QBDI Documentation

Release 0.7.1

Quarkslab

Jul 15, 2020

Contents

QuarkslaB Dynamic binary Instrumentation (QBDI) is a modular, cross-platform and cross-architecture DBI framework. It aims to support Linux, macOS, Android, iOS and Windows operating systems running on x86, x86-64, ARM and AArch64 architectures. Information about what is a DBI framework and how QBDI works can be found in the user documentation introduction (User Documentation).

QBDI modularity means it doesn't contain a preferred injection method and it is designed to be used in conjunction with an external injection tool. QBDI includes a tiny (`LD_PRELOAD` based) Linux and macOS injector for dynamic executables (QBDIPreload), which acts as the foundation for our Python bindings (pyQBDI). QBDI is also fully integrated with Frida, a reference dynamic instrumentation toolkit, allowing anybody to use their combined powers.

x86-64 support is mature (even if SIMD memory access are not yet reported). The support of x86 is new and some bug may occur. ARM architecture is a work in progress but already sufficient to execute simple CLI program like *ls* or *cat*. AArch64 is planned, but currently unsupported.

A current limitation is that QBDI doesn't handle signals, multithreading (it doesn't deal with new threads creation) and C++ exception mechanisms. However, those system-dependent features will probably not be part of the core library (KISS), and should be integrated as a new layer (to be determined how).

CHAPTER 1

Status

CPU	Operating Systems	Execution	Memory Access Information
x86-64	Android, Linux, macOS, Windows	Supported	Partial (only non SIMD)
x86	Android, Linux, macOS, Windows	Supported	Partial (only non SIMD)
ARM	Linux, Android, iOS	Planned (*)	Planned (*)
AArch64	Android	Planned (*)	Planned (*)

Warning: The ARM and AArch64 instruction sets are supported but they still need to be integrated along with x86 and x86-64.

stable dev

CHAPTER 2

Contents

2.1 User Documentation

2.1.1 Introduction

Why a DBI?

Debuggers are a popular approach to analyze the execution of a binary. While those tools are convenient, they are also quite slow. This performance problem is imperceptible to human users but really takes its toll on automated tools trying to single step through a complete program. Such automated tools are useful for tracking the evolution of the program states, extracting execution statistics and verifying that some runtime conditions hold true. Examples of usage include memory corruption debuggers, profilers, timeless debuggers and side-channel attack tools.

This performance cost is due to the kernel playing the role of a middleman between the debugger and the debuggee. The only way to get rid of the problem is to place the tool inside the binary being analyzed and this is what Dynamic Binary Instrumentation does: injecting instrumentation code inside the binary at runtime.

Why QBDI?

Existing DBI framework were designed more than 15 years ago, focusing on features and platforms that made sense at the time. Mobile platform support is often unstable or inexistent and instrumentation features are either simplistic or buried in low-level details.

QBDI attempts to retain the interesting features of those frameworks while avoiding their pitfalls and bringing new designs and ideas. Its goal is to be a cross-platform and multi-architectures modular DBI framework. The modular design exposes the DBI engine as a library that can start an instrumented execution anywhere, anytime and easily be incorporated in other tools.

QBDI : How does it work?

The core of DBI frameworks relies on the Just-In-Time (JIT) recompilation of the original program. This allows to interleave additional assembly code which can instrument any part of the execution. The DBI engine performing the JIT recompilation and the JITed code itself run in the same process but need to each have their own processor context. This requires to perform context switches between the two like a virtual machine would. We thus call the DBI context the **host** and the original program context the **guest**.

The **host** is composed of QBDI components and the **instrumentation tool**. The **instrumentation tool** is the code written by the user which interacts with the **QBDI VM** through a **C API** or a **C++ API**. The **instrumentation tool** can register **callbacks** to occur on specific events triggered either by QBDI or by the instrumentation code inserted inside the original program. The user documentation further details these APIs and how callbacks work.

Inside the VM resides the **QBDI Engine** which manages the instrumented execution. The engine runs the JIT loop which reads the **original program** code and generates the **instrumented code** which is then executed. Each loop iteration operates on a basic block, a sequence of instructions which ends with a branching instruction. This basic block is first patched, to accommodate the JIT process, and then instrumented as instructed by the instrumentation tool. This instrumented basic block is written in executable memory, executed and returns the address of the next basic block to execute. To avoid doing twice the same work, this **instrumented code** is actually written in a code cache.

Limitations

The **host** and the **guest** share the same process and thus the same resources. This means that they use the same heap and the same libraries and this will cause issues with any non-reentrant code. We could have chosen to shield users from those issues by forbidding instrumentation tools to use any external libraries like some other DBI frameworks have done. However we believe this is an overblown issue and that there are effective mechanisms to mitigate the problem. Nonetheless users need to be aware of this design limitation and the mitigations side-effects.

For example, tracing the heap memory allocator will cause deadlocks because it is not reentrant. There are other problematic cases but they are mostly limited to the standard C library and the OS loader. To avoid such issues we have an execution brokering system that allows to whitelist/blacklist specific pieces of code. These will be executed outside of the instrumentation process via a call hooking mechanism. This execution broker system is documented in *Execution Filtering* and *Execution Filtering*.

Moreover, the **host** relies on the loader loading and initializing its library dependencies which means the instrumentation process cannot be started before the loader has finished its job. As a result the loader itself cannot be instrumented.

2.1.2 Installation

QBDI can be installed using our prebuilt packages, docker images or from the source. For a more relaxing experience, we recommend using the stable release's prebuilt packages, following per-platform steps detailed on this page. The docker images is recommended for the x86 to have a full environment in 32bits. In order to install from source please refer to the developer documentation: *Compilation From Source*.

Using Pre-Built Packages

> **Warning:** The package x86 can be incompatible with the package x86_64.

Two different kinds of packages exist:

- systemwide installation packages

- archive packages

Systemwide packages install headers and binaries in system directories. Archives are created to be extracted and used in a local folder. They are provided for targets where systemwide install is not supported yet or doesn't make sense (like cross-compiled targets).

Debian / Ubuntu

Debian and Ubuntu packages are provided for stable and LTS releases, and can be installed using `dpkg`:

```
$ dpkg -i QBDI-*-*-X86_64.deb
```

Arch Linux

Archlinux packages can be installed using `pacman`:

```
$ pacman -U QBDI-*-*-X86_64.tar.xz
```

macOS

A software installer is provided for macOS. Opening the `.pkg` in Finder and following the instructions should install QBDI seamlessly.

Windows

A software installer is provided for Windows. Running the `.exe` and following the instructions should install QBDI seamlessly.

> **Warning:** Do not modify the default installation path. If you do so you will need to fix frida-qbdi.js with the new path.

Docker

The docker image is available in Docker Hub. The image is minimal and does not contain any compiler. You need to install the needed application or adapt the following dockerfile.

```
FROM qbdi/qbdi:x86_ubuntu

ENV USER="docker" \
    HOME="/home/docker"

# install some needed tools
RUN apt-get update && \
    apt-get upgrade -y && \
    apt-get install -y \
        build-essential \
        cmake \
        libstdc++-8-dev \
        zlib1g-dev \
```

```
        python \
        python-dev \
        #gdb \
        #vim \
        sudo \
        bash && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# create a user
RUN adduser --disabled-password --gecos '' $USER && \
    adduser $USER sudo && \
    echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers

# switch to new user
USER $USER
WORKDIR $HOME

# TODO : Add yours needed files
#ADD --chown=$USER . $HOME/

CMD ["/bin/bash"]
```

To run the container, we recommend allowing the usage of PTRACE that is necessary to use QBDIPreload.

```
$ docker run -it --rm --cap-add=SYS_PTRACE --security-opt seccomp:unconfined <image>␣
↪bash
```

Testing Your Installation

A QBDI template project is distributed along binaries and headers. On archive package targets, it is located in the `template` root-directory. On systemwide targets, a shell command is provided to populate a directory with this template:

```
$ mkdir test
$ cd test
$ qbdi-template
```

The template consists in a sample annotated source code with a basic CMake build script. A `README` is also provided with simple steps to compile the template. Successful compilation of the template and execution of the resulting binary should produce something similar:

```
$ ./qbdi_template
0x556a4a023e0a:     push    rbp
0x556a4a023e0b:     mov rbp, rsp
0x556a4a023e0e:     mov dword ptr [rbp - 4], edi
0x556a4a023e11:     mov eax, dword ptr [rbp - 4]
0x556a4a023e14:     xor eax, 92
0x556a4a023e17:     pop rbp
0x556a4a023e18:     ret
[*] retval=0x2c6
```

2.1.3 API

QBDI offers two APIs. The C++ API offers direct access to the VM interface and deeper integration with QBDI internals while the C API offers a simpler interface using a wrapper and can potentially be used in other languages as well (through a Foreign Function Interface).

API Basics

A step-by-step illustrating a basic (yet powerful) usage of QBDI C APIs.

Program initialization

First we need a `VMInstanceRef` variable and a pointer to the *fake* stack that will be used by the instrumented binary:

```
int main(int argc, char** argv) {
    VMInstanceRef vm = NULL;
    uint8_t *fakestack = NULL;
```

Virtual Machine initialization

We then need to initialize the Virtual Machine itself using `qbdi_initVM()`:

```
qbdi_initVM(&vm, NULL, NULL);
```

Pointer to VM context

In order to initialize the virtual stack properly (see next section), we need a pointer to the virtual machine state (`GPRState`), that can be obtained using `qbdi_getGPRState()`:

```
GPRState *state = qbdi_getGPRState(vm);
assert(state != NULL);
```

Allocate a virtual stack

Now that we have a pointer to the virtual state, we can allocate and initialize a virtual stack:

```
bool res = qbdi_allocateVirtualStack(state, STACK_SIZE, &fakestack);
assert(res == true);
```

Our first callback function

Now that the virtual machine has been set up, we can start playing with QBDI core features.

We want a callback function to be called every time we encounter an instruction, and that's exactly the purpose of `qbdi_addCodeRangeCB()`. Our callback will print current (guest) instruction address and disassembly. The range is here being defined from the start of the `qbdi_secretFunc()` until `qbdi_secretFunc()` + *100* (doesn't need to be accurate in that case):

```
VMAction showInstruction(VMInstanceRef vm, GPRState *gprState, FPRState *fprState,␣
↪void *data) {
    // Obtain an analysis of the instruction from the VM
    const InstAnalysis* instAnalysis = qbdi_getInstAnalysis(vm, QBDI_ANALYSIS_
↪INSTRUCTION | QBDI_ANALYSIS_DISASSEMBLY);
    // Printing disassembly
    printf("0x%" PRIRWORD ": %s\n", instAnalysis->address, instAnalysis->disassembly);
    return QBDI_CONTINUE;
}

uint32_t uid = qbdi_addCodeRangeCB(vm, (rword) &secretFunc, (rword) &secretFunc + 100,
↪ QBDI_PREINST, showInstruction, vm);
assert(uid != QBDI_INVALID_EVENTID);
```

Set instrumented range

Callback has been set to a specific range, we now need to enable the instrumentation on a specific code range (like a whole module). We can use `qbdi_addInstrumentedModuleFromAddr()`, using our main function address, to add our current executable code range:

```
res = qbdi_addInstrumentedModuleFromAddr(vm, (rword) &main);
assert(res == true);
```

Setup virtual stack

QBDI also provides some helpers, aiming to simplify common usage. We want to use our stack and state to *call* a function. In fact, this operation will be simulated, and we need to prepare the state to be *like* after a *call* instruction (with arguments in registers / stack). `qbdi_simulateCall()` will do this for us, respecting the ABI of current architecture. It will take a return address (that can be *fake*, acting like a terminator), and a list of arguments:

```
qbdi_simulateCall(state, FAKE_RET_ADDR, 1, (rword) 666);
```

Run the instrumentation

We can finally run the instrumentation using the `qbdi_run()` function. Here it will start the execution at `qbdi_secretFunc()`, until guest returns to our *fake* address.

Keep in mind that we have previously initialized the state to simulate a call, so QBDI will instrument the equivalent of a call to *secretFunc(666);*

```
res = qbdi_run(vm, (rword) secretFunc, (rword) FAKE_RET_ADDR);
assert(res == true);

// get return value from current state
rword retval = QBDI_GPR_GET(state, REG_RETURN);
printf("[*] retval=0x%"PRIRWORD"\n", retval);
```

End program properly

Finally, we need to free allocated stack and exit the virtual machine properly using `qbdi_alignedFree()` and `qbdi_terminateVM()`:

```
qbdi_alignedFree(fakestack);
qbdi_terminateVM(vm);
```

Fully working example

You can find a fully working example below, based on the precedent explanations.

```
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

#include <QBDI.h>

#define FAKE_RET_ADDR 42
#define STACK_SIZE 0x100000


QBDI_NOINLINE int secretFunc(unsigned int value) {
    return value ^ 0x5c;
}


VMAction showInstruction(VMInstanceRef vm, GPRState *gprState, FPRState *fprState,␣
↪void *data) {
    // Obtain an analysis of the instruction from the VM
    const InstAnalysis* instAnalysis = qbdi_getInstAnalysis(vm, QBDI_ANALYSIS_
↪INSTRUCTION | QBDI_ANALYSIS_DISASSEMBLY);
    // Printing disassembly
    printf("0x%" PRIRWORD ": %s\n", instAnalysis->address, instAnalysis->disassembly);
    return QBDI_CONTINUE;
}


int main(int argc, char** argv) {
    VMInstanceRef vm = NULL;
    uint8_t *fakestack = NULL;

    // init VM
    qbdi_initVM(&vm, NULL, NULL);

    // Get a pointer to the GPR state of the VM
    GPRState *state = qbdi_getGPRState(vm);
    assert(state != NULL);

    // Setup initial GPR state, this fakestack will produce a ret NULL at the end of␣
↪the execution
    bool res = qbdi_allocateVirtualStack(state, STACK_SIZE, &fakestack);
    assert(res == true);

    // Add callback on our instruction range
    uint32_t uid = qbdi_addCodeRangeCB(vm, (rword) &secretFunc, (rword) &secretFunc +␣
↪100, QBDI_PREINST, showInstruction, vm);
    assert(uid != QBDI_INVALID_EVENTID);

    // add executable code range
    res = qbdi_addInstrumentedModuleFromAddr(vm, (rword) &main);
```

```
    assert(res == true);

    // Simulate a call in stack
    qbdi_simulateCall(state, FAKE_RET_ADDR, 1, (rword) 666);

    // call secretFunc using VM, custom state and fake stack
    // eq: secretFunc(666);
    res = qbdi_run(vm, (rword) secretFunc, (rword) FAKE_RET_ADDR);
    assert(res == true);

    // get return value from current state
    rword retval = QBDI_GPR_GET(state, REG_RETURN);
    printf("[*] retval=0x%"PRIRWORD"\n", retval);

    // free everything
    qbdi_alignedFree(fakestack);
    qbdi_terminateVM(vm);

    return 0;
}
```

VM C

Initialization

The `qbdi_initVM()` function is used to create a new VM instance, using host CPU as a reference:

```
VMInstanceRef vm;
qbdi_initVM(&vm, NULL, NULL);
```

void **qbdi_initVM**(VMInstanceRef **instance*, **const** char **cpu*, **const** char ***mattrs*)

Create and initialize a VM instance.

Parameters

- `instance`: VM instance created.
- `cpu`: A C string naming the CPU model to use. If NULL, the default architecture CPU model is used (see LLVM documentation for more details).
- `mattrs`: A NULL terminated array of C strings specifying the attributes of the cpu model. If NULL, no additional features are specified.

The `qbdi_initVM()` function takes two optional parameters used to specify the CPU type or architecture and additional attributes of the CPU. Those are the same used by LLVM. Here's how you would initialize a VM for an ARM Cortex-A9 supporting the VFP2 floating point instructions:

```
VMInstanceRef vm;
const char* mattrs[] = {"vfp2"};
qbdi_initVM(&vm, "cortex-a9", mattrs);
```

State Management

The state of the guest processor can be queried and modified using two architectures dependent structures `GPRState` and `FPRState` which are detailed below for each architecture (*X86*, *X86_64* and *ARM*).

The `qbdi_getGPRState()` and `qbdi_getFPRState()` allow to query the current state of the guest processor while the `qbdi_setGPRState()` and `qbdi_setFPRState()` allow to modify it:

```
GPRState gprState; // local GPRState structure
gprState = *qbdi_getGPRState(vm); // Copy the VM structure into our local structure
gprState.rax = (rword) 42; // Set the value of the RAX register
qbdi_setGPRState(vm, &gprState); // Copy our local structure to the VM structure
```

As `qbdi_getGPRState()` and `qbdi_getFPRState()` return pointers to the VM context, it is possible to use them to modify the GPR and FPR states directly. The code below is equivalent to the example from above but avoid unnecessary copies:

```
qbdi_getGPRState(vm)->rax = (rword) 42;
```

GPRState *__qbdi_getGPRState__(VMInstanceRef *instance*)
Obtain the current general purpose register state.

Return A structure containing the General Purpose Registers state.

Parameters

- `instance`: VM instance.

FPRState *__qbdi_getFPRState__(VMInstanceRef *instance*)
Obtain the current floating point register state.

Return A structure containing the Floating Point Registers state.

Parameters

- `instance`: VM instance.

void __qbdi_setGPRState__(VMInstanceRef *instance*, GPRState **gprState*)
Set the GPR state

Parameters

- `instance`: VM instance.
- `gprState`: A structure containing the General Purpose Registers state.

void __qbdi_setFPRState__(VMInstanceRef *instance*, FPRState **fprState*)
Set the GPR state

Parameters

- `instance`: VM instance.
- `fprState`: A structure containing the Floating Point Registers state.

X86

The X86 `GPRState` structure is the following:

```
/*! X86 General Purpose Register context.
 */
typedef struct {
```

```
    rword eax;
    rword ebx;
    rword ecx;
    rword edx;
    rword esi;
    rword edi;
    rword ebp;
    rword esp;
    rword eip;
    rword eflags;
} GPRState;
//
```

The X86 `FPRState` structure is the following:

```
/*! X86 Floating Point Register context.
 */
typedef struct {
    union {
        FPControl     fcw;      /* x87 FPU control word */
        uint16_t      rfcw;
    };
    union {
        FPStatus      fsw;      /* x87 FPU status word */
        uint16_t      rfsw;
    };
    uint8_t           ftw;          /* x87 FPU tag word */
    uint8_t           rsrv1;        /* reserved */
    uint16_t          fop;          /* x87 FPU Opcode */
    uint32_t          ip;           /* x87 FPU Instruction Pointer offset */
    uint16_t          cs;           /* x87 FPU Instruction Pointer Selector */
    uint16_t          rsrv2;        /* reserved */
    uint32_t          dp;           /* x87 FPU Instruction Operand(Data) Pointer␣
↪offset */
    uint16_t          ds;           /* x87 FPU Instruction Operand(Data) Pointer␣
↪Selector */
    uint16_t          rsrv3;        /* reserved */
    uint32_t          mxcsr;        /* MXCSR Register state */
    uint32_t          mxcsrmask;    /* MXCSR mask */
    MMSTReg           stmm0;        /* ST0/MM0   */
    MMSTReg           stmm1;        /* ST1/MM1  */
    MMSTReg           stmm2;        /* ST2/MM2  */
    MMSTReg           stmm3;        /* ST3/MM3  */
    MMSTReg           stmm4;        /* ST4/MM4  */
    MMSTReg           stmm5;        /* ST5/MM5  */
    MMSTReg           stmm6;        /* ST6/MM6  */
    MMSTReg           stmm7;        /* ST7/MM7  */
    char              xmm0[16];     /* XMM 0  */
    char              xmm1[16];     /* XMM 1  */
    char              xmm2[16];     /* XMM 2  */
    char              xmm3[16];     /* XMM 3  */
    char              xmm4[16];     /* XMM 4  */
    char              xmm5[16];     /* XMM 5  */
    char              xmm6[16];     /* XMM 6  */
    char              xmm7[16];     /* XMM 7  */
    char              reserved[14*16];
    char              ymm0[16];     /* YMM0[255:128] */
```

```
    char                ymm1[16];      /* YMM1[255:128] */
    char                ymm2[16];      /* YMM2[255:128] */
    char                ymm3[16];      /* YMM3[255:128] */
    char                ymm4[16];      /* YMM4[255:128] */
    char                ymm5[16];      /* YMM5[255:128] */
    char                ymm6[16];      /* YMM6[255:128] */
    char                ymm7[16];      /* YMM7[255:128] */
} FPRState;
//
```

X86_64

The X86_64 `GPRState` structure is the following:

```
/*! X86_64 General Purpose Register context.
 */
typedef struct {
    rword rax;
    rword rbx;
    rword rcx;
    rword rdx;
    rword rsi;
    rword rdi;
    rword r8;
    rword r9;
    rword r10;
    rword r11;
    rword r12;
    rword r13;
    rword r14;
    rword r15;
    rword rbp;
    rword rsp;
    rword rip;
    rword eflags;

} GPRState;
//
```

The X86_64 `FPRState` structure is the following:

```
/*! X86_64 Floating Point Register context.
 */
typedef struct {
    union {
        FPControl      fcw;       /* x87 FPU control word */
        uint16_t       rfcw;
    };
    union {
        FPStatus       fsw;       /* x87 FPU status word */
        uint16_t       rfsw;
    };
    uint8_t            ftw;           /* x87 FPU tag word */
    uint8_t            rsrv1;         /* reserved */
```

```
    uint16_t            fop;            /* x87 FPU Opcode */
    uint32_t            ip;             /* x87 FPU Instruction Pointer offset */
    uint16_t            cs;             /* x87 FPU Instruction Pointer Selector */
    uint16_t            rsrv2;          /* reserved */
    uint32_t            dp;             /* x87 FPU Instruction Operand(Data) Pointer␣
↪offset */
    uint16_t            ds;             /* x87 FPU Instruction Operand(Data) Pointer␣
↪Selector */
    uint16_t            rsrv3;          /* reserved */
    uint32_t            mxcsr;          /* MXCSR Register state */
    uint32_t            mxcsrmask;      /* MXCSR mask */
    MMSTReg             stmm0;          /* ST0/MM0   */
    MMSTReg             stmm1;          /* ST1/MM1  */
    MMSTReg             stmm2;          /* ST2/MM2  */
    MMSTReg             stmm3;          /* ST3/MM3  */
    MMSTReg             stmm4;          /* ST4/MM4  */
    MMSTReg             stmm5;          /* ST5/MM5  */
    MMSTReg             stmm6;          /* ST6/MM6  */
    MMSTReg             stmm7;          /* ST7/MM7  */
    char                xmm0[16];       /* XMM 0  */
    char                xmm1[16];       /* XMM 1  */
    char                xmm2[16];       /* XMM 2  */
    char                xmm3[16];       /* XMM 3  */
    char                xmm4[16];       /* XMM 4  */
    char                xmm5[16];       /* XMM 5  */
    char                xmm6[16];       /* XMM 6  */
    char                xmm7[16];       /* XMM 7  */
    char                xmm8[16];       /* XMM 8  */
    char                xmm9[16];       /* XMM 9  */
    char                xmm10[16];      /* XMM 10  */
    char                xmm11[16];      /* XMM 11  */
    char                xmm12[16];      /* XMM 12  */
    char                xmm13[16];      /* XMM 13  */
    char                xmm14[16];      /* XMM 14  */
    char                xmm15[16];      /* XMM 15  */
    char                reserved[6*16];
    char                ymm0[16];       /* YMM0[255:128] */
    char                ymm1[16];       /* YMM1[255:128] */
    char                ymm2[16];       /* YMM2[255:128] */
    char                ymm3[16];       /* YMM3[255:128] */
    char                ymm4[16];       /* YMM4[255:128] */
    char                ymm5[16];       /* YMM5[255:128] */
    char                ymm6[16];       /* YMM6[255:128] */
    char                ymm7[16];       /* YMM7[255:128] */
    char                ymm8[16];       /* YMM8[255:128] */
    char                ymm9[16];       /* YMM9[255:128] */
    char                ymm10[16];      /* YMM10[255:128] */
    char                ymm11[16];      /* YMM11[255:128] */
    char                ymm12[16];      /* YMM12[255:128] */
    char                ymm13[16];      /* YMM13[255:128] */
    char                ymm14[16];      /* YMM14[255:128] */
    char                ymm15[16];      /* YMM15[255:128] */
} FPRState;
//
```

ARM

The ARM `GPRState` structure is the following:

```
/*! ARM General Purpose Register context.
 */
typedef struct {
    rword r0;
    rword r1;
    rword r2;
    rword r3;
    rword r4;
    rword r5;
    rword r6;
    rword r7;
    rword r8;
    rword r9;
    rword r10;
    rword r12;
    rword fp;
    rword sp;
    rword lr;
    rword pc;
    rword cpsr;

} GPRState;
//
```

The ARM `FPRState` structure is the following:

```
/*! ARM Floating Point Register context.
 */
typedef struct {
    float s[QBDI_NUM_FPR];
} FPRState;
//
```

State Initialization

Since the instrumented software and the VM need to run on different stacks, helper initialization functions exist for this purpose. The `qbdi_alignedAlloc()` function offers a cross-platform interface for aligned allocation which is required for allocating a stack. The `qbdi_allocateVirtualStack()` allows to allocate this stack and also setup the `GPRState` accordingly such that the required registers point to this stack. The `qbdi_simulateCall()` allows to simulate the call to a function by modifying the `GPRState` and stack to setup the return address and the arguments. The *Fibonacci* example is using those functions to call a function through the QBDI.

void *__qbdi_alignedAlloc__(size_t *size*, size_t *align*)

Allocate a block of memory of a specified sized with an aligned base address.

Return Pointer to the allocated memory or NULL in case an error was encountered.

Parameters

- `size`: Allocation size in bytes.

- `align`: Base address alignement in bytes.

bool **`qbdi_allocateVirtualStack`**(GPRState *_ctx_, uint32_t _stackSize_, uint8_t **_stack_)
Allocate a new stack and setup the GPRState accordingly. The allocated stack needs to be freed with alignedFree().

Return True if stack allocation was successfull.

Parameters

- `ctx`: GPRState which will be setup to use the new stack.
- `stackSize`: Size of the stack to be allocated.
- `stack`: The newly allocated stack pointer will be returned in the variable pointed by stack.

void **`qbdi_simulateCall`**(GPRState *_ctx_, rword _returnAddress_, uint32_t _argNum_, ...)
Simulate a call by modifying the stack and registers accordingly.

Parameters

- `ctx`: GPRState where the simulated call will be setup. The state needs to point to a valid stack for example setup with allocateVirtualStack().
- `returnAddress`: Return address of the call to simulate.
- `argNum`: The number of arguments in the variadic list.
- `...`: A variadic list of arguments.

Execution

Starting a Run

The run function is used to start the execution of a piece of code by the DBI. It first sets the Program Counter of the VM CPU state to the start parameter then runs the code through the DBI until a control flow instruction branches on the stop parameter. See the *Fibonacci* example for a basic usage of QBDI and the `qbdi_run()` function.

bool **`qbdi_run`**(VMInstanceRef _instance_, rword _start_, rword _stop_)
Start the execution by the DBI from a given address (and stop when another is reached).

Return True if at least one block has been executed.

Parameters

- `instance`: VM instance.
- `start`: Address of the first instruction to execute.
- `stop`: Stop the execution when this instruction is reached.

`qbdi_run()` is a low level function which implies that the `GPRState` has been properly initialized before.

But a very common usage of QBDI is to start instrumentation on a function call. `qbdi_call()` is a helper which behaves like a function call. It takes a function pointer, a list of arguments, and return the call's result. The only requirement is a valid (and properly aligned) stack pointer. In most cases, using `qbdi_allocateVirtualStack()` is enough, as a call rarely needs to work on the original thread stack.

bool **qbdi_call**(VMInstanceRef *instance*, rword **retval*, rword *function*, uint32_t *argNum*, ...)
Call a function using the DBI (and its current state).

Example:

```
// perform (with QBDI) a call similar to (*funcPtr)(42);
uint8_t *fakestack = NULL;
VMInstanceRef vm;
qbdi_initVM(&vm, NULL, NULL);
GPRState* gprState = qbdi_getGPRState(vm);
qbdi_allocateVirtualStack(gprState, 0x1000000, &fakestack);
qbdi_addInstrumentedModuleFromAddr(vm, funcPtr);
rword retVal;
qbdi_call(vm, &retVal, funcPtr, 1, 42);
```

Return True if at least one block has been executed.

Parameters

- `instance`: VM instance.
- `[retval]`: Pointer to the returned value (optional).
- `function`: Address of the function start instruction.
- `argNum`: The number of arguments in the variadic list.
- `...`: A variadic list of arguments.

Execution Filtering

For performance reasons but also integrity reasons it is rarely preferable to instrument the entirety of the guest code. The VM stores a set of instrumented ranges, when the execution gets out of those ranges the VM attempts to transfer the execution to a real execution via the `ExecBroker` which implements a return address hooking mechanism. As this hooking process is only a heuristic, it is neither guaranteed to be triggered as soon as the execution gets out of the set of instrumented ranges nor guaranteed to actually catch the return to the set of instrumented ranges.

Warning: By default the set of instrumented ranges is empty, a call to `qbdi_run()` will thus very quickly escape from the instrumentation process through an execution transfer made by the ExecBroker.

The basic functions to manipulate this set of instrumented ranges are `qbdi_addInstrumentedRange()` and `qbdi_removeInstrumentedRange()`.

void **qbdi_addInstrumentedRange**(VMInstanceRef *instance*, rword *start*, rword *end*)
Add an address range to the set of instrumented address ranges.

Parameters

- `instance`: VM instance.
- `start`: Start address of the range (included).
- `end`: End address of the range (excluded).

void **qbdi_removeInstrumentedRange**(VMInstanceRef *instance*, rword *start*, rword *end*)
Remove an address range from the set of instrumented address ranges.

Parameters

- `instance`: VM instance.
- `start`: Start address of the range (included).
- `end`: End address of the range (excluded).

As manipulating address ranges can be problematic, helpers API allow to use process memory maps information. The `qbdi_addInstrumentedModule()` and `qbdi_removeInstrumentedModule()` allow to use executable module names instead of address ranges. The currently loaded module names can be queried using `qbdi_getModuleNames()`. The `qbdi_addInstrumentedModuleFromAddr()` and `qbdi_removeInstrumentedModuleFromAddr()` functions allow to use any known address from a module to instrument it.

The `qbdi_instrumentAllExecutableMaps()` allows to instrument every memory map which is executable, the undesired address range can later be removed using the previous APIs in a blacklist approach. This approach is demonstrated in the *The Dude* example where sub-functions are removed in function of a trace level supplied as argument.

The `qbdi_removeAllInstrumentedRanges()` can be used to remove **all** ranges recorded through previous APIs (after this call, the VM will have no range left to instrument).

Below is an example of how to obtain, iterate then destroy the module names using the API.

```
#include <stdlib.h>
#include <stdio.h>

#include <QBDI.h>

int main(int argc, char** argv) {
    size_t size = 0, i = 0;
    char **modules = qbdi_getModuleNames(&size);

    for(i = 0; i < size; i++) {
        printf("%s\n", modules[i]);
    }

    for(i = 0; i < size; i++) {
        free(modules[i]);
    }
    free(modules);

    qbdi_MemoryMap *maps = qbdi_getCurrentProcessMaps(false, &size);
    for(i = 0; i < size; i++) {
        printf("%s (%c%c%c) ", maps[i].name,
                maps[i].permission & QBDI_PF_READ ? 'r' : '-',
                maps[i].permission & QBDI_PF_WRITE ? 'w' : '-',
                maps[i].permission & QBDI_PF_EXEC ? 'x' : '-');
        printf("(%#" PRIRWORD ", %#" PRIRWORD ")\n", maps[i].start, maps[i].end);
    }
    qbdi_freeMemoryMapArray(maps, size);

    return 0;
}
```

struct qbdi_MemoryMap
Map of a memory area (region).

Public Members

rword **start**
Range start value.

rword **end**
Range end value (always excluded).

qbdi_Permission **permission**
Region access rights (PF_READ, PF_WRITE, PF_EXEC).

char ***name**
Region name or path (useful when a region is mapping a module).

enum qbdi_Permission
Memory access rights.

Values:

QBDI_PF_NONE = 0
No access

QBDI_PF_READ = 1
Read access

QBDI_PF_WRITE = 2
Write access

QBDI_PF_EXEC = 4
Execution access

qbdi_MemoryMap ***qbdi_getCurrentProcessMaps**(bool *full_path*, size_t **size*)
Get a list of all the memory maps (regions) of the current process.

Return An array of MemoryMap object.

Parameters

- `full_path`: Return the full path of the module in name field
- `size`: Will be set to the number of strings in the returned array.

void **qbdi_freeMemoryMapArray**(*qbdi_MemoryMap* **arr*, size_t *size*)
Free an array of memory maps objects.

Parameters

- `arr`: An array of MemoryMap object.
- `size`: Number of elements in the array.

char ****qbdi_getModuleNames**(size_t **size*)
Get a list of all the module names loaded in the process memory. If no modules are found, size is set to 0 and this function returns NULL.

Return An array of C strings, each one containing the name of a loaded module. This array needs to be free'd by the caller by repetively calling free() on each of its element then finally on the array itself.

Parameters

- `size`: Will be set to the number of strings in the returned array.

bool **qbdi_addInstrumentedModule**(VMInstanceRef *instance*, **const** char **name*)
Add the executable address ranges of a module to the set of instrumented address ranges.

Return True if at least one range was added to the instrumented ranges.

Parameters

- `instance`: VM instance.
- `name`: The module's name.

bool **qbdi_removeInstrumentedModule**(VMInstanceRef *instance*, **const** char **name*)
Remove the executable address ranges of a module from the set of instrumented address ranges.

Return True if at least one range was removed from the instrumented ranges.

Parameters

- `instance`: VM instance.
- `name`: The module's name.

bool **qbdi_addInstrumentedModuleFromAddr**(VMInstanceRef *instance*, rword *addr*)
Add the executable address ranges of a module to the set of instrumented address ranges using an address belonging to the module.

Return True if at least one range was added to the instrumented ranges.

Parameters

- `instance`: VM instance.
- `addr`: An address contained by module's range.

bool **qbdi_removeInstrumentedModuleFromAddr**(VMInstanceRef *instance*, rword *addr*)
Remove the executable address ranges of a module from the set of instrumented address ranges using an address belonging to the module.

Return True if at least one range was removed from the instrumented ranges.

Parameters

- `instance`: VM instance.
- `addr`: An address contained by module's range.

bool **qbdi_instrumentAllExecutableMaps**(VMInstanceRef *instance*)
Adds all the executable memory maps to the instrumented range set.

Return True if at least one range was added to the instrumented ranges.

Parameters

- `instance`: VM instance.

Warning: Instrumenting the libc very often results in deadlock problems as it is also used by QBDI. It is thus recommended to always exclude it from the instrumentation.

void **qbdi_removeAllInstrumentedRanges**(VMInstanceRef *instance*)
Remove all instrumented ranges.

Parameters

- `instance`: VM instance.

Instrumentation

Instrumentation allows to run additional code alongside the original code of the software. There are three types of instrumentations implemented by the VM. The first one are instruction event callbacks where the execution of an instruction, under certain conditions, triggers a callback from the guest to the host. The second one are VM event callbacks where certain actions taken by the VM itself trigger a callback. The last one are custom instrumentations which allow to use the internal instrumentation mechanism of the VM.

Instruction Callback

The instruction callbacks follow the prototype of `InstCallback`. A user supplied data pointer parameter can be passed to the callback. Here's a simple callback which would increment an integer passed as a pointer through the data parameter:

```
VMAction increment(VMInstanceRef vm, GPRState *gprState, FPRState *fprState, void␣
↪*data) {
    int *counter = (int*) data;
    *counter += 1;
    return QBDI_CONTINUE;
}
```

The return value of this function is very important to determine how the VM should handle the resuming of execution. *QBDI_CONTINUE* will directly switch back to the guest code in the current ExecBlock without further processing by the VM. This means that modification of the Program Counter will not be taken into account and modifications of the program code will only be effective after the end of the current basic block (provided the code cache is cleared, which is not currently exported via the API). One can force those changes to be taken into account by breaking from the ExecBlock execution and returning early inside the VM using the *QBDI_BREAK_TO_VM*. *QBDI_STOP* causes the VM to stop the execution and the `qbdi_run()` to return.

The `qbdi_addCodeCB()` function allows to add a callback on every instruction (inside the instrumented range). If we register the previous callback to be called before every instruction, we effectively get an instruction counting instrumentation:

```
int counter = 0;
qbdi_addCodeCB(vm, QBDI_PREINST, increment, &counter);
qbdi_run(vm, ...); // Run the VM of some piece of code
printf("Instruction count: %d\n", counter); // counter contains the number of␣
↪instructions executed
```

enum InstPosition
Position relative to an instruction.

Values:

QBDI_PREINST = 0
Positioned before the instruction.

QBDI_POSTINST
Positioned after the instruction.

typedef *VMAction* (***InstCallback**)(VMInstanceRef vm, GPRState *gprState, FPRState *fprState, void *data)

Instruction callback function type.

Return The callback result used to signal subsequent actions the VM needs to take.

Parameters

- `vm`: VM instance of the callback.
- `gprState`: A structure containing the state of the General Purpose Registers. Modifying it affects the VM execution accordingly.
- `fprState`: A structure containing the state of the Floating Point Registers. Modifying it affects the VM execution accordingly.
- `data`: User defined data which can be defined when registering the callback.

enum VMAction

The callback results.

Values:

QBDI_CONTINUE = 0

The execution of the basic block continues.

QBDI_BREAK_TO_VM = 1

The execution breaks and returns to the VM causing a complete reevaluation of the execution state. A BREAK_TO_VM is needed to ensure that modifications of the Program Counter or the program code are taken into account.

QBDI_STOP = 2

Stops the execution of the program. This causes the run function to return early.

uint32_t **qbdi_addCodeCB**(VMInstanceRef *instance*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

Register a callback event for a specific instruction event.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `pos`: Relative position of the event callback (QBDI_PREINST / QBDI_POSTINST).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

It is also possible to register an `InstCallback` for a specific instruction address or address range with the `qbdi_addCodeAddrCB()` and `qbdi_addCodeRangeCB()` functions. These allow to fine-tune the instrumentation to specific codes or even portions of them.

uint32_t **qbdi_addCodeAddrCB**(VMInstanceRef *instance*, rword *address*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

Register a callback for when a specific address is executed.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `address`: Code address which will trigger the callback.

- `pos`: Relative position of the callback (QBDI_PREINST / QBDI_POSTINST).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

uint32_t **qbdi_addCodeRangeCB**(VMInstanceRef *instance*, rword *start*, rword *end*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

Register a callback for when a specific address range is executed.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `start`: Start of the address range which will trigger the callback.
- `end`: End of the address range which will trigger the callback.
- `pos`: Relative position of the callback (QBDI_PREINST / QBDI_POSTINST).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

uint32_t **qbdi_addMnemonicCB**(VMInstanceRef *instance*, **const** char **mnemonic*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

Register a callback event if the instruction matches the mnemonic.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `mnemonic`: Mnemonic to match.
- `pos`: Relative position of the event callback (QBDI_PREINST / QBDI_POSTINST).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

Note: Mnemonics can be instrumented using LLVM convention (You can register a callback on *ADD64rm* or *ADD64rr* for instance).

Note: You can also use "*" as a wildcard. (eg : *ADD*rr*)

If the execution of an instruction triggers more than one callback, those will be called in the order they were added to the VM.

Memory Callback

The memory callbacks (currently only supported under X86_64) allow to trigger callbacks on specific memory events. They use the same callback prototype, `InstCallback`, than instruction callback. They can be registered using the `qbdi_addMemAccessCB()` API. The API takes care of enabling the corresponding inline memory access logging

instrumentation using `qbdi_recordMemoryAccess()`. The memory accesses themselves are not directly provided as a callback parameter and need to be retrieved using the memory access APIs (see *Memory Access Analysis*). The *Cryptolock* example shows how to use those APIs to log the memory writes.

The type parameter allows to filter the callback on specific memory access type:

- *QBDI_MEMORY_READ* triggers the callback **before** every instruction performing a memory read.
- *QBDI_MEMORY_WRITE* triggers the callback **after** every instruction performing a memory write.
- *QBDI_MEMORY_READ_WRITE* triggers the callback **after** every instruction performing a memory read and/or write.

uint32_t **qbdi_addMemAccessCB**(VMInstanceRef *instance*, *MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)

Register a callback event for every memory access matching the type bitfield made by the instructions.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `type`: A mode bitfield: either QBDI_MEMORY_READ, QBDI_MEMORY_WRITE or both (QBDI_MEMORY_READ_WRITE).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

To enable more flexibility on the callback filtering `qbdi_addMemAddrCB()` and `qbdi_addMemRangeCB()` allow to filter for memory accesses targeting a specific memory address or range. However those callbacks are virtual callbacks: they cannot be directly triggered by the instrumentation because the access address needs to be checked dynamically but are instead triggered by a gate callback which takes care of the dynamic access address check and forwards or not the event to the virtual callback. This system thus has the same overhead as `qbdi_addMemAccessCB()` and is only meant as an API helper.

uint32_t **qbdi_addMemAddrCB**(VMInstanceRef *instance*, rword *address*, *MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)

Add a virtual callback which is triggered for any memory access at a specific address matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `address`: Code address which will trigger the callback.
- `type`: A mode bitfield: either QBDI_MEMORY_READ, QBDI_MEMORY_WRITE or both (QBDI_MEMORY_READ_WRITE).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

uint32_t **qbdi_addMemRangeCB**(VMInstanceRef *instance*, rword *start*, rword *end*, *MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)

Add a virtual callback which is triggered for any memory access in a specific address range matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `start`: Start of the address range which will trigger the callback.
- `end`: End of the address range which will trigger the callback.
- `type`: A mode bitfield: either QBDI_MEMORY_READ, QBDI_MEMORY_WRITE or both (QBDI_MEMORY_READ_WRITE).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

VM Events

VM events are triggered by the VM itself when it takes specific actions related to the execution of the instrumented program. The callback prototype `VMCallback` is different as it is triggered by the VM and not by an instruction. The callback receives a `VMState` structure. `VMCallback` can be registered for several events at the same time by combining several values of `VMEvent` in a mask using the `|` binary operator.

typedef *VMAction* (***VMCallback**)(VMInstanceRef vm, **const** *VMState* *vmState, GPRState *gprState, FPRState *fprState, void *data)

VM callback function type.

Return The callback result used to signal subsequent actions the VM needs to take.

Parameters

- `vm`: VM instance of the callback.
- `vmState`: A structure containing the current state of the VM.
- `gprState`: A structure containing the state of the General Purpose Registers. Modifying it affects the VM execution accordingly.
- `fprState`: A structure containing the state of the Floating Point Registers. Modifying it affects the VM execution accordingly.
- `data`: User defined data which can be defined when registering the callback.

uint32_t **qbdi_addVMEventCB**(VMInstanceRef *instance*, *VMEvent* *mask*, *VMCallback* *cbk*, void **data*)

Register a callback event for a specific VM event.

Return The id of the registered instrumentation (or QBDI_INVALID_EVENTID in case of failure).

Parameters

- `instance`: VM instance.
- `mask`: A mask of VM event type which will trigger the callback.
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

struct VMState

Structure describing the current VM state

Public Members

VMEvent **event**
The event(s) which triggered the callback (must be checked using a mask: event & BASIC_BLOCK_ENTRY).

rword **basicBlockStart**
The current basic block start address which can also be the execution transfer destination.

rword **basicBlockEnd**
The current basic block end address which can also be the execution transfer destination.

rword **sequenceStart**
The current sequence start address which can also be the execution transfer destination.

rword **sequenceEnd**
The current sequence end address which can also be the execution transfer destination.

rword **lastSignal**
Not implemented.

enum VMEvent
Values:

QBDI_SEQUENCE_ENTRY = 1
Triggered when the execution enters a sequence.

QBDI_SEQUENCE_EXIT = 1<<1
Triggered when the execution exits from the current sequence.

QBDI_BASIC_BLOCK_ENTRY = 1<<2
Triggered when the execution enters a basic block.

QBDI_BASIC_BLOCK_EXIT = 1<<3
Triggered when the execution exits from the current basic block.

QBDI_BASIC_BLOCK_NEW = 1<<4
Triggered when the execution enters a new (~unknown) basic block.

QBDI_EXEC_TRANSFER_CALL = 1<<5
Triggered when the ExecBroker executes an execution transfer.

QBDI_EXEC_TRANSFER_RETURN = 1<<6
Triggered when the ExecBroker returns from an execution transfer.

QBDI_SYSCALL_ENTRY = 1<<7
Not implemented.

QBDI_SYSCALL_EXIT = 1<<8
Not implemented.

QBDI_SIGNAL = 1<<9
Not implemented.

Custom Instrumentation

Custom instrumentation is not available via the C API as it requires manipulating internal C++ classes.

Removing Instrumentations

The id returned by the previous instrumentation functions can be used to modify the instrumentation afterward. The `qbdi_deleteInstrumentation()` function allows to remove an instrumentation by id while the `qbdi_deleteAllInstrumentations()` function will remove all instrumentations. This code would do two different executions with different callbacks:

```
uint32_t cb1 = qbdi_addCodeCB(vm, QBDI_PREINST, Callback1, &some_data);
uint32_t cb2 = qbdi_addVMEventCB(vm, QBDI_EXEC_TRANSFER_CALL | QBDI_BASIC_BLOCK_ENTRY,
↪ Callback2, &some_data);
qbdi_run(vm, ...); // Run the VM of some piece of code with all instrumentations
qbdi_deleteInstrumentation(vm, cb1);
qbdi_run(vm, ...); // Run the VM of some piece of code with some instrumentation
qbdi_deleteAllInstrumentations(vm);
qbdi_run(vm, ...); // Run the VM of some piece of code without instrumentation
```

bool **qbdi_deleteInstrumentation**(VMInstanceRef *instance*, uint32_t *id*)

Remove an instrumentation.

Return True if instrumentation has been removed.

Parameters

- `instance`: VM instance.
- `id`: The id of the instrumentation to remove.

void **qbdi_deleteAllInstrumentations**(VMInstanceRef *instance*)

Remove all the registered instrumentations.

Parameters

- `instance`: VM instance.

Instruction Analysis

Callback based instrumentation has only limited utility if the current context of execution is not available. To obtain more information about an instruction, the VM can parse its internal structures for us and provide analysis results in a `InstAnalysis` structure:

```
CallbackResult increment(VMInstanceRef vm, GPRState *gprState, FPRState *fprState,␣
↪void *data) {
    const InstAnalysis *instAnalysis = qbdi_getInstAnalysis(vm, QBDI_ANALYSIS_
↪INSTRUCTION | QBDI_ANALYSIS_DISASSEMBLY);

    printf("%s\n", instAnalysis->disassembly);

    return QBDI_CONTINUE;
}
```

const *InstAnalysis* ***qbdi_getInstAnalysis**(VMInstanceRef *instance*, *AnalysisType* *type*)

Obtain the analysis of an instruction metadata. Analysis results are cached in the VM. The validity of the returned pointer is only guaranteed until the end of the callback, else a deepcopy of the structure is required.

Return A *InstAnalysis* structure containing the analysis result.

Parameters

- `instance`: VM instance.
- `type`: Properties to retrieve during analysis.

Analysis can gather all kinds of properties linked with the instruction. It's possible to select which one will be retrieved using *type* argument.

enum AnalysisType
Instruction analysis type

Values:

QBDI_ANALYSIS_INSTRUCTION = 1
Instruction analysis (address, mnemonic, . . .)

QBDI_ANALYSIS_DISASSEMBLY = 1<<1
Instruction disassembly

QBDI_ANALYSIS_OPERANDS = 1<<2
Instruction operands analysis

QBDI_ANALYSIS_SYMBOL = 1<<3
Instruction symbol

Every analysis has a performance cost, which can be reduced by selecting types carefully, and is amortized using a cache inside the VM.

struct InstAnalysis
Structure containing analysis results of an instruction provided by the VM.

Public Members

const char ***mnemonic**
LLVM mnemonic (warning: NULL if !ANALYSIS_INSTRUCTION)

rword **address**
Instruction address

uint32_t **instSize**
Instruction size (in bytes)

bool **affectControlFlow**
true if instruction affects control flow

bool **isBranch**
true if instruction acts like a 'jump'

bool **isCall**
true if instruction acts like a 'call'

bool **isReturn**
true if instruction acts like a 'return'

bool **isCompare**
true if instruction is a comparison

bool **isPredicable**
true if instruction contains a predicate (~is conditional)

bool **mayLoad**
true if instruction 'may' load data from memory

bool **mayStore**
: true if instruction 'may' store data to memory

char ***disassembly**
: Instruction disassembly (warning: NULL if !ANALYSIS_DISASSEMBLY)

uint8_t **numOperands**
: Number of operands used by the instruction

OperandAnalysis ***operands**
: Structure containing analysis results of an operand provided by the VM. (warning: NULL if !ANALYSIS_OPERANDS)

const char ***symbol**
: Instruction symbol (warning: NULL if !ANALYSIS_SYMBOL or not found)

uint32_t **symbolOffset**
: Instruction symbol offset

const char ***module**
: Instruction module name (warning: NULL if !ANALYSIS_SYMBOL or not found)

uint32_t **analysisType**
: INTERNAL: Instruction analysis type (this should NOT be used)

If provided `AnalysisType` is equal to *QBDI_ANALYSIS_OPERANDS*, then `qbdi_getInstAnalysis()` will also analyze all instruction operands, and fill an array of `OperandAnalysis` (of length *numOperands*).

struct OperandAnalysis
: Structure containing analysis results of an operand provided by the VM.

Public Members

OperandType **type**
: Operand type

OperandFlag **flag**
: Operand flag

rword **value**
: Operand value (if immediate), or register Id

uint8_t **size**
: Operand size (in bytes)

uint8_t **regOff**
: Sub-register offset in register (in bits)

uint16_t **regCtxIdx**
: Register index in VM state

const char ***regName**
: Register name

RegisterAccessType **regAccess**
: Register access type (r, w, rw)

enum OperandType
: Operand type

Values:

QBDI_OPERAND_INVALID = 0
Invalid operand

QBDI_OPERAND_IMM
Immediate operand

QBDI_OPERAND_GPR
Register operand

QBDI_OPERAND_PRED
Predicate operand

enum OperandFlag
Values:

QBDI_OPERANDFLAG_NONE = 0
No flag

QBDI_OPERANDFLAG_ADDR = 1<<0
The operand is used to compute an address

QBDI_OPERANDFLAG_PCREL = 1<<1
The value of the operand is PC relative

QBDI_OPERANDFLAG_UNDEFINED_EFFECT = 1<<2
The operand role isn't fully defined

enum RegisterAccessType
Access type (R/W/RW) of a register operand

Values:

QBDI_REGISTER_UNUSED = 0
Unused register

QBDI_REGISTER_READ = 1
Register read access

QBDI_REGISTER_WRITE = 1<<1
Register write access

QBDI_REGISTER_READ_WRITE = 3
Register read/write access

Memory Access Analysis

QBDI VM can log memory access using inline instrumentation. This adds instrumentation rules which store the accessed addresses and values into specific memory variables called *instruction shadows*. To enable this memory logging, the `qbdi_recordMemoryAccess()` API can be used. This memory logging is also automatically enabled when calling one of the memory callbacks.

bool **qbdi_recordMemoryAccess**(VMInstanceRef *instance*, *MemoryAccessType type*)
Add instrumentation rules to log memory access using inline instrumentation and instruction shadows.

Return True if inline memory logging is supported, False if not or in case of error.

Parameters

- `instance`: VM instance.
- `type`: Memory mode bitfield to activate the logging for: either QBDI_MEMORY_READ, QBDI_MEMORY_WRITE or both (QBDI_MEMORY_READ_WRITE).

The memory access type always refers to either *QBDI_MEMORY_READ*, *QBDI_MEMORY_WRITE*, *QBDI_MEMORY_READ_WRITE* (which is a bitfield combination of the two previous ones).

Once the logging has been enabled, `qbdi_getInstMemoryAccess()` and `qbdi_getBBMemoryAccess()` can be used to retrieve the memory accesses made by the last instruction or by the last basic block. These two APIs return `qbdi_MemoryAccess()` structures. Write memory accesses are only returned if the instruction has already been executed (i.e. in the case of a *QBDI_POSTINST*). The *Cryptolock* example shows how to use those APIs to log the memory writes.

struct MemoryAccess
Describe a memory access

Public Members

rword **instAddress**
Address of instruction making the access

rword **accessAddress**
Address of accessed memory

rword **value**
Value read from / written to memory

uint8_t **size**
Size of memory access (in bytes)

MemoryAccessType **type**
Memory access type (READ / WRITE)

enum MemoryAccessType
Memory access type (read / write / . . .)

Values:

QBDI_MEMORY_READ = 1
Memory read access

QBDI_MEMORY_WRITE = 1<<1
Memory write access

QBDI_MEMORY_READ_WRITE = 3
Memory read/write access

MemoryAccess ***qbdi_getInstMemoryAccess**(VMInstanceRef *instance*, size_t **size*)
Obtain the memory accesses made by the last executed instruction. Return NULL and a size of 0 if the instruction made no memory access.

Return An array of memory accesses made by the instruction.

Parameters

- `instance`: VM instance.
- `size`: Will be set to the number of elements in the returned array.

MemoryAccess ***qbdi_getBBMemoryAccess**(VMInstanceRef *instance*, size_t **size*)
Obtain the memory accesses made by the last executed basic block. Return NULL and a size of 0 if the basic block made no memory access.

Return An array of memory accesses made by the basic block.

Parameters

- `instance`: VM instance.
- `size`: Will be set to the number of elements in the returned array.

Free resources

Some resources returned by a VM instance must be manually freed using specialized functions. A VM instance itself must be destroyed using `qbdi_terminateVM()`.

void **qbdi_terminateVM**(VMInstanceRef *instance*)
Destroy an instance of VM.

Parameters

- `instance`: VM instance.

void **qbdi_alignedFree**(void **ptr*)
Free a block of aligned memory allocated with alignedAlloc.

Parameters

- `ptr`: Pointer to the allocated memory.

Cache management

QBDI provides a cache system for basic blocks that you might want to deal directly with it. There are a few functions that can help you with that.

bool **qbdi_precacheBasicBlock**(VMInstanceRef *instance*, rword *pc*)
Pre-cache a known basic block

Return True if basic block has been inserted in cache.

Parameters

- `instance`: VM instance.
- `pc`: Start address of a basic block

void **qbdi_clearCache**(VMInstanceRef *instance*, rword *start*, rword *end*)
Clear a specific address range from the translation cache.

Parameters

- `instance`: VM instance.
- `start`: Start of the address range to clear from the cache.
- `end`: End of the address range to clear from the cache.

void **qbdi_clearAllCache**(VMInstanceRef *instance*)
Clear the entire translation cache.

Parameters

- `instance`: VM instance.

Examples

Fibonacci

This example instruments its own code to count the executed instructions and displays the disassembly of every instruction executed.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>

#include "QBDI.h"

#define FAKE_RET_ADDR 42


int fibonacci(int n) {
    if(n <= 2)
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}

VMAction countRecursion(VMInstanceRef vm, GPRState *gprState, FPRState *fprState,␣
↪void *data) {
    *((unsigned*) data) += 1;
    return QBDI_CONTINUE;
}

VMAction countInstruction(VMInstanceRef vm, GPRState *gprState, FPRState *fprState,␣
↪void *data) {
    // Cast data to our counter
    uint32_t* counter = (uint32_t *) data;
    // Obtain an analysis of the instruction from the VM
    const InstAnalysis* instAnalysis = qbdi_getInstAnalysis(vm, QBDI_ANALYSIS_
↪INSTRUCTION | QBDI_ANALYSIS_DISASSEMBLY);
    // Printing disassembly
    printf("%" PRIRWORD ": %s\n", instAnalysis->address, instAnalysis->disassembly);
    // Incrementing the instruction counter
    (*counter)++;
    // Signaling the VM to continue execution
    return QBDI_CONTINUE;
}

static const size_t STACK_SIZE = 0x100000; // 1MB

int main(int argc, char **argv) {
    int n = 0;
    unsigned recursions = 0;
    uint32_t counter = 0;
    uint8_t *fakestack;
    VMInstanceRef vm;
    GPRState *state;

    printf("Initializing VM ...\n");
    // Constructing a new QBDI VM
```

```
    qbdi_initVM(&vm, NULL, NULL);
    // Registering countInstruction() callback to be called after every instruction
    qbdi_addCodeCB(vm, QBDI_POSTINST, countInstruction, &counter);
    qbdi_addCodeAddrCB(vm, (rword) &fibonacci, QBDI_PREINST, countRecursion, &
↪recursions);

    // Get a pointer to the GPR state of the VM
    state = qbdi_getGPRState(vm);
    // Setup initial GPR state, this fakestack will produce a ret NULL at the end of␣
↪the execution
    qbdi_allocateVirtualStack(state, STACK_SIZE, &fakestack);
    // Argument to the fibonnaci call
    if(argc >= 2) {
        n = atoi(argv[1]);
    }
    if(n < 1) {
        n = 1;
    }
    qbdi_simulateCall(state, FAKE_RET_ADDR, 1, (rword) n);

    printf("Running fibonacci(%d) ...\n", n);
    // Running DBI execution
    qbdi_instrumentAllExecutableMaps(vm);
    qbdi_run(vm, (rword) fibonacci, (rword) FAKE_RET_ADDR);
    printf("fibonnaci ran in %u instructions, recursed %u times and returned %d\n",
            counter, recursions - 1, (int) QBDI_GPR_GET(state, REG_RETURN));
    qbdi_terminateVM(vm);
    qbdi_alignedFree(fakestack);

    return 0;
}
```

The Dude

This example instruments its own code to count the executed instructions and displays the disassembly of every instruction executed. A supplied trace level argument allows to disable the tracing in specific sub-functions using the APIs presented in *Execution Filtering*.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <time.h>
#include <inttypes.h>

#include "QBDI.h"


void stripLF(char *s) {
    char* pos = strrchr(s, '\n');
    if(pos != NULL) {
        *pos = '\0';
    }
}

QBDI_NOINLINE uint64_t magicPow(uint64_t n, uint64_t e) {
```

```
    uint64_t r = 1;
    uint64_t i = 0;

    for(i = 0; i < e; i++) {
        r = (r*n);
    }
    return r;
}

QBDI_NOINLINE uint64_t magicHash(char* secret) {
    uint64_t hash = 0;
    uint64_t acc = 1;
    int len = strlen(secret);
    int i = 0;

    for(i = 0; i < len; i++) {
        uint64_t magic = magicPow(secret[i], acc);
        hash ^= magic;
        acc = (acc + magic) % 256;
    }

    return hash;
}

QBDI_NOINLINE int thedude() {
    static int BUFFER_SIZE = 64;
    time_t t = 0;
    char *name = (char*) malloc(BUFFER_SIZE);
    char *secret = (char*) malloc(2*BUFFER_SIZE);
    uint64_t hash = 0;

    printf("Hi I'm the dude.\n");
    printf("Give me your name and I'll give you a hash.\n");
    printf("So what's your name ? ");
    if (!fgets(name, BUFFER_SIZE, stdin)) {
        return 1;
    }
    stripLF(name);
    time(&t);
    sprintf(secret, "%" PRIu64 ":%s", (uint64_t) t, name);
    printf("Ok I'll give you the hash of %s.\n", secret);
    hash = magicHash(secret);
    printf("Your hash is %" PRIx64 ".\n", hash);
    printf("No need to thank me.\n");

    free(name);
    free(secret);

    return 0;
}

VMAction count(VMInstanceRef vm, GPRState *gprState, FPRState *fprState, void *data) {
    // Cast data to our CallbackInfo struct
    uint32_t* counter = (uint32_t *) data;
    // Obtain an analysis of the instruction from the VM
    const InstAnalysis* instAnalysis = qbdi_getInstAnalysis(vm, QBDI_ANALYSIS_
↪INSTRUCTION | QBDI_ANALYSIS_DISASSEMBLY);
    // Printing disassembly
```

```
    printf("%" PRIRWORD ": %s\n", instAnalysis->address, instAnalysis->disassembly);
    // Incrementing the instruction counter
    (*counter)++;
    // Signaling the VM to continue execution
    return QBDI_CONTINUE;
}

static const size_t STACK_SIZE = 0x100000; // 1MB
static const rword FAKE_RET_ADDR = 0x40000;

int main(int argc, char **argv) {
    uint8_t *fakestack;
    VMInstanceRef vm;
    GPRState *state;
    int traceLevel = 0;
    uint32_t counter = 0;
    bool instrumented;

    if(argc > 1) {
        traceLevel = atoi(argv[1]);
        if(traceLevel > 2) {
            traceLevel = 2;
        }
    }

    printf("Initializing VM ...\n");
    // Constructing a new QBDI VM
    qbdi_initVM(&vm, NULL, NULL);
    // Registering count() callback to be called after every instruction
    qbdi_addCodeCB(vm, QBDI_POSTINST, count, &counter);

    // Get a pointer to the GPR state of the VM
    state = qbdi_getGPRState(vm);
    // Setup initial GPR state, this fakestack will produce a ret NULL at the end of␣
↪the execution
    qbdi_allocateVirtualStack(state, STACK_SIZE, &fakestack);
    qbdi_simulateCall(state, FAKE_RET_ADDR, 0);


    printf("Running thedude() with trace level %d...\n", traceLevel);
    // Running DBI execution
    instrumented = qbdi_addInstrumentedModuleFromAddr(vm, (rword) &main);
    if (instrumented) {
        if(traceLevel < 1) {
            qbdi_removeInstrumentedRange(vm, (rword) magicHash, (rword) magicHash +␣
↪32);
        }
        if(traceLevel < 2) {
            qbdi_removeInstrumentedRange(vm, (rword) magicPow, (rword) magicPow + 32);
        }
        qbdi_run(vm, (rword) thedude, (rword) FAKE_RET_ADDR);
        printf("thedude ran in %u instructions\n", counter);
    } else {
        printf("failed to find main module...\n");
    }
    qbdi_terminateVM(vm);
    qbdi_alignedFree(fakestack);
```

```
    return 0;
}
```

Cryptolock

This example instruments its own code to display instruction performing memory writes and the written values. This can be used to discover the password of the cryptolock and reveal the message (hint: if the password is correct then the hash buffer is all zero).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>

#include "QBDI.h"


QBDI_NOINLINE void hashPassword(char* hash, const char* password) {
    char acc = 42;
    size_t i = 0;
    size_t hash_len = strlen(hash);
    size_t password_len = strlen(password);

    for(i = 0; i < hash_len && i < password_len; i++) {
        hash[i] = (hash[i] ^ acc) - password[i];
        acc = password[i];
    }

}

char SECRET[] =
↪"\x29\x0d\x20\x00\x00\x00\x00\x0a\x65\x1f\x32\x00\x19\x0c\x4e\x1b\x2d\x09\x66\x0c\x1a\x06\x05\x06\x
↪";

QBDI_NOINLINE const char* getSecret(const char* password) {
    size_t i = 0;
    size_t password_len = strlen(password);
    for(i = 0; i < sizeof(SECRET); i++) {
        SECRET[i] ^= password[i%password_len];
    }
    return SECRET;
}

QBDI_NOINLINE const char* cryptolock(const char* password) {
    char hash[] = "\x6f\x29\x2a\x29\x1a\x1c\x07\x01";
    size_t i = 0;

    hashPassword(hash, password);

    bool good = true;
    for(i = 0; i < sizeof(hash); i++) {
        if(hash[i] != 0) {
            good = false;
        }
    }
```

```
    if(good) {
        return getSecret(password);
    }
    else {
        return NULL;
    }
}

VMAction onwrite(VMInstanceRef vm, GPRState *gprState, FPRState *fprState, void␣
↪*data) {
    size_t i = 0;
    size_t num_access = 0;
    // Obtain an analysis of the instruction from the vm
    const InstAnalysis* instAnalysis = qbdi_getInstAnalysis(vm, QBDI_ANALYSIS_
↪INSTRUCTION | QBDI_ANALYSIS_DISASSEMBLY);
    // Obtain the instruction memory accesses
    MemoryAccess* memAccesses = qbdi_getInstMemoryAccess(vm, &num_access);

    // Printing disassembly
    printf("%" PRIRWORD ": %s\n", instAnalysis->address, instAnalysis->disassembly);
    // Printing write memory accesses
    for(i = 0; i < num_access; i++) {
        // Checking the access mode
        if(memAccesses[i].type == QBDI_MEMORY_WRITE) {
            // Writing the written value, the size and the address
            printf("\tWrote 0x%" PRIRWORD " on %u bytes at 0x%" PRIRWORD "\n",␣
↪memAccesses[i].value, (unsigned) memAccesses[i].size, memAccesses[i].accessAddress);
        }
    }
    printf("\n");
    free(memAccesses);

    return QBDI_CONTINUE;
}

static const size_t STACK_SIZE = 0x100000; // 1MB
static const rword FAKE_RET_ADDR = 0x40000;

int main(int argc, char **argv) {
    uint8_t *fakestack = NULL;
    VMInstanceRef vm;
    GPRState *state;

    if(argc < 2) {
        printf("Please give a password as first argument\n");
        return 1;
    }

    printf("Initializing VM ...\n");
    // Constructing a new QBDI vm
    qbdi_initVM(&vm, NULL, NULL);
    // Registering a callback on every memory write to our onwrite() function
    qbdi_addMemAccessCB(vm, QBDI_MEMORY_WRITE, onwrite, NULL);
    // Instrument this module
    qbdi_addInstrumentedModuleFromAddr(vm, (rword) &cryptolock);
    // Get a pointer to the GPR state of the vm
    state = qbdi_getGPRState(vm);
```

```
    // Setup initial GPR state, this fakestack will produce a ret FAKE_RET_ADDR at␣
↪the end of the execution
    // Also setup one argument on the stack which is the password string
    qbdi_allocateVirtualStack(state, STACK_SIZE, &fakestack);
    qbdi_simulateCall(state, FAKE_RET_ADDR, 1, argv[1]);

    printf("Running cryptolock(\"%s\")\n", argv[1]);
    qbdi_run(vm, (rword) cryptolock, (rword) FAKE_RET_ADDR);
    // Getting the return value from the call
    const char* ret = (const char*)QBDI_GPR_GET(state, REG_RETURN);
    // If it is not null, display it
    if(ret != NULL) {
        printf("Returned \"%s\"\n", ret);
    }
    else {
        printf("Returned null\n");
    }

    qbdi_terminateVM(vm);
    qbdi_alignedFree(fakestack);

    return 0;
}
```

VM C++

Initialization

The *QBDI::VM::VM()* constructor is used to create a new VM instance, using host CPU as a reference:

```
QBDI::VM *vm = new QBDI::VM();
```

QBDI::VM::**VM**(**const** std::string &*cpu* = "", **const** std::vector<std::string> &*mattrs* = {})
Construct a new VM for a given CPU with specific attributes

Parameters

- `cpu`: The name of the CPU
- `mattrs`: A list of additional attributes

The *QBDI::VM::VM()* constructor can also take two optional parameters used to specify the CPU type or architecture and additional attributes of the CPU. Those are the same used by LLVM. Here's how you would initialize a VM for an ARM Cortex-A9 supporting the VFP2 floating point instructions:

```
QBDI::VM *vm = new QBDI::VM("cortex-a9", {"vfp2"});
```

State Management

The state of the guest processor can be queried and modified using two architectures dependent structures `GPRState` and `FPRState` which are detailed below for each architecture (*X86*, *X86_64* and *ARM*).

The *QBDI::VM::getGPRState()* and *QBDI::VM::getFPRState()* allow to query the current state of the guest processor while the *QBDI::VM::setGPRState()* and *QBDI::VM::setFPRState()* allow to modify it:

```
QBDI::GPRState gprState; // local GPRState structure
gprState = *(vm->getGPRState()); // Copy the vm structure into our local structure
gprState.rax = (QBDI::rword) 42; // Set the value of the RAX register
vm->setGPRState(&gprState); // Copy our local structure to the vm structure
```

As *QBDI::VM::getGPRState()* and *QBDI::VM::getFPRState()* return pointers to the VM context, it is possible to use them to modify the GPR and FPR states directly. The code below is equivalent to the example from above but avoid unnecessary copies:

```
vm->getGPRState()->rax = (QBDI::rword) 42;
```

GPRState *QBDI::VM::**getGPRState**() **const**
Obtain the current general purpose register state.

Return A structure containing the GPR state.

FPRState *QBDI::VM::**getFPRState**() **const**
Obtain the current floating point register state.

Return A structure containing the FPR state.

void QBDI::VM::**setGPRState**(GPRState **gprState*)
Set the GPR state

Parameters

- gprState: A structure containing the GPR state.

void QBDI::VM::**setFPRState**(FPRState **fprState*)
Set the FPR state

Parameters

- fprState: A structure containing the FPR state.

X86

The X86 QBDI::GPRState structure is the following:

```
/*! X86 General Purpose Register context.
 */
typedef struct {
    rword eax;
    rword ebx;
    rword ecx;
    rword edx;
    rword esi;
    rword edi;
    rword ebp;
    rword esp;
    rword eip;
    rword eflags;
} GPRState;
//
```

The X86 `FPRState` structure is the following:

```
/*! X86 Floating Point Register context.
 */
typedef struct {
    union {
        FPControl     fcw;      /* x87 FPU control word */
        uint16_t      rfcw;
    };
    union {
        FPStatus      fsw;      /* x87 FPU status word */
        uint16_t      rfsw;
    };
    uint8_t           ftw;          /* x87 FPU tag word */
    uint8_t           rsrv1;        /* reserved */
    uint16_t          fop;          /* x87 FPU Opcode */
    uint32_t          ip;           /* x87 FPU Instruction Pointer offset */
    uint16_t          cs;           /* x87 FPU Instruction Pointer Selector */
    uint16_t          rsrv2;        /* reserved */
    uint32_t          dp;           /* x87 FPU Instruction Operand(Data) Pointer␣
↪offset */
    uint16_t          ds;           /* x87 FPU Instruction Operand(Data) Pointer␣
↪Selector */
    uint16_t          rsrv3;        /* reserved */
    uint32_t          mxcsr;        /* MXCSR Register state */
    uint32_t          mxcsrmask;    /* MXCSR mask */
    MMSTReg           stmm0;        /* ST0/MM0   */
    MMSTReg           stmm1;        /* ST1/MM1  */
    MMSTReg           stmm2;        /* ST2/MM2  */
    MMSTReg           stmm3;        /* ST3/MM3  */
    MMSTReg           stmm4;        /* ST4/MM4  */
    MMSTReg           stmm5;        /* ST5/MM5  */
    MMSTReg           stmm6;        /* ST6/MM6  */
    MMSTReg           stmm7;        /* ST7/MM7  */
    char              xmm0[16];     /* XMM 0  */
    char              xmm1[16];     /* XMM 1  */
    char              xmm2[16];     /* XMM 2  */
    char              xmm3[16];     /* XMM 3  */
    char              xmm4[16];     /* XMM 4  */
    char              xmm5[16];     /* XMM 5  */
    char              xmm6[16];     /* XMM 6  */
    char              xmm7[16];     /* XMM 7  */
    char              reserved[14*16];
    char              ymm0[16];     /* YMM0[255:128] */
    char              ymm1[16];     /* YMM1[255:128] */
    char              ymm2[16];     /* YMM2[255:128] */
    char              ymm3[16];     /* YMM3[255:128] */
    char              ymm4[16];     /* YMM4[255:128] */
    char              ymm5[16];     /* YMM5[255:128] */
    char              ymm6[16];     /* YMM6[255:128] */
    char              ymm7[16];     /* YMM7[255:128] */
} FPRState;
//
```

X86_64

The X86_64 `QBDI::GPRState` structure is the following:

```
/*! X86_64 General Purpose Register context.
 */
typedef struct {
    rword rax;
    rword rbx;
    rword rcx;
    rword rdx;
    rword rsi;
    rword rdi;
    rword r8;
    rword r9;
    rword r10;
    rword r11;
    rword r12;
    rword r13;
    rword r14;
    rword r15;
    rword rbp;
    rword rsp;
    rword rip;
    rword eflags;

} GPRState;
//
```

The X86_64 `FPRState` structure is the following:

```
/*! X86_64 Floating Point Register context.
 */
typedef struct {
    union {
        FPControl     fcw;      /* x87 FPU control word */
        uint16_t      rfcw;
    };
    union {
        FPStatus      fsw;      /* x87 FPU status word */
        uint16_t      rfsw;
    };
    uint8_t           ftw;          /* x87 FPU tag word */
    uint8_t           rsrv1;        /* reserved */
    uint16_t          fop;          /* x87 FPU Opcode */
    uint32_t          ip;           /* x87 FPU Instruction Pointer offset */
    uint16_t          cs;           /* x87 FPU Instruction Pointer Selector */
    uint16_t          rsrv2;        /* reserved */
    uint32_t          dp;           /* x87 FPU Instruction Operand(Data) Pointer␣
↪offset */
    uint16_t          ds;           /* x87 FPU Instruction Operand(Data) Pointer␣
↪Selector */
    uint16_t          rsrv3;        /* reserved */
    uint32_t          mxcsr;        /* MXCSR Register state */
    uint32_t          mxcsrmask;    /* MXCSR mask */
    MMSTReg           stmm0;        /* ST0/MM0   */
    MMSTReg           stmm1;        /* ST1/MM1  */
    MMSTReg           stmm2;        /* ST2/MM2  */
    MMSTReg           stmm3;        /* ST3/MM3  */
    MMSTReg           stmm4;        /* ST4/MM4  */
    MMSTReg           stmm5;        /* ST5/MM5  */
    MMSTReg           stmm6;        /* ST6/MM6  */
```

```
    MMSTReg             stmm7;          /* ST7/MM7  */
    char                xmm0[16];       /* XMM 0  */
    char                xmm1[16];       /* XMM 1  */
    char                xmm2[16];       /* XMM 2  */
    char                xmm3[16];       /* XMM 3  */
    char                xmm4[16];       /* XMM 4  */
    char                xmm5[16];       /* XMM 5  */
    char                xmm6[16];       /* XMM 6  */
    char                xmm7[16];       /* XMM 7  */
    char                xmm8[16];       /* XMM 8  */
    char                xmm9[16];       /* XMM 9  */
    char                xmm10[16];      /* XMM 10  */
    char                xmm11[16];      /* XMM 11  */
    char                xmm12[16];      /* XMM 12  */
    char                xmm13[16];      /* XMM 13  */
    char                xmm14[16];      /* XMM 14  */
    char                xmm15[16];      /* XMM 15  */
    char                reserved[6*16];
    char                ymm0[16];       /* YMM0[255:128] */
    char                ymm1[16];       /* YMM1[255:128] */
    char                ymm2[16];       /* YMM2[255:128] */
    char                ymm3[16];       /* YMM3[255:128] */
    char                ymm4[16];       /* YMM4[255:128] */
    char                ymm5[16];       /* YMM5[255:128] */
    char                ymm6[16];       /* YMM6[255:128] */
    char                ymm7[16];       /* YMM7[255:128] */
    char                ymm8[16];       /* YMM8[255:128] */
    char                ymm9[16];       /* YMM9[255:128] */
    char                ymm10[16];      /* YMM10[255:128] */
    char                ymm11[16];      /* YMM11[255:128] */
    char                ymm12[16];      /* YMM12[255:128] */
    char                ymm13[16];      /* YMM13[255:128] */
    char                ymm14[16];      /* YMM14[255:128] */
    char                ymm15[16];      /* YMM15[255:128] */
} FPRState;
//
```

ARM

The ARM `GPRState` structure is the following:

```
/*! ARM General Purpose Register context.
 */
typedef struct {
    rword r0;
    rword r1;
    rword r2;
    rword r3;
    rword r4;
    rword r5;
    rword r6;
    rword r7;
    rword r8;
    rword r9;
    rword r10;
    rword r12;
```

```
    rword fp;
    rword sp;
    rword lr;
    rword pc;
    rword cpsr;

} GPRState;
//
```

The ARM `FPRState` structure is the following:

```
/*! ARM Floating Point Register context.
 */
typedef struct {
    float s[QBDI_NUM_FPR];
} FPRState;
//
```

State Initialization

Since the instrumented software and the VM need to run on different stacks, helper initialization functions exist for this purpose. The *`QBDI::alignedAlloc()`* function offers a cross-platform interface for aligned allocation which is required for allocating a stack. The *`QBDI::allocateVirtualStack()`* allows to allocate this stack and also setup the `QBDI::GPRState` accordingly such that the required registers point to this stack. The *`QBDI::simulateCall()`* allows to simulate the call to a function by modifying the `QBDI::GPRState` and stack to setup the return address and the arguments. The *Fibonacci* example is using those functions to call a function through the QBDI.

void *`QBDI::`**`alignedAlloc`**(size_t *size*, size_t *align*)

Allocate a block of memory of a specified sized with an aligned base address.

Return Pointer to the allocated memory or NULL in case an error was encountered.

Parameters

- `size`: Allocation size in bytes.
- `align`: Base address alignement in bytes.

bool `QBDI::`**`allocateVirtualStack`**(GPRState **ctx*, uint32_t *stackSize*, uint8_t ***stack*)

Allocate a new stack and setup the GPRState accordingly. The allocated stack needs to be freed with *alignedFree()*.

Return True if stack allocation was successfull.

Parameters

- `ctx`: GPRState which will be setup to use the new stack.
- `stackSize`: Size of the stack to be allocated.
- `stack`: The newly allocated stack pointer will be returned in the variable pointed by stack.

void `QBDI::`**`simulateCall`**(GPRState **ctx*, rword *returnAddress*, **const** std::vector<rword> &*args* = {})

Simulate a call by modifying the stack and registers accordingly (std::vector version).

Parameters

- `ctx`: GPRState where the simulated call will be setup. The state needs to point to a valid stack for example setup with *allocateVirtualStack()*.
- `returnAddress`: Return address of the call to simulate.
- `args`: A list of arguments.

Execution

Starting a Run

The *QBDI::VM::run()* method is used to start the execution of a piece of code by the DBI. It first sets the Program Counter of the VM CPU state to the start parameter then runs the code through the DBI until a control flow instruction branches on the stop parameter. See the *Fibonacci* example for a basic usage of QBDI and the *QBDI::VM::run()* method.

bool `QBDI::VM::run` (rword *start*, rword *stop*)

Start the execution by the DBI.

Return True if at least one block has been executed.

Parameters

- `start`: Address of the first instruction to execute.
- `stop`: Stop the execution when this instruction is reached.

QBDI::VM::run() is a low level method which implies that the `QBDI::GPRState` has been properly initialized before.

But a very common usage of QBDI is to start instrumentation on a function call. The *QBDI::VM::call()* is a helper method which behaves like a function call. It takes a function pointer, a list of arguments, and return the call's result. The only requirement is a valid (and properly aligned) stack pointer. In most cases, using *QBDI::allocateVirtualStack()* is enough, as a call rarely needs to work on the original thread stack.

bool `QBDI::VM::call` (rword *retval, rword *function*, **const** std::vector<rword> &*args* = {})

Call a function using the DBI (and its current state).

Example:

```
// perform (with QBDI) a call similar to (*funcPtr)(42);
uint8_t *fakestack = nullptr;
QBDI::VM *vm = new QBDI::VM();
QBDI::GPRState *state = vm->getGPRState();
QBDI::allocateVirtualStack(state, 0x1000000, &fakestack);
vm->addInstrumentedModuleFromAddr(funcPtr);
rword retVal;
vm->call(&retVal, funcPtr, {42});
```

Return True if at least one block has been executed.

Parameters

- `[retval]`: Pointer to the returned value (optional).
- `function`: Address of the function start instruction.
- `args`: A list of arguments.

Execution Filtering

For performance reasons but also integrity reasons it is rarely preferable to instrument the entirety of the guest code. The VM stores a set of instrumented ranges, when the execution gets out of those ranges the VM attempts to transfer the execution to a real execution via the `QBDI::ExecBroker` which implements a return address hooking mechanism. As this hooking process is only a heuristic, it is neither guaranteed to be triggered as soon as the execution gets out of the set of instrumented ranges nor guaranteed to actually catch the return to the set of instrumented ranges.

Warning: By default the set of instrumented ranges is empty, a call to *QBDI::VM::run()* will thus very quickly escape from the instrumentation process through an execution transfer made by the `QBDI::ExecBroker`.

The basic methods to manipulate this set of instrumented ranges are *QBDI::VM::addInstrumentedRange()* and *QBDI::VM::removeInstrumentedRange()*.

void `QBDI::VM::`**`addInstrumentedRange`**(rword *start*, rword *end*)
Add an address range to the set of instrumented address ranges.

Parameters

- `start`: Start address of the range (included).
- `end`: End address of the range (excluded).

void `QBDI::VM::`**`removeInstrumentedRange`**(rword *start*, rword *end*)
Remove an address range from the set of instrumented address ranges.

Parameters

- `start`: Start address of the range (included).
- `end`: End address of the range (excluded).

As manipulating address ranges can be problematic, helpers API allow to use process memory maps information. The *QBDI::VM::addInstrumentedModule()* and *QBDI::VM::removeInstrumentedModule()* allow to use executable module names instead of address ranges. The currently loaded module names can be queried using *QBDI::getModuleNames()*. The *QBDI::VM::addInstrumentedModuleFromAddr()* and *QBDI::VM::removeInstrumentedModuleFromAddr()* methods allow to use any known address from a module to instrument it.

The *QBDI::VM::instrumentAllExecutableMaps()* allows to instrument every memory map which is executable, the undesired address range can later be removed using the previous APIs in a blacklist approach. This approach is demonstrated in the *The Dude* example where sub-functions are removed in function of a trace level supplied as argument.

The *QBDI::VM::removeAllInstrumentedRanges()* can be used to remove **all** ranges recorded through previous APIs (after this call, the VM will have no range left to instrument).

Below is an example of how to obtain, iterate then print the module names using the API.

```
#include <iostream>

#include <QBDI.h>

int main(int argc, char** argv) {
    for(const QBDI::MemoryMap& m :  QBDI::getCurrentProcessMaps()) {
        std::cout << m.name << " (" << m.permission << ") ";
```

```
        m.range.display(std::cout);
        std::cout << std::endl;
    }
}
```

struct QBDI::**MemoryMap**

Map of a memory area (region).

Public Functions

MemoryMap(rword *start*, rword *end*, *Permission* *permission*, std::string *name*)

Construct a new *MemoryMap* (given some properties).

Parameters

- `start`: *Range* start value.
- `end`: *Range* end value (always excluded).
- `permission`: Region access rights (PF_READ, PF_WRITE, PF_EXEC).
- `name`: Region name (useful when a region is mapping a module).

MemoryMap(*Range*<rword> *range*, *Permission* *permission*, std::string *name*)

Construct a new *MemoryMap* (given some properties).

Parameters

- `range`: A range of memory (region), delimited between a start and an (excluded) end address.
- `permission`: Region access rights (PF_READ, PF_WRITE, PF_EXEC).
- `name`: Region name (useful when a region is mapping a module).

Public Members

Range<rword> **range**

A range of memory (region), delimited between a start and an (excluded) end address.

Permission **permission**

Region access rights (PF_READ, PF_WRITE, PF_EXEC).

std::string **name**

Region name or path (useful when a region is mapping a module).

enum QBDI::**Permission**

Memory access rights.

Values:

PF_NONE = 0

No access

PF_READ = 1

Read access

PF_WRITE = 2

Write access

PF_EXEC = 4
Execution access

template <typename *T*>
class QBDI::**Range**

Public Functions

Range(T *start*, T *end*)
Construct a new range.

Parameters

- start: *Range* start value.
- end: *Range* end value (excluded).

T **size**() **const**
Return the total length of a range.

bool **operator==**(**const** *Range* &*r*) **const**
Return True if two ranges are equal (same boundaries).

Return True if equal.

Parameters

- r: *Range* to check.

bool **contains**(T *t*) **const**
Return True if an value is inside current range boundaries.

Return True if contained.

Parameters

- t: Value to check.

bool **contains**(*Range*<T> *r*) **const**
Return True if a range is inside current range boundaries.

Return True if contained.

Parameters

- r: *Range* to check.

bool **overlaps**(*Range*<T> *r*) **const**
Return True if a range is overlapping current range lower or/and upper boundary.

Return True if overlapping.

Parameters

- r: *Range* to check.

void **display**(std::ostream &*os*) **const**
Pretty print a range

Parameters

- `os`: An output stream.

Range<T> **intersect** (*Range*<T> *r*) **const**
Return the intersection of two ranges.

Return A new range.

Parameters

- `r`: *Range* to intersect with current range.

Public Members

T **start**
Range start value.

T **end**
Range end value (always excluded).

std::vector<*MemoryMap*> `QBDI::`**getCurrentProcessMaps** (bool *full_path* = false)
Get a list of all the memory maps (regions) of the current process.

Return A vector of *MemoryMap* object.

Parameters

- `full_path`: Return the full path of the module in name field

std::vector<std::string> `QBDI::`**getModuleNames** ()
Get a list of all the module names loaded in the process memory.

Return A vector of string of module names.

bool `QBDI::VM::`**addInstrumentedModule** (**const** std::string &*name*)
Add the executable address ranges of a module to the set of instrumented address ranges.

Return True if at least one range was added to the instrumented ranges.

Parameters

- `name`: The module's name.

bool `QBDI::VM::`**removeInstrumentedModule** (**const** std::string &*name*)
Remove the executable address ranges of a module from the set of instrumented address ranges.

Return True if at least one range was removed from the instrumented ranges.

Parameters

- `name`: The module's name.

bool `QBDI::VM::`**addInstrumentedModuleFromAddr** (rword *addr*)
Add the executable address ranges of a module to the set of instrumented address ranges using an address belonging to the module.

Return True if at least one range was added to the instrumented ranges.

Parameters

- addr: An address contained by module's range.

bool QBDI::VM::**removeInstrumentedModuleFromAddr**(rword *addr*)

Remove the executable address ranges of a module from the set of instrumented address ranges using an address belonging to the module.

Return True if at least one range was removed from the instrumented ranges.

Parameters

- addr: An address contained by module's range.

bool QBDI::VM::**instrumentAllExecutableMaps**()

Adds all the executable memory maps to the instrumented range set.

Return True if at least one range was added to the instrumented ranges.

Warning: Instrumenting the libc very often results in deadlock problems as it is also used by QBDI. It is thus recommended to always exclude it from the instrumentation.

void QBDI::VM::**removeAllInstrumentedRanges**()

Remove all instrumented ranges.

Instrumentation

Instrumentation allows to run additional code alongside the original code of the software. There are three types of instrumentations implemented by the VM. The first one are instruction event callbacks where the execution of an instruction, under certain conditions, triggers a callback from the guest to the host. The second one are VM event callbacks where certain actions taken by the VM itself trigger a callback. The last one are custom instrumentations which allow to use the internal instrumentation mechanism of the VM.

Instruction Callback

The instruction callbacks follow the prototype of *QBDI::InstCallback*. A user supplied data pointer parameter can be passed to the callback. Here's a simple callback which would increment an integer passed as a pointer through the data parameter:

```
QBDI::VMAction increment(QBDI::VMInstanceRef vm, QBDI::GPRState *gprState,␣
↪QBDI::FPRState *fprState, void *data) {
    int *counter = (int*) data;
    *counter += 1;
    return QBDI::VMAction::CONTINUE;
}
```

The return value of this method is very important to determine how the VM should handle the resuming of execution. CONTINUE will directly switch back to the guest code in the current ExecBlock without further processing by the VM. This means that modification of the Program Counter will not be taken into account and modifications of the program code will only be effective after the end of the current basic block (provided the code cache is cleared, which is not currently exported via the API). One can force those changes to be taken into account by breaking from the ExecBlock execution and returning early inside the VM using the BREAK_TO_VM. STOP is not yet implemented.

The *QBDI::VM::addCodeCB()* method allows to add a callback on every instruction (inside the instrumented range). If we register the previous callback to be called before every instruction, we effectively get an instruction counting instrumentation:

```
int counter = 0;
vm->addCodeCB(QBDI::PREINST, increment, &counter);
vm->run(...); // Run the VM of some piece of code
// counter contains the number of instructions executed
std::cout << "Instruction count: " << counter << std::endl;
```

enum QBDI::**InstPosition**

Position relative to an instruction.

Values:

PREINST = 0

Positioned before the instruction.

POSTINST

Positioned after the instruction.

typedef *VMAction* (*QBDI::**InstCallback**)(VMInstanceRef vm, GPRState *gprState, FPRState *fprState, void *data)

Instruction callback function type.

Return The callback result used to signal subsequent actions the VM needs to take.

Parameters

- vm: VM instance of the callback.
- gprState: A structure containing the state of the General Purpose Registers. Modifying it affects the VM execution accordingly.
- fprState: A structure containing the state of the Floating Point Registers. Modifying it affects the VM execution accordingly.
- data: User defined data which can be defined when registering the callback.

enum QBDI::**VMAction**

The callback results.

Values:

CONTINUE = 0

The execution of the basic block continues.

BREAK_TO_VM = 1

The execution breaks and returns to the VM causing a complete reevaluation of the execution state. A BREAK_TO_VM is needed to ensure that modifications of the Program Counter or the program code are taken into account.

STOP = 2

Stops the execution of the program. This causes the run function to return early.

uint32_t QBDI::VM::**addCodeCB**(*InstPosition pos*, *InstCallback cbk*, void **data*)

Register a callback event for every instruction executed.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- pos: Relative position of the event callback (PREINST / POSTINST).

- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

It is also possible to register an *`QBDI::InstCallback`* for a specific instruction address or address range with the *`QBDI::VM::addCodeAddrCB()`* and *`QBDI::VM::addCodeRangeCB()`* methods. These allow to fine-tune the instrumentation to specific codes or portions of them.

uint32_t `QBDI::VM::`**`addCodeAddrCB`**(rword *address*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

Register a callback for when a specific address is executed.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- `address`: Code address which will trigger the callback.
- `pos`: Relative position of the callback (PREINST / POSTINST).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

uint32_t `QBDI::VM::`**`addCodeRangeCB`**(rword *start*, rword *end*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

Register a callback for when a specific address range is executed.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- `start`: Start of the address range which will trigger the callback.
- `end`: End of the address range which will trigger the callback.
- `pos`: Relative position of the callback (PREINST / POSTINST).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

uint32_t `QBDI::VM::`**`addMnemonicCB`**(**`const`** char **mnemonic*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

Register a callback event if the instruction matches the mnemonic.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- `mnemonic`: Mnemonic to match.
- `pos`: Relative position of the event callback (PREINST / POSTINST).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

Note: Mnemonics can be instrumented using LLVM convention (You can register a callback on *ADD64rm* or *ADD64rr* for instance).

Note: You can also use "*" as a wildcard. (eg : *ADD*rr*)

If the execution of an instruction triggers more than one callback, those will be called in the order they were added to the VM.

Memory Callback

The memory callbacks (currently only supported under x86-64 and x86) allow to trigger callbacks on specific memory events. They use the same callback prototype, *QBDI::InstCallback*, than instruction callback. They can be registered using the *QBDI::VM::addMemAccessCB()* API. The API takes care of enabling the corresponding inline memory access logging instrumentation using *QBDI::VM::recordMemoryAccess()*. The memory accesses themselves are not directly provided as a callback parameter and need to be retrieved using the memory access APIs (see *Memory Access Analysis*). The *Cryptolock* example shows how to use those APIs to log the memory writes.

The type parameter allows to filter the callback on specific memory access type:

- *QBDI::MEMORY_READ* triggers the callback **before** every instruction performing a memory read.
- *QBDI::MEMORY_WRITE* triggers the callback **after** every instruction performing a memory write.
- *QBDI::MEMORY_READ_WRITE* triggers the callback **after** every instruction performing a memory read and/or write.

uint32_t QBDI::VM::**addMemAccessCB**(*MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)
Register a callback event for every memory access matching the type bitfield made by the instructions.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- `type`: A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

To enable more flexibility on the callback filtering, *QBDI::VM::addMemAddrCB()* and *QBDI::VM::addMemRangeCB()* allow to filter for memory accesses targeting a specific memory address or range. However those callbacks are virtual callbacks: they cannot be directly triggered by the instrumentation because the access address needs to be checked dynamically but are instead triggered by a gate callback which takes care of the dynamic access address check and forwards or not the event to the virtual callback. This system thus has the same overhead as *QBDI::VM::addMemAccessCB()* and is only meant as an API helper.

uint32_t QBDI::VM::**addMemAddrCB**(rword *address*, *MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)
Add a virtual callback which is triggered for any memory access at a specific address matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- `address`: Code address which will trigger the callback.
- `type`: A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).

- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

uint32_t `QBDI::VM::addMemRangeCB`(rword *start*, rword *end*, *MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)

Add a virtual callback which is triggered for any memory access in a specific address range matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- `start`: Start of the address range which will trigger the callback.
- `end`: End of the address range which will trigger the callback.
- `type`: A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).
- `cbk`: A function pointer to the callback.
- `data`: User defined data passed to the callback.

VM Events

VM events are triggered by the VM itself when it takes specific actions related to the execution of the instrumented program. The callback prototype *QBDI::VMCallback* is different as it is triggered by the VM and not by an instruction. The callback receives a *QBDI::VMState* structure. *QBDI::VMCallback* can be registered for several events at the same time by combining several values of *QBDI::VMEvent* in a mask using the | binary operator.

typedef *VMAction* (*`QBDI::VMCallback`)(VMInstanceRef vm, **const** *VMState* *vmState, GPRState *gprState, FPRState *fprState, void *data)

VM callback function type.

Return The callback result used to signal subsequent actions the VM needs to take.

Parameters

- `vm`: VM instance of the callback.
- `vmState`: A structure containing the current state of the VM.
- `gprState`: A structure containing the state of the General Purpose Registers. Modifying it affects the VM execution accordingly.
- `fprState`: A structure containing the state of the Floating Point Registers. Modifying it affects the VM execution accordingly.
- `data`: User defined data which can be defined when registering the callback.

uint32_t `QBDI::VM::addVMEventCB`(*VMEvent* *mask*, *VMCallback* *cbk*, void **data*)

Register a callback event for a specific VM event.

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- `mask`: A mask of VM event type which will trigger the callback.
- `cbk`: A function pointer to the callback.

- `data`: User defined data passed to the callback.

struct QBDI::**VMState**

Structure describing the current VM state

Public Members

VMEvent **event**

The event(s) which triggered the callback (must be checked using a mask: event & BASIC_BLOCK_ENTRY).

rword **basicBlockStart**

The current basic block start address which can also be the execution transfer destination.

rword **basicBlockEnd**

The current basic block end address which can also be the execution transfer destination.

rword **sequenceStart**

The current sequence start address which can also be the execution transfer destination.

rword **sequenceEnd**

The current sequence end address which can also be the execution transfer destination.

rword **lastSignal**

Not implemented.

enum QBDI::**VMEvent**

Values:

SEQUENCE_ENTRY = 1

Triggered when the execution enters a sequence.

SEQUENCE_EXIT = 1<<1

Triggered when the execution exits from the current sequence.

BASIC_BLOCK_ENTRY = 1<<2

Triggered when the execution enters a basic block.

BASIC_BLOCK_EXIT = 1<<3

Triggered when the execution exits from the current basic block.

BASIC_BLOCK_NEW = 1<<4

Triggered when the execution enters a new (~unknown) basic block.

EXEC_TRANSFER_CALL = 1<<5

Triggered when the ExecBroker executes an execution transfer.

EXEC_TRANSFER_RETURN = 1<<6

Triggered when the ExecBroker returns from an execution transfer.

SYSCALL_ENTRY = 1<<7

Not implemented.

SYSCALL_EXIT = 1<<8

Not implemented.

SIGNAL = 1<<9

Not implemented.

Custom Instrumentation

Custom instrumentation can be pushed by inserting your own *QBDI::InstrRule* in the VM. This requires using private headers to define your own rules but allows a very high level of flexibility.

uint32_t QBDI::VM::**addInstrRule**(*InstrRule rule*)
Add a custom instrumentation rule to the VM. Requires internal headers

Return The id of the registered instrumentation (or VMError::INVALID_EVENTID in case of failure).

Parameters

- rule: A custom instrumentation rule.

Removing Instrumentations

Several callbacks can be registered for the same event and will run in the order they were added. The id returned by this method can be used to modify the callback afterward. The *QBDI::VM::deleteInstrumentation()* method allows to remove an instrumentation by id while the *QBDI::VM::deleteAllInstrumentations()* method will remove all instrumentations. This code would do two different executions with different callbacks:

```
uint32_t cb1 = vm->addCodeCB(QBDI::InstPosition::PREINST, Callback1, &some_data);
uint32_t cb2 = vm->addVMEventCB(QBDI::VMEvent::EXEC_TRANSFER_CALL |␣
↪QBDI::VMEvent::BASIC_BLOCK_ENTRY, Callback2, &some_data);
vm->run(...); // Run the VM of some piece of code with all instrumentations
vm->deleteInstrumentation(cb1);
vm->run(...); // Run the VM of some piece of code with some instrumentation
vm->deleteAllInstrumentations();
vm->run(...); // Run the VM of some piece of code without instrumentation
```

bool QBDI::VM::**deleteInstrumentation**(uint32_t *id*)
Remove an instrumentation.

Return True if instrumentation has been removed.

Parameters

- id: The id of the instrumentation to remove.

void QBDI::VM::**deleteAllInstrumentations**()
Remove all the registered instrumentations.

Instruction Analysis

Callback based instrumentation has only limited utility if the current context of execution is not available. To obtain more information about an instruction, the VM can parse its internal structures for us and provide analysis results in a *QBDI::InstAnalysis* structure:

```
QBDI::CallbackResult increment(QBDI::VMInstanceRef vm, QBDI::GPRState *gprState,␣
↪QBDI::FPRState *fprState, void *data) {
    const QBDI::InstAnalysis *instAnalysis = vm->getInstAnalysis();

    std::cout << instAnalysis->disassembly << std::endl;

    return QBDI::CallbackResult::CONTINUE;
}
```

const *InstAnalysis* *QBDI::VM::**getInstAnalysis**(*AnalysisType* *type* = ANALYSIS_INSTRUCTION | ANALYSIS_DISASSEMBLY)

Obtain the analysis of an instruction metadata. Analysis results are cached in the VM. The validity of the returned pointer is only guaranteed until the end of the callback, else a deepcopy of the structure is required.

Return A *InstAnalysis* structure containing the analysis result.

Parameters

- [type]: Properties to retrieve during analysis. This argument is optional, defaulting to *QBDI::ANALYSIS_INSTRUCTION* | *QBDI::ANALYSIS_DISASSEMBLY*.

Analysis can gather all kinds of properties linked with the instruction. By default, instructions basic properties (address, mnemonic, ...), as well as disassembly, are returned (as this is a very common case). It's also possible to select which properties will be retrieved using *type* argument.

enum QBDI::**AnalysisType**

Instruction analysis type

Values:

ANALYSIS_INSTRUCTION = 1

Instruction analysis (address, mnemonic, ...)

ANALYSIS_DISASSEMBLY = 1<<1

Instruction disassembly

ANALYSIS_OPERANDS = 1<<2

Instruction operands analysis

ANALYSIS_SYMBOL = 1<<3

Instruction symbol

Every analysis has a performance cost, which can be reduced by selecting types carefully, and is amortized using a cache inside the VM.

struct QBDI::**InstAnalysis**

Structure containing analysis results of an instruction provided by the VM.

Public Members

const char ***mnemonic**

LLVM mnemonic (warning: NULL if !ANALYSIS_INSTRUCTION)

rword **address**

Instruction address

uint32_t **instSize**

Instruction size (in bytes)

bool **affectControlFlow**

true if instruction affects control flow

bool **isBranch**

true if instruction acts like a 'jump'

bool **isCall**

true if instruction acts like a 'call'

bool **isReturn**
: true if instruction acts like a 'return'

bool **isCompare**
: true if instruction is a comparison

bool **isPredicable**
: true if instruction contains a predicate (~is conditional)

bool **mayLoad**
: true if instruction 'may' load data from memory

bool **mayStore**
: true if instruction 'may' store data to memory

char ***disassembly**
: Instruction disassembly (warning: NULL if !ANALYSIS_DISASSEMBLY)

uint8_t **numOperands**
: Number of operands used by the instruction

OperandAnalysis ***operands**
: Structure containing analysis results of an operand provided by the VM. (warning: NULL if !ANALYSIS_OPERANDS)

const char ***symbol**
: Instruction symbol (warning: NULL if !ANALYSIS_SYMBOL or not found)

uint32_t **symbolOffset**
: Instruction symbol offset

const char ***module**
: Instruction module name (warning: NULL if !ANALYSIS_SYMBOL or not found)

uint32_t **analysisType**
: INTERNAL: Instruction analysis type (this should NOT be used)

If provided *QBDI::AnalysisType* is equal to *QBDI::ANALYSIS_OPERANDS*, then `QBDI::getInstAnalysis()` will also analyze all instruction operands, and fill an array of *QBDI::OperandAnalysis* (of length *numOperands*).

struct `QBDI::`**OperandAnalysis**
: Structure containing analysis results of an operand provided by the VM.

Public Members

OperandType **type**
: *Operand* type

OperandFlag **flag**
: *Operand* flag

rword **value**
: *Operand* value (if immediate), or register Id

uint8_t **size**
: *Operand* size (in bytes)

uint8_t **regOff**
: Sub-register offset in register (in bits)

uint16_t **regCtxIdx**
Register index in VM state

const char ***regName**
Register name

RegisterAccessType **regAccess**
Register access type (r, w, rw)

enum QBDI::**OperandType**
Operand type

Values:

OPERAND_INVALID = 0
Invalid operand

OPERAND_IMM
Immediate operand

OPERAND_GPR
Register operand

OPERAND_PRED
Predicate operand

enum QBDI::**OperandFlag**
Values:

OPERANDFLAG_NONE = 0
No flag

OPERANDFLAG_ADDR = 1<<0
The operand is used to compute an address

OPERANDFLAG_PCREL = 1<<1
The value of the operand is PC relative

OPERANDFLAG_UNDEFINED_EFFECT = 1<<2
The operand role isn't fully defined

enum QBDI::**RegisterAccessType**
Access type (R/W/RW) of a register operand

Values:

REGISTER_UNUSED = 0
Unused register

REGISTER_READ = 1
Register read access

REGISTER_WRITE = 1<<1
Register write access

REGISTER_READ_WRITE = 3
Register read/write access

Memory Access Analysis

QBDI VM can log memory access using inline instrumentation. This adds instrumentation rules which store the accessed addresses and values into specific memory variables called *instruction shadows*. To enable this memory

logging, the *QBDI::VM::recordMemoryAccess()* API can be used. This memory logging is also automatically enabled when calling one of the memory callbacks.

bool QBDI::VM::**recordMemoryAccess**(*MemoryAccessType type*)
Add instrumentation rules to log memory access using inline instrumentation and instruction shadows.

Return True if inline memory logging is supported, False if not or in case of error.

Parameters

- type: Memory mode bitfield to activate the logging for: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).

The memory access type always refers to either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE*, *QBDI::MEMORY_READ_WRITE* (which is a bit field combination of the two previous ones).

Once the logging has been enabled, *QBDI::VM::getInstMemoryAccess()* and *QBDI::VM::getBBMemoryAccess()* can be used to retrieve the memory accesses made by the last instruction or by the last basic block. These two APIs return *QBDI::MemoryAccess* structures. Write memory accesses are only returned if the instruction has already been executed (i.e. in the case of a QBDI::InstPosition::POSTINST). The *Cryptolock* example shows how to use those APIs to log the memory writes.

struct QBDI::**MemoryAccess**
Describe a memory access

Public Members

rword **instAddress**
Address of instruction making the access

rword **accessAddress**
Address of accessed memory

rword **value**
Value read from / written to memory

uint8_t **size**
Size of memory access (in bytes)

MemoryAccessType **type**
Memory access type (READ / WRITE)

enum QBDI::**MemoryAccessType**
Memory access type (read / write / . . .)

Values:

MEMORY_READ = 1
Memory read access

MEMORY_WRITE = 1<<1
Memory write access

MEMORY_READ_WRITE = 3
Memory read/write access

std::vector<*MemoryAccess*> QBDI::VM::**getInstMemoryAccess**() **const**
Obtain the memory accesses made by the last executed instruction.

Return List of memory access made by the instruction.

std::vector<*MemoryAccess*> `QBDI::VM::getBBMemoryAccess() const`
Obtain the memory accesses made by the last executed basic block.

Return List of memory access made by the instruction.

Cache management

QBDI provides a cache system for basic blocks that you might want to deal directly with it. There are a few functions that can help you with that.

bool `QBDI::VM::precacheBasicBlock`(rword *pc*)
Pre-cache a known basic block

Return True if basic block has been inserted in cache.

Parameters

- `pc`: Start address of a basic block

void `QBDI::VM::clearCache`(rword *start*, rword *end*)
Clear a specific address range from the translation cache.

Parameters

- `start`: Start of the address range to clear from the cache.
- `end`: End of the address range to clear from the cache.

void `QBDI::VM::clearAllCache`()
Clear the entire translation cache.

Free resources

Some resources returned by a VM instance must be manually freed using specialized functions. A VM instance itself must be destroyed using *delete*.

void `QBDI::alignedFree`(void **ptr*)
Free a block of aligned memory allocated with alignedAlloc.

Parameters

- `ptr`: Pointer to the allocated memory.

Examples

Fibonacci

This example instruments its own code to count the executed instructions and displays the disassembly of every instruction executed.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstdint>
```

```
#include <cstring>

#include "QBDI.h"

#define FAKE_RET_ADDR 42

int fibonacci(int n) {
    if(n <= 2)
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}


QBDI::VMAction countInstruction(QBDI::VMInstanceRef vm, QBDI::GPRState *gprState,
                                QBDI::FPRState *fprState, void *data) {
    // Cast data to our counter
    uint32_t* counter = (uint32_t*) data;
    // Obtain an analysis of the instruction from the vm
    const QBDI::InstAnalysis* instAnalysis = vm->getInstAnalysis();
    // Printing disassembly
    std::cout << std::setbase(16) << instAnalysis->address << ": "
        << instAnalysis->disassembly << std::endl << std::setbase(10);
    // Incrementing the instruction counter
    (*counter)++;
    // Signaling the VM to continue execution
    return QBDI::VMAction::CONTINUE;
}

QBDI::VMAction countRecursion(QBDI::VMInstanceRef vm, QBDI::GPRState *gprState,
                              QBDI::FPRState *fprState, void *data) {
    *((unsigned*) data) += 1;
    return QBDI::VMAction::CONTINUE;
}

static const size_t STACK_SIZE = 0x100000; // 1MB

int main(int argc, char** argv) {
    int n = 0;
    uint32_t counter = 0;
    unsigned recursions = 0;
    uint8_t *fakestack = nullptr;
    QBDI::GPRState *state;

    std::cout << "Initializing VM ..." << std::endl;
    // Constructing a new QBDI vm
    QBDI::VM *vm = new QBDI::VM();
    // Registering countInstruction() callback to be called after every instruction
    vm->addCodeCB(QBDI::POSTINST, countInstruction, &counter);
    // Registering countRecursion() callback to be called before the first␣
↪instruction of fibonacci
    vm->addCodeAddrCB((QBDI::rword) &fibonacci, QBDI::PREINST, countRecursion, &
↪recursions);

    // Get a pointer to the GPR state of the vm
    state = vm->getGPRState();
    // Setup initial GPR state, this fakestack will produce a ret NULL at the end of␣
↪the execution
    QBDI::allocateVirtualStack(state, STACK_SIZE, &fakestack);
```

```
    // Argument to the fibonnaci call
    if(argc >= 2) {
        n = atoi(argv[1]);
    }
    if(n < 1) {
        n = 1;
    }
    QBDI::simulateCall(state, FAKE_RET_ADDR, {(QBDI::rword) n});

    std::cout << "Running fibonacci(" << n << ") ..." << std::endl;
    // Instrument everything
    vm->instrumentAllExecutableMaps();
    // Run the DBI execution
    vm->run((QBDI::rword) fibonacci, (QBDI::rword) FAKE_RET_ADDR);
    std::cout << "fibonnaci ran in " << counter << " instructions, recursed " <<␣
↪recursions - 1
        << " times and returned " << QBDI_GPR_GET(state, QBDI::REG_RETURN) <<␣
↪std::endl;

    delete vm;
    QBDI::alignedFree(fakestack);

    return 0;
}
```

The Dude

This example instruments its own code to count the executed instructions and displays the disassembly of every instruction executed. A supplied trace level argument allows to disable the tracing in specific sub-functions using the APIs presented in *Execution Filtering*.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstdint>
#include <cstring>
#include <ctime>
#include <cinttypes>

#include "QBDI.h"


QBDI_NOINLINE uint64_t magicPow(uint64_t n, uint64_t e) {
    uint64_t r = 1;
    uint64_t i = 0;

    for(i = 0; i < e; i++) {
        r = (r*n);
    }
    return r;
}

QBDI_NOINLINE uint64_t magicHash(const char* secret) {
    uint64_t hash = 0;
    uint64_t acc = 1;
    int len = strlen(secret);
```

```
    int i = 0;

    for(i = 0; i < len; i++) {
        uint64_t magic = magicPow(secret[i], acc);
        hash ^= magic;
        acc = (acc + magic) % 256;
    }

    return hash;
}

QBDI_NOINLINE int thedude() {
    time_t t = 0;
    std::string name;
    uint64_t hash = 0;
    char *secret = nullptr;

    std::cout << "Hi I'm the dude." << std::endl;
    std::cout << "Give me your name and I'll give you a hash." << std::endl;
    std::cout << "So what's your name ? ";
    std::cin  >> name;
    time(&t);
    secret = new char[name.length() + 16 + 2];
    sprintf(secret, "%" PRIu64 ":%s", (uint64_t) t, name.c_str());
    std::cout << "Ok I'll give you the hash of " << secret << "." << std::endl;
    hash = magicHash(secret);
    std::cout << "Your hash is " << hash << "." << std::endl;
    std::cout << "No need to thank me." << std::endl;

    delete[] secret;
    return 0;
}

QBDI::VMAction count(QBDI::VMInstanceRef vm, QBDI::GPRState *gprState, QBDI::FPRState␣
↪*fprState, void *data) {
    // Cast data to our counter
    uint32_t* counter = (uint32_t*) data;
    // Obtain an analysis of the instruction from the vm
    const QBDI::InstAnalysis* instAnalysis = vm->getInstAnalysis();
    // Printing disassembly
    std::cout << std::setbase(16) << instAnalysis->address << ": " << instAnalysis->
↪disassembly << std::endl << std::setbase(10);
    // Incrementing the instruction counter
    (*counter)++;
    // Signaling the VM to continue execution
    return QBDI::VMAction::CONTINUE;
}

static const size_t STACK_SIZE = 0x100000; // 1MB
static const QBDI::rword FAKE_RET_ADDR = 0x40000;

int main(int argc, char **argv) {
    uint8_t *fakestack = nullptr;
    QBDI::GPRState *state;
    int traceLevel = 0;
    bool instrumented;
    uint32_t counter = 0;
```

```
    if(argc > 1) {
        traceLevel = atoi(argv[1]);
        if(traceLevel > 2) {
            traceLevel = 2;
        }
    }

    std::cout << "Initializing VM ..." << std::endl;
    // Constructing a new QBDI vm
    QBDI::VM *vm = new QBDI::VM();
    // Registering count() callback to be called after every instruction
    vm->addCodeCB(QBDI::POSTINST, count, &counter);

    // Get a pointer to the GPR state of the vm
    state = vm->getGPRState();
    // Setup initial GPR state, this fakestack will produce a ret NULL at the end of␣
↪the execution
    QBDI::allocateVirtualStack(state, STACK_SIZE, &fakestack);
    QBDI::simulateCall(state, FAKE_RET_ADDR);

    std::cout << "Running thedude() with trace level " << traceLevel << "..." <<␣
↪std::endl;
    // Select which part to instrument
    instrumented = vm->addInstrumentedModuleFromAddr((QBDI::rword) &main);
    if (instrumented) {
        if(traceLevel < 1) {
            vm->removeInstrumentedRange((QBDI::rword) magicHash, (QBDI::rword)␣
↪magicHash + 32);
        }
        if(traceLevel < 2) {
            vm->removeInstrumentedRange((QBDI::rword) magicPow, (QBDI::rword)␣
↪magicPow + 32);
        }
        // Run the DBI execution
        vm->run((QBDI::rword) thedude, (QBDI::rword) FAKE_RET_ADDR);
        std::cout << "thedude ran in " << counter << " instructions" << std::endl;
    } else {
        std::cout << "failed to find main module..." << std::endl;
    }
    delete vm;
    QBDI::alignedFree(fakestack);

    return 0;
}
```

Cryptolock

This example instruments its own code to display instruction performing memory writes and the written values. This can be used to discover the password of the cryptolock and reveal the message (hint: if the password is correct then the hash buffer is all zero).

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cstring>
```

```
#include "QBDI.h"

QBDI_NOINLINE void hashPassword(char* hash, const char* password) {
    char acc = 42;
    size_t hash_len = strlen(hash);
    size_t password_len = strlen(password);

    for(size_t i = 0; i < hash_len && i < password_len; i++) {
        hash[i] = (hash[i] ^ acc) - password[i];
        acc = password[i];
    }

}

char SECRET[] =
↪"\x29\x0d\x20\x00\x00\x00\x00\x0a\x65\x1f\x32\x00\x19\x0c\x4e\x1b\x2d\x09\x66\x0c\x1a\x06\x05\x06\x
↪";

QBDI_NOINLINE const char* getSecret(const char* password) {
    size_t password_len = strlen(password);
    for(size_t i = 0; i < sizeof(SECRET); i++) {
        SECRET[i] ^= password[i%password_len];
    }
    return SECRET;
}

QBDI_NOINLINE const char* cryptolock(const char* password) {
    char hash[] = "\x6f\x29\x2a\x29\x1a\x1c\x07\x01";

    hashPassword(hash, password);

    bool good = true;
    for(size_t i = 0; i < sizeof(hash); i++) {
        if(hash[i] != 0) {
            good = false;
        }
    }

    if(good) {
        return getSecret(password);
    }
    else {
        return nullptr;
    }
}

QBDI::VMAction onwrite(QBDI::VMInstanceRef vm, QBDI::GPRState *gprState,
                       QBDI::FPRState *fprState, void *data) {
    // Obtain an analysis of the instruction from the vm
    const QBDI::InstAnalysis* instAnalysis = vm->getInstAnalysis();
    // Obtain the instruction memory accesses
    std::vector<QBDI::MemoryAccess> memAccesses = vm->getInstMemoryAccess();

    // Printing disassembly
    std::cout << std::setbase(16) << instAnalysis->address << ": "
              << instAnalysis->disassembly << std::endl << std::setbase(10);
    // Printing write memory accesses
    for(const QBDI::MemoryAccess& memAccess : memAccesses) {
```

```
        // Checking the access mode
        if(memAccess.type == QBDI::MEMORY_WRITE) {
            // Writing the written value, the size and the address
            std::cout << "\t" << "Wrote 0x" << std::setbase(16) << memAccess.value <<
↪" on "
                      << std::setbase(10) << (int) memAccess.size << " bytes at 0x"
                      << std::setbase(16) << memAccess.accessAddress << std::endl;
        }
    }
    std::cout << std::endl;

    return QBDI::CONTINUE;
}

static const size_t STACK_SIZE = 0x100000; // 1MB
static const QBDI::rword FAKE_RET_ADDR = 0x40000;

int main(int argc, char **argv) {
    uint8_t *fakestack = nullptr;
    QBDI::GPRState *state;

    if(argc < 2) {
        std::cout << "Please give a password as first argument" << std::endl;
        return 1;
    }

    std::cout << "Initializing VM ..." << std::endl;
    // Constructing a new QBDI vm
    QBDI::VM *vm = new QBDI::VM();
    // Registering a callback on every memory write to our onwrite() function
    vm->addMemAccessCB(QBDI::MEMORY_WRITE, onwrite, nullptr);
    // Instrument this module
    vm->addInstrumentedModuleFromAddr((QBDI::rword) &cryptolock);
    // Get a pointer to the GPR state of the vm
    state = vm->getGPRState();
    // Setup initial GPR state, this fakestack will produce a ret FAKE_RET_ADDR at␣
↪the end of the execution
    // Also setup one argument on the stack which is the password string
    QBDI::allocateVirtualStack(state, STACK_SIZE, &fakestack);
    QBDI::simulateCall(state, FAKE_RET_ADDR, {(QBDI::rword) argv[1]});

    std::cout << "Running cryptolock(\"" << argv[1] <<"\")" << std::endl;
    vm->run((QBDI::rword) cryptolock, (QBDI::rword) FAKE_RET_ADDR);
    // Getting the return value from the call
    const char* ret = (const char*)QBDI_GPR_GET(state, QBDI::REG_RETURN);
    // If it is not null, display it
    if(ret != nullptr) {
        std::cout << "Returned \"" << ret << "\"" << std::endl;
    }
    else {
        std::cout << "Returned null" << std::endl;
    }

    delete vm;
    QBDI::alignedFree(fakestack);

    return 0;
}
```

Reference

class QBDI::**VM**

Public Functions

VM(**const** std::string &*cpu* = "", **const** std::vector<std::string> &*mattrs* = {})

GPRState ***getGPRState**() **const**

FPRState ***getFPRState**() **const**

void **setGPRState**(GPRState **gprState*)

void **setFPRState**(FPRState **fprState*)

void **addInstrumentedRange**(rword *start*, rword *end*)

bool **addInstrumentedModule**(**const** std::string &*name*)

bool **addInstrumentedModuleFromAddr**(rword *addr*)

bool **instrumentAllExecutableMaps**()

void **removeInstrumentedRange**(rword *start*, rword *end*)

bool **removeInstrumentedModule**(**const** std::string &*name*)

bool **removeInstrumentedModuleFromAddr**(rword *addr*)

void **removeAllInstrumentedRanges**()

bool **run**(rword *start*, rword *stop*)

bool **call**(rword **retval*, rword *function*, **const** std::vector<rword> &*args* = {})

bool **callA**(rword **retval*, rword *function*, uint32_t *argNum*, **const** rword **args*)

bool **callV**(rword **retval*, rword *function*, uint32_t *argNum*, va_list *ap*)

uint32_t **addInstrRule**(*InstrRule* *rule*)

uint32_t **addMnemonicCB**(**const** char **mnemonic*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

uint32_t **addCodeCB**(*InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

uint32_t **addCodeAddrCB**(rword *address*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

uint32_t **addCodeRangeCB**(rword *start*, rword *end*, *InstPosition* *pos*, *InstCallback* *cbk*, void **data*)

uint32_t **addMemAccessCB**(*MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)

uint32_t **addMemAddrCB**(rword *address*, *MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)

uint32_t **addMemRangeCB**(rword *start*, rword *end*, *MemoryAccessType* *type*, *InstCallback* *cbk*, void **data*)

uint32_t **addVMEventCB**(*VMEvent* *mask*, *VMCallback* *cbk*, void **data*)

bool **deleteInstrumentation**(uint32_t *id*)

void **deleteAllInstrumentations**()

const *InstAnalysis* ***getInstAnalysis**(*AnalysisType* *type* = ANALYSIS_INSTRUCTION | ANALYSIS_DISASSEMBLY)

bool **recordMemoryAccess**(*MemoryAccessType* *type*)

std::vector<*MemoryAccess*> **getInstMemoryAccess**() **const**

std::vector<*MemoryAccess*> **getBBMemoryAccess**() **const**

bool **precacheBasicBlock**(rword *pc*)

void **clearCache**(rword *start*, rword *end*)

void **clearAllCache**()

2.1.4 QBDIPreload

The examples from the API sections demonstrated a use case where the instrumented code, QBDI and the instrumentation are compiled in the same binary. This, however, only works for a very limited amount of real-world scenarios. Other scenarios require some injection tool that can load QBDI and your instrumentation code in another binary.

QBDIPreload is a small utility library that provides code injection capabilities using dynamic library injection. It currently only works under **linux** using the `LD_PRELOAD` mechanism and **macOS** using the `DYLD_INSERT_LIBRARIES` mechanism. For other platforms please look into using *Frida/QBDI* instead.

QBDIPreload exploits these library injection mechanisms to hijack the normal program startup. During the hijacking process QBDIPreload will callback your code allowing you to setup and start your instrumentation. The compilation should produce a dynamic library (`.so` under **linux**, `.dylib` under **macOS**) which should then be added to the matching environment variable (`LD_PRELOAD` under **linux** and `DYLD_INSERT_LIBRARIES` under **macOS**) when running the target binary.

You can look at the *QBDIPreload Template* for a working example with build and usage instructions.

Note: QBDIPreload automatically takes care of blacklisting instrumentation of the C standard library and the OS loader as described in *Limitations*.

Note: Please note that QBDIPreload does not allow to instrument a binary before the main function (inside the loader and the library constructors / init) as explained in *Limitations*.

QBDIPreload API

Initialization

Initialization is performed using a constructor:

```
#include <QBDIPreload.h>

QBDIPRELOAD_INIT;
```

QBDIPRELOAD_INIT

A C pre-processor macro declaring a constructor.

Warning `QBDIPRELOAD_INIT` must be used once in any project using QBDIPreload. It declares a constructor, so it must be placed like a function declaration on a single line.

Return Codes

QBDIPRELOAD_NO_ERROR
No error.

QBDIPRELOAD_NOT_HANDLED
Startup step not handled by callback.

QBDIPRELOAD_ERR_STARTUP_FAILED
Error in the startup (preload) process.

User callbacks

Each of these functions must be declared and will be called (in the same order than in this documentation) by QBDIPreload during the hijacking process.

int `QBDI::`**qbdipreload_on_start**(void **main*)
Function called when preload is on a program entry point (interposed start or an early constructor). It provides the main function address, that can be used to place a hook using the `qbdipreload_hook_main` API.

Return int QBDIPreload state

Parameters

- `main`: Address of the main function

int `QBDI::`**qbdipreload_on_premain**(void **gprCtx*, void **fpuCtx*)
Function called when preload hook on main function is triggered. It provides original (and platforms dependent) GPR and FPR contexts. They can be converted to QBDI states, using `qbdipreload_threadCtxToGPRState` and `qbdipreload_floatCtxToFPRState` APIs.

Return int QBDIPreload state

Parameters

- `gprCtx`: Original GPR context
- `fpuCtx`: Original FPU context

int `QBDI::`**qbdipreload_on_main**(int *argc*, char ***argv*)
Function called when preload has successfully hijacked the main thread and we are in place of the original main function (with the same thread state).

Return int QBDIPreload state

Parameters

- `argc`: Original argc
- `argv`: Original argv

int `QBDI::`**qbdipreload_on_run**(VMInstanceRef *vm*, rword *start*, rword *stop*)
Function called when preload is done and we have a valid QBDI VM object on which we can call run (after some last initializations).

Return int QBDIPreload state

Parameters

- `vm`: VM instance.
- `start`: Start address of the range (included).
- `stop`: End address of the range (excluded).

int QBDI::**qbdipreload_on_exit**(int *status*)
Function called when process is exiting (using _exit or exit).

Return int QBDIPreload state

Parameters

- status: exit status

Helpers

qbdipreload_hook_main() can be used to hook any address as *main* during the hijacking process.

int QBDI::**qbdipreload_hook_main**(void **main*)
Enable QBDIPreload hook on the main function (using its address)

Warning It MUST be used in qbdipreload_on_start if you want to handle this step. The assumed main address is provided as a callback argument.

Parameters

- main: Pointer to the main function

Contexts related helpers allow to convert a platform dependent GPR or FPR state structure to a QBDI structure. Under **linux** both functions should receive a ucontext_t* and under **macOS** they should receive a x86_thread_state64_t* or a x86_float_state64_t*. Please look into QBDIPreload source code for more information.

void QBDI::**qbdipreload_threadCtxToGPRState**(**const** void **gprCtx*, GPRState **gprState*)
Convert a QBDIPreload GPR context (platform dependent) to a QBDI GPR state.

Parameters

- gprCtx: Platform GPRState pointer
- gprState: QBDI GPRState pointer

void QBDI::**qbdipreload_floatCtxToFPRState**(**const** void **fprCtx*, FPRState **fprState*)
Convert a QBDIPreload FPR context (platform dependent) to a QBDI FPR state.

Parameters

- fprCtx: Platform FPRState pointer
- fprState: QBDI FPRState pointer

QBDIPreload Template

To get started with QBDIPreload you can follow those few simple steps:

```
$ mkdir QBDIPreload && cd QBDIPreload
$ qbdi-preload-template
$ mkdir build && cd build
$ cmake ..
$ make
```

This will simply build the default QBDIPreload template (which prints instruction address and disassembly) and can be executed doing the following under **linux**:

```
$ LD_PRELOAD=./libqbdi_tracer.so /bin/ls
```

Or the following under **macOS**:

```
$ cp /bin/ls ./ls
$ sudo DYLD_INSERT_LIBRARIES=./libqbdi_tracer.so ./ls
```

Note: Please note that, under **macOS**, the *System Integrity Protection* (SIP) will prevent you from instrumenting system binaries. You must either use a local copy of the binary or disable SIP.

2.1.5 PyQBDI

PyQBDI is a set of bindings of QBDI for python. You can use them like any python library or like a preloaded library (with **LD_PRELOAD** or **DYLD_INSERT_LIBRARIES**).

PyQBDI offers you a way to script your instrumentation, allowing a fast and easy way to design tools. It also lets you interact with python environment, so you can for example, post process any data you collected during the instrumentation.

PyQBDI is available on Pypi and can be installed with the following command:

```
pip install --user PyQBDI
```

Note: If you use PyQBDI like a preloaded library, the library has the same limitations as QBDIPreload which are described in *QBDIPreload*. Additionnaly, the shared library libpython3.x.so must be installed (it is not included within the python package in some linux distribution).

Note: It is not possible to instrument a python process using PyQBDI because there will be a conflict between the **host** and the **guest** both trying to use the python runtime as described in *Limitations*. We would recommend directly using *QBDIPreload*.

Note: Only python3 is supported. If you want to use python2, please use QBDI 0.7.0.

Note: A version 32 bits of python is needed for PyQBDI in x86.

We provide examples along with the API documentation in the following sections.

PyQBDI Usage

Limitations

pyQBDI has some limitations that need to be considered before using those bindings :

- The library can handle only one VM at time.
- The VM must not be used in atexit module.

PyQBDI as a preloaded library

PyQBDI can be used as a preloaded library to instrument a whole binary, as QBDIPreload.

PyQBDI provide the helper script `pyqbdipreload` that preloads the library in the new process. `pyqbdipreload` is currently supported on Linux and macOS.

```
$ python3 -m pyqbdipreload ./example.py cat anyfile
```

The following actions are performed with the API of QBDIPreload :

- *QBDI::qbdipreload_on_main()*: The python script is loaded. The program arguments can be found in `sys.argv`.
- *QBDI::qbdipreload_on_run()*: The method `pyqbdipreload_on_run` is called. This method must be implemented in the specified script.
- *QBDI::qbdipreload_on_exit()*: The module `atexit` is triggered once.

Example

Here is a simple example, where we define a callback set on all instructions that displays their address and disassembly.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import pyqbdi

def mycb(vm, gpr, fpr, data):
    inst = vm.getInstAnalysis()
    print("0x{:x}: {}".format(inst.address, inst.disassembly))
    return pyqbdi.CONTINUE

def pyqbdipreload_on_run(vm, start, stop):
    vm.addCodeCB(pyqbdi.PREINST, mycb, None)
    vm.run(start, stop)
```

PyQBDI as a python library

When PyQBDI is used as a python library, the target code can be loaded in the process with `ctypes` python module.

Example

This example defines multiple callbacks, based on events, instructions, and mnemonics. Those are instrumenting the **sin()** native function and its result is being compared with the python result.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import math
import ctypes
import pyqbdi
import struct
```

```
def vmCB(vm, evt, gpr, fpr, data):
    if evt.event & pyqbdi.BASIC_BLOCK_ENTRY:
        print("[*] Basic Block: 0x{:x} -> 0x{:x}".format(evt.basicBlockStart,
                                                         evt.basicBlockEnd))
    elif evt.event & pyqbdi.BASIC_BLOCK_EXIT:
        for acs in vm.getBBMemoryAccess():
            print("@ {:#x} {:#x}:{:#x}".format(acs.instAddress,
                                               acs.accessAddress, acs.value))
    return pyqbdi.CONTINUE


def insnCB(vm, gpr, fpr, data):
    data['insn'] += 1
    types = pyqbdi.ANALYSIS_INSTRUCTION | pyqbdi.ANALYSIS_DISASSEMBLY
    types |= pyqbdi.ANALYSIS_OPERANDS | pyqbdi.ANALYSIS_SYMBOL
    inst = vm.getInstAnalysis(types)
    print("{};0x{:x}: {}".format(inst.module, inst.address, inst.disassembly))
    for op in inst.operands:
        if op.type == pyqbdi.OPERAND_IMM:
            print("const: {:d}".format(op.value))
        elif op.type == pyqbdi.OPERAND_GPR:
            print("reg: {:s}".format(op.regName))
    return pyqbdi.CONTINUE


def cmpCB(vm, gpr, fpr, data):
    data['cmp'] += 1
    return pyqbdi.CONTINUE


def run():
    # get sin function ptr
    if sys.platform == 'darwin':
        libmname = 'libSystem.dylib'
    elif sys.platform == 'win32':
        libmname = 'api-ms-win-crt-math-l1-1-0.dll'
    else:
        libmname = 'libm.so.6'
    libm = ctypes.cdll.LoadLibrary(libmname)
    funcPtr = ctypes.cast(libm.sin, ctypes.c_void_p).value

    # init VM
    vm = pyqbdi.VM()

    # create stack
    state = vm.getGPRState()
    addr = pyqbdi.allocateVirtualStack(state, 0x100000)
    assert addr is not None

    # instrument library and register memory access
    vm.addInstrumentedModuleFromAddr(funcPtr)
    vm.recordMemoryAccess(pyqbdi.MEMORY_READ_WRITE)

    # add callbacks on instructions
    udata = {"insn": 0, "cmp": 0}
    vm.addCodeCB(pyqbdi.PREINST, insnCB, udata)
```

```
    vm.addMnemonicCB("CMP*", pyqbdi.PREINST, cmpCB, udata)
    vm.addVMEventCB(pyqbdi.BASIC_BLOCK_ENTRY | pyqbdi.BASIC_BLOCK_EXIT,
                    vmCB, None)

    # Cast double arg to long and set FPR
    arg = 1.0
    carg = struct.pack('<d', arg)
    fpr = vm.getFPRState()
    fpr.xmm0 = carg

    # call sin(1.0)
    pyqbdi.simulateCall(state, 0x42424242)
    success = vm.run(funcPtr, 0x42424242)
    print(udata)

    # Retrieve output FPR state
    fpr = vm.getFPRState()
    # Cast long arg to double
    res = struct.unpack('<d', fpr.xmm0[:8])[0]
    print("%f (python) vs %f (qbdi)" % (math.sin(arg), res))

    # cleanup
    pyqbdi.alignedFree(addr)

if __name__ == "__main__":
    run()
```

PyQBDI Bindings API

The PyQBDI API is almost the same as the *C++ API*.

When the library is loaded as a preloaded library, the variable `pyqbdi.__preload__` is set to `True`.

State

The state available depends of the architecture (X86 or X86-64).

class pyqbdi.**GPRState**

AVAILABLE_GPR
shadow of rbp

NUM_GPR
shadow of eflags

REG_BP
shadow of rbp

REG_LR
not available on X86_64

REG_PC
shadow of rip

REG_RETURN
shadow of rax

REG_SP
shadow of rsp

__getitem__(*self: pyqbdi.GPRState*, *index: int*) → int
Get a register like QBDI_GPR_GET

__setitem__(*self: pyqbdi.GPRState*, *index: int*, *value: int*) → int
Set a register like QBDI_GPR_SET

eflags

r10

r11

r12

r13

r14

r15

r8

r9

rax

rbp

rbx

rcx

rdi

rdx

rip

rsi

rsp

class pyqbdi.**FPRState**

cs
x87 FPU Instruction Pointer Selector

dp
x87 FPU Instruction Operand(Data) Pointer offset

ds
x87 FPU Instruction Operand(Data) Pointer Selector

fcw
x87 FPU control word

fop
x87 FPU Opcode

fsw
x87 FPU status word

ftw
x87 FPU tag word

ip
x87 FPU Instruction Pointer offset

mxcsr
MXCSR Register state

mxcsrmask
MXCSR mask

rfcw
x87 FPU control word

rfsw
x87 FPU status word

stmm0
ST0/MM0

stmm1
ST1/MM1

stmm2
ST2/MM2

stmm3
ST3/MM3

stmm4
ST4/MM4

stmm5
ST5/MM5

stmm6
ST6/MM6

stmm7
ST7/MM7

xmm0
XMM 0

xmm1
XMM 1

xmm10
XMM 10

xmm11
XMM 11

xmm12
XMM 12

xmm13
XMM 13

xmm14
XMM 14

xmm15
XMM 15

xmm2
XMM 2

xmm3
XMM 3

xmm4
XMM 4

xmm5
XMM 5

xmm6
XMM 6

xmm7
XMM 7

xmm8
XMM 8

xmm9
XMM 9

ymm0
YMM0[255:128]

ymm1
YMM1[255:128]

ymm10
YMM10[255:128]

ymm11
YMM11[255:128]

ymm12
YMM12[255:128]

ymm13
YMM13[255:128]

ymm14
YMM14[255:128]

ymm15
YMM15[255:128]

ymm2
YMM2[255:128]

ymm3
YMM3[255:128]

ymm4
YMM4[255:128]

ymm5
YMM5[255:128]

ymm6
YMM6[255:128]

ymm7
YMM7[255:128]

ymm8
YMM8[255:128]

ymm9
YMM9[255:128]

VM

class pyqbdi.**VM**

addCodeAddrCB(*self: pyqbdi.VM, address: int, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object*) → object
Register a callback for when a specific address is executed.

addCodeCB(*self: pyqbdi.VM, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object*) → object
Register a callback event for every instruction executed.

addCodeRangeCB(*self: pyqbdi.VM, start: int, end: int, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object*) → object
Register a callback for when a specific address range is executed.

addInstrumentedModule(*self: pyqbdi.VM*, *name: str*) → bool
Add the executable address ranges of a module to the set of instrumented address ranges.

addInstrumentedModuleFromAddr(*self: pyqbdi.VM*, *addr: int*) → bool
Add the executable address ranges of a module to the set of instrumented address ranges using an address belonging to the module.

addInstrumentedRange(*self: pyqbdi.VM*, *start: int*, *end: int*) → None
Add an address range to the set of instrumented address ranges.

addMemAccessCB(*self: pyqbdi.VM, type: pyqbdi.MemoryAccessType, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object*) → object
Register a callback event for every memory access matching the type bitfield made by the instructions.

addMemAddrCB(*self: pyqbdi.VM, address: int, type: pyqbdi.MemoryAccessType, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object*) → object
Add a virtual callback which is triggered for any memory access at a specific address matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

addMemRangeCB(*self: pyqbdi.VM, start: int, end: int, type: pyqbdi.MemoryAccessType, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object*) → object
Add a virtual callback which is triggered for any memory access at a specific address range matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

addMnemonicCB(*self: pyqbdi.VM, mnemonic: str, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object*) → object
Register a callback event if the instruction matches the mnemonic.

addVMEventCB(**args*, ***kwargs*)
Overloaded function.

1. addVMEventCB(self: pyqbdi.VM, mask: pyqbdi.VMEvent, cbk: Callable[[pyqbdi.VM, pyqbdi.VMState, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object) -> object

Register a callback event for a specific VM event.

2. addVMEventCB(self: pyqbdi.VM, mask: int, cbk: Callable[[pyqbdi.VM, pyqbdi.VMState, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object) -> object

Register a callback event for a specific VM event.

call(*self: pyqbdi.VM, function: int, args: List[int]*) → Tuple[bool, int]
Call a function using the DBI (and its current state).

clearAllCache(*self: pyqbdi.VM*) → None
Clear the entire translation cache.

clearCache(*self: pyqbdi.VM*, *start: int*, *end: int*) → None
Clear a specific address range from the translation cache.

deleteAllInstrumentations(*self: pyqbdi.VM*) → None
Remove all the registered instrumentations.

deleteInstrumentation(*self: pyqbdi.VM*, *id: int*) → None
Remove an instrumentation.

getBBMemoryAccess(*self: pyqbdi.VM*) → List[pyqbdi.MemoryAccess]
Obtain the memory accesses made by the last executed basic block.

getFPRState(*self: pyqbdi.VM*) → pyqbdi.FPRState
Obtain the current floating point register state.

getGPRState(*self: pyqbdi.VM*) → pyqbdi.GPRState
Obtain the current general purpose register state.

getInstAnalysis(**args*, ***kwargs*)
Overloaded function.

1. getInstAnalysis(self: pyqbdi.VM, type: pyqbdi.AnalysisType = AnalysisType.ANALYSIS_INSTRUCTION|AnalysisType.ANALYSIS_DISASSEMBLY) -> pyqbdi.InstAnalysis

Obtain the analysis of an instruction metadata. Analysis results are cached in the VM.

2. getInstAnalysis(self: pyqbdi.VM, type: int) -> pyqbdi.InstAnalysis

Obtain the analysis of an instruction metadata. Analysis results are cached in the VM.

getInstMemoryAccess(*self: pyqbdi.VM*) → List[pyqbdi.MemoryAccess]
Obtain the memory accesses made by the last executed instruction.

instrumentAllExecutableMaps(*self: pyqbdi.VM*) → bool
Adds all the executable memory maps to the instrumented range set.

precacheBasicBlock(*self: pyqbdi.VM*, *pc: int*) → bool
Pre-cache a known basic block

recordMemoryAccess (*self: pyqbdi.VM*, *type: pyqbdi.MemoryAccessType*) → bool
Add instrumentation rules to log memory access using inline instrumentation and instruction shadows.

removeAllInstrumentedRanges (*self: pyqbdi.VM*) → None
Remove all instrumented ranges.

removeInstrumentedModule (*self: pyqbdi.VM*, *name: str*) → bool
Remove the executable address ranges of a module from the set of instrumented address ranges.

removeInstrumentedModuleFromAddr (*self: pyqbdi.VM*, *addr: int*) → bool
Remove the executable address ranges of a module from the set of instrumented address ranges using an address belonging to the module.

removeInstrumentedRange (*self: pyqbdi.VM*, *start: int*, *end: int*) → None
Remove an address range from the set of instrumented address ranges.

run (*self: pyqbdi.VM*, *start: int*, *stop: int*) → bool
Start the execution by the DBI.

setFPRState (*self: pyqbdi.VM*, *fprState: pyqbdi.FPRState*) → None
Set the FPR state.

setGPRState (*self: pyqbdi.VM*, *gprState: pyqbdi.GPRState*) → None
Set the GPR state.

Callback

pyqbdi.**InstPosition = <class 'pyqbdi.InstPosition'>**
Position relative to an instruction.

Members:

PREINST : Positioned before the instruction.

POSTINST : Positioned after the instruction.

class pyqbdi.**MemoryAccess**

accessAddress
Address of accessed memory

instAddress
Address of instruction making the access

size
Size of memory access (in bytes)

type
Memory access type (READ / WRITE)

value
Value read from / written to memory

pyqbdi.**MemoryAccessType = <class 'pyqbdi.MemoryAccessType'>**
Memory access type (read / write / . . .)

Members:

MEMORY_READ : Memory read access

MEMORY_WRITE : Memory write access

MEMORY_READ_WRITE : Memory read/write access

pyqbdi.**VMAction = <class 'pyqbdi.VMAction'>**
The callback results.

Members:

CONTINUE : The execution of the basic block continues.

BREAK_TO_VM : The execution breaks and returns to the VM causing a complete reevaluation of the execution state. A BREAK_TO_VM is needed to ensure that modifications of the Program Counter or the program code are taken into account.

STOP : Stops the execution of the program. This causes the run function to return early.

pyqbdi.**VMEvent = <class 'pyqbdi.VMEvent'>**
Members:

SEQUENCE_ENTRY : Triggered when the execution enters a sequence.

SEQUENCE_EXIT : Triggered when the execution exits from the current sequence.

BASIC_BLOCK_ENTRY : Triggered when the execution enters a basic block.

BASIC_BLOCK_EXIT : Triggered when the execution exits from the current basic block.

BASIC_BLOCK_NEW : Triggered when the execution enters a new (~unknown) basic block.

EXEC_TRANSFER_CALL : Triggered when the ExecBroker executes an execution transfer.

EXEC_TRANSFER_RETURN : Triggered when the ExecBroker returns from an execution transfer.

class pyqbdi.**VMState**

basicBlockEnd
The current basic block end address which can also be the execution transfer destination.

basicBlockStart
The current basic block start address which can also be the execution transfer destination.

event
The event(s) which triggered the callback (must be checked using a mask: event & BASIC_BLOCK_ENTRY).

sequenceEnd
The current sequence end address which can also be the execution transfer destination.

sequenceStart
The current sequence start address which can also be the execution transfer destination.

InstAnalysis

pyqbdi.**AnalysisType = <class 'pyqbdi.AnalysisType'>**
Instruction analysis type

Members:

ANALYSIS_INSTRUCTION : Instruction analysis (address, mnemonic, ...)

ANALYSIS_DISASSEMBLY : Instruction disassembly

ANALYSIS_OPERANDS : Instruction operands analysis

ANALYSIS_SYMBOL : Instruction symbol

class pyqbdi.**InstAnalysis**

address
Instruction address (if ANALYSIS_INSTRUCTION)

affectControlFlow
True if instruction affects control flow (if ANALYSIS_INSTRUCTION)

disassembly
Instruction disassembly (if ANALYSIS_DISASSEMBLY)

instSize
Instruction size (in bytes) (if ANALYSIS_INSTRUCTION)

isBranch
True if instruction acts like a 'jump' (if ANALYSIS_INSTRUCTION)

isCall
True if instruction acts like a 'call' (if ANALYSIS_INSTRUCTION)

isCompare
True if instruction is a comparison (if ANALYSIS_INSTRUCTION)

isPredicable
True if instruction contains a predicate (~is conditional) (if ANALYSIS_INSTRUCTION)

isReturn
True if instruction acts like a 'return' (if ANALYSIS_INSTRUCTION)

mayLoad
True if instruction 'may' load data from memory (if ANALYSIS_INSTRUCTION)

mayStore
True if instruction 'may' store data to memory (if ANALYSIS_INSTRUCTION)

mnemonic
LLVM mnemonic (if ANALYSIS_INSTRUCTION)

module
Instruction module name (if ANALYSIS_SYMBOL and found)

numOperands
Number of operands used by the instruction (if ANALYSIS_OPERANDS)

operands
Structure containing analysis results of an operand provided by the VM (if ANALYSIS_OPERANDS)

symbol
Instruction symbol (if ANALYSIS_SYMBOL and found)

symbolOffset
Instruction symbol offset (if ANALYSIS_SYMBOL)

class pyqbdi.**OperandAnalysis**

flag
Operand flag

regAccess
Register access type (r, w, rw)

regCtxIdx
Register index in VM state

regName
Register name

regOff
Sub-register offset in register (in bits)

size
Operand size (in bytes)

type
Operand type

value
Operand value (if immediate), or register Id

pyqbdi.**RegisterAccessType = <class 'pyqbdi.RegisterAccessType'>**
Access type (R/W/RW) of a register operand

Members:

REGISTER_UNUSED : Unused register

REGISTER_READ : Register read access

REGISTER_WRITE : Register write access

REGISTER_READ_WRITE : Register read/write access

pyqbdi.**OperandType = <class 'pyqbdi.OperandType'>**
Operand type

Members:

OPERAND_INVALID : Invalid operand

OPERAND_IMM : Immediate operand

OPERAND_GPR : Register operand

OPERAND_PRED : Predicate operand

pyqbdi.**OperandFlag = <class 'pyqbdi.OperandFlag'>**
Operand flag

Members:

OPERANDFLAG_NONE : No flag

OPERANDFLAG_ADDR : The operand is used to compute an address

OPERANDFLAG_PCREL : The value of the operand is PC relative

OPERANDFLAG_UNDEFINED_EFFECT : The operand role isn't fully defined

Memory and process map

pyqbdi.**getModuleNames**() → List[str]
Get a list of all the module names loaded in the process memory.

pyqbdi.**getCurrentProcessMaps**(*full_path: bool = False*) → List[pyqbdi.MemoryMap]
Get a list of all the memory maps (regions) of the current process.

pyqbdi.**getRemoteProcessMaps**(*pid: int*, *full_path: bool = False*) → List[pyqbdi.MemoryMap]
Get a list of all the memory maps (regions) of a process.

pyqbdi.**Permission = <class 'pyqbdi.Permission'>**
Memory access rights.

Members:

PF_NONE : No access

PF_READ : Read access

PF_WRITE : Write access

PF_EXEC : Execution access

class pyqbdi.**MemoryMap**

name
Region name (useful when a region is mapping a module).

permission
Region access rights (PF_READ, PF_WRITE, PF_EXEC).

range
A range of memory (region), delimited between a start and an (excluded) end address.

pyqbdi.**alignedAlloc**(*size: int*, *align: int*) → int
Allocate a block of memory of a specified sized with an aligned base address.

pyqbdi.**alignedFree**(*ptr: int*) → None
Free a block of aligned memory allocated with alignedAlloc.

pyqbdi.**allocateVirtualStack**(*gprstate: pyqbdi.GPRState*, *size: int*) → object
Allocate a new stack and setup the GPRState accordingly. The allocated stack needs to be freed with alignedFree(). The result was int, or None if the allocation fails.

pyqbdi.**simulateCall**(*ctx: pyqbdi.GPRState*, *returnAddress: int*, *args: List[int] = []*) → None
Simulate a call by modifying the stack and registers accordingly.

Range

class pyqbdi.**Range**

contains(**args*, ***kwargs*)
Overloaded function.

1. contains(self: pyqbdi.Range, t: int) -> bool

Return True if an value is inside current range boundaries.

2. contains(self: pyqbdi.Range, r: pyqbdi.Range) -> bool

Return True if a range is inside current range boundaries.

end
Range end value (always excluded).

intersect(*self: pyqbdi.Range*, *r: pyqbdi.Range*) → pyqbdi.Range
Return the intersection of two ranges.

overlaps (*self: pyqbdi.Range*, *r: pyqbdi.Range*) → bool
Return True if a range is overlapping current range lower or/and upper boundary.

size (*self: pyqbdi.Range*) → int
Return the total length of a range.

start
Range start value.

Miscellaneous

pyqbdi.**__arch__**

pyqbdi.**__platform__**

pyqbdi.**__preload__**
Library load with pyqbdipreload

pyqbdi.**__os__**

pyqbdi.**__version__**
Version of QBDI

PyQBDI Bindings Helpers

In addition of C++ API, some helper are available:

Memory helpers

pyqbdi.**readMemory** (*address: int*, *size: int*) → bytes
Read a content from a base address.

Parameters

- **address** – Base address
- **size** – Read size

Returns Bytes of content.

> **Warning:** This API is hazardous as the whole process memory can be read.

pyqbdi.**readRword** (*address: int*) → int
Read a rword to the specified address

Parameters **address** – Base address

Returns the value as a unsigned integer

> **Warning:** This API is hazardous as the whole process memory can be read.

pyqbdi.**writeMemory** (*address: int*, *bytes: str*) → None
Write a memory content to a base address.

Parameters

- **address** – Base address
- **bytes** – Memory content

Warning: This API is hazardous as the whole process memory can be written.

pyqbdi.**writeRword**(*address: int*, *value: int*) → None
Write a rword in a base address.

Parameters

- **address** – Base address
- **value** – The value to write, as a unsigned integer

Warning: This API is hazardous as the whole process memory can be written.

pyqbdi.**allocateRword**() → int
Allocate a raw memory space to store a rword.

Returns Address to a memory space to store a rword

pyqbdi.**allocateMemory**(*length: int*) → int
Allocate a raw memory space of specified length.

Parameters **length** – length of the memory space to allocate

Returns Address to the allocated memory

pyqbdi.**freeMemory**(*address: int*) → None
Free a memory space allocate with allocateRword or allocateMemory.

Parameters **address** – Address of the allocated memory

Float helpers

pyqbdi.**encodeFloat**(*val: float*) → int
Encode a float as a signed integer.

Parameters **val** – Float value

Returns a sigend integer

pyqbdi.**decodeFloat**(*val: int*) → float
Encode a sigend integer as a float.

Parameters **val** – signed integer value

Returns a float

pyqbdi.**encodeFloatU**(*val: float*) → int
Encode a float as an unsigned interger.

Parameters **val** – Float value

Returns an unsigned integer

pyqbdi.**decodeFloatU**(*val: int*) → float
Encode an unsigend integer as a float.

Parameters **val** – unsigned integer value

Returns a float

pyqbdi.**encodeDouble**(*val: float*) → int
Encode a double as a signed integer.

Parameters **val** – Double value

Returns a sigend integer

pyqbdi.**decodeDouble**(*val: int*) → float
Encode a sigend integer as a double.

Parameters **val** – signed integer value

Returns a double

pyqbdi.**encodeDoubleU**(*val: float*) → int
Encode a double as an unsigned interger.

Parameters **val** – Double value

Returns an unsigned integer

pyqbdi.**decodeDoubleU**(*val: int*) → float
Encode an unsigend integer as a double.

Parameters **val** – unsigned integer value

Returns a double

For more conversion, you may want to use the struct library of Python.

2.1.6 Frida/QBDI

Frida/QBDI is a full-featured support of *QBDI* inside *Frida*. It can be used in ways similar to *Frida Stalker*, while leveraging all the power of *QBDI*.

Integration aims to be as tight as possible with Frida spirit, with instrumentation callbacks that can be written in JavaScript, calls performed directly on *Frida NativeFunction* objects, bindings imported as *nodejs* scripts (*frida-compile* support), etc.

Introduction

Why Frida?

QBDI is quite easy to use when it comes to embedding it inside a binary at compile time (see `qbdi-template`), but we had no easy way to instrument an already compiled binary. Even if multiple ways can be used to inject QBDI in a process (from a simple LD_PRELOAD to a tailored injector), we can also rely on Frida core to inject our framework into an existing process and manipulate QBDI from Frida interface. Frida already supports various platforms (iOS, Android, Windows, macOS) making it a good choice for any cross platform tool. We developed `frida-qbdi.js` that makes all QBDI bindings available through Frida.

How to use Frida / QBDI?

To use QBDI on an already existing process you can use the following syntax:

```
frida -n processName -l frida-qbdi.js
```

You can also spawn the process using Frida to instrument it with QBDI as soon as it starts:

```
frida -f binaryPath Arguments -l frida-qbdi.js
```

If you want to get started using QBDI bindings, you can create a new default project doing:

```
make NewProject
cd NewProject
qbdi-frida-template
# If you want to build the demo binary
mkdir build && cd build
cmake ..
make
# If frida-compile is not installed
npm install frida-compile babelify
./node_modules/.bin/frida-compile ../FridaQBDI_sample.js -o RunMe.js
# else
frida-compile ../FridaQBDI_sample.js -o RunMe.js
frida ./demo.bin -l ./RunMe.js
```

You can find all the information about QBDI bindings in the next section along with a writeup that demonstrates what can be achieved using QBDI with Frida injection.

Get Started with Frida/QBDI

To be able to use QBDI bindings while injecting into a process, it is necessary to understand a bit of Frida to perform some common tasks. Through this simple example based on *qbdi-frida-template* we will explain a basic usage of Frida & QBDI.

Common tasks

Most actions described here are listed in the Frida documentation, this is mostly a reminder for those used to interact with Frida.

Read Memory

Sometimes it may be necessary to have a look at a buffer or specific part of the memory. We rely on Frida to do it.

```
var arrayPtr = ptr(0xDEADBEEF)
var size = 0x80
var buffer = Memory.readByteArray(arrayPtr, size)
```

Write Memory

We also need to be able to write memory:

```
var arrayPtr = ptr(0xDEADBEEF)
var size = 0x80
var toWrite = new Uint8Array(size);
```

```
// Fill your buffer eventually
Memory.writeByteArray(arrayPtr, toWrite)
```

Allocate an array

If you have a function that takes a buffer or a string as an input, you might need to allocate a new buffer using Frida:

```
// allocate and write a 2 bytes buffer
var buffer = Memory.alloc(2);
Memory.writeByteArray(buffer, [0x42, 0x42])
// allocate and write an UTF8 string
var str = Memory.allocUtf8String("Hello World !");
```

Initialize a QBDI object

If *frida-qbdi.js* (or a script requiring it) is successfully loaded in Frida, a new QBDI object become available. It provides an object oriented access to the framework features.

```
// Initialize QBDI
var vm = new QBDI();
console.log("QBDI version is " + vm.version.string);
var state = vm.getGPRState();
```

Instrument a function with QBDI

You can instrument a function using QBDI bindings. They are really close to the C++ ones, with more information is available in Frida/QBDI *API bindings* documentation.

```
var functionPtr = DebugSymbol.fromName("function_name").address;
vm.addInstrumentedModule("demo.bin");

var InstructionCallback = vm.newInstCallback(function(vm, gpr, fpr, data) {
    inst = vm.getInstAnalysis();
    gpr.dump(); // Display context
    console.log("0x" + inst.address.toString(16) + " " + inst.disassembly); //␣
↪Display instruction
    return VMAction.CONTINUE;
});
var iid = vm.addCodeCB(InstPosition.PREINST, instructionCallback, NULL);

vm.call(functionPtr, []);
```

If you ever want to pass argument to your callback, this can be done via the **data** argument :

```
// This callback is used to count the number of basicblocks executed
var userData = { counter: 0};
var BasicBlockCallback = vm.newVMCallback(function(vm, evt, gpr, fpr, data) {
    data.counter++;
    return VMAction.CONTINUE;
});
vm.addVMEventCB(VMEvent.BASIC_BLOCK_ENTRY, BasicBlockCallback, userData);
console.log(userData.counter);
```

Scripts

Bindings can simply be used in Frida REPL, or imported in a Frida script, empowering the bindings with all the *nodejs* ecosystem.

```
const qbdi = require('/usr/local/share/qbdi/frida-qbdi'); // import QBDI bindings
qbdi.import(); // Set bindings to global environment

var vm = new QBDI();
console.log("QBDI version is " + vm.version.string);
```

This simple script can be compiled with *frida-compile* utility (see *Frida* documentation). It will be possible to load it in Frida in place of *frida-qbdi.js*, allowing to easily create custom instrumentation tools with in-process scripts written in JavaScript and external control in Python (or any language supported by *Frida*).

Complete example

If you already had a look at the default instrumentation of the template generated with *qbdi-frida-template* you will be familiar with the following example. What it does is creating a native call to the *Secret()* function, and instrument it looking for *XOR*.

Source code

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#if defined(_MSC_VER)
# define EXPORT __declspec(dllexport)
#else  // _MSC_VER
# define EXPORT __attribute__ ((visibility ("default")))
#endif


EXPORT int Secret(char* str)
{
    int i;
    unsigned char XOR[] = {0x51,0x42,0x44,0x49,0x46,0x72,0x69,0x64,0x61};
    size_t len = strlen(str);

    printf("Input string is : %s\nEncrypted string is : \n", str);

    for (i = 0; i < len; i++) {
        printf("0x%x,", str[i]^XOR[i%sizeof(XOR)]);
    }
    printf("\n");
    fflush(stdout);
    return 0;
}

void Hello()
{
    Secret("Hello world !");
```

```
}

int main()
{
    Hello();
}
```

Instrumentation code

```
// QBDI
const qbdi = require('/usr/local/share/qbdi/frida-qbdi'); // import QBDI bindings
qbdi.import(); // Set bindings to global environment

// Initialize QBDI
var vm = new QBDI();
var state = vm.getGPRState();
var stack = vm.allocateVirtualStack(state, 0x100000);

// Instrument "Secret" function from demo.bin
var funcPtr = Module.findExportByName(null, "Secret");
if (!funcPtr) {
    funcPtr = DebugSymbol.fromName("Secret");
}
vm.addInstrumentedModuleFromAddr(funcPtr);

// Callback on every instruction
// This callback will print context and display current instruction address and␣
↪dissassembly
// We choose to print only XOR instructions
var icbk = vm.newInstCallback(function(vm, gpr, fpr, data) {
    inst = vm.getInstAnalysis();
    if (inst.mnemonic.search("XOR")){
        return VMAction.CONTINUE;
    }
    gpr.dump(); // Display context
    console.log("0x" + inst.address.toString(16) + " " + inst.disassembly); //␣
↪Display instruction dissassembly
    return VMAction.CONTINUE;
});
var iid = vm.addCodeCB(InstPosition.PREINST, icbk);

// Allocate a string in remote process memory
var strP = Memory.allocUtf8String("Hello world !");
// Call the Secret function using QBDI and with our string as argument
vm.call(funcPtr, [strP]);
```

Frida bindings

We propose bindings for Frida, the whole C/C++ API is available through them, but they are also backed up by helpers that fluidify script writing.

API bindings

Initialization

_QBDI
Create a new instrumentation virtual machine using "**new QBDI**()"

Returns QBDI virtual machine instance

Rtype object

State Management

GPRState.prototype.**getRegister**(*rid*)
This function is used to get the value of a specific register.

Arguments

- **rid** – Register (register name or ID can be used e.g : "RAX", "rax", 0)

Returns GPR value (ex: 0x42)

Return type NativePointer

GPRState.prototype.**setRegister**(*rid*, *value*)
This function is used to set the value of a specific register.

Arguments

- **rid** – Register (register name or ID can be used e.g : "RAX", "rax", 0)
- **value** – Register value (use **strings** for big integers)

GPRState.prototype.**getRegisters**()
This function is used to get values of all registers.

Returns GPRs of current context (ex: {"RAX":0x42, ...})

Return type {String:rword, ...}

GPRState.prototype.**setRegisters**(*gprs*)
This function is used to set values of all registers.

Arguments

- **gprs** – Array of register values

GPRState.prototype.**synchronizeRegister**(*FridaCtx*, *rid*, *direction*)
This function is used to synchronise a specific register between Frida and QBDI.

Arguments

- **FridaCtx** – Frida context
- **rid** – Register (register name or ID can be used e.g : "RAX", "rax", 0)
- **direction** – Synchronization direction. (*FRIDA_TO_QBDI* or *QBDI_TO_FRIDA*)

Warning: Currently QBDI_TO_FRIDA is experimental. (E.G : RIP cannot be synchronized)

GPRState.prototype.**synchronizeContext**(*FridaCtx*, *direction*)
This function is used to synchronise context between Frida and QBDI.

Arguments

- **FridaCtx** – Frida context
- **direction** – Synchronization direction. (*FRIDA_TO_QBDI* or *QBDI_TO_FRIDA*)

Warning: Currently QBDI_TO_FRIDA is not implemented (due to Frida limitations).

GPRState.prototype.**pp**([*color*])
Pretty print QBDI context.

Arguments

- **[color]** – Will print a colored version of the context if set.

Returns dump of all GPRs in a pretty format

Return type String

GPRState.prototype.**dump**([*color*])
Pretty print and log QBDI context.

Arguments

- **[color]** – Will print a colored version of the context if set.

QBDI.prototype.**getGPRState**(*state*)
Obtain the current general register state.

Returns An object containing the General Purpose Registers state.

Return type object

QBDI.prototype.**setGPRState**(*state*)
Set the GPR state

Arguments

- **state** – Array of register values

State Initialization

QBDI.prototype.**alignedAlloc**(*size*, *align*)
Allocate a block of memory of a specified sized with an aligned base address.

Arguments

- **size** – Allocation size in bytes.
- **align** – Base address alignement in bytes.

Returns Pointer to the allocated memory or NULL in case an error was encountered.

Return type rword

QBDI.prototype.**allocateVirtualStack**(*gprs*, *stackSize*)
Allocate a new stack and setup the GPRState accordingly. The allocated stack needs to be freed with alignedFree().

Arguments

- **gprs** – Array of register values
- **stackSize** – Size of the stack to be allocated.

`QBDI.prototype.`**`setFPRState`**(*state*)
Set the FPR state

Arguments

- **`state`** – Array of register values

`QBDI.prototype.`**`setGPRState`**(*state*)
Set the GPR state

Arguments

- **`state`** – Array of register values

`QBDI.prototype.`**`simulateCall`**(*state*, *retAddr*[, *args*])
Simulate a call by modifying the stack and registers accordingly.

Arguments

- **`state`** – Array of register values
- **`retAddr`** – Return address of the call to simulate.
- **`args`** – A variadic list of arguments.

Execution

`QBDI.prototype.`**`addInstrumentedModule`**(*name*)
Add the executable address ranges of a module to the set of instrumented address ranges.

Arguments

- **`name`** – The module's name.

Returns True if at least one range was added to the instrumented ranges.

Return type boolean

`QBDI.prototype.`**`addInstrumentedModuleFromAddr`**(*addr*)
Add the executable address ranges of a module to the set of instrumented address ranges. using an address belonging to the module.

Arguments

- **`addr`** – An address contained by module's range.

Returns True if at least one range was removed from the instrumented ranges.

Return type boolean

`QBDI.prototype.`**`addInstrumentedRange`**(*start*, *end*)
Add an address range to the set of instrumented address ranges.

Arguments

- **`start`** – Start address of the range (included).
- **`end`** – End address of the range (excluded).

`QBDI.prototype.`**`call`**(*address*[, *args*])
Call a function by its address (or through a Frida `NativePointer`).

Arguments

- **`address`** – function address (or Frida `NativePointer`).

- **[args]** – optional list of arguments

Arguments can be provided, but their types need to be compatible with the `.toRword()` interface (like `NativePointer` or `UInt64`).

Example:

```
>>> var vm = new QBDI();
>>> var state = vm.getGPRState();
>>> vm.allocateVirtualStack(state, 0x1000000);
>>> var aFunction = Module.findExportByName(null, "Secret");
>>> vm.addInstrumentedModuleFromAddr(aFunction);
>>> vm.call(aFunction, [42]);
```

`QBDI.prototype.`**`getModuleNames`**`()`
Use QBDI engine to retrieve loaded modules.

Returns list of module names (ex: [“ls”, “libc”, “libz”])

Return type [String]

`QBDI.prototype.`**`removeInstrumentedRange`**`(`*start*, *end*`)`
Remove an address range from the set of instrumented address ranges.

Arguments

- **start** – Start address of the range (included).
- **end** – End address of the range (excluded).

`QBDI.prototype.`**`run`**`(`*start*, *stop*`)`
Start the execution by the DBI from a given address (and stop when another is reached).

Arguments

- **start** – Address of the first instruction to execute.
- **stop** – Stop the execution when this instruction is reached.

Returns True if at least one block has been executed.

Return type boolean

Instrumentation

`QBDI.prototype.`**`addCodeAddrCB`**`(`*addr*, *pos*, *cbk*, *data*`)`
Register a callback for when a specific address is executed.

Arguments

- **addr** – Code address which will trigger the callback.
- **pos** – Relative position of the event callback (PreInst / PostInst).
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

`QBDI.prototype.`**`addCodeCB`**`(`*pos*, *cbk*, *data*`)`
Register a callback event for a specific instruction event.

Arguments

- **pos** – Relative position of the event callback (PreInst / PostInst).
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

QBDI.prototype.**addCodeRangeCB**(*start*, *end*, *pos*, *cbk*, *data*)
Register a callback for when a specific address range is executed.

Arguments

- **start** – Start of the address range which will trigger the callback.
- **end** – End of the address range which will trigger the callback.
- **pos** – Relative position of the event callback (PreInst / PostInst).
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

QBDI.prototype.**addMnemonicCB**(*mnem*, *pos*, *cbk*, *data*)
Register a callback event if the instruction matches the mnemonic.

Arguments

- **mnem** – Mnemonic to match.
- **pos** – Relative position of the event callback (PreInst / PostInst).
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

QBDI.prototype.**deleteAllInstrumentations**()
Remove all the registered instrumentations.

QBDI.prototype.**deleteInstrumentation**(*id*)
Remove an instrumentation.

Arguments

- **id** – The id of the instrumentation to remove.

Returns True if instrumentation has been removed.

Return type boolean

Memory Callback

QBDI.prototype.**addMemAccessCB**(*type*, *cbk*, *data*)

Register a callback event for every memory access matching the type bitfield made by the instruction in the range codeStart to codeEnd.

Arguments

- **type** – A mode bitfield: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

QBDI.prototype.**addMemAddrCB**(*addr*, *type*, *cbk*, *data*)

Add a virtual callback which is triggered for any memory access at a specific address matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

Arguments

- **addr** – Code address which will trigger the callback.
- **type** – A mode bitfield: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

QBDI.prototype.**addMemRangeCB**(*start*, *end*, *type*, *cbk*, *data*)

Add a virtual callback which is triggered for any memory access in a specific address range matching the access type. Virtual callbacks are called via callback forwarding by a gate callback triggered on every memory access. This incurs a high performance cost.

Arguments

- **start** – Start of the address range which will trigger the callback.
- **end** – End of the address range which will trigger the callback.
- **type** – A mode bitfield: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

QBDI.prototype.**recordMemoryAccess**(*type*)
Obtain the memory accesses made by the last executed instruction. Return NULL and a size of 0 if the instruction made no memory access.

Arguments

- **type** – Memory mode bitfield to activate the logging for: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).

Analysis

QBDI.prototype.**getBBMemoryAccess**()

Obtain the memory accesses made by the last executed basic block. Return NULL and a size of 0 if the basic block made no memory access.

Returns An array of memory accesses made by the basic block.

Return type Array

QBDI.prototype.**getInstAnalysis**()
Obtain the analysis of an instruction metadata. Analysis results are cached in the VM. The validity of the returned pointer is only guaranteed until the end of the callback, else a deepcopy of the structure is required.

Arguments

- **[type]** – Properties to retrieve during analysis (default to ANALYSIS_INSTRUCTION | ANALYSIS_DISASSEMBLY).

Returns A InstAnalysis object containing the analysis result.

Return type Object

QBDI.prototype.**getInstMemoryAccess**()
Obtain the memory accesses made by the last executed instruction. Return NULL and a size of 0 if the instruction made no memory access.

Returns An array of memory accesses made by the instruction.

Return type Array

Cache management

QBDI.prototype.**precacheBasicBlock**(*state*)
Pre-cache a known basic block.

Arguments

- **pc** – Start address of a basic block

Returns True if basic block has been inserted in cache.

Return type bool

QBDI.prototype.**precacheBasicBlock**(*state*)
Clear a specific address range from the translation cache.

Arguments

- **start** – Start of the address range to clear from the cache.

- **end** – End of the address range to clear from the cache.

QBDI.prototype.**precacheBasicBlock**(*state*)
Clear the entire translation cache.

VM Events

QBDI.prototype.**addVMEventCB**(*mask*, *cbk*, *data*)
Register a callback event for a specific VM event.

Arguments

- **mask** – A mask of VM event type which will trigger the callback.
- **cbk** – A function pointer to the callback.
- **data** – User defined data passed to the callback.

Returns The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Return type integer

VMError

static INVALID_EVENTID
Returned event is invalid.

Globals

QBDI_LIB_FULLPATH
Fullpath of the QBDI library

GPR_NAMES
An array holding register names.

REG_RETURN
A constant string representing the register carrying the return value of a function.

REG_PC
String of the instruction pointer register.

REG_SP
String of the stack pointer register.

SyncDirection

Synchronisation direction between Frida and QBDI GPR contexts

static QBDI_TO_FRIDA
Constant variable used to synchronize QBDI's context to Frida's.

> **Warning:** This is currently not supported due to the lack of context updating in Frida.

static FRIDA_TO_QBDI
Constant variable used to synchronize Frida's context to QBDI's.

VMAction

The callback results.

static CONTINUE
The execution of the basic block continues.

static BREAK_TO_VM
The execution breaks and returns to the VM causing a complete reevaluation of the execution state. A `VMAction.BREAK_TO_VM` is needed to ensure that modifications of the Program Counter or the program code are taken into account.

static STOP
Stops the execution of the program. This causes the run function to return early.

InstPosition

Position relative to an instruction.

static PREINST
Positioned **before** the instruction..

static POSTINST
Positioned **after** the instruction..

VMEvent

Events triggered by the virtual machine.

static SEQUENCE_ENTRY
Triggered when the execution enters a sequence.

static SEQUENCE_EXIT
Triggered when the execution exits from the current sequence.

static BASIC_BLOCK_ENTRY
Triggered when the execution enters a basic block.

static BASIC_BLOCK_EXIT
Triggered when the execution exits from the current basic block.

static BASIC_BLOCK_NEW
Triggered when the execution enters a new (~unknown) basic block.

static EXEC_TRANSFER_CALL
Triggered when the ExecBroker executes an execution transfer.

static EXEC_TRANSFER_RETURN
Triggered when the ExecBroker returns from an execution transfer.

static SYSCALL_ENTRY
Not implemented.

static SYSCALL_EXIT
Not implemented.

static SIGNAL
Not implemented.

MemoryAccessType

Memory access type (read / write / . . .)

static MEMORY_READ
Memory read access.

static MEMORY_WRITE
Memory write access.

static MEMORY_READ_WRITE
Memory read/write access.

AnalysisType

Properties to retrieve during an instruction analysis.

static ANALYSIS_INSTRUCTION
Instruction analysis (address, mnemonic, ...).

static ANALYSIS_DISASSEMBLY
Instruction disassembly.

static ANALYSIS_OPERANDS
Instruction operands analysis.

static ANALYSIS_SYMBOL
Instruction nearest symbol (and offset).

Register values

The size of a general register depends of the architecture. QBDI uses a custom type (*rword*) to represent a register value.

This binding provides a common interface (`.toRword()`) to cast values into JS types compatible with the C *rword* type.

rword
An alias to Frida uint type with the size of general registers (**uint64** or **uint32**)

`NativePointer.prototype.`**toRword**`()`
Convert a NativePointer into a type with the size of a register (`Number` or `UInt64`).

`Number.prototype.`**toRword**`()`
Convert a number into a type with the size of a register (`Number` or `UInt64`). Can't be used for numbers > 32 bits, would cause weird results due to IEEE-754.

`UInt64.prototype.`**toRword**`()`
An identity function (returning the same `UInt64` object). It exists only to provide a unified **toRword** interface.

Helpers

Some functions helpful to interact with Frida interface and write scripts.

hexPointer (*ptr*)
This function is used to pretty print a pointer, padded with 0 to the size of a register.

Arguments

- **ptr** – Pointer you want to pad

Returns pointer value as padded string (ex: "0x00004242")

Return type String

`QBDI.prototype.`**getModuleNames**`()`
Use QBDI engine to retrieve loaded modules.

Returns list of module names (ex: ["ls", "libc", "libz"])

Return type [String]

QBDI.prototype.**newInstCallback**(*cbk*)

Create a native **Instruction callback** from a JS function.

Arguments

- **cbk** – an instruction callback (ex: function(vm, gpr, fpr, data) {};)

Returns an instruction callback

Return type NativeCallback

Example:

```
>>> var icbk = vm.newInstCallback(function(vm, gpr, fpr, data) {
>>>   inst = vm.getInstAnalysis();
>>>   console.log("0x" + inst.address.toString(16) + " " + inst.disassembly);
>>>   return VMAction.CONTINUE;
>>> });
```

QBDI.prototype.**newVMCallback**(*cbk*)

Create a native **VM callback** from a JS function.

Arguments

- **cbk** – a VM callback (ex: function(vm, state, gpr, fpr, data) {};)

Returns a VM callback

Return type NativeCallback

Example:

```
>>> var vcbk = vm.newVMCallback(function(vm, evt, gpr, fpr, data) {
>>>   if (evt.event & VMEvent.EXEC_TRANSFER_CALL) {
>>>     console.warn("[!] External call to 0x" + evt.basicBlockStart.
↪toString(16));
>>>   }
>>>   return VMAction.CONTINUE;
>>> });
```

QBDI.**version**

QBDI version (major, minor, patch).

{string:String,integer:Number,major:Number,minor:Number,patch:Number}

2.1.7 Architecture Support

Our patching system uses a generic approach which means instruction support is not explicit but implicit. Only instructions that use the program counter register (`rip` under x86-64 and `pc` under ARM) or modify the control flow needs patching and we hope the current patching rules cover all such cases. However some corner cases or bugs in the disassembly and assembly backend of LLVM might still cause troubles.

To guarantee that instrumented programs run smoothly and that no such bugs exist, we are running automated tests on a wide variety of binaries (see *Testing* for more details). From those tests we can establish an instruction coverage

which provides an estimation of the instructions supported by QBDI. This estimation is far from complete because the other instructions were simply probably never encountered in the test suite.

Instructions below are listed using their LLVM MC naming. This naming convention distinguishes between size and operand variants of the same mnemonic: `ADD64rm` is a 64 bits add of a memory value to a register while `ADD32ri8` is a 32 bits add of a 8 bits immediate to a register.

Intel x86-64

The x86-64 support is complete and mature although there are still some rare bugs linked to problems in the assembly and disassembly backend of LLVM. Only a small part of SIMD instructions are covered by our tests but we do not expect any problems with the uncovered ones because their semantic are closely related with the covered ones.

Memory access information is only provided for general instructions, not for SIMD ones.

Instruction Coverage

```
ADC32mi8 ADC64mi8
ADD16mi8 ADD16mr ADD16rm ADD32i32 ADD32mi8 ADD32mr ADD32ri ADD32ri8 ADD32rm ADD32rr␣
↪ADD64i32 ADD64mi32 ADD64mi8 ADD64mr ADD64ri32 ADD64ri8 ADD64rm ADD64rr ADD8rr␣
↪ADDSDrm
AESENCLASTrr AESENCrr
AND16mi AND16mr AND32i32 AND32mi8 AND32mr AND32ri AND32ri8 AND32rm AND32rr AND64mi8␣
↪AND64mr AND64ri8 AND64rr AND8mi AND8mr AND8ri AND8rm AND8rr ANDPDrr
BSWAP32r BSWAP64r
BT64rr
CALL64m CALL64pcrel32 CALL64r
CDQ
CDQE
CMOVA32rr CMOVA64rr CMOVAE32rm CMOVAE32rr CMOVAE64rr CMOVB32rr CMOVB64rr CMOVBE32rr␣
↪CMOVBE64rm CMOVBE64rr CMOVE32rm CMOVE32rr CMOVE64rr CMOVG32rr CMOVG64rr CMOVGE32rr␣
↪CMOVL32rr CMOVLE32rr CMOVNE32rm CMOVNE32rr CMOVNE64rm CMOVNE64rr CMOVNS32rr␣
↪CMOVNS64rr CMOVS32rr CMOVS64rr
CMP16mi8 CMP16mr CMP16rm CMP16rr CMP32i32 CMP32mi CMP32mi8 CMP32mr CMP32ri CMP32ri8␣
↪CMP32rm CMP32rr CMP64i32 CMP64mi32 CMP64mi8 CMP64mr CMP64ri32 CMP64ri8 CMP64rm␣
↪CMP64rr CMP8i8 CMP8mi CMP8mr CMP8ri CMP8rm CMP8rr CMPSB
CMPXCHG32rm CMPXCHG64rm
COMISDrm COMISDrr
CQO
CVTSI2SDrr CVTSI642SDrr CVTTSD2SI64rr CVTTSD2SIrr
DEC32r
DIV64m DIV64r DIVSDrm DIVSDrr IDIV32m IDIV32r IDIV64r
IMUL32r IMUL32rm IMUL32rr IMUL64rmi8 IMUL64rr IMUL64rri32
JA_1 JA_4 JAE_1 JAE_4 JB_1 JB_4 JBE_1 JBE_4 JE_1 JE_4 JG_1 JG_4 JGE_1 JGE_4 JL_1 JL_4␣
↪JLE_1 JLE_4 JMP_1 JMP_4 JMP64m JMP64r JNE_1 JNE_4 JNP_1 JNS_1 JNS_4 JS_1 JS_4
LEA64_32r LEA64r
MFENCE
MOV16mi MOV16mr MOV32mi MOV32mr MOV32ri MOV32rm MOV32rr MOV64mi32 MOV64mr MOV64ri␣
↪MOV64ri32 MOV64rm MOV64rr MOV8mi MOV8mr MOVAPDrr MOVAPSmr MOVDQArm MOVDQArr␣
↪MOVDQUmr MOVDQUrm MOVQI2PQIrm MOVSDmr MOVSDrm MOVSL MOVSQ MOVSX32rm8 MOVSX32rr8␣
↪MOVSX64rm32 MOVSX64rm8 MOVSX64rr16 MOVSX64rr32 MOVSX64rr8 MOVUPSmr MOVUPSrm␣
↪MOVZX32rm16 MOVZX32rm8 MOVZX32rr16 MOVZX32rr8
MUL32r MUL64r MULSDrr
NEG32r NEG64r
NOOP NOOPL NOOPW
NOT32r NOT64m NOT64r
```

```
OR16rm OR32i32 OR32mi OR32mi8 OR32mr OR32ri8 OR32rm OR32rr OR64i32 OR64ri8 OR64rm
↪OR64rr OR8i8 OR8mi OR8mr OR8rm OR8rr
POP64r
PSHUFBrr PSHUFDri
PSLLDQri PSLLDri
PUSH64i32 PUSH64i8 PUSH64r PUSH64rmm
PXORrr
RETQ
ROL32r1 ROL32ri ROL64r1 ROL64ri
ROR32ri ROR64r1 ROR64ri
SAR32r1 SAR32rCL SAR32ri SAR64r1 SAR64ri
SBB32ri8 SBB32rr SBB64ri8 SBB64rr SBB8i8 SBB8ri
SCASB
SETAEr SETAr SETBEr SETBr SETEm SETEr SETGr SETLEr SETLr SETNEm SETNEr SETOr
SHL32rCL SHL32ri SHL64rCL SHL64ri
SHR16ri SHR32r1 SHR32rCL SHR32ri SHR64r1 SHR64rCL SHR64ri SHR8r1 SHR8ri
STOSQ
SUB32i32 SUB32mr SUB32ri SUB32ri8 SUB32rm SUB32rr SUB64mi8 SUB64mr SUB64ri32 SUB64ri8
↪SUB64rm SUB64rr
SUB8mr
SYSCALL
TEST16mi TEST16ri TEST16rr TEST32i32 TEST32mi TEST32mr TEST32ri TEST32rr TEST64ri32
↪TEST64rr TEST8i8 TEST8mi TEST8ri TEST8rr
UCOMISDrr
VADDSDrm VADDSDrr
VCVTSI2SDrr
VFMADD132SDm VFMADD132SDr VFMADD213SDm VFMADD231SDm
VFMSUB132SDr
VMOVAPDrr VMOVPQIto64rr VMOVSDrm
VMULSDrm VMULSDrr
VSUBSDrr
VXORPDrr
XADD32rm
XCHG32rm XCHG64rr
XOR32ri XOR32ri8 XOR32rm XOR32rr XOR64rm XOR64rr XOR8rm XORPSrr
```

Intel x86

The x86 support is based on x86-64.

Memory access information is only provided for general instructions, not for SIMD ones.

Instruction Coverage

```
ABS_F
ADC32mi8 ADC32mr ADC32ri ADC32ri8 ADC32rm ADC32rr
ADD16mi8 ADD16mr ADD16ri ADD16rm ADD32i32 ADD32mi ADD32mi8 ADD32mr ADD32ri ADD32ri8
↪ADD32rm ADD32rr ADD8rr ADD_F32m
AESENCLASTrr AESENCrr
AND16mi AND16mr AND32i32 AND32mi8 AND32mr AND32ri AND32ri8 AND32rm AND32rr AND8mi
↪AND8mr AND8ri AND8rm AND8rr
BSWAP32r
BT32rr
CALL32m CALL32r CALLpcrel32
CDQ
```

```
CHS_F
CLD
CMOVA32rr CMOVAE32rm CMOVAE32rr CMOVB32rm CMOVB32rr CMOVBE32rm CMOVBE32rr CMOVE32rm␣
↪CMOVE32rr CMOVG32rm CMOVG32rr CMOVGE32rr CMOVL32rr CMOVLE32rr CMOVNE32rm CMOVNE32rr␣
↪CMOVNS32rm CMOVNS32rr CMOVS32rr
CMP16mi8 CMP16mr CMP16rm CMP16rr CMP32i32 CMP32mi CMP32mi8 CMP32mr CMP32ri CMP32ri8␣
↪CMP32rm CMP32rr CMP8i8 CMP8mi CMP8mr CMP8ri CMP8rm CMP8rr CMPSB CMPSW CMPXCHG32rm
COM_FIPr COM_FIr
DEC32r_alt
DIV32m DIV32r DIV_F64m DIV_FPrST0 DIVR_F32m DIVR_F64m
FCOMP64m
FLDCW16m
FNSTCW16m FNSTSW16r
FXAM
IDIV32m
ILD_F32m
ILD_F64m
IMUL32r IMUL32rm IMUL32rmi8 IMUL32rr IMUL32rri
INC32r_alt
IST_FP32m IST_FP64m
JA_1 JA_4 JAE_1 JAE_4 JB_1 JB_4 JBE_1 JBE_4 JE_1 JE_4 JG_1 JG_4 JGE_1 JGE_4 JL_1 JL_4␣
↪JLE_1 JLE_4 JMP_1 JMP32m JMP32r JMP_4 JNE_1 JNE_4 JNP_1 JNS_1 JNS_4 JS_1 JS_4
LD_F0 LD_F1 LD_F32m LD_F64m LD_F80m LD_Frr
LEA32r
MOV16mi MOV16mr MOV16o32a MOV16rm MOV32ao32 MOV32mi MOV32mr MOV32o32a MOV32ri MOV32rm␣
↪MOV32rr MOV8mi MOV8mr MOV8o32a MOV8rm MOV8rr MOVAPSrr MOVDQArm MOVDQArr MOVDQUmr␣
↪MOVSB MOVSL MOVSX32rm8 MOVSX32rr16 MOVSX32rr8 MOVUPSmr MOVUPSrm MOVZX32rm16␣
↪MOVZX32rm8 MOVZX32rr16 MOVZX32rr8
MUL32m MUL32r MUL_F32m MUL_FPrST0 MUL_FST0r
NEG32r
NOOP
NOT32m NOT32r
OR16rm OR32i32 OR32mi OR32mi8 OR32mr OR32ri8 OR32rm OR32rr OR8i8 OR8mi OR8mr OR8ri␣
↪OR8rm OR8rr
POP32r
PSHUFBrr
PSHUFDri
PSLLDQri
PSLLDri
PUSH32i8 PUSH32r PUSH32rmm PUSHi32
PXORrr
RETL
ROL32r1 ROL32rCL ROL32ri
ROR32ri
SAHF
SAR32r1 SAR32rCL SAR32ri
SBB32mi8 SBB32ri8 SBB32rm SBB32rr SBB8i8 SBB8ri
SCASB
SETAEr SETAr SETBEr SETBr SETEm SETEr SETGr SETLEr SETLr SETNEm SETNEr SETOr
SHL32rCL SHL32ri SHLD32rrCL SHLD32rri8
SHR16ri SHR32r1 SHR32rCL SHR32ri SHR8m1 SHR8r1 SHR8ri SHRD32rri8
ST_F64m ST_FP64m ST_FP80m ST_FPrr
STOSL
SUB32i32 SUB32mi8 SUB32mr SUB32ri SUB32ri8 SUB32rm SUB32rr SUB8mr SUB8ri SUB8rm
TEST16mi TEST16ri TEST16rr TEST32i32 TEST32mi TEST32mr TEST32ri TEST32rr TEST8i8␣
↪TEST8mi TEST8mr TEST8ri TEST8rr
UCOM_FIr UCOM_FPr
XADD32rm
```

```
XCH_F
XCHG32ar XCHG32rm XCHG32rr
XOR16rr XOR32i32 XOR32mr XOR32ri XOR32ri8 XOR32rm XOR32rr XOR8rm XORPSrr
```

ARM

The public ARM support is currently limited to the ARM instruction set of ARMv6, there is no support for Thumb or Thumb2 yet. While this subset is quite small, it is enough to run pure ARM mode binary and demonstrate that complete ARM support is possible.

No memory access information is provided at the moment.

Instruction Coverage

```
ADCri ADCrr
ADDri ADDrr ADDrsi
ANDri ANDrr
BICri BICrsi
BL BLX BX_RET
Bcc # Encompass all conditionnal branching instructions
CLZ
CMNri CMNzrr
CMPri CMPrr CMPrsi
EORri EORrr EORrsi EORrsr
LDMIA LDMIA_UPD
LDRB_POST_IMM LDRB_PRE_IMM LDRBi12 LDRBrs
LDRD
LDRH
LDR_POST_IMM LDR_PRE_IMM LDRi12 LDRrs
MLA
MOVi MOVr MOVsi MOVsr
MUL
MVNi MVNr
ORRri ORRrr ORRrsi ORRrsr
REV
RSBri RSBrr
SBCri SBCrr
STMDB_UPD
STMIA STMIB
STRB_POST_IMM STRB_PRE_IMM STRBi12 STRBrs
STRD STRD_PRE
STR_POST_IMM STR_PRE_IMM STRi12 STRrs
SUBri SUBrr SUBrsi
TSTri TSTrr
UMULL
UXTB
VLDMDIA_UPD
VLDRD
VSTMDDB_UPD
```

2.2 Developer Documentation

2.2.1 Compilation From Source

To build this project the following dependencies are needed on your system: cmake, make (for Linux and macOS), ninja (for Android), Visual Studio (for Windows) and a C++11 toolchain for the platform of your choice.

The compilation is a two-step process:

- A local binary distribution of the dependencies is built.
- QBDI is built using those binaries.

The current dependencies which need to be built are LLVM and Google Test. This local built of LLVM is required because QBDI uses private APIs not exported by regular LLVM installations and because our code is only compatible with a specific version of those APIs. This first step is cached and only needs to be run once, subsequent builds only need to repeat the second step.

QBDI build system relies on CMake and requires to pass build configuration flags. To help with this step we provide shell scripts for common build configurations which follow the naming pattern `config-OS-ARCH.sh`. Modifying these scripts is necessary if you want to compile in debug or cross compile QBDI.

Linux

x86-64

Create a new directory at the root of the source tree, and execute the Linux configuration script:

```
mkdir build
cd build
../cmake/config-linux-X86_64.sh
```

If the build script warns you of missing dependencies for your platform (in the case of a first compilation), or if you want to rebuild them, execute the following commands:

```
make llvm
make gtest
```

This will rebuild the binary distribution of those dependencies for your platform. You can then relaunch the configuration script from above and compile:

```
../cmake/config-linux-X86_64.sh
make -j4
```

x86

The previous step can be follow but using the `config-linux-X86.sh` configuration script instead.

Cross-compiling for ARM

The same step as above can be used but using the `config-linux-ARM.sh` configuration script instead. This script however needs to be customized for your cross-compilation toolchain:

- The right binaries must be exported in the `AS`, `CC`, `CXX` and `STRIP` environment variables.

- The `-DCMAKE_C_FLAGS` and `-DCMAKE_CXX_FLAGS` should contain the correct default flags for your toolchain. At least the `ARM_ARCH`, `ARM_C_INCLUDE` and `ARM_CXX_INCLUDE` should be modified to match your toolchain but more might be needed.

macOS

Compiling QBDI on macOS requires a few things:

- A modern version of **macOS** (like Sierra)
- **Xcode** (from the *App Store* or *Apple Developer Tools*)
- the **Command Line Tools** (`xcode-select --install`)
- a package manager (preferably **MacPorts**, but *HomeBrew* should also be fine)
- some packages (`port install cmake wget`)

Once requirements are met, create a new directory at the root of the source tree, and execute the macOS configuration script:

```
mkdir build
cd build
../cmake/config-macOS-X86_64.sh
```

If the build script warns you of missing dependencies for your platform (in the case of a first compilation), or if you want to rebuild them, execute the following commands:

```
make llvm
make gtest
```

This will rebuild the binary distribution of those dependencies for your platform. You can then relaunch the build script from above and compile:

```
../cmake/config-macOS-X86_64.sh
make -j4
```

Windows

Building on Windows requires a pure Windows installation of *Python 3* (from the official packages, this is mandatory) in order to build our dependencies (we really hope to improve this in the future). It also requires an up-to-date CMake.

First, the `config-win-X86_64.py` should be edited to use the generator (the `-G` flag) matching your Visual Studio installation. Then the following command should be run:

```
mkdir build
cd build
python ../cmake/config-win-X86_64.py
```

If the build script warns you of missing dependencies for your platform (in the case of a first compilation), or if you want to rebuild them, execute the following commands:

```
MSBuild.exe deps\llvm.vcxproj
MSBuild.exe deps\gtest.vcxproj
```

This will rebuild the binary distribution of those dependencies for your platform. You can then relaunch the build script from above and compile:

```
python ../cmake/config-win-X86_64.py
MSBuild.exe /p:Configuration=Release ALL_BUILD.vcxproj
```

Android

Cross-compiling for Android requires the Android NDK and has only been tested under Linux. The `config-android-ARM.sh` configuration script should be customized to match your NDK installation and target platform:

- `NDK_PATH` should point to your Android NDK

From that point on the Linux guide can be followed using this configuration script.

2.2.2 Repository Organization

Root Tree

The root of the source tree is organized as follows:

`cmake/` Contains files required by the CMake build system.

`deps/` Contains the dependency build system and cache.

`docs/` Contains this documentation source and configuration.

`example/` Contains various QBDI usage examples.

`include/` Contains the public include files.

`package/` Contains the package generation scripts.

`src/` Contains QBDI source code. This tree is described further in section *Source Tree*.

`templates/` Contains QBDI usage templates.

`test/` Contains the functional test suite.

`tools/` Contains QBDI development tools: the validator and the validation runner.

Source Tree

The source files are organized as follows:

`src/Engine` Contains code related to the VM API and underlying Engine controlling the overall execution.

`src/ExecBlock` Contains code used to generate, store and execute JITed code inside ExecBlocks which form a code cache managed by the ExecBlockManager.

`src/ExecBroker` Contains code implementing the execution brokering mechanism allowing to switch between instrumented execution and real execution.

`src/Patch` Contains the PatchDSL implementation and per architecture support.

`src/Utility` Contains various utility class and functions.

2.2.3 Testing

Making mistakes while writing a DBI framework is particularly easy and often fatal. Executing code generated at runtime is a dangerous game and close attention should be paid to the patching engine. That is why we use two different testing methods.

Functional Test Suite

QBDI has a small functional test suite implemented using Google Test which verifies the essential functions of QBDI. This test suite is automatically built and made available in the `test/` build subdirectory:

```
$ ./test/QBDITest --gtest_color=yes
[==========] Running 57 tests from 9 test cases.
[----------] Global test environment set-up.
...
[==========] 57 tests from 9 test cases ran. (47 ms total)
[  PASSED  ] 57 tests.
```

Validator

The validator is a generic test system allowing to compare a normal execution with an instrumented execution. This way a large number of programs can be used as a test suite to ensure that QBDI does not alter their normal functions. It is only compatible with Linux and macOS at the moment.

The idea is to pilot a debugging session using an instrumented execution of the same program. These two instances share the same environment and arguments and originated from the same fork thus have the same memory layout. This allows to compare the program state at each step and verify that both execution take the same path and give the same result. The validator is not only capable of determining if two executions differ but also capable of identifying where they diverged. It thus double down as a debugging tool.

There is, however, a few caveats to this approach. First, the two instances of the program will compete for resources. This means running `sha1sum test.txt` will work because the two instances can read the same file at the same time, but removing a directory `rmdir testdir/` will always fail because only one instance will be able to delete the directory. Second, the dynamic memory allocations will not match on the two instances. This is because the instrumented instance is running the whole QBDI framework and validator instrumentation which is making extra allocations in between the original program allocations. A partial mitigation was nevertheless implemented which tracks allocations on both sides and compute an allowed offset for specific memory address ranges.

One of the essential concepts of the validator are error cascades. They establish a probable causality chain between errors and allow to backtrack from the point where the execution crashed or diverged to the probable cause. Indeed, an error might only cause problems thousands of instructions later.

The validator uses dynamic library injection to take control of a program startup. Options are communicated using environment variables. The following options are available:

`VALIDATOR_VERBOSITY`

- `Stat`: Only display execution statistics. This is the default.
- `Summary`: Display execution statistics and error cascade summaries.
- `Detail`: Display execution statistics and complete error cascades.
- `Full`: Display full execution trace, execution statistics and complete error cascades.

`VALIDATOR_COVERAGE` Specify a file name where instruction coverage statistics will be written out.

Linux

The validator uses `LD_PRELOAD` for injection under Linux and an example command line would be:

```
$ LD_PRELOAD=./tools/validator/libvalidator.so VALIDATOR_VERBOSITY=Detail VALIDATOR_
↪COVERAGE=coverage.txt ls
```

macOS

The validator uses `DYLD_INSERT_LIBRARIES` for injection under macOS and an example command line would be:

```
$ DYLD_INSERT_LIBRARIES=./tools/validator/libvalidator.dyld VALIDATOR_
↪VERBOSITY=Detail VALIDATOR_COVERAGE=coverage.txt ./ls
```

Please note that, under macOS, SIP prevents from debugging system binaries. A workaround is to copy the target binary in a local directory. Also, for the moment, the validator requires root to obtain debugging rights.

Validation Runner

The validation runner is an automation system to run a series of validation task specified in a configuration file and aggregate the results. Those results are stored in a database which enables to perform historical comparison between validation runs and warns in case of an anomaly.

2.2.4 Technical Reference

Instrumentation Process

Introduction

The Engine reads and disassembles the instrumented binary basic block per basic block. In case a conditional branch terminates a basic block, the execution result of this basic block is needed to determine the next basic block to execute. This makes the case for a per basic block processing and execution.

Every basic block is first patched to solve two main problems:

- **Relocation:** The basic block will be executed at a different location and thus every usage of the Program Counter, either directly as an operand or indirectly when using relative memory addressing, needs to be patched to make the code relocatable.
- **Control Flow Control:** Branching instructions should not be directly executed as this would result in the execution escaping from the instrumentation process. Thus the resulting target of a branching instruction needs to be computed without being taken.

Once a basic block has been patched, the instrumentations requested by the user code are applied. Both the patching and the instrumentation are expressed in an *Embedded Domain Specific Language*[1] called *PatchDSL* which is executed by the engine.

[1] https://en.wikipedia.org/wiki/Domain-specific_language

The resulting instrumented basic block is then handed over to the ExecBlockManager which handles a cache of basic blocks placed inside execution units called *ExecBlock*. The ExecBlockManager is tasked with finding memory space inside an ExecBlock to place the instrumented basic block and also retrieving cached basic blocks.

An ExecBlock manages on the guest side two memory pages: one for the code, the code block, and one for the data, the data block. The ExecBlock also handles the resolution of the relocation of the patched code before assembling it in the code block.

The instrumentation of the code allows to make callbacks to the user code directly from the instrumented binary through the ExecBlock. These callbacks allow to inspect and modify the state of execution of the guest on the host side at every point in time.

Implementation

The figure below presents the *life of an instruction* and summarizes the main steps and classes involved along the way. This is intended to give an overview of what the internals do.

An instruction exists in three different representations inside QBDI:

Bytes Raw bytes of machine code in memory.

MCInst LLVM machine code representation. The instruction is only partially disassembled but still provides a list of operands. One interested in more details regarding this representation should refer to the official LLVM documentation and experiment with *llvm-mc -show-inst*.

RelocatableInst QBDI representation of a relocatable MCInst. It consists in an MCInst and relocation information.

There is another important class: `QBDI::Patch`. A `QBDI::Patch` aggregates the patch and the instrumentation of a single instruction in the form of a list of `QBDI::RelocatableInst`. It is the smallest unit of code which can be assembled inside an *`QBDI::ExecBlock`* as patching or instrumentation code cannot be split in parts without problematic side effects.

The assembly and disassembly steps are directly handled by LLVM for us. The Engine takes care of the patching and instrumentation using a programmable list of *`QBDI::PatchRule`* and *`QBDI::InstrRule`*. More details on those rules can be found in the *PatchDSL* chapter. Relocation is handled directly in the *`QBDI::ExecBlock`*.

ExecBlock

Introduction

The ExecBlock is a concept which tries to simplify the problem of context switching between the host and the guest.

The main problem behind context switching is to be able to reference a memory location owned by the host in the context of the guest. Loading the memory location in a register effectively destroys a guest value which would thus need to be saved somewhere. One could allow the usage of the guest stack and save values on it but this design has two major drawbacks. Firstly, although supposedly unused, this modifies guest memory and could have side effects if the program uses uninitialized stack values or in the case of unforeseen optimizations. Secondly, this assumes that the stack registers always point on the stack or that a stack exists at all which might not be the case for more exotic languages or assembly code. The only possible mechanism left is to use relative addressing load and store. While x86 and x86-64 allow 32 bits offset, ARM only allows 11 bits offset[1]. The ExecBlock design thus only requires to be able to load and store general purpose registers with an address offset up to 4096 bytes.

[1] Thumb supports even less but this is not a problem as, with the exception of embedded ARM architectures (the Cortex-M series), one can always switch between ARM mode and Thumb mode.

Note: By default, AVX registers are also saved, one may want to disable this in case where it is not necessary, thus improving perfomances. It can be achieved using the environment flag **QBDI_FORCE_DISABLE_AVX**.

Some modern operating systems do not allow the allocation of memory pages with read, write and execute permissions (RWX) for security reasons as this greatly facilitates remote code execution exploits. It is thus necessary to allocate two separate pages, with different permissions, for code and data. By exploiting the fact that most architectures use memory pages of 4096 bytes[2], allocating a data memory page next to a code memory page would allow the first instruction to address at least the first address of the data memory page.

Memory Layout

An ExecBlock is thus composed of two contiguous memory pages: the first one is called the code block and has read and execute permissions (RX) and the second one is called the data block and has read and write permissions (RW). The code block contains a prologue, responsible for the host to guest context switching, followed by the code of the translated basic block and finally the epilogue responsible for the guest to host switching. Both the prologue and the epilogue use the beginning of the data block to store the context data. The context is split in three parts: the GPR context, the FPR context and the host context. The GPR and FPR context are straight forward and documented in the API itself as the GPRState and FPRState (see the State Management part of the API). The host context is used to store host data and a memory pointer called the selector. The selector is used by the prologue to determine on which basic block to jump next. The remaining space in the data block is used for shadows which can be used to store any data needed by the patching or instrumentation process.

Reference

class QBDI::**ExecBlock**

Manages the concept of an exec block made of two contiguous memory blocks (one for the code, the other for the data) used to store and execute instrumented basic blocks.

Public Functions

ExecBlock(Assembly &*assembly*, VMInstanceRef *vminstance* = nullptr)

Construct a new *ExecBlock*

Parameters

- `assembly`: Assembly used to assemble instructions in the *ExecBlock*.
- `vminstance`: Pointer to public engine interface

void **show**() **const**

Display the content of an exec block to stderr.

VMAction **execute**()

Execute the sequence currently programmed in the selector of the exec block. Take care of the callbacks handling.

[2] This is at least true for x86, x86_64, ARM, ARM64 and PowerPC.

SeqWriteResult **writeSequence**(std::vector<Patch>::const_iterator *seqStart*, std::vector<Patch>::const_iterator *seqEnd*, SeqType *seqType*)

Write a new sequence in the exec block. This function does not guarantee that the sequence will be written in its entierty and might stop before the end using an architecture specific terminator. Return 0 if the exec block was full and no instruction was written.

Return A structure detailling the write operation result.

Parameters

- `seqStart`: [in] Iterator to the start of a list of patches.
- `seqEnd`: [in] Iterator to the end of a list of patches.
- `seqType`: [in] Type of the sequence.

uint16_t **splitSequence**(uint16_t *instID*)

Split an existing sequence at instruction instID to create a new sequence.

Return The new sequence ID.

Parameters

- `instID`: [in] ID of the instruction where to split the sequence at.

rword **getDataBlockOffset**() **const**

Compute the offset between the current code stream position and the start of the data block. Used for pc relative memory access to the data block.

Return The computed offset.

rword **getEpilogueOffset**() **const**

Compute the offset between the current code stream position and the start of the exec block epilogue code. Used for computing the remaining code space left or jumping to the exec block epilogue at the end of a sequence.

Return The computed offset.

rword **getCurrentPC**() **const**

Obtain the value of the PC where the *ExecBlock* is currently writing instructions.

Return The PC value.

uint16_t **getNextInstID**() **const**

Obtain the current instruction ID.

Return The current instruction ID.

uint16_t **getInstID**(rword *address*) **const**

Obtain the instruction ID for a specific address (the address must exactly match the start of the instruction).

Return The instruction ID or NOT_FOUND.

Parameters

- `address`: The address of the start of the instruction.

uint16_t **getCurrentInstID**() **const**
Obtain the current instruction ID.

Return The ID of the current instruction.

const InstMetadata ***getInstMetadata**(uint16_t *instID*) **const**
Obtain the instruction metadata for a specific instruction ID.

Return The metadata of the instruction.

Parameters

- `instID`: The instruction ID.

rword **getInstAddress**(uint16_t *instID*) **const**
Obtain the instruction address for a specific instruction ID.

Return The address of the instruction.

Parameters

- `instID`: The instruction ID.

const llvm::MCInst ***getOriginalMCInst**(uint16_t *instID*) **const**
Obtain the original MCInst for a specific instruction ID.

Return The original MCInst of the instruction.

Parameters

- `instID`: The instruction ID.

uint16_t **getNextSeqID**() **const**
Obtain the next sequence ID.

Return The next sequence ID.

uint16_t **getSeqID**(rword *address*) **const**
Obtain the sequence ID for a specific address (the address must exactly match the start of the sequence).

Return The sequence ID or NOT_FOUND.

Parameters

- `address`: The address of the start of the sequence.

uint16_t **getSeqID**(uint16_t *instID*) **const**
Obtain the sequence ID containing a specific instruction ID.

Return The sequence ID or NOT_FOUND.

Parameters

- `instID`: The instruction ID.

uint16_t **getCurrentSeqID**() **const**
Obtain the current sequence ID.

Return The ID of the current sequence.

uint16_t **getSeqStart**(uint16_t *seqID*) **const**
Obtain the sequence start address for a specific sequence ID.

Return The start address of the sequence.

Parameters

- `seqID`: The sequence ID.

uint16_t **getSeqEnd**(uint16_t *seqID*) **const**
Obtain the instruction id of the sequence end address for a specific sequence ID.

Return The end address of the sequence.

Parameters

- `seqID`: The sequence ID.

void **selectSeq**(uint16_t *seqID*)
Set the selector of the exec block to a specific sequence offset. Used to program the execution of a specific sequence within the exec block.

Parameters

- `seqID`: [in] Basic block ID within the exec block.

Context ***getContext**() **const**
Get a pointer to the context structure stored in the data block.

Return The context pointer.

uint16_t **newShadow**(uint16_t *tag* = NO_REGISTRATION)
Allocate a new shadow within the data block. Used by relocation to load or store data from the instrumented code.

Return The shadow id (which is its index within the shadow array).

Parameters

- `tag`: The tag associated with the registration, 0xFFFF is reserved for unregistered shadows.

void **setShadow**(uint16_t *id*, rword *v*)
Set the value of a shadow.

Parameters

- `id`: [in] ID of the shadow to set.
- `v`: [in] Value to assigne to the shadow.

rword **getShadow**(uint16_t *id*) **const**
Get the value of a shadow.

Return Value of the shadow.

Parameters

- `id`: [in] ID of the shadow.

rword **`getShadowOffset`**(uint16_t *id*) **`const`**
Get the offset of a shadow within the data block.

Return *Offset* of the shadow.

Parameters

- `id`: [in] ID of the shadow.

PatchDSL

Contents

Language Concepts

Type System

The PatchDSL nomenclature is formalized in function of where information belongs using two dichotomies:

- A physical dichotomy is made between data stored in the machine registers and data stored in memory.
- A conceptual dichotomy is made between data belonging or generated by the target program and data belonging or generated by QBDI.

This creates four main categories of data being manipulated by the PatchDSL. These categories and the nomenclature for their interaction are represented below.

Reg: They represent machine registers storing data created and used by the target program.

Temp: They represent machine registers storing data used by the instrumentation. Those are temporary scratch registers which were allocated by saving a Reg into the context and are bound to be deallocated by restoring the Reg value from the context.

Context: The context stores in memory the processor state associated with the target program. It is mostly used for context switching between the target program and the instrumentation process and also for allocating temporary registers.

Shadows: They represent shadow data associated with a patch and instrumented instruction. They can be used by QBDI to store constants or Tagged Shadows.

Metadata: Any data regarding the execution which can be generated by QBDI. For examples obtaining instruction operand values or memory access addresses.

The main objective of this strict naming convention is to make the distinction between pure (no side effects) operations and operations affecting the program state as clear as possible. To make this distinction even more apparent the DSL is strongly typed forcing variables to be declared by allocating one of the typed structures below with its constructor. Because some of those structures simply alias an integer value it can be tempting to directly use the integer value, letting the compiler do the implicit type conversion. However the point of those structures is to give a context to what those integer constants represent and the best practice is to use them everywhere possible.

struct QBDI::**Reg**
Structure representing a register variable in PatchDSL.

Public Functions

Reg(unsigned int *id*)
Create a new register variable.

Parameters

- id: The id of the register to represent.

operator unsigned int()
Convert this structure to an LLVM register id.

Return LLVM register id.

rword **offset**()
Return the offset of this register storage in the context part of the data block.

Return The offset.

struct QBDI::**Temp**
Structure representing a temporary register variable in PatchDSL.

Public Functions

Temp(unsigned int *id*)
Represent a temporary register variable idenified by a unique ID. Inside a patch rules or a instrumentation rules, *Temp* with identical ids point to the same physical register. The id 0xFFFFFFFF is reserved for internal uses. The mapping from id to physical register is determined at generation time and the allocation and deallocation instructions are automatically added to the patch.

Parameters

- `id`: The id of the temp to represent.

operator unsigned int()
Convert this *Temp* to its id.

Return This *Temp* id.

struct QBDI::**Shadow**
Structure representing a shadow variable in PatchDSL.

Public Functions

Shadow(uint16_t *tag*)
Allocate a new shadow variable in the data block with the corresponding tag.

Parameters

- `tag`: The tag of the new shadow variable.

rword **getTag**()
Return the tag associated with this shadow variable.

Return The tag of the shadow variable.

struct QBDI::**Constant**
Structure representing a constant value in PatchDSL.

Public Functions

Constant(rword *v*)
Represent a constant value.

Parameters

- `v`: The represented value.

operator rword()
Convert this structure to its value.

Return This constant value.

struct QBDI::**Offset**
Structure representing a memory offset variable in PatchDSL.

Public Functions

Offset (int64_t *offset*)
Allocate a new offset variable with its offset value.

Parameters

- `offset`: The offset value

Offset (*Reg reg*)
Allocate a new offset variable with the offset in the context of a specific register.

Parameters

- `reg`: The register whose offset to represent.

operator int64_t ()
Convert this structure to its value.

Return This offset value.

struct `QBDI::Operand`
Structure representing an operand instruction variable in PatchDSL.

Public Functions

Operand (unsigned int *idx*)
Represent an operand instruction identified by its index in the LLVM MCInst representation of the instruction.

Parameters

- `idx`: The operand index.

operator unsigned int ()
Convert this *Operand* to its idx.

Return This *Operand* idx.

Statements

There are three main categories of statements composing PatchDSL, each characterized by a different virtual base classes. The specialization of those base classes are PatchDSL statements.

QBDI::PatchCondition They are used to match specific instructions. They take the instruction and its context as an input and return a boolean.

QBDI::PatchGenerator They represent operations generating new instructions. They take the instruction and its context as an input and return a list of `QBDI::RelocatableInst` constituting the patch. In some exceptional cases no output is generated.

QBDI::InstTransform They represent operations transforming an instruction. They only manipulate an instruction and need to be used with a `QBDI::PatchGenerator` to output a `QBDI::RelocatableInst`.

Those statements are all evaluated on an implicit context. In the case of `QBDI::InstTransform` the context is the instruction to modify which is determined by the `QBDI::PatchGenerator` which use it. In the case of `QBDI::PatchCondition` and `QBDI::PatchGenerator` this context is made of:

- the current instruction
- the current instruction size
- the current address

The output of each statement thus depends on the statement parameters and this implicit context.

Rules

PatchDSL is used to write short sequence of statements called *rules*. There exists two variants of rules, patching rules (*QBDI::PatchRule*) and instrumentation rules (*QBDI::InstrRule*), but they both relies on the same principle. A rule is composed of two parts:

Condition: A `QBDI::PatchCondition` statement which express the condition under which the rule should be applied. Multiple statements can be combined in a boolean expression using `QBDI::Or` and `QBDI::And`. If the evaluation of this expression returns `true` then the generation part of the rule is evaluated.

Generation: A list of `QBDI::PatchGenerator` statements which will generate the patch rule. Each statement can output one or several `QBDI::RelocatableInst`, the resulting patch being the aggregation of all those statement outputs.

class QBDI::**PatchRule**

A patch rule written in PatchDSL.

Inherits from QBDI::AutoAlloc< PatchRule, PatchRule >

Public Functions

PatchRule(PatchCondition::SharedPtr *condition*, PatchGenerator::SharedPtrVec *generators*)

Allocate a new patch rule with a condition and a list of generators.

Parameters

- `condition`: A PatchCondition which determine wheter or not this *PatchRule* applies.
- `generators`: A vector of PatchGenerator which will produce the patch instructions.

bool **canBeApplied**(**const** llvm::MCInst **inst*, rword *address*, rword *instSize*, llvm::MCInstrInfo **MCII*)

Determine wheter this rule applies by evaluating this rule condition on the current context.

Return True if this patch condition evaluate to true on this context.

Parameters

- `inst`: The current instruction.
- `address`: The current instruction address.
- `instSize`: The current instruction size.
- `MCII`: An LLVM MC instruction info context.

Patch **generate**(**const** llvm::MCInst *_inst_, rword _address_, rword _instSize_, llvm::MCInstrInfo *_MCII_, llvm::MCRegisterInfo *_MRI_, **const** Patch *_toMerge_ = nullptr)

Generate this rule output patch by evaluating its generators on the current context. Also handles the temporary register management for this patch.

Return A Patch which is composed of the input context and a series of RelocatableInst.

Parameters

- `inst`: The current instruction.
- `address`: The current instruction address.
- `instSize`: The current instruction size.
- `MCII`: A LLVM::MCInstrInfo classes used for internal architecture specific queries.
- `MRI`: A LLVM::MCRegisterInfo classes used for internal architecture specific queries.
- `toMerge`: An eventual previous patch which is to be merged with the current instruction.

class QBDI::**InstrRule**

An instrumentation rule written in PatchDSL.

Inherits from QBDI::AutoAlloc< InstrRule, InstrRule >

Public Functions

InstrRule(PatchCondition::SharedPtr _condition_, PatchGenerator::SharedPtrVec _patchGen_, _InstPosition_ _position_, bool _breakToHost_)

Allocate a new instrumentation rule with a condition, a list of generators, an instrumentation position and a breakToHost request.

Parameters

- `condition`: A PatchCondition which determine wheter or not this *PatchRule* applies.
- `patchGen`: A vector of PatchGenerator which will produce the patch instructions.
- `position`: An enum indicating wether this instrumentation should be positioned before the instruction or after it.
- `breakToHost`: A boolean determining whether this instrumentation should end with a break to host (in the case of a callback for example).

bool **canBeApplied**(**const** Patch &_patch_, llvm::MCInstrInfo *_MCII_)

Determine wheter this rule applies by evaluating this rule condition on the current context.

Return True if this instrumentation condition evaluate to true on this patch.

Parameters

- `patch`: A patch containing the current context.
- `MCII`: An LLVM MC instruction info context.

void **instrument**(Patch &_patch_, llvm::MCInstrInfo *_MCII_, llvm::MCRegisterInfo *_MRI_)

Instrument a patch by evaluating its generators on the current context. Also handles the temporary register management for this patch.

Parameters

- `patch`: The current patch to instrument.
- `MCII`: A LLVM::MCInstrInfo classes used for internal architecture specific queries.
- `MRI`: A LLVM::MCRegisterInfo classes used for internal architecture specific queries.

Transforms

Transform statements, with the `QBDI::InstTransform` virtual base class, are a bit more subtle than other statements.

Currently their operation is limited to the `QBDI::ModifyInstruction` generators which always operate on the instruction of the implicit context of a patch or instrumentation rule. However their usage could be extended in the future.

Their purpose is to allow to write more generic rules by allowing modifications which can operate on a class of instructions. Using instruction transforms requires to understand the underlying LLVM MCInst representation of an instruction and `llvm-mc -show-inst` is an helpful tool for this task.

PatchDSL Examples

Below some real examples of patch and instrumentation rules are shown.

Basic Patching

Generic PC Substitution Patch Rule

Instructions using the Program Counter (PC) in their computations are problematic because QBDI will reassemble and execute the code at another address than the original code location and thus the value of the PC will change. This kind of computation using the PC is often found when using relative memory addressing.

Some cases can be more difficult to handle, but most of these instructions can be patched using a very simple generic rule performing the following steps:

1. Allocate a scratch register by saving a register value in the context part of the data block.
2. Load the value the PC register should have, into the scratch register.
3. Perform the original instruction but with PC replaced by the scratch register.
4. Deallocate the scratch register by restoring the register value from context part of the data block.

The PatchDSL *`QBDI::PatchRule`* handles step 1 and 4 automatically for us. Expressing step 2 and 3 is relatively simple:

```
PatchRule(
    // Condition: Applies on every instruction using the register REG_PC
    UseReg(Reg(REG_PC)),
    // Generators: list of statements generating the patch
    {
        // Compute PC + 0 and store it in a new temp with id 0
        GetPCOffset(Temp(0), Constant(0)),
        // Modify the instruction by substituting REG_PC with the temp having id 0
        ModifyInstruction({
            SubstituteWithTemp(Reg(REG_PC), Temp(0))
```

```
        })
    }
)
```

This rule is generic and works under X86_64 as well as ARM. Some more complex cases of instructions using PC need to be handled another way though.

Simple Branching Instruction Patch

Another simple case which needs to be handled using a patch rule is branching instructions. They cannot be executed because that would mean DBI process would lose the hand on the execution. Instead of executing the branch operation, the branch target is computed and used to overwrite the value of the PC in the context part of the data block. This is followed by a context switch back to the VM which will use this target as the address where to continue the execution.

The simplest cases are the "branch to an address stored in a register" instructions. Again the temporary register allocation is automatically taken care of by the *QBDI::PatchRule* and we only need to write the patching logic:

```
PatchRule(
    // Condition: only on BX or BX_pred LLVM MCInst
    Or({
        OpIs(llvm::ARM::BX),
        OpIs(llvm::ARM::BX_pred)
    }),
    // Generators
    {
        // Obtain the value of the operand with index 0 and store it in a new temp␣
↪with id 0
        GetOperand(Temp(0), Operand(0)),
        // Write the temp with id 0 at the offset in the data block of the context␣
↪value of REG_PC.
        WriteTemp(Temp(0), Offset(Reg(REG_PC)))
    }
)
```

Two things are important to notice here. First we use `QBDI::Or` to combine multiple `QBDI::PatchCondition`. Second the fact we need to stop the execution here and switch back to the context of the VM is not expressed in the patch. Indeed the patching engine simply notices that this patch overwrites the value of the PC and thus needs to end the basic block after it.

Advanced Patching

Conditional Branching Instruction Patch

The previous section dealt with simple patching cases where the rule does not need to be very complex. Conditional instructions can add a significant amount of complexity to the writing of a patch rules and requires some tricks. Below is the patch for the ARM conditional branching instruction:

```
PatchRule(
    // Condition: every Bcc instructions (e.g. BNE, BEQ, etc.)
    OpIs(llvm::ARM::Bcc),
    // Generators
    {
        // Compute the Bcc target (which is PC relative) and store it in a new temp␣
↪with id 0
```

```
        GetPCOffset(Temp(0), Operand(0)),
        // Modify the jump target such as it potentially skips the next generator
        ModifyInstruction({
            SetOperand(Operand(0), Constant(0))
        }),
        // Compute the next instruction address and store it in temp with id 0
        GetPCOffset(Temp(0), Constant(-4)),
        // At this point either:
        //  * The jump was not taken and Temp(0) stores the next instruction address.
        //  * The jump was taken and Temp(0) stores the Bcc target
        // We thus write Temp(0) which has the correct next address to execute in the␣
↪REG_PC
        // value in the context part of the data block.
        WriteTemp(Temp(0), Offset(Reg(REG_PC)))
    }
)
```

As we can see, this code reuses the original conditional branching instruction to create a conditional move. While this is a trick, it is an architecture independent trick which is also used under X86_64. Some details can be noted though. First the next instruction address is PC - 4 which is an ARM specificity. Secondly, the constant used to overwrite the jump target needs to be determined by hand as QBDI does not have the capacity to compute it automatically.

Complex InstTransform

The patch below is used to patch instructions which load their branching target from a memory address under X86_64. It exploits `QBDI::InstTransform` to convert the instruction into a load from memory to obtain this branching target:

```
PatchRule(
    // Condition: applies on CALL where the target is at a relative memory location␣
↪(thus uses REG_PC)
    And({
        OpIs(llvm::X86::CALL64m),
        UseReg(Reg(REG_PC))
    }),
    // Generators
    {
        // First compute PC + 0 and stores it into a new temp with id 0
        GetPCOffset(Temp(0), Constant(0)),
        // Transforms the CALL *[RIP + ...] into MOV Temp(1), *[Temp(0) + ...]
        ModifyInstruction({
            // RIP is replaced with Temp(0)
            SubstituteWithTemp(Reg(REG_PC), Temp(0)),
            // The opcode is changed to a 64 bits MOV from memory to a register
            SetOpcode(llvm::X86::MOV64rm),
            // We insert the destination register, a new temp with id 1,  at the␣
↪beginning of
            // the operand list
            AddOperand(Operand(0), Temp(1))
        }),
        // Temp(1) thus contains the CALL target.
        // We use the X86_64 specific SimulateCall with this target.
        SimulateCall(Temp(1))
    }
)
```

A few things need to be noted. First the sequence of `QBDI::InstTransform` is complex because it substitutes RIP and it mutates the CALL into a MOV. Secondly, new *QBDI::Temp* can be instantiated and used anywhere in the program. Lastly, some complex architecture specific mechanisms have been abstracted in single `QBDI::PatchGenerator`, like `QBDI::SimulateCall`.

Instrumentation Callbacks

QBDI::InstrRule allows to insert inline instrumentation inside the patch with a concept similar to the rules shown previously. Callbacks to host code are triggered by a break to host with specific variables set correctly in the host state part of the context:

- the hostState.callback should be set to the callback function address to call.
- the hostState.data should be set to the callback function data parameter.
- the hostState.origin should be set to the ID of the current instruction (see `QBDI::GetInstId`).

In practice, there exists a function which can generate the PatchGenerator needed to setup those variables correctly:

PatchGenerator::SharedPtrVec `QBDI::`**`getCallbackGenerator`**(*InstCallback cbk*, void **data*)
Output a list of PatchGenerator which would set up the host state part of the context for a callback.

Return A list of PatchGenerator to set up this callback call.

Parameters

- `cbk`: The callback function to call.
- `data`: The data to pass as an argument to the callback function.

Thus, in practice, a *QBDI::InstrRule* which would set up a callback on every instruction writing data in memory would look like this:

```
InstrRule(
    // Condition: on every instruction making write access
    DoesWriteAccess(),
    // Generators: set up a callback to someCallbackFunction with someParameter
    getCallbackGenerator(someCallbackFunction, someParameter),
    // Position this instrumentation after the instruction
    InstPosition::POSTINST,
    // Break to the host after the instrumentation (required for the callback to be␣
↪made)
    true
));
```

However the callback generator can be written directly in PatchDSL for more advantageous usages. The instrumentation rules below pass directly the written data as the callback parameter:

```
InstrRule(
    // Condition: on every instruction making write access
    DoesWriteAccess(),
    // Generators: set up a callback to someCallbackFunction with someParameter
    {
        // Set hostState.callback to the callback function address
        GetConstant(Temp(0), Constant((rword) someCallbackFunction)),
        WriteTemp(Temp(0), Offset(offsetof(Context, hostState.callback))),
        // Set hostState.data as the written value
        GetWriteValue(Temp(0)),
        WriteTemp(Temp(0), Offset(offsetof(Context, hostState.data))),
```

```
        // Set hostState.origin as the current instID
        GetInstId(Temp(0)),
        WriteTemp(Temp(0), Offset(offsetof(Context, hostState.origin)))
    }
    // Position this instrumentation after the instruction
    QBDI::InstPosition::POSTINST,
    // Break to the host after the instrumentation (required for the callback to be
↪made)
    true
));
```

Inline Instrumentation

PatchDSL Reference

PatchConditions

Boolean Arithmetic Operators

- `QBDI::And`
- `QBDI::Or`
- `QBDI::Not`
- `QBDI::True`

`QBDI::And::`**`And`**(PatchCondition::SharedPtrVec *conditions*)
Return true if every PatchCondition of the list conditions return true (lazy evaluation).

Parameters

- `conditions`: List of conditions to evaluate.

`QBDI::Or::`**`Or`**(PatchCondition::SharedPtrVec *conditions*)
Return true if one of the PatchCondition of the list conditions return true (lazy evaluation).

Parameters

- `conditions`: List of conditions to evaluate.

`QBDI::Not::`**`Not`**(PatchCondition::SharedPtr *condition*)
Return the logical inverse of condition.

Parameters

- `condition`: Condition to evaluate.

`QBDI::True::`**`True`**()
Return true.

Instruction Conditions

- QBDI::OpIs
- QBDI::RegIs
- QBDI::UseReg
- QBDI::AddressInRange
- QBDI::OperandIsReg
- QBDI::OperandIsImm

QBDI::OpIs::**OpIs** (unsigned int *op*)
Return true if the instruction opcode is equal to op.

Parameters

- op: LLVM instruction opcode ID.

QBDI::RegIs::**RegIs** (*Operand* *opn*, *Reg* *reg*)
Return true if the instruction operand with index opn is a register equal to reg.

Parameters

- opn: The instruction operand.
- reg: The register to compare with.

QBDI::UseReg::**UseReg** (*Reg* *reg*)
Return true if the instruction uses reg as one of its operand.

Parameters

- reg: The register to compare with.

QBDI::InstructionInRange::**InstructionInRange** (*Constant* *start*, *Constant* *end*)
Return true if the instruction address is in the range [start, end[(end not included).

Parameters

- start: Start of the range.
- end: End of the range (not included).

QBDI::OperandIsReg::**OperandIsReg** (*Operand* *opn*)
Return true if the instruction operand opn is a register.

Parameters

- opn: The instruction operand.

QBDI::OperandIsImm::**OperandIsImm** (*Operand* *opn*)
Return true if the instruction operand opn is an immediate.

Parameters

- opn: The instruction operand.

Memory Access Conditions

- QBDI::DoesReadAccess
- QBDI::DoesWriteAccess
- QBDI::ReadAccessSizeIs
- QBDI::WriteAccessSizeIs
- QBDI::IsStackRead
- QBDI::IsStackWrite

QBDI::DoesReadAccess::**DoesReadAccess**()
Return true if the instruction read data from memory.

QBDI::DoesWriteAccess::**DoesWriteAccess**()
Return true if the instruction write data to memory.

QBDI::ReadAccessSizeIs::**ReadAccessSizeIs**(*Constant* *size*)
Return true if the instruction read access size equal to size.

Parameters

- size: Size to compare with.

QBDI::WriteAccessSizeIs::**WriteAccessSizeIs**(*Constant* *size*)
Return true if the instruction write access size equal to size.

Parameters

- size: Size to compare with.

QBDI::IsStackRead::**IsStackRead**()
Return true if the instruction is reading data from the stack.

QBDI::IsStackWrite::**IsStackWrite**()
Return true if the instruction is writing data to the stack.

PatchGenerators

Registry and Temporary Operations

- QBDI::LoadReg
- QBDI::SaveReg
- QBDI::CopyReg
- QBDI::WriteTemp

QBDI::LoadReg::**LoadReg**(*Reg* *reg*, *Offset* *offset*)
Load a register from the data block at the specified offset. This can be used to load register values from the context part of the data block.

Parameters

- reg: A register where the value will be loaded.
- offset: The offset in the data block from where the value will be loaded.

QBDI::SaveReg::**SaveReg**(*Reg reg*, *Offset offset*)
Save a register in the data block at the specified offset. This can be used to save register values in the context part of the data block.

Parameters

- reg: A register which will be saved.
- offset: The offset in the data block where the register will be written.

QBDI::CopyReg::**CopyReg**(*Temp temp*, *Reg reg*)
Copy a register in a temporary.

Parameters

- temp: A temporary where the register will be copied.
- reg: The register which will be copied

QBDI::WriteTemp::**WriteTemp**(*Temp temp*, *Offset offset*)
Write a temporary value in the data block at the specified offset. This can be used to overwrite register values in the context part of the data block.

Parameters

- temp: A temporary which will be written.
- offset: The offset in the data block where the temporary will be written.

QBDI::WriteTemp::**WriteTemp**(*Temp temp*, *Shadow shadow*)
Write a temporary value in a shadow in the data block.

Parameters

- temp: A temporary which will be written.
- shadow: The shadow use to store the value.

Instruction Information Queries

- QBDI::GetOperand
- QBDI::GetConstant
- QBDI::GetPCOffset

QBDI::GetOperand::**GetOperand**(*Temp temp*, *Operand op*)
Obtain the value of the operand op and copy it's value in a temporary. If op is an immediate the immediate value is copied, if op is a register the register value is copied.

Parameters

- temp: A temporary where the value will be copied.
- op: The operand index (relative to the instruction LLVM MCInst representation) to be copied.

QBDI::GetConstant::**GetConstant**(*Temp temp*, *Constant cst*)
Copy a constant in a temporary.

Parameters

- `temp`: A temporary where the value will be copied.
- `cst`: The constant to copy.

`QBDI::GetPCOffset::`**`GetPCOffset`**(*Temp temp*, *Constant cst*)
Interpret a constant as a PC relative offset and copy it in a temporary. It can be used to obtain the current value of PC by using a constant of 0.

Parameters

- `temp`: A temporary where the value will be copied.
- `cst`: The constant to be used.

`QBDI::GetPCOffset::`**`GetPCOffset`**(*Temp temp*, *Operand op*)
Interpret an operand as a PC relative offset and copy it in a temporary. It can be used to obtain jump/call targets or relative memory access addresses.

Parameters

- `temp`: A temporary where the value will be copied.
- `op`: The operand index (relative to the instruction LLVM MCInst representation) to be used.

Memory Access Information Queries

- `QBDI::GetReadAddress`
- `QBDI::GetWriteAddress`
- `QBDI::GetReadValue`
- `QBDI::GetWriteValue`

`QBDI::GetReadAddress::`**`GetReadAddress`**(*Temp temp*)
Resolve the memory address where the instructions will read its value and copy the address in a temporary. This PatchGenerator is only guaranteed to work before the instruction has been executed.

Parameters

- `temp`: A temporary where the memory address will be copied.

`QBDI::GetWriteAddress::`**`GetWriteAddress`**(*Temp temp*)
Resolve the memory address where the instructions will write its value and copy the address in a temporary. This PatchGenerator is only guaranteed to work before the instruction has been executed.

Parameters

- `temp`: A temporary where the memory address will be copied.

`QBDI::GetReadValue::`**`GetReadValue`**(*Temp temp*)
Resolve the memory address where the instructions will read its value and copy the value in a temporary. This PatchGenerator is only guaranteed to work before the instruction has been executed.

Parameters

- `temp`: A temporary where the memory value will be copied.

QBDI::GetWriteValue::**GetWriteValue**(*Temp temp*)
Resolve the memory address where the instructions has written its value and copy back the value in a temporary. This PatchGenerator is only guaranteed to work after the instruction has been executed.

Parameters

- temp: A temporary where the memory value will be copied.

Special

- QBDI::ModifyInstruction
- QBDI::DoNotInstrument

QBDI::ModifyInstruction::**ModifyInstruction**(InstTransform::SharedPtrVec *transforms*)
Apply a list of InstTransform to the current instruction and output the result.

Parameters

- transforms: Vector of InstTransform to be applied.

QBDI::DoNotInstrument::**DoNotInstrument**()
Adds a "do not instrument" flag to the resulting patch which allows it to skip the instrumentation process of the engine.

X86_64 Specific

- QBDI::SimulateCall
- QBDI::SimulateRet

QBDI::SimulateCall::**SimulateCall**(*Temp temp*)
Simulate the effects of a call to the address stored in a temporary. The target address overwrites the stored value of RIP in the context part of the data block and the return address is pushed onto the stack. This generator signals a PC modification and triggers and end of basic block.

Parameters

- temp: Stores the call target address. Overwritten by this generator.

QBDI::SimulateRet::**SimulateRet**(*Temp temp*)
Simulate the effects of a return instruction. First the return address is popped from the stack into a temporary, then an optional stack deallocation is performed and finally the return address is written in the stored value of RIP in the context part of the data block. This generator signals a PC modification and triggers an end of basic block.

The optional deallocation is performed if the current instruction has one single immediate operand (which is the case of RETIQ and RETIW).

Parameters

- temp: Any unused temporary, overwritten by this generator.

ARM Specific

- QBDI::SimulateLink
- QBDI::SimulatePopPC

QBDI::SimulateLink::**SimulateLink**(*Temp temp*)
Simulate the effects of the link operation performed by BL and BLX instructions: the address of the next instruction is copied into the LR register. A temp and a shadow are needed to compute this address.

Parameters

- temp: Any unused temporary, overwritten by this generator.

QBDI::SimulatePopPC::**SimulatePopPC**(*Temp temp*)
Simulate an (eventually conditional) return instruction (POPcc PC). The conditional code of the current instruction is used by this generator. This generator signals a PC modification and triggers an end of basic block.

Parameters

- temp: Any unused temporary, overwritten by this generator.

Internals

- QBDI::GetInstId
- QBDI::JmpEpilogue

QBDI::GetInstId::**GetInstId**(*Temp temp*)
Copy an *ExecBlock* specific id for the current instruction in a temporary. This id is used to identify the instruction responsible for a callback in the engine and is only meant for internal use.

Parameters

- temp: A temporary where the id will be copied.

QBDI::JmpEpilogue::**JmpEpilogue**()
Generate a jump instruction which target the epilogue of the *ExecBlock*.

InstTransform

- QBDI::SetOperand
- QBDI::AddOperand
- QBDI::SubstituteWithTemp
- QBDI::RemoveOperand
- QBDI::SetOpcode

QBDI::SetOperand::**SetOperand**(*Operand opn*, *Temp temp*)
Set the operand opn of the instruction as the *Temp* temp.

Parameters

- opn: *Operand* index in the LLVM MCInst representation.
- temp: Temporary register which will be set as the new operand

QBDI::SetOperand::**SetOperand**(*Operand* *opn*, *Reg* *reg*)
Set the operand opn of the instruction as the *Reg* reg.

Parameters

- opn: *Operand* index in the LLVM MCInst representation.
- reg: Register which will be set as the new operand.

QBDI::SetOperand::**SetOperand**(*Operand* *opn*, *Constant* *imm*)
Set the operand opn of the instruction as the immediate imm.

Parameters

- opn: *Operand* index in the LLVM MCInst representation.
- imm: *Constant* which will be set as the new immediate operand.

QBDI::AddOperand::**AddOperand**(*Operand* *opn*, *Temp* *temp*)
Add a new temporary register operand to the instruction by inserting it at operand index opn.

Parameters

- opn: *Operand* index in LLVM MCInst representation.
- temp: *Temp* to be inserted as a new operand.

QBDI::AddOperand::**AddOperand**(*Operand* *opn*, *Reg* *reg*)
Add a new register operand to the instruction by inserting it at operand index opn.

Parameters

- opn: *Operand* index in LLVM MCInst representation.
- reg: Register to be inserted as a new operand.

QBDI::AddOperand::**AddOperand**(*Operand* *opn*, *Constant* *imm*)
Add a new immediate operand to the instruction by inserting it at operand index opn.

Parameters

- opn: *Operand* index in LLVM MCInst representation.
- imm: *Constant* to be inserted as a new immediate operand.

QBDI::SubstituteWithTemp::**SubstituteWithTemp**(*Reg* *reg*, *Temp* *temp*)
Substitute every reference to reg in the operands of the instruction with temp.

Parameters

- reg: Register which will be substituted.
- temp: Temporary register which will be substituted with.

QBDI::RemoveOperand::**RemoveOperand**(*Reg* *reg*)
Remove the first occurence of reg in the operands of the instruction.

Parameters

- reg: Register to remove from the operand list.

`QBDI::SetOpcode::`**`SetOpcode`**(unsigned int *opcode*)
Set the opcode of the instruction.

Parameters

- `opcode`: New opcode to set as the instruction opcode.

Logging System

LogSys is a singleton controlling the logging output of QBDI (see `src/Utility/LogSys.h`). It relies on a tag and priority filtering system to provide a fine-grained control of the output. Indeed QBDI logging system is quite verbose, detailing at every step and every action taken by the Engine.

Each log entry is assigned a tag describing the component it comes from and a priority describing its importance. The tags follow the convention `Class::Method` or just `function`.

The methods generating the most important logs are:

- `Engine::run` for everything related to execution decisions.
- `Engine::patch` for patch generations.
- `Engine::instrument` for instrumentation generations.
- `ExecBlockManager::getProgrammedExecBlock` for code cache lookup.
- `ExecBlock::writeSequence` for JIT code generation.
- `ExecBlock::execute` for sequence execution and callback execution.

The priority comes in three different levels:

`enum LogPriority`
Each log has a priority (or level) which can be used to control verbosity. In production builds, only Warning and Error logs are kept.

Values:

`QBDI_DEBUG` = 0
Debug logs

`QBDI_WARNING`
Warning logs

`QBDI_ERROR`
Error logs

> **Warning:** It is important to note that `Debug` level logging is only enabled in debug build. It can, however, be manually re-enabled in release build, please look into `src/Utility/LogSys.h` for more information.

Controlling Log Output

The log filters act as a whitelist of which logs to output. By default there are no filters and thus no output. A filter can be added by calling the `addLogFilter()`.

void `QBDI::`**`addLogFilter`**(**`const`** char **tag*, *LogPriority priority*)
Enable logs matching tag and priority.

Parameters

- `tag`: Logs are identified using a tag (ex: Engine::patch). "*" will log everything matching priority.
- `priority`: Filter logs with greater or equal priority.

The filters specify the minimum log priority to output for every entry matching the tag specified. For example, a filter with a tag `Engine::patch` and priority `Warning` would output all logs with priority `Warning` and `Error` coming from `Engine::patch`. The tag parameter is actually fuzzy matched so `Engine::*` would match everything coming out of the engine. A good default filter to have to ensure that all errors and warnings are reported is:

```
QBDI::addLogFilter("*", QBDI::LogPriority::Warning);
```

By default logs are output to `stderr` but this can be changed with `setLogOutput()`.

void `QBDI::`**`setLogOutput`**(FILE *_output_)
Redirect logs to an opened file.

Parameters

- `output`: Pointer to an opened file where logs will be append.

For example, redirecting the logs to a file can be done with the line below:

```
FILE* logFile = fopen(".qbdi_log", "w");
QBDI::setLogOutput(logFile);
```

Adding Log Output

The log system can only be used inside C++ source code. It relies on using the *`QBDI::LOGSYS`* singleton to log data using the *`QBDI::LogSys::log()`* and *`QBDI::LogSys::logCallback()`*. The later is useful in the case the computation required to generate the text of the log output is costly: the callback is only called in the case the tag and priority was matched by a filter.

QBDI::LogSys `QBDI::`**`LOGSYS`**
Global logging system singleton.

void `QBDI::LogSys::`**`log`**(*LogPriority priority*, **`const`** char *_tag_, **`const`** char *_fmt_, ...)
Output a formatted log entry with the designated priority and tag. The formatted output follow the fprintf syntax.

Parameters

- `priority`: This log entry priority.
- `tag`: This log entry tag.
- `fmt`: This log entry format string followed by the formatting argument.

void `QBDI::LogSys::`**`logCallback`**(*LogPriority priority*, **`const`** char *_tag_, std::function<void) FILE *log
> *callback*Output a formatted log entry with the designated priority and tag. This version takes a callback in parameter which is called if the tag and priority are matched by a filter and is in charge of printing the log entry. The tag of the log line and the line ending is handled by logsys.

Parameters

- `priority`: This log entry priority.
- `tag`: This log entry tag.
- `callback`: This log entry callback. It can be a function or a closure.

It is, however, not recommended to use those methods directly and use our macro system which controls the presence of logging in release builds.

LogCallback (p, t, c)
: Shortcut macro for `logCallback`. Disabled in release.

LogDebug (t, ...)
: Shortcut macro for `log(Debug, ...)`. Disabled in release.

LogWarning (t, ...)
: Shortcut macro for `log(Warning, ...)`. Enabled in release.

LogError (t, ...)
: Shortcut macro for `log(Error, ...)`. Enabled in release.

Below is an example of adding (very verbose) logging to a simple source code:

```
bool appendToFile(const char* fileName, const char* toAppend) {
    FILE* f = nullptr;
    size_t len = strlen(toAppend);

    LogDebug("appendToFile", "Appending %s to file %s", toAppend, fileName);

    if(len == 0) {
        LogWarning("appendToFile", "toAppend is empty!");
        return true;
    }

    f = fopen(fileName, "a");
    if(f == nullptr) {
        LogError("appendToFile", "Failed to open file %s", fileName);
        return false;
    }

    if(fwrite((const void*) toAppend, len, 1, f) != len) {
        LogError("appendToFile", "Failed to append to file %s", fileName);
        fclose(f);
        return false;
    }

    fclose(f);
    return true;
}
```

2.3 CHANGELOG

2.3.1 Next Version

2.3.2 Version 0.7.1

2020-02-27 QBDI Team <qbdi@quarkslab.com>

- Refactor PyQBDI, support python3, PyQBDI without Preload (#67, #121)
- Remove ncurses dependency (#123)
- Fix initFPRState (#114)

2.3.3 Version 0.7.0

2019-09-10 QBDI Team <qbdi@quarkslab.com>

- Add support for the x86 architecture
- Add new platforms related to Android: android-X86 and android-X86_64
- Improve `MemoryMap` structure by adding the module's full path if available (#62, #71)
- Create docker images for QBDI (available on DockerHub qbdi/qbdi) (#56)
- Fix and improve operands analysis involved in memory accesses (#58) :

  In the previous version, the output of the instruction analysis for **some** instructions did not contain the information related to memory accesses.

  For instance, the *operand analysis* of `cmp MEM, IMM` misses information about the first operand:

```
cmp dword ptr [rbp + 4 * rbx - 4], 12345678
    [0] optype: 1, value : 12345678, size: 8, regOff: 0, regCtxIdx: 0, regName:␣
↪(null), regaccess : 0
```

  This issue has been fixed and the `OperandAnalysis` structure contains a new attribute: `flag`, which is used to distinct `OperandAnalysis` involved in memory accesses from the others.

  Here is an example of output:

```
cmp dword ptr [rbp + 4*rbx - 4], 12345678
    [0] optype: 2, flag: 1, value : 48, size: 8, regOff: 0, regCtxIdx: 14,␣
↪regName: RBP, regaccess : 1
    [1] optype: 1, flag: 1, value : 4, size: 8, regOff: 0, regCtxIdx: 0, regName:␣
↪(null), regaccess : 0
    [2] optype: 2, flag: 1, value : 49, size: 8, regOff: 0, regCtxIdx: 1,␣
↪regName: RBX, regaccess : 1
    [3] optype: 1, flag: 1, value : -4, size: 8, regOff: 0, regCtxIdx: 0,␣
↪regName: (null), regaccess : 0
    [4] optype: 1, flag: 0, value : 12345678, size: 4, regOff: 0, regCtxIdx: 0,␣
↪regName: (null), regaccess : 0
mov rax, qword ptr [rbp - 4]
    [0] optype: 2, flag: 0, value : 47, size: 8, regOff: 0, regCtxIdx: 0,␣
↪regName: RAX, regaccess : 2
    [1] optype: 2, flag: 1, value : 48, size: 8, regOff: 0, regCtxIdx: 14,␣
↪regName: RBP, regaccess : 1
    [2] optype: 1, flag: 1, value : 1, size: 8, regOff: 0, regCtxIdx: 0, regName:␣
↪(null), regaccess : 0
    [3] optype: 1, flag: 1, value : -4, size: 8, regOff: 0, regCtxIdx: 0,␣
↪regName: (null), regaccess : 0
mov rax, qword ptr [4*rbx]
    [0] optype: 2, flag: 0, value : 47, size: 8, regOff: 0, regCtxIdx: 0,␣
↪regName: RAX, regaccess : 2
    [1] optype: 1, flag: 1, value : 4, size: 8, regOff: 0, regCtxIdx: 0, regName:␣
↪(null), regaccess : 0
    [2] optype: 2, flag: 1, value : 49, size: 8, regOff: 0, regCtxIdx: 1,␣
↪regName: RBX, regaccess : 1
    [3] optype: 1, flag: 1, value : 0, size: 8, regOff: 0, regCtxIdx: 0, regName:␣
↪(null), regaccess : 0
jne -6115
    [0] optype: 1, flag: 2, value : -6115, size: 4, regOff: 0, regCtxIdx: 0,␣
↪regName: (null), regaccess : 0
lea rax, [rbp + 4*rbx - 4]
```

```
    [0] optype: 2, flag: 0, value : 47, size: 8, regOff: 0, regCtxIdx: 0,␣
↪regName: RAX, regaccess : 2
    [1] optype: 2, flag: 4, value : 48, size: 8, regOff: 0, regCtxIdx: 14,␣
↪regName: RBP, regaccess : 1
    [2] optype: 1, flag: 4, value : 4, size: 8, regOff: 0, regCtxIdx: 0, regName:␣
↪(null), regaccess : 0
    [3] optype: 2, flag: 4, value : 49, size: 8, regOff: 0, regCtxIdx: 1,␣
↪regName: RBX, regaccess : 1
    [4] optype: 1, flag: 4, value : -4, size: 8, regOff: 0, regCtxIdx: 0,␣
↪regName: (null), regaccess : 0
```

2.3.4 Version 0.6.2

2018-10-19 Cedric TESSIER <ctessier@quarkslab.com>

- Add support for a public CI (based on Travis and AppVeyor)
- Fix instruction operands analysis (#57, #59)
- Add missing MEMORY_READ enum value in Python bindings (#61)
- Fix cache misbehavior on corner cases (#49, #51)
- Add missing memory access instructions on x86_64 (#45, #47, #72)
- Enable asserts in Debug builds (#48)

2.3.5 Version 0.6.1

2018-03-22 Charles HUBAIN <chubain@quarkslab.com>

- Fixing a performance regression with the addCodeAddrCB (#42):

  Since 0.6, this API would trigger a complete cache flush forcing the engine to regenerate all the instrumented code after each call. Since this API is used inside VM:run(), this had the effect of completely canceling pre-caching optimization where used.

- Fixing support for AVX host without AVX2 support (#19):

  Context switching was wrongly using AVX2 instructions instead of AVX instructions causing segfaults under hosts supporting AVX but not AVX2.

2.3.6 Version 0.6

2018-03-02 Charles HUBAIN <chubain@quarkslab.com>

- Important performance improvement in the core engine (#30) **This slightly changes the behavior of VMEvents.**
- Fix the addCodeAddrCB API (#37)
- atexit and getCurrentProcessMap in python bindings (#35)
- Fix getInstAnalysis on BASIC_BLOCK_ENTRY (#28)
- Various documentation improvements (#34, #37, #38, #40) and an API uniformisation (#29)

2.3.7 Version 0.5

2017-12-22 Cedric TESSIER <ctessier@quarkslab.com>

- Official public release!

2.3.8 Version 0.5 RC3

2017-12-10 Cedric TESSIER <ctessier@quarkslab.com>

- Introducing pyqbdi, full featured python bindings based on QBDIPreload library
- Revising variadic API to include more friendly prototypes
- Various bug, compilation and documentation fixes

2.3.9 Version 0.5 RC2

2017-10-30 Charles HUBAIN <chubain@quarkslab.com>

- Apache 2 licensing
- New QBDIPreload library for easier dynamic injection under linux and macOS
- Various bug, compilation and documentation fixes
- Big tree cleanup

2.3.10 Version 0.5 RC1

2017-10-09 Charles HUBAIN <chubain@quarkslab.com>

- New Frida bindings
- Upgrade to LLVM 5.0
- Support for AVX registers
- New callback helpers on mnemonics and memory accesses
- Basic block precaching API
- Automatic cache invalidation when a new instrumentation is added
- Instruction and sequence level cache avoids needless retranslation
- Upgrade of the validator which now supports Linux and macOS

2.3.11 Version 0.4

2017-01-06 Charles HUBAIN <chubain@quarkslab.com>

- Basic Instruction Shadows concept
- Memory access PatchDSL statements with support under X86_64 (non SIMD memory access only)
- Shadow based memory access API and instrumentation
- C and C++ API stabilization
- Out-of-tree build and SDK

- Overhaul of the entire documentation with a complete PatchDSL explanation and a split between user and developer documentation.

2.3.12 Version 0.3

2016-04-29 Charles HUBAIN <chubain@quarkslab.com>

- Partial ARM support, sufficient to run simple program e.g cat, ls, ...
- Instrumentation filtering system, ExecBroker, allowing the engine to switch between non instrumented and instrumented execution
- Complex execution validation system under linux which allows to do instruction per instruction compared execution between a non instrumented and an instrumented instance of a program
- New callback system for Engine related event e.g basic block entry / exit, ExecBroker transfer / return.
- New (internal) logging system, LogSys, which allows to do priority and tag based filtering of the debug logs.

2.3.13 Version 0.2

2016-01-29 Charles HUBAIN <chubain@quarkslab.com>

- Upgrade to LLVM 3.7
- Complete X86_64 patching support
- Support of Windows X86_64
- Basic callback based instrumentation
- Usable C++ and C API
- User documentation with examples
- Uniformisation of PatchDSL

2.3.14 Version 0.1

2015-10-09 Charles HUBAIN <chubain@quarkslab.com>

- Ported the PatchDSL from the minijit PoC
- Corrected several design flaws in the PatchDSL
- Implemented a comparated execution test setup to prove the execution via the JIT yields the same registers and stack state as a normal execution
- Basic patching working for ARM and X86_64 architectures as shown by the compared execution tests

2.3.15 Version 0.0

2015-09-17 Charles HUBAIN <chubain@quarkslab.com>

- Working dependency system for LLVM and Google Test
- ExecBlock working and tested on linux-X86_64, linux-ARM, android-ARM and macOS-X86_64
- Deployed buildbot infrastructure for automated build and test on linux-X86_64 and linux-ARM

CHAPTER 3

Indices and Tables

- genindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y