Wisecracker™
A high performance distributed cryptanalysis framework

API Documentation

Version 1.0

October 30 2012

Written by
Vikas N Kumar

Introduction

Wisecracker is a distributed high performance computing framework that enables users to perform cryptanalysis easily by distributing their computations across a cluster of multiple multi-core processors and graphical processing units (GPUs). The high level design information about Wisecracker can be obtained by reading the technical white paper. This document focuses on the API of the framework itself and outlines how to use the framework. It also describes some details about the source code structure. This document is for Wisecracker version 1.0. We recommend that the reader read the technical white paper before reading this document.

The framework provides both a C and C++ API. In this document we shall focus on the C API. The C++ API is similar to the C API and we shall outline briefly how to use the C++ API in later sections.

Installation and Pre-requisites

The user can download the source code for Wisecracker from https://github.com/vikasnkumar/wisecracker using the program `git` or by using the download link provided on `github.com` or on our website `selectiveintellect.com`. Once downloaded, the user should follow instructions given in the file called `INSTALL` for setting up their system to compile and run Wisecracker and its sample applications. These instructions will always be up-to-date regarding installation of pre-requisites, and directions on compiling Wisecracker.

The user will need to install OpenCL libraries and header files for his system configurations. This can be downloaded from Intel, AMD and/or NVIDIA's websites, respectively. Apple's version of OpenCL is provided with Mac OSX Snow Leopard or higher. The user will need to install an implementation of MPI such as OpenMPI or MPICH if they want to run Wisecracker across multiple systems.

Understanding The Directory Structure

Below are the list of important directories that are part of the source code:

- `include/` - This directory holds all the header files that are needed for compiling Wisecracker and also for compiling any applications written using Wisecracker. The main header file in here is `wisecracker.h` which includes all the other header files present in the `include/wisecracker/` directory as needed.
- `src/` - This directory holds all the source code that compiles into the library for Wisecracker which can then be linked by applications using Wisecracker.
- `apps/` - This directory holds sample applications that are provided with Wisecracker as examples and also as those that can be used out-of-the-box by users.
- `scripts/` - This directory contains helper scripts that can be used by the user to install pre-requisites or perform other tasks. Currently in version 1.0 only one script exists in this directory called `setup_amazonaws.sh` which will setup an Amazon EC2 GPU virtual machine to be

ready to run and install Wisecracker.
- `deps/` - This directory holds custom `cmake` files that are dependency oriented.
- `docs/` - This directory holds any documents that might be useful to the user.

Using the Framework

The user has to only include the `wisecracker.h` header file for writing an application as it internally includes multiple header files and also includes the C++ API header file which will get included if being compiled by a C++ compiler.

The user can pick either the static library or the shared library (DLL) to link with depending on their choice. The static library is called `libwisecracker_s.a` on Linux and Mac OSX and `wisecracker_s.lib` on Windows. The shared library is called `libwisecracker.so` on Linux, `libwisecracker.dylib` on Mac OSX and `wisecracker.dll` on Windows.

If the user chooses to link with the static library they have to define the macro `WC_LINKING_STATIC` before including the header file or as a compiler option. If the user chooses to link with the shared library or DLL they do not need to define the afore-mentioned macro.

The C API

After the user has included the header file `wisecracker.h`, they can initialize the *executor*, setup the callbacks for the *executor* to invoke and run the *executor*. When the user is done using the *executor* they can destroy the *executor* object. In this section we discuss every function, callback function and useful macros needed for a user to use Wisecracker correctly.

Main *Executor* Functions
The *executor* object is an opaque object always returned and used as a pointer to `wc_exec_t` by all the *executor* functions. These functions are declared in the `include/wisecracker/executor.h` header file.
- `wc_exec_t *wc_executor_init(int *argc, char ***argv)`
  - If and only if the user is **not** using MPI, these arguments may be `NULL`. If the user is using MPI, these arguments necessarily *have to be* pointer to the `argc` and `argv` parameters passed to the `main()` function in the program. MPI needs these arguments for initializing itself. It is recommended that the user not use `NULL` as parameters.
  - This function initializes MPI and other internal details of the *executor* object.
  - Only after this function has been successfully initialized can any of the *executor* property functions (described later) will return valid values.
  - The function returns a valid pointer on success and `NULL` on failure.
  - This function should be called only **once** if using MPI.
  - This pointer object is not reference counted and will need to be used safely across all the other functions. We do **not** recommend using this pointer object simultaneously from

  separate threads.
  - This function is re-entrant.

- `void wc_executor_destroy(wc_exec_t *wc)`
  - This function destroys the *executor* object and shuts down the usage of OpenCL and MPI.
  - If using MPI, once this function has been called, the user cannot call the initialize function again to recreate an *executor*.
  - If the user wants to reuse an *executor* they can do so with the same *executor* object.
  - This function returns no values and will cleanup the *executor's* allocated memory.

-
  ```
  wc_err_t wc_executor_setup(wc_exec_t *wc,
                             const wc_exec_callbacks_t *cb)
  ```
  - The user should call this function and provide a pointer to a `wc_exec_callbacks_t` object which will hold pointers to all the callback functions that will be invoked by the *executor*. This callback datatype is described later in this section.
  - This function will return an error of type `wc_err_t` which can have any of the values given in the `executor.h` header file. Success is denoted by the `WC_EXE_OK` value.
  - The user can call this function multiple times to change the values of the callbacks before each run. This way the user can create different callback objects for different problems and use the same *executor* object for execution.

- `wc_err_t wc_executor_run(wc_exec_t *wc)`
  - The user should call this function with the `wc_exec_t` object that has already been setup by the `wc_executor_setup()` function.
  - This function can be called multiple times by the user to execute multiple runs.
  - The function returns the value `WC_EXE_OK` on success and any of the descriptive error values given in the `executor.h` on failure.
  - If during a run an MPI error occurs the function might abort the run depending on the severity of the MPI related error. It tries its best to return cleanly without aborting the run.

Executor Property Functions

These functions are declared in the `include/wisecracker/executor.h` header file.

- `int wc_executor_num_systems(const wc_exec_t *wc)`
  - This function returns the number of systems that the *executor* supports which can be 1 or more if using MPI.
  - If not using MPI this function will return the value 1.
  - If the parameter is `NULL` or there has been an error, the function will return -1.

- `int wc_executor_system_id(const wc_exec_t *wc)`
  - This function returns the ID for the running *executor* which is used by MPI internally.
  - If the user is not using MPI, this function will return the value 0.
  - The master *executor* will always have the value 0.

    - The slave *executors* will have the values greater than 0.
    - On error, this function will return -1.

- `uint64_t wc_executor_num_tasks(const wc_exec_t *wc)`
    - This function returns the number of tasks that the current run of the *executor* will be processing.
    - On error it will return 0 or if called before the `wc_executor_run()` function has been called will also return 0.
    - This is useful if the user wants to know how many total tasks will be executing and the recommended place to call this function would be in the callbacks.

- `uint32_t wc_executor_num_devices(const wc_exec_t *wc)`
    - This function returns the number of OpenCL devices that exist on the current system.
    - It might return 0 if called before the `wc_executor_setup()` function.
    - On error it will return 0.
    - It might be useful to call this function in some of the callbacks if necessary.

- `void wc_executor_dump(const wc_exec_t *wc)`
    - This function is a generic function that will print the description of the contents of the `wc_exec_t` object to `stderr`.

Executor Callbacks

The definitions of the callbacks are part of a structure `wc_exec_callbacks_t` which is declared in the `include/wisecracker/executor.h` header file. We shall describe each one of the members of the structure here. Some callbacks get exclusively called on the master *executor* and some exclusively on the slave *executor.* If the application has been built without MPI or is being used in single system mode, all the slave *executor* calls will be done on the master itself. Most callbacks are optional but some are required and are noted accordingly.

- `void *user` – This is a pointer to an opaque object that the user can pass to the *executor*. The *executor* will pass this object to the user in every callback so that the user can share information between callbacks if needed.

- `uint32_t max_devices` – This is the maximum number of OpenCL devices per system that the user wants to use. Some systems might have many devices and the user might want just a few depending on the problem. If set to 0, the *executor's* OpenCL runtime will use all available devices that match the device type. If using MPI, once this is set on the master all the slaves will be using the same value.

- `wc_devtype_t device_type` – This denotes the type of device to be used by the *executor's* OpenCL runtime. If using MPI, once this is set on the master all the slaves will be using the same value. It can take three values – `WC_DEVTYPE_CPU`, `WC_DEVTYPE_GPU` and `WC_DEVTYPE_ANY`. The default value is `WC_DEVTYPE_ANY`.

- `wc_err_t (*on_start)(const wc_exec_t *wc, void *user)`
  - This function callback gets called by ***both*** the master and slave *executors* at the start of the *executor* run if it has been set.
  - The callback can return `WC_EXE_OK` on success and any of the other `wc_err_t` values present in `executor.h` for failure.
  - If this callback returns failure, the `wc_executor_run()` function will return immediately.
  - The user can use this function for initialization of their user data structures if needed.

- `wc_err_t (*on_finish)(const wc_exec_t *wc, void *user)`
  - This function callback gets called by ***both*** the master and slave *executors* at the end of the *executor* run if it has been set.
  - The callback can return `WC_EXE_OK` on success and any of the other `wc_err_t` values present in `executor.h` for failure.
  - If this callback returns failure, the `wc_executor_run()` function will return immediately.
  - The user can use this function for initialization of their user data structures if needed.

-
  ```
  char *(*get_code)(const wc_exec_t *wc, void *user,
                    size_t *codelen)
  ```
  - This function callback gets called on ***both*** the master and slave *executors* for retrieving the OpenCL code.
  - The user ***must*** return the code string as a buffer allocated using any of the `WC_MALLOC`, `WC_CALLOC` or `WC_REALLOC` macros. The buffer need not be `NULL` terminated.
  - The user also ***must*** return the length of the buffer in the `codelen` argument.
  - If the user returns a `NULL`, the *executor* will stop the execution as this will be considered an error.
  - The reason this is needed since every system might have different OpenCL devices, and the *executor* will compile the code for each device on each system.
  - This callback is **required**.

- `char *(*get_build_options)(const wc_exec_t *wc, void *user)`
  - This function callback gets called on ***both*** the master and slave *executors* for retrieving the OpenCL code compile options if any. The user might need to define some compile time macros or include file directories for their code.
  - The user ***must*** return the compile option string as a `NULL` terminated buffer allocated using any of the `WC_MALLOC`, `WC_CALLOC` or `WC_REALLOC` macros.

-
  ```
  void (*on_code_compile)(const wc_exec_t *wc, void *user,
                          uint8_t success)
  ```
  - This function callback gets called on ***both*** the master and slave *executors* on compilation of

the OpenCL code. It informs the user if the compilation was successful or not.
  - The value of the `success` variable can be 0 or 1.
  - The user might be able to use this in event-based applications that might be using Wisecracker.
  - Very complex OpenCL code might take many seconds to compile, and hence having this event might be useful to the user.

- `uint64_t (*get_num_tasks)(const wc_exec_t *wc, void *user)`
  - This function callback gets called by the master *executor* to query how many total tasks the user wants to distribute across the multiple systems and devices.
  - If the user can return 0 on error.
  - If MPI is being used, the master will communicate this value across to the slaves for the `wc_executor_num_tasks()` property function to return the correct value.
  - This callback is **required**.

- `uint32_t (*get_task_range_multiplier)(const wc_exec_t *wc,`
  `void *user)`
  - If set, this function callback gets called by the master *executor* to query a possible task range multiplier for the task distribution across OpenCL devices.
  - The user can return 1 if they do not know what to return as a value. If not set, the value is assumed to be 1.
  - An example of usefulness of this multiplier is when the devices are GPUs and the number of tasks run into *billions* of tasks. Then the user can use the multiplier to scale the ranges distributed to different OpenCL devices simultaneously to speed up computation across GPUs. This can be seen in the code for the `wisecrackmd5` sample application in the `apps/crackmd5.c` file. This also helps reduce the communications for results for every range between the slaves and the master *executors*.
  - The user can pick an appropriate value by trying out different combinations and seeing which value gives the best return in terms of speed and memory usage.
  - If using MPI, the master *executor* will send this value to the slave *executor.*

- `wc_err_t (*get_global_data)(const wc_exec_t *wc, void *user,`
  `wc_data_t *out)`
  - This function callback is called by the master *executor* to retrieve a data buffer that will need to be communicated to the slaves for sharing possible common data structures.
  - This is an optional callback. This gets called in both the single system mode and multiple system modes.
  - The user might choose to allocate any buffers returned using any of the `WC_MALLOC`, `WC_CALLOC` or `WC_REALLOC` macros.
  - If the user chooses not to do so, they ***must*** set the `free_global_data` callback function (described later) so that the user can free the memory using their own methods.
  - The user should return `WC_EXE_OK` on success and an appropriate value on error.

- `wc_err_t (*on_receive_global_data)(const wc_exec_t *wc, void *user, const wc_data_t *gdata)`
  - This function callback is called on the slave *executors* once they receive the global data buffers from the master when Wisecracker is being used with MPI for multiple systems.
  - In single system mode, this function does **not** get called.
  - The user can use this callback to inflate the buffers into custom data structures for possible use by later callbacks and other functions.
  - The user should return `WC_EXE_OK` on success and an appropriate value on error.

- `wc_err_t (*on_device_start)(const wc_exec_t *wc, wc_cldev_t *dev, uint32_t devindex, void *user, const wc_data_t *gdata)`
  - This function callback is called on the slave *executor* in the multiple system run and on the master *executor* on the single system run on the start of the *executor* run for the device.
  - This gets called once per OpenCL device before the processing of the task ranges begins.
  - The argument `dev` is an object of datatype `wc_cldev_t` which is defined in the `executor.h` header file.
  - This object can be used by the user in this callback to create OpenCL kernels and memory buffers for the device. The *executor* will already have created OpenCL command queues and contexts, so the user does not have to. It will also provide access to the pre-compiled OpenCL program object for creating the kernels.
  - The `devindex` argument points to the index to the array of OpenCL devices in the current *executor* system.
  - The global data is also passed as an argument to this function in `gdata` in case the user wants to use it.
  - This callback is optional.
  - The user should return `WC_EXE_OK` on success and an appropriate value on error.

- `wc_err_t (*on_device_finish)(const wc_exec_t *wc, wc_cldev_t *dev, uint32_t devindex, void *user, const wc_data_t *gdata)`
  - This function callback is called on the slave *executor* in the multiple system run and on the master *executor* on the single system run on the start of the *executor* run for the device.
  - This gets called once per OpenCL device after all the task ranges provided to the system have been processed.
  - This callback function can be used to free memory by releasing OpenCL memory buffers and kernel objects. The command queues and contexts that are created by the *executor* should **not** be released here.
  - The arguments `dev`, `gdata` and `devindex` have the same meanings as defined for the `on_device_start` callback.
  - The user should return `WC_EXE_OK` on success and an appropriate value on error.

- `wc_err_t (*on_device_range_exec)(const wc_exec_t *wc,`
  ```
              wc_cldev_t *dev, uint32_t devindex, void *user,
              const wc_data_t *gdata, uint64_t start, uint64_t end,
              cl_event *out_event)
  ```
  - This function callback is called on the slave *executor* in the multiple system run and on the master *executor* on the single system run on the start of the *executor* run for the device.
  - This callback is **required**.
  - This callback gets invoked per device with a range of tasks to complete where `start` and `end` are the indexes of each range. The range is inclusive of the `start` value but not of the `end` value, i.e. it is `[start, end)`.
  - For a more efficient run, the user should enqueue all the OpenCL kernels and read-write tasks, and return a single OpenCL event in the `out_event` pointer that the *executor* can wait on per device per task range.
  - The arguments `dev, gdata` and `devindex` have the same meanings as defined for the `on_device_start` callback.
  - The user should return `WC_EXE_OK` on success and an appropriate value on error.

- `wc_err_t (*on_device_range_done)(const wc_exec_t *wc,`
  ```
                     wc_cldev_t *dev, uint32_t devindex,
                     void *user, const wc_data_t *gdata,
                     uint64_t start, uint64_t end,
                     wc_data_t *results)
  ```
  - This function callback is called on the slave *executor* in the multiple system run and on the master *executor* on the single system run on the start of the *executor* run for the device.
  - This callback is **required** only if the slave wants to send results back to the master.
  - This callback gets invoked per device when a range of tasks completes processing where `start` and `end` are the indexes of each range. The range is inclusive of the `start` value but not of the `end` value, i.e. it is `[start, end)`.
  - The user should write the processing code in this callback.
  - The arguments `dev, gdata` and `devindex` have the same meanings as defined for the `on_device_start` callback.
  - The user should return `WC_EXE_OK` on success and an appropriate value on error.
  - If the user wants to stop processing because results might be acceptable, such as in the case of successful reversing of an MD5 cryptographic checksum, the user can return `WC_EXE_STOP`. This will be sent to the master and it will instruct all the slaves to immediately stop processing.
  - The user should pack the results for the given range into a buffer that ***must*** be allocated using the `WC_MALLOC`, `WC_CALLOC` or `WC_REALLOC` macros. This is because memory management of these buffers is done by the *executor* which uses these macros internally.
  - The slave sends the results and the error return value for the given range back to the master.

- `wc_err_t (*on_receive_range_results)(const wc_exec_t *wc,`
  `void *user, uint64_t start, uint64_t end,`
  `wc_err_t slverr, const wc_data_t *results)`
    - This function callback is called on the master *executor* in both the multiple and single system runs.
    - Each set of results and the error return code from the `on_device_range_done` callback function is returned back in this callback to be unpacked into custom data structures if necessary or for further processing.
    - The slave error can be checked for errors or stop indications and the user can appropriately return either `WC_EXE_OK` or any other error value.
    - The user should return `WC_EXE_OK` on success and an appropriate value on error.
    - If the user wants to stop processing because results might be acceptable, such as in the case of successful reversing of an MD5 cryptographic checksum, the user can return `WC_EXE_STOP`. The master will then instruct all the slaves to stop processing.

- `void (*free_global_data)(const wc_exec_t *wc, void *user,`
  `wc_data_t *gdata)`
    - This function callback is called on all the *executors* for freeing the memory that was allocated in `get_global_data` and `on_receive_global_data`.
    - If this function callback is not set, the *executor* uses the `WC_FREE` macro to free the memory.

- `void (*progress)(float percent, void *user)`
    - This function callback gets called periodically by the master *executor* to notify of progress that has taken place in processing of all the tasks.
    - The user can use this to update any user interface with details about how much computation has completed.

Utility Functions

These functions are declared in the `include/wisecracker/utils.h` header file.
They are helpful for the user to perform certain tasks which are system oriented and might need cross-platform implementations.

- `int wc_util_timeofday(struct timeval *tv)`
    - This is a cross-platform function for retrieving the time of day in epoch time for Windows, Linux and Mac OSX.
    - It returns -1 on error and 0 on success.
    - It will fill the `struct timeval` parameter with the values of the time of day in seconds and microseconds.

- `char *wc_util_strdup(const char *str)`
    - This is an implementation of the `strdup()` function that is cross-platform and checks for allocation errors as well.

    - It will return `NULL` on error or if the string length was 0.
    - To free the memory returned by this function the user **should** use the macro `WC_FREE`. The `WC_FREE` macro is explained later.

-
  ```
  int wc_util_glob_file(const char *filename,
                        unsigned char **outdata, size_t *outlen)
  ```
    - This function takes a filename and reads the contents of the file into an `unsigned char` buffer and returns the length of the buffer.
    - The allocation of the buffer is done by the function itself.
    - The function returns 0 on success and -1 on error.
    - The user should free the returned buffer using `WC_FREE`.

- `const char *wc_util_license()`
    - This function returns a constant pointer to a character buffer to the license for the framework.
    - This function should be used if the user is adding applications using the licensed framework .

- `size_t wc_util_charset_size(wc_util_charset_t chs)`
    - There are different character sets supported internally by Wisecracker which can be found defined in the `utils.h` header file. This function returns the number of characters in the given character set.
    - It will return 0 on error.

- `wc_util_charset_t wc_util_charset_fromstring(const char *str)`
    - This function takes a string and returns the corresponding character set enumeration value.
    - On error it will return the default character set value and will print a warning message on `stderr.`

- `const char *wc_util_charset_tostring(wc_util_charset_t chs)`
    - This function is for printing the string form of the enumeration values of the character set.

Useful cross-platform Macros

These macros are declared in the `include/wisecracker/macros.h` header file. Some of them are cross-platform for portability. Below are the most useful ones:

- Memory Allocation
    - `WC_MALLOC` – This macro is used all over the source code for allocating memory blocks. It uses the `malloc()` function on Linux and Mac OSX and the `HeapAlloc()` function on Windows. It is recommended that this always be used in tandem with `WC_FREE`. Any memory blocks that are exchanged between *executors* and callbacks **must** be allocated using the `WC_MALLOC`, `WC_CALLOC` or `WC_REALLOC` macros and freed with `WC_FREE`.
    - `WC_CALLOC` – This macro is just a cross-platform wrapper around the `calloc()` and

    `HeapAlloc()` functions for Linux and Windows, respectively.
  - `WC_REALLOC` – This macro is also a cross-platform wrapper around the `realloc()` and `HeapReAlloc()` functions for Linux and Windows, respectively.
  - `WC_FREE` – This macros is a wrapper for the `free()` and `HeapFree()` functions for Linux and Windows, respectively. This should always be used to free any memory returned by the *executor.* It also sets the input pointer value to `NULL` after freeing the memory.
- Logging
  - `WC_SET_LOG_LEVEL` – This macro sets the log level to any of the options present in the `WC_LOGLEVEL_*` enumerations present in the `macros.h` header file. The default level is `WC_LOGLEVEL_DEBUG`.
  - `WC_ERROR` – This is a `printf()` style macro which takes in variable arguments and prints to `stderr`. It also prints a prefix string of `"ERROR:"` followed by the function name and line number. It should be used for error logging. It checks for the log level before printing.
  - `WC_WARN` – This is a `printf()` style macro which takes in variable arguments and prints to `stderr`. It also prints a prefix string of `"WARN:"` followed by the function name and line number. It should be used for logging warnings. It checks for the log level before printing.
  - `WC_INFO` – This is a `printf()` style macro which takes in variable arguments and prints to `stderr`. It also prints a prefix string of `"INFO:"` followed by the function name and line number. It should be used for logging informative statements. It checks for the log level before printing.
  - `WC_DEBUG` – This is a `printf()` style macro which takes in variable arguments and prints to `stderr`. It also prints a prefix string of `"DEBUG:"` followed by the function name and line number. It should be used for logging very informative statements that might be helpful in debugging the software. It checks for the log level before printing.
  - `WC_NULL` – This is a `printf()` style macro which takes in variable arguments and prints to `stderr`. It prints no prefixes and is just a wrapper around the `fprintf()` function.
  - `WC_ERROR_OUTOFMEMORY` – This takes in an allocation size value that failed to allocate and prints a default error statement citing an out of memory violation for that size.
  - `WC_ERROR_OPENCL` – This takes in the name of an OpenCL function and the error code returned when the OpenCL function was invoked and prints an error statement citing the OpenCL error that occurred for that function.
- String comparisons
  - `WC_STRCMPI` – This is a cross-platform wrapper around the case-insensitive string comparison between two strings. It wraps the `strcasecmp()` function on Linux and Mac OSX, and `stricmp()` function on Windows.
  - `WC_STRNCMPI` – This is a cross-platform wrapper around the case-insensitive string comparison of a given length between two strings. It wraps the `strncasecmp()` function on Linux and Mac OSX, and `strincmp()` function on Windows.
- Thread, Thread Signal and Locks – Various macros for threads, condition variables (or signals) and locks have been provided in the `macros.h` header file. They can be used by the user if they like or need cross-platform thread functions but it is not necessary. The user can read the

macros in detail and refer to the corresponding function documentation. We shall not be explaining each macro here.

We now move on to explaining the classes and member functions of the C++ API, which is very similar to the C API in concept and design as it is just a C++ wrapper around the C API.

The C++ API

The C++ wrapper classes for the *executor* and the callback structure can be found in the `include/wisecracker/cppwrapper.h` file which automatically gets included in the `include/wisecracker.h` file if the compiler is C++. The static and dynamic libraries for Wisecracker already have the C++ wrapper compiled in, so the user will not have to link in any special C++ libraries.

We describe each class and member function of the class in the C++ API here by referring to the corresponding C API documentation in this document as appropriate.

All the C++ classes are in the `wc` namespace.

The `Executor` Class

The `Executor` class wraps all the `wc_executor_*()` functions. The member functions are as follows:

- `Executor(int *argc, char ***argv)` – This constructor is the only constructor for the class and should be invoked in the `main()` function of the program. It has the same properties as the `wc_executor_init()` function. On error it throws a `std::runtime_error` exception if MPI failed to initialize or there was a memory allocation failure.
- `~Executor()` – This destructor wraps the `wc_executor_destroy()` function and works in the same way as described for the C API.
- `wc_err_t setup(CallbackInterface *cb)` – This member function is representative of the `wc_executor_setup()` function and takes a pointer to a derived object of the `CallbackInterface` abstract base class which represents the callbacks that the *executor* will invoke. This object should not be freed by the user until the *executor* run has completed.
- `wc_err_t run()` – This member function is a wrapper around the `wc_executor_run()` function and invokes all the callbacks that have been setup earlier using the `setup()` member function.
- `int num_systems()` – This function is a wrapper around the `wc_executor_num_systems()` C API property function.
- `int num_tasks()` – This function is a wrapper around the `wc_executor_num_tasks()` C API property function.

- `int my_id()` – This function is a wrapper around the `wc_executor_system_id()` C API property function.
- `int num_system_devices()` – This function is a wrapper around the `wc_executor_num_devices()` C API property function.
- `bool is_master()` – This function returns whether the *executor* is a master or not.
- `void dump()` – This function is a wrapper around the `wc_executor_dump()` C API property function.

The `CallbackInterface` Class

The `CallbackInterface` is a base class from which the user should derive another class which will be actually used by the *executor*. The beauty of this derived class method is that the user can add their own custom member functions and data members in addition to those required by `CallbackInterface`. The constructors and destructors of `CallbackInterface` are obvious, but the member functions are described below. All the optional member functions are *virtual* and the required member functions are *pure virtual*.

- `void set(uint32_t max_devices, wc_devtype_t devtype)` – This member function can be optionally used by the user to set the maximum OpenCL devices and the device type they want to use.
- `const Executor *get_executor()` – This member function can be called from any callback and will return a constant pointer to the `Executor` object which was used for setting up the `CallbackInterface`. It will return a valid value only if the setup has been done.
- `uint32_t max_devices()` – This member function returns the current value of the maximum devices that the user has selected.
- `wc_devtype_t device_type()` – This member function returns the selected value of the device type that the user has selected.
- `wc_err_t on_start()` – This is the same as the `on_start` C API callback.
- `wc_err_t on_finish()` – This is the same as the `on_finish` C API callback.
- `wc_err_t get_code(std::string &code)` – This is the same as the `get_code` C API callback except for the fact that the user can return the code in a `std::string` object. This is a pure virtual function and the user has to override it.
- `wc_err_t get_build_options(std::string &options)` – This is the same as the `get_build_options` C API callback except for the fact that the user can return the code in a `std::string` object.
- `void on_code_compile(bool success)` – This is the same as the `on_code_compile` C API callback with the `success` parameter taking the value `true` or `false`.
- `uint64_t get_num_tasks()` – This is the same as the `get_num_tasks` C API callback. It is a pure virtual function and the user has to override it.
- `uint32_t get_task_multiplier()` – This is the same as the `get_task_range_multiplier` C API callback.
- `wc_err_t get_global_data(wc_data_t &gdata)` – This is the same as the

`get_global_data` C API callback except that the output variable is a reference instead of a pointer.

- `void free_global_data(wc_data_t &gdata)`: This function is the same as the `free_global_data` C API callback except that this is pure virtual. In C++ it is necessary for the user to manage the memory correctly. We still recommend using the `WC_MALLOC` style macros for exchanging allocated buffers between the callbacks and the *executors* to avoid random memory corruption bugs.
- `wc_err_t on_receive_global_data(wc_data_t &gdata)` : This function is the same as the `on_receive_global_data` C API callback.
- `wc_err_t on_device_start(wc_cldev_t &dev, uint32_t devindex, const wc_data_t &gdata)` – This function is the same as the `on_device_start` C API callback.
- `wc_err_t on_device_finish(wc_cldev_t &dev, uint32_t devindex, const wc_data_t &gdata)` – This function is the same as the `on_device_finish` C API callback.
- `wc_err_t on_device_range_exec(wc_cldev_t &dev, uint32_t devindex, const wc_data_t &gdata, Range &range, cl_event *outevent)` – This function is the same as the `on_device_range_exec` C API callback. The `Range` data structure is a member data structure of the `CallbackInterface` class and holds the `start` and `end` of the range.
- `wc_err_t on_device_range_done(wc_cldev_t &dev, uint32_t devindex, const wc_data_t &gdata, Range &range, wc_data_t &results)` – This function is the same as the `on_device_range_done` C API callback. The `Range` data structure is a member data structure of the `CallbackInterface` class and holds the `start` and `end` of the range. Results **have** to be allocated using `WC_MALLOC` style macros.
- `wc_err_t on_receive_range_results(Range &range, wc_err_t range_error, const wc_data_t &results)` – This function is the same as the `on_receive_range_results` C API callback. The `Range` data structure is a member data structure of the `CallbackInterface` class and holds the `start` and `end` of the range. The `range_error` parameter is the error sent back to the master which has been the return value of the `on_device_range_done()` function for that range.
- `void progress(float percent)` – This function is the same as the `progress` C API callback.

Conclusion

The C API is the main API for Wisecracker but users can also use the C++ API to write their applications. There is hardly any overhead added by the C++ API as the user can read the source code and see that the C++ wrapper has been kept as tight as possible to the C API.

The source code of wisecracker can be downloaded from https://github.com/vikasnkumar/wisecracker. Professional level technical support can be obtained by contacting the developers at Selective Intellect

LLC by emailing them at wisecracker@selectiveintellect.com.