QBDI Documentation

Release 0.11.0

Quarkslab

May 17, 2024

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. In addition of C/C++ API, Python and JS/frida bindings are available to script QBDI. Information about what is a DBI framework and how QBDI works can be found in the documentation introduction.

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). QBDI is also fully integrated with Frida, a reference dynamic instrumentation toolkit, allowing anybody to use their combined powers.

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

ONE

STATUS

CPU	Operating Systems	Execution	Memory Access Information
x86-64	Android, Linux, macOS, Windows	Supported	Supported
x86	Android, Linux, macOS, Windows	Supported	Supported
ARM	Android, Linux	Supported (*)	Supported (*)
AArch64	Android, Linux, macOS	Supported (*)	Supported (*)

* The ARM and AArch64 instruction sets are supported but in early support.

stable

dev

CHAPTER

TWO

CONTENTS

2.1 Introduction

2.1.1 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.

2.1.2 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.

2.1.3 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.

2.1.4 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 the *API description*.

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.

2.2 Installation and Integration

2.2.1 C/C++ API installation

When a new stable version of QBDI is released, prebuilt packages can be downloaded through the release page on Github. In order to make sure you have downloaded the right package, you can verify its integrity with the file `SHA256`.

A GPG signature of `SHA256` by our developer key (2763 2215 DED8 D717 AD08 477D 874D 3F16 4D45 2193) is available in `SHA256.sig`.

```
# check the hash of the prebuilt package
sha256 -c SHA256

# verify the signature
wget https://qbdi.quarkslab.com/qbdi.asc -O - | gpg --import -
gpg --verify SHA256.sig
```

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

Arch Linux packages can be installed using `pacman`:

```
pacman -U QBDI-*-*-X86_64.tar.zst
```

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.

Android

The Android package is an archive you solely need to extract. Afterwards, you have to manually push the files onto the device.

Devel packages

Devel packages embed the latest features the developers are currently working on for the next release (available on the dev-next branch). It's worth mentioning that since some parts are still under development, those are likely to be **unstable** – you must be aware that it may contain some bugs and are not as reliable as release packages.

- Windows devel packages
- Linux (Ubuntu) devel packages
- Android devel packages
- OSX devel packages

2.2.2 PyQBDI installation

Every time a new stable release of PyQBDI is available, it is automatically pushed onto the PyPI platform, thus can be easily installed with `pip` (>= 19.3).

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

If you want to use a devel version, download the corresponding prebuilt wheel file and run the following commands:

```
pip install --user --update pip
pip install --user PyQBDI-*.whl
```

The devel wheel files which contain the latest versions of the dev-next branch are available at:

- PyQBDI for Windows
- PyQBDI for Linux

- PyQBDI for OSX

Note: Only Python3 is supported. If you need to use Python2, we recommend using QBDI 0.7.0 instead.

Note: A 32-bit version of Python is needed if you want to use PyQBDI on x86 targets.

2.2.3 Frida/QBDI installation

QBDI can be used alongside Frida to make it even more powerful. This feature is included in the C/C++ package. Using it requires having Frida installed (>= 14.0) on your workstation as well as frida-compile for compiling your scripts.

```
# install Frida
pip install frida-tools

# install frida-compile and add the binary directory to your PATH env variable
npm install frida-compile babelify
export PATH=$PATH:$(pwd)/node_modules/.bin
```

Android target

In order to use Frida/QBDI on an Android device, the Frida server must be running on the target device and the `libQBDI.so` library have to be placed in `/data/local/tmp`.

Note: Latest Frida server binaries are available on the Frida official release page.

2.2.4 Docker Linux images

The Docker image is available on Docker Hub. It has been built to keep it as small as possible so it does not contain any compiler. You have to install the needed application or modify the following Dockerfile according to your needs.

```
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 \
        python \
        python-dev \
        #gdb \
        #vim \
```

(continues on next page)

(continued from previous page)

```
        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 which is mandatory to use QBDIPreload.

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

2.2.5 Compilation from source code

To build this project, the following dependencies are needed on your system:

- cmake >= 3.12
- ninja or make
- C++17 toolchain (gcc, clang, Visual Studio 2019, ...)

A local version of llvm is statically built within QBDI 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.

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 mode 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/config-linux-X86_64.sh
ninja
```

x86

You can follow the same instructions as for x86-64 but instead, use the `config-linux-X86.sh` configuration script.

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 ninja`)

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/config-macOS-X86_64.sh
ninja
```

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 and Ninja.

First of all, the Visual Studio environment must be set up. This can be done with a command such as:

```
"C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build\
↪vcvarsall.bat" x64
```

Then, the following commands must be run:

```
mkdir build
cd build
python ../cmake/config/config-win-X86_64.py
ninja
```

Android

Cross-compiling for Android requires the NDK (or the SDK) to be installed on your workstation. For now, it has only been tested under Linux. If not already installed, you can download the latest Android NDK package through the official website and extract it. Afterwards, the `config-android-*.sh` configuration script needs to be customised to match your NDK installation directory and the target platform.:

```
# Configure and compile QBDI X86_64 with a NDK
mkdir build && cd build
NDK_PATH=<your_NDK_PATH> ../cmake/config/config-android-X86_64.sh
```

(continues on next page)

(continued from previous page)

```
ninja

# Configure and compile QBDI X86 with a SDK
mkdir build && cd build
ANDROID_SDK_ROOT=<your_SDK_PATH> ../cmake/config/config-android-X86.sh
ninja
```

PyQBDI compilation

The PyQDBI library (apart from the wheel package) can be built by solely passing the **'-DQBDI_TOOLS_PYQBDI=ON'** option to the CMake build system.

However, if you want to build the wheel package, you can run these commands:

```
python -m pip install --upgrade pip
python -m pip install setuptools wheel build
python -m build -w
```

A 32-bit version of Python is mandatory for the X86 architecture whereas a 64-bit one is required for the X86-64 architecture.

2.2.6 CMake integration

Single architecture

If you want to use only one architecture of QBDI in your CMake project, you can import the `QBDI` and `QBDIPreload` packages:

```
find_package(QBDI REQUIRED)
find_package(QBDIPreload REQUIRED) # if available for your current platform
# or
find_package(QBDI REQUIRED HINTS "${EXTRACT_DIRECTORY}" NO_DEFAULT_PATH)
find_package(QBDIPreload REQUIRED HINTS "${EXTRACT_DIRECTORY}" NO_DEFAULT_PATH)
```

Once the CMake package is found, you can link your executable either with the dynamic or the static library:

```
add_executable(example example.c)

target_link_libraries(example QBDI::QBDI)
# or
target_link_libraries(example QBDI::QBDI_static)

add_executable(example_preload example_preload.c)
target_link_libraries(example_preload QBDI::QBDI_static QBDIPreload::QBDIPreload)
```

Multi-architecture

If two or more architectures of QBDI are needed within the same project, you should import the package `QBDI<arch>` and `QBDIPreload<arch>`.

```
find_package(QBDIX86 REQUIRED HINTS "${EXTRACT_DIRECTORY_X86}" NO_DEFAULT_PATH)
find_package(QBDIPreloadX86 REQUIRED HINTS "${EXTRACT_DIRECTORY_X86}" NO_DEFAULT_PATH)

add_executable(example_preload86 example_preload.c)
set_target_properties(example_preload86 PROPERTIES COMPILE_FLAGS "-m32" LINK_FLAGS "-m32
↪")
target_link_libraries(example_preload86 QBDI::X86::QBDI_static␣
↪QBDIPreload::X86::QBDIPreload)

find_package(QBDIX86_64 REQUIRED HINTS "${EXTRACT_DIRECTORY_X86_64}" NO_DEFAULT_PATH)
find_package(QBDIPreloadX86_64 REQUIRED HINTS "${EXTRACT_DIRECTORY_X86_64}" NO_DEFAULT_
↪PATH)

add_executable(example_preload64 example_preload.c)
target_link_libraries(example_preload64 QBDI::X86_64::QBDI_static QBDIPreload::X86_
↪64::QBDIPreload)
```

2.3 Getting Started

The following sections explain how one can take advantage of QBDI through different use cases.

2.3.1 C API

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

Load the target code

In this tutorial, we aim at figuring out how many iterations a Fibonacci function is doing. To do so, we will rely on QBDI to instrument the function. For convenience sake, its source code is compiled along with the one we are about to write.

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

However, it's not always the case. Sometimes, we need to look into a function we don't have the source code of – that is, it has been already compiled. As a result, we have to find a way to load the code we want to inspect into our process' memory space. For instance, if the function of interest is embedded in a dynamic library, we can link our code with this library when compiling or import it at runtime by calling either `dlopen` or `LoadLibraryA`.

Note that if you want to instrument a whole binary, QBDIPreload should be preferred (see *QBDIPreload in C*).

Initialise the virtual machine

First off, we need to initialise the virtual machine (often referred to as VM) itself. The type `VMInstanceRef` represents an instance of the VM. The *qbdi_initVM()* function needs to be called to set the instance up. The second, third and fourth arguments are used to customise the instance depending on what you want to do.

```
#include "QBDI.h"

VMInstanceRef vm;
qbdi_initVM(&vm, NULL, NULL, 0);
```

Retrieve the VM context

Prior to initialising the virtual stack (see next section), we need to get a pointer to the virtual machine state (*GPRState*), which can be obtained by calling *qbdi_getGPRState()*. This object represents the current VM's context.

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

Allocate a virtual stack

The virtual machine does not work with the regular stack that your process uses – instead, QBDI needs its own stack. Therefore, we have to ask for a virtual stack using *qbdi_allocateVirtualStack()*. This function is responsible for allocating an aligned memory space, set the stack pointer register accordingly and return the top address of this brand-new memory region.

```
uint8_t* fakestack;
bool res = qbdi_allocateVirtualStack(state, STACK_SIZE, &fakestack);
assert(res == true);
```

Write our first callback function

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

To have a trace of our execution, we will need a callback that will retrieve the current address and the disassembly of the instruction and print it.

As the callback will be called on an instruction, the callback must follow the *InstCallback* type. Inside the callback, we can get an *InstAnalysis* of the current instruction with *qbdi_getInstAnalysis()*. To have the address and the disassembly, the *InstAnalysis* needs to have the type *QBDI_ANALYSIS_INSTRUCTION* (for the address) and *QBDI_ANALYSIS_DISASSEMBLY* (for the disassembly).

```
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);
```

(continues on next page)

(continued from previous page)

```
    return QBDI_CONTINUE;
}
```

An *InstCallback* must always return an action (*VMAction*) to the VM to specify if the execution should continue or stop. In most cases *QBDI_CONTINUE* should be returned to continue the execution.

Register a callback

The callback must be registered in the VM. The function *qbdi_addCodeCB()* allows registering a callback for every instruction. The callback can be called before the instruction (*QBDI_PREINST*) or after the instruction (*QBDI_POSTINST*).

```
uint32_t cid = qbdi_addCodeCB(vm, QBDI_PREINST, showInstruction, NULL, 0);
assert(cid != QBDI_INVALID_EVENTID);
```

The function returns a callback ID or the special ID *QBDI_INVALID_EVENTID* if the registration fails. The callback ID can be kept if you want to unregister the callback later.

Count the iterations

With the current implementation of Fibonacci, the function will iterate by recursively calling itself. Consequently, we can determine the number of iterations the function is doing by counting the number of calls. *qbdi_addMnemonicCB()* can be used to register a callback which is solely called when encountering specific instructions. All QBDI callbacks allow users to pass a custom parameter `data` of type `void *`.

```
VMAction countIteration(VMInstanceRef vm, GPRState *gprState, FPRState *fprState, void␣
↪*data) {
    (*((unsigned*) data))++;

    return QBDI_CONTINUE;
}

unsigned iterationCount = 0;
qbdi_addMnemonicCB(vm, "CALL*", QBDI_PREINST, countIteration, &iterationCount, 0);
```

Set instrumented ranges

QBDI needs a range of addresses where the code should be instrumented. If the execution goes out of this scope, QBDI will try to restore an uninstrumented execution.

In our example, we need to include the function we are looking into in the instrumented range. *qbdi_addInstrumentedModuleFromAddr()* can be used to add a whole module (binary or library) in the range of instrumentation with a single address of this module.

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

Run the instrumentation

We can finally run the instrumentation using the *qbdi_call()* function. It aligns the stack, sets the argument(s) (if needed) and a fake return address and calls the target function through QBDI. The execution stops when the instrumented code returns to the fake address.

```
rword retval;
res = qbdi_call(vm, &retval, (rword) fibonacci, 1, 25);
assert(res == true);
```

qbdi_call() returns if the function has completely run in the context of QBDI. The first argument has been filled with the value of the return register (e.g. RAX for X86_64).

It may turn out that the function does not expect the calling convention *qbdi_call()* uses. In this precise case, you must set up the proper context and the stack yourself and call *qbdi_run()* afterwards.

Terminate the execution properly

At last, before exiting, we need to free up the resources we have allocated: both the virtual stack and the virtual machine by respectively calling *qbdi_alignedFree()* and *qbdi_terminateVM()*.

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

Full example

Merging everything we have learnt throughout this tutorial, we are now able to write our C source code file:

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

#include "QBDI.h"

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

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;
```

(continues on next page)

(continued from previous page)

```
}

VMAction countIteration(VMInstanceRef vm, GPRState *gprState,
                        FPRState *fprState, void *data) {
  (*((int *)data))++;

  return QBDI_CONTINUE;
}

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

int main(int argc, char **argv) {
  int n = 0;

  int iterationCount = 0;
  uint8_t *fakestack;
  VMInstanceRef vm;
  GPRState *state;
  rword retvalue;

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

  // Constructing a new QBDI VM
  qbdi_initVM(&vm, NULL, NULL, 0);

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

  // Setup initial GPR state,
  qbdi_allocateVirtualStack(state, STACK_SIZE, &fakestack);

  // Registering showInstruction() callback to print a trace of the execution
  uint32_t cid = qbdi_addCodeCB(vm, QBDI_PREINST, showInstruction, NULL, 0);
  assert(cid != QBDI_INVALID_EVENTID);

  // Registering countIteration() callback
  qbdi_addMnemonicCB(vm, "CALL*", QBDI_PREINST, countIteration, &iterationCount,
                     0);

  // Setup Instrumentation Range
  bool res = qbdi_addInstrumentedModuleFromAddr(vm, (rword)&fibonacci);
  assert(res == true);

  // Running DBI execution
  printf("Running fibonacci(%d) ...\n", n);
  res = qbdi_call(vm, &retvalue, (rword)fibonacci, 1, n);
```

(continues on next page)

(continued from previous page)

```
  assert(res == true);

  printf("fibonnaci(%d) returns %ld after %d recursions\n", n, retvalue,
         iterationCount);

  // cleanup
  qbdi_alignedFree(fakestack);
  qbdi_terminateVM(vm);

  return 0;
}
```

Generate a template

A QBDI template can be considered as a baseline project, a minimal component you can modify and build your instrumentation tool on. They are provided to help you effortlessly start off a new QBDI based project. The binary responsible for generating a template is shipped in the release packages and can be used as follows:

```
mkdir new_project
cd new_project
qbdi-template
```

A template consists of a simple C source code file and a basic CMake build script. A file called `README.txt` is also present, it describes the compilation procedure.

2.3.2 C++ API

A step-by-step example illustrating a basic (yet powerful) usage of the QBDI C++ API.

Load the target code

In this tutorial, we aim at figuring out how many iterations a Fibonacci function is doing. To do so, we will rely on QBDI to instrument the function. For convenience sake, its source code is compiled along with the one we are about to write.

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

However, it's not always the case. Sometimes, we need to look into a function we don't have the source code of – that is, it has been already compiled. As a result, we have to find a way to load the code we want to inspect into our process' memory space. For instance, if the function of interest is embedded in a dynamic library, we can link our code with this library when compiling or import it at runtime by calling either `dlopen` or `LoadLibraryA`.

Note that if you want to instrument a whole binary, QBDIPreload should be preferred (see *QBDIPreload in C++*).

Initialise the virtual machine

First off, we need to initialise the virtual machine (*QBDI::VM*) itself.

```
#include "QBDI.h"

QBDI::VM vm;
```

Retrieve the VM context

Prior to initialising the virtual stack (see next section), we need to get a pointer to the virtual machine state (*QBDI::GPRState*), which can be obtained by calling *QBDI::VM::getGPRState()*. This object represents the current VM's context.

```
QBDI::GPRState *state = vm.getGPRState();
assert(state != NULL);
```

Allocate a virtual stack

The virtual machine does not work with the regular stack that your process uses – instead, QBDI needs its own stack. Therefore, we have to ask for a virtual stack using *QBDI::allocateVirtualStack()*. This function is responsible for allocating an aligned memory space, set the stack pointer register accordingly and return the top address of this brand-new memory region.

```
uint8_t* fakestack;
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.

To have a trace of our execution, we will need a callback that will retrieve the current address and the disassembly of the instruction and print it.

As the callback will be called on an instruction, the callback must follow the *QBDI::InstCallback* type. Inside the callback, we can get an *QBDI::InstAnalysis* of the current instruction with *QBDI::VM::getInstAnalysis()*. To have the address and the disassembly, the *QBDI::InstAnalysis* need to have the type *QBDI::AnalysisType::ANALYSIS_INSTRUCTION* (for the address) and *QBDI::AnalysisType::ANALYSIS_DISASSEMBLY* (for the disassembly). These two *QBDI::AnalysisType* are the default parameter of *QBDI::VM::getInstAnalysis()* and can be omitted.

```
QBDI::VMAction showInstruction(QBDI::VM* vm, QBDI::GPRState *gprState, QBDI::FPRState␣
↪*fprState, void *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);
```

(continues on next page)

(continued from previous page)

```
    return QBDI::VMAction::CONTINUE;
}
```

An *QBDI::InstCallback* must always return an action (*QBDI::VMAction*) to the VM to specify if the execution should continue or stop. In most cases *QBDI::VMAction::CONTINUE* should be returned to continue the execution.

Register a callback

The callback must be registered in the VM. The function *QBDI::VM::addCodeCB()* allows registering a callback for every instruction. The callback can be called before the instruction (*QBDI::InstPosition::PREINST*) or after the instruction (*QBDI::InstPosition::POSTINST*).

```
uint32_t cid = vm.addCodeCB(QBDI::PREINST, showInstruction, nullptr);
assert(cid != QBDI::INVALID_EVENTID);
```

The function returns a callback ID or the special ID *QBDI::VMError::INVALID_EVENTID* if the registration fails. The callback ID can be kept if you want to unregister the callback later.

Count the iterations

With the current implementation of Fibonacci, the function will iterate by recursively calling itself. Consequently, we can determine the number of iterations the function is doing by counting the number of calls. *QBDI::VM::addMnemonicCB()* can be used to register a callback which is solely called when encountering specific instructions. All QBDI callbacks allow users to pass a custom parameter `data` of type `void *`.

```
QBDI::VMAction countIteration(QBDI::VMInstanceRef vm, QBDI::GPRState *gprState,␣
↪QBDI::FPRState *fprState, void *data) {
    (*((unsigned*) data))++;

    return QBDI::CONTINUE;
}

unsigned iterationCount = 0;
vm.addMnemonicCB("CALL*", QBDI::PREINST, countIteration, &iterationCount);
```

Set instrumented ranges

QBDI needs a range of addresses where the code should be instrumented. If the execution goes out of this scope, QBDI will try to restore an uninstrumented execution.

In our example, we need to include the method in the instrumented range. The method *QBDI::VM::addInstrumentedModuleFromAddr()* can be used to add a whole module (binary or library) in the range of instrumentation with a single address of this module.

```
res = vm.addInstrumentedModuleFromAddr(reinterpret_cast<QBDI::rword>(fibonacci));
assert(res == true);
```

Run the instrumentation

We can finally run the instrumentation using the *QBDI::VM::call()* function. It aligns the stack, sets the argument(s) (if needed) and a fake return address and calls the target function through QBDI. The execution stops when the instrumented code returns to the fake address.

```
rword retval;
res = vm.call(&retval, reinterpret_cast<QBDI::rword>(fibonacci), {25});
assert(res == true);
```

QBDI::VM::call() returns if the function has completely run in the context of QBDI. The first argument has been filled with the value of the return register (e.g. `RAX` for X86_64).

It may turn out that the function does not expect the calling convention *QBDI::VM::call()* uses. In this precise case, you must set up the proper context and the stack yourself and call *QBDI::VM::run()* afterwards.

Terminate the execution properly

At last, before exiting, we need to free up the virtual stack we have allocated calling *QBDI::alignedFree()*.

```
QBDI::alignedFree(fakestack);
```

Full example

Merging everything we have learnt throughout this tutorial, we are now able to write our C++ source code file:

```
#include <assert.h>
#include <cstdint>
#include <cstdlib>
#include <cstring>
#include <iomanip>
#include <iostream>

#include "QBDI.h"

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

QBDI::VMAction showInstruction(QBDI::VM *vm, QBDI::GPRState *gprState,
                               QBDI::FPRState *fprState, void *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);

  return QBDI::VMAction::CONTINUE;
```

(continues on next page)

(continued from previous page)

```
}

QBDI::VMAction countIteration(QBDI::VM *vm, QBDI::GPRState *gprState,
                              QBDI::FPRState *fprState, void *data) {
  (*((unsigned *)data))++;

  return QBDI::CONTINUE;
}

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

int main(int argc, char **argv) {
  int n = 0;
  bool res;
  uint32_t cid;

  unsigned iterationCount = 0;
  uint8_t *fakestack;
  QBDI::GPRState *state;
  QBDI::rword retvalue;

  // Argument to the fibonnaci call
  if (argc >= 2) {
    n = atoi(argv[1]);
  }
  if (n < 1) {
    n = 1;
  }

  // Constructing a new QBDI vm
  QBDI::VM vm{};

  // Get a pointer to the GPR state of the VM
  state = vm.getGPRState();
  assert(state != nullptr);

  // Setup initial GPR state,
  res = QBDI::allocateVirtualStack(state, STACK_SIZE, &fakestack);
  assert(res == true);

  // Registering showInstruction() callback to print a trace of the execution
  cid = vm.addCodeCB(QBDI::PREINST, showInstruction, nullptr);
  assert(cid != QBDI::INVALID_EVENTID);

  // Registering countIteration() callback
  vm.addMnemonicCB("CALL*", QBDI::PREINST, countIteration, &iterationCount);

  // Setup Instrumentation Range
  res = vm.addInstrumentedModuleFromAddr(
      reinterpret_cast<QBDI::rword>(fibonacci));
  assert(res == true);
```

(continues on next page)

(continued from previous page)

```
  // Running DBI execution
  std::cout << "Running fibonacci(" << n << ") ..." << std::endl;
  res = vm.call(&retvalue, reinterpret_cast<QBDI::rword>(fibonacci),
                {static_cast<QBDI::rword>(n)});
  assert(res == true);

  std::cout << "fibonnaci(" << n << ") returns " << retvalue << " after "
            << iterationCount << " recursions" << std::endl;

  QBDI::alignedFree(fakestack);

  return 0;
}
```

2.3.3 PyQBDI

PyQBDI brings Python3 bindings over the QBDI API. That way, you can take advantage of the QBDI features directly from your Python scripts without bothering using C/C++. It may be pretty useful if you need to build something quickly. However, it introduces some limitations:

- PyQBDI cannot be used to instrument a Python process
- The performances are poorer than when using the C/C++ APIs
- The Python runtime's and the target's architectures must be the same

Memory allocation

Unlike the C/C++ APIs, interacting with the process' memory is much more complicated while in Python – that is, memory regions cannot be allocated, read or written. Luckily, PyQBDI offers helpers to allow users perform these actions.

```
import pyqbdi

value = b"bytes array"

addr = pyqbdi.allocateMemory(len(value))
pyqbdi.writeMemory(addr, value)
value2 = pyqbdi.readMemory(addr, len(value))
assert value == value2
pyqbdi.freeMemory(addr)
```

Load the target code

In this tutorial, we aim at executing the `foo` function which lies in a shared library whose name is `mylib.so`, in the context of QBDI. PyQBDI will give us a hand doing so.

```
import pyqbdi
import ctypes

mylib = ctypes.cdll.LoadLibrary("mylib.so")
funcPtr = ctypes.cast(mylib.foo, ctypes.c_void_p).value
```

Note that if you want to instrument a whole binary, PyQBDIPreload should be preferred (see *PyQBDIPreload*).

Initialise the virtual machine

First off, we need to initialise the virtual machine (*VM*) itself. Calling the *pyqbdi.VM()* is needed to craft a new instance.

```
vm = pyqbdi.VM()
```

Allocate a virtual stack

The virtual machine does not work with the regular stack that your process uses – instead, QBDI needs its own stack. Therefore, we have to ask for a virtual stack using *pyqbdi.allocateVirtualStack()*. This function is responsible for allocating an aligned memory space, set the stack pointer register accordingly and return the top address of this brand-new memory region.

```
state = vm.getGPRState()
fakestack = pyqbdi.allocateVirtualStack(state, 0x100000)
assert fakestack != None
```

Write our first callback function

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

To have a trace of our execution, we will need a callback that will retrieve the current address and the disassembly of the instruction and print it.

As the callback will be called on an instruction, the callback must follow the *InstCallback* type. Inside the callback, we can get an *InstAnalysis* of the current instruction with *pyqbdi.VM.getInstAnalysis()*. To have the address and the disassembly, the *InstAnalysis* need to have the type *pyqbdi.ANALYSIS_INSTRUCTION* (for the address) and *pyqbdi.ANALYSIS_DISASSEMBLY* (for the disassembly). These two *pyqbdi.AnalysisType* are the default parameter of *pyqbdi.VM.getInstAnalysis()* and can be omitted.

```
def showInstruction(vm, gpr, fpr, data):
    # Obtain an analysis of the instruction from the VM
    instAnalysis = vm.getInstAnalysis()

    # Printing disassembly
    print("0x{:x}: {}".format(instAnalysis.address, instAnalysis.disassembly))

    return pyqbdi.CONTINUE
```

An *InstCallback* must always return an action (*pyqbdi.VMAction*) to the VM to specify if the execution should continue or stop. In most cases *CONTINUE* should be returned to continue the execution.

Register a callback

The callback must be registered in the VM. The function *pyqbdi.VM.addCodeCB()* allows registering a callback for every instruction. The callback can be called before the instruction (*pyqbdi.PREINST*) or after the instruction (*pyqbdi.POSTINST*).

```
cid = vm.addCodeCB(pyqbdi.PREINST, showInstruction, None)
assert cid != pyqbdi.INVALID_EVENTID
```

The function returns a callback ID or the special ID *pyqbdi.INVALID_EVENTID* if the registration fails. The callback ID can be kept if you want to unregister the callback later.

Set instrumented ranges

QBDI needs a range of addresses where the code should be instrumented. If the execution goes out of this scope, QBDI will try to restore an uninstrumented execution.

In our example, we need to include the method in the instrumented range. The method *pyqbdi.VM.addInstrumentedModuleFromAddr()* can be used to add a whole module (binary or library) in the range of instrumentation with a single address of this module.

```
assert vm.addInstrumentedModuleFromAddr(funcPtr)
```

Run the instrumentation

We can finally run the instrumentation using the *pyqbdi.VM.call()* function. It aligns the stack, sets the argument(s) (if needed) and a fake return address and calls the target function through QBDI. The execution stops when the instrumented code returns to the fake address.

```
asrun, retval = vm.call(funcPtr, [args1, args2])
assert asrun
```

pyqbdi.VM.call() returns if the function has completely run in the context of QBDI. The first argument has been filled with the value of the return register (e.g. `RAX` for X86_64).

It may turn out that the function does not expect the calling convention *pyqbdi.VM.call()* uses. In this precise case, you must set up the proper context and the stack yourself and call *pyqbdi.VM.run()* afterwards.

Terminate the execution properly

At last, before exiting, we need to free up the virtual stack we have allocated calling *pyqbdi.alignedFree()*.

```
pyqbdi.alignedFree(fakestack)
```

Full example

Merging everything we have learnt throughout this tutorial, we are now able to solve real problems. For instance, the following example shows how one can generate an execution trace of the `sin` function by using a PyQBDI script:

```
#!/usr/bin/env python3

import sys
import math
import ctypes
import pyqbdi
import struct

def insnCB(vm, gpr, fpr, data):
    instAnalysis = vm.getInstAnalysis()
    print("0x{:x}: {}".format(instAnalysis.address, instAnalysis.disassembly))
    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
    vm.addCodeCB(pyqbdi.PREINST, insnCB, 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)
```

(continues on next page)

(continued from previous page)

```
    success = vm.run(funcPtr, 0x42424242)

    # 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()
```

2.3.4 Frida/QBDI

QBDI can team up with Frida to be even more powerful together. To be able to use QBDI bindings while injected into a process with Frida, it is necessary to understand how to use Frida to perform some common tasks beforehand. Through this simple example based on *qbdi-frida-template* (see *Generate a template*), we will explain a basic usage of Frida and QBDI.

Common tasks

This section solely shows a few basic actions and gives you a general overview of what you can do with Frida. Keep in mind that Frida offers many more awesome features, all listed in the Javascript API documentation.

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-byte buffer
var buffer = Memory.alloc(2);
Memory.writeByteArray(buffer, [0x42, 0x42])
// allocate and write an UTF8 string
var str = Memory.allocUtf8String("Hello World !");
```

Initialise a VM object

If *frida-qbdi.js* (or a script requiring it) is successfully loaded in Frida, a new *VM()* object becomes available. It provides an object oriented access to the framework features.

```
// initialise QBDI
var vm = new VM();
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, further details about the exposed APIs are available in the *Frida/QBDI API* 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 the context
    console.log("0x" + inst.address.toString(16) + " " + inst.disassembly); // display␣
↪the instruction
    return VMAction.CONTINUE;
});
var iid = vm.addCodeCB(InstPosition.PREINST, instructionCallback, NULL);

vm.call(functionPtr, []);
```

If you ever want to pass custom arguments to your callback, this can be done via the **data** argument:

```
// this callback is used to count the number of basic blocks 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.

```
import { VM } from "./frida-qbdi.js";

var vm = new VM();
console.log("QBDI version is " + vm.version.string);
```

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*).

Compilation

In order to actually import QBDI bindings into your project, your script needs be *compiled* with the frida-compile utility. Installing it requires you to have `npm` installed. The babelify package might be also needed. Otherwise, you will not be able to successfully compile/load it and some errors will show up once running it with Frida.

Before running frida-compile, be sure that the script `frida-qbdi.js` is inside you current directory.

```
find <archive_extracted_path> -name frida-qbdi.js -exec cp {} . \;

# if frida-compile is not already installed
npm install frida-compile babelify
./node_modules/.bin/frida-compile MyScript.js -o MyScriptCompiled.js
# else
frida-compile MyScript.js -o MyScriptCompiled.js
```

Run Frida/QBDI on a workstation

To use QBDI on an already existing process you can use the following syntax:

```
frida -n processName -l MyScriptCompiled.js
```

You can also spawn the process using Frida to instrument it with QBDI as soon as it starts:

```
frida -f binaryPath Arguments -l MyScriptCompiled.js
```

Run Frida/QBDI on an Android device

Since Frida provides a great interface to instrument various types of target, we can also rely on it to use QBDI on Android, especially when it comes to inspecting applications. Nevertheless, it has some specificities you need to be aware of. Before running your script, make sure that:

- a Frida server is running on the remote device and is reachable from your workstation
- the `libQBDI.so` library has been placed in `/data/local/tmp` (available in Android packages)
- SELinux has been turned into *permissive* mode (through `setenforce 0`)

Then, you should be able to inject your script into a specific Android application:

```
# if the application is already running
frida -Un com.app.example -l MyScriptCompiled.js

# if you want to spawn the application
frida -Uf com.app.example -l MyScriptCompiled.js
```

Concrete example

If you have already had a look at the default instrumentation of the template generated with *qbdi-frida-template* (see *Generate a template*), you are probably familiar with the following example. Roughly speaking, what it does is creating a native call to the *Secret()* function, and instrument it looking for *XOR*.

Source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.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(); }
```

Frida/QBDI script

```
// QBDI
import { VM, InstPosition, VMAction } from "./frida-qbdi.js";

// Initialize QBDI
var vm = new VM();
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").address;
}
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) {
    var 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]);
```

Generate a template

A QBDI template can be considered as a baseline project, a minimal component you can modify and build your instrumentation tool on. They are provided to help you effortlessly start off a new QBDI based project. If you want to get started using QBDI bindings, you can create a brand-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
```

(continues on next page)

(continued from previous page)

```
# if frida-compile is not already 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 -f ./demo.bin -l ./RunMe.js
```

2.3.5 QBDIPreload

QBDIPreload is a small utility library that provides code injection capabilities using dynamic library injection. It works on Linux and macOS respectively with the `LD_PRELOAD` and `DYLD_INSERT_LIBRARIES` mechanisms.

Thanks to QBDIPreload, you can instrument the main function of an executable that has been dynamically linked. You can also define various callbacks that are called at specific times throughout the execution.

Main hook process

To use QBDIPreload, you must have a minimal codebase: a constructor and several *hook* functions. Like callbacks, hook functions are directly called by QBDIPreload.

First of all, the constructor of QBDIPreload has to be initialised through declaring the macro *QBDIPRELOAD_INIT*. It's worth noting that this macro must be only defined once in your code.

The *qbdipreload_on_start()* and *qbdipreload_on_premain()* hook functions are called at different stages during the execution of the programme. They only need to return *QBDIPRELOAD_NOT_HANDLED* if you don't want to modify the hook procedure.

```
#include "QBDIPreload.h"

QBDIPRELOAD_INIT;

int qbdipreload_on_start(void *main) {
    return QBDIPRELOAD_NOT_HANDLED;
}

int qbdipreload_on_premain(void *gprCtx, void *fpuCtx) {
    return QBDIPRELOAD_NOT_HANDLED;
}
```

Instrumentation

Once the main function is hooked by QBDIPreload, two methods are called: *qbdipreload_on_main()* and *qbdipreload_on_run()*.

At this point, you are able to capture the executable arguments inside of the *qbdipreload_on_main()* scope. The *qbdipreload_on_run()* function is called right afterwards with a ready-to-run QBDI virtual machine as first argument. Obviously, don't forget to register your callback(s) prior to running the VM.

```
static VMAction onInstruction(VMInstanceRef vm, GPRState *gprState, FPRState *fprState,␣
↪void *data) {
    // ...
    return QBDI_CONTINUE;
}

int qbdipreload_on_main(int argc, char** argv) {
    return QBDIPRELOAD_NOT_HANDLED;
}

int qbdipreload_on_run(VMInstanceRef vm, rword start, rword stop) {
    // add user callbacks
    qbdi_addCodeCB(vm, QBDI_PREINST, onInstruction, NULL);

    // run the VM
    qbdi_run(vm, start, stop);
    return QBDIPRELOAD_NO_ERROR;
}
```

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

Exit hook

QBDIPreload also intercepts the calls on standard exit functions (`exit` and `_exit`). Typically, these are called when the executable is about to terminate. If so, the *qbdipreload_on_exit()* method is called and can be used to save some data about the execution you want to keep before exiting. Note that the hook function is not called if the executable exits with a direct system call or a segmentation fault.

```
int qbdipreload_on_exit(int status) {
    return QBDIPRELOAD_NO_ERROR;
}
```

Compilation and execution

Finally, you need to compile your source code to a dynamic library. Your output binary has to be statically linked with both the QBDIPreload library and the QBDI library.

Then, in order to test it against a target, simply running the following command should do the job:

```
# on Linux
LD_BIND_NOW=1 LD_PRELOAD=./libqbdi_mytracer.so <executable> [<parameters> ...]

# on macOS
sudo DYLD_BIND_AT_LAUNCH=1 DYLD_INSERT_LIBRARIES=./libqbdi_mytracer.so <executable> [
↪<parameters> ...]
```

As the loader is not in the instrumentation range, we recommend setting `LD_BIND_NOW` or `DYLD_BIND_AT_LAUNCH` in order to resolve and bind all symbols before the instrumentation.

Full example

Merging everything we have learnt throughout this tutorial, we are now able to write our C/C++ source code files. In the following examples, we aim at displaying every executed instruction of the binary we are running against.

QBDIPreload in C

```c
#include <stdio.h>
#include "QBDIPreload.h"

QBDIPRELOAD_INIT;

static VMAction onInstruction(VMInstanceRef vm, GPRState *gprState,
                               FPRState *fprState, void *data) {
  const InstAnalysis *instAnalysis = qbdi_getInstAnalysis(
      vm, QBDI_ANALYSIS_INSTRUCTION | QBDI_ANALYSIS_DISASSEMBLY);
  printf("0x%" PRIRWORD " %s\n", instAnalysis->address,
         instAnalysis->disassembly);
  return QBDI_CONTINUE;
}

int qbdipreload_on_start(void *main) { return QBDIPRELOAD_NOT_HANDLED; }

int qbdipreload_on_premain(void *gprCtx, void *fpuCtx) {
  return QBDIPRELOAD_NOT_HANDLED;
}

int qbdipreload_on_main(int argc, char **argv) {
  return QBDIPRELOAD_NOT_HANDLED;
}

int qbdipreload_on_run(VMInstanceRef vm, rword start, rword stop) {
  qbdi_addCodeCB(vm, QBDI_PREINST, onInstruction, NULL, 0);
  qbdi_run(vm, start, stop);
  return QBDIPRELOAD_NO_ERROR;
}

int qbdipreload_on_exit(int status) { return QBDIPRELOAD_NO_ERROR; }
```

QBDIPreload in C++

```cpp
#include <iomanip>
#include <iostream>

#include "QBDIPreload.h"

static QBDI::VMAction onInstruction(QBDI::VMInstanceRef vm,
                                    QBDI::GPRState *gprState,
                                    QBDI::FPRState *fprState, void *data) {
  const QBDI::InstAnalysis *instAnalysis = vm->getInstAnalysis();
```

(continues on next page)

(continued from previous page)

```
  std::cout << std::setbase(16) << instAnalysis->address << ": "
            << instAnalysis->disassembly << std::endl
            << std::setbase(10);
  return QBDI::CONTINUE;
}

extern "C" {

QBDIPRELOAD_INIT;

int qbdipreload_on_start(void *main) { return QBDIPRELOAD_NOT_HANDLED; }

int qbdipreload_on_premain(void *gprCtx, void *fpuCtx) {
  return QBDIPRELOAD_NOT_HANDLED;
}

int qbdipreload_on_main(int argc, char **argv) {
  return QBDIPRELOAD_NOT_HANDLED;
}

int qbdipreload_on_run(QBDI::VMInstanceRef vm, QBDI::rword start,
                       QBDI::rword stop) {
  vm->addCodeCB(QBDI::PREINST, onInstruction, nullptr);
  vm->run(start, stop);
  return QBDIPRELOAD_NO_ERROR;
}

int qbdipreload_on_exit(int status) { return QBDIPRELOAD_NO_ERROR; }
}
```

Generate a template

A QBDI template can be considered as a baseline project, a minimal component you can modify and build your instrumentation tool on. They are provided to help you effortlessly start off a new QBDI based project. The binary responsible for generating a template is shipped in the release packages and can be used as follows:

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

MacOS setup

When QBDIPreload (and also PyQBDIPreload) are used on macOS, the injection can failed for various reasons.

Copy the target binary

If the binary is a system binary, the System Integrity Protection prevent the injection of library with `DYLD_INSERT_LIBRARIES`. You can try to copy the binary in your user home folder before the instrumentation to avoid the protection.

Run the instrumentation as root

The injection of library can be disable for standard user. Run the injection with `sudo` can bypass this limitation.

Instrument `arm64e` binary

QBDIPreload is not compatible with `arm64e` binary. To verify if your binary is a `arm64e` binary, you can execute the followed command:

```
lipo -archs $BINARY_PATH
```

You can try to convert your target binary from `arm64e` to `arm64` with the followed script. However, this don't work if the target binary use some `arm64e` bind opcode (like `BIND_OPCODE_THREADED`).

```
import lief
import subprocess

binary = lief.MachO.parse(inputfile)
for index in range(binary.size):
    if binary.at(index).header.cpu_type == lief.MachO.CPU_TYPES.ARM64:
        binary.at(index).header.cpu_subtype ^= (~2)
binary.write(outputfile)
subprocess.run(["codesign", "--force", "--sign", "-", outputfile], check=True)
```

Disable System Integrity Protection

We recommand to disable the *System Integrity Protection* only as a last resort, as we successfully inject the QBDIPreload under `macOS Ventura` (version 13.1). However, this may help you to debug the injection and bypass some signature validation. You can found the procedure here.

Don't forget to reenable it afterwards.

2.3.6 PyQBDIPreload

PyQBDIPreload is an implementation of QBDIPreload for PyQBDI. It allows users to inject the Python runtime into a target process and execute their own script in it. The limitations are pretty much the same as those we face on QBDIPreload and PyQBDI:

- Only Linux and macOS are currently supported
- The executable should be injectable with `LD_PRELOAD` or `DYLD_INSERT_LIBRARIES`
- PyQBDIPreload cannot be injected in a Python process
- The Python runtime and the target must share the same architecture
- An extra *VM* must not be created. An already prepared *VM* is provided to *pyqbdipreload_on_run()*.

Note: The Python library `libpython3.x.so` must be installed.

Main hook process

Unlike QBDIPreload, the hook of the main function cannot be customised so you are unable to alter the hook process. The Python interpreter is only initialised once the main function is hooked and the script is loaded. Furthermore, some modifications are made on the environment of the interpreter before the user script loading:

- `sys.argv` are the argument of the executable
- `LD_PRELOAD` or `DYLD_INSERT_LIBRARIES` are removed from `os.environ`
- `pyqbdi.__preload__` is set to `True`

Instrumentation

Once the script is loaded, the *pyqbdipreload_on_run()* function is called with a ready-to-run *VM*. Any user callbacks should be registered to the *VM*, then the VM can be run with *pyqbdi.VM.run()*.

```
import pyqbdi

def instCallback(vm, gpr, fpr, data):
    # User code ...
    return pyqbdi.CONTINUE

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

Exit hook

The `atexit` module is triggered when the execution is finished, or when `exit` or `_exit` are called.

Note: Any *VM* object is invalided when the `atexit` module is triggered and should never be used.

Execution

A script called `pyqbdipreload.py` is brought to set the environment up and run the executable. The first parameter is the PyQBDIPreload script. Next are the binary followed by its respective arguments if any.

```
python3 -m pyqbdipreload <script> <executable> [<arguments> ...]
```

Full example

Merging everything we have learnt throughout this tutorial, we are now able to write our Python script which prints the instructions that are being executed by the target executable.

```
#!/usr/bin/env python3

import pyqbdi

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


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

2.4 Tutorials

This section contains some tutorials on specifics points of QBDI

2.4.1 Basic block events

Introduction

The *Instrument Callback* can insert callbacks on all or specific instructions. With a callback on every instruction, it's trivial to follow the execution pointer and obtain a trace of the execution. However, performances are bad because the execution is stopped on each instruction for the callback to be run. For some traces, a higher-level callback may have better performances.

The *VM callbacks* is called when some conditions are reached during the execution. This tutorial introduces 3 *VMEvent*:

- `BASIC_BLOCK_NEW` is triggered when a new basic block has been instrumented and added to the cache. It can be used to create coverage of the execution.

- `BASIC_BLOCK_ENTRY` is triggered before the execution of a basic block.
- `BASIC_BLOCK_EXIT` is triggered after the execution of a basic block.

A basic block in QBDI

QBDI doesn't analyze the whole program before the run. Basic blocks are dynamically detected and so may not match basic blocks given by other tools. In QBDI, a basic block is a sequence of consecutive instructions that do not modify the instruction pointer except for the last one. Any instruction that may modify the instruction pointer (method call, jump, conditional jump, method return, ...) are always the last instruction of a basic block.

For QBDI, is the beginning for a basic block:

- the very first instruction to be executed in QBDI;
- the first instruction to be executed after the end of the previous basic block;
- the first instruction to be executed if the user modifies the execution flow (add a new callback, clear the cache, return `BREAK_TO_VM`, ...)

Due to the dynamic detection of the basic block, basic blocks may overlap each other. This behavior can be observed in the following code:

```
# BB
   push rbp                           # 0x1000
   mov  rbp, rsp                      # 0x1001
   mov  dword ptr [rbp - 0x14], edi   # 0x1004
   mov  edx, dword ptr [rbp - 0x14]   # 0x1007
   mov  eax, edx                      # 0x100a
   shl  eax, 2                        # 0x100c
   add  eax, edx                      # 0x100f
   mov  dword ptr [rbp - 4], eax      # 0x1011
   cmp  dword ptr [rbp - 0x14], 0xa   # 0x1014
   jle  0x1027                        # 0x1018
# BB
   add  dword ptr [rbp - 4], 0x33     # 0x101e
   jmp  0x1033                        # 0x1022
# BB
   mov  eax, dword ptr [rbp - 4]      # 0x1027
   imul eax, eax                      # 0x102a
   add  eax, 0x57                     # 0x102d
   mov  dword ptr [rbp - 4], eax      # 0x1030
# BB
   mov  edx, dword ptr [rbp - 4]      # 0x1033
   mov  eax, dword ptr [rbp - 0x14]   # 0x1036
   add  eax, edx                      # 0x1039
   pop  rbp                           # 0x103b
   ret                                # 0x103c
```

In this snippet, QBDI can detect 4 different basic blocks. If the first jump isn't taken:

- The begin of the method, between 0x1000 and 0x101e;
- The block between 0x101e and 0x1027;
- The last block between 0x1033 and 0x103d.

If the first jump is taken:

- The begin of the method, between 0x1000 and 0x101e;
- The last block between 0x1027 and 0x103d.

Getting basic block information

To receive basic block information, a `VMCallback` should be registered to the VM with `addVMEventCB` for one of `BASIC_BLOCK_*` events. Once a registered event occurs, the callback is ran with a description of the VM (`VMState`).

The address of the current basic block can be retrieved with `VMState.basicBlockStart` and `VMState.basicBlockEnd`.

Note: A callback may register for both `BASIC_BLOCK_NEW` and `BASIC_BLOCK_ENTRY` events but would be called only once would these two events happen at the same time. You can retrieve the events that triggered the callback in `VMState.event`.

The following example registers for the three events in the VM and displays the basic block's bounds.

C basic block information

Reference: *VMCallback*, *qbdi_addVMEventCB()*, *VMState*

```
VMAction vmcbk(VMInstanceRef vm, const VMState* vmState, GPRState* gprState, FPRState*␣
↪fprState, void* data) {

    printf("start:0x%" PRIRWORD ", end:0x%" PRIRWORD "%s%s%s\n",
        vmState->basicBlockStart,
        vmState->basicBlockEnd,
        (vmState->event & QBDI_BASIC_BLOCK_NEW)? " BASIC_BLOCK_NEW":"",
        (vmState->event & QBDI_BASIC_BLOCK_ENTRY)? " BASIC_BLOCK_ENTRY":"",
        (vmState->event & QBDI_BASIC_BLOCK_EXIT)? " BASIC_BLOCK_EXIT":"");
    return QBDI_CONTINUE;
}

qbdi_addVMEventCB(vm, QBDI_BASIC_BLOCK_NEW | QBDI_BASIC_BLOCK_ENTRY | QBDI_BASIC_BLOCK_
↪EXIT, vmcbk, NULL);
```

C++ basic block information

Reference: *QBDI::VMCallback*, *QBDI::VM::addVMEventCB()*, *QBDI::VMState*

```
QBDI::VMAction vmcbk(QBDI::VMInstanceRef vm, const QBDI::VMState* vmState,␣
↪QBDI::GPRState* gprState, QBDI::FPRState* fprState, void* data) {

    std::cout << std::setbase(16) << "start:0x" << vmState->basicBlockStart
              << ", end:0x" << vmState->basicBlockEnd;
    if (vmState->event & QBDI::BASIC_BLOCK_NEW) {
        std::cout << " BASIC_BLOCK_NEW";
    }
    if (vmState->event & QBDI::BASIC_BLOCK_ENTRY) {
```

(continues on next page)

(continued from previous page)

```
        std::cout << " BASIC_BLOCK_ENTRY";
    }
    if (vmState->event & QBDI::BASIC_BLOCK_EXIT) {
        std::cout << " BASIC_BLOCK_EXIT";
    }
    std::cout << std::endl;
    return QBDI::CONTINUE;
}

vm.addVMEventCB(QBDI::BASIC_BLOCK_NEW | QBDI::BASIC_BLOCK_ENTRY | QBDI::BASIC_BLOCK_EXIT,
↪ vmcbk, nullptr);
```

PyQBDI basic block information

Reference: *pyqbdi.VMCallback()*, *pyqbdi.VM.addVMEventCB()*, *pyqbdi.VMState*

```
def vmcbk(vm, vmState, gpr, fpr, data):
    # user callback code

    print("start:0x{:x}, end:0x{:x} {}".format(
        vmState.basicBlockStart,
        vmState.basicBlockEnd,
        vmState.event & (pyqbdi.BASIC_BLOCK_NEW | pyqbdi.BASIC_BLOCK_ENTRY | pyqbdi.
↪BASIC_BLOCK_EXIT) ))
    return pyqbdi.CONTINUE

vm.addVMEventCB(pyqbdi.BASIC_BLOCK_NEW | pyqbdi.BASIC_BLOCK_ENTRY | pyqbdi.BASIC_BLOCK_
↪EXIT, vmcbk, None)
```

Frida/QBDI basic block information

Reference: *VMCallback()*, *VM.addVMEventCB()*, *VMState()*

```
var vmcbk = vm.newVMCallback(function(vm, state, gpr, fpr, data) {
    var msg = "start:0x" + state.basicBlockStart.toString(16) + ", end:0x" + state.
↪basicBlockEnd.toString(16);
    if (state.event & VMEvent.BASIC_BLOCK_NEW) {
        msg = msg + " BASIC_BLOCK_NEW";
    }
    if (state.event & VMEvent.BASIC_BLOCK_ENTRY) {
        msg = msg + " BASIC_BLOCK_ENTRY";
    }
    if (state.event & VMEvent.BASIC_BLOCK_EXIT) {
        msg = msg + " BASIC_BLOCK_EXIT";
    }
    console.log(msg);
    return VMAction.CONTINUE;
});
```

(continues on next page)

(continued from previous page)

```
vm.addVMEventCB(VMEvent.BASIC_BLOCK_NEW | VMEvent.BASIC_BLOCK_ENTRY | VMEvent.BASIC_
↪BLOCK_EXIT, vmcbk, null);
```

Basic block coverage

To perform code coverage, `BASIC_BLOCK_NEW` can be used to detect the new basic block JITed by QBDI. However, it wouldn't work in the following cases:

- If the code jumps outside of the instrumented range.
- If the code triggers an interruption (exception, signal, ...)
- If the code uses overlapping instructions or other forms of obfuscation.

Moreover, if a VM is reused from an execution to another, cache will be kept and so coverage would be incremental. Clear the cache between every run to have independent coverage results

For more precise coverage, a user may register `BASIC_BLOCK_ENTRY` or `BASIC_BLOCK_EXIT` events and handle deduplication themselves.

C coverage

```
// your own coverage library
#include "mycoverage.h"

VMAction covcbk(VMInstanceRef vm, const VMState* vmState, GPRState* gprState, FPRState*␣
↪fprState, void* data) {

    myCoverageAdd( (myCoverageData*) data, vmState->basicBlockStart, vmState->
↪basicBlockEnd);
    return QBDI_CONTINUE;
}

myCoverageData cover;

qbdi_addVMEventCB(vm, QBDI_BASIC_BLOCK_NEW, covcbk, &cover);

// run the VM
// ....

// print the coverage
myCoveragePrint(&cover);
```

C++ coverage

QBDI has a tiny range set class (*QBDI::RangeSet*), usable only with the C++ API.

```
QBDI::VMAction covcbk(QBDI::VMInstanceRef vm, const QBDI::VMState* vmState,␣
↪QBDI::GPRState* gprState, QBDI::FPRState* fprState, void* data) {

    QBDI::RangeSet<QBDI::rword>* rset = static_cast<QBDI::RangeSet<QBDI::rword>*>(data);
    rset->add({vmState->basicBlockStart, vmState->basicBlockEnd});

    return QBDI::CONTINUE;
}

QBDI::RangeSet<QBDI::rword> rset;

vm.addVMEventCB(QBDI::BASIC_BLOCK_NEW, covcbk, &rset);

// run the VM
// ....

// print the coverage
for (const auto r: rset.getRanges()) {
    std::cout << std::setbase(16) << "0x" << r.start() << " to 0x" << r.end() <<␣
↪std::endl;
}
```

PyQBDI coverage

```
def covcbk(vm, vmState, gpr, fpr, data):

    if vmState.basicBlockEnd not in data['cov'] or vmState.basicBlockStart < data['cov
↪'][vmState.basicBlockEnd][0]:
        data['cov'][vmState.basicBlockEnd] = (vmState.basicBlockStart, vmState.
↪basicBlockEnd)
    return pyqbdi.CONTINUE


cov = {"cov": {}}

vm.addVMEventCB(pyqbdi.BASIC_BLOCK_NEW, covcbk, cov)

# run the VM
# ....

for _, c in cov['cov'].items():
    print(f"0x{c[0]:x} to 0x{c[1]:x}")
```

In addition, a coverage script that generates DRCOV coverage is available in examples/pyqbdi/coverage.py.

Frida/QBDI coverage

```
var covcbk = vm.newVMCallback(function(vm, state, gpr, fpr, cov) {
    if ( (! cov[state.basicBlockEnd]) || state.basicBlockStart < cov[state.
↪basicBlockEnd][0] ) {
        cov[state.basicBlockEnd] = [state.basicBlockStart, state.basicBlockEnd]
    }
    return VMAction.CONTINUE;
});

var cov = {};

vm.addVMEventCB(VMEvent.BASIC_BLOCK_NEW, covcbk, cov);

// run the VM
// ....

for(var c in cov){
    console.log("0x" + cov[c][0].toString(16) + " to 0x" + cov[c][1].toString(16));
}
```

Edge coverage

The `BASIC_BLOCK_EXIT` event can be used to detect the edge between basic blocks. As the event is triggered at the end of a basic block (ie. after instruction pointer is modified), the next address can be found in the GPRState. So, the couple `(state.basicBlockEnd, gpr.rip)` is the edge to store in the coverage.

2.4.2 Execution transfert event

Introduction

One limitation of QBDI is it shares the heap and some library with the instrumented code. With this design, the user may use any shared library and doesn't need to statically link all their dependencies with their code. However, some method must not be instrumented in QBDI:

- The heap allocator method (malloc, free, ...).
- Any no reentrant method shared between the target code and QBDI itself.
- Any no reentrant method shared between the target code and user callbacks.

When the target code calls one of these methods, QBDI restores the native execution. The return address is changed in order to catch the return and to continue the instrumentation of the code. Two events allow the user to detect this mechanism:

- `EXEC_TRANSFER_CALL`: called before restoring native execution for the method.
- `EXEC_TRANSFER_RETURN`: called after the execution of the method.

These two events can be used to retrieve the method and its parameters before the call and its return value after. `EXEC_TRANSFER_CALL` can also be used to emulate a method call.

Get native call symbols

When QBDI needs to restore the native execution, the user may retrieve the name of the calling method based on the current address. The associated symbol can be found with `dladdr` (on Linux and OSX) or `SymFromAddr` (on Windows).

We recommend forcing the linker to resolve all symbols before running the VM. This can be achieved with:

- `LD_BIND_NOW=1` on Linux
- `DYLD_BIND_AT_LAUNCH=1` on OSX

With dladdr

`dladdr` may not find the symbol associated with an address if it's not an exported symbol. If several symbols are associated, only one is returned.

```
static VMAction transfertcbk(VMInstanceRef vm, const VMState *vmState, GPRState␣
↪*gprState, FPRState *fprState, void *data) {
    Dl_info info = {0};
    dladdr((void*)gprState->rip, &info);

    if (info.dli_sname != NULL) {
        printf("Call %s (addr: 0x%" PRIRWORD ")\n", info.dli_sname, gprState->rip);
    } else {
        printf("Call addr: 0x%" PRIRWORD "\n", gprState->rip);
    }
    return QBDI_CONTINUE;
}

qbdi_addVMEventCB(vm, QBDI_EXEC_TRANSFER_CALL, transfertcbk, NULL);
```

```
import ctypes
import ctypes.util

class Dl_info(ctypes.Structure):
    _fields_ = [('dli_fname', ctypes.c_char_p),
                ('dli_fbase', ctypes.c_void_p),
                ('dli_sname', ctypes.c_char_p),
                ('dli_saddr', ctypes.c_void_p)]

libdl_path = ctypes.util.find_library('dl')
assert libdl_path != None
libdl = ctypes.cdll.LoadLibrary(libdl_path)
libdl.dladdr.argtypes = (ctypes.c_void_p, ctypes.POINTER(Dl_info))

def dladdr(addr):

    res = Dl_info()
    result = libdl.dladdr(ctypes.cast(addr, ctypes.c_void_p), ctypes.byref(res))

    return res.dli_sname

def transfertcbk(vm, vmState, gpr, fpr, data):
```

(continues on next page)

(continued from previous page)

```
    print("Call {} (addr: 0x{:x})".format(
        dladdr(gpr.rip),
        gpr.rip))

    return pyqbdi.CONTINUE

vm.addVMEventCB(pyqbdi.EXEC_TRANSFER_CALL, transfertcbk, None)
```

With lief

Lief is a C, C++ and python library that aims to parse ELF, PE and MachO file formats. This library can extract all the symbols associated with an address, including the non-exported one. This solution can resolve more addresses, but could be slower than `dladdr`.

For ELF binary, the following code prints for each `EXEC_TRANSFER_CALL` event, the symbols associated with the target address. For PE library, the user may need to parse the PDB file of the library to get the symbol associated with the target address.

```
#include <LIEF/LIEF.hpp>

class Module {
    public:
        std::string path;
        QBDI::Range<QBDI::rword> range;

        Module(const QBDI::MemoryMap& m) : path(m.name), range(m.range) {}

        void append(const QBDI::MemoryMap& m) {
            if (m.range.start() < range.start()) {
                range.setStart(m.range.start());
            }
            if (m.range.end() > range.end()) {
                range.setEnd(m.range.end());
            }
        }
};

class AddrResolver {
    private:
        std::vector<Module> modules;
        std::unordered_set<std::string> loaded_path;
        std::unordered_map<QBDI::rword, std::unordered_set<std::string>> resolv_cache;

        void cacheModules();
        const Module* getModule(QBDI::rword addr, bool reload = true);
        void loadModule(const Module& m);

    public:
        AddrResolver() {
            cacheModules();
```

(continues on next page)

(continued from previous page)

```
        }

        const std::unordered_set<std::string>& resolve(QBDI::rword addr);
};

void AddrResolver::cacheModules() {
    modules.clear();

    for (const auto& map : QBDI::getCurrentProcessMaps(true)) {
        auto r = std::find_if(std::begin(modules), std::end(modules),
                [&](const Module& m){return m.path == map.name;});
        if (r != std::end(modules)) {
            r->append(map);
        } else if (map.name.find("/") != std::string::npos) {
            modules.emplace_back(map);
        }
    }
}

const Module* AddrResolver::getModule(QBDI::rword addr, bool reload) {
    const auto r = std::find_if(std::begin(modules), std::end(modules),
            [&](const Module& m){return m.range.contains(addr);});
    if (r != std::end(modules)) {
        return &*r;
    } else if (reload) {
        cacheModules();
        return getModule(addr, false);
    } else {
        return nullptr;
    }
}

void AddrResolver::loadModule(const Module& m) {
    std::cout << "Load Module " << m.path << std::endl;
    if (loaded_path.find(m.path) != loaded_path.end()) {
        return;
    }
    std::unique_ptr<LIEF::ELF::Binary> externlib = LIEF::ELF::Parser::parse(m.path);
    if (not externlib) {
        return;
    }
    for (const auto& s: externlib->symbols()) {
        QBDI::rword addr = s.value() + m.range.start();
        resolv_cache[addr].emplace(s.demangled_name());
    }

    loaded_path.emplace(m.path);
}

const std::unordered_set<std::string>& AddrResolver::resolve(QBDI::rword addr) {
    const auto & symnames = resolv_cache[addr];
    if (!symnames.empty()) {
```

(continues on next page)

(continued from previous page)

```
        return symnames;
    }
    std::cout << std::setbase(16) << "Fail to found 0x" << addr << std::endl;
    const Module* m = getModule(addr);
    if (m != nullptr) {
        loadModule(*m);
    }
    return symnames;
}

QBDI::VMAction transfertCBK(QBDI::VMInstanceRef vm, const QBDI::VMState* vmState,␣
↪QBDI::GPRState* gprState, QBDI::FPRState* fprState, void* data) {
    const std::unordered_set<std::string>& r = static_cast<AddrResolver*>(data)->
↪resolve(gprState->rip);

    if (r.empty()) {
        std::cout << std::setbase(16) << "Call addr: 0x" << gprState->rip << std::endl;
    } else {
        std::cout << "Call ";
        for (const auto& s: r) {
            std::cout << s << " ";
        }
        std::cout << std::setbase(16) << "(addr: 0x" << gprState->rip << ")" <<␣
↪std::endl;
    }
    return QBDI::CONTINUE;
}

AddrResolver data;
vm->addVMEventCB(QBDI::EXEC_TRANSFER_CALL, transfertCBK, &data);
```

```
import lief
import pyqbdi

class Module:
    def __init__(self, module):
        self.name = module.name
        self.range = pyqbdi.Range(module.range.start, module.range.end)

    def append(self, module):
        assert module.name == self.name
        if module.range.start < self.range.start:
            self.range.start = module.range.start
        if self.range.end < module.range.end:
            self.range.end = module.range.end

class AddrResolver:

    def __init__(self):
        self.lib_cache = []
        self.resolv_cache = {}
        self.map_cache = self.get_exec_maps()
```

(continues on next page)

(continued from previous page)

```
    def get_exec_maps(self):
        maps = {}
        for m in pyqbdi.getCurrentProcessMaps(True):
            if m.name in maps:
                maps[m.name].append(m)
            elif '/' in m.name:
                maps[m.name] = Module(m)
        return maps

    def get_addr_maps(self, addr):
        for _, m in self.map_cache.items():
            if addr in m.range:
                return m
        self.map_cache = self.get_exec_maps()
        for _, m in self.map_cache.items():
            if addr in m.range:
                return m
        return None

    def load_lib(self, maps):
        if maps.name in self.lib_cache:
            return

        # use lief.PE or lief.MACO if not ELF file
        lib = lief.ELF.parse(maps.name)
        if lib is None:
            return

        for s in lib.symbols:
            addr = s.value + maps.range.start
            if addr in self.resolv_cache:
                if s.name not in self.resolv_cache[addr]:
                    self.resolv_cache[addr].append(s.name)
            else:
                self.resolv_cache[addr] = [s.name]

        self.lib_cache.append(maps.name)

    def get_names(self, addr):

        if addr in self.resolv_cache:
            return self.resolv_cache[addr]

        maps = self.get_addr_maps(addr)
        if maps == None:
            return []
        self.load_lib(maps)
        if addr in self.resolv_cache:
            return self.resolv_cache[addr]
        self.resolv_cache[addr] = []
        return []
```

(continues on next page)

(continued from previous page)

```
def transfertcbk(vm, vmState, gpr, fpr, data):

    f_names = data['resolver'].get_names(gpr.rip)
    if f_names != []:
        print("Call {} (addr: 0x{:x})".format(f_names, gpr.rip))
    else:
        print("Call addr: 0x{:x}".format(gpr.rip))

    return pyqbdi.CONTINUE

ctx = {
    "resolver": AddrResolver(),
}

vm.addVMEventCB(pyqbdi.EXEC_TRANSFER_CALL, transfertcbk, ctx)
```

Using this snippet with PyQBDIPreload prints the libc calls.

```
$ python -m pyqbdipreload test.py ls
Call ['__strrchr_avx2'] (addr: 0x7f2aed2a8330)
Call ['setlocale', '__GI_setlocale'] (addr: 0x7f2aed17a7f0)
Call ['bindtextdomain', '__bindtextdomain'] (addr: 0x7f2aed17e000)
Call ['textdomain', '__textdomain'] (addr: 0x7f2aed1815f0)
Call ['__cxa_atexit', '__GI___cxa_atexit'] (addr: 0x7f2aed1879b0)
Call ['getopt_long'] (addr: 0x7f2aed22d3f0)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
Call ['isatty', '__isatty'] (addr: 0x7f2aed239250)
Call ['ioctl', '__ioctl', '__GI_ioctl', '__GI___ioctl'] (addr: 0x7f2aed23d590)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
Call ['__errno_location', '__GI___errno_location'] (addr: 0x7f2aed16fde0)
Call ['__libc_malloc', 'malloc', '__GI___libc_malloc', '__malloc'] (addr: 0x7f2aed1d3320)
Call ['__memcpy_avx_unaligned', '__memmove_avx_unaligned'] (addr: 0x7f2aed2ab4a0)
Call ['__errno_location', '__GI___errno_location'] (addr: 0x7f2aed16fde0)
Call ['__libc_malloc', 'malloc', '__GI___libc_malloc', '__malloc'] (addr: 0x7f2aed1d3320)
Call ['__memcpy_avx_unaligned', '__memmove_avx_unaligned'] (addr: 0x7f2aed2ab4a0)
Call ['getenv', '__GI_getenv'] (addr: 0x7f2aed186b20)
....
```

2.5 API

This section contains the documentation of every QBDI API.

2.5.1 API description

This section gives the big picture of all the APIs QBDI offers. It may help you better understand the way QBDI works and the overall design behind it.

Injection

Instrumentation with QBDI implies having both QBDI and the target program inside the same process. Three types of injections are currently possible with QBDI.

Create a QBDI program

The first type of injection is to create an executable that uses QBDI and loads the target code. This injection is available with C, C++ and PyQBDI APIs and should be used if you can compile the target code inside the client binary or load it as a library.

The main steps of this injection are:

- Load the code (`dlopen`, `LoadLibrary`, ...)
- Initialise a new QBDI VM
- Select instrumentation ranges
- Allocate a stack for the target process
- Define callbacks
- Call a target method inside the VM

Inject a preload library

This injection forces the loader to add a custom library inside a new process. The preloaded library will catch the main address and allow users to instrument it. This injection should be used if you want to instrument a program from the start of the `main` method to the end.

We provide a small tool called *QBDIPreload* for doing so on Linux and macOS. QBDIPreload (and its implementation *PyQBDIPreload* for *PyQBDI*) automatically puts a hook on the `main` method and accordingly prepares a QBDI VM that can be used out of the box.

Typically, the different stages are:

- Wait for *qbdipreload_on_main()* or *pyqbdipreload_on_run()* to be called with an initialised QBDI VM
- Define callbacks
- Run the `main` method inside the VM

Inject with Frida

Frida works on a wide variety of systems and brings powerful and handy injection mechanisms. It is why the Frida/QBDI subproject relies on those to inject itself into processes. On top of that, while developing their scripts, users are free to take advantage of both Frida and QBDI APIs.

Tracing a specific function can be summed up as:

- Inject a script into a process with Frida
- Put callbacks on the target functions
- Wait for the hooks to be triggered
- Create a QBDI VM, synchronise the context and define instrumentation ranges
- Register callbacks
- Remove the instrumentation added by Frida
- Run the function in the context of QBDI
- Restore the Frida hook for potential future calls
- Return the return value obtained from QBDI

Instrumentation ranges

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

The Instrumentation Range is a range of addresses where QBDI will intrument the original code. When the programme counter gets out of this scope, QBDI will try to restore a native and uninstrumented execution. For performance reasons, it is rarely recommended to instrument all the guest code. Moreover, you must bear in mind that QBDI shares the same standard library as the guest and some methods are not *reentrant* (more details in *Limitations*).

The current mechanism is implemented by the `ExecBroker` and only supports external calls out of the instrumentation range. When the execution gets out of the range, `ExecBroker` will try to find an address on the stack that is inside the instrumentation range. If an address is found, it will be replaced by a custom one and the execution is restored without instrumentation. When the process returns to this address `ExecBroker` will capture its state and continue the execution at the expected address. If any valid return address is found, the instrumentation will continue until finding a valid return address.

The following limitations are known:

- The instrumentation range must be at a function level, and if possible, at library level. A range that includes only some instructions of a function will produce an unpredictable result.
- When the native instrumentation goes out of the instrumentation range, the only method to restore the instrumentation is to return to the modified address. Any other executions of code inside the instrumentation range will not be caught (callbacks, ...).
- The current `ExecBroker` doesn't support any exception mechanism, included the *setjmp/longjmp*.
- The instrumentation range, and QBDI in general, are **not** a security sandbox. The code may escape and runs without instrumentation.

The instrumentation ranges can be managed through:

- `addInstrumentedRange` and `removeInstrumentedRange` to add or remove a specific range of address
- `addInstrumentedModule` and `removeInstrumentedModule` to add or remove a library/module with his name

- `addInstrumentedModuleFromAddr` and `removeInstrumentedModuleFromAddr` to add or remove a library/module with one of his addresses
- `instrumentAllExecutableMaps` and `removeAllInstrumentedRanges` to add or remove all the executable range

Register state

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*
- Management APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

QBDI defines two structures for the registers: `GPRState` and `FPRState`.

- `GPRState` contains all the General Purpose registers such as `rax`, `rsp`, `rip` or `eflags` on X86_64.
- `FPRState` contains the the Floating Point registers.

Inside a `InstCallback` and `VMCallback`, the current state is passed as a parameter and any change on it will affect the execution. Outside of a callback, `GPRState` and `FPRState` can be retrieved and set with `getGPRState`, `getFPRState`, `getGPRState` and `getFPRState`.

Note: A modification of the instruction counter (e.g. `RIP`) in an `InstCallback` or a `VMCallback` is not effective if `BREAK_TO_VM` is not returned.

User callbacks

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

QBDI allows users to register callbacks that are called throughout the execution of the target. These callbacks can be used to determine what the program is doing or to modify its state. Some callbacks must return an action to specify whether the execution should continue or stop or if the context needs to be reevaluated.

All user callbacks must be written in C, C++, Python or JS. However, there are a few limitations:

- As the target registers are saved, the callback can use any register by respecting the standard calling convention of the current platform.
- Some methods of the VM are not *reentrant* and must not be called within the scope of a callback. (`run`, `call`, `setOptions`, `precacheBasicBlock`, destructor, copy and move operators)
- The `BREAK_TO_VM` action should be returned instead of the `CONTINUE` action if the state of the VM is somehow changed. It covers:
  - Add or remove callbacks
  - Modify instrumentation ranges
  - Clear the cache
  - Change the instruction counter register in `GPRState` (the other registers can be altered without the need of returning `BREAK_TO_VM`).

Instruction callbacks

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

An instruction callback (`InstCallback`) is a callback that will be called **before** or **after** executing an instruction. Therefore, an `InstCallback` can be inserted at two different positions:

- before the instruction (`PREINST` – short for *pre-instruction*)
- after the instruction (`POSTINST` – short for *post-instruction*). At this point, the register state has been automatically updated so the instruction counter points to the next code address.

Note: A `POSTINST` callback will be called after the instruction and before the next one. If on a call instruction, the callback is then called before the first instruction of the called method.

An `InstCallback` can be registered for a specific instruction (`addCodeAddrCB`), any instruction in a specified range (`addCodeRangeCB`) or any instrumented instruction (`addCodeCB`). The instruction also be targeted by their mnemonic (or LLVM opcode) (`addMnemonicCB`).

VM callbacks

- `VMCallback` APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*
- `VMEvent` APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

A `VMEvent` callback (`VMCallback`) is a callback that will be called when the VM reaches a specific state. The current supported events are:

- At the beginning and the end of a basic block (`BASIC_BLOCK_ENTRY` and `BASIC_BLOCK_EXIT`). A basic block in QBDI consists of consecutive instructions that don't change the instruction counter except the last one. These events are triggered respectively by `SEQUENCE_ENTRY` and `SEQUENCE_EXIT` as well.
- At the beginning and the end of a sequence (`SEQUENCE_ENTRY` and `SEQUENCE_EXIT`). A sequence is a part of a basic block that has been *JIT'd* consecutively. These events should only be used for `getBBMemoryAccess`.
- When a new uncached basic block is being *JIT'd* (`BASIC_BLOCK_NEW`). This event is also always triggered by `BASIC_BLOCK_ENTRY` and `SEQUENCE_ENTRY`.
- Before and after executing some uninstrumented code with the `ExecBroker` (`EXEC_TRANSFER_CALL` and `EXEC_TRANSFER_RETURN`).

When a `VMCallback` is called, a state of the VM (`VMState`) is passed in argument. This state contains:

- A set of events that trigger the callback. If the callback is registered for several events that trigger at the same moment, the callback will be called only once.
- If the event is related to a basic block or a sequence, the start and the end addresses of the current basic block and sequence are provided.

Memory callbacks

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

The memory callback is an `InstCallback` that will be called when the target program reads or writes the memory. The callback can be called only when a specific address is accessed (`addMemAddrCB`), when a range of address is accessed (`addMemRangeCB`) or when any memory is accessed (`addMemAccessCB`).

Unlike with the instruction callback registration, the position of a memory callback cannot be manually specified. If a memory callback is solely registered for read accesses, it will be called **before** the instruction. Otherwise, it will be called **after** executing the instruction.

Instrumentation rule callbacks

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

Instrumentation rule callbacks are an advanced feature of QBDI. It allows users to define a callback (`InstrRuleCallback`) that will be called during the instrumentation process. The callback will be called for each instruction and can define an `InstCallback` to call before or after the current instruction. An argument contains an `InstAnalysis` of the current instruction and can be used to define the callback to insert for this instruction.

An `InstrRuleCallback` can be registered for all instructions (`addInstrRule`) or only for a specific range (`addInstrRuleRange`).

Note: The instrumentation process of QBDI responsible of *JITing* instructions may analyse more than once the same instruction. Consequently, the instrumentation rule callback must always return the same result even though the instruction has already been instrumented.

Instruction analysis

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*
- Getter APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

QBDI provides some basic analysis of the instruction through the `InstAnalysis` object. Within an `InstCallback`, the analysis of the current instruction should be retrieved with `getInstAnalysis`. Otherwise, the analysis of any instruction in the cache can be obtained with `getCachedInstAnalysis`. The `InstAnalysis` is cached inside QBDI and is valid until the next cache modification (add new instructions, clear cache, ...).

Four types of analysis are available. If a type of analysis is not selected, the corresponding field of the `InstAnalysis` object remains empty and should not be used.

- `ANALYSIS_INSTRUCTION`: This analysis type provides some generic information about the instruction, like its address, its size, its mnemonic (LLVM opcode) or its condition type if the instruction is conditional.
- `ANALYSIS_DISASSEMBLY`: This analysis type provides the disassembly of the instruction. For X86 and X86_64, the syntax Intel is used by default. The syntax can be changed with the option `OPT_ATT_SYNTAX`.
- `ANALYSIS_OPERANDS`: This analysis type provides information about the operand of the instruction. An operand can be a register or an immediate. If a register operand can be empty, the special type `OPERAND_INVALID` is used. The implicit register of instruction is also present with a specific flag. Moreover, the member `flagsAccess` specifies whether the instruction will use or set the generic flag.
- `ANALYSIS_SYMBOL`: This analysis type detects whether a symbol is associated with the current instruction.

The source file `test/API/InstAnalysisTest_<arch>.cpp` shows how one can deal with instruction analysis and may be taken as a reference for the `ANALYSIS_INSTRUCTION` and `ANALYSIS_OPERANDS` types.

Memory accesses

- Global APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*
- Getter APIs: *C*, *C++*, *PyQBDI*, *Frida/QBDI*

Due to performance considerations, the capture of memory accesses (`MemoryAccess`) is not enabled by default. It is only turned on when a memory callback is registered or explicitly requested with `recordMemoryAccess`. Collecting read and written accesses can be enabled either together or separately.

Two APIs can be used to get the memory accesses:

- `getInstMemoryAccess` can be used within an instruction callback or a memory callback to retrieve the access of the current instruction. If the callback is before the instruction (`PREINST`), only read accesses will be available.
- `getBBMemoryAccess` must be used in a `VMEvent` callback with `SEQUENCE_EXIT` to get all the memory accesses for the last sequence.

Both return a list of `MemoryAccess`. Generally speaking, a `MemoryAccess` will have the address of the instruction responsible of the access, the access address and size, the type of access and the value read or written. However, some instructions can do complex accesses and some information can be missing or incomplete. The `flags` of `MemoryAccess` can be used to detect these cases:

- `MEMORY_UNKNOWN_SIZE`: the size of the access is unknown. This is currently used for instruction with `REP` prefix before the execution of the instruction. The size is determined after the instruction when the access has been completed.
- `MEMORY_MINIMUM_SIZE`: The size of the access is a minimum size. The access is complex but at least `size` of memory is accessed. This is currently used for the `XSAVE*` and `XRSTOR*` instructions.
- `MEMORY_UNKNOWN_VALUE`: The value of the access hasn't been captured. This flag will be used when the access size is greater than the size of a `rword`. It's also used for instructions with `REP` in `X86` and `X86_64`.

Options

The `options` of the VM allow changing some internal mechanisms of QBDI. For most uses of QBDI, no option needs to be specified. The options are specified when the VM is created and can be changed with `setOptions` when the VM is not running. After changing the options, the cache needs to be cleared to apply the changes to all the instructions.

- `OPT_DISABLE_FPR`: This option disables the Floating Point Registers support. QBDI will not back up and restore any `FPRState` registers.
- `OPT_DISABLE_OPTIONAL_FPR`: if `OPT_DISABLE_FPR` is not enabled, this option will force the `FPRState` to be restored and saved before and after any instruction. By default, QBDI will try to detect the instructions that make use of floating point registers and only restore for these precise instructions.
- `OPT_ATT_SYNTAX`: For X86 and X86_64 architectures, this option changes the syntax of `InstAnalysis.disassembly` to AT&T instead of the Intel one.

2.5.2 C API

Introduction

The C API offers bindings over the C++ API. The VM class' methods are replaced by C functions that receive an object of type `VMInstanceRef` as a first parameter.

This API is compatible with *QBDIPreload* on Linux and macOS.

VM object

type **VMInstanceRef**

An abstract pointer to the VM object.

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

Create and initialize a VM instance.

Parameters

- **instance** – **[out]** VM instance created.
- **cpu** – **[in]** 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** – **[in]** A NULL terminated array of C strings specifying the attributes of the cpu model. If NULL, no additional features are specified.
- **opts** – **[in]** The options to enable in the VM

void **qbdi_terminateVM**(*VMInstanceRef* instance)

Destroy an instance of VM. This method mustn't be called when the VM runs.

Parameters

instance – **[in]** VM instance.

Options

Options **qbdi_getOptions**(*VMInstanceRef* instance)

Get the current Options

Parameters

instance – **[in]** VM instance.

Returns

The current options of the VM

void **qbdi_setOptions**(*VMInstanceRef* instance, *Options* options)

Set the Options This method mustn't be called when the VM runs.

Parameters

- **instance** – **[in]** VM instance.
- **options** – **[in]** The new options of the VM.

State management

GPRState ***qbdi_getGPRState**(*VMInstanceRef* instance)

Obtain the current general purpose register state.

Parameters
instance – **[in]** VM instance.

Returns
A structure containing the General Purpose Registers state.

FPRState ***qbdi_getFPRState**(*VMInstanceRef* instance)

Obtain the current floating point register state.

Parameters
instance – **[in]** VM instance.

Returns
A structure containing the Floating Point Registers state.

void **qbdi_setGPRState**(*VMInstanceRef* instance, *GPRState* *gprState)

Set the GPR state

Parameters

- **instance** – **[in]** VM instance.
- **gprState** – **[in]** A structure containing the General Purpose Registers state.

void **qbdi_setFPRState**(*VMInstanceRef* instance, *FPRState* *fprState)

Set the FPR state

Parameters

- **instance** – **[in]** VM instance.
- **fprState** – **[in]** A structure containing the Floating Point Registers state.

Instrumentation range

Addition

void **qbdi_addInstrumentedRange**(*VMInstanceRef* instance, *rword* start, *rword* end)

Add an address range to the set of instrumented address ranges.

Parameters

- **instance** – **[in]** VM instance.
- **start** – **[in]** Start address of the range (included).
- **end** – **[in]** End address of the range (excluded).

bool **qbdi_addInstrumentedModule**(*VMInstanceRef* instance, const char *name)

Add the executable address ranges of a module to the set of instrumented address ranges.

Parameters

- **instance** – **[in]** VM instance.
- **name** – **[in]** The module's name.

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

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.

Parameters

- **instance** – **[in]** VM instance.
- **addr** – **[in]** An address contained by module's range.

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

bool **qbdi_instrumentAllExecutableMaps**(*VMInstanceRef* instance)

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

Parameters
instance – **[in]** VM instance.

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

Removal

void **qbdi_removeInstrumentedRange**(*VMInstanceRef* instance, *rword* start, *rword* end)

Remove an address range from the set of instrumented address ranges.

Parameters

- **instance** – **[in]** VM instance.
- **start** – **[in]** Start address of the range (included).
- **end** – **[in]** End address of the range (excluded).

bool **qbdi_removeInstrumentedModule**(*VMInstanceRef* instance, const char *name)

Remove the executable address ranges of a module from the set of instrumented address ranges.

Parameters

- **instance** – **[in]** VM instance.
- **name** – **[in]** The module's name.

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

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.

Parameters

- **instance** – **[in]** VM instance.
- **addr** – **[in]** An address contained by module's range.

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

void **qbdi_removeAllInstrumentedRanges**(*VMInstanceRef* instance)

Remove all instrumented ranges.

Parameters
instance – **[in]** VM instance.

Callback management

InstCallback

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

Register a callback event for a specific instruction event.

Parameters

- **instance** – **[in]** VM instance.
- **pos** – **[in]** Relative position of the event callback (QBDI_PREINST / QBDI_POSTINST).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

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

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

Register a callback for when a specific address is executed.

Parameters

- **instance** – **[in]** VM instance.
- **address** – **[in]** Code address which will trigger the callback.
- **pos** – **[in]** Relative position of the callback (QBDI_PREINST / QBDI_POSTINST).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

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

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

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

Parameters

- **instance** – **[in]** VM instance.
- **start** – **[in]** Start of the address range which will trigger the callback.
- **end** – **[in]** End of the address range which will trigger the callback.
- **pos** – **[in]** Relative position of the callback (QBDI_PREINST / QBDI_POSTINST).

- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

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

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

Register a callback event if the instruction matches the mnemonic.

Parameters

- **instance** – **[in]** VM instance.
- **mnemonic** – **[in]** Mnemonic to match.
- **pos** – **[in]** Relative position of the event callback (QBDI_PREINST / QBDI_POSTINST).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

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

VMEvent

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

Register a callback event for a specific VM event.

Parameters

- **instance** – **[in]** VM instance.
- **mask** – **[in]** A mask of VM event type which will trigger the callback.
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.

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

MemoryAccess

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

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

Parameters

- **instance** – **[in]** VM instance.
- **type** – **[in]** A mode bitfield: either QBDI_MEMORY_READ, QBDI_MEMORY_WRITE or both (QBDI_MEMORY_READ_WRITE).
- **cbk** – **[in]** A function pointer to the callback.

- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

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

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.

Parameters

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

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

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.

Parameters

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

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

InstrRuleCallback

uint32_t **qbdi_addInstrRule**(*VMInstanceRef* instance, *InstrRuleCallbackC* cbk, *AnalysisType* type, void *data)

Add a custom instrumentation rule to the VM.

Parameters

- **instance** – **[in]** VM instance.
- **cbk** – **[in]** A function pointer to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback
- **data** – **[in]** User defined data passed to the callback.

Returns

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

uint32_t **qbdi_addInstrRuleRange**(*VMInstanceRef* instance, *rword* start, *rword* end, *InstrRuleCallbackC* cbk, *AnalysisType* type, void *data)

Add a custom instrumentation rule to the VM for a range of address

Parameters

- **instance** – **[in]** VM instance.
- **start** – **[in]** Begin of the range of address where apply the rule
- **end** – **[in]** End of the range of address where apply the rule
- **cbk** – **[in]** A function pointer to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback
- **data** – **[in]** User defined data passed to the callback.

Returns

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

Removal

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

Remove an instrumentation.

Parameters

- **instance** – **[in]** VM instance.
- **id** – **[in]** The id of the instrumentation to remove.

Returns

True if instrumentation has been removed.

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

Remove all the registered instrumentations.

Parameters

instance – **[in]** VM instance.

Run

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). This method mustn't be called when the VM already runs.

Parameters

- **instance** – **[in]** VM instance.
- **start** – **[in]** Address of the first instruction to execute.
- **stop** – **[in]** Stop the execution when this instruction is reached.

Returns

True if at least one block has been executed.

bool **qbdi_call**(*VMInstanceRef* instance, *rword* *retval, *rword* function, uint32_t argNum, ...)

Call a function using the DBI (and its current state). This method mustn't be called when the VM already runs.

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);
qbdi_alignedFree(fakestack);
```

Parameters

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

Returns

True if at least one block has been executed.

bool **qbdi_callA**(*VMInstanceRef* instance, *rword* *retval, *rword* function, uint32_t argNum, const *rword* *args)

Call a function using the DBI (and its current state). This method mustn't be called when the VM already runs.

Parameters

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

- **args** – **[in]** Arguments as an array of rword values.

Returns
True if at least one block has been executed.

bool **qbdi_callV**(*VMInstanceRef* instance, *rword* *retval, *rword* function, uint32_t argNum, va_list ap)

Call a function using the DBI (and its current state). This method mustn't be called when the VM already runs.

Parameters

- **instance** – **[in]** VM instance.
- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **argNum** – **[in]** The number of arguments in the variadic list.
- **ap** – **[in]** An stdarg va_list object.

Returns
True if at least one block has been executed.

bool **qbdi_switchStackAndCall**(*VMInstanceRef* instance, *rword* *retval, *rword* function, uint32_t stackSize, uint32_t argNum, ...)

Switch the stack and call a function using the DBI (and its current state). This method will allocate a new stack and switch to this stack. The remaining space on the current stack will be use by the called method. This method mustn't be called if the VM already runs. The stack pointer in the state must'nt be used after the end of this method.

Example:

```
// perform (with QBDI) a call similar to (*funcPtr)(42);
VMInstanceRef vm;
qbdi_initVM(&vm, NULL, NULL);
GPRState* gprState = qbdi_getGPRState(vm);
qbdi_addInstrumentedModuleFromAddr(vm, funcPtr);
rword retVal;
qbdi_switchStackAndCall(vm, &retVal, funcPtr, 0x20000, 1, 42);
```

Parameters

- **instance** – **[in]** VM instance.
- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **stackSize** – **[in]** The size of the stack for the engine.
- **argNum** – **[in]** The number of arguments in the variadic list.
- **...** – **[in]** A variadic list of arguments.

Returns
True if at least one block has been executed.

bool **qbdi_switchStackAndCallA**(*VMInstanceRef* instance, *rword* *retval, *rword* function, uint32_t stackSize, uint32_t argNum, const *rword* *args)

Switch the stack and call a function using the DBI (and its current state). This method mustn't be called if the VM already runs. The stack pointer in the state must'nt be used after the end of this method.

Parameters

- **instance** – **[in]** VM instance.
- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **stackSize** – **[in]** The size of the stack for the engine.
- **argNum** – **[in]** The number of arguments in the variadic list.
- **args** – **[in]** Arguments as an array of rword values.

Returns

True if at least one block has been executed.

bool **qbdi_switchStackAndCallV**(*VMInstanceRef* instance, *rword* *retval, *rword* function, uint32_t stackSize, uint32_t argNum, va_list ap)

Switch the stack and call a function using the DBI (and its current state). This method mustn't be called if the VM already runs. The stack pointer in the state must'nt be used after the end of this method.

Parameters

- **instance** – **[in]** VM instance.
- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **stackSize** – **[in]** The size of the stack for the engine.
- **argNum** – **[in]** The number of arguments in the variadic list.
- **ap** – **[in]** An stdarg va_list object.

Returns

True if at least one block has been executed.

InstAnalysis

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

Obtain the analysis of the current instruction. 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. This method must only be used in an InstCallback.

Parameters

- **instance** – **[in]** VM instance.
- **type** – **[in]** Properties to retrieve during analysis.

Returns

A *InstAnalysis* structure containing the analysis result.

const *InstAnalysis* ***qbdi_getCachedInstAnalysis**(const *VMInstanceRef* instance, *rword* address, *AnalysisType* type)

Obtain the analysis of a cached instruction. Analysis results are cached in the VM. The validity of the returned pointer is only guaranteed until the end of the callback or a call to a noconst method of the VM instance.

Parameters

- **instance** – **[in]** VM instance.
- **address** – **[in]** The address of the instruction to analyse.
- **type** – **[in]** Properties to retrieve during analysis.

Returns

A *InstAnalysis* structure containing the analysis result. null if the instruction isn't in the cache.

MemoryAccess

MemoryAccess ***qbdi_getInstMemoryAccess**(*VMInstanceRef* instance, size_t *size)

Obtain the memory accesses made by the last executed instruction. The method should be called in an InstCallback. Return NULL and a size of 0 if the instruction made no memory access.

Parameters

- **instance** – **[in]** VM instance.
- **size** – **[out]** Will be set to the number of elements in the returned array.

Returns

An array of memory accesses made by the instruction.

MemoryAccess ***qbdi_getBBMemoryAccess**(*VMInstanceRef* instance, size_t *size)

Obtain the memory accesses made by the last executed basic block. The method should be called in a VMCallback with QBDI_SEQUENCE_EXIT. Return NULL and a size of 0 if the basic block made no memory access.

Parameters

- **instance** – **[in]** VM instance.
- **size** – **[out]** Will be set to the number of elements in the returned array.

Returns

An array of memory accesses made by the basic block.

bool **qbdi_recordMemoryAccess**(*VMInstanceRef* instance, *MemoryAccessType* type)

Add instrumentation rules to log memory access using inline instrumentation and instruction shadows.

Parameters

- **instance** – **[in]** VM instance.
- **type** – **[in]** Memory mode bitfield to activate the logging for: either QBDI_MEMORY_READ, QBDI_MEMORY_WRITE or both (QBDI_MEMORY_READ_WRITE).

Returns

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

Cache management

bool **qbdi_precacheBasicBlock**(*VMInstanceRef* instance, *rword* pc)

Pre-cache a known basic block This method mustn't be called when the VM runs.

Parameters

- **instance** – **[in]** VM instance.
- **pc** – **[in]** Start address of a basic block

Returns

True if basic block has been inserted in cache.

void **qbdi_clearCache**(*VMInstanceRef* instance, *rword* start, *rword* end)

Clear a specific address range from the translation cache.

Parameters

- **instance** – **[in]** VM instance.
- **start** – **[in]** Start of the address range to clear from the cache.
- **end** – **[in]** End of the address range to clear from the cache.

void **qbdi_clearAllCache**(*VMInstanceRef* instance)

Clear the entire translation cache.

Parameters

instance – **[in]** VM instance.

Register state

type **rword**

An integer of the size of a register

- uint32_t for X86
- uint64_t for X86_64

struct **GPRState**

General Purpose Register context.

For X86 architecture:

```
typedef struct QBDI_ALIGNED(4) {
  rword eax;
  rword ebx;
  rword ecx;
  rword edx;
  rword esi;
  rword edi;
  rword ebp;
  rword esp;
  rword eip;
  rword eflags;
} GPRState;
```

For X86_64 architecture:

```
typedef struct QBDI_ALIGNED(8) {
  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;
  // Only backup and restore with OPT_ENABLE_FS_GS
  rword fs;
  rword gs;
} GPRState;
```

For ARM architecture:

```
/*! ARM General Purpose Register context.
 */
typedef struct QBDI_ALIGNED(4) {
  rword r0;
  rword r1;
  rword r2;
  rword r3;
  rword r4;
  rword r5;
  rword r6;
  rword r7;
  rword r8;
  rword r9;
  rword r10;
  rword r11;
  rword r12;
  rword sp;
  rword lr;
  rword pc;
  rword cpsr;

  /* Internal CPU state
   * Local monitor state for exclusive load/store instruction
   */
```

(continues on next page)

(continued from previous page)

```
  struct {
    rword addr;
    rword enable; /* 0=>disable,
                   * 1=>enable by ldrexb,
                   * 2=>enable by ldrexh,
                   * 4=>enable by ldrex,
                   * 8=>enable by ldrexd
                   */
  } localMonitor;

} GPRState;
```

For AARCH64 architecture:

```
/*! ARM General Purpose Register context.
 */
typedef struct QBDI_ALIGNED(8) {
  rword x0;
  rword x1;
  rword x2;
  rword x3;
  rword x4;
  rword x5;
  rword x6;
  rword x7;
  rword x8;
  rword x9;
  rword x10;
  rword x11;
  rword x12;
  rword x13;
  rword x14;
  rword x15;
  rword x16;
  rword x17;
  rword x18;
  rword x19;
  rword x20;
  rword x21;
  rword x22;
  rword x23;
  rword x24;
  rword x25;
  rword x26;
  rword x27;
  rword x28;
  rword x29; // FP (x29)
  rword lr;  // LR (x30)

  rword sp;
  rword nzcv;
```

(continues on next page)

(continued from previous page)

```
  rword pc;
  // ? rword daif; ?

  /* Internal CPU state
   * Local monitor state for exclusive load/store instruction
   */
  struct {
    rword addr;
    rword enable; /* 0=>disable, 1=>exclusive state, use a rword to not break
                     align */
  } localMonitor;

} GPRState;
```

struct **FPRState**

Floating Point Register context.

For X86 architecture:

```
typedef struct QBDI_ALIGNED(16) {
  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  */
```

(continues on next page)

(continued from previous page)

```
  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;
```

For X86_64 architecture:

```
typedef struct QBDI_ALIGNED(16) {
  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  */
```

(continues on next page)

(continued from previous page)

```
  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;
```

For ARM architecture:

```
/*! ARM Floating Point Register context.
 */
typedef union {
  float QBDI_ALIGNED(8) s[32];
  double QBDI_ALIGNED(8) d[QBDI_NUM_FPR];
  uint8_t QBDI_ALIGNED(8) q[QBDI_NUM_FPR / 2][16];
} FPRStateVReg;

typedef struct QBDI_ALIGNED(8) {
  FPRStateVReg vreg;

  rword fpscr;
} FPRState;
```

For AARCH64 architecture:

```
/*! ARM Floating Point Register context.
 */
typedef struct QBDI_ALIGNED(8) {
```

(continues on next page)

(continued from previous page)

```
  __uint128_t v0;
  __uint128_t v1;
  __uint128_t v2;
  __uint128_t v3;

  __uint128_t v4;
  __uint128_t v5;
  __uint128_t v6;
  __uint128_t v7;

  __uint128_t v8;
  __uint128_t v9;
  __uint128_t v10;
  __uint128_t v11;

  __uint128_t v12;
  __uint128_t v13;
  __uint128_t v14;
  __uint128_t v15;

  __uint128_t v16;
  __uint128_t v17;
  __uint128_t v18;
  __uint128_t v19;

  __uint128_t v20;
  __uint128_t v21;
  __uint128_t v22;
  __uint128_t v23;

  __uint128_t v24;
  __uint128_t v25;
  __uint128_t v26;
  __uint128_t v27;

  __uint128_t v28;
  __uint128_t v29;
  __uint128_t v30;
  __uint128_t v31;

  rword fpcr;
  rword fpsr;
} FPRState;
```

struct **MMSTReg**

Public Members

char **reg**[10]

char **rsrv**[6]

struct **FPControl**

Public Members

uint16_t **invalid**

uint16_t **denorm**

uint16_t **zdiv**

uint16_t **ovrfl**

uint16_t **undfl**

uint16_t **precis**

uint16_t **__pad0__**

uint16_t **pc**

uint16_t **rc**

uint16_t **__pad1__**

uint16_t **__pad2__**

struct **FPStatus**

Public Members

uint16_t **invalid**

uint16_t **denorm**

uint16_t **zdiv**

uint16_t **ovrfl**

uint16_t **undfl**

uint16_t **precis**

uint16_t **stkflt**

uint16_t **errsumm**

uint16_t **c0**

uint16_t **c1**

uint16_t **c2**

uint16_t **tos**

uint16_t **c3**

uint16_t **busy**

REG_RETURN

REG_BP

REG_SP

REG_PC

NUM_GPR

Callback

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

Instruction callback function type.

Param vm
[in] VM instance of the callback.

Param gprState
[in] A structure containing the state of the General Purpose Registers. Modifying it affects the VM execution accordingly.

Param fprState
[in] A structure containing the state of the Floating Point Registers. Modifying it affects the VM execution accordingly.

Param data
[in] User defined data which can be defined when registering the callback.

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

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

VM callback function type.

Param vm
[in] VM instance of the callback.

Param vmState
[in] A structure containing the current state of the VM.

Param gprState
[in] A structure containing the state of the General Purpose Registers. Modifying it affects the VM execution accordingly.

Param fprState
[in] A structure containing the state of the Floating Point Registers. Modifying it affects the VM execution accordingly.

Param data
[in] User defined data which can be defined when registering the callback.

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

typedef void (***InstrRuleCallbackC**)(*VMInstanceRef* vm, const *InstAnalysis* *inst, *InstrRuleDataVec* cbks, void *data)

Instrumentation rule callback function type for C API.

Param vm
[in] VM instance of the callback.

Param inst
[in] AnalysisType of the current instrumented Instruction.

Param cbks
[in] An object to add the callback to apply for this instruction. InstCallback can be add with qbdi_addInstrRuleData.

Param data
[in] User defined data which can be defined when registering the callback.

void **qbdi_addInstrRuleData**(*InstrRuleDataVec* cbks, *InstPosition* position, *InstCallback* cbk, void *data, int priority)

Add a callback for the current instruction

Parameters

- **cbks** – **[in]** InstrRuleDataVec given in argument
- **position** – **[in]** Relative position of the callback (QBDI_PREINST / QBDI_POSTINST).
- **cbk** – **[in]** A function pointer to the callback
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** Priority of the callback

type **InstrRuleDataVec**

An abstract type to append InstCallback for the current instruction

enum **InstPosition**

Position relative to an instruction.

Values:

enumerator **QBDI_PREINST**

Positioned before the instruction.

enumerator **QBDI_POSTINST**

Positioned after the instruction.

enum **CallbackPriority**

Priority of callback

A callback with an higher priority will be call before a callback with a lower priority.

ie:

a. CBpre(p = 10)
b. CBpre(p = 0)
c. CBpre(p = -10)
d. instrumented instruction
e. CBpost(p = 10)
f. CBpost(p = 0)
g. CBpost(p = -10)

When the *MemoryAccess* API is used in a callback, the priority of the callback must not be greater than PRIORITY_MEMACCESS_LIMIT

Values:

enumerator **QBDI_PRIORITY_DEFAULT**

Default priority for callback

enumerator **QBDI_PRIORITY_MEMACCESS_LIMIT**

Maximum priority if getInstMemoryAccess is used in the callback

enum **VMAction**

The callback results.

Values:

enumerator **QBDI_CONTINUE**

The execution of the basic block continues.

enumerator **QBDI_SKIP_INST**

Available only with PREINST InstCallback. The instruction and the remained PREINST callbacks are skip. The execution continue with the POSTINST instruction.

We recommand to used this result with a low priority PREINST callback in order to emulate the instruction without skipping the POSTINST callback.

enumerator **QBDI_SKIP_PATCH**

Available only with InstCallback. The current instruction and the reminding callback (PRE and POST) are skip. The execution continues to the next instruction.

For instruction that change the instruction pointer (jump/call/ret), BREAK_TO_VM must be used insted of SKIP.

SKIP can break the record of *MemoryAccess* for the current instruction.

enumerator **QBDI_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.

enumerator **QBDI_STOP**

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

InstAnalysis

enum **AnalysisType**

Instruction analysis type

Values:

enumerator **QBDI_ANALYSIS_INSTRUCTION**

Instruction analysis (address, mnemonic, ...)

enumerator **QBDI_ANALYSIS_DISASSEMBLY**

Instruction disassembly

enumerator **QBDI_ANALYSIS_OPERANDS**

Instruction operands analysis

enumerator **QBDI_ANALYSIS_SYMBOL**

Instruction symbol

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)

CPUMode cpuMode
: Instruction CPU mode

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 isMoveImm
: true if this instruction is a move immediate (including conditional moves) instruction.

bool mayLoad
: true if QBDI detects a load for this instruction

bool mayStore
: true if QBDI detects a store for this instruction

uint32_t loadSize
: size of the expected read access, may be 0 with mayLoad if the size isn't determined

uint32_t storeSize
: size of the expected write access, may be 0 with mayStore if the size isn't determined

ConditionType **condition**

Condition associated with the instruction

char ***disassembly**

Instruction disassembly (warning: NULL if !ANALYSIS_DISASSEMBLY)

RegisterAccessType **flagsAccess**

Flag access type (noaccess, r, w, rw) (warning: REGISTER_UNUSED if !ANALYSIS_OPERANDS)

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 ***symbolName**

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

uint32_t **symbolOffset**

Instruction symbol offset

const char ***moduleName**

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

uint32_t **analysisType**

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

enum **ConditionType**

Instruction Condition

Values:

enumerator **QBDI_CONDITION_NONE**

The instruction is unconditionnal

enumerator **QBDI_CONDITION_ALWAYS**

The instruction is always true

enumerator **QBDI_CONDITION_NEVER**

The instruction is always false

enumerator **QBDI_CONDITION_EQUALS**

Equals (‘==’)

enumerator **QBDI_CONDITION_NOT_EQUALS**
Not Equals (‘!=’)

enumerator **QBDI_CONDITION_ABOVE**
Above (‘>’ unsigned)

enumerator **QBDI_CONDITION_BELOW_EQUALS**
Below or Equals (‘<=’ unsigned)

enumerator **QBDI_CONDITION_ABOVE_EQUALS**
Above or Equals (‘>=’ unsigned)

enumerator **QBDI_CONDITION_BELOW**
Below (‘<’ unsigned)

enumerator **QBDI_CONDITION_GREAT**
Great (‘>’ signed)

enumerator **QBDI_CONDITION_LESS_EQUALS**
Less or Equals (‘<=’ signed)

enumerator **QBDI_CONDITION_GREAT_EQUALS**
Great or Equals (‘>=’ signed)

enumerator **QBDI_CONDITION_LESS**
Less (‘<’ signed)

enumerator **QBDI_CONDITION_EVEN**
Even

enumerator **QBDI_CONDITION_ODD**
Odd

enumerator **QBDI_CONDITION_OVERFLOW**
Overflow

enumerator **QBDI_CONDITION_NOT_OVERFLOW**
Not Overflow

enumerator **QBDI_CONDITION_SIGN**
Sign

enumerator **QBDI_CONDITION_NOT_SIGN**
Not Sign

struct **OperandAnalysis**

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

Public Members

OperandType **type**

Operand type

OperandFlag **flag**

Operand flag

sword **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)

int16_t **regCtxIdx**

Register index in VM state (< 0 if not know)

const char ***regName**

Register name

RegisterAccessType **regAccess**

Register access type (r, w, rw)

enum **OperandType**

Operand type

Values:

enumerator **QBDI_OPERAND_INVALID**

Invalid operand

enumerator **QBDI_OPERAND_IMM**

Immediate operand

enumerator **QBDI_OPERAND_GPR**

Register operand

enumerator **QBDI_OPERAND_PRED**

Predicate operand

enumerator **QBDI_OPERAND_FPR**

Float register operand

enumerator **QBDI_OPERAND_SEG**

Segment or unsupported register operand

enum **OperandFlag**

Values:

enumerator **QBDI_OPERANDFLAG_NONE**

No flag

enumerator **QBDI_OPERANDFLAG_ADDR**

The operand is used to compute an address

enumerator **QBDI_OPERANDFLAG_PCREL**

The value of the operand is PC relative

enumerator **QBDI_OPERANDFLAG_UNDEFINED_EFFECT**

The operand role isn't fully defined

enumerator **QBDI_OPERANDFLAG_IMPLICIT**

The operand is implicit

enum **RegisterAccessType**

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

Values:

enumerator **QBDI_REGISTER_UNUSED**

Unused register

enumerator **QBDI_REGISTER_READ**

Register read access

enumerator **QBDI_REGISTER_WRITE**

Register write access

enumerator **QBDI_REGISTER_READ_WRITE**

Register read/write access

MemoryAccess

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

uint16_t **size**

Size of memory access (in bytes)

MemoryAccessType **type**

Memory access type (READ / WRITE)

MemoryAccessFlags **flags**

Memory access flags

enum **MemoryAccessType**

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

Values:

enumerator **QBDI_MEMORY_READ**

Memory read access

enumerator **QBDI_MEMORY_WRITE**

Memory write access

enumerator **QBDI_MEMORY_READ_WRITE**

Memory read/write access

enum **MemoryAccessFlags**

Memory access flags

Values:

enumerator **QBDI_MEMORY_NO_FLAGS**

enumerator **QBDI_MEMORY_UNKNOWN_SIZE**

The size of the access isn't known.

enumerator **QBDI_MEMORY_MINIMUM_SIZE**

The given size is a minimum size.

enumerator **QBDI_MEMORY_UNKNOWN_VALUE**

The value of the access is unknown or hasn't been retrived.

VMEvent

enum **VMEvent**

Values:

enumerator **QBDI_NO_EVENT**

enumerator **QBDI_SEQUENCE_ENTRY**

Triggered when the execution enters a sequence.

enumerator **QBDI_SEQUENCE_EXIT**

Triggered when the execution exits from the current sequence.

enumerator **QBDI_BASIC_BLOCK_ENTRY**

Triggered when the execution enters a basic block.

enumerator **QBDI_BASIC_BLOCK_EXIT**

Triggered when the execution exits from the current basic block.

enumerator **QBDI_BASIC_BLOCK_NEW**

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

enumerator **QBDI_EXEC_TRANSFER_CALL**

Triggered when the ExecBroker executes an execution transfer.

enumerator **QBDI_EXEC_TRANSFER_RETURN**

Triggered when the ExecBroker returns from an execution transfer.

enumerator **QBDI_SYSCALL_ENTRY**

Not implemented.

enumerator **QBDI_SYSCALL_EXIT**

Not implemented.

enumerator **QBDI_SIGNAL**

Not implemented.

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.

Memory management

Allocation

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

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

Parameters

- **size** – **[in]** Allocation size in bytes.
- **align** – **[in]** Base address alignement in bytes.

Returns

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

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().

Parameters

- **ctx** – **[in]** GPRState which will be setup to use the new stack.

- **stackSize** – **[in]** Size of the stack to be allocated.
- **stack** – **[out]** The newly allocated stack pointer will be returned in the variable pointed by stack.

Returns
True if stack allocation was successfull.

void **qbdi_alignedFree**(void *ptr)

Free a block of aligned memory allocated with alignedAlloc.

Parameters
ptr – **[in]** Pointer to the allocated memory.

void **qbdi_simulateCall**(*GPRState* *ctx, *rword* returnAddress, uint32_t argNum, ...)

Simulate a call by modifying the stack and registers accordingly.

Parameters

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

void **qbdi_simulateCallV**(*GPRState* *ctx, *rword* returnAddress, uint32_t argNum, va_list ap)

Simulate a call by modifying the stack and registers accordingly (stdarg version).

Parameters

- **ctx** – **[in]** GPRState where the simulated call will be setup. The state needs to point to a valid stack for example setup with allocateVirtualStack().
- **returnAddress** – **[in]** Return address of the call to simulate.
- **argNum** – **[in]** The number of arguments in the va_list object.
- **ap** – **[in]** An stdarg va_list object.

void **qbdi_simulateCallA**(*GPRState* *ctx, *rword* returnAddress, uint32_t argNum, const *rword* *args)

Simulate a call by modifying the stack and registers accordingly (C array version).

Parameters

- **ctx** – **[in]** GPRState where the simulated call will be setup. The state needs to point to a valid stack for example setup with allocateVirtualStack().
- **returnAddress** – **[in]** Return address of the call to simulate.
- **argNum** – **[in]** The number of arguments in the array args.
- **args** – **[in]** An array or arguments.

Exploration

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.

Parameters
size – **[out]** Will be set to the number of strings in the returned array.

Returns
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.

qbdi_MemoryMap *qbdi_getCurrentProcessMaps(bool full_path, size_t *size)

Get a list of all the memory maps (regions) of the current process.

Parameters
- **full_path** – **[in]** Return the full path of the module in name field
- **size** – **[out]** Will be set to the number of strings in the returned array.

Returns
An array of MemoryMap object.

qbdi_MemoryMap *qbdi_getRemoteProcessMaps(*rword* pid, bool full_path, size_t *size)

Get a list of all the memory maps (regions) of a process.

Parameters
- **pid** – **[in]** The identifier of the process.
- **full_path** – **[in]** Return the full path of the module in name field
- **size** – **[out]** Will be set to the number of strings in the returned array.

Returns
An array of MemoryMap object.

void qbdi_freeMemoryMapArray(*qbdi_MemoryMap* *arr, size_t size)

Free an array of memory maps objects.

Parameters
- **arr** – **[in]** An array of MemoryMap object.
- **size** – **[in]** Number of elements in the array.

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:

enumerator **QBDI_PF_NONE**

No access

enumerator **QBDI_PF_READ**

Read access

enumerator **QBDI_PF_WRITE**

Write access

enumerator **QBDI_PF_EXEC**

Execution access

Other globals

enum **Options**

Note: some value are available only for some architecture

Values for all architecture :

enumerator **NO_OPT**

Default Value

enumerator **OPT_DISABLE_FPR**

Disable all operation on FPU (SSE, AVX, SIMD). May break the execution if the target use the FPU

enumerator **OPT_DISABLE_OPTIONAL_FPR**

Disable context switch optimisation when the target execblock doesn't used FPR

Values for AARCH64 and ARM only :

enumerator **OPT_DISABLE_LOCAL_MONITOR**

Disable the local monitor for instruction like strex

Values for AARCH64 only :

enumerator **OPT_BYPASS_PAUTH**

Disable pointeur authentication

enumerator **OPT_ENABLE_BTI**

Enable BTI on instrumented code

Values for ARM only :

enumerator **OPT_DISABLE_D16_D31**

Disable the used of D16-D31 register

enumerator **OPT_ARMv4**

Change between ARM and Thumb as an ARMv4 CPU

enumerator **OPT_ARMv5T_6**

Change between ARM and Thumb as an ARMv5T or ARMv6 CPU

enumerator **OPT_ARMv7**

Change between ARM and Thumb as an ARMv7 CPU (default)

enumerator **OPT_ARM_MASK**

When apply *OPT_ARMv4*, *OPT_ARMv5T_6* or *OPT_ARMv7*, this mask must be clear.

Values for X86 and X86_64 only :

enumerator **OPT_ATT_SYNTAX**

Used the AT&T syntax for instruction disassembly

Values for X86_64 only :

enumerator **OPT_ENABLE_FS_GS**

Enable Backup/Restore of FS/GS segment. This option uses the instructions (RD|WR)(FS|GS)BASE that must be supported by the operating system

enum **VMError**

QBDI Error values

Values:

enumerator **QBDI_INVALID_EVENTID**

Mark a returned event id as invalid

Miscellaneous

Version

const char ***qbdi_getVersion**(uint32_t *version)

Return QBDI version.

Parameters

version – **[out]** QBDI version encoded as an unsigned integer (0xMMmmpp).

Returns
QBDI version as a string (major.minor.patch).

Log

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:

enumerator **QBDI_DEBUG**

Debug logs

enumerator **QBDI_INFO**

Info logs (default)

enumerator **QBDI_WARNING**

Warning logs

enumerator **QBDI_ERROR**

Error logs

enumerator **QBDI_DISABLE**

Disable logs message

void **qbdi_setLogFile**(const char *filename, bool truncate)

Redirect logs to a file.

Parameters

- **filename** – **[in]** the path of the file to append the log
- **truncate** – **[in]** Set to true to clear the file before append the log

void **qbdi_setLogConsole**()

Write log to the console (stderr)

void **qbdi_setLogDefault**()

Write log to the default location (stderr for linux, android_logger for android)

void **qbdi_setLogPriority**(*LogPriority* priority)

Enable logs matching priority.

Parameters
priority – **[in]** Filter logs with greater or equal priority.

2.5.3 C++ API

Introduction

The C++ API is the primary API of QBDI. The API is compatible with *QBDIPreload* on Linux and macOS.

VM class

class **VM**

Public Functions

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

Construct a new *VM* for a given CPU with specific attributes

Parameters

- **cpu** – **[in]** The name of the CPU
- **mattrs** – **[in]** A list of additional attributes
- **opts** – **[in]** The options to enable in the *VM*

VM(*VM* &&vm)

Move constructors. All the cache is keep. All registered callbacks will be called with the new pointer of the *VM*.

Parameters

vm – **[in]** The *VM* to move

VM &**operator=**(*VM* &&vm)

Move assignment operator All the cache is keep. All registered callbacks will be called with the new pointer of the *VM*. This operator mustn't be called when the target *VM* runs.

Parameters

vm – **[in]** The *VM* to move

VM(const *VM* &vm)

Copy constructors The state and the configuration is copied. The cache isn't duplicate. The assigned *VM* begin with an empty cache.

Parameters

vm – **[in]** The *VM* to copy

VM &**operator=**(const *VM* &vm)

Copy assignment operator The state and the configuration is copied. The cache isn't duplicate. The assigned *VM* begin with an empty cache. This operator mustn't be called when the target *VM* runs.

Parameters

vm – **[in]** The *VM* to copy

Options

Options QBDI::*VM*::**getOptions**() const

Get the current Options of the *VM*

void QBDI::*VM*::**setOptions**(*Options* options)

Set the Options of the *VM* This method mustn't be called when the *VM* runs.

If the new options is different that the current ones, the cache will be clear.

Parameters

options – **[in]** the new options of the *VM*

State management

GPRState *QBDI::*VM*::**getGPRState**() const

Obtain the current general purpose register state.

Returns

A structure containing the GPR state.

FPRState *QBDI::*VM*::**getFPRState**() const

Obtain the current floating point register state.

Returns

A structure containing the FPR state.

void QBDI::*VM*::**setGPRState**(const *GPRState* *gprState)

Set the GPR state

Parameters

gprState – **[in]** A structure containing the GPR state.

void QBDI::*VM*::**setFPRState**(const *FPRState* *fprState)

Set the FPR state

Parameters

fprState – **[in]** A structure containing the FPR state.

Instrumentation range

Addition

void QBDI::*VM*::**addInstrumentedRange**(*rword* start, *rword* end)

Add an address range to the set of instrumented address ranges.

Parameters

- **start** – **[in]** Start address of the range (included).
- **end** – **[in]** End address of the range (excluded).

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

Add the executable address ranges of a module to the set of instrumented address ranges.

Parameters
name – **[in]** The module's name.

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

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.

Parameters
addr – **[in]** An address contained by module's range.

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

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

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

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

Removal

void QBDI::*VM*::**removeInstrumentedRange**(*rword* start, *rword* end)

Remove an address range from the set of instrumented address ranges.

Parameters

- **start** – **[in]** Start address of the range (included).
- **end** – **[in]** End address of the range (excluded).

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

Remove the executable address ranges of a module from the set of instrumented address ranges.

Parameters
name – **[in]** The module's name.

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

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.

Parameters
addr – **[in]** An address contained by module's range.

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

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

Remove all instrumented ranges.

Callback management

InstCallback

uint32_t QBDI::*VM*::**addCodeCB**(*InstPosition* pos, *InstCallback* cbk, void *data, int priority = PRIORITY_DEFAULT)

Register a callback event for every instruction executed.

Parameters

- **pos** – **[in]** Relative position of the event callback (PREINST / POSTINST).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

Returns

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

uint32_t QBDI::*VM*::**addCodeCB**(*InstPosition* pos, *InstCbLambda* &&cbk, int priority = PRIORITY_DEFAULT)

uint32_t QBDI::*VM*::**addCodeCB**(*InstPosition* pos, const *InstCbLambda* &cbk, int priority = PRIORITY_DEFAULT)

Register a callback event for every instruction executed.

Parameters

- **pos** – **[in]** Relative position of the event callback (PREINST / POSTINST).
- **cbk** – **[in]** A lambda function to the callback
- **priority** – **[in]** The priority of the callback.

Returns

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

uint32_t QBDI::*VM*::**addCodeAddrCB**(*rword* address, *InstPosition* pos, *InstCallback* cbk, void *data, int priority = PRIORITY_DEFAULT)

Register a callback for when a specific address is executed.

Parameters

- **address** – **[in]** Code address which will trigger the callback.
- **pos** – **[in]** Relative position of the callback (PREINST / POSTINST).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

Returns

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

uint32_t QBDI::*VM*::**addCodeAddrCB**(*rword* address, *InstPosition* pos, *InstCbLambda* &&cbk, int priority = PRIORITY_DEFAULT)

uint32_t QBDI::*VM*::**addCodeAddrCB**(*rword* address, *InstPosition* pos, const *InstCbLambda* &cbk, int priority = PRIORITY_DEFAULT)

Register a callback for when a specific address is executed.

Parameters

- **address** – **[in]** Code address which will trigger the callback.
- **pos** – **[in]** Relative position of the callback (PREINST / POSTINST).
- **cbk** – **[in]** A lambda function to the callback
- **priority** – **[in]** The priority of the callback.

Returns

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

uint32_t QBDI::*VM*::**addCodeRangeCB**(*rword* start, *rword* end, *InstPosition* pos, *InstCallback* cbk, void *data, int priority = PRIORITY_DEFAULT)

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

Parameters

- **start** – **[in]** Start of the address range which will trigger the callback.
- **end** – **[in]** End of the address range which will trigger the callback.
- **pos** – **[in]** Relative position of the callback (PREINST / POSTINST).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

Returns

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

uint32_t QBDI::*VM*::**addCodeRangeCB**(*rword* start, *rword* end, *InstPosition* pos, *InstCbLambda* &&cbk, int priority = PRIORITY_DEFAULT)

uint32_t QBDI::*VM*::**addCodeRangeCB**(*rword* start, *rword* end, *InstPosition* pos, const *InstCbLambda* &cbk, int priority = PRIORITY_DEFAULT)

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

Parameters

- **start** – **[in]** Start of the address range which will trigger the callback.
- **end** – **[in]** End of the address range which will trigger the callback.
- **pos** – **[in]** Relative position of the callback (PREINST / POSTINST).
- **cbk** – **[in]** A lambda function to the callback
- **priority** – **[in]** The priority of the callback.

Returns

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

uint32_t QBDI::*VM*::**addMnemonicCB**(const char *mnemonic, *InstPosition* pos, *InstCallback* cbk, void *data, int priority = PRIORITY_DEFAULT)

Register a callback event if the instruction matches the mnemonic.

Parameters

- **mnemonic** – **[in]** Mnemonic to match.
- **pos** – **[in]** Relative position of the event callback (PREINST / POSTINST).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

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

uint32_t QBDI::*VM*::**addMnemonicCB**(const char *mnemonic, *InstPosition* pos, *InstCbLambda* &&cbk, int priority = PRIORITY_DEFAULT)

uint32_t QBDI::*VM*::**addMnemonicCB**(const char *mnemonic, *InstPosition* pos, const *InstCbLambda* &cbk, int priority = PRIORITY_DEFAULT)

Register a callback event if the instruction matches the mnemonic.

Parameters

- **mnemonic** – **[in]** Mnemonic to match.
- **pos** – **[in]** Relative position of the event callback (PREINST / POSTINST).
- **cbk** – **[in]** A lambda function to the callback
- **priority** – **[in]** The priority of the callback.

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

VMEvent

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

Register a callback event for a specific *VM* event.

Parameters

- **mask** – **[in]** A mask of *VM* event type which will trigger the callback.
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.

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

uint32_t QBDI::*VM*::**addVMEventCB**(*VMEvent* mask, *VMCbLambda* &&cbk)

uint32_t QBDI::*VM*::**addVMEventCB**(*VMEvent* mask, const *VMCbLambda* &cbk)

Register a callback event for a specific *VM* event.

Parameters

- **mask** – **[in]** A mask of *VM* event type which will trigger the callback.
- **cbk** – **[in]** A lambda function to the callback.

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

MemoryAccess

uint32_t QBDI::*VM*::**addMemAccessCB**(*MemoryAccessType* type, *InstCallback* cbk, void *data, int priority = PRIORITY_DEFAULT)

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

Parameters

- **type** – **[in]** A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.
- **priority** – **[in]** The priority of the callback.

Returns

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

uint32_t QBDI::*VM*::**addMemAccessCB**(*MemoryAccessType* type, *InstCbLambda* &&cbk, int priority = PRIORITY_DEFAULT)

uint32_t QBDI::*VM*::**addMemAccessCB**(*MemoryAccessType* type, const *InstCbLambda* &cbk, int priority = PRIORITY_DEFAULT)

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

Parameters

- **type** – **[in]** A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).
- **cbk** – **[in]** A lambda function to the callback
- **priority** – **[in]** The priority of the callback.

Returns

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

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. The callback has the default priority.

Parameters

- **address** – **[in]** Code address which will trigger the callback.
- **type** – **[in]** A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).
- **cbk** – **[in]** A function pointer to the callback.
- **data** – **[in]** User defined data passed to the callback.

Returns

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

uint32_t QBDI::*VM*::**addMemAddrCB**(*rword* address, *MemoryAccessType* type, *InstCbLambda* &&cbk)

uint32_t QBDI::*VM*::**addMemAddrCB**(*rword* address, *MemoryAccessType* type, const *InstCbLambda* &cbk)

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. The callback has the default priority.

Parameters

- **address** – **[in]** Code address which will trigger the callback.
- **type** – **[in]** A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).
- **cbk** – **[in]** A lambda function to the callback

Returns

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

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. The callback has the default priority.

Parameters

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

Returns

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

uint32_t QBDI::*VM*::**addMemRangeCB**(*rword* start, *rword* end, *MemoryAccessType* type, *InstCbLambda* &&cbk)

uint32_t QBDI::*VM*::**addMemRangeCB**(*rword* start, *rword* end, *MemoryAccessType* type, const *InstCbLambda* &cbk)

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. The callback has the default priority.

Parameters

- **start** – **[in]** Start of the address range which will trigger the callback.
- **end** – **[in]** End of the address range which will trigger the callback.
- **type** – **[in]** A mode bitfield: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).
- **cbk** – **[in]** A lambda function to the callback

Returns

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

InstrRuleCallback

uint32_t QBDI::*VM*::**addInstrRule**(*InstrRuleCallback* cbk, *AnalysisType* type, void *data)

Add a custom instrumentation rule to the *VM*.

Parameters

- **cbk** – **[in]** A function pointer to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback
- **data** – **[in]** User defined data passed to the callback.

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

uint32_t QBDI::*VM*::**addInstrRule**(*InstrRuleCbLambda* &&cbk, *AnalysisType* type)

uint32_t QBDI::*VM*::**addInstrRule**(const *InstrRuleCbLambda* &cbk, *AnalysisType* type)

Add a custom instrumentation rule to the *VM*.

Parameters

- **cbk** – **[in]** A lambda function to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback

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

uint32_t QBDI::*VM*::**addInstrRuleRange**(*rword* start, *rword* end, *InstrRuleCallback* cbk, *AnalysisType* type, void *data)

Add a custom instrumentation rule to the *VM* on a specify range

Parameters

- **start** – **[in]** Begin of the range of address where apply the rule
- **end** – **[in]** End of the range of address where apply the rule
- **cbk** – **[in]** A function pointer to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback
- **data** – **[in]** User defined data passed to the callback.

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

uint32_t QBDI::*VM*::**addInstrRuleRange**(*rword* start, *rword* end, *InstrRuleCbLambda* &&cbk, *AnalysisType* type)

uint32_t QBDI::*VM*::**addInstrRuleRange**(*rword* start, *rword* end, const *InstrRuleCbLambda* &cbk, *AnalysisType* type)

Add a custom instrumentation rule to the *VM* on a specify range

Parameters

- **start** – **[in]** Begin of the range of address where apply the rule
- **end** – **[in]** End of the range of address where apply the rule
- **cbk** – **[in]** A lambda function to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback

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

uint32_t QBDI::*VM*::**addInstrRuleRangeSet**(*RangeSet*<*rword*> range, *InstrRuleCallback* cbk, *AnalysisType* type, void *data)

Add a custom instrumentation rule to the *VM* on a specify set of range

Parameters

- **range** – **[in]** *Range* of address where apply the rule
- **cbk** – **[in]** A function pointer to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback
- **data** – **[in]** User defined data passed to the callback.

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

uint32_t QBDI::*VM*::**addInstrRuleRangeSet**(*RangeSet*<*rword*> range, *InstrRuleCbLambda* &&cbk, *AnalysisType* type)

uint32_t QBDI::*VM*::**addInstrRuleRangeSet**(*RangeSet*<*rword*> range, const *InstrRuleCbLambda* &cbk, *AnalysisType* type)

Add a custom instrumentation rule to the *VM* on a specify set of range

Parameters

- **range** – **[in]** *Range* of address where apply the rule
- **cbk** – **[in]** A lambda function to the callback
- **type** – **[in]** Analyse type needed for this instruction function pointer to the callback

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

Removal

bool QBDI::*VM*::**deleteInstrumentation**(uint32_t id)

Remove an instrumentation.

Parameters
id – **[in]** The id of the instrumentation to remove.

Returns
True if instrumentation has been removed.

void QBDI::*VM*::**deleteAllInstrumentations**()

Remove all the registered instrumentations.

Run

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

Start the execution by the DBI. This method mustn't be called if the *VM* already runs.

Parameters

- **start** – **[in]** Address of the first instruction to execute.
- **stop** – **[in]** Stop the execution when this instruction is reached.

Returns

True if at least one block has been executed.

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

Call a function using the DBI (and its current state). This method mustn't be called if the *VM* already runs.

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});
QBDI::alignedFree(fakestack);
```

Parameters

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

Returns

True if at least one block has been executed.

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

Call a function using the DBI (and its current state). This method mustn't be called if the *VM* already runs.

Parameters

- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **argNum** – **[in]** The number of arguments in the array of arguments.
- **args** – **[in]** An array of arguments.

Returns

True if at least one block has been executed.

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

Call a function using the DBI (and its current state). This method mustn't be called if the *VM* already runs.

Parameters

- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **argNum** – **[in]** The number of arguments in the variadic list.
- **ap** – **[in]** An stdarg va_list object.

Returns
True if at least one block has been executed.

bool QBDI::*VM*::**switchStackAndCall**(*rword* *retval, *rword* function, const std::vector<*rword*> &args = {}, uint32_t stackSize = 0x20000)

Switch the stack and call a function using the DBI (and its current state). This method will allocate a new stack and switch to this stack. The remaining space on the current stack will be use by the called method. This method mustn't be called if the *VM* already runs. The stack pointer in the state must'nt be used after the end of this method.

Example:

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

Parameters

- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **args** – **[in]** A list of arguments.
- **stackSize** – **[in]** The size of the stack for the engine.

Returns
True if at least one block has been executed.

bool QBDI::*VM*::**switchStackAndCallA**(*rword* *retval, *rword* function, uint32_t argNum, const *rword* *args, uint32_t stackSize = 0x20000)

Switch the stack and call a function using the DBI (and its current state). This method mustn't be called if the *VM* already runs. The stack pointer in the state must'nt be used after the end of this method.

Parameters

- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **argNum** – **[in]** The number of arguments in the array of arguments.
- **args** – **[in]** An array of arguments.
- **stackSize** – **[in]** The size of the stack for the engine.

Returns
True if at least one block has been executed.

bool QBDI::*VM*::**switchStackAndCallV**(*rword* *retval, *rword* function, uint32_t argNum, va_list ap, uint32_t stackSize = 0x20000)

Switch the stack and call a function using the DBI (and its current state). This method mustn't be called if the *VM* already runs. The stack pointer in the state must'nt be used after the end of this method.

Parameters

- **[retval]** – **[in]** Pointer to the returned value (optional).
- **function** – **[in]** Address of the function start instruction.
- **argNum** – **[in]** The number of arguments in the variadic list.
- **ap** – **[in]** An stdarg va_list object.
- **stackSize** – **[in]** The size of the stack for the engine.

Returns

True if at least one block has been executed.

InstAnalysis

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

Obtain the analysis of the current instruction. 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. This method must only be used in an InstCallback.

Parameters

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

Returns

A *InstAnalysis* structure containing the analysis result.

const *InstAnalysis* *QBDI::*VM*::**getCachedInstAnalysis**(*rword* address, *AnalysisType* type = ANALYSIS_INSTRUCTION | ANALYSIS_DISASSEMBLY) const

Obtain the analysis of a cached instruction. Analysis results are cached in the *VM*. The validity of the returned pointer is only guaranteed until the end of the callback or a call to a noconst method of the *VM* object.

Parameters

- **address** – **[in]** The address of the instruction to analyse.
- **[type]** – **[in]** Properties to retrieve during analysis. This argument is optional, defaulting to *QBDI::ANALYSIS_INSTRUCTION* | *QBDI::ANALYSIS_DISASSEMBLY*

Returns

A *InstAnalysis* structure containing the analysis result. null if the instruction isn't in the cache.

MemoryAccess

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

Obtain the memory accesses made by the last executed instruction. The method should be called in an InstCallback.

Returns
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. The method should be called in a VMCallback with *VMEvent::SEQUENCE_EXIT*.

Returns
List of memory access made by the instruction.

bool QBDI::*VM*::**recordMemoryAccess**(*MemoryAccessType* type)

Add instrumentation rules to log memory access using inline instrumentation and instruction shadows.

Parameters
type – **[in]** Memory mode bitfield to activate the logging for: either *QBDI::MEMORY_READ*, *QBDI::MEMORY_WRITE* or both (*QBDI::MEMORY_READ_WRITE*).

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

Cache management

bool QBDI::*VM*::**precacheBasicBlock**(*rword* pc)

Pre-cache a known basic block This method mustn't be called if the *VM* already runs.

Parameters
pc – **[in]** Start address of a basic block

Returns
True if basic block has been inserted in cache.

void QBDI::*VM*::**clearCache**(*rword* start, *rword* end)

Clear a specific address range from the translation cache.

Parameters

- **start** – **[in]** Start of the address range to clear from the cache.
- **end** – **[in]** End of the address range to clear from the cache.

void QBDI::*VM*::**clearAllCache**()

Clear the entire translation cache.

Register state

type QBDI::**rword**

An integer of the size of a register

- uint32_t for X86
- uint64_t for X86_64

struct QBDI::**GPRState**

General Purpose Register context.

For X86 architecture:

```
typedef struct QBDI_ALIGNED(4) {
  rword eax;
  rword ebx;
  rword ecx;
  rword edx;
  rword esi;
  rword edi;
  rword ebp;
  rword esp;
  rword eip;
  rword eflags;
} GPRState;
```

For X86_64 architecture:

```
typedef struct QBDI_ALIGNED(8) {
  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;
  // Only backup and restore with OPT_ENABLE_FS_GS
  rword fs;
  rword gs;
} GPRState;
```

For ARM architecture:

```
/*! ARM General Purpose Register context.
 */
typedef struct QBDI_ALIGNED(4) {
  rword r0;
  rword r1;
  rword r2;
  rword r3;
  rword r4;
  rword r5;
  rword r6;
  rword r7;
  rword r8;
  rword r9;
  rword r10;
  rword r11;
  rword r12;
  rword sp;
  rword lr;
  rword pc;
  rword cpsr;

  /* Internal CPU state
   * Local monitor state for exclusive load/store instruction
   */
  struct {
    rword addr;
    rword enable; /* 0=>disable,
                   * 1=>enable by ldrexb,
                   * 2=>enable by ldrexh,
                   * 4=>enable by ldrex,
                   * 8=>enable by ldrexd
                   */
  } localMonitor;

} GPRState;
```

For AARCH64 architecture:

```
/*! ARM General Purpose Register context.
 */
typedef struct QBDI_ALIGNED(8) {
  rword x0;
  rword x1;
  rword x2;
  rword x3;
  rword x4;
  rword x5;
  rword x6;
  rword x7;
  rword x8;
  rword x9;
  rword x10;
  rword x11;
```

(continues on next page)

(continued from previous page)

```
  rword x12;
  rword x13;
  rword x14;
  rword x15;
  rword x16;
  rword x17;
  rword x18;
  rword x19;
  rword x20;
  rword x21;
  rword x22;
  rword x23;
  rword x24;
  rword x25;
  rword x26;
  rword x27;
  rword x28;
  rword x29; // FP (x29)
  rword lr;  // LR (x30)

  rword sp;
  rword nzcv;
  rword pc;
  // ? rword daif; ?

  /* Internal CPU state
   * Local monitor state for exclusive load/store instruction
   */
  struct {
    rword addr;
    rword enable; /* 0=>disable, 1=>exclusive state, use a rword to not break
                     align */
  } localMonitor;

} GPRState;
```

struct QBDI::**FPRState**

Floating Point Register context.

For X86 architecture:

```
typedef struct QBDI_ALIGNED(16) {
  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 */
```

(continues on next page)

(continued from previous page)

```
  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;
```

For X86_64 architecture:

```
typedef struct QBDI_ALIGNED(16) {
  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 */
```

(continues on next page)

(continued from previous page)

```
  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;
```

For ARM architecture:

```
/*! ARM Floating Point Register context.
```

(continues on next page)

(continued from previous page)

```
 */
typedef union {
  float QBDI_ALIGNED(8) s[32];
  double QBDI_ALIGNED(8) d[QBDI_NUM_FPR];
  uint8_t QBDI_ALIGNED(8) q[QBDI_NUM_FPR / 2][16];
} FPRStateVReg;

typedef struct QBDI_ALIGNED(8) {
  FPRStateVReg vreg;

  rword fpscr;
} FPRState;
```

For AARCH64 architecture:

```
/*! ARM Floating Point Register context.
 */
typedef struct QBDI_ALIGNED(8) {
  __uint128_t v0;
  __uint128_t v1;
  __uint128_t v2;
  __uint128_t v3;

  __uint128_t v4;
  __uint128_t v5;
  __uint128_t v6;
  __uint128_t v7;

  __uint128_t v8;
  __uint128_t v9;
  __uint128_t v10;
  __uint128_t v11;

  __uint128_t v12;
  __uint128_t v13;
  __uint128_t v14;
  __uint128_t v15;

  __uint128_t v16;
  __uint128_t v17;
  __uint128_t v18;
  __uint128_t v19;

  __uint128_t v20;
  __uint128_t v21;
  __uint128_t v22;
  __uint128_t v23;

  __uint128_t v24;
  __uint128_t v25;
  __uint128_t v26;
  __uint128_t v27;
```

(continues on next page)

(continued from previous page)

```
  __uint128_t v28;
  __uint128_t v29;
  __uint128_t v30;
  __uint128_t v31;

  rword fpcr;
  rword fpsr;
} FPRState;
```

struct **MMSTReg**

Public Members

char **reg**[10]

char **rsrv**[6]

struct **FPControl**

Public Members

uint16_t **invalid**

uint16_t **denorm**

uint16_t **zdiv**

uint16_t **ovrfl**

uint16_t **undfl**

uint16_t **precis**

uint16_t **__pad0__**

uint16_t **pc**

uint16_t **rc**

uint16_t **__pad1__**

uint16_t **__pad2__**

struct **FPStatus**

Public Members

uint16_t **invalid**

uint16_t **denorm**

uint16_t **zdiv**

uint16_t **ovrfl**

uint16_t **undfl**

uint16_t **precis**

uint16_t **stkflt**

uint16_t **errsumm**

uint16_t **c0**

uint16_t **c1**

uint16_t **c2**

uint16_t **tos**

uint16_t **c3**

uint16_t **busy**

QBDI::REG_RETURN

QBDI::REG_BP

QBDI::REG_SP

QBDI::REG_PC

QBDI::NUM_GPR

Callback

using QBDI::**VMInstanceRef** = *VM**

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

Instruction callback function type.

Param vm
[in] *VM* instance of the callback.

Param gprState
[in] A structure containing the state of the General Purpose Registers. Modifying it affects the *VM* execution accordingly.

Param fprState
[in] A structure containing the state of the Floating Point Registers. Modifying it affects the *VM* execution accordingly.

Param data
[in] User defined data which can be defined when registering the callback.

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

typedef std::function<*VMAction*(*VMInstanceRef* vm, *GPRState* *gprState, *FPRState* *fprState)> QBDI::**InstCbLambda**

Instruction callback lambda type.

Param vm
[in] *VM* instance of the callback.

Param gprState
[in] A structure containing the state of the General Purpose Registers. Modifying it affects the *VM* execution accordingly.

Param fprState
[in] A structure containing the state of the Floating Point Registers. Modifying it affects the *VM* execution accordingly.

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

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

VM callback function type.

Param vm
[in] *VM* instance of the callback.

Param vmState
[in] A structure containing the current state of the *VM*.

Param gprState
[in] A structure containing the state of the General Purpose Registers. Modifying it affects the *VM* execution accordingly.

Param fprState
[in] A structure containing the state of the Floating Point Registers. Modifying it affects the *VM* execution accordingly.

Param data
[in] User defined data which can be defined when registering the callback.

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

typedef std::function<*VMAction*(*VMInstanceRef* vm, const *VMState* *vmState, *GPRState* *gprState, *FPRState* *fprState)> `QBDI::VMCbLambda`

VM callback lambda type.

Param vm
[in] *VM* instance of the callback.

Param vmState
[in] A structure containing the current state of the *VM*.

Param gprState
[in] A structure containing the state of the General Purpose Registers. Modifying it affects the *VM* execution accordingly.

Param fprState
[in] A structure containing the state of the Floating Point Registers. Modifying it affects the *VM* execution accordingly.

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

typedef std::vector<*InstrRuleDataCBK*> (*`QBDI::InstrRuleCallback`)(*VMInstanceRef* vm, const *InstAnalysis* *inst, void *data)

Instrumentation rule callback function type.

Param vm
[in] *VM* instance of the callback.

Param inst
[in] AnalysisType of the current instrumented Instruction.

Param data
[in] User defined data which can be defined when registering the callback.

Return
Return cbk to call when this instruction is run.

typedef std::function<std::vector<*InstrRuleDataCBK*>(*VMInstanceRef* vm, const *InstAnalysis* *inst)> `QBDI::InstrRuleCbLambda`

Instrumentation rule callback lambda type.

Param vm
[in] *VM* instance of the callback.

Param inst
[in] AnalysisType of the current instrumented Instruction.

Return
Return cbk to call when this instruction is run.

struct **InstrRuleDataCBK**

Public Members

InstPosition **position**

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

InstCallback **cbk**

Address of the function to call when the instruction is executed.

void ***data**

User defined data which will be forward to cbk

InstCbLambda **lambdaCbk**

Lambda callback. Replace cbk and data if not nullptr

int **priority**

Priority of the callback

enum QBDI::**InstPosition**

Position relative to an instruction.

Values:

enumerator **PREINST**

Positioned before the instruction.

enumerator **POSTINST**

Positioned after the instruction.

enum QBDI::**CallbackPriority**

Priority of callback

A callback with an higher priority will be call before a callback with a lower priority.

ie:

a. CBpre(p = 10)
b. CBpre(p = 0)
c. CBpre(p = -10)
d. instrumented instruction
e. CBpost(p = 10)
f. CBpost(p = 0)
g. CBpost(p = -10)

When the *MemoryAccess* API is used in a callback, the priority of the callback must not be greater than PRIORITY_MEMACCESS_LIMIT

Values:

enumerator **PRIORITY_DEFAULT**

Default priority for callback

enumerator **PRIORITY_MEMACCESS_LIMIT**

Maximum priority if getInstMemoryAccess is used in the callback

enum `QBDI::VMAction`

The callback results.

Values:

enumerator **CONTINUE**

The execution of the basic block continues.

enumerator **SKIP_INST**

Available only with PREINST InstCallback. The instruction and the remained PREINST callbacks are skip. The execution continue with the POSTINST instruction.

We recommand to used this result with a low priority PREINST callback in order to emulate the instruction without skipping the POSTINST callback.

enumerator **SKIP_PATCH**

Available only with InstCallback. The current instruction and the reminding callback (PRE and POST) are skip. The execution continues to the next instruction.

For instruction that change the instruction pointer (jump/call/ret), BREAK_TO_VM must be used insted of SKIP.

SKIP can break the record of *MemoryAccess* for the current instruction.

enumerator **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.

enumerator **STOP**

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

InstAnalysis

enum QBDI::**AnalysisType**

Instruction analysis type

Values:

enumerator **ANALYSIS_INSTRUCTION**

Instruction analysis (address, mnemonic, ...)

enumerator **ANALYSIS_DISASSEMBLY**

Instruction disassembly

enumerator **ANALYSIS_OPERANDS**

Instruction operands analysis

enumerator **ANALYSIS_SYMBOL**

Instruction symbol

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)

CPUMode **cpuMode**

Instruction CPU mode

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 **isMoveImm**

true if this instruction is a move immediate (including conditional moves) instruction.

bool **mayLoad**

true if QBDI detects a load for this instruction

bool **mayStore**

true if QBDI detects a store for this instruction

uint32_t **loadSize**

size of the expected read access, may be 0 with mayLoad if the size isn't determined

uint32_t **storeSize**

size of the expected write access, may be 0 with mayStore if the size isn't determined

ConditionType **condition**

Condition associated with the instruction

char ***disassembly**

Instruction disassembly (warning: NULL if !ANALYSIS_DISASSEMBLY)

RegisterAccessType **flagsAccess**

Flag access type (noaccess, r, w, rw) (warning: REGISTER_UNUSED if !ANALYSIS_OPERANDS)

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 ***symbolName**

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

uint32_t **symbolOffset**

Instruction symbol offset

const char *moduleName

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

uint32_t analysisType

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

enum QBDI::ConditionType

Instruction Condition

Values:

enumerator CONDITION_NONE

The instruction is unconditionnal

enumerator CONDITION_ALWAYS

The instruction is always true

enumerator CONDITION_NEVER

The instruction is always false

enumerator CONDITION_EQUALS

Equals (‘==’)

enumerator CONDITION_NOT_EQUALS

Not Equals (‘!=’)

enumerator CONDITION_ABOVE

Above (‘>’ unsigned)

enumerator CONDITION_BELOW_EQUALS

Below or Equals (‘<=’ unsigned)

enumerator CONDITION_ABOVE_EQUALS

Above or Equals (‘>=’ unsigned)

enumerator CONDITION_BELOW

Below (‘<’ unsigned)

enumerator CONDITION_GREAT

Great (‘>’ signed)

enumerator CONDITION_LESS_EQUALS

Less or Equals (‘<=’ signed)

enumerator CONDITION_GREAT_EQUALS

Great or Equals (‘>=’ signed)

enumerator **CONDITION_LESS**

Less ('<' signed)

enumerator **CONDITION_EVEN**

Even

enumerator **CONDITION_ODD**

Odd

enumerator **CONDITION_OVERFLOW**

Overflow

enumerator **CONDITION_NOT_OVERFLOW**

Not Overflow

enumerator **CONDITION_SIGN**

Sign

enumerator **CONDITION_NOT_SIGN**

Not Sign

struct **OperandAnalysis**

Structure containing analysis results of an operand provided by the *VM*.

Public Members

OperandType **type**

Operand type

OperandFlag **flag**

Operand flag

sword **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)

int16_t **regCtxIdx**

Register index in *VM* state (< 0 if not know)

const char *regName

Register name

RegisterAccessType regAccess

Register access type (r, w, rw)

enum QBDI::OperandType

Operand type

Values:

enumerator OPERAND_INVALID

Invalid operand

enumerator OPERAND_IMM

Immediate operand

enumerator OPERAND_GPR

Register operand

enumerator OPERAND_PRED

Predicate operand

enumerator OPERAND_FPR

Float register operand

enumerator OPERAND_SEG

Segment or unsupported register operand

enum QBDI::OperandFlag

Values:

enumerator OPERANDFLAG_NONE

No flag

enumerator OPERANDFLAG_ADDR

The operand is used to compute an address

enumerator OPERANDFLAG_PCREL

The value of the operand is PC relative

enumerator OPERANDFLAG_UNDEFINED_EFFECT

The operand role isn't fully defined

enumerator OPERANDFLAG_IMPLICIT

The operand is implicit

enum QBDI::**RegisterAccessType**

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

Values:

enumerator **REGISTER_UNUSED**

Unused register

enumerator **REGISTER_READ**

Register read access

enumerator **REGISTER_WRITE**

Register write access

enumerator **REGISTER_READ_WRITE**

Register read/write access

MemoryAccess

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

uint16_t **size**

Size of memory access (in bytes)

MemoryAccessType **type**

Memory access type (READ / WRITE)

MemoryAccessFlags **flags**

Memory access flags

enum QBDI::**MemoryAccessType**

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

Values:

enumerator **MEMORY_READ**

Memory read access

enumerator **MEMORY_WRITE**

Memory write access

enumerator **MEMORY_READ_WRITE**

Memory read/write access

enum QBDI::**MemoryAccessFlags**

Memory access flags

Values:

enumerator **MEMORY_NO_FLAGS**

enumerator **MEMORY_UNKNOWN_SIZE**

The size of the access isn't known.

enumerator **MEMORY_MINIMUM_SIZE**

The given size is a minimum size.

enumerator **MEMORY_UNKNOWN_VALUE**

The value of the access is unknown or hasn't been retrived.

VMEvent

enum QBDI::**VMEvent**

Values:

enumerator **NO_EVENT**

enumerator **SEQUENCE_ENTRY**

Triggered when the execution enters a sequence.

enumerator **SEQUENCE_EXIT**

Triggered when the execution exits from the current sequence.

enumerator **BASIC_BLOCK_ENTRY**

Triggered when the execution enters a basic block.

enumerator **BASIC_BLOCK_EXIT**

Triggered when the execution exits from the current basic block.

enumerator **BASIC_BLOCK_NEW**

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

enumerator **EXEC_TRANSFER_CALL**

Triggered when the ExecBroker executes an execution transfer.

enumerator **EXEC_TRANSFER_RETURN**

Triggered when the ExecBroker returns from an execution transfer.

enumerator **SYSCALL_ENTRY**

Not implemented.

enumerator **SYSCALL_EXIT**

Not implemented.

enumerator **SIGNAL**

Not implemented.

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.

Memory management

Allocation

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

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

Parameters

- **size** – **[in]** Allocation size in bytes.
- **align** – **[in]** Base address alignement in bytes.

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

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()*.

Parameters

- **ctx** – **[in]** GPRState which will be setup to use the new stack.
- **stackSize** – **[in]** Size of the stack to be allocated.
- **stack** – **[out]** The newly allocated stack pointer will be returned in the variable pointed by stack.

Returns
True if stack allocation was successfull.

void QBDI::**alignedFree**(void *ptr)

Free a block of aligned memory allocated with alignedAlloc.

Parameters
ptr – **[in]** Pointer to the allocated memory.

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

void QBDI::**simulateCallV**(*GPRState* *ctx, *rword* returnAddress, uint32_t argNum, va_list ap)

Simulate a call by modifying the stack and registers accordingly (stdarg version).

Parameters

- **ctx** – **[in]** GPRState where the simulated call will be setup. The state needs to point to a valid stack for example setup with *allocateVirtualStack()*.
- **returnAddress** – **[in]** Return address of the call to simulate.
- **argNum** – **[in]** The number of arguments in the va_list object.
- **ap** – **[in]** An stdarg va_list object.

void QBDI::**simulateCallA**(*GPRState* *ctx, *rword* returnAddress, uint32_t argNum, const *rword* *args)

Simulate a call by modifying the stack and registers accordingly (C array version).

Parameters

- **ctx** – **[in]** GPRState where the simulated call will be setup. The state needs to point to a valid stack for example setup with *allocateVirtualStack()*.
- **returnAddress** – **[in]** Return address of the call to simulate.
- **argNum** – **[in]** The number of arguments in the array args.
- **args** – **[in]** An array or arguments.

Exploration

std::vector<std::string> QBDI::**getModuleNames**()

Get a list of all the module names loaded in the process memory.

Returns

A vector of string of module names.

std::vector<*MemoryMap*> QBDI::**getCurrentProcessMaps**(bool full_path = false)

Get a list of all the memory maps (regions) of the current process.

Parameters

full_path – **[in]** Return the full path of the module in name field

Returns

A vector of *MemoryMap* object.

std::vector<*MemoryMap*> QBDI::**getRemoteProcessMaps**(QBDI::*rword* pid, bool full_path = false)

Get a list of all the memory maps (regions) of a process.

Parameters

- **pid** – **[in]** The identifier of the process.
- **full_path** – **[in]** Return the full path of the module in name field

Returns

A vector of *MemoryMap* object.

struct **MemoryMap**

Map of a memory area (region).

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:

enumerator **PF_NONE**

No access

enumerator **PF_READ**

Read access

enumerator **PF_WRITE**

Write access

enumerator **PF_EXEC**

Execution access

Other globals

enum QBDI::**Options**

Note: some value are available only for some architecture

Values for all architecture :

enumerator **NO_OPT**

Default Value

enumerator **OPT_DISABLE_FPR**

Disable all operation on FPU (SSE, AVX, SIMD). May break the execution if the target use the FPU

enumerator **OPT_DISABLE_OPTIONAL_FPR**

Disable context switch optimisation when the target execblock doesn't used FPR

Values for AARCH64 and ARM only :

enumerator **OPT_DISABLE_LOCAL_MONITOR**

Disable the local monitor for instruction like strex

Values for AARCH64 only :

enumerator **OPT_BYPASS_PAUTH**

Disable pointeur authentication

enumerator **OPT_ENABLE_BTI**

Enable BTI on instrumented code

Values for ARM only :

enumerator **OPT_DISABLE_D16_D31**

Disable the used of D16-D31 register

enumerator **OPT_ARMv4**

Change between ARM and Thumb as an ARMv4 CPU

enumerator **OPT_ARMv5T_6**

Change between ARM and Thumb as an ARMv5T or ARMv6 CPU

enumerator **OPT_ARMv7**

Change between ARM and Thumb as an ARMv7 CPU (default)

enumerator **OPT_ARM_MASK**

When apply *OPT_ARMv4*, *OPT_ARMv5T_6* or *OPT_ARMv7*, this mask must be clear.

Values for X86 and X86_64 only :

enumerator **OPT_ATT_SYNTAX**

Used the AT&T syntax for instruction disassembly

Values for X86_64 only :

enumerator **OPT_ENABLE_FS_GS**

Enable Backup/Restore of FS/GS segment. This option uses the instructions (RD|WR)(FS|GS)BASE that must be supported by the operating system

enum QBDI::**VMError**

QBDI Error values

Values:

enumerator **INVALID_EVENTID**

Mark a returned event id as invalid

Miscellaneous

Version

inline const char *QBDI::**getVersion**(uint32_t *version)

Return QBDI version.

Parameters

version – **[out]** QBDI version encoded as an unsigned integer (0xMMmmpp).

Returns

QBDI version as a string (major.minor.patch).

Log

enum QBDI::**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:

enumerator **DEBUG**

Debug logs

enumerator **INFO**

Info logs (default)

enumerator **WARNING**

Warning logs

enumerator **ERROR**

Error logs

enumerator **DISABLE**

Disable logs message

void QBDI::**setLogFile**(const std::string &filename, bool truncate = false)

Redirect logs to a file.

Parameters

- **filename** – **[in]** the path of the file to append the log
- **truncate** – **[in]** Set to true to clear the file before append the log

inline void QBDI::**setLogPriority**(*LogPriority* priority = *LogPriority*::*INFO*)

Enable logs matching priority.

Parameters

priority – **[in]** Filter logs with greater or equal priority.

inline void QBDI::**setLogConsole**()

Write log to the console (stderr)

inline void QBDI::**setLogDefault**()

Write log to the default location (stderr for linux, android_logger for android)

Range

template<typename **T**>

class **Range**

Public Functions

inline *T* **start**() const

inline *T* **end**() const

inline void **setStart**(const *T* start)

inline void **setEnd**(const *T* end)

inline **Range**(const *T* start, const *T* end)

Construct a new range.

Parameters

- **start** – **[in]** *Range* start value.
- **end** – **[in]** *Range* end value (excluded).

inline *T* **size**() const

Return the total length of a range.

inline bool **operator==**(const *Range* &r) const

Return True if two ranges are equal (same boundaries).

Parameters
r – **[in]** *Range* to check.

Returns
True if equal.

inline bool **contains**(const *T* t) const

Return True if an value is inside current range boundaries.

Parameters
t – **[in]** Value to check.

Returns
True if contained.

inline bool **contains**(const *Range*<*T*> &r) const

Return True if a range is inside current range boundaries.

Parameters
r – **[in]** *Range* to check.

Returns
True if contained.

inline bool **overlaps**(const *Range*<*T*> &r) const

Return True if a range is overlapping current range lower or/and upper boundary.

Parameters
r – **[in]** *Range* to check.

Returns
True if overlapping.

inline void **display**(std::ostream &os) const

Pretty print a range

Parameters
os – **[in]** An output stream.

inline *Range*<*T*> **intersect**(const *Range*<*T*> &r) const

Return the intersection of two ranges.

Parameters
r – **[in]** *Range* to intersect with current range.

Returns
A new range.

template<typename **T**>

class **RangeSet**

Public Functions

inline **RangeSet**()

inline const std::vector<*Range*<*T*>> &**getRanges**() const

inline *T* **size**() const

inline const *Range*<*T*> ***getElementRange**(const *T* &t) const

inline bool **contains**(const *T* &t) const

inline bool **contains**(const *Range*<*T*> &t) const

inline bool **overlaps**(const *Range*<*T*> &t) const

inline void **add**(const *Range*<*T*> &t)

inline void **add**(const *RangeSet*<*T*> &t)

inline void **remove**(const *Range*<*T*> &t)

inline void **remove**(const *RangeSet*<*T*> &t)

inline void **intersect**(const *RangeSet*<*T*> &t)

inline void **intersect**(const *Range*<*T*> &t)

inline void **clear**()

inline void **display**(std::ostream &os) const

inline bool **operator==**(const *RangeSet* &r) const

2.5.4 PyQBDI API

Introduction

We offer bindings for Python, the whole C/C++ API is available through them, but they are also backed up by plenty of helpers that fluidify the script writing.

On Linux and macOS, PyQBDI supports QBDIPreload as *PyQBDIPreload*.

However, you must be aware that PyQBDI has some limitations:

- The library can handle only one *VM* at a time.
- The *VM* must not be used in the `atexit` module.
- 32-bit versions of PyQBDI and Python are needed to instrument 32-bit targets.
- PyQBDI cannot be used to instrument a Python process since both the **host** and the **target** will use the Python runtime.

VM class

class pyqbdi.**VM**

__init__(*self: pyqbdi.VM*, *cpu: str = ''*, *mattrs: List[str] = []*, *options: pyqbdi.Options = <Options.NO_OPT: 0>*) → None

Construct a new VM for a given CPU with specific attributes

property options

Options of the VM

State management

pyqbdi.VM.**getGPRState**(*self:* pyqbdi.VM) → *pyqbdi.GPRState*

Obtain the current general purpose register state.

pyqbdi.VM.**getFPRState**(*self:* pyqbdi.VM) → *pyqbdi.FPRState*

Obtain the current floating point register state.

pyqbdi.VM.**setGPRState**(*self:* pyqbdi.VM, *gprState:* pyqbdi.GPRState) → None

Set the GPR state.

pyqbdi.VM.**setFPRState**(*self:* pyqbdi.VM, *fprState:* pyqbdi.FPRState) → None

Set the FPR state.

Instrumentation range

Addition

pyqbdi.VM.**addInstrumentedRange**(*self:* pyqbdi.VM, *start: int*, *end: int*) → None

Add an address range to the set of instrumented address ranges.

pyqbdi.VM.**addInstrumentedModule**(*self:* pyqbdi.VM, *name: str*) → bool

Add the executable address ranges of a module to the set of instrumented address ranges.

pyqbdi.VM.**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.

pyqbdi.VM.**instrumentAllExecutableMaps**(*self:* pyqbdi.VM) → bool

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

Removal

pyqbdi.VM.**removeInstrumentedRange**(*self:* pyqbdi.VM, *start: int*, *end: int*) → None

Remove an address range from the set of instrumented address ranges.

pyqbdi.VM.**removeInstrumentedModule**(*self:* pyqbdi.VM, *name: str*) → bool

Remove the executable address ranges of a module from the set of instrumented address ranges.

pyqbdi.VM.**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.

pyqbdi.VM.**removeAllInstrumentedRanges**(*self:* pyqbdi.VM) → None

Remove all instrumented ranges.

Callback management

InstCallback

pyqbdi.VM.**addCodeCB**(*self: pyqbdi.VM, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object, priority: int = <CallbackPriority.PRIORITY_DEFAULT: 0>*) → object

Register a callback event for every instruction executed.

pyqbdi.VM.**addCodeAddrCB**(*self: pyqbdi.VM, address: int, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object, priority: int = <CallbackPriority.PRIORITY_DEFAULT: 0>*) → object

Register a callback for when a specific address is executed.

pyqbdi.VM.**addCodeRangeCB**(*self: pyqbdi.VM, start: int, end: int, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object, priority: int = <CallbackPriority.PRIORITY_DEFAULT: 0>*) → object

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

pyqbdi.VM.**addMnemonicCB**(*self: pyqbdi.VM, mnemonic: str, pos: pyqbdi.InstPosition, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object, priority: int = <CallbackPriority.PRIORITY_DEFAULT: 0>*) → object

Register a callback event if the instruction matches the mnemonic.

VMEvent

pyqbdi.VM.**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.

MemoryAccess

pyqbdi.VM.**addMemAccessCB**(*self: pyqbdi.VM, type: pyqbdi.MemoryAccessType, cbk: Callable[[pyqbdi.VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object, priority: int = <CallbackPriority.PRIORITY_DEFAULT: 0>*) → object

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

pyqbdi.VM.**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.

pyqbdi.VM.**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.

InstrRuleCallback

pyqbdi.VM.**addInstrRule**(*self:* pyqbdi.VM, *cbk: Callable[[*pyqbdi.VM*,* pyqbdi.InstAnalysis*, object], List[*pyqbdi.InstrRuleDataCBK*]]*, *type: pyqbdi.AnalysisType*, *data: object*) → object

Add a custom instrumentation rule to the VM.

pyqbdi.VM.**addInstrRuleRange**(*self:* pyqbdi.VM, *start: int*, *end: int*, *cbk: Callable[[*pyqbdi.VM*,* pyqbdi.InstAnalysis*, object], List[*pyqbdi.InstrRuleDataCBK*]]*, *type: pyqbdi.AnalysisType*, *data: object*) → object

Add a custom instrumentation rule to the VM on a specify range.

Removal

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

Remove an instrumentation.

pyqbdi.VM.**deleteAllInstrumentations**(*self:* pyqbdi.VM) → None

Remove all the registered instrumentations.

Run

pyqbdi.VM.**run**(*self:* pyqbdi.VM, *start: int*, *stop: int*) → bool

Start the execution by the DBI.

pyqbdi.VM.**call**(*self:* pyqbdi.VM, *function: int*, *args: List[int]*) → Tuple[bool, int]

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

InstAnalysis

pyqbdi.VM.**getInstAnalysis**(*self:* pyqbdi.VM, *type: pyqbdi.AnalysisType* = *AnalysisType.ANALYSIS_INSTRUCTION* | *AnalysisType.ANALYSIS_DISASSEMBLY*) → *pyqbdi.InstAnalysis*

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

pyqbdi.VM.**getCachedInstAnalysis**(*self:* pyqbdi.VM, *address: int*, *type: pyqbdi.AnalysisType* = *AnalysisType.ANALYSIS_INSTRUCTION* | *AnalysisType.ANALYSIS_DISASSEMBLY*) → *pyqbdi.InstAnalysis*

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

MemoryAccess

pyqbdi.VM.**getInstMemoryAccess**(*self:* pyqbdi.VM) → List[*pyqbdi.MemoryAccess*]

Obtain the memory accesses made by the last executed instruction.

pyqbdi.VM.**getBBMemoryAccess**(*self:* pyqbdi.VM) → List[*pyqbdi.MemoryAccess*]

Obtain the memory accesses made by the last executed sequence.

pyqbdi.VM.**recordMemoryAccess**(*self:* pyqbdi.VM, *type: pyqbdi.MemoryAccessType*) → bool

Add instrumentation rules to log memory access using inline instrumentation and instruction shadows.

Cache management

pyqbdi.VM.**precacheBasicBlock**(*self:* pyqbdi.VM, *pc: int*) → bool

Pre-cache a known basic block

pyqbdi.VM.**clearCache**(*self:* pyqbdi.VM, *start: int*, *end: int*) → None

Clear a specific address range from the translation cache.

pyqbdi.VM.**clearAllCache**(*self:* pyqbdi.VM) → None

Clear the entire translation cache.

Register state

This documentation only shows the register for the X86_64 architectures. For ARM and AARCH64, the GPRState and FPRState are similar to the C/C++ *QBDI::GPRState* and *QBDI::FPRState*.

class pyqbdi.**GPRState**

property AVAILABLE_GPR

shadow of rbp

property NUM_GPR

shadow of eflags

property REG_BP

shadow of rbp

property REG_FLAG

shadow of eflags

property REG_LR

not available on X86_64

property REG_PC

shadow of rip

property REG_RETURN

shadow of rax

property 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

property eflags

property fs

property gs

property r10

property r11

property r12

property r13

property r14

property r15

property r8

property r9

property rax

property rbp

property rbx

property rcx

property rdi

property rdx

property rip

property rsi

property rsp

class pyqbdi.FPRState

property cs
x87 FPU Instruction Pointer Selector

property dp
x87 FPU Instruction Operand(Data) Pointer offset

property ds
x87 FPU Instruction Operand(Data) Pointer Selector

property fcw
x87 FPU control word

property fop
x87 FPU Opcode

property fsw
x87 FPU status word

property ftw
x87 FPU tag word

property ip
x87 FPU Instruction Pointer offset

property mxcsr
MXCSR Register state

property mxcsrmask
MXCSR mask

property rfcw
x87 FPU control word

property rfsw
x87 FPU status word

property stmm0
ST0/MM0

property stmm1
ST1/MM1

property stmm2
ST2/MM2

property stmm3
ST3/MM3

property stmm4
ST4/MM4

property stmm5
ST5/MM5

property stmm6
ST6/MM6

property stmm7
ST7/MM7

property xmm0
XMM 0

property xmm1
XMM 1

property xmm10
XMM 10

property xmm11
XMM 11

property xmm12
XMM 12

property xmm13
XMM 13

property xmm14
XMM 14

property xmm15
XMM 15

property xmm2
XMM 2

property xmm3
XMM 3

property xmm4
XMM 4

property xmm5
XMM 5

property xmm6
XMM 6

property xmm7
XMM 7

property xmm8
XMM 8

property xmm9
XMM 9

property ymm0
YMM0[255:128]

property ymm1
YMM1[255:128]

property ymm10
YMM10[255:128]

property ymm11
YMM11[255:128]

property ymm12
YMM12[255:128]

property ymm13
YMM13[255:128]

property ymm14
YMM14[255:128]

property ymm15
YMM15[255:128]

property ymm2
YMM2[255:128]

property ymm3
YMM3[255:128]

property ymm4
YMM4[255:128]

property ymm5
YMM5[255:128]

property ymm6
YMM6[255:128]

property ymm7
YMM7[255:128]

property ymm8
YMM8[255:128]

property ymm9
YMM9[255:128]

pyqbdi.**REG_RETURN**

pyqbdi.**REG_BP**

pyqbdi.**REG_SP**

pyqbdi.**REG_PC**

pyqbdi.**NUM_GPR**

Callback

pyqbdi.**InstCallback**(*vm:* pyqbdi.VM, *gpr:* pyqbdi.GPRState, *fpr:* pyqbdi.FPRState, *data: object*) → pyqbdi.VMAction

This is the prototype of a function callback for:

- *pyqbdi.VM.addCodeCB()*, *pyqbdi.VM.addCodeAddrCB()* and *pyqbdi.VM.addCodeRangeCB()*
- *pyqbdi.VM.addMnemonicCB()*
- *pyqbdi.VM.addMemAccessCB()*, *pyqbdi.VM.addMemAddrCB()* and *pyqbdi.VM.addMemRangeCB()*
- *pyqbdi.InstrRuleDataCBK*.

Parameters

- **vm** (VM) – The current QBDI object
- **gpr** (GPRState) – The current GPRState
- **fpr** (FPRState) – The current FPRState
- **data** (*Object*) – A user-defined object.

Returns
the *VMAction* to continue or stop the execution

pyqbdi.**VMCallback**(*vm:* pyqbdi.VM, *vmState:* pyqbdi.VMState, *gpr:* pyqbdi.GPRState, *fpr:* pyqbdi.FPRState, *data: object*) → pyqbdi.VMAction

This is the prototype of a function callback for *pyqbdi.VM.addVMEventCB()*.

Parameters

- **vm** (VM) – The current QBDI object
- **vmState** (VMState) – A structure containing the current state of the VM.
- **gpr** (GPRState) – The current GPRState
- **fpr** (FPRState) – The current FPRState
- **data** (*Object*) – A user-defined object

Returns

the *VMAction* to continue or stop the execution

pyqbdi.**InstrRuleCallback**(*vm:* pyqbdi.VM, *ana:* pyqbdi.InstAnalysis, *data: object*) → List[*pyqbdi.InstrRuleDataCBK*]

This is the prototype of a function callback for *pyqbdi.VM.addInstrRule()* and *pyqbdi.VM.addInstrRuleRange()*.

Parameters

- **vm** (VM) – The current QBDI object
- **ana** (InstAnalysis) – The current QBDI object
- **data** (*Object*) – A user-defined object

Returns

A list of *pyqbdi.InstrRuleDataCBK*

class pyqbdi.**InstrRuleDataCBK**

__init__(*self: pyqbdi.InstrRuleDataCBK, cbk: Callable[[QBDI::VM, pyqbdi.GPRState, pyqbdi.FPRState, object], pyqbdi.VMAction], data: object, position: pyqbdi.InstPosition, priority: int = <CallbackPriority.PRIORITY_DEFAULT: 0>*) → None

property cbk

Address of the function to call when the instruction is executed

property data

User defined data which will be forward to cbk.

property position

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

property priority

Priority of the callback.

pyqbdi.**InstPosition = <class 'pyqbdi.InstPosition'>**

Position relative to an instruction.

Members:

PREINST : Positioned before the instruction.

POSTINST : Positioned after the instruction.

pyqbdi.**CallbackPriority = <class 'pyqbdi.CallbackPriority'>**

Priority of callback.

Members:

PRIORITY_DEFAULT : Default priority for callback.

PRIORITY_MEMACCESS_LIMIT : Maximum priority if getInstMemoryAccess is used in the callback.

pyqbdi.**VMAction = <class 'pyqbdi.VMAction'>**

The callback results.

Members:

CONTINUE : The execution of the basic block continues.

SKIP_INST : Available only with PREINST InstCallback. The instruction and the remained PREINST callbacks are skip. The execution continue with the POSTINST instruction.

We recommand to used this result with a low priority PREINST callback in order to emulate the instruction without skipping the POSTINST callback.

SKIP_PATCH : Available only with InstCallback. The current instruction and the reminding callback (PRE and POST) are skip. The execution continues to the next instruction.

For instruction that change the instruction pointer (jump/call/ret), BREAK_TO_VM must be used insted of SKIP.

SKIP can break the record of MemoryAccess for the current instruction.

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.

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**

property address

Instruction address (if ANALYSIS_INSTRUCTION)

property affectControlFlow

True if instruction affects control flow (if ANALYSIS_INSTRUCTION)

property condition

Condition associated with the instruction (if ANALYSIS_INSTRUCTION)

property disassembly

Instruction disassembly (if ANALYSIS_DISASSEMBLY)

property flagsAccess

Flag access type (noaccess, r, w, rw) (if ANALYSIS_OPERANDS)

property instSize

Instruction size (in bytes) (if ANALYSIS_INSTRUCTION)

property isBranch

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

property isCall

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

property isCompare

True if instruction is a comparison (if ANALYSIS_INSTRUCTION)

property isMoveImm

True if this instruction is a move immediate (including conditional moves) instruction. (if ANALYSIS_INSTRUCTION)

property isPredicable

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

property isReturn

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

property loadSize

size of the expected read access (if ANALYSIS_INSTRUCTION)

property mayLoad

True if QBDI detects a load for this instruction (if ANALYSIS_INSTRUCTION)

property mayStore

True if QBDI detects a store for this instruction (if ANALYSIS_INSTRUCTION)

property mnemonic

LLVM mnemonic (if ANALYSIS_INSTRUCTION)

property module

Instruction module name (if ANALYSIS_SYMBOL and found) (deprecated)

property moduleName

Instruction module name (if ANALYSIS_SYMBOL and found)

property numOperands

Number of operands used by the instruction (if ANALYSIS_OPERANDS)

property operands

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

property storeSize

size of the expected write access (if ANALYSIS_INSTRUCTION)

property symbol

Instruction symbol (if ANALYSIS_SYMBOL and found) (deprecated)

property symbolName

Instruction symbol (if ANALYSIS_SYMBOL and found)

property symbolOffset

Instruction symbol offset (if ANALYSIS_SYMBOL)

pyqbdi.**ConditionType = <class 'pyqbdi.ConditionType'>**

Condition type

Members:

CONDITION_NONE : The instruction is unconditionnal

CONDITION_ALWAYS : The instruction is always true

CONDITION_NEVER : The instruction is always false

CONDITION_EQUALS : Equals ('==')

CONDITION_NOT_EQUALS : Not Equals ('!=')

CONDITION_ABOVE : Above ('>' unsigned)

CONDITION_BELOW_EQUALS : Below or Equals ('<=' unsigned)

CONDITION_ABOVE_EQUALS : Above or Equals ('>=' unsigned)

CONDITION_BELOW : Below ('<' unsigned)

CONDITION_GREAT : Great ('>' signed)

CONDITION_LESS_EQUALS : Less or Equals ('<=' signed)

CONDITION_GREAT_EQUALS : Great or Equals ('>=' signed)

CONDITION_LESS : Less ('<' signed)

CONDITION_EVEN : Even

CONDITION_ODD : Odd

CONDITION_OVERFLOW : Overflow

CONDITION_NOT_OVERFLOW : Not Overflow

CONDITION_SIGN : Sign

CONDITION_NOT_SIGN : Not Sign

class pyqbdi.**OperandAnalysis**

property flag

Operand flag

property regAccess

Register access type (r, w, rw)

property regCtxIdx

Register index in VM state

property regName

Register name

property regOff

Sub-register offset in register (in bits)

property size

Operand size (in bytes)

property type

Operand type

property value

Operand value (if immediate), or register Id

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

Operand type

Members:

OPERAND_INVALID : Invalid operand

OPERAND_IMM : Immediate operand

OPERAND_GPR : Register operand

OPERAND_PRED : Predicate operand

OPERAND_FPR : Float register operand

OPERAND_SEG : Segment or unsupported register 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

OPERANDFLAG_IMPLICIT : The operand is implicit

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

MemoryAccess

class pyqbdi.**MemoryAccess**

property accessAddress

Address of accessed memory

property flags

Memory access flags

property instAddress

Address of instruction making the access

property size

Size of memory access (in bytes)

property type

Memory access type (READ / WRITE)

property 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.**MemoryAccessFlags = <class 'pyqbdi.MemoryAccessFlags'>**

Memory access flags

Members:

MEMORY_NO_FLAGS : Empty flags

MEMORY_UNKNOWN_SIZE : The size of the access isn't known.

MEMORY_MINIMUM_SIZE : The given size is a minimum size.

MEMORY_UNKNOWN_VALUE : The value of the access is unknown or hasn't been retrived.

VMEvent

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**

property basicBlockEnd

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

property basicBlockStart

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

property event

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

property sequenceEnd

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

property sequenceStart

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

Memory management

Allocation

pyqbdi.**alignedAlloc**(*size: int*, *align: int*) → int

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

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.**alignedFree**(*ptr: int*) → None

Free a block of aligned memory allocated with alignedAlloc.

pyqbdi.**simulateCall**(*ctx:* pyqbdi.GPRState, *returnAddress: int*, *args: List[int] = []*) → None

Simulate a call by modifying the stack and registers accordingly.

Exploration

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.

class pyqbdi.**MemoryMap**

property name

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

property permission

Region access rights (PF_READ, PF_WRITE, PF_EXEC).

property range

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

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

Other globals

This documentation only shows the options available for the X86_64 architectures. For ARM and AARCH64, refer the C/C++ API *QBDI::Options*.

pyqbdi.**Options = <class 'pyqbdi.Options'>**

VM options

Members:

NO_OPT : Default value

OPT_DISABLE_FPR : Disable all operation on FPU (SSE, AVX, SIMD). May break the execution if the target use the FPU.

OPT_DISABLE_OPTIONAL_FPR : Disable context switch optimisation when the target execblock doesn't used FPR

OPT_ATT_SYNTAX : Used the AT&T syntax for instruction disassembly

OPT_ENABLE_FS_GS : Enable Backup/Restore of FS/GS segment. This option uses the instructions (RD|WR)(FS|GS)BASE that must be supported by the operating system.

pyqbdi.**VMError = <class 'pyqbdi.VMError'>**

QBDI Error values

Members:

INVALID_EVENTID : Mark a returned event id as invalid

Miscellaneous

Version & info

pyqbdi.**__arch__**

pyqbdi.**__platform__**

pyqbdi.**__preload__**

Library load with pyqbdipreload

pyqbdi.**__version__**

Version of QBDI

Log

pyqbdi.**LogPriority = <class 'pyqbdi.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.

Members:

DEBUG : Debug logs

INFO : Info logs (default)

WARNING : Warning logs

ERROR : Error logs

DISABLE : Disable logs message

pyqbdi.**setLogPriority**(*priority: pyqbdi.LogPriority*) → None

Enable logs matching priority.

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.

property 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.

property start

Range start value.

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 utilities, check out the Python's struct library.

2.5.5 Frida/QBDI API

Introduction

We provide bindings for Frida, most C/C++ APIs are exported and available through them, but they are also backed up by plenty of helpers that fluidify the script writing.

Nevertheless, this API slightly differs from the C++ API:

- Every callback must be created as a native function. The callback registration can be done through calling *VM.newInstCallback()*, *VM.newVMCallback()* and *VM.newInstrRuleCallback()*.
- The *QBDI* class is the equivalent of the *VM* class we have in the C++ API.

QBDI class

With Frida API, the QBDI object is the equivalent of the VM object in C++ API.

class VM()

Create a new instrumentation virtual machine using "**new VM()**"

Options

`VM.getOptions()`

Get the current options of the VM

Returns
Options – The current option

`VM.setOptions(options)`

Set the options of the VM

Arguments

- **options** (Options) – The new options of the VM.

State management

`VM.getGPRState()`

Obtain the current general register state.

Returns
GPRState – An object containing the General Purpose Registers state.

`VM.getFPRState()`

Obtain the current floating point register state.

Returns
FPRState – An object containing the Floating point Purpose Registers state.

`VM.setGPRState(state)`

Set the GPR state

Arguments

- **state** (GPRState) – Array of register values

`VM.setFPRState(state)`

Set the FPR state

Arguments

- **state** (*FPRState*) – Array of register values

Instrumentation range

Addition

`VM.addInstrumentedRange(start, end)`

Add an address range to the set of instrumented address ranges.

Arguments

- **start** (*String|Number|NativePointer*) – Start address of the range (included).
- **end** (*String|Number|NativePointer*) – End address of the range (excluded).

VM.**addInstrumentedModule**(*name*)

Add the executable address ranges of a module to the set of instrumented address ranges.

Arguments

- **name** (*String*) – The module's name.

Returns

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

VM.**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** (*String|Number|NativePointer*) – An address contained by module's range.

Returns

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

VM.**instrumentAllExecutableMaps**()

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

Returns

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

Removal

VM.**removeInstrumentedRange**(*start*, *end*)

Remove an address range from the set of instrumented address ranges.

Arguments

- **start** (*String|Number|NativePointer*) – Start address of the range (included).
- **end** (*String|Number|NativePointer*) – End address of the range (excluded).

VM.**removeInstrumentedModule**(*name*)

Remove the executable address ranges of a module from the set of instrumented address ranges.

Arguments

- **name** (*String*) – The module's name.

Returns

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

VM.**removeInstrumentedModuleFromAddr**(*addr:*)

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

Arguments

- **addr:** (*String|Number|NativePointer*) – An address contained by module's range.

Returns

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

VM.**removeAllInstrumentedRanges**()

Remove all instrumented ranges.

Callback management

Creation

VM.**newInstCallback**(*cbk*)

Create a native **Instruction callback** from a JS function.

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;
>>> });
```

Arguments

- **cbk** (*InstCallbackRaw*) – an instruction callback (ex: function(vm, gpr, fpr, data) {};)

Returns

an native InstCallback

VM.**newInstrRuleCallback**(*cbk*)

Create a native **Instruction rule callback** from a JS function.

Example:

```
>>> var icbk = vm.newInstrRuleCallback(function(vm, ana, data) {
>>>   console.log("0x" + ana.address.toString(16) + " " + ana.disassembly);
>>>   return [new InstrRuleDataCBK(InstPosition.POSTINST, printCB, ana.
↪disassembly)];
>>> });
```

Arguments

- **cbk** (*InstrRuleCallbackRaw*) – an instruction callback (ex: function(vm, ana, data) {};)

Returns

an native InstrRuleCallback

VM.**newVMCallback**(*cbk*)

Create a native **VM callback** from a JS function.

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;
>>> });
```

Arguments

- **cbk** (*VMCallbackRaw*) – a VM callback (ex: function(vm, state, gpr, fpr, data) {};)

Returns
a native VMCallback

InstCallback

VM.**addCodeCB**(*pos*, *cbk*, *data*, *priority*)

Register a callback event for a specific instruction event.

Arguments

- **pos** (InstPosition) – Relative position of the callback (PreInst / PostInst).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.
- **priority** (*Int*) – The priority of the callback.

Returns
Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

VM.**addCodeAddrCB**(*addr*, *pos*, *cbk*, *data*, *priority*)

Register a callback for when a specific address is executed.

Arguments

- **addr** (*String|Number|NativePointer*) – Code address which will trigger the callback.
- **pos** (InstPosition) – Relative position of the callback (PreInst / PostInst).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.
- **priority** (*Int*) – The priority of the callback.

Returns
Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

VM.**addCodeRangeCB**(*start*, *end*, *pos*, *cbk*, *data*, *priority*)

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

Arguments

- **start** (*String|Number|NativePointer*) – Start of the address range which will trigger the callback.
- **end** (*String|Number|NativePointer*) – End of the address range which will trigger the callback.
- **pos** (InstPosition) – Relative position of the callback (PreInst / PostInst).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.
- **priority** (*Int*) – The priority of the callback.

Returns
Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

VM.**addMnemonicCB**(*mnem*, *pos*, *cbk*, *data*, *priority*)

Register a callback event if the instruction matches the mnemonic.

Arguments

- **mnem** (*String*) – Mnemonic to match.
- **pos** (InstPosition) – Relative position of the callback (PreInst / PostInst).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.
- **priority** (*Int*) – The priority of the callback.

Returns

Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

VMEvent

VM.**addVMEventCB**(*mask*, *cbk*, *data*)

Register a callback event for a specific VM event.

Arguments

- **mask** (VMEvent) – A mask of VM event type which will trigger the callback.
- **cbk** (VMCallback) – A **native** VMCallback returned by *VM.newVMCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.

Returns

Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

MemoryAccess

VM.**addMemAccessCB**(*type*, *cbk*, *data*, *priority*)

Register a callback event for every memory access matching the type bitfield made by the instruction in the range codeStart to codeEnd.

Arguments

- **type** (MemoryAccessType) – A mode bitfield: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.
- **priority** (*Int*) – The priority of the callback.

Returns

Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

VM.**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** (*String|Number|NativePointer*) – Code address which will trigger the callback.
- **type** (MemoryAccessType) – A mode bitfield: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.

Returns

Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

VM.**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** (*String|Number|NativePointer*) – Start of the address range which will trigger the callback.
- **end** (*String|Number|NativePointer*) – End of the address range which will trigger the callback.
- **type** (MemoryAccessType) – A mode bitfield: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object|null*) – User defined data passed to the callback.

Returns

Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

InstrRuleCallback

VM.**addInstrRule**(*cbk*, *type*, *data*)

Add a custom instrumentation rule to the VM.

Arguments

- **cbk** (InstrRuleCallback) – A **native** InstrRuleCallback returned by *VM.newInstrRuleCallback()*.
- **type** (AnalysisType) – Analyse type needed for this instruction function pointer to the callback
- **data** (*Object|null*) – User defined data passed to the callback.

Returns

Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

VM.**addInstrRuleRange**(*start*, *end*, *cbk*, *type*, *data*)

Add a custom instrumentation rule to the VM for a range of address.

Arguments

- **start** (*String|Number|NativePointer*) – Begin of the range of address where apply the rule
- **end** (*String|Number|NativePointer*) – End of the range of address where apply the rule
- **cbk** (InstrRuleCallback) – A **native** InstrRuleCallback returned by *VM.newInstrRuleCallback()*.
- **type** (AnalysisType) – Analyse type needed for this instruction function pointer to the callback
- **data** (*Object|null*) – User defined data passed to the callback.

Returns
Number – The id of the registered instrumentation (or VMError.INVALID_EVENTID in case of failure).

Removal

VM.**deleteInstrumentation**(*id*)

Remove an instrumentation.

Arguments

- **id** (*Number*) – The id of the instrumentation to remove.

Returns
bool – True if instrumentation has been removed.

VM.**deleteAllInstrumentations**()

Remove all the registered instrumentations.

Memory management

Allocation

VM.**alignedAlloc**(*size*, *align*)

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

Arguments

- **size** (*Number*) – Allocation size in bytes.
- **align** (*Number*) – Base address alignement in bytes.

Returns
Pointer (rword) to the allocated memory or NULL in case an error was encountered.

VM.**allocateVirtualStack**(*state*, *stackSize*)

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

Arguments

- **state** (GPRState) – Array of register values
- **stackSize** (*Number*) – Size of the stack to be allocated.

Returns
Pointer (rword) to the allocated memory or NULL in case an error was encountered.

VM.**alignedFree**(*ptr*)

Free a block of aligned memory allocated with alignedAlloc or allocateVirtualStack

Arguments

- **ptr** (*NativePtr*) – Pointer to the allocated memory.

Exploration

VM.**getModuleNames**()

Use QBDI engine to retrieve loaded modules.

Returns
list of module names (ex: ["ls", "libc", "libz"])

Run

VM.**run**(*start*, *stop*)

Start the execution by the DBI from a given address (and stop when another is reached).

Arguments

- **start** (*String|Number|NativePointer*) – Address of the first instruction to execute.
- **stop** (*String|Number|NativePointer*) – Stop the execution when this instruction is reached.

Returns
bool – True if at least one block has been executed.

VM.**call**(*address*, *args*)

Call a function by its address (or through a Frida `NativePointer`).

Arguments can be provided, but their types need to be compatible with the `.toRword()` interface (like `NativePointer` or `UInt64`).

Example:

```
>>> var vm = new VM();
>>> var state = vm.getGPRState();
>>> var stackTopPtr = vm.allocateVirtualStack(state, 0x1000000);
>>> var aFunction = Module.findExportByName(null, "Secret");
>>> vm.addInstrumentedModuleFromAddr(aFunction);
>>> vm.call(aFunction, [42]);
>>> vm.alignedFree(stackTopPtr);
```

Arguments

- **address** (*String|Number|NativePointer*) – function address (or Frida `NativePointer`).
- **args** (*StringArray|NumberArray*) – optional list of arguments

VM.**switchStackAndCall**(*address*, *args*, *stackSize*)

Call a function by its address (or through a Frida `NativePointer`). QBDI will allocate his one stack to run, while the instrumented code will use the top of the current stack.

Arguments can be provided, but their types need to be compatible with the `.toRword()` interface (like `NativePointer` or `UInt64`).

Example:

```
>>> var vm = new VM();
>>> var state = vm.getGPRState();
>>> var aFunction = Module.findExportByName(null, "Secret");
>>> vm.addInstrumentedModuleFromAddr(aFunction);
>>> vm.switchStackAndCall(aFunction, [42]);
```

Arguments

- **address** (*String|Number|NativePointer*) – function address (or Frida `NativePointer`).
- **args** (*StringArray|NumberArray*) – optional list of arguments
- **stackSize** (*String|Number*) – stack size for the engine.

VM.**simulateCall**(*state*, *retAddr*, *args*)

Simulate a call by modifying the stack and registers accordingly.

Arguments

- **state** (GPRState) – Array of register values
- **retAddr** (*String|Number|NativePointer*) – Return address of the call to simulate.
- **args** (*StringArray|NumberArray*) – A variadic list of arguments.

InstAnalysis

VM.**getInstAnalysis**(*type*)

Obtain the analysis of the current instruction. 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** (AnalysisType) – Properties to retrieve during analysis (default to ANALYSIS_INSTRUCTION | ANALYSIS_DISASSEMBLY).

Returns

InstAnalysis – A *InstAnalysis()* object containing the analysis result.

VM.**getCachedInstAnalysis**(*addr*, *type*)

Obtain the analysis of a cached instruction. 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

- **addr** (*String|Number|NativePointer*) – The address of the instruction to analyse.
- **type** (AnalysisType) – Properties to retrieve during analysis (default to ANALYSIS_INSTRUCTION | ANALYSIS_DISASSEMBLY).

Returns
InstAnalysis – A *InstAnalysis()* object containing the analysis result. null if the instruction isn't in the cache.

MemoryAccess

`VM.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
Array.<MemoryAccess> – An array of *MemoryAccess()* made by the instruction.

`VM.getBBMemoryAccess()`

Obtain the memory accesses made by the last executed sequence. Return NULL and a size of 0 if the basic block made no memory access.

Returns
Array.<MemoryAccess> – An array of *MemoryAccess()* made by the sequence.

`VM.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** (MemoryAccessType) – Memory mode bitfield to activate the logging for: either MEMORY_READ, MEMORY_WRITE or both (MEMORY_READ_WRITE).

Cache management

`VM.precacheBasicBlock(pc)`

Pre-cache a known basic block.

Arguments

- **pc** (*String|Number|NativePointer*) – Start address of a basic block

Returns
bool – True if basic block has been inserted in cache.

`VM.clearCache(start, end)`

Clear a specific address range from the translation cache.

Arguments

- **start** (*String|Number|NativePointer*) – Start of the address range to clear from the cache.
- **end** (*String|Number|NativePointer*) – End of the address range to clear from the cache.

`VM.clearAllCache()`

Clear the entire translation cache.

Register state

class GPRState()

General Purpose Register context

GPRState.**dump**(*color*)

Pretty print and log QBDI context.

Arguments

- **color** (*bool*) – Will print a colored version of the context if set.

GPRState.**getRegister**(*rid*)

This function is used to get the value of a specific register.

Arguments

- **rid** (*String|Number*) – Register (register name or ID can be used e.g : "RAX", "rax", 0)

Returns
GPR value (ex: 0x42)

GPRState.**getRegisters**()

This function is used to get values of all registers.

Returns
GPRs of current context (ex: {"RAX":0x42, ...})

GPRState.**pp**(*color*)

Pretty print QBDI context.

Arguments

- **color** (*bool*) – Will print a colored version of the context if set.

Returns
dump of all GPRs in a pretty format

GPRState.**setRegister**(*rid*, *value*)

This function is used to set the value of a specific register.

Arguments

- **rid** (*String|Number*) – Register (register name or ID can be used e.g : "RAX", "rax", 0)
- **value** (*String|Number*) – Register value (use **strings** for big integers)

GPRState.**setRegisters**(*gprs*)

This function is used to set values of all registers.

Arguments

- **gprs** – Array of register values

GPRState.**synchronizeContext**(*FridaCtx*, *direction*)

This function is used to synchronise context between Frida and QBDI.

> **Warning:** Currently QBDI_TO_FRIDA is not implemented (due to Frida limitations).

Arguments

- **FridaCtx** – Frida context

- **direction** (SyncDirection) – Synchronization direction. (FRIDA_TO_QBDI or QBDI_TO_FRIDA)

GPRState.**synchronizeRegister**(*FridaCtx*, *rid*, *direction*)

This function is used to synchronise a specific register between Frida and QBDI.

> **Warning:** Currently QBDI_TO_FRIDA is experimental. (E.G : RIP cannot be synchronized)

Arguments

- **FridaCtx** – Frida context
- **rid** (*String|Number*) – Register (register name or ID can be used e.g : "RAX", "rax", 0)
- **direction** (SyncDirection) – Synchronization direction. (FRIDA_TO_QBDI or QBDI_TO_FRIDA)

class SyncDirection()

Synchronisation direction between Frida and QBDI GPR contexts

SyncDirection.**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.

SyncDirection.**FRIDA_TO_QBDI**

Constant variable used to synchronize Frida's context to QBDI's.

GPR_NAMES

An array holding register names.

REG_PC

String of the instruction pointer register.

REG_RETURN

A constant string representing the register carrying the return value of a function.

REG_SP

String of the stack pointer register.

Callback

InstCallback(*vm*, *gpr*, *fpr*, *data*)

This is the prototype of a function callback for:

- *VM.addCodeCB()*, *VM.addCodeAddrCB()* and *VM.addCodeRangeCB()*
- *VM.addMnemonicCB()*
- *VM.addMemAccessCB()*, *VM.addMemAddrCB()* and *VM.addMemRangeCB()*
- *InstRuleDataCBK()*.

The function must be registered with *VM.newInstCallback()*.

Arguments

- **vm** (*QBDI*) – The current QBDI object
- **gpr** (GPRState) – The current GPRState
- **fpr** (*FPRState*) – The current FPRState
- **data** (*Object*) – A user-defined object.

Returns
the *VMAction()* to continue or stop the execution

VMCallback(*vm*, *vmState*, *gpr*, *fpr*, *data*)

This is the prototype of a function callback for *VM.addVMEventCB()*. The function must be registered with *VM.newVMCallback()*.

Arguments

- **vm** (*QBDI*) – The current QBDI object
- **vmState** (VMState) – A structure containing the current state of the VM.
- **gpr** (GPRState) – The current GPRState
- **fpr** (*FPRState*) – The current FPRState
- **data** (*Object*) – A user-defined object

Returns
the *VMAction()* to continue or stop the execution

InstrRuleCallback(*vm*, *ana*, *data*)

This is the prototype of a function callback for *VM.addInstrRule()* and *VM.addInstrRuleRange()*. The function must be registered with *VM.newInstrRuleCallback()*.

Arguments

- **vm** (*QBDI*) – The current QBDI object
- **ana** (InstAnalysis) – The current QBDI object
- **data** (*Object*) – A user-defined object

Returns
An Array of *InstrRuleDataCBK()*

class InstrRuleDataCBK(*pos*, *cbk*, *data*, *priority*)

Object to define an *InstCallback()* in an *InstrRuleCallback()*

Arguments

- **pos** (InstPosition) – Relative position of the callback (PreInst / PostInst).
- **cbk** (InstCallback) – A **native** InstCallback returned by *VM.newInstCallback()*.
- **data** (*Object*|*null*) – User defined data passed to the callback.
- **priority** (*Int*) – The priority of the callback.

class VMAction()

The callback results.

VMAction.**CONTINUE**

The execution of the basic block continues.

VMAction.**SKIP_INST**

Available only with PREINST InstCallback. The instruction and the remained PREINST callbacks are skip. The execution continue with the POSTINST instruction.

We recommand to used this result with a low priority PREINST callback in order to emulate the instruction without skipping the POSTINST callback.

VMAction.**SKIP_PATCH**

Available only with InstCallback. The current instruction and the reminding callback (PRE and POST) are skip. The execution continues to the next instruction.

For instruction that change the instruction pointer (jump/call/ret), BREAK_TO_VM must be used insted of SKIP.

SKIP can break the record of MemoryAccess for the current instruction.

VMAction.**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.

VMAction.**STOP**

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

class InstPosition()

Position relative to an instruction.

InstPosition.**PREINST**

Positioned **before** the instruction.

InstPosition.**POSTINST**

Positioned **after** the instruction.

class CallbackPriority()

Priority of callback

CallbackPriority.**PRIORITY_DEFAULT**

Default priority for callback.

CallbackPriority.**PRIORITY_MEMACCESS_LIMIT**

Maximum priority if getInstMemoryAccess is used in the callback.

InstAnalysis

class AnalysisType()

Properties to retrieve during an instruction analysis.

AnalysisType.**ANALYSIS_INSTRUCTION**

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

AnalysisType.**ANALYSIS_DISASSEMBLY**

Instruction disassembly.

AnalysisType.**ANALYSIS_OPERANDS**

Instruction operands analysis.

AnalysisType.**ANALYSIS_SYMBOL**

Instruction nearest symbol (and offset).

class InstAnalysis()

Object that describes the analysis of an instruction

InstAnalysis.**address**

Instruction address (if ANALYSIS_INSTRUCTION)

InstAnalysis.**affectControlFlow**

True if instruction affects control flow (if ANALYSIS_INSTRUCTION)

InstAnalysis.**disassembly**

Instruction disassembly (if ANALYSIS_DISASSEMBLY)

InstAnalysis.**instSize**

Instruction size (in bytes) (if ANALYSIS_INSTRUCTION)

InstAnalysis.**isBranch**

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

InstAnalysis.**isCall**

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

InstAnalysis.**isCompare**

True if instruction is a comparison (if ANALYSIS_INSTRUCTION)

InstAnalysis.**isPredicable**

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

InstAnalysis.**isMoveImm**

True if this instruction is a move immediate (including conditional moves) instruction (if ANALYSIS_INSTRUCTION)

InstAnalysis.**isReturn**

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

InstAnalysis.**mayLoad**

True if QBDI detects a load for this instruction (if ANALYSIS_INSTRUCTION)

InstAnalysis.**mayStore**

True if QBDI detects a store for this instruction (if ANALYSIS_INSTRUCTION)

InstAnalysis.**loadSize**

size of the expected read access (if ANALYSIS_INSTRUCTION)

InstAnalysis.**storeSize**

size of the expected written access (if ANALYSIS_INSTRUCTION)

InstAnalysis.**condition**

Condition associated with the instruction (if ANALYSIS_INSTRUCTION)

InstAnalysis.**mnemonic**

LLVM mnemonic (if ANALYSIS_INSTRUCTION)

InstAnalysis.**flagsAccess**

Flag access type (noaccess, r, w, rw) (if ANALYSIS_OPERANDS)

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

InstAnalysis.**moduleName**
: Instruction module name (if ANALYSIS_SYMBOL and found)

InstAnalysis.**symbolName**
: Instruction symbol (if ANALYSIS_SYMBOL and found)

InstAnalysis.**symbolOffset**
: Instruction symbol offset (if ANALYSIS_SYMBOL and found)

class ConditionType()
: Instruction Condition

ConditionType.**CONDITION_NONE**
: The instruction is unconditionnal

ConditionType.**CONDITION_ALWAYS**
: The instruction is always true

ConditionType.**CONDITION_NEVER**
: The instruction is always false

ConditionType.**CONDITION_EQUALS**
: Equals ('==')

ConditionType.**CONDITION_NOT_EQUALS**
: Not Equals ('!=')

ConditionType.**CONDITION_ABOVE**
: Above ('>' unsigned)

ConditionType.**CONDITION_BELOW_EQUALS**
: Below or Equals ('<=' unsigned)

ConditionType.**CONDITION_ABOVE_EQUALS**
: Above or Equals ('>=' unsigned)

ConditionType.**CONDITION_BELOW**
: Below ('<' unsigned)

ConditionType.**CONDITION_GREAT**
: Great ('>' signed)

ConditionType.**CONDITION_LESS_EQUALS**
: Less or Equals ('<=' signed)

ConditionType.**CONDITION_GREAT_EQUALS**
: Great or Equals ('>=' signed)

ConditionType.**CONDITION_LESS**
: Less ('<' signed)

ConditionType.**CONDITION_EVEN**
: Even

ConditionType.**CONDITION_ODD**
: Odd

ConditionType.**CONDITION_OVERFLOW**
: Overflow

ConditionType.**CONDITION_NOT_OVERFLOW**
: Not Overflow

ConditionType.**CONDITION_SIGN**
: Sign

ConditionType.**CONDITION_NOT_SIGN**
: Not Sign

class OperandAnalysis()
: Structure containing analysis results of an operand provided by the VM.

OperandAnalysis.**type**
: Operand type

OperandAnalysis.**flag**
: Operand flag

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

OperandAnalysis.**size**
: Operand size (in bytes)

OperandAnalysis.**regOff**
: Sub-register offset in register (in bits)

OperandAnalysis.**regCtxIdx**
: Register index in VM state

OperandAnalysis.**regName**
: Register name

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

class OperandType()
: Register access type (read / write / rw)

OperandType.**OPERAND_INVALID**
: Invalid operand.

OperandType.**OPERAND_IMM**
: Immediate operand.

OperandType.**OPERAND_GPR**
: General purpose register operand.

OperandType.**OPERAND_PRED**
: Predicate special operand.

OperandType.**OPERAND_FPR**
: Float register operand.

OperandType.**OPERAND_SEG**
Segment or unsupported register operand

class OperandFlag()
Operand flag

OperandFlag.**OPERANDFLAG_NONE**
No flag

OperandFlag.**OPERANDFLAG_ADDR**
The operand is used to compute an address

OperandFlag.**OPERANDFLAG_PCREL**
The value of the operand is PC relative

OperandFlag.**OPERANDFLAG_UNDEFINED_EFFECT**
The operand role isn't fully defined

OperandFlag.**OPERANDFLAG_IMPLICIT**
The operand is implicit

class RegisterAccessType()
Register access type (read / write / rw)

RegisterAccessType.**REGISTER_READ**
Register is read.

RegisterAccessType.**REGISTER_WRITE**
Register is written.

RegisterAccessType.**REGISTER_READ_WRITE**
Register is read/written.

MemoryAccess

class MemoryAccess()
Object that describes a memory access

MemoryAccess.**accessAddress**
Address of accessed memory

MemoryAccess.**instAddress**
Address of instruction making the access

MemoryAccess.**size**
Size of memory access (in bytes)

MemoryAccess.**type**
Memory access type (READ / WRITE)

MemoryAccess.**value**
Value read from / written to memory

MemoryAccess.**flags**
Memory access flags

class MemoryAccessType()

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

MemoryAccessType.**MEMORY_READ**

Memory read access.

MemoryAccessType.**MEMORY_WRITE**

Memory write access.

MemoryAccessType.**MEMORY_READ_WRITE**

Memory read/write access.

class MemoryAccessFlags()

Memory access flags

MemoryAccessFlags.**MEMORY_NO_FLAGS**

Empty flag.

MemoryAccessFlags.**MEMORY_UNKNOWN_SIZE**

The size of the access isn't known.

MemoryAccessFlags.**MEMORY_MINIMUM_SIZE**

The given size is a minimum size.

MemoryAccessFlags.**MEMORY_UNKNOWN_VALUE**

The value of the access is unknown or hasn't been retrived.

VMEvent

class VMEvent()

Events triggered by the virtual machine.

VMEvent.**SEQUENCE_ENTRY**

Triggered when the execution enters a sequence.

VMEvent.**SEQUENCE_EXIT**

Triggered when the execution exits from the current sequence.

VMEvent.**BASIC_BLOCK_ENTRY**

Triggered when the execution enters a basic block.

VMEvent.**BASIC_BLOCK_EXIT**

Triggered when the execution exits from the current basic block.

VMEvent.**BASIC_BLOCK_NEW**

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

VMEvent.**EXEC_TRANSFER_CALL**

Triggered when the ExecBroker executes an execution transfer.

VMEvent.**EXEC_TRANSFER_RETURN**

Triggered when the ExecBroker returns from an execution transfer.

VMEvent.**SYSCALL_ENTRY**

Not implemented.

VMEvent.**SYSCALL_EXIT**

Not implemented.

VMEvent.**SIGNAL**

Not implemented.

class VMState()

Object that describes the current VM state

VMState.**event**

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

VMState.**sequenceStart**

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

VMState.**sequenceEnd**

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

VMState.**basicBlockStart**

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

VMState.**basicBlockEnd**

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

VMState.**lastSignal**

Not implemented.

Other globals

QBDI_LIB_FULLPATH

Fullpath of the QBDI library

class Options()

QBDI VM Options

Options.**NO_OPT**

Default value

Options.**OPT_DISABLE_FPR**

Disable all operation on FPU (SSE, AVX, SIMD). May break the execution if the target use the FPU.

Options.**OPT_DISABLE_OPTIONAL_FPR**

Disable context switch optimisation when the target execblock doesn't used FPR.

Options.**OPT_ATT_SYNTAX**

Used the AT&T syntax for instruction disassembly (for X86 and X86_64)

Options.**OPT_ENABLE_FS_GS**

Enable Backup/Restore of FS/GS segment. This option uses the instructions (RD|WR)(FS|GS)BASE that must be supported by the operating system.

class VMError()

Error return by the QBDI VM

VMError.**INVALID_EVENTID**

Returned event is invalid.

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's 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")

2.5.6 QBDIPreload API

Introduction

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 check out *Frida/QBDI API* instead.

QBDIPreload exploits these library injection mechanisms to hijack the normal program startup. During the hijacking process QBDIPreload will call 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 *Generate a 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 instrumenting a binary before the main function (inside the loader and the library constructors / init) as explained in *Limitations*.

Note: QBDIPreload is supposed to be used with `LD_PRELOAD` or `DYLD_INSERT_LIBRARIES` mechanisms to inject some code into the target process. Hence, the limitations of these also affect QBDIPreload (cannot inject suid binary, ...).

Initialisation

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

int **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.

Parameters
main – **[in]** Address of the main function

Returns
int QBDIPreload state

int **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.

Parameters

- **gprCtx** – **[in]** Original GPR context

- **fpuCtx** – **[in]** Original FPU context

Returns
int QBDIPreload state

int **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).

Parameters

- **argc** – **[in]** Original argc
- **argv** – **[in]** Original argv

Returns
int QBDIPreload state

int **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).

Parameters

- **vm** – **[in]** VM instance.
- **start** – **[in]** Start address of the range (included).
- **stop** – **[in]** End address of the range (excluded).

Returns
int QBDIPreload state

int **qbdipreload_on_exit**(int status)

Function called when process is exiting (using `_exit` or `exit`).

Parameters
status – **[in]** exit status

Returns
int QBDIPreload state

Helpers

int **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 – **[in]** Pointer to the main function

void **qbdipreload_threadCtxToGPRState**(const void *gprCtx, *GPRState* *gprState)

Convert a QBDIPreload GPR context (platform dependent) to a QBDI GPR state.

Parameters

- **gprCtx** – **[in]** Platform GPRState pointer

- **gprState** – **[in]** QBDI GPRState pointer

void **qbdipreload_floatCtxToFPRState**(const void *fprCtx, *FPRState* *fprState)

Convert a QBDIPreload FPR context (platform dependent) to a QBDI FPR state.

Parameters

- **fprCtx** – **[in]** Platform FPRState pointer
- **fprState** – **[in]** QBDI FPRState pointer

2.5.7 PyQBDIPreload API

Introduction

PyQBDIPreload consists of two main components:

- The QBDIPreload script included in the PyQBDI library. It will load the Python runtime and execute the user script in order to instrument the target.
- An injector script called *pyqbdipreload.py* that sets up the environment variables to inject the library and the Python runtime.

Note: PyQBDIPreload has the same limitations as QBDIPreload and PyQBDI.

User callback

pyqbdipreload_on_run(*vm:* pyqbdi.VM, *start: int*, *end: int*)

This function must be present in the preloaded script

Parameters

- **vm** (VM) – VM Instance
- **start** (*int*) – Start address of the range (included).
- **end** (*int*) – End address of the range (excluded).

QBDIPreload steps

qbdipreload_on_start() and *qbdipreload_on_premain()* are not available on PyQBDIPreload.

The Python interpreter and the preloaded script are loaded while executing *qbdipreload_on_main()*.

The *pyqbdipreload_on_run()* method is called while executing *qbdipreload_on_main()*.

The interpreter is shut down while executing *qbdipreload_on_exit()*, after calling all the registered methods of atexit module.

PyQBDIPreload script

You can use *pyqbdipreload.py* as follows:

```
$ python3 -m pyqbdipreload <script> <target> [<args> ...]
```

with:

- `script`: The Python script
- `target`: The binary you are targeting
- `args`: Argument(s) to be passed to the binary (if any)

2.6 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 mnemonic (or LLVM MC opcode). 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 an 8 bits immediate to a register.

2.6.1 AARCH64

The AARCH64 support with the Memory Access. The support is more recent that X86_64 and some bug should be expected.

Some extensions isn't supported by QBDI, in particular:

- Scalable Vector Extension (SVE and SVE2)
- Scalable Matrix Extension (SME)
- Memory Tagging Extension (MTE)
- Transactional Memory Extension (TME)

For OSX, the register X18 is defined as platforms reserved. QBDI doesn't set or used this register.

Local Monitor

The AARCH64 support includes an emulator of the Local Monitor. You can disable it with the option `OPT_DISABLE_LOCAL_MONITOR`.

Instruction Coverage

TODO

2.6.2 ARMv7

The ARMv7 has been refactor to support both ARM and Thumb mode with the Memory Access. The support is more recent that X86_64 and some bug should be expected.

Thumb Mode

When the CPU is in Thumb Mode, the LSB of `GPRState.PC` is set to 1. However, QBDI doesn't set the corrsponding byte in `GPRState.CPSR`. In addition, when an InstAnalysis is provided for a Thumb instruction, the `address` field always has his LSB set to 0 and the field `cpuMode` must be used to distinguish an ARM instruction from a Thumb instruction.

QBDI support the Thumb2 instruction IT. However, this instruction is handle internally. The `GPRState.CPSR` doesn't include the ITSTATE. To avoid undefined behavior, you must not: - Clear the VM cache during the execution of an ITBlock. - Start or restart the execution of QBDI in the middle of a ITBlock.

Local Monitor

The ARM support includes an emulator of the Local Monitor. You can disable it with the option `OPT_DISABLE_LOCAL_MONITOR`.

Instruction Coverage

```
ADCri
ADDri ADDrr ADDrsi
ANDri
Bcc BL BX_RET
CMNri
CMPri CMPrr
EORri
FCONSTD
FMSTAT
LDMIA_UPD
LDRBi12 LDRD LDRH LDRi12 LDR_PRE_IMM LDRrs
MOVi MOVr MOVsi
MVNi
ORRri ORRrr
STMDB_UPD
STRBi12 STRBrs STRD STRi12
SUBri
UBFX
```

(continues on next page)

(continued from previous page)

```
UXTB UXTH
VABSD VADD VCMPD VCMPE VCMPEZD VDIVD
VLDMDIA_UPD VLDRD VLDRS
VMLAD VMOVD VMOVDRR VMOVRRD
VMOVRS VMOVSR VMULD VNMLSD VSITOD
VSTMDDB_UPD VSTRD VSTRS
VSUBD VTOSIZD VTOUIZD VUITOD

tADC t2ADCri t2ADCrr t2ADCrs
tADDhirr tADDi3 tADDi8 tADDrr tADDrSPi tADDspi t2ADDri t2ADDri12 t2ADDrr t2ADDrs␣
↪t2ADDspImm t2ADDspImm12
tADR t2ADR
tAND t2ANDri t2ANDrr t2ANDrs
tASRri tASRrr t2ASRri t2ASRrr
tB tBcc tBL tBLXi tBLXr tBX t2B t2Bcc
t2BICri t2BICrr t2BICrs
t2CLZ
tCBNZ tCBZ
tCMNz t2CMNri t2CMNzrr
tCMPhir tCMPi8 tCMPr t2CMPri t2CMPrr t2CMPrs
t2DMB
tEOR t2EORri t2EORrr t2EORrs
tHINT t2HINT
t2IT
tLDMIA t2LDMIA t2LDMIA_UPD
t2LDRBi12 t2LDRBi8
tLDRBi tLDRBr t2LDRB_POST t2LDRB_PRE t2LDRBs
t2LDRDi8
t2LDREX
tLDRHi tLDRHr t2LDRHi12 t2LDRHi8 t2LDRH_POST t2LDRHs
tLDRi tLDRpci tLDRr tLDRspi t2LDRi12 t2LDRi8 t2LDRpci t2LDR_POST t2LDR_PRE t2LDRs
tLDRSB t2LDRSBi12
tLSLri tLSLrr tLSRri tLSRrr t2LSLri t2LSLrr t2LSRri t2LSRrr
t2MLA t2MLS
tMOVi8 tMOVr tMOVSr t2MOVi t2MOVi16 t2MOVTi16
t2MRC
t2MUL
tMVN t2MVNi t2MVNr
t2ORNri t2ORNrr
tORR t2ORRri t2ORRrr t2ORRrs
tPOP
tPUSH
tREV tREV16 t2REV
t2RORri t2RSBri
tRSB
tSBC t2SBCri t2SBCrr t2SBCrs
t2SMULL
tSTMIA_UPD t2STMDB_UPD t2STMIA t2STMIA_UPD
tSTRBi tSTRBr t2STRBi12 t2STRBi8 t2STRB_POST t2STRB_PRE t2STRBs
t2STRDi8 t2STRD_PRE
t2STREX
tSTRHi tSTRHr t2STRHi12 t2STRHi8 t2STRH_PRE t2STRHs
```

(continues on next page)

(continued from previous page)

```
tSTRi TSTri tSTRr tSTRspi t2STRi12 t2STRi8 t2STR_POST t2STR_PRE t2STRs
tSUBi3 tSUBi8 tSUBrr tSUBspi t2SUBri t2SUBri12 t2SUBrr t2SUBrs t2SUBspImm t2SUBspImm12
tSXTB tSXTH
t2TBB t2TBH
t2TEQri
tTST t2TSTri t2TSTrr
t2UBFX
t2UMLAL t2UMULL
tUXTB
tUXTH
t2UXTAH t2UXTB
```

2.6.3 Intel x86-64

The x86-64 support is complete and mature. 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 to the covered ones. We currently don't support the following features:

- AVX512: the register of this extension isn't supported and will not be restored/backup during the execution
- privileged instruction: QBDI is an userland (ring3) application and privileged registers aren't managed
- CET feature: shadow stack is not implemented and the current instrumentation doesn't support indirect branch tracking.
- HLE and RTM features and any instructions for multithreading: QBDI allows inserting callbacks at any position and cannot guarantee that instructions of the same transactions unit will not be split.
- MPX feature: bound registers aren't supported.
- XOP instructions for ADM processors: The instructions are never been tested.

The memory access information is provided for most of general and SIMD instructions. The information is missing for:

- The instructions include in an unsupported feature
- VGATHER* and VPGATHER* instructions of AVX2.

Instruction Coverage

```
ADC32mi8 ADC64mi8
ADD16mi8 ADD16mr ADD16rm ADD32i32 ADD32mi8 ADD32mr ADD32ri ADD32ri8 ADD32rm ADD32rr␣
↪ADD64i32 ADD64mi32 ADD64mi8 ADD64mr ADD64ri32 ADD64ri8 ADD64rm ADD64rr ADD8rr ADDSDrm_
↪Int ADDSDrr_Int
AESENCLASTrr AESENCrr
AND16mi AND16mr AND32i32 AND32mi8 AND32mr AND32ri AND32ri8 AND32rm AND32rr AND64mi8␣
↪AND64mr AND64ri8 AND64rr AND8mi AND8mr AND8ri AND8rm AND8rr ANDNPDrr ANDPDrm ANDPDrr
BSWAP32r BSWAP64r BT64rr
CALL64m CALL64pcrel32 CALL64r
CDQ CDQE
CMOV32rm CMOV32rr CMOV64rm CMOV64rr
CMP16mi8 CMP16mr CMP16rm CMP16rr CMP32i32 CMP32mi CMP32mi8 CMP32mr CMP32ri CMP32ri8␣
↪CMP32rm CMP32rr CMP64i32 CMP64mi32 CMP64mi8 CMP64mr CMP64ri32 CMP64ri8 CMP64rm CMP64rr␣
```

(continues on next page)

(continued from previous page)

```
↪CMP8i8 CMP8mi CMP8mr CMP8ri CMP8rm CMP8rr
CMPSB
CMPSDrr_Int
CMPXCHG32rm CMPXCHG64rm
COMISDrm COMISDrr
CQO
CVTSI2SDrr_Int CVTSI642SDrm_Int CVTSI642SDrr_Int CVTTSD2SI64rr_Int CVTTSD2SIrr_Int
DEC32m DEC32r
DIV32r DIV64m DIV64r DIVSDrm_Int DIVSDrr_Int
IDIV32m IDIV32r IDIV64r
IMUL32r IMUL32rm IMUL32rr IMUL64rmi8 IMUL64rr IMUL64rri32
JCC_1 JCC_4
JMP64m JMP64r
JMP_1 JMP_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_Int
NEG32r NEG64r
NOOP NOOP NOOPW
NOT32r NOT64m NOT64r
OR16rm OR32i32 OR32mi OR32mi8 OR32mr OR32ri OR32ri8 OR32rm OR32rr OR64i32 OR64ri8 OR64rm␣
↪OR64rr OR8i8 OR8mi OR8mr OR8ri OR8rm OR8rr
ORPDrr
POP64r
PSHUFBrr PSHUFDri
PSLLDQri PSLLDri
PUSH64i8 PUSH64r PUSH64rmm
PXORrr
RETQ
ROL32r1 ROL32ri ROL64r1 ROL64ri
ROR32ri ROR64r1 ROR64ri
ROUNDSDr_Int
SAR32r1 SAR32rCL SAR32ri SAR64r1 SAR64ri
SBB32ri8 SBB32rr SBB64ri8 SBB64rr SBB8i8 SBB8ri
SCASB
SETCCm SETCCr
SHL32rCL SHL32ri SHL64rCL SHL64ri
SHR16ri SHR32r1 SHR32rCL SHR32ri SHR64r1 SHR64rCL SHR64ri SHR8r1 SHR8ri
STOSQ
SUB32mi8 SUB32mr SUB32ri SUB32ri8 SUB32rm SUB32rr SUB64mi8 SUB64mr SUB64ri32 SUB64ri8␣
↪SUB64rm SUB64rr SUB8mr SUBSDrm_Int SUBSDrr_Int
```

(continues on next page)

(continued from previous page)

```
SYSCALL
TEST16mi TEST16ri TEST16rr TEST32i32 TEST32mi TEST32mr TEST32ri TEST32rr TEST64ri32␣
↪TEST64rr TEST8i8 TEST8mi TEST8ri TEST8rr
UCOMISDrr
VADDSDrm_Int VADDSDrr_Int
VANDPDrr
VCOMISDrr
VFMADD132SDm_Int VFMADD132SDr_Int VFMADD213SDm_Int VFMADD231SDr_Int
VFNMADD231SDm_Int
VMOVAPDrr
VMOVPDI2DIrr
VMOVPQIto64rr
VMOVQI2PQIrm
VMOVSDrm
VMULSDrm_Int VMULSDrr_Int
VSTMXCSR
VSUBSDrm_Int VSUBSDrr_Int
VUCOMISDrm VUCOMISDrr
VXORPDrr
XADD32rm
XCHG32rm XCHG64rr
XOR32ri XOR32ri8 XOR32rm XOR32rr XOR64rm XOR64rr XOR8rm
XORPSrr
```

2.6.4 Intel x86

The x86 support is based on x86_64 and has the same limitations. However, its integration is more recent than X86_64 and has less been tested.

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 ADD_FPrST0 ADD_FrST0
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
CMOV32rm CMOV32rr
CMOVE_F
CMP16mi8 CMP16mr CMP16rm CMP32i32 CMP32mi CMP32mi8 CMP32mr CMP32ri CMP32ri8 CMP32rm␣
↪CMP32rr CMP8i8 CMP8mi CMP8mr CMP8ri CMP8rm CMP8rr
CMPSB CMPSW
```

(continues on next page)

(continued from previous page)

```
CMPXCHG32rm
COM_FIPr
COM_FIr
DEC32r_alt
DIV32m DIV32r
DIVR_F32m DIVR_F64m
DIV_F32m DIV_F64m
DIV_FPrST0
FCOMP64m
FLDCW16m
FLDENVm
FNSTCW16m FNSTSW16r
FRNDINT
FSTENVm
FXAM
IDIV32m
ILD_F32m ILD_F64m
IMUL32r IMUL32rm IMUL32rmi8 IMUL32rr IMUL32rri
INC32r_alt
IST_FP32m IST_FP64m
JCC_1 JCC_4
JMP32m JMP32r
JMP_1 JMP_4
LD_F0 LD_F1 LD_F32m LD_F64m LD_F80m LD_Frr
LEA32r
MOV16mi MOV16mr MOV16rm MOV32ao32 MOV32mi MOV32mr MOV32o32a MOV32ri MOV32rm MOV32rr␣
↪MOV8mi MOV8mr 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
OR16ri OR16rm OR32i32 OR32mi OR32mi8 OR32mr OR32ri 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
```

(continues on next page)

(continued from previous page)

```
SCASB
SETCCm SETCCr
SHL32rCL SHL32ri
SHLD32rrCL SHLD32rri8
SHR16ri SHR32r1 SHR32rCL SHR32ri SHR8m1 SHR8ri
SHRD32rri8
STOSL
ST_F64m ST_FP64m ST_FP80m ST_FPrr
SUB32i32 SUB32mi8 SUB32mr SUB32ri SUB32ri8 SUB32rm SUB32rr SUB8mr SUB8ri SUB8rm
SUBR_F64m SUBR_FPrST0
SUB_FPrST0 SUB_FrST0
TEST16mi TEST16ri TEST16rr TEST32i32 TEST32mi TEST32mr TEST32ri TEST32rr TEST8i8 TEST8mi␣
↪TEST8mr TEST8ri TEST8rr
UCOM_FIr UCOM_FPr
XADD32rm
XCHG32ar XCHG32rm XCHG32rr
XCH_F
XOR16rr XOR32i32 XOR32mr XOR32ri XOR32ri8 XOR32rm XOR32rr XOR8rm
XORPSrr
```

2.7 Developer Documentation

2.7.1 Compilation From Source

To build this project, the following dependencies are needed on your system:

- cmake >= 3.12
- ninja or make
- C++17 toolchain (gcc, clang, Visual Studio 2019, ...)

A local version of llvm is statically built within QBDI 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.

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 mode 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/config-linux-X86_64.sh
ninja
```

x86

You can follow the same instructions as for x86-64 but instead, use the `config-linux-X86.sh` configuration script.

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 ninja`)

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/config-macOS-X86_64.sh
ninja
```

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 and Ninja.

First of all, the Visual Studio environment must be set up. This can be done with a command such as:

```
"C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build\
↪vcvarsall.bat" x64
```

Then, the following commands must be run:

```
mkdir build
cd build
python ../cmake/config/config-win-X86_64.py
ninja
```

Android

Cross-compiling for Android requires the NDK (or the SDK) to be installed on your workstation. For now, it has only been tested under Linux. If not already installed, you can download the latest Android NDK package through the official website and extract it. Afterwards, the `config-android-*.sh` configuration script needs to be customised to match your NDK installation directory and the target platform.:

```
# Configure and compile QBDI X86_64 with a NDK
mkdir build && cd build
NDK_PATH=<your_NDK_PATH> ../cmake/config/config-android-X86_64.sh
```

(continues on next page)

(continued from previous page)

```
ninja

# Configure and compile QBDI X86 with a SDK
mkdir build && cd build
ANDROID_SDK_ROOT=<your_SDK_PATH> ../cmake/config/config-android-X86.sh
ninja
```

PyQBDI compilation

The PyQDBI library (apart from the wheel package) can be built by solely passing the **'-DQBDI_TOOLS_PYQBDI=ON'** option to the CMake build system.

However, if you want to build the wheel package, you can run these commands:

```
python -m pip install --upgrade pip
python -m pip install setuptools wheel build
python -m build -w
```

A 32-bit version of Python is mandatory for the X86 architecture whereas a 64-bit one is required for the X86-64 architecture.

CMake Parameters

The compilation of QBDI can be configured with the command line. Each parameter should be placed on the command line with the form `-D<param>=<value>`.

- `QBDI_PLATFORM` (mandatory) : Target platform of the compilation (supported: `windows`, `linux`, `android`, `osx`)
- `QBDI_ARCH` (mandatory) : Target architecture of the compilation (supported: `X86_64`, `X86`)
- `QBDI_CCACHE` (default ON) : enable compilation optimisation with ccache or sccache.
- `QBDI_DISABLE_AVX` (default OFF) : disable the support of AVX instruction on X86 and X86_64
- `QBDI_ASAN` (default OFF) : compile with ASAN to detect memory leak in QBDI.
- `QBDI_LOG_DEBUG` (default OFF) : enable the debug level of the logging system. Note that the support of this level has an impact on the performances, even if this log level is not enabled.
- `QBDI_STATIC_LIBRARY` (default ON) : build the static library of QBDI. Note than some subproject need `QBDI_STATIC_LIBRARY` (test, PyQBDI, ...)
- `QBDI_SHARED_LIBRARY` (default ON) : build the shared library of QBDI. Note than some subproject need `QBDI_SHARED_LIBRARY` (Frida/QBDI, examples, ...)
- `QBDI_TEST` (default ON) : build the tests suite
- `QBDI_BENCHMARK` (default OFF) : build the benchmark tools
- `QBDI_TOOLS_QBDIPRELOAD` (default ON on supported platform) : build QBDIPreload static library (supported on Linux and OSX).
- `QBDI_TOOLS_VALIDATOR` (default ON on supported platform) : build the validator library (supported on Linux and OSX).
- `QBDI_TOOLS_PYQBDI` (default ON on X86_64) : build PyQBDI library. Supported on Linux, Windows and OSX.

- `QBDI_TOOLS_FRIDAQBDI` (default ON) : add Frida/QBDI in the package.

2.7.2 Repository Organization

Root Tree

The root of the source tree is organized as follows:

cmake/
Contains files required by the CMake build system.

docker/
Contains the Dockerfile for the docker image and the CI.

docs/
Contains this documentation source and configuration.

examples/
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.

third-party/
Contains the third party dependency downloaded by cmake.

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.

2.7.3 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 Catch2 which verifies the essential functions of QBDI. This test suite is automatically built and made available in the `test/` build subdirectory:

```
$ ./test/QBDITest
===============================================================================
All tests passed (110977 assertions in 106 test cases)
```

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.

2.7.4 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.

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

[1] https://en.wikipedia.org/wiki/Domain-specific_language

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.

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 **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(const LLVMCPUs &llvmCPUs, *VMInstanceRef* vminstance = nullptr, const std::vector<std::unique_ptr<RelocatableInst>> *execBlockPrologue = nullptr, const std::vector<std::unique_ptr<RelocatableInst>> *execBlockEpilogue = nullptr, uint32_t epilogueSize = 0)

Construct a new *ExecBlock*

Parameters

- **llvmCPUs** – **[in]** LLVMCPU used to assemble instructions in the *ExecBlock*.
- **vminstance** – **[in]** Pointer to public engine interface
- **execBlockPrologue** – **[in]** cached prologue of ExecManager

[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.

[2] This is at least true for x86, x86_64, ARM, ARM64 and PowerPC.

- **execBlockEpilogue** – **[in]** cached epilogue of ExecManager
- **epilogueSize** – **[in]** size in bytes of the epilogue (0 is not know)

void **changeVMInstanceRef**(*VMInstanceRef* vminstance)

Change vminstance when *VM* object is moved

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.

SeqWriteResult **writeSequence**(std::vector<Patch>::const_iterator seqStart,
std::vector<Patch>::const_iterator seqEnd)

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.

Parameters

- **seqStart** – [in] Iterator to the start of a list of patches.
- **seqEnd** – [in] Iterator to the end of a list of patches.

Returns
A structure detailling the write operation result.

uint16_t **splitSequence**(uint16_t instID)

Split an existing sequence at instruction instID to create a new sequence.

Parameters
instID – [in] ID of the instruction where to split the sequence at.

Returns
The new sequence ID.

inline *rword* **getDataBlockBase**() const

Get the address of the DataBlock

Returns
The DataBlock offset.

inline *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.

Returns
The computed offset.

inline *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.

Returns
The computed offset.

inline uint32_t **getEpilogueSize**() const

Get the size of the epilogue

Returns
The size of the epilogue.

inline *rword* **getCurrentPC**() const

Obtain the value of the PC where the *ExecBlock* is currently writing instructions.

Returns
The PC value.

inline uint16_t **getNextInstID**() const

Obtain the current instruction ID.

Returns
The current instruction ID.

uint16_t **getInstID**(*rword* address, CPUMode cpumode) const

Obtain the instruction ID for a specific address (the address must exactly match the start of the instruction).

Parameters

- **address** – The address of the start of the instruction.
- **cpumode** – The mode of the instruction

Returns
The instruction ID or NOT_FOUND.

inline uint16_t **getCurrentInstID**() const

Obtain the current instruction ID.

Returns
The ID of the current instruction.

const InstMetadata &**getInstMetadata**(uint16_t instID) const

Obtain the instruction metadata for a specific instruction ID.

Parameters
instID – The instruction ID.

Returns
The metadata of the instruction.

rword **getInstAddress**(uint16_t instID) const

Obtain the instruction address for a specific instruction ID.

Parameters
instID – The instruction ID.

Returns
The real address of the instruction.

rword **getInstInstrumentedAddress**(uint16_t instID) const

Obtain the instrumented address for a specific instruction ID.

Parameters
instID – The instruction ID.

Returns
The address in the BasicBlock of the instruction.

const llvm::MCInst &**getOriginalMCInst**(uint16_t instID) const

Obtain the original MCInst for a specific instruction ID.

Parameters
instID – The instruction ID.

Returns
The original MCInst of the instruction.

const *InstAnalysis* ***getInstAnalysis**(uint16_t instID, *AnalysisType* type) const

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

Parameters

- **instID** – **[in]** The ID of the instruction to analyse.
- **type** – **[in]** Properties to retrieve during analysis.

Returns
A *InstAnalysis* structure containing the analysis result.

inline uint16_t **getNextSeqID**() const

Obtain the next sequence ID.

Returns
The next sequence ID.

uint16_t **getSeqID**(*rword* address, CPUMode cpumode) const

Obtain the sequence ID for a specific address (the address must exactly match the start of the sequence).

Parameters
address – The address of the start of the sequence.

Returns
The sequence ID or NOT_FOUND.

uint16_t **getSeqID**(uint16_t instID) const

Obtain the sequence ID containing a specific instruction ID.

Parameters
instID – The instruction ID.

Returns
The sequence ID or NOT_FOUND.

inline uint16_t **getCurrentSeqID**() const

Obtain the current sequence ID.

Returns
The ID of the current sequence.

uint16_t **getSeqStart**(uint16_t seqID) const

Obtain the sequence start address for a specific sequence ID.

Parameters
seqID – The sequence ID.

Returns
The start address of the sequence.

uint16_t **getSeqEnd**(uint16_t seqID) const

Obtain the instruction id of the sequence end address for a specific sequence ID.

Parameters
seqID – The sequence ID.

Returns
The end address of the sequence.

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.

inline Context ***getContext**() const

Get a pointer to the context structure stored in the data block.

Returns
The context pointer.

uint16_t **newShadow**(uint16_t tag = ShadowReservedTag::Untagged)

Allocate a new shadow within the data block. Used by relocation to load or store data from the instrumented code.

Parameters
tag – The tag associated with the registration, 0xFFFF is reserved for unregistered shadows.

Returns
The shadow id (which is its index within the shadow array).

uint16_t **getLastShadow**(uint16_t tag)

Search the last *Shadow* with the tag for the current instruction. Used by relocation to load or store data from the instrumented code.

Parameters
tag – The tag associated with the registration

Returns
The shadow id (which is its index within the shadow array).

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.

Parameters
id – [in] ID of the shadow.

Returns
Value of the shadow.

rword **getShadowOffset**(uint16_t id) const

Get the offset of a shadow within the data block.

Parameters
id – [in] ID of the shadow.

Returns
Offset 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 **Reg**

Structure representing a register variable in PatchDSL.

Public Functions

inline **Reg**(unsigned int id)

Create a new register variable.

Parameters
id – **[in]** The id of the register to represent.

inline unsigned int **getID**() const

Get back the id of the register in GPRState

Returns
GPRState register id.

inline **operator RegLLVM**() const

Convert this structure to an LLVM register id.

Returns
LLVM register id.

inline unsigned int **getValue**() const

Get the llvm value of the register

Returns
llvm register id.

inline *rword* **offset**() const

Return the offset of this register storage in the context part of the data block.

Returns
The offset.

inline bool **operator<**(const *Reg* &o) const

Needed to create a std::set

struct **Temp**

Structure representing a temporary register variable in PatchDSL.

Public Functions

inline **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 – **[in]** The id of the temp to represent.

inline **operator unsigned int**() const

Convert this *Temp* to its id.

Returns
This *Temp* id.

struct **Shadow**

Structure representing a shadow variable in PatchDSL.

Public Functions

inline **Shadow**(uint16_t tag)

Allocate a new shadow variable in the data block with the corresponding tag.

Parameters
tag – **[in]** The tag of the new shadow variable.

inline *rword* **getTag**() const

Return the tag associated with this shadow variable.

Returns
The tag of the shadow variable.

struct **Constant**

Structure representing a constant value in PatchDSL.

Public Functions

inline **Constant**(*rword* v)

Represent a constant value.

Parameters
v – **[in]** The represented value.

inline **operator** ***rword***() const

Convert this structure to its value.

Returns
This constant value.

struct **Offset**

Structure representing a memory offset variable in PatchDSL.

Public Functions

inline **Offset**(int64_t offset)

Allocate a new offset variable with its offset value.

Parameters
offset – **[in]** The offset value

inline **Offset**(*Reg* reg)

Allocate a new offset variable with the offset in the context of a specific register.

Parameters
reg – **[in]** The register whose offset to represent.

inline **operator int64_t**() const

Convert this structure to its value.

Returns
This offset value.

struct **Operand**

Structure representing an operand instruction variable in PatchDSL.

Public Functions

inline **Operand**(unsigned int idx)

Represent an operand instruction identified by its index in the LLVM MCInst representation of the instruction.

Parameters
idx – **[in]** The operand index.

inline **operator unsigned int**() const

Convert this *Operand* to its idx.

Returns
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 **PatchRule**

A patch rule written in PatchDSL.

Public Functions

PatchRule(std::unique_ptr<PatchCondition> &&condition, std::vector<std::unique_ptr<PatchGenerator>> &&generators)

Allocate a new patch rule with a condition and a list of generators.

Parameters

- **condition** – **[in]** A PatchCondition which determine wheter or not this *PatchRule* applies.
- **generators** – **[in]** A vector of PatchGenerator which will produce the patch instructions.

bool **canBeApplied**(const Patch &patch, const LLVMCPU &llvmcpu) const

Determine wheter this rule applies by evaluating this rule condition on the current context.

Parameters

- **patch** – **[in]** The Patch to check
- **llvmcpu** – **[in]** LLVMCPU object

Returns
True if this patch condition evaluate to true on this context.

void **apply**(Patch &patch, const LLVMCPU &llvmcpu) const

Generate this rule output patch by evaluating its generators on the current context. Also handles the temporary register management for this patch.

Parameters

- **patch** – **[in]** The Patch where to apply the rule
- **llvmcpu** – **[in]** LLVMCPU object

class **InstrRule**

An instrumentation rule written in PatchDSL.

Subclassed by QBDI::AutoUnique< InstrRule, InstrRuleBasicCBK >, QBDI::AutoUnique< InstrRule, InstrRuleDynamic >, QBDI::AutoUnique< InstrRule, InstrRuleUser >

Public Functions

virtual bool **tryInstrument**(Patch &patch, const LLVMCPU &llvmcpu) const = 0

Determine wheter this rule have to be apply on this Path and instrument if needed.

Parameters

- **patch** – **[in]** The current patch to instrument.
- **llvmcpu** – **[in]** LLVMCPU object

void **instrument**(Patch &patch, const PatchGeneratorUniquePtrVec &patchGen, bool breakToHost, *InstPosition* position, int priority, RelocatableInstTag tag) const

Instrument a patch by evaluating its generators on the current context. Also handles the temporary register management for this patch.

Parameters

- **patch** – **[in]** The current patch to instrument.
- **patchGen** – **[in]** The list of patchGenerator to apply
- **breakToHost** – **[in]** Add a break to *VM* need to be add after the patch
- **position** – **[in]** Add the patch before or after the instruction
- **priority** – **[in]** The priority of this patch
- **tag** – **[in]** The tag for this patch

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)))
```

(continues on next page)

(continued from previous page)

```
    }
)
```

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::UniquePtrVec `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.

Parameters

- **cbk** – **[in]** The callback function to call.
- **data** – **[in]** The data to pass as an argument to the callback function.

Returns

A list of PatchGenerator to set up this callback call.

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)))
    }
```

(continues on next page)

(continued from previous page)

```
    // Position this instrumentation after the instruction
    QBDI::InstPosition::POSTINST,
    // Break to the host after the instrumentation (required for the callback to be made)
    true
));
```

2.8 CHANGELOG

2.8.1 Version (0.11.0)

2024-05-17 QBDI Team <qbdi@quarkslab.com>

- Fix ARM instrumentation for 'mov pc, lr' (#241)
- Add switchStackAndCall API (#245)
- Rename `QBDI::InstAnalysis::module` and `QBDI::InstAnalysis::symbol` to `QBDI::InstAnalysis::moduleName` and `QBDI::InstAnalysis::symbolName`. The same changed applied in C, C++, Python and JS API, but Python and JS API deprecated but still support the previous name.
- Update LLVM to LLVM17 (#253)
- Support copy and pickle for GPRState and FPRState in PyQBDI (#247, #248)
- Support python 3.12 (#247)

2.8.2 Version 0.10.0

2023-01-26 QBDI Team <qbdi@quarkslab.com>

- Fix Ubuntu package (#217)
- Support ARMv7 and AArch64 architecture (#222)
- Support python 3.11 (#222)
- Support Frida >= 15.2 (#222 and #223)

Internal update:

- Move windows CI to Github Actions (#222)
- Support python build with `pyproject.toml` (#222)
- Update LLVM to LLVM15 (#224)
- Add CI for ARMv7 and AArch64 (#222 and #225)

2.8.3 Version 0.9.0

2022-03-31 QBDI Team <qbdi@quarkslab.com>

- Change internal log system (#174).
  - The API `QBDI::addLogFilter` has been replaced by *QBDI::setLogPriority()*.
  - The API `QBDI::setLogOutput` has been replaced by *QBDI::setLogFile()*, *QBDI::setLogConsole()* and *QBDI::setLogDefault()*.
- Fix templates (#186)
- Fix Frida-QBDI for Frida 15.0.0 (#192)
- Change behavior of *QBDI::VM::addInstrumentedModuleFromAddr()* to work with mmap region (#193)
- Add Priority to InstCallback API (#194). If two or more InstCallback target the same position (PRE or POST) of the same instruction, the priority parameter allows to specify which InstCallback should be called first. see *QBDI::CallbackPriority*
- Support for X86 `loop`, `loope` and `loopne` instructions (#200)
- Add support for `FS` and `GS` segment in X86_64 (#190). To support the feature, the kernel must support `RDFSBASE`, `RDGSBASE`, `WRFSBASE` and `WRGSBASE` instructions (linux >= 5.9). To enable the support, the option *QBDI::Options::OPT_ENABLE_FS_GS* must be enabled.
- Hide LLVM symbols from shared library and QBDIPreload (#205)
- Support python 3.10 for PyQBDI (#206)
- Add VMAction *QBDI::VMAction::SKIP_INST* and *QBDI::VMAction::SKIP_PATCH* (#197)
  - *QBDI::VMAction::SKIP_INST* can be used to emulate the instruction with a PREINST callback. When this VMAction is returned by a PREINST callback, QBDI will directly jump to the POSTINST callback.
  - *QBDI::VMAction::SKIP_PATCH* can be used to jump over all the reminding callback for the current instruction. If uses in PREINST position, the instruction will not be executed.

  The value associated with the existing *QBDI::VMAction* has changed.
- Add tutorial for basic block VMEvent (#165)
- Support C++ lambda with capture. (#207) see *QBDI::InstCbLambda*, *QBDI::VMCbLambda* and *QBDI::InstrRuleCbLambda*
- Fix a bug where some symbols were missing in QBDIPreload (#209)
- Remove new name of libc in QBDIPreload (#211)
- Support of some self-modifying code (#212). QBDI will not crash if invalid instructions are found when handling a new basic block.
- Add tutorial for ExecBroker VMEvent (#166)
- Change *QBDI::getVersion()* out parameter to return version to the form `0xMMmmpp` (#214)

Internal update:

- Add static library licenses in LICENSE.txt (#169)
- Format code with clang-format and cmake-format (#175)
- Fix integer overflow in cache optimisation (#168)
- Refactor build system, add llvm build in main cmake with FetchContent, move all LLVM object in a new LLVMCPU class, split internal class, split state by architecture (#178, #179 and #188)

- Update LLVM to LLVM 13.0.0, remove zlib dependency (#180, #196)
- Remove empty Patch not associated to an MCInst (#195)
- Compile assembly with `--noexecstack` to have a `rw-` stack when using QBDI on linux (#201)
- Use build directory to build the documentation (#213)
- Use Doxygen 1.9.2 in readthedocs (#214)

2.8.4 Version 0.8.0

2021-02-11 QBDI Team <qbdi@quarkslab.com>

- Fix android compilation (#126)
- Fix instrumentation of Pusha and Popa on X86 (#127)
- Fix getBBMemoryAccess (#128)
  - Improve the documentation of getBBMemoryAccess
  - Add recordMemoryAccess callback before any InstCallback
- Refactor ExecBlockManager to work with unaligned instruction on X86 and X86-64 (#129)
- Drop early support for ARM. The support hasn't been tested since 0.6.2.
- Rework cmake package export to import X86 and X86_64 version of QBDI in one CMake (#146 and #132)
- Add *QBDI::VM::getCachedInstAnalysis()* to retrieve an InstAnalysis from an address. The address must be cached in the VM. (#148)
- Change in `InstAnalysis` and `OperandAnalysis` (#153):
  - Add `InstAnalysis.flagsAccess` to determine if the instruction uses or sets the flags (`EFLAGS` register). The analysis `ANALYSIS_OPERANDS` is needed to use this field.
  - Change `InstAnalysis.mayLoad` and `InstAnalysis.mayStore` definition. The field will be true if QBDI detects memory access for the instruction.
  - Add `InstAnalysis.loadSize` and `InstAnalysis.storeSize`. If the instruction will read or write the memory, the expected size of the access is given by these fields. The analysis `ANALYSIS_INSTRUCTION` is needed to use this field.
  - Add `InstAnalysis.condition`. With the update of LLVM, the mnemonic for conditional jump (like `JE_4`) are merged in a unique mnemonic `JCC_4`. This new field will contain the condition. The analysis `ANALYSIS_INSTRUCTION` is needed to use this field. A new enum `ConditionType` has all the possible value.
  - Add `OPERANDFLAG_IMPLICIT` for `OperandAnalysis.flag`. An operand will have this flag when a register is implicit to the instruction.
  - Add `OPERAND_FPR` for `OperandAnalysis.type`. This type is used for floating point registers. For this type, `OperandAnalysis.regCtxIdx` is the offset in `FPRState` or -1 when an offset cannot be provided.
  - Add `OPERAND_SEG` for `OperandAnalysis.type`. This type is used for segments or other unsupported register (like `SSP`).
  - Change type of `OperandAnalysis.regCtxIdx` to signed integer. When the value is less than 0, the index is invalid.
  - Change algorithm for `OperandAnalysis`. The type `OPERAND_INVALID` may be present in the list of operands when a register is unset with the current instruction. Many operands may describe the used of the same register when a register is used multiple times for different purposes by the instruction.

- Add Instrumentation rule callback `QBDI_InstrRuleDataCBK` and *QBDI::InstrRuleDataCBK* (#151)

  The Instrumentation rule callback receives an InstAnalysis of each instruction during the instrumentation process. Based on this analysis, the callback may insert custom InstCallback for each instruction.

  The call order of the callback has changed for the PREINST callback. If an instruction has multiple callbacks in PREINST position, they will be called in the reverse order of registration.

- Support SIMD MemoryAccess and change *QBDI::MemoryAccess* structure (#154)
  - Add *QBDI::MemoryAccess::flags*. In some cases, QBDI cannot provide all information about the access. This field describes the limitation for each access. Three limitations may be reached:
    - *QBDI::MemoryAccessFlags::MEMORY_UNKNOWN_SIZE*: the size of the access isn't known. Only the address is valid. The flag is only set for instruction with REP prefix before the execution of the instruction.
    - *QBDI::MemoryAccessFlags::MEMORY_MINIMUM_SIZE*: the size isn't the real size of the access, but the expected minimal size. This flag is used for instruction with complex access like `XSAVE` and `XRSTOR`.
    - *QBDI::MemoryAccessFlags::MEMORY_UNKNOWN_VALUE*: the value of the access hasn't been saved. The more common reason is that the access size is greater than the size of *QBDI::MemoryAccess::value*. This flag is also used for instruction with REP prefix when the access size cannot be determined during the instrumentation.
  - Fix MemoryAccess for some generic instruction.

- Add VM Options. (#144)

  Some options can be provided to the VM to enable or disable some features:
  - *QBDI::Options::OPT_DISABLE_FPR*: Disable FPRState backup and restore in context switches. Only the GPRState will be used.
  - *QBDI::Options::OPT_DISABLE_OPTIONAL_FPR*: When *QBDI::Options::OPT_DISABLE_FPR* isn't selected, QBDI will detect if a BasicBlock needs FPRState. When BasicBlock doesn't need FPRState, the state will not be restored. This option forces the restoration and backup of FPRState to every BasicBlock.
  - *QBDI::Options::OPT_ATT_SYNTAX* for X86 and X86_64: *QBDI::InstAnalysis::disassembly* will be in AT&T syntax instead of Intel Syntax.

- Rework documentation (#156)

Internal update:

- Update LLVM to LLVM 10.0.1 (#104 and #139)
- Reduce LLVM library included in QBDI static library and reduce QBDI package size (#139 and #70)
- Replace GTest by Catch2 (#140)
- Refactor code and switch to cpp17 (#140 and #155)
- Use Github Actions to build dev-next package of QBDI (linux, osx and android) and PyQBDI (linux and osx) (#147, #159)
- Rewrite frida-qbdi.js and use sphinx-js for frida-QBDI documentation (#146). A version of frida greater or equals to 14.0 is needed to run frida-qbdi.js (need support of ES2019).
- Refactor MemoryAccess Code and add new tests (#154)
- Handle VMCallback return value (#155)
- Optimize Context Switch and FPRState restoration (#144)

- Add commit hash in devel version (#158)

2.8.5 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)

2.8.6 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:␣
```

(continues on next page)

(continued from previous page)

```
↪(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
```

2.8.7 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)

2.8.8 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 precaching 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.

2.8.9 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)

2.8.10 Version 0.5

2017-12-22 Cedric TESSIER <ctessier@quarkslab.com>

- Official public release!

2.8.11 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

2.8.12 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

2.8.13 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

2.8.14 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.

2.8.15 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.

2.8.16 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

2.8.17 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

2.8.18 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

2.9 References

2.9.1 Conferences and Workshops

- 2020-10-15: BSides Delhi - Analysing programs through dynamic instrumentation with QBDI by Tom Czayka and Nicolas Surbayrole
- 2020-06-29: Pass The Salt - Why are Frida and QBDI a Great Blend on Android? by Tom Czayka
- 2019-04-23: French-Japan cybersecurity workshop - Fuzzing binaries using Dynamic Instrumentation by Paul Hernault
- 2017-12-28: 34C3 conferences - Implementing an LLVM based Dynamic Binary Instrumentation framework by Charles Hubain and Cédric Tessier

2.9.2 Blog Post

- 2020-08-18: Introduction to Whiteboxes and Collision-Based Attacks With QBDI by Paul Hernault
- 2020-08-04: Why are Frida and QBDI a Great Blend on Android? by Tom Czayka
- 2019-09-10: QBDI 0.7.0
- 2019-06-03: Android Native Library Analysis with QBDI by Romain Thomas
- 2018-01-24: Slaying Dragons with QBDI by Paul Hernault

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

P

Q

R

S

T

U

V

W

X

Y