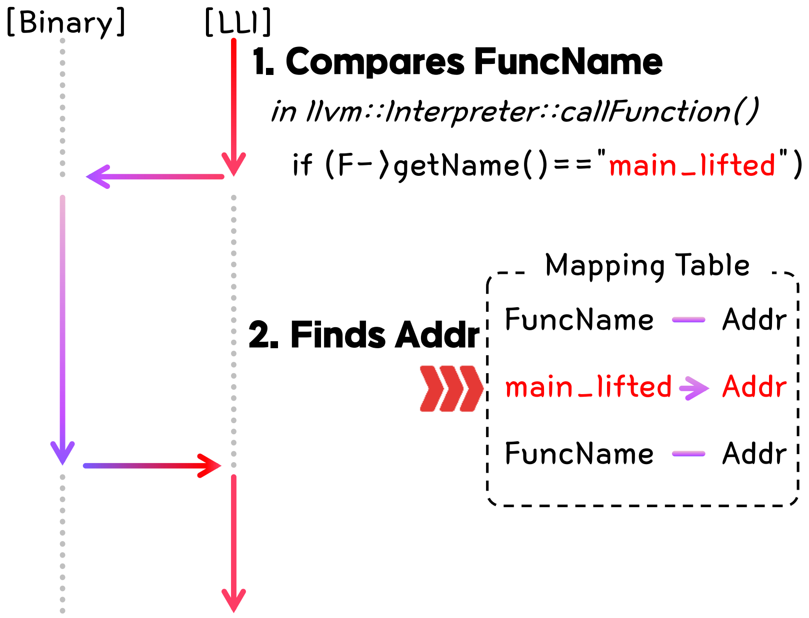
1. Compares FuncName

in llvm::Interpreter::callFunction()

```
if (F->getName()=="main_lifted")
```

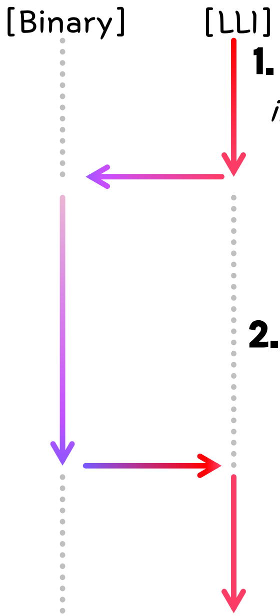
Domain Transition : Implementation details

- **# Change program counter**



Domain Transition : Implementation details

- **# Change program counter**



1. Compares FuncName

in llvm::Interpreter::callFunction()

```
if (F->getName()=="main_lifted")
```

2. Finds Addr

Mapping Table

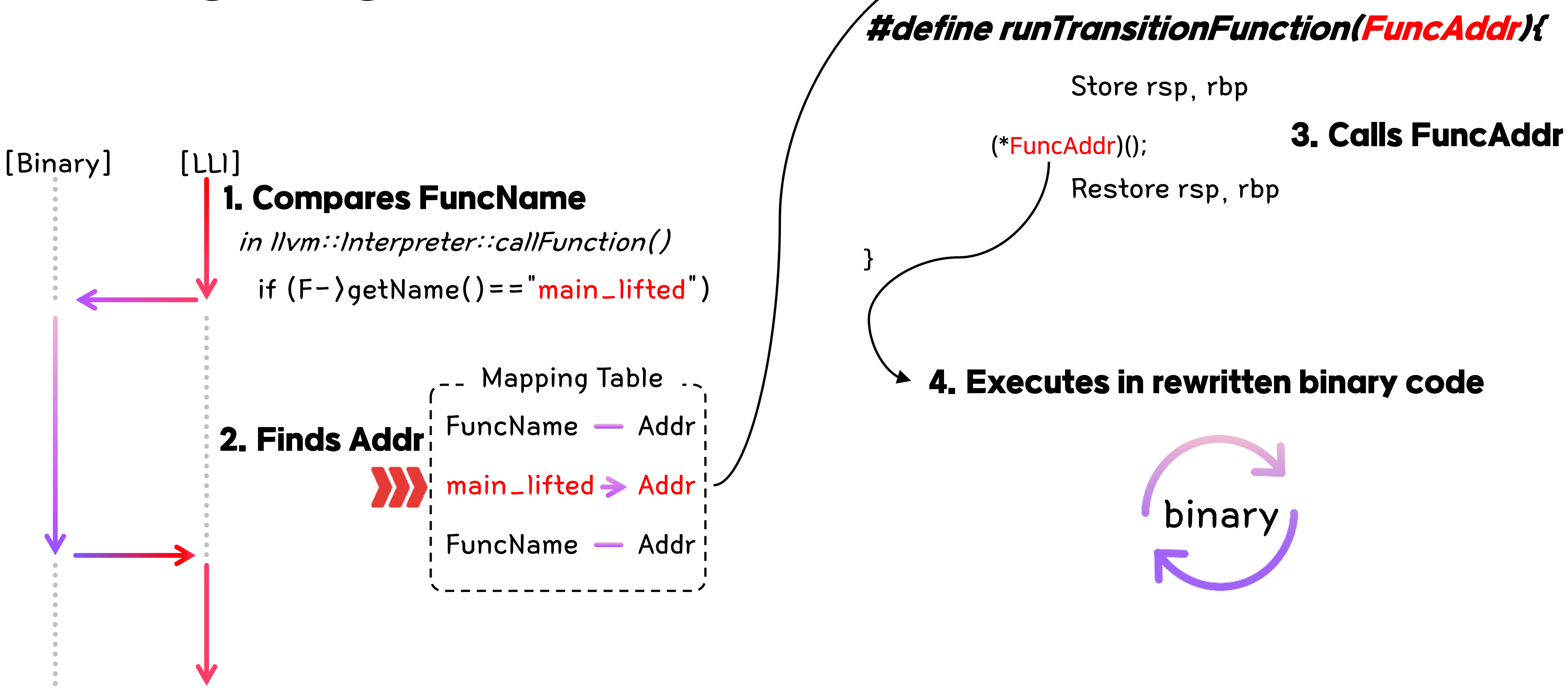
FuncName	Addr
main_lifted	Addr
FuncName	Addr

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

3. Calls FuncAddr

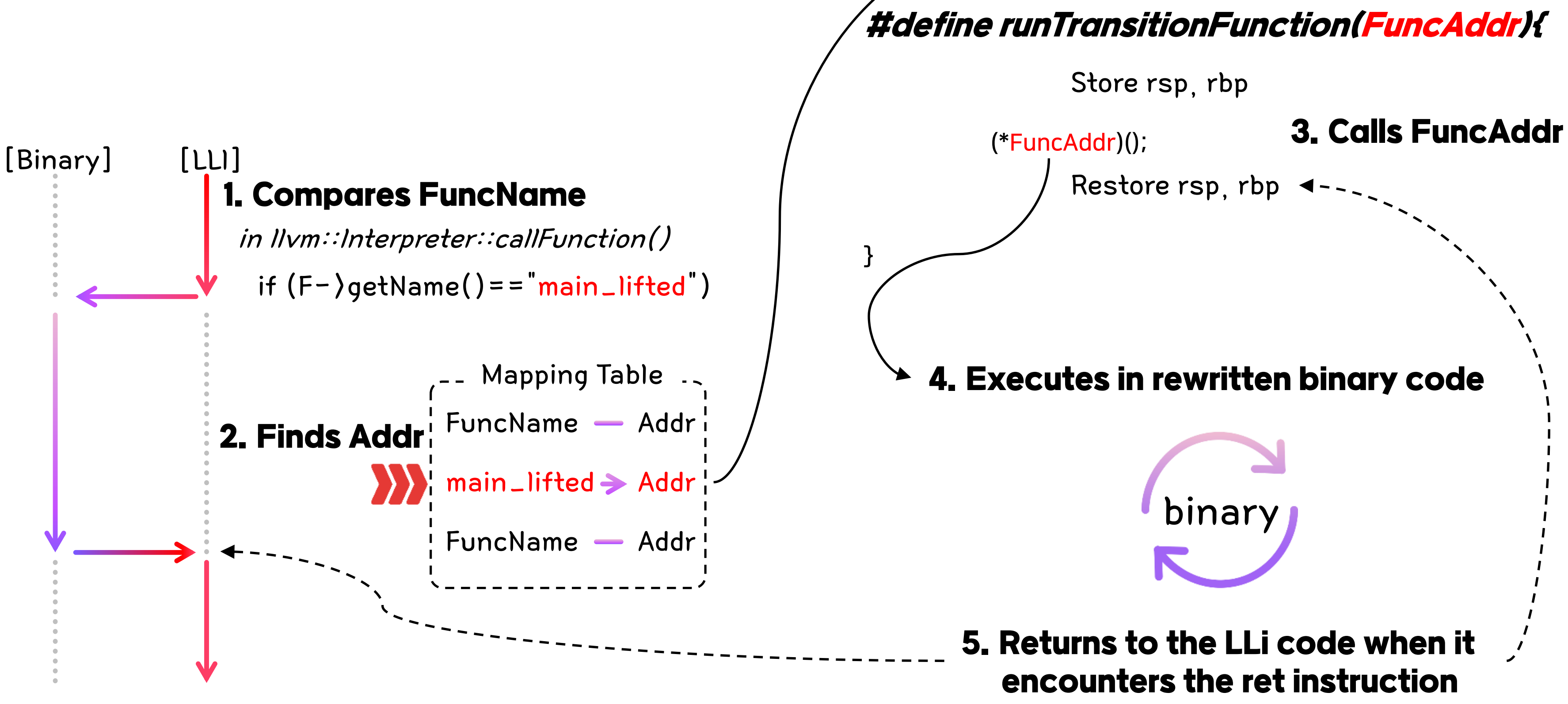
Domain Transition : Implementation details

- **# Change program counter**



Domain Transition : Implementation details

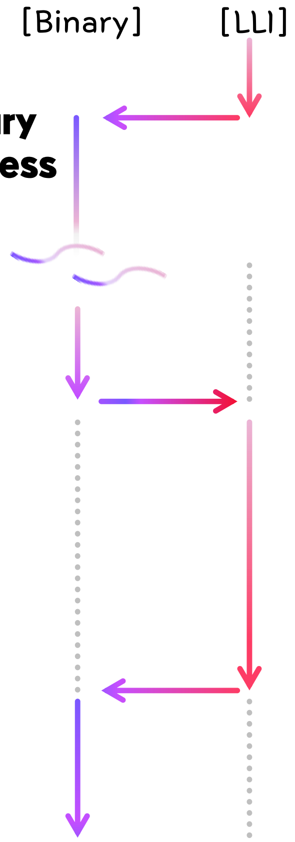
- **# Change program counter**



Domain Transition : Implementation details

- **# Change program counter**

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

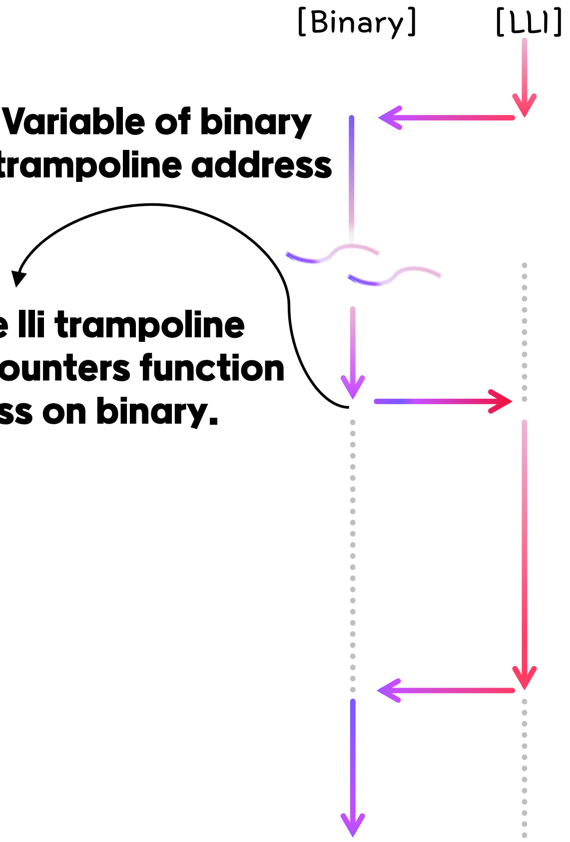


Domain Transition : Implementation details

- **# Change program counter**

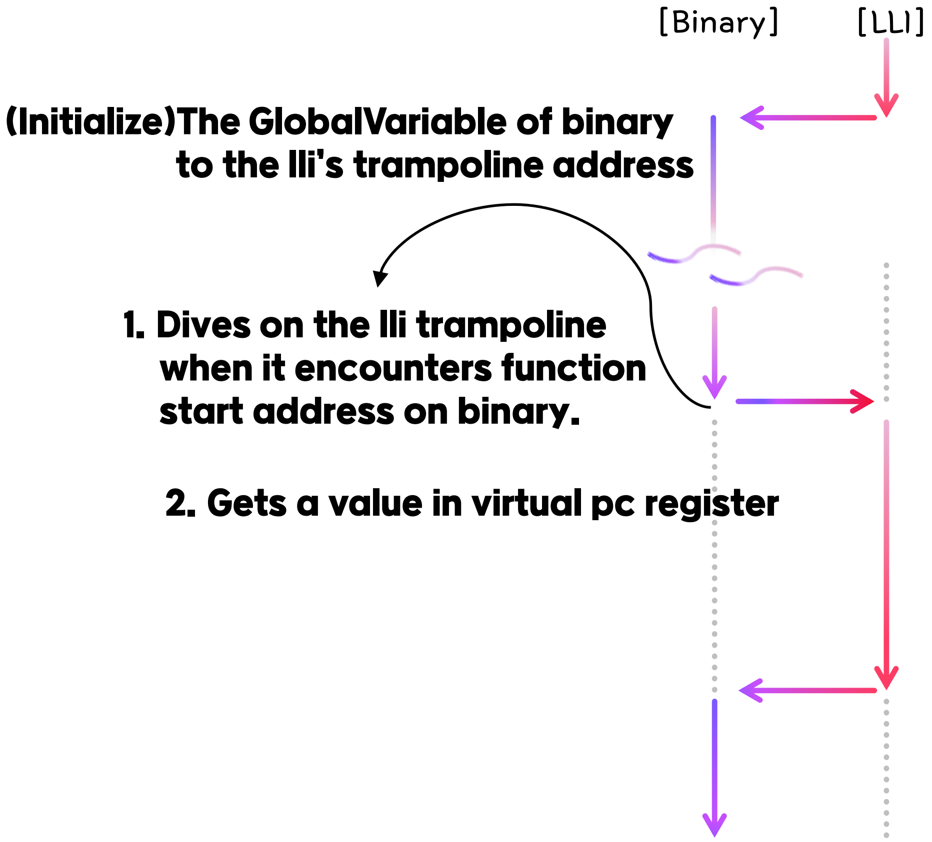
(Initialize)The GlobalVariable of binary to the lli's trampoline address

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



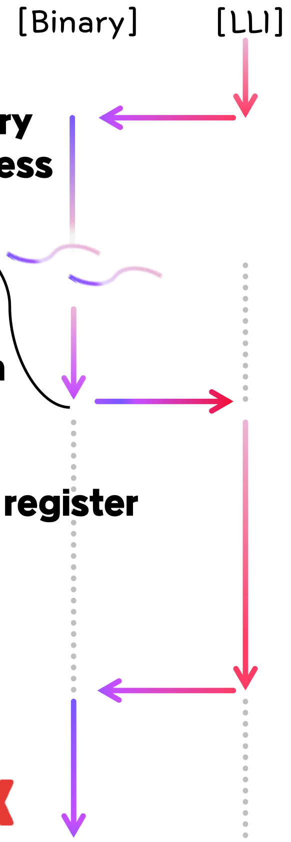
Domain Transition : Implementation details

- **# Change program counter**



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

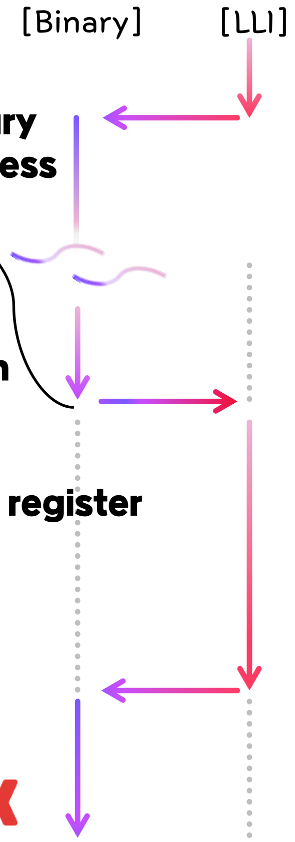
3. Finds a FuncName

Mapping Table

FuncName	—	Addr
malloc_lifted	←	Addr
FuncName	—	Addr

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

4. Setups a ExecutionContext

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

Mapping Table

FuncName	—	Addr
malloc_lifted	←	Addr
FuncName	—	Addr

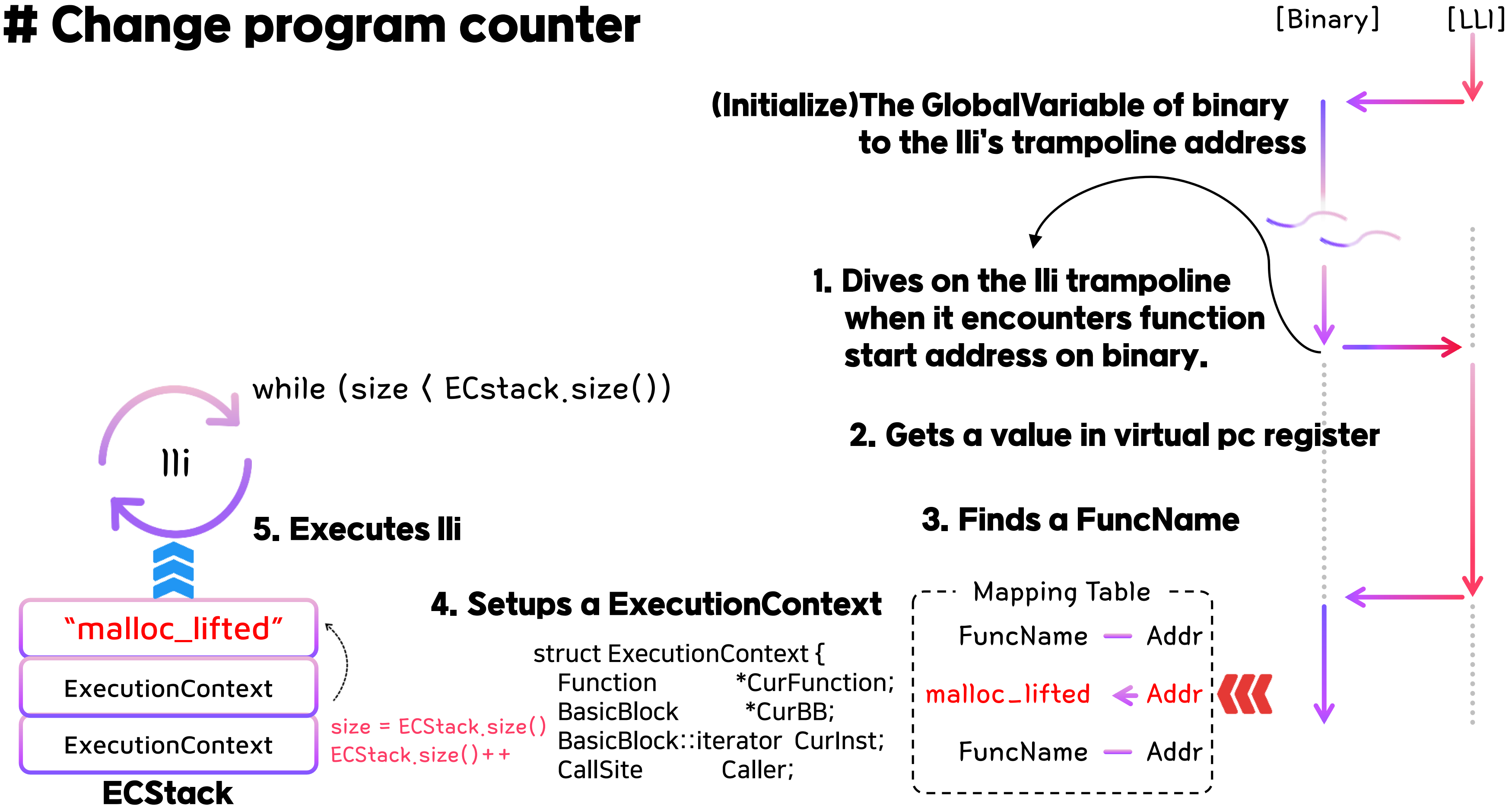
ECStack
"malloc_lifted"
ExecutionContext
ExecutionContext

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

ECStack

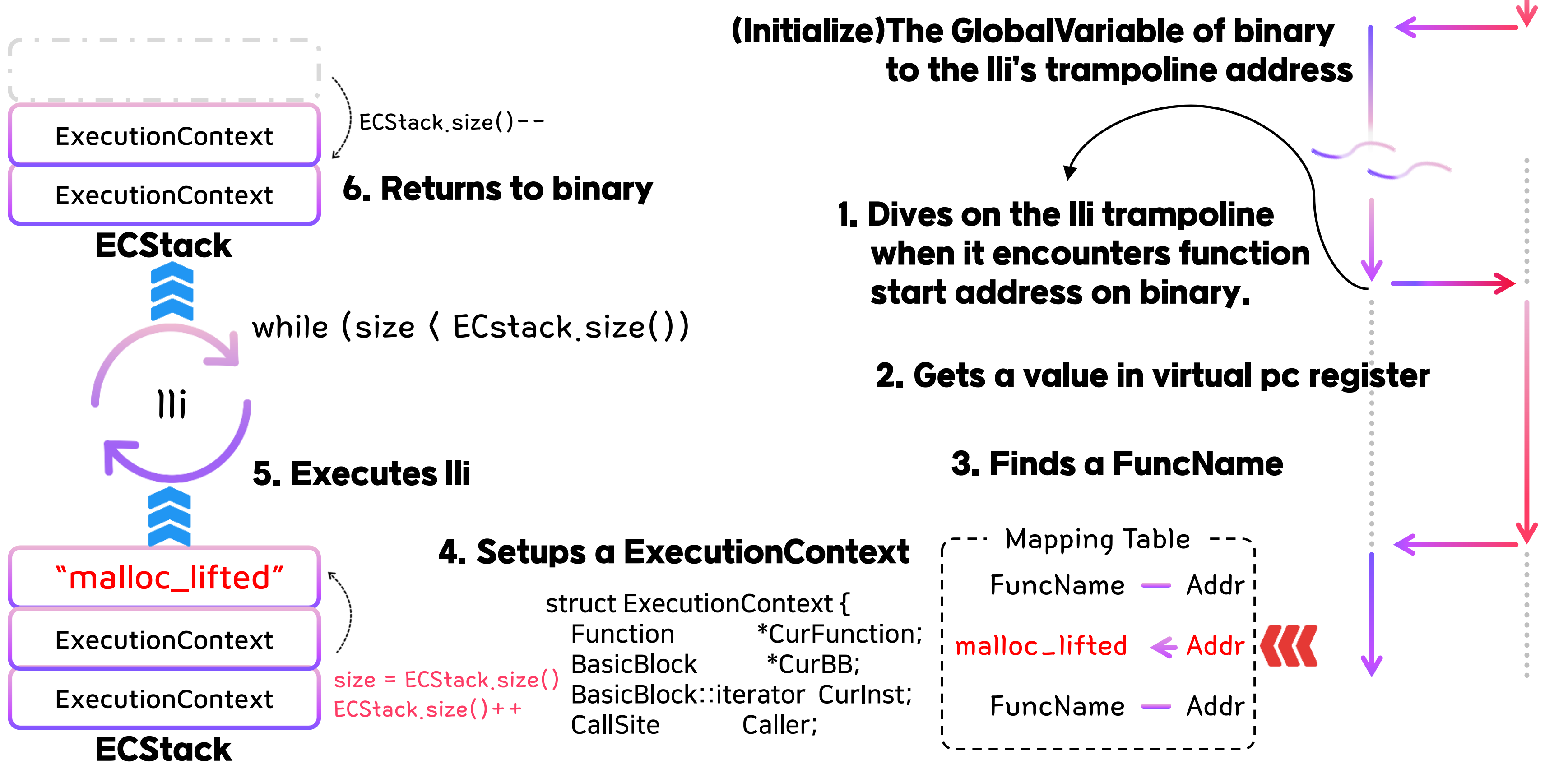
Domain Transition : Implementation details

- # Change program counter



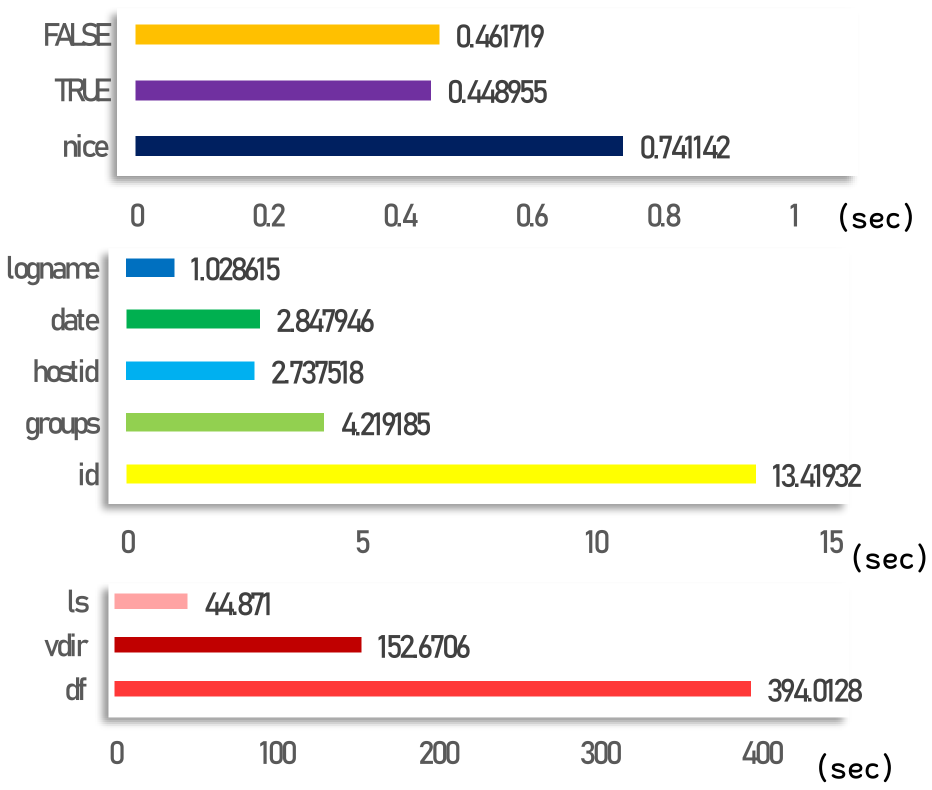
Domain Transition : Implementation details

- **# Change program counter**

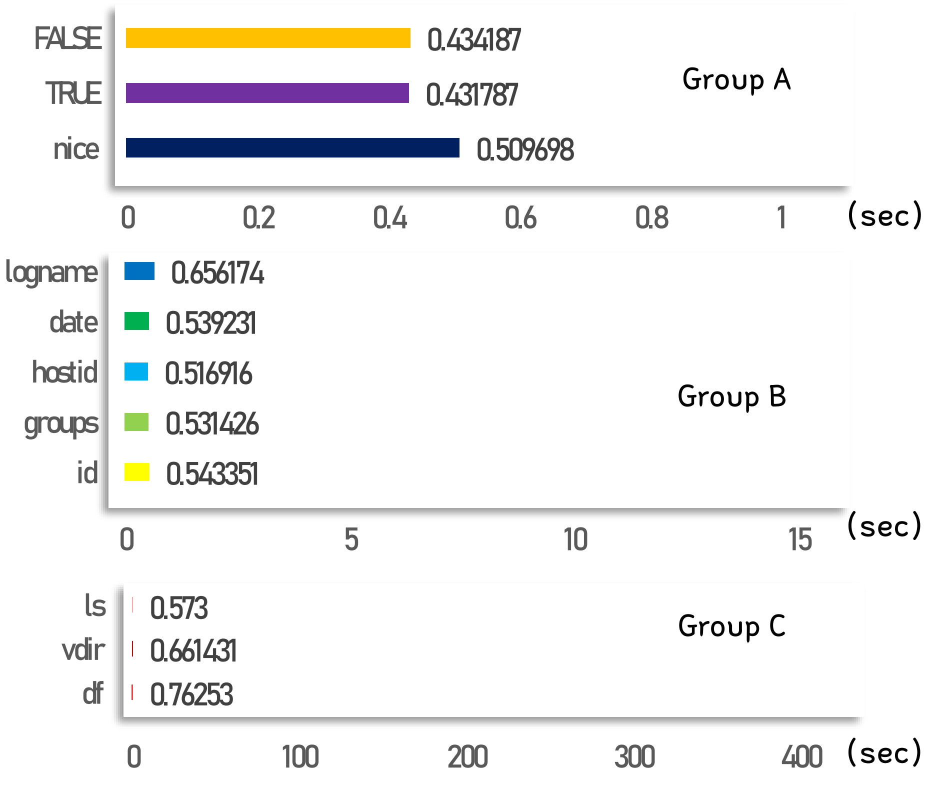


Experiment Result

- **Running time comparison**



The results of LLI



The results of conducted domain transition on LLI

Experiment Result

- **Running time comparison**
  - Result
    - Rate of Change = $(T_L - T_B)/T_L * 100$
    - Percentage= T_L / T_B

      ※ T_L : Only lli execution, no domain transition

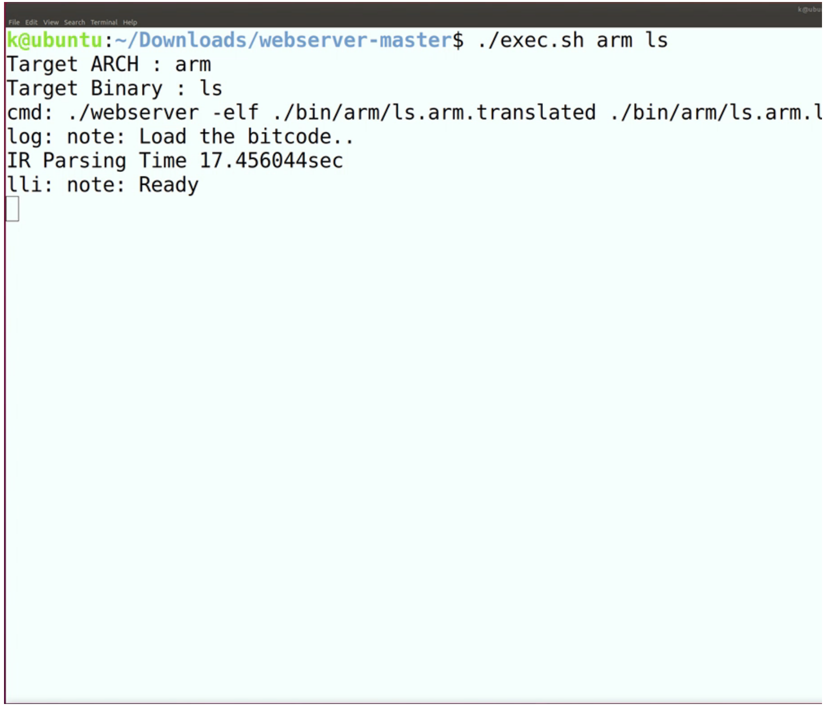
      ※ T_B : Domain transition for "main_lifted" function
  - Maximum 99.82% reduction on execution time
  - Achieved outstanding performance improvement especially on programs with frequent file searching and looping
  - conducted Domain Transition for lifted main function

	Rate of Change	Percentage
false	3.821%	x1.06
true	5.96%	x1.25
nice	31.23%	x1.45
logname	36.21%	x1.57
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

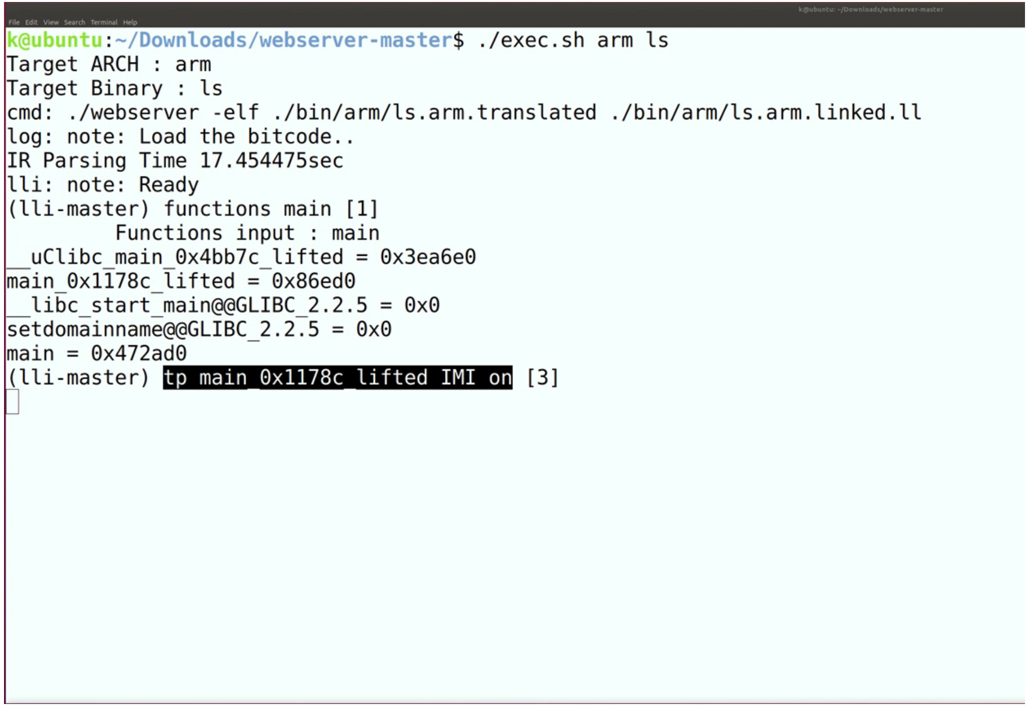
DEMO

DEMO

Demo



Before Transitioning Execution Domain



After Transitioning Execution Domain

Demo

```
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
lli: note: Ready
```

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

END

Thank you

bbxiix.4.4@gmail.com

Q & A

bbxiix.4.4@gmail.com