Implementing an LLVM based Dynamic Binary Instrumentation framework

34C3:tuwat!

Quarkslab
SECURING EVERY BIT OF YOUR DATA

Charles Hubain

Cédric Tessier

Introduction to Instrumentation

What is Instrumentation?

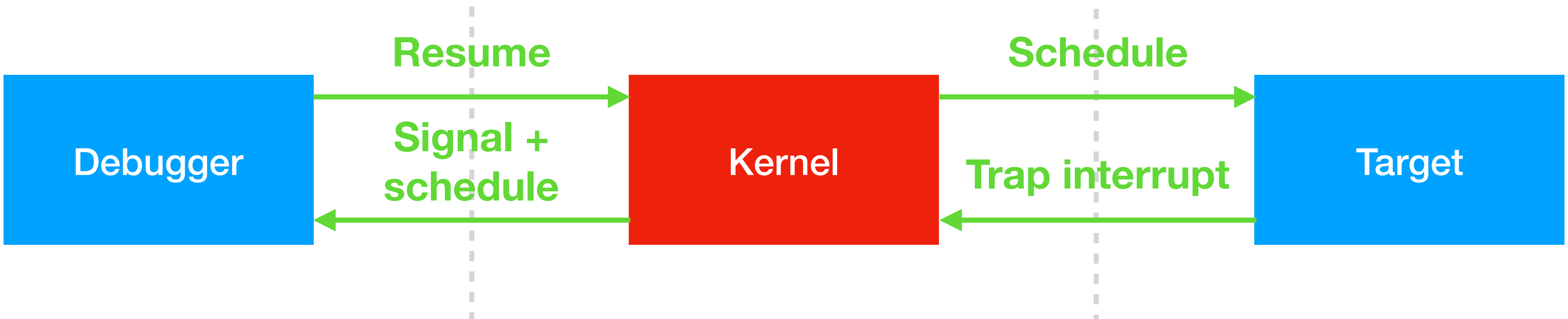
- *“Transformation of a program into its own measurement tool”*
- Observe **any state** of a program **anytime** during runtime
- **Automate** the data collection and processing

Use Cases

- Finding memory bugs:
  - Track memory allocations / deallocations
  - Track memory accesses
- Fuzzing:
  - Measure code coverage
  - Build symbolic representation of code
- Recording execution traces
  - Replay them for “timeless” debugging
  - Software side-channel attacks against crypto

"Why not … debuggers?"

- Debuggers are awesome but slooooooooow



```
[haxelion@elarion]~/documents/QB/esiea_ese_2017/demo %  python attack_gdb.py
```

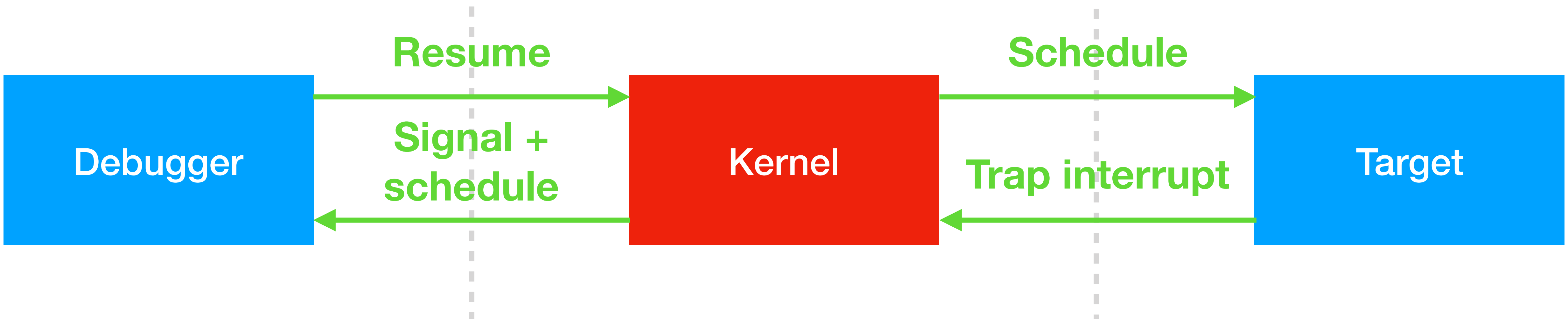
https://asciinema.org/a/17nynlopg5a18e1qps3r9ou7g

```
 python attack_pin.py

[haxelion@elarion]~/documents/QB/esiea_ese_2017/demo %  python attack_pin.py
d                                 2201228
```

“Why not … debuggers?”

- Debuggers are awesome but sloooooooow



- Solution? Get rid of the kernel
- How? Run the instrumentation **inside** the target

Instrumentation Techniques

- From source cod
  - Manually, y… now … `printf(…)` **BORING**
  - At comp… tim
- From binary:
  - Static binary patching & hooking — **Crude and barbaric**
  - Dynamic Binary Instrumentation — **This talk**

Existing Frameworks

- Valgrind since 2000
  - Open source, only *nix platforms, very complex
- DynamoRIO since 2002
  - Open source, cross-platforms, very raw
- Intel Pin since 2004
  - Closed source, only Intel platforms, user friendly

“Why we made our own”

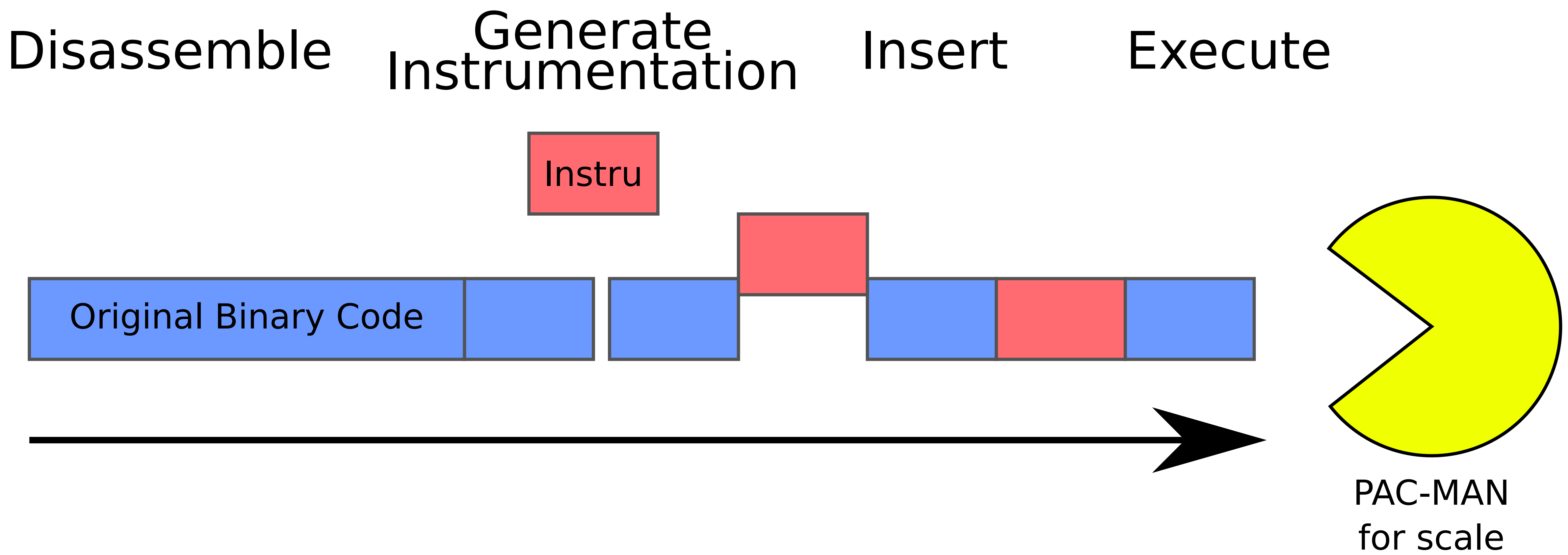
What we wanted from a DBI framework in 2015

- Cross-platform and cross-architecture
- Mobile and embedded targets support
- Simpler and modular design
- Focus on “heavy” instrumentation

Introduction to DBI

Dynamic Binary Instrumentation

- Dynamically insert the instrumentation at runtime



Disassembling

- What part of the binary is the code is **unknown**

  ➡ Disassembling the **whole binary** in advance is **impossible**

- We need to **discover** the code **as we go**

Code Discovery

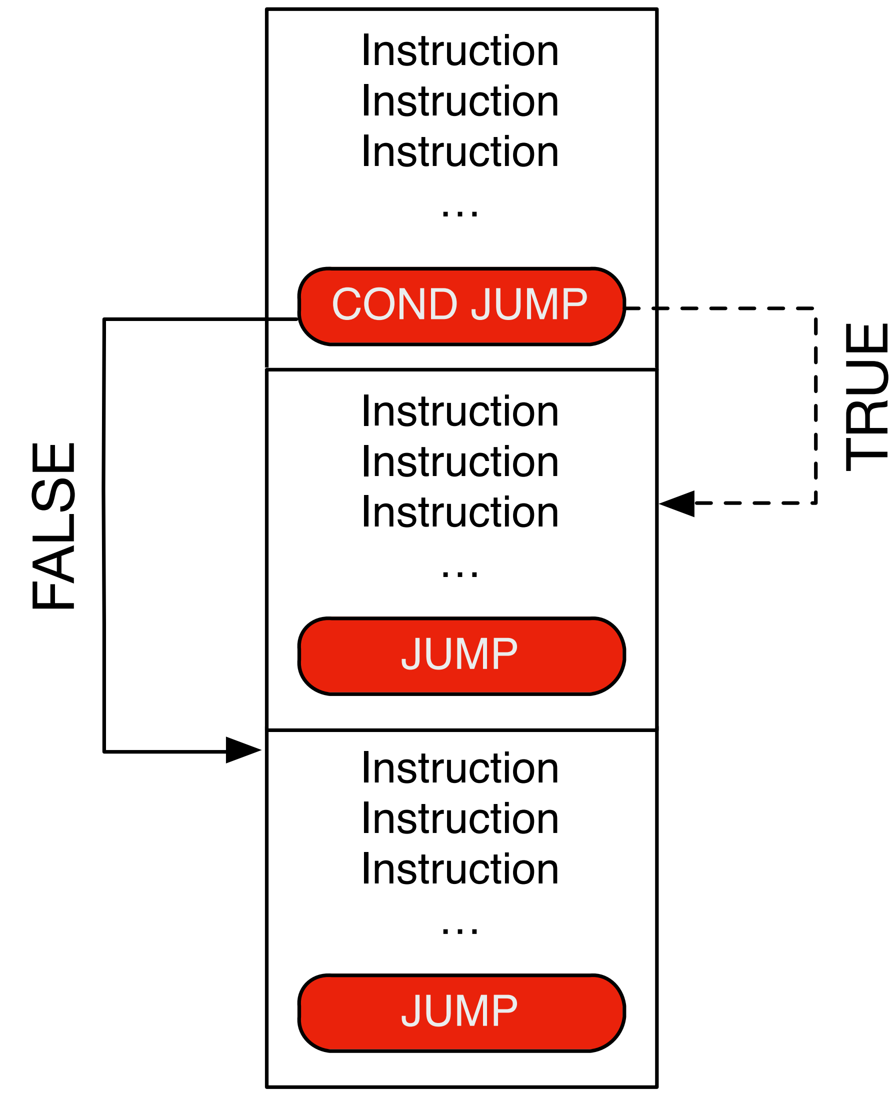
- How?
  - Execute a **block of code**
  - **Discover** where the execution flow after the block
  - Execute the next block of code
- This forms a short execution **cycle**

No Free Space

- The instrumented code is **larger** than the original code
- Binaries are usually **tightly packed** with little free space

➡ The instrumentation cannot be inserted **in-place**

➡ It needs to be "**relocated**"



Relocating

- Code contains **relative** reference to memory addresses
- These become **invalid** once we move the code
- We need to completely **rewrite** the code to **fix** those references

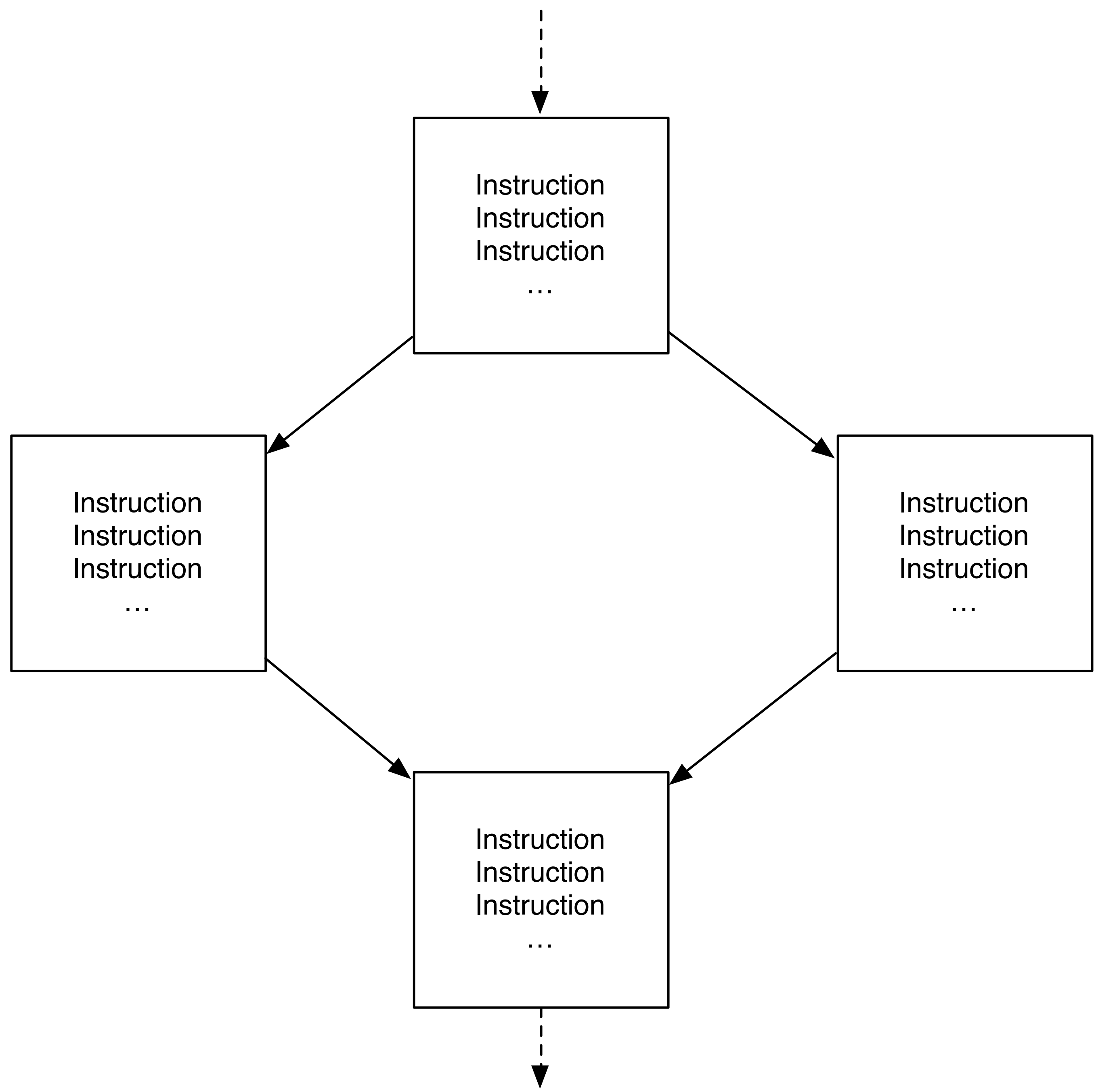
➡ This is what we call “**patching**”

The “Cycle of Life”

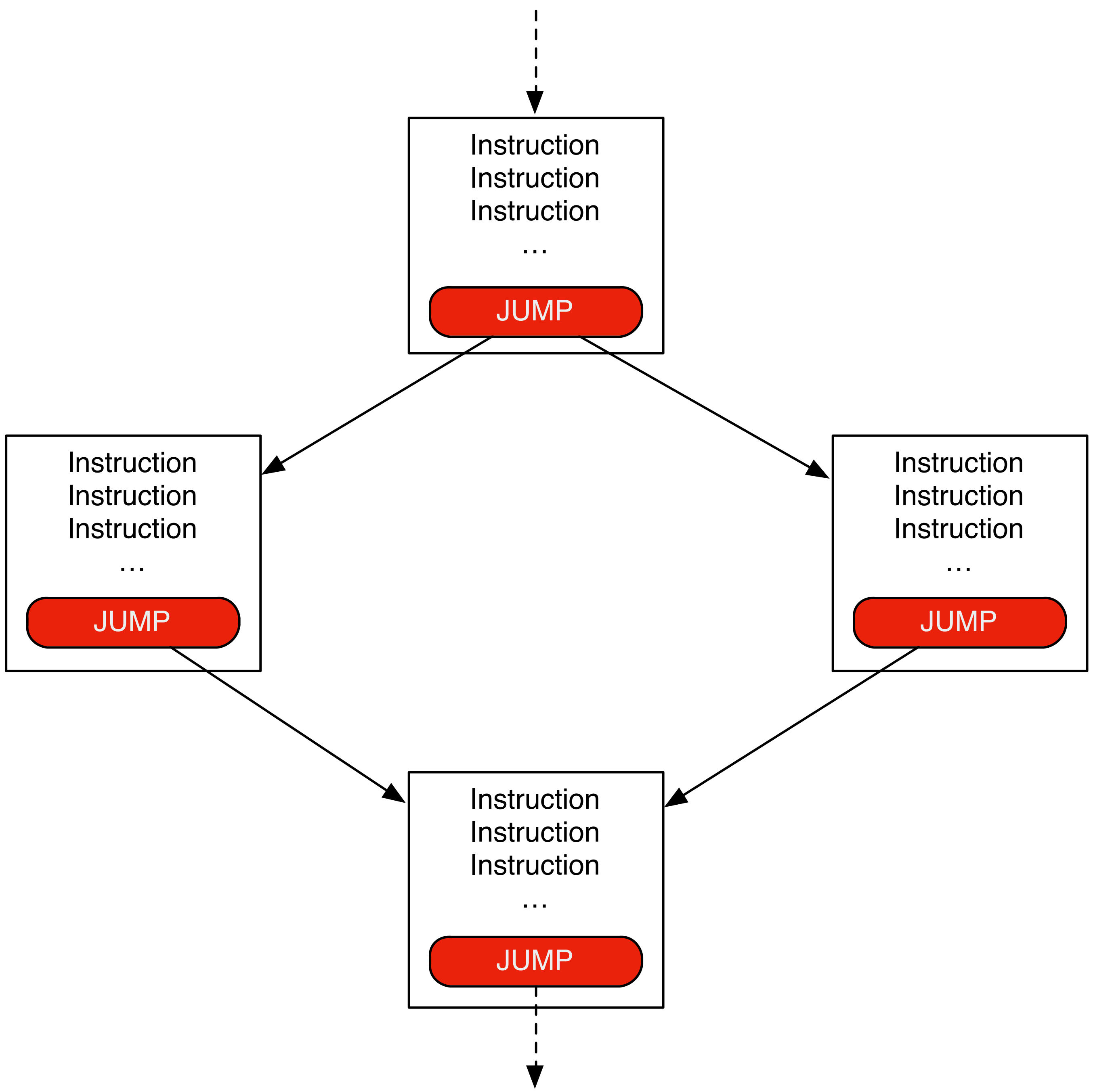
Designing a DBI:

1. Low Level Abstractions

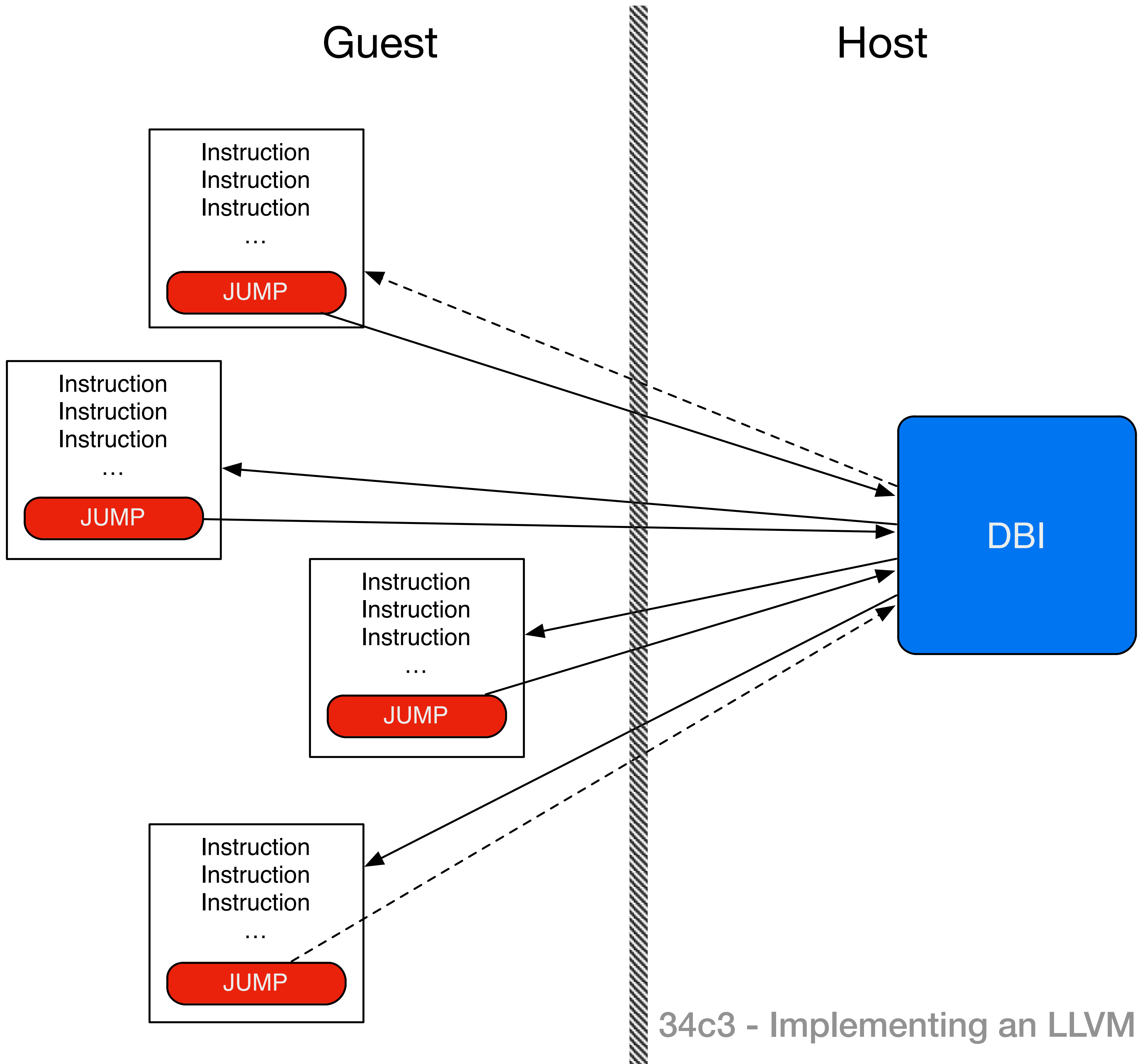
Basic Blocks



Control Flow



Under Control Flow



Under Control

DBI is all about **keeping control** of the execution

Under Control

- Keeping control of the execution
  - requires **modifying** original instructions…
  - …without modifying original **behaviour**

What We Need

- A multi-architecture **disassembler**
- A multi-architecture **assembler**
- A **generic** intermediate representation to apply modifications on

We Don't Want

Actually we don’t have 10 years and unlimited ressources

- To implement a multi-architecture **disassembler** and **assembler**
- To abstract every single **instruction semantic**
  - Architectures Developer Manuals are not that fun…

Here Be Dragons

This has nothing to do with 26C3

LLVM

Bleeding Edge Compiler Technology

To the rescue

- LLVM already has everything
  - It supports all major architectures
  - It provides a **disassembler** and an **assembler**…
  - …and both work on the same **intermediate representation**
- LLVM Machine Code (aka MC) to the rescue

LLVM MC

Instruction

```
movq    rax, 42
```

Binary

```
[0x48,0x89,0x04,0x25,0x2a,0x00,0x00,0x00]
```

LLVM MC

```
<MCInst #1670 MOV64mr
  <MCOperand Reg:0>
  <MCOperand Imm:1>
  <MCOperand Reg:0>
  <MCOperand Imm:42>
  <MCOperand Reg:0>
  <MCOperand Reg:35>>
```

LLVM MC

- It’s minimalist
- It’s totally **generic**
    - still encodes a lot of things about an instruction
- But very **raw**
    - **genericness** means some heavy **compromises**
    - doesn’t encode everything about an instruction

Creation

Every instruction is encoded using the **same representation**…

… but in a **different way**

```
movq [rip+0x2600], rax
```

```
<MCInst #1139 MOV64mr
 <MCOperand Reg:41>
 <MCOperand Imm:1>
 <MCOperand Reg:0>
 <MCOperand Imm:0x2600>
 <MCOperand Reg:0>
 <MCOperand Reg:35>>
```

Modification

jmp 0x41424242

```
<MCInst #1141 JMP_1
 <MCOperand Imm: 0x41424242>>
```

jmp [rip+0x2600]

```
<MCInst #1139 JMP64m
 <MCOperand Reg:41>
 <MCOperand Imm:1>
 <MCOperand Reg:0>
 <MCOperand Imm:0x2600>
 <MCOperand Reg:0>>
```

Patch

```
0x410000:   mov r0, [r0+pc]          ; Load a value relative to PC
```

Patch

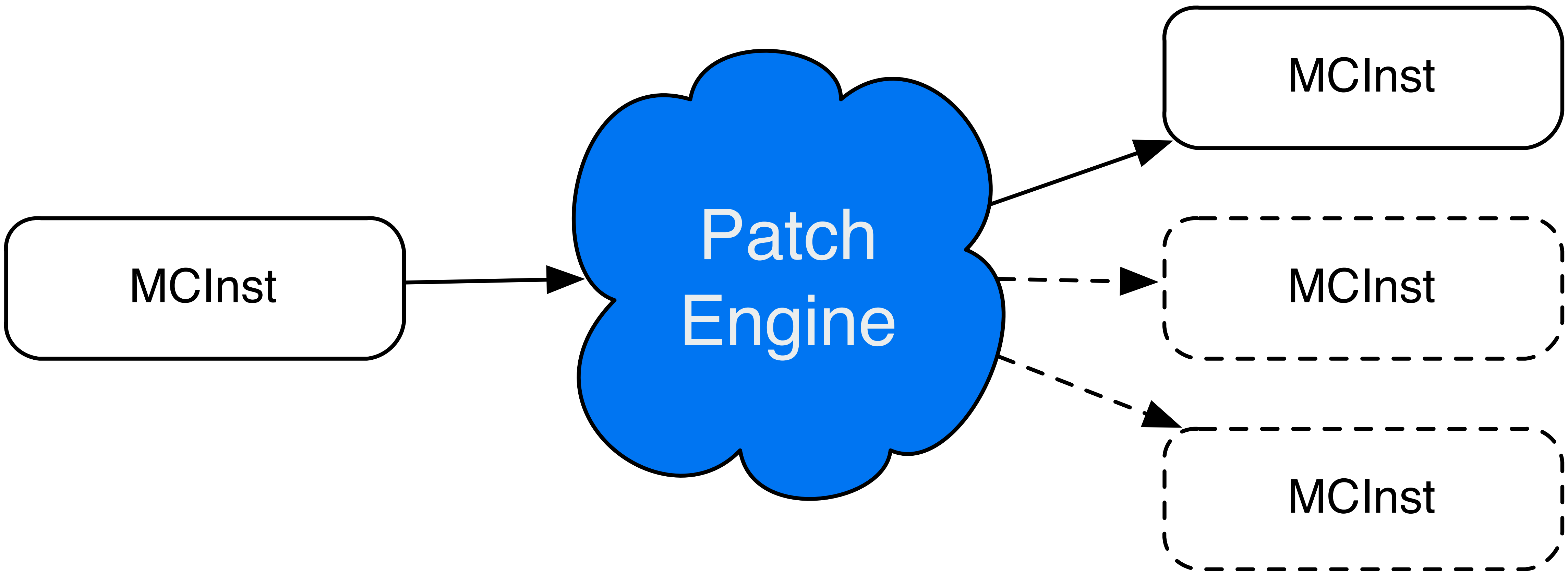
```
            mov [pc+0x2600], r1    ; Backup R1
            mov r1, 0x410000       ; Set original instruction address
0x7f10000:  mov r0, [r0+r1]        ; Load a value relative to R1
            mov r1, [pc+0x2600]    ; Restore R1
```

Abstractions

- MCInst **encoding** make transformations **painful**
- **Patches** can be really **complex**
- Many transformations are composed of **generic steps**

➡ **we need abstractions**

Patch Engine



Abstractions Inside™

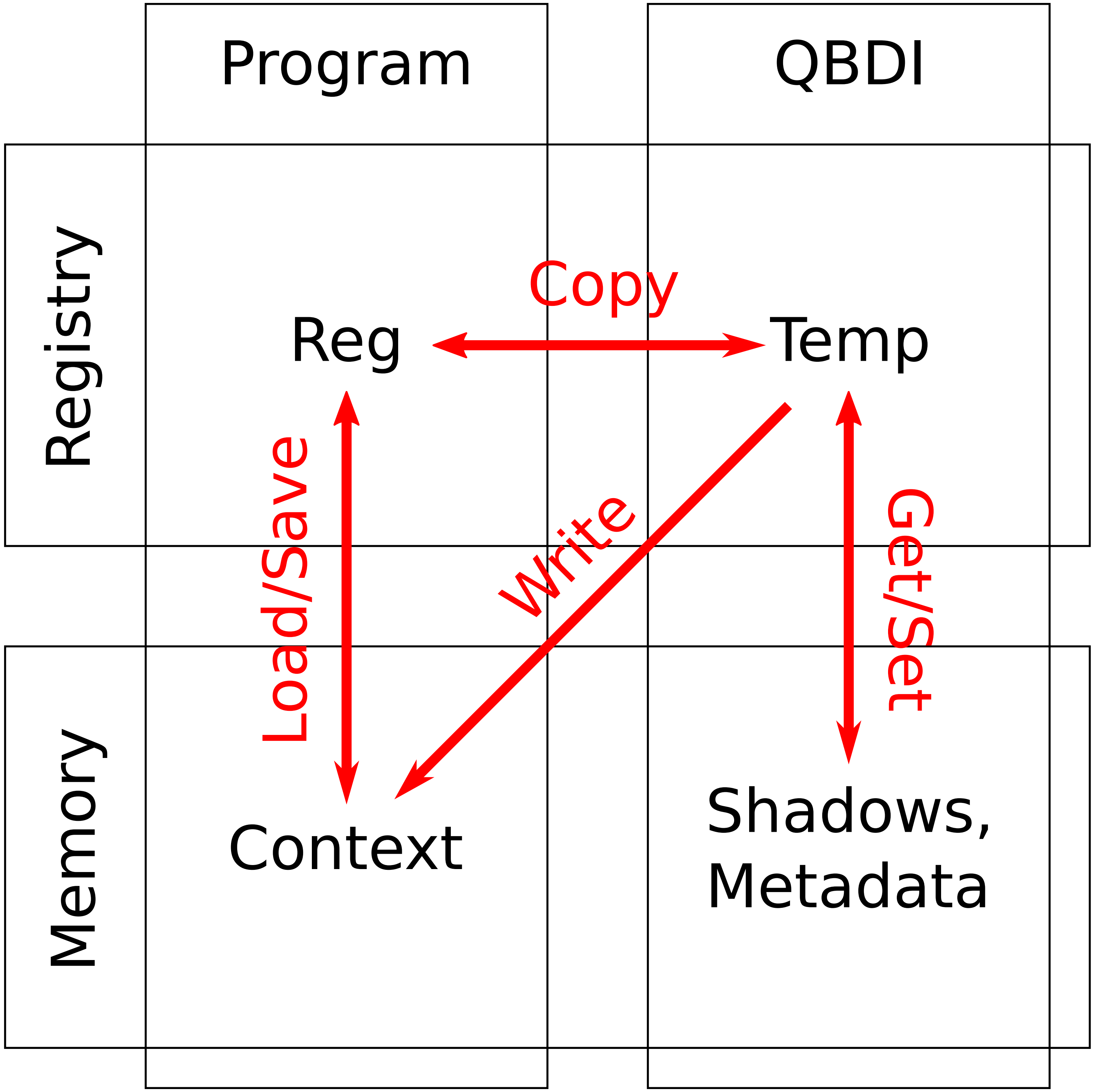
HTTP://PHAEDRIS.
DEVIANTART.COM

Patch DSL

Abstractions you said?

- Identify **transformation** steps required to patch instructions
- Regroup and integrate them as a **domain-specific language**
- Instructions are architecture specifics…
  - …DSL should be **generic** (as much as possible)

Patch DSL



Program
QBDI
Registry
Memory
Reg
Copy
Temp
Load/Save
Write
Get/Set
Context
Shadows,
Metadata

Patch DSL

```
mov [pc+0x2600], r1

mov r1, 0x410000

[...]

mov r1, [pc+0x2600]
```

Temp(0)

Patch DSL

```
mov [pc+0x2600], r1

mov r1, 0x410000

mov r0, [r0+r1]

mov r1, [pc+0x2600]
```

SubstituteWithTemp(Reg(REG_PC), Temp(0))

Patch DSL

- Modifications are defined in **rules**
- A rule is composed of
  - one (or several) **condition(s)**
  - one (or several) **action(s)**
- Actions can modify or replace an instruction

Patch DSL

```
/* Rule #3: Generic RIP patching.
 * Target:   Any instruction with RIP as operand, e.g. LEA RAX, [RIP + 1]
 * Patch:    Temp(0) := rip
 *             LEA RAX, [RIP + IMM] --> LEA RAX, [Temp(0) + IMM]
 */
    PatchRule(
        UseReg(Reg(REG_PC)),
        {
            GetPCOffset(Temp(0), Constant(0)),
            ModifyInstruction({
                SubstituteWithTemp(Reg(REG_PC), Temp(0))
            })
        }
    );
```

Patch DSL

```
/* Rule #0: Simulating BX instructions.
 * Target:  BX REG
 * Patch:   Temp(0) := Operand(0)
 *           DataOffset[Offset(PC)] := Temp(0)
 */
  PatchRule(
     Or({
        OpIs(llvm::ARM::BX),
        OpIs(llvm::ARM::BX_pred)
     }),
     {
        GetOperand(Temp(0), Operand(0)),
        WriteTemp(Temp(0), Offset(Reg(REG_PC)))
     }
  );
```

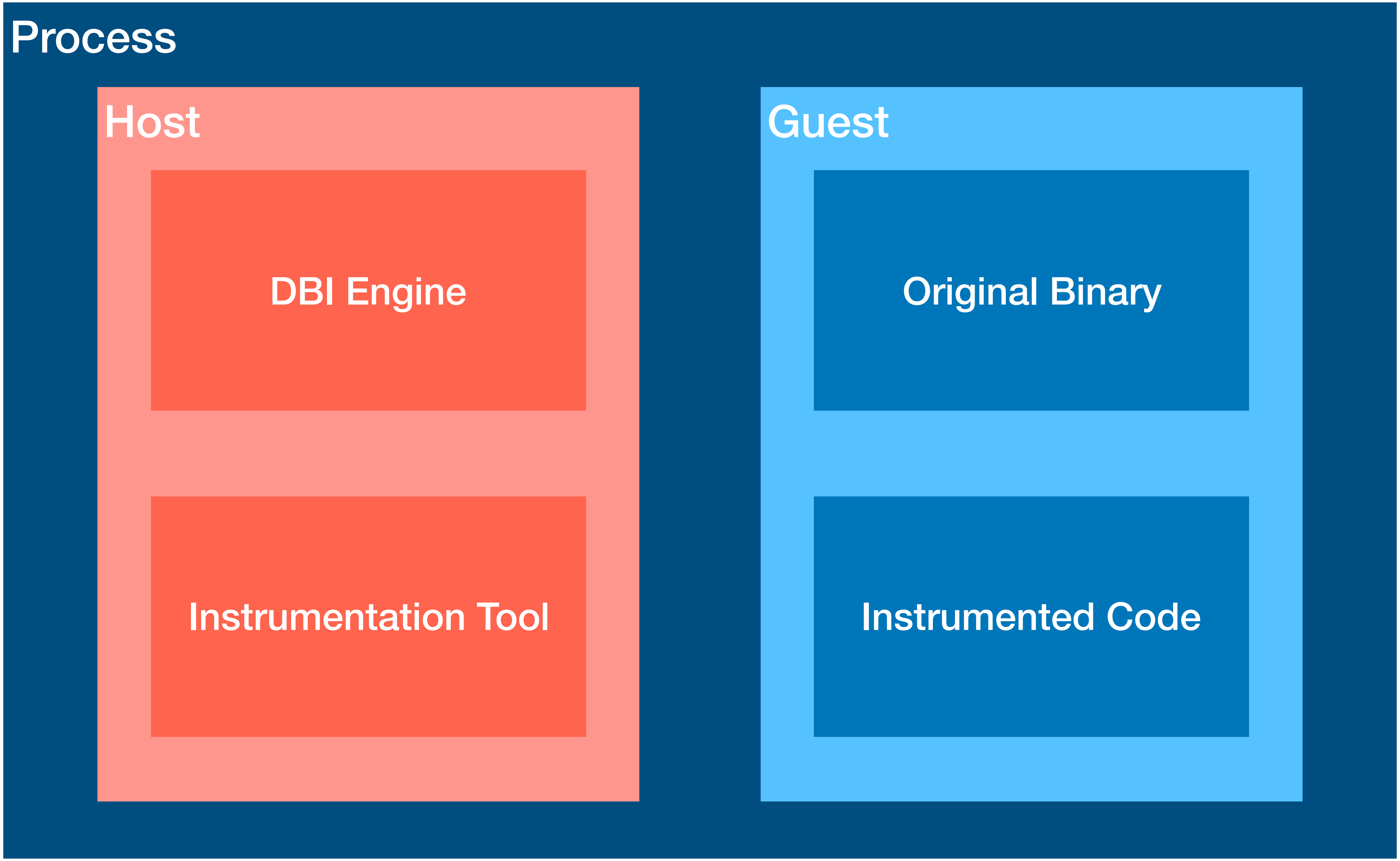
Lessons Learned

- LLVM provides **robust foundations** for modifying binary code
- **Abstractions** on top of it are:
  - **vital** to make quite a simple intermediate representation do complex things
  - very (very) hard to **conceptualise**

Designing a DBI:

2. Cross-Architecture Support

Host and Guest



Context Switch

- They share the **same memory** and the **same CPU context**
- We need to switch between those two contexts at **every cycle**
- No help from the kernel or the CPU

Context Switch

- **Save / restore** CPU context from guest / host
- **Avoid** any **side effects** on the guest
  - We can't modify its stack
  - We can't erase any register

➡ We need to **relatively address** host memory from the guest

Relative Addressing

- **Constrained** by CPU architecture capabilities
  - Limited to +/- 4096 under ARM

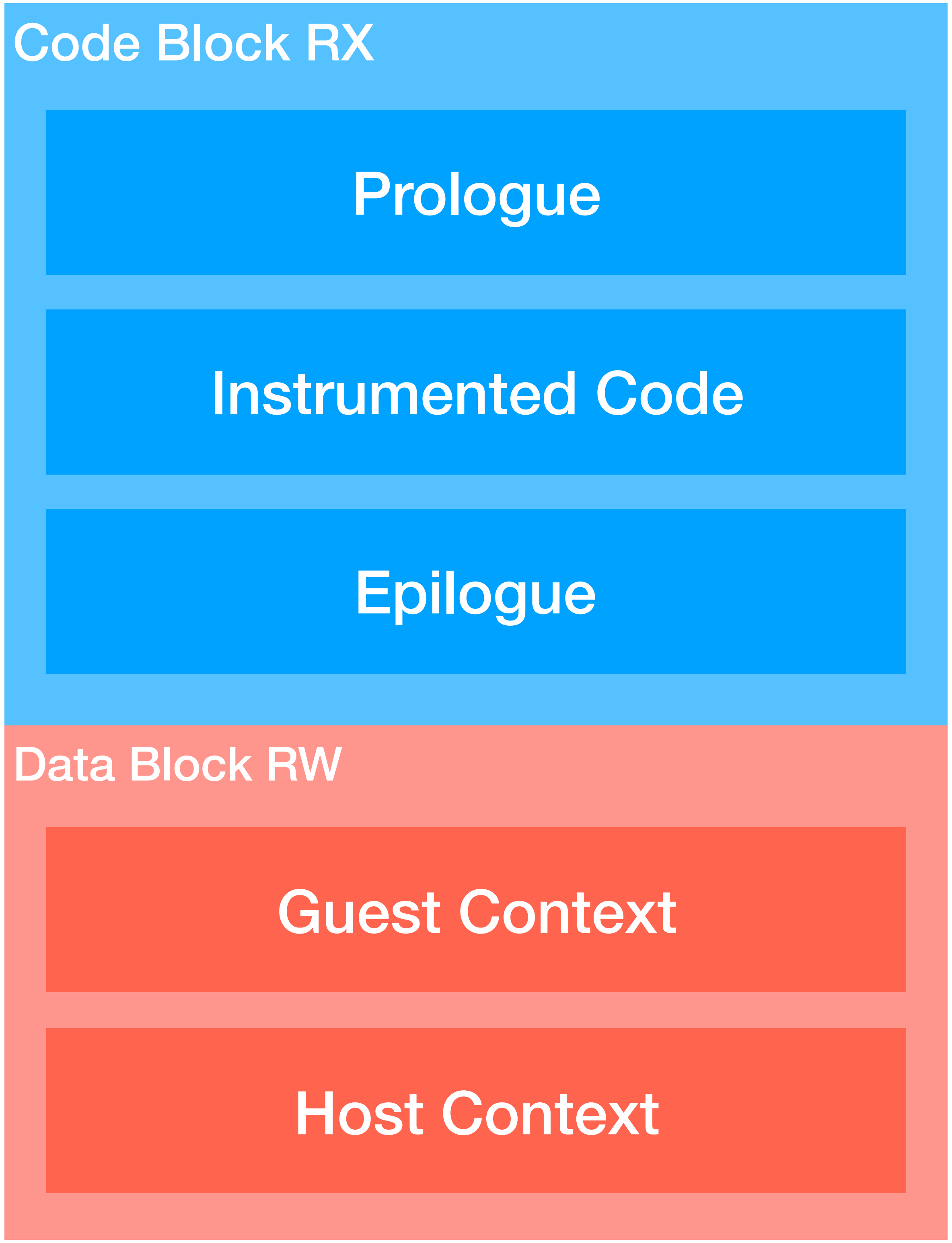
  ➡ We need **host memory** next to **guest code**

- We want to play nice with **D**ata **E**xecution **P**revention

➡ Allocate **2** contiguous memory pages:

  - Code block in **R**ead e**X**ecute
  - Data block in **R**ead **W**rite

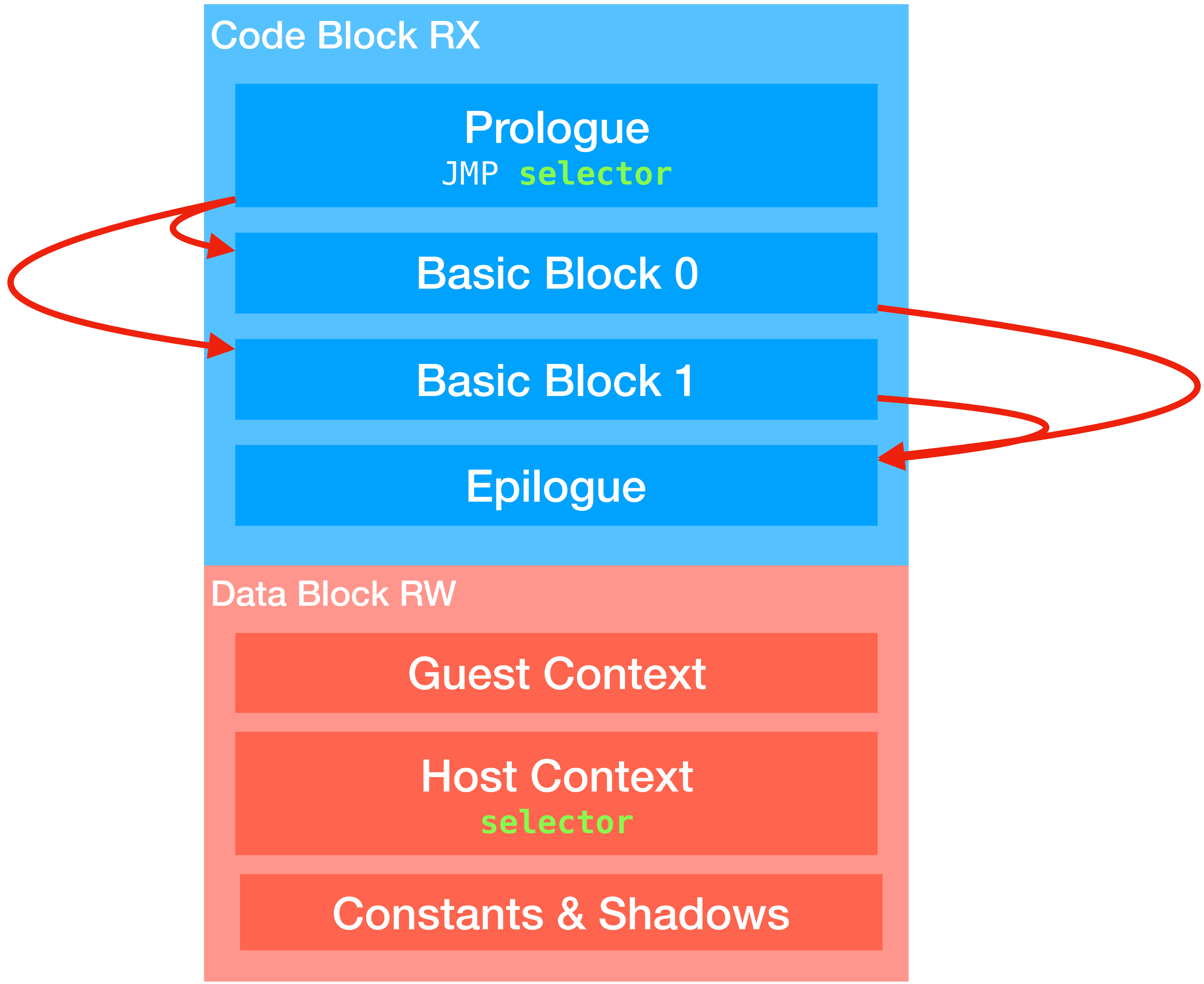
ExecBlock


Code Block RX
Prologue
Instrumented Code
Epilogue
Data Block RW
Guest Context
Host Context

ExecBlock

- Bind **instrumented code** and **instrumentation data**
- Data is guaranteed to be **directly addressable**
- 4 KB pages give us a lot of space…
  - We can put **multiple instrumented basic blocks** in the code block
  - We can put **more than just context** in the data block

Things Got More Complex ...



Making 4K Useful

- Instrumentation **constants**
  - used in the same way as ARM’s literal pool
- Instruction **shadows**
  - “instruction analog” to Valgrind's memory shadow
  - instrumentation variable abstraction
  - can be used to record memory accesses

What We Need

- A cross-platform **memory management abstraction**
  - allocating memory pages
  - changing page permissions
- A cross-architecture **assembler** working **in-memory**
  - It’s not just about building binary objects in-memory

Guess What?

LLVM JIT

- LLVM already has **several** JIT engine
  - They are very well designed…
  - …but **none** of them **fitted** our strict constraints
- LLVM provides everything to **create** a custom one
  - cross-architecture memory management abstraction
  - powerful **in-memory** assembler (LLVM MC)

Lessons learned

- LLVM is perfect for creating a JIT
- Designing a JIT engine for DBI is **hard**
  - Really easy to make a design **locked down** on a particular CPU architecture
- **Portability** need to be taken into account **from the start**

QBDI

Quarksla**B** **D**ynamic binary **I**nstrumentation is a modular, cross-platform and cross-architecture DBI framework

- Linux, macOS, Windows, Android and iOS
- User friendly
    - easy to use **C/C++** APIs
    - extensive **documentation**
    - binary **packages** for all major OS
- Modular design (Unix philosophy)

QBDI

- Modularity stands for:
  - core only provides what is **essential**
  - don’t force users to do thing in your way
  - easy **integration** everywhere
- Fun and flexible **Python** bindings
- Full featured integration with **Frida**

Roadmap

- Improve ARM architecture support
  - Thumb-2
  - Memory Access information
  - ARMv8 (AArch64)
- Add SIMD memory access
- Multithreading and exceptions
  - probably not as part of the core engine (KISS)

Demo time!

pyQBDI

```python
import pyqbdi;

def printInstruction(vm, gpr, fpr, data):
    inst = vm.getInstAnalysis()
    print "0x%x %s" % (inst.address, inst.disassembly)
    return pyqbdi.CONTINUE

def pyqbdipreload_on_run(vm, start, stop):
    state = vm.getGPRState()
    success, addr = pyqbdi.allocateVirtualStack(state, 0x100000)
    funcPtr = ctypes.cast(aLib.aFunction, ctypes.c_void_p).value
    vm.addInstrumentedModuleFromAddr(funcPtr)
    vm.addCodeCB(pyqbdi.PREINST, printInstruction, None)
    vm.call(funcPtr, [42])
```

Frida / QBDI

```
# frida --enable-jit -l /usr/local/share/qbdi/frida-qbdi.js ./demo.bin
     ____
    / _  |   Frida 10.6.26 - A world-class dynamic instrumentation framework
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/
Spawned `./demo.bin`. Use %resume to let the main thread start executing!
[Local::demo.bin]-> var vm = new QBDI()
undefined
[Local::demo.bin]-> var state = vm.getGPRState()
undefined
[Local::demo.bin]-> vm.addInstrumentedModule("demo.bin")
true
[Local::demo.bin]->
```

Give it a try

- https://qbdi.quarkslab.com/
- https://github.com/quarkslab/QBDI
    - Free software under permissive license (Apache 2)
- All suggestions / pull requests are most welcome
    - #qbdi on freenode

Many thanks to Paul and djo for their major contributions to this release!

Any questions?