HDF5-UDF

Release 2.1

Lucas C. Villa Real

Mar 20, 2023

CONTENTS:

1 Installation **3**

2 Getting started **5**

3 The lib interface for UDF writers **7**

4 UDF compilation and storage interfaces **11**
- 4.1 Command-line interface 11
- 4.2 Python interface 12
- 4.3 JSON schema for HDF5-UDF datasets 14

5 Security considerations **15**
- 5.1 Sandboxing 15
- 5.2 Trust profiles 15
- 5.3 Signing UDFs 16
- 5.4 Associating UDFs with a trust profile 16

6 Configuration **17**
- 6.1 Allowing system calls 17
- 6.2 Access to files and directories 18

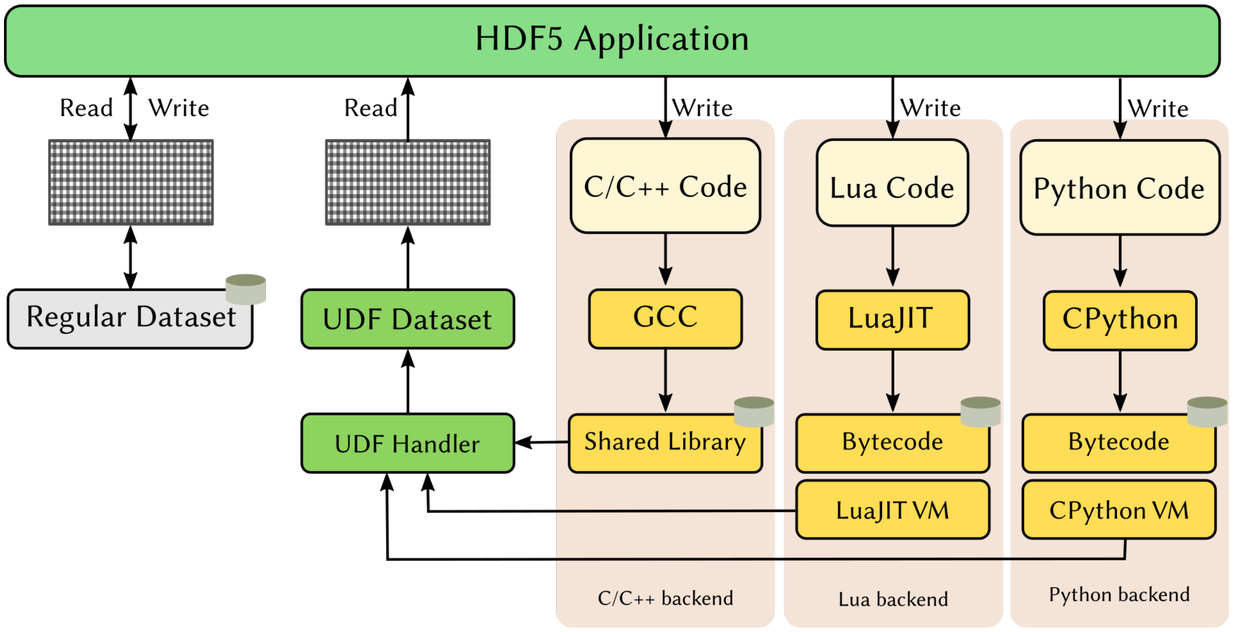
7 Indices and tables **19**

Python Module Index **21**

Index **23**

HDF5-UDF is a mechanism to generate HDF5 dataset values on-the-fly using user-defined functions (UDFs). The platform supports UDFs written in Python, C++, and Lua under Linux, macOS, and Windows.

This page provides documentation for UDF writers and for those wishing to programmatically compile and store UDFs through HDF5-UDF's C and Python APIs.



INSTALLATION

Please refer to the project page on GitHub for up-to-date instructions on how to install the HDF5-UDF library and its utilities from binary packages or in source code form.

Once the main library has been installed on your system, PIP installs the Python APIs needed to programmatically compile and attach UDFs to HDF5 files:

```
pip install pyhdf5-udf
```

CHAPTER

TWO

GETTING STARTED

We begin by writing the function that we will compile and store on HDF5. In the example below, the dataset will be named “simple”. Note the calls to `lib.getData()` and `lib.getDims()`: they are part of HDF5-UDF runtime and provide direct access to the dataset buffer and its dimensions, respectively. See *The lib interface for UDF writers* for details on that interface. The entry point of the UDF is always a function named `dynamic_dataset`.

```
from hdf5_udf import UserDefinedFunction

def dynamic_dataset():
   from math import sin
   data = lib.getData('simple')
   dims = lib.getDims('simple')
   for i in range(dims[0]):
      data[i] = sin(i)
```

Now, we use the `inspect` module to capture the function’s source code and save it as a file on disk. HDF5-UDF recognizes files ending on `.py`, `.cpp`, and `.lua`.

```
import inspect
with open("/tmp/udf.py", "w") as f:
   f.write(inspect.getsource(dynamic_dataset))
```

Last, we declare a `UserDefinedFunction` object and describe the dataset: its name, type, and dimensions. Next, we compile it into a bytecode form and store it in the provided HDF5 file.

```
with UserDefinedFunction(hdf5_file='/path/to/file.h5', udf_file='/tmp/udf.py') as udf:
   udf.push_dataset({'name': 'simple', 'datatype': 'float', 'resolution': [1000]})
   udf.compile()
   udf.store()
```

At this point the dataset has been created and it’s possible to retrieve its data. Note that the call to `f['simple'][:]` triggers the execution of the bytecode we just compiled. There’s more to the picture than meets the eye!

```
import h5py
f = h5py.File('/path/to/file.h5')
simple = f['simple][:]
...
f.close()
```

CHAPTER

THREE

THE LIB INTERFACE FOR UDF WRITERS

HDF5-UDF comes with a few primary interfaces and helper methods to ease storing and retrieving values of string-based datasets. It is not necessary to explicitly instantiate a `PythonLib` object: you can simply use the `lib` one provided by HDF5-UDF as seen in the examples below.

`class udf_template.`**`PythonLib`**

Interfaces between User-Defined Functions and the HDF5 API.

`getFilePath``()`

Retrieve the path to the input HDF5 file.

`string``(`*structure*`)`

Retrieve the value of a HDF5 string datatype.

Parameters

`structure` (*`obj`*) – Object holding the string element to be retrieved

Examples

Print the i-th member of a string datatype

```
print(lib.string(item[i], flush=True))
```

Returns

The string element (decoded from UTF-8)

Return type

str

`setString``(`*structure*, *s*`)`

Set the value of a HDF5 string datatype.

Write the given string to the provided object. No encoding assumptions are made; the application is expected to encode() the given string. This function does boundary checks to prevent buffer overflows.

Parameters

- **`structure`** (*`obj`*) – String object whose value is to be set
- **`s`** (*`str`*) – String value to write to the given object

Examples

Write to the i-th member of a string datatype

```
lib.setString(item[i], "Too much monkey business".encode("utf-8"))
```

Write a string to a compound member 'album'

```
lib.setString(item[i].album, "Electric Ladyland".encode("utf-8"))
```

`getData`(*name*)

Get a data pointer to the given dataset.

This method fetches the given dataset name from the HDF5 file and loads it into memory. If the given name refers to a dataset that the UDF uses as input, the returned object holds the contents (values) of that dataset. Otherwise, if the given name is the output dataset being programmatically generated by the function, then the returned object is theh actual buffer which the user-defined function needs to populate.

Parameters
`name` (*str*) – Dataset name

Examples

Retrieve pointers to the input and output dataset, then copy the first 100 elements of the input dataset to the output. Note that start and end indexes must always be provided when reading or writing data to data retrieved by this API.

```
input_ds = lib.getData("input_dataset")
output_ds = lib.getData("output_dataset")
input_ds[0:100] = output_ds[0:100]
```

Returns
Data pointer to the requested dataset

Return type
obj

`getType`(*name*)

Get the data type of the given dataset.

Parameters
`name` (*str*) – Dataset name

Returns
The corresponding data type name. The following is a list of possible values returned by this function: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64", "float", "double", "compound", and "string".

Return type
str

`getDims`(*name*)

Get the dimensions of the given dataset.

Parameters
`name` (*str*) – Dataset name

Returns
A list of integers containing the size of each dimension of the given dataset.

Return type
list

CHAPTER

FOUR

UDF COMPILATION AND STORAGE INTERFACES

User-defined functions can be compiled and stored in HDF5 from the command-line or from HDF5-UDF's APIs. To date, the project provides a low-level C API and a Python abstraction of that API, documented below.

4.1 Command-line interface

Given an UDF file with a supported backend (C++, Python, or Lua) and a target HDF5 file, the `hdf5-udf` utility can be used to compile and store the UDF on that file.

```
Syntax: hdf5-udf [flags] <hdf5_file> <udf_file> [udf_dataset..]

Flags:
  --overwrite               Overwrite existing UDF dataset(s)
  --save-sourcecode         Include source code as metadata of the UDF
```

When populating values for a dataset with a native data type, the following syntax applies to describe that dataset:

```
name:resolution:type
```

where:

- name: name of the UDF dataset
- resolution: dataset resolution, with dimensions separated by the `x` character. Examples: `XSize`, `XSize``x``YSize`, `XSize``x``YSize``x``ZSize`
- type: `[u]int8`, `[u]int16`, `[u]int32`, `[u]int64`, `float`, `double`, `string`, or `string(NN)`. If unset, strings have a fixed size of 32 characters.

When populating values for a dataset with a compound data type, the following syntax is used instead:

```
name:{member:type[,member:type...]}:resolution
```

4.2 Python interface

It is also possible to bypass the command line and embed the instructions to compile and store the UDF in a HDF5 file using HDF5-UDF's Python API. This API is an abstraction of the low-level C API.

class hdf5_udf.UserDefinedFunction(*hdf5_file=''*, *udf_file=''*)

Store a user-defined function on a HDF5 file.

Parameters

- **hdf5_file** (*str*) – Path to existing HDF5 file (required)
- **udf_file** (*str*) – Path to file implementing the user-defined function (optional)

set_option(*option=''*, *value=''*)

Set an option given by a key/value pair.

Parameters

- **option** (*str*) – Name of the option to configure. Recognized option names include:
  - "overwrite": Overwrite existing UDF dataset? (default: False)
  - "save_sourcecode": Save the source code as metadata? (default: False)
- **value** (*str, bool*) – Value to set *option* to.

Raises
TypeError – If the given data type is not recognized

Returns
True if successful, False otherwise.

Return type
bool

push_dataset(*description*)

Define a new UserDefinedFunction dataset.

Parameters
description (*dict*) – Describe the dataset: its name, data type, size, and members (if a compound data type). For native datasets the following keys are expected: `name`, `datatype`, `resolution`.

Compound datasets must provide an extra `members` key. Objects of the `members` array must include two properties: `name` and `datatype`.

Examples

Dataset with a native data type:

```
{"name": "MyDataset", "datatype": "int32", "resolution": [100,100]}
```

Dataset with a compound data type: .. code-block:

```
{
    "name": "MyCompoundDataset",
    "datatype": "compound",
    "resolution": 100,
```

(continues on next page)

(continued from previous page)

```
    "members": [
        {"name": "Identifier", "datatype": "int64"},
        {"name": "Description", "datatype": "string(80)"}
    ]
}
```

Raises

- **`TypeError`** – If *description* or its members hold an unexpected data type
- **`ValueError`** – If description dictionary misses mandatory keys

Returns
True if successful, False otherwise.

Return type
bool

`compile()`

Compile the UserDefinedFunction into bytecode form.

Returns
True if successful, False otherwise

Return type
bool

`store()`

Store the compiled bytecode on the HDF5 file.

Returns
A dictionary with the metadata written to the HDF5 file, on success. On failure, returns a dictionary with an *error* member and an associated string with a description of that error.

Return type
dict

`get_metadata`(*dataset*)

Retrieve metadata of an existing UDF dataset.

Returns
A dictionary with the UDF metadata: its name, the backend needed to execute it, the bytecode size, the dataset resolution, data type, and other relevant information. On failure, returns a dictionary with an *error* member and an associated string with a description of that error.

Return type
dict

`destroy()`

Release resources allocated for the object.

This function must be called to ensure handles are closed and avoid resource leaks. It is best, however, to use simply use a context manager to define *UserDefinedFunction* objects:

```python
with UserDefinedFunction(hdf5_file='file.h5', udf_file='udf.py') as udf:
    udf.push_dataset(...)
    udf.compile()
    udf.store()
```

4.3 JSON schema for HDF5-UDF datasets

The dictionary used to describe the UDF dataset(s) must follow the schema below.

Schema				
type	*object*			
properties				
• **name**	**Dataset name**			
	type	*string*		
• **datatype**	**Data type**			
	type	*string*		
	enum	int8, uint8, int16, uint16, int32, uint32, int64, uint64, float, double, string, compound		
• **resolution**	**Dataset dimensions**			
	type	*array*		
	items	type	*number*	
		exclusiveMinimum	0	
	minItems	1		
• members	**Compound members**			
	type	*array*		
	items	type	*object*	
		properties		
		• name	**Compound member name**	
			type	*string*
		• datatype	**Compound member data type**	
			type	*string*
			enum	int8, uint8, int16, uint16, int32, uint32, int64, uint64, float, double, string
	minItems	1		

CHAPTER

FIVE

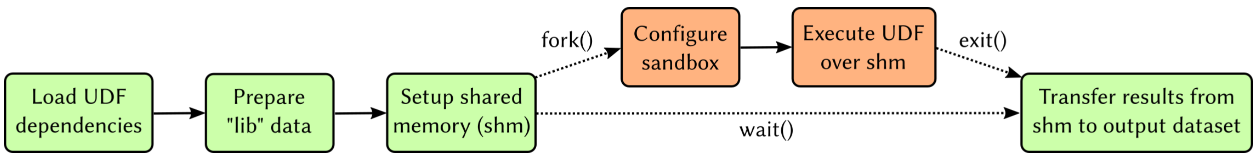
SECURITY CONSIDERATIONS

Trusting user-defined functions to execute arbitrary code on someone else's computer is always hard. All is fine until some malicious piece of code decides to explore the filesystem and mess around with system resources and sensitive data. This is such an important topic that several projects delegate dedicated teams to reduce the attack surface of components that execute code from external parties.

Because users cannot tell in advance what a certain user-defined function is about to do, HDF5-UDF uses a few mechanisms to limit what system calls the UDF can execute.

5.1 Sandboxing

The Linux implementation uses the `Seccomp` interface to determine which system calls UDFs are allowed to invoke – the UDF process is terminated if it tries to run a function that does not belong to the allow-list. The following image shows the overall architecture of our seccomp-based sandboxing.



Note that even though one could use a static list of system calls allowed to execute, that does not reflect real-world scenarios in which few people are trusted (and thus should be OK to have access to a larger set of system calls than ordinary users). We account for that situation by introducing the concept of Trust profiles.

5.2 Trust profiles

Starting with HDF5-UDF 2.0, a private and public key pair is automatically generated and saved to the user's home directory (under `~/.config/hdf5-udf`) the first time a dataset is created. The files are named after the currently logged user name:

- `~/.config/hdf5-udf/username.priv` private key
- `~/.config/hdf5-udf/username.pub`: public information: public key, email, and full name. The last two pieces of information are automatically assembled from `hostname` and `/etc/passwd`. Please review and adjust the file as you see fit

A directory structure providing different *trust profiles* is also created. Inside each profile directory exists a JSON file which states the system calls allowed to be executed by members of that profile. Three profiles are created:

- **deny**: strict settings that allow writing to `stdout` and `stderr`, memory allocation, and basic process management (so the process can `exit()`).

- **default**: a sane configuration that allows memory allocation, opening files in read-only mode, writing to `stdout` and `stderr`, and interfacing with the terminal device.
- **allow**: poses no restrictions. The UDF is treated as a regular process with no special requirements.

The profile configuration files also state which filesystem paths can be accessed by UDFs. An attempt to access a filesystem object not covered in the config file causes the UDF to be terminated with *SIGKILL*.

5.3 Signing UDFs

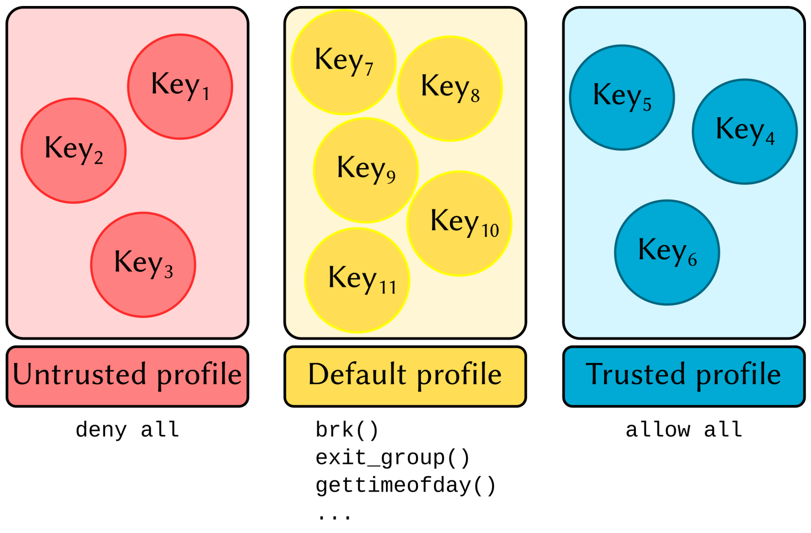
UDFs are **automatically signed** at the time of their attachment to the HDF5 file. The public key from `username.pub` and contact information from `username.meta` are incorporated as metadata and saved next to the UDF bytecode in the HDF5 file.

5.4 Associating UDFs with a trust profile

Self-signed UDFs are automatically placed on the `allow` profile. This means that UDFs you create on your own machine will run, on that same machine, as a regular process would.

HDF5 files with UDFs signed by a different user are automatically placed on the `deny` profile: the public key is extracted from the metadata and saved as `~/.config/hdf5-udf/deny/foo.pub`. In other words, when you receive a file from an unknown party and load a UDF dataset, the bytecode will not be able to perform any actions that require the execution of system calls (other than writing to `stdout` and `stderr`).

It is possible to change the trust level by simply **moving that public key to a different profile directory**. The next time a UDF signed by that key is read, the seccomp rules associated with that profile will be enforced.



CHAPTER

SIX

CONFIGURATION

Configuration of trust profiles is made by adjusting the JSON files under the corresponding trust directory:

- `~/.config/hdf5-udf/deny/deny.json`: UDFs signed by previously unseen keys are associated with this profile.
- `~/.config/hdf5-udf/deny/default.json`: settings for reasonably trusted keys
- `~/.config/hdf5-udf/deny/allow.json`: settings for UDFs signed by trusted keys

The JSON file comprises the following keys:

- `"sandbox"` (boolean): set to *false* if you don't want to enforce a sandbox to UDFs associated with this profile, *true* otherwise.
- `"syscalls"`: array of *key-value* objects describing the system call names and the conditions to allow UDFs to run them.
- `"filesystem"`: array of *key-value* objects describing the paths an UDF can access on the filesystem.

6.1 Allowing system calls

By default, all system calls are disallowed from being called by an UDF. The `"syscalls"` JSON array must explicitly state each system call a UDF can call. There are two syntaxes to state so:

1. Allow a named system call to execute regardless of the arguments provided by the user:

```
{"syscall_name": true}
```

2. Allow a named system call to execute as long as the arguments match a given criteria. Examples:

```
# A rule for write(int fd, const void *buf, size_t count)
{
  "write": {
    "arg": 0,              # First syscall argument (fd) ...
    "op": "equals",        # ... must be equal to ...
    "value": 1             # ... 1 (stdout)
  }
}

# A rule for open(const char *pathname, int flags)
{
  "open": {
    "arg": 1,              # Second syscall argument (flags) ...
```

(continues on next page)

(continued from previous page)

```
    "op": "masked_equals",  # ... when applied to bitmask ...
    "mask": "O_ACCMODE",    # ... O_ACCMODE ...
    "value": "O_RDONLY"     # ... must be equal to O_RDONLY
  }
}
```

Mnemonic values such as "O_RDONLY" are automatically translated into their numerical representation. They must be quoted as JSON strings.

Currently the two only possible values for `op` are `equals` and `masked_equals`.

String-based filtering is not supported. Selection of which filesystem paths a registered system call can access, however, is possible by setting the `"filesystem"` array. See the next section for details.

6.2 Access to files and directories

By default, access to any filesystem object is denied. The `"filesystem"` array can be used to state which parts of the filesystem can be accessed and if they're available for programs opening objects in write mode or not.

The filesystem path component can be an absolute path or a string containing wildcards (`*`). Two consecutive wildcards (`**`) can be used to recurse into subdirectories. The supported open modes are `ro` for read-only access and `rw` for both write-only and read-write requests.

Here are some examples. More settings can be found on the files shipped with HDF5-UDF (e.g., `~/.config/hdf5-udf/default/default.json`).

To allow access to any file, as long as the requested operation is read-only:

```
"filesystem": [
  {"/**": "ro"}
]
```

To allow access to Python packages:

```
"filesystem": [
  {"/**/python*/site-packages/**": "ro"}
]
```

To allow write access to /tmp:

```
"filesystem": [
  {"/tmp/**": "rw"}
]
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

hdf5_udf, 12

u

udf_template, 7

INDEX

C

compile() (*hdf5_udf.UserDefinedFunction method*), 13

D

destroy() (*hdf5_udf.UserDefinedFunction method*), 13

G

get_metadata() (*hdf5_udf.UserDefinedFunction method*), 13
getData() (*udf_template.PythonLib method*), 8
getDims() (*udf_template.PythonLib method*), 8
getFilePath() (*udf_template.PythonLib method*), 7
getType() (*udf_template.PythonLib method*), 8

H

hdf5_udf
    module, 12

M

module
    hdf5_udf, 12
    udf_template, 7

P

push_dataset() (*hdf5_udf.UserDefinedFunction method*), 12
PythonLib (*class in udf_template*), 7

S

set_option() (*hdf5_udf.UserDefinedFunction method*), 12
setString() (*udf_template.PythonLib method*), 7
store() (*hdf5_udf.UserDefinedFunction method*), 13
string() (*udf_template.PythonLib method*), 7

U

udf_template
    module, 7
UserDefinedFunction (*class in hdf5_udf*), 12