NPS-CS-17-001

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

BE SCAN TOOLSET

by

Bruce Allen

September 29, 2017

Approved for public release; distribution is unlimited

Prepared for: DIA

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
09-30-2017	Technical Report	2016-10-01—2017-09-30

4. TITLE AND SUBTITLE

BE SCAN TOOLSET

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Bruce Allen

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

NPS-CS-17-001

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DIA

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.IRB Protocol Number:N/A.

14. ABSTRACT

BE Scan provides a software stack of libraries and tools which, collectively, enable an infrastructure for scanning for digital artifacts in a big-data compute environment.This software stack includes a `lightgrep_wrapper` library for easing interfacing to `liblightgrep`, a `be_scan` library which provides scanner interfaces, an `artscan.py` tool for scanning using multiple processes, and a *spark_be_scan* infrastructure for scanning using a compute node cluster.This document describes the libraries and tools provided in this software stack.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Ronald A. Route President	Steven Lerman Provost

The report entitled *"BE Scan Toolset"* was prepared for and funded by Defense Intelligence Agency (DIA), Washington, D.C.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Bruce Allen
Faculty Associate Research
Computer Sciences Department

Reviewed by:	**Released by:**
Peter Denning, Chairman Computer Sciences Department	Jeffrey D. Paduan Dean of Research

Table of Contents

Abstract

BE Scan provides a software stack of libraries and tools which, collectively, enable an infrastructure for scanning for digital artifacts in a big-data compute environment. This software stack includes a `lightgrep_wrapper` library for easing interfacing to `liblightgrep`, a `be_scan` library which provides scanner interfaces, an `artscan.py` tool for scanning using multiple processes, and a *spark_be_scan* infrastructure for scanning using a compute node cluster. This document describes the libraries and tools provided in this software stack.

1 Introduction

The *BE Scan Toolset* is comprised of the following softwares:

- The *Lightgrep Wrapper* C++ library.
- The *be_scan Library* C++ library with Python and Java bindings.
- The *artscan.py Tool* for scanning using multiple processors.
- The *spark_be_scan Infrastructure* for scanning in a compute node cluster.

The scanner engine used by the *BE Scan Toolset* is *Liblightgrep*, a regular expression parser (c) 2010-2015, Stroz Friedberg, LLC, available under the GPLV3 license at https://github.com/strozfriedberg/liblightgrep. The regex syntax used by *Liblightgrep* is similar to POSIX regex syntax and is described in Section 2.

The interface to *Liblightgrep* exposes internal complexity and does not assist with callback management, so we created *Lightgrep Wrapper* to encapsulate *Liblightgrep* and to provide the basic requirements of configuring a scanner and scanning data. *Lightgrep Wrapper* is discussed in Section 3.

The heart of the *BE Scan Toolset* is the *be_scan Library*, the API for scanning for matches in buffers of data. The *be_scan Library* contains the infrastructure to support scanning and also for recursive uncompression. We have added scanners specifically targeted to find digital forensic artifacts. The *be_scan Library* supports **C++**, **Python**, and **Java** interfaces. The *be_scan Library* is described in Section 4.

We offer the *be_scan Library* to users who wish to embed scanning for forensic artifacts into their forensic toolchain. Additionally, we offer the *artscan.py Tool* as an end solution for scanning on a system using multiple processors, described in Section 5, and the *spark_be_scan Infrastructure* as a template for scanning in a distributed compute environment, described in Section 6.

Notes for installing the *BE Scan Toolset* and its requisite libraries are presented in Section 7. We conclude with a discussion of future work in Section 10.

2 Regular Expressions and *Liblightgrep*

Internally, the *BE Scan Toolset* uses liblightgrep which requires syntax that is very similar to standard regex syntax. Here are some links to lightgrep and its regex syntax:

- Lightgrep Cheat Sheet: http://strozfriedberg.github.io/liblightgrep/resources/LightgrepSyntaxCheatSheet.pdf
- Lightgrep syntax and other info: http://strozfriedberg.github.io/liblightgrep
- The liblightgrep repository: https://github.com/LightboxTech/liblightgrep

Use of the unbounded `S*` repitition operator will result in very slow performance because *Liblightgrep* must track long matches. To maintain good performance, please avoid the unbounded `S*` repetition operator in your regular expressions. Please instead use the bounded `S{n,m}` repetition operator.

3 *Lightgrep Wrapper*

The *Lightgrep Wrapper* wraps *Liblightgrep* to provide a simple regular expression scanner interface:

- Sequences of setup steps are hidden.
- Callback table management is hidden.
- Internal lightgrep interfaces are decoupled and are not exposed.

Liblightgrep is described in Section 2.

Here is the basic *Lightgrep Wrapper* workflow:

- Create a *Liblightgrep* scanner program using `lw_scanner_program_t` and configure it with regular expressions and callback functions.
- Create scanner instances of `lw_scanner_t` and scan data streams for matches. On matches, your callback function will be called. Within your callback function, you may use the `read_buffer` helper function to read actual match data and use your `user_data` to help consume your matches.

For additional resources, please refer to the following:

- The **C++** header file: https://github.com/NPS-DEEP/lightgrep_wrapper/blob/master/src/lightgrep_wrapper.hpp.

- Examile usage in the *Lightgrep Wrapper* Test file: https://github.com/NPS-DEEP/lightgrep_wrapper/blob/master/test/tests.cpp.
- Also, please refer to the *be_scan Library* at https://github.com/NPS-DEEP/be_scan for a reference implementation of *Lightgrep Wrapper*. The *be_scan Library* implements some of the scanners in *bulk_extractor*, ref. https://github.com/simsong/bulk_extractor.

Now we describe the components of the *Lightgrep Wrapper* API.

3.1 Callback Functions

Prepare callback functions to go with your regular expressions. When a scanner finds a regex match, its associated callback function will be called. Here is the syntax for callback functions:

```
typedef void (*scan_callback_function_t)(const uint64_t start,
                                         const uint64_t size,
                                         void* user_data);
```

- Parameter `start` is the offset of the scan hit with respect to the beginning of your scan stream.
- Parameter `size` is the size of the scan hit.
- Parameter `user_data` is your user data. You consume the match using information in `user_data`. In example file `tests.cpp`, `user_data` contains buffer pointers suitable for printing matches directly. In *be_scan Library*, header file `scanner_data.hpp`, `user_data` includes a pointer to an Artifacts queue. The callback function pushes to this Artifacts queue. The Artifacts queue is consumed when the *be_scan Library* interface is used to pull from it, see Section 4.1.4.

3.2 Public Interfaces

3.2.1 Scanner Program

Scanner program `lw_scanner_program_t` contains the regular expressions and the callback functions that scanner `scanner_t` will use. The scanner program has two states:

- Before `finalize_program`: Call `add_regex` to add regular expressions and callback functions. You cannot create `scanner_t` objects from scanner programs that have not been finalized.
- After `finalize_program`: Create `scanner_t` objects that use the scanner program. Calls to `add_regex` will return an error.

Here are the `lw_scanner_program_t` interfaces:

- Constructor `lw_scanner_program_t()` Obtains a fresh unconfigured scanner program.
- Interface `add_regex(regex, character_encoding, is_case_insensitive, is_fixed_string, callback_function` adds a regular expression to the program. Parameters are:
  - Parameter `‘regex‘`contians the regular expression, see http://strozfriedberg.github.io/liblightgrep for syntax.
  - Parameter `character_encoding` specifies the character encoding to target the scan for, for example `UTF-8` or `UTF-16LE`.
  - Parameter `is_case_insensitive` defines case sensitivity. True to match upper and lower case, false to match exact case.
  - Parameter `is_fixed_string` defines fixed-string mode, False=grep, true=fixed-string. Use false.
  - Parameter `callback_function` points to your callback function that will be called by patterns that match this regular expression.
- Interface `finalize_program(is_determinized)` Finalizes the scanner program. Once finalized, you can create scanners using `lw_scanner_t` and you can no longer use `lw_scanner_program.add_regex`. Parameters are:
  - `is_determinized` defines determinism. False=NFA, true=DFA(pseudo). Use false. See https://github.com/LightboxTech/liblightgrep.

3.2.2 Scanner

Use lightgrep wrapper scanner `lw_scanner_t` to perform scans on buffers of data.

- Constructor `lw_scanner_t(lw_scanner_program, *user_data)` Creates a lightweight scanner object to use for scanning. Parameters:
  - Parameter `lw_scanner_program` provides a constant reference to an immutable scanner program.
  - Parameter `*user_data` is a pointer to your user data containing the context you will need for your callback function to consume your scan hits.
- Variable `program_is_finalized` is a convenience variable you can use to make sure your scanner program was finalized. If the scanner program has not been finalized then the scanner will fail. You may check this to see if the scanner initialized correctly.
- Interface `scan(stream_offset, buffer, size)` Scans the `size` bytes of `buffer`. Use `stream_offset` as the offset to the start of the buffer so that scan hits can know the hit offset with respect to the beginning of the stream. Call `scan` repeatedly to scan a stream of buffers. Hits near the end of a given scanned buffer may not be reported because the match may extend into the next buffer. This is good because

it ensures the full match can be captured. These uncompleted matches are captured while scanning the next buffer or are captured during a finalize operation.
- Interrface `scan_finalize()` finalizes the scan, capturing any uncompleted matches.
- Interface `scan_fence_finalize(stream_offset, buffer, size)` Scans the `size` bytes of `buffer`, using `stream_offset` as the offset to the start of the buffer. Captures uncompleted hits that start before `buffer` and may extend into the buffer. Does not capture hits that start in `buffer`; they are after the fence.

3.2.3 Helper Function

Helper function `read_buffer(buffer_offset, previous_buffer, previous_buffer_size, buffer, buffer_size, offset, length, padding)` is a convenience function that reads a string of bytes given a buffer offset, two buffers that should enclose the bytes to read, the offset into the buffers to begin the read, the length to read, and padding bytes to read:

- Parameter `buffer_offset` specifies the stream offset to `buffer`.
- Parameter `previous_buffer` is the buffer before but adjacent to `buffer`
- Parameter `previous_buffer_size` is the size of `previous_buffer`.
- Parameter `buffer` The buffer that `buffer_offset` points to the beginning of.
- Parameter `buffer_size` The size of `buffer`.
- Parameter `offset` The stream offset to the data to read. Unless constrained by the beginning or end of your stream, the two buffers should fully contain the byte string to read.
- Parameter `length` The number of bytes to read.
- Parameter `padding` The number of padding bytes to read before `offset` and after `offset+length` or `0` for none. Padding allows context bytes around the match being read.

4 *be_scan Library*

The *be_scan Library* is a scanner library for scanning for digital forensic artifacts. It does the heavy work of scanning and returning found artifacts. It does not read files or store found artifacts. Use *be_scan Library* to scan what you like and consume found artifacts as you like using the *be_scan Library*'s **Python** or **Java** language bindings.

The *be_scan Library* was created in order to offer a simple, modular API for scanning for digital forensic artifacts and for working with these artifacts in a big-data context:

- The *be_scan Library* works well in a system with **multiple processors** because it works with discrete buffers and buffer streams. For example **Python** tool *artscan.py Tool* which uses the *be_scan Library* to recursively scan a media image file using

multiple processors to find artifacts. The *be_scan Library* tool is described in Section 4.
- *be_scan Library* works well in a compute cluster where media images are in a distributed file system such as **HDFS**. For example a **Java**-based scanner using *be_scan Library* with **Spark** and **HDFS** on a compute cluster, described in Section 6.

be_scan Library works well in these situations:

- Embedded within larger Forensics applications.
- Your own custom programs requiring your own custom behavior.
- Big-data processing.
- Or use it through the *artscan.py Tool*.

Currently, the *be_scan Library* contains an **email** scanner and **zip** and **gzip** decompression scanners. Please see Section 10 for information on future work. Please see Section 4.2 and 4.3 for information on adding your own scanners and decompressors.

Interfaces in the *be_scan Library* support creating a scan engine, scanning buffers of data, uncompressing regions of data, and managing found artifacts. The *be_scan Library* API is available through a **C++** header file and via **Python**, and **Java**. The C++ header file is available at https://github.com/NPS-DEEP/be_scan/blob/master/src/be_scan.hpp. Usage varies slightly by language. Example usage by language is available here:

- **C++**: https://github.com/NPS-DEEP/be_scan/blob/master/test/basic_tests.cpp
- **Python**: https://github.com/NPS-DEEP/be_scan/blob/master/python_bindings/artscan_multi.py
- **Java**: https://github.com/NPS-DEEP/be_scan/blob/master/java_bindings/Tests.java

The basic work flow for using the *be_scan Library* is:

- Add any custom regular expressions you may want.
- Build your scanner engine.
- Allocate your artifacts queue.
- Read data, scan it, and consume found artifacts.
- For compression artifacts, use the uncompressor to obtain uncompressed buffers to scan.

4.1 Public Interfaces

The *be_scan Library* provides three categories of interfaces:

- General functions provide general information and include an interface for configuring the dedicated `custom_regex` scanner.

- Scanner related interfaces provide scan support including creating a scanner engine, and scanner and managing artifacts.
- Uncompression related interfaces consist of the uncompressor and uncompressed data.

4.1.1 General Functions

- Function `version()` returns the version of *be_scan Library*.
- Function `available_scanners()` returns a list of available scanners.
- Function `escape(unescaped_string)` returns the string with unprintable ASCII charactors escaped.
- Function `add_custom_regex_pattern(pattern)` adds a custom pattern to the *be_scan Library*'s `custom_regex` scanner.
- Function `clear_custom_regex_patterns()` clears the list of custom regex patterns. This function is provided for completeness and for testing, and is typically not used.

4.1.2 Scan Engine

Obtain a scan engine configured with your set of requested scanners. It can be used by scanners.

- Constructor `scan_engine_t(requested_scanners)` creates a regex scan engine that scans for your list of requested scanners.
  - Parameter `requested_scanners` contains the list of available scanners separated by space characters.
- Variable `status` holds the working status of the scan engine, "" if valid else error message.

4.1.3 Artifact

When scanners find a scan match, they prepare an Artifact object of type `artifact_t` containing information relating to the match.

- Member `artifact_class` indicates the artifact type, for example `email` or `zip`.
- Member `stream_name` contains the stream name supplied to the scanner when the scan was invoked.
- Member `recursion_prefix` contains any forensic path recursion prefix, or "" on no prefix.
- Member `offset` indicates the stream offset as supplied to the scanner when the scan was invoked.
- Member `artifact` contains the text returned for the found artifact.

- Member `context` contains context information returned for the found artifact.
- Interface `blank()` returns true if the artifact is completely blank.
- Interface `to_string()` returns the artifact as formatted escaped text in the form `<stream filename><space><artifact class><space><path><tab> <escaped artifact><tab><escaped context>`.
- Interface `javaArtifact(java_artifact)` allows Java to return the `artifact` member as a byte array.
- Interface `javaContext(java_context)` allows Java to return the `context` member as a byte array.

4.1.4 Artifacts

Artifacts is a threadsafe FIFO queue for holding found artifacts. Scanners use its `put()` method to enqueue artifacts. You use its `get()` method to dequeue them.

- Member `put()` adds an artifact to the queue. Scanners call this as artifacts are found.
- Member `get()` returns an artifact from the queue or returns an empty artifact if the queue is empty. You call this to consume found artifacts.
- Member `empty()` returns true when the Artifacts queue is empty.

4.1.5 Scanner

You obtain and use a scanner for scanning. Supply the scan engine containing your regex scan program and supply the Artifacts queue that you would like found artifacts to go into.

- Constructor `scanner_t(scan_engine, artifacts)` allocates a scanner given your scan engine to use and your artifacts queue to put found artifacts into.
- Interface `scan_setup(stream_name, recursion_prefix)` sets up the stream name and recursion prefix values so that they may be placed into `artifact_t` data for found artifacts. The stream name is typically the filename of a media image. The recursion prefix typically contains forensic path prefix text for artifacts obtained in uncompressed data extracted from the stream.
- Interface `scan_stream(stream_offset, previous_bytes, bytes)` scans bytes from the `stream_offset` offset provided. Provide `previous_bytes` so that artifact text can be captured for artifacts that start in the previous buffer of the stream. `previous_bytes` only needs to contain a few hundred bytes at most, just enough to capture your longest possible artifact. `bytes` will be scanned. Artifacts that start in `bytes` but are not captured until they are concluded in the next part of the stream or are concluded as the final bytes of the stream via `scan_final` or `scan_fence_final`.
- Interface `scan_final(stream_offset, previous_bytes, bytes)` is similar to interface `scan_stream`, but the scan will be finalized at the end of the `bytes`

buffer, concluding and capturing any started artifacts.
- Interface `scan_fence_final(stream_offset, previous_bytes, bytes)` is similar to interface `scan_stream`, but artifacts started in `previous_bytes` that may extend into `bytes` will be captured. Artifacts that start in `bytes` are on the other side of the fence and are not captured.

4.1.6 Uncompressed

When the uncompressor uncompresses, it returns data of type `uncompressed_t`. Uncompressed data comes from the uncompressor, so there is no constructor.

- Variable `buffer` contains the bytes of uncompressed content, which can be of size zero if the uncompressor could not uncompress anything.
- Variable `status` contains "" or error text from the uncompressor indicating why the uncompress request failed.

4.1.7 Uncompressor

You use an uncompressor object to uncompress content from within a buffer. Because megabytes can be returned by uncompression, the uncompressor object keeps a large internal scratch buffer. During scanning, try to keep uncompressor objects persistent. They are not threadsafe, so use one per thread.

- Constructor = `uncompressor_t()` creates your uncompressor object. Internally, it will maintain a large internal scratch buffer, so avoid having a large turnover of uncompressor objects.
- Interface `uncompress(buffer, buffer_offset)` uncompresses content at location `buffer_offset` within buffer `buffer` and returns an `uncompressed_t` object containing a buffer of uncompressed content and status indicating whether the uncompress operation was successful.
- Interface `close()` closes the uncompressor's resources by deallocating the internal scratch buffer. It is just fine to let the destructor do this. Java users may want to use this in an effort to free the internal buffer before waiting for the garbage collector to free the uncompressor which will then deallocate its internal scratch buffer.

4.2 Developing New Scanners

This section describes what is necessary to develop your own scanner. Please see existing `scan_*` scanners for examples. We recommend that you develop on a *be_scan Library* clone. If you like, you may request that your scanners be merged into *be_scan Library*.

4.2.1 Working with the scanner_data_t Interface

The `scanner_data_t` structure gets passed to your regex handler. It contains all the information your handler needs to create complete `artifact_t` artifacts:

- Variable `artifacts_t` points to the artifacts FIFO queue that you will `put` `artifact_t` artifacts onto.
- Variable `stream_name` contains the name of the stream provided in interface `scanner_t.scan_setup`.
- Variable `stream_offset` contains the stream offset to the buffer provided by the scanner's scan operation.
- Variable `recursion_prefix` contains any recursion prefix provided via interface `scanner.scan_setup`.
- Variable `buffer` contains a pointer to the buffer being scanned. You may pass it along with `previous_buffer` to *Lightgrep Wrapper* interface `lw::read_buffer` to obtain artifact bytes and context bytes.
- Variable `previous_buffer` points to a buffer containing any amount of the previous buffer adjoining `buffer` so that you may pass it along with `previous_buffer` to `lw::read_buffer` to obtain artifact bytes and context bytes.

4.2.2 Creating your scan_<your scanner>.cpp File

- Please use a unique namespace for your scanner to keep it well separated from other scanners.
- Write your regular expression callback function handlers, one per regex, or one handler if your expressions can use the same handler. The general form is:
  ```
  void your_regex_handler(const uint64_t start, const uint64_t size,
                          void* scanner_data_t);
  ```
  where `start` and `size` are the offset and length of the artifact matched, and `*scanner_data_t` points to your scanner data.
- Write a function for adding your regular expressions to the scanner program. The general form is:
  ```
  std::string add_your_regex(lw::lw_scanner_program_t& lw_scanner_program);
  ```
  where `lw_scanner_program` is the the scanner program you will add your regular expressions and handlers to. This function will call `lw_scanner_program.add_regex` for each regex you want to add. Basically, it includes the regex, the regex character encoding, control flags, and the callback function pointer.

For examples of existing scanners please see https://github.com/NPS-DEEP/be_scan/blob/master/src/scan_email.cpp for the *email* scanner or https://github.com/NPS-DEEP/be_scan/blob/master/src/scan_uncompression.cpp for the *uncompression* scanner.

4.2.3 Modifying the scanners.hpp File

- Add your `your_regex_handler` functions and your `add_your_regex` function headers into `scanners.hpp` using your namespace.

4.2.4 Modifying the scanners.cpp File

- Add your scanner name to the list of available scanners in the `available_scanners` variable.
- Add the call to your `add_your_regex` function. Make its invocation conditional on whether the user requested your scanner.

4.2.5 Modifying the makefile.am File

- Add your new `scan_<your scanner>.cpp` filename to the list of files that contain code that goes into the *be_scan Library*.

4.3 Developing New Uncompressors

Uncompressors uncompress data that has been compressed or encoded. The *be_scan Library* provides *zip* and *gzip* uncompressors. This page describes what is necessary to develop uncompressors. There are two parts to developing uncompressors:

- The scanner part, which simply contains regex patterns for finding probable compression regions such as *zip* data. You will add your regular expression pattern and your uncompression callback function handler to the *uncompression* scanner, file https://github.com/NPS-DEEP/be_scan/blob/master/src/scan_uncompression.cpp.
- The uncompression part, which is the topic of this section.

If you like, you may request that the uncompressors you develop be merged into the *be_scan Library*.

4.3.1 Working with the uncompressor_t interface

You will add code in file `uncompressor.cpp` to see if data matches the pattern for your uncompressor. When it matches it will call your uncompressor.

4.3.2 Working with the uncompressed_t interface

Your uncompressor will return an object of type `uncompressed_t` containing the uncompressed data and status of success. Status will be `""` unless there is an uncompression error, which your particular uncompressor can generate. An uncompression error can also be generated by `uncompressed_t` if it is unable to allocate space to hold your uncompressed data.

4.3.3 Creating your uncompress_<your uncompressor>.cpp File

Your uncompressor file will contain the uncompress function that you will write. Here is the general form:

```
uncompressed_t uncompress_<your uncompressor>(const unsigned char* const in_buf,
                                              const size_t in_size,
                                              unsigned char* const scratch_buf,
                                              const size_t scratch_buf_size)
```

This function takes the following input:

- A input buffer and input buffer size of the buffer containing the compressed data.
- The offset in the input buffer to where the compressed data is.
- A scratch buffer and scratch buffer size that your uncompressor can use while uncompressing. Use `scratch_buffer` from `uncompressor_t`.

It returns `uncompressed_t` which contains your uncompressed data and status of success or "" if no error.

4.3.4 Modifying the uncompressors.hpp File

- Add your `uncompress_<your uncompressor>` function to this header file.

4.3.5 Modifying the uncompressor.cpp File

Add a conditional to recognize the pattern for your type of compressed content and to call your uncompressor, `uncompress_<your uncompressor>`, if it matches. Here is an example for recognizing and then uncompressing content for *zip*:

```
// zip
if (buffer_offset + 30 < buffer_size &&
    buffer[buffer_offset+0]==0x50 && buffer[buffer_offset+1]==0x4B &&
    buffer[buffer_offset+2]==0x03 && buffer[buffer_offset+3]==0x04) {

  return uncompress_zip(buffer, buffer_size, buffer_offset,
                        scratch_buffer, max_uncompressed_size);
}
```

4.3.6 Modifying the Makefile.am File

- Add your new `uncompress_<your uncompressor>.cpp` filename to the list of files that contain code that goes into the *be_scan Library*.

5 *artscan.py Tool*

The *artscan.py Tool* tool scans media image files for digital artifacts. Here are some highlights:

- It scans recursively into compressed regions.
- It scans using multiple processors.
- It uses the *be_scan Library* which uses the *Lightgrep Wrapper* which uses *Liblightgrep*.
- It is written in Python.

5.1 Setup

Be sure to set `PYTHONPATH` as described in Section 7.3.2. For a user setting install, also set your `PATH` variable to point to the `bin/*` directory where `artscan.py` is installed, for example this might be: `export PATH=$HOME/local/bin:$PATH`. For an administrator setting install, the *artscan.py Tool* tool should already be visible.

5.2 Input

Required input to the *artscan.py Tool* consists of:

- The filename of a raw media image file.

Optional input to the *artscan.py Tool* consists of:

- The set of scanners to use, or all scanners by default.
- The maximum recursion depth for the forensic path, 7 by default.
- Verbosity of runtime status, no status by default.
- An optional regex file containing a list of custom regular expressions to scan for, see [[Adding Custom regex Patterns]].

5.3 Output

Output consists of all artifacts found, formatted as follows:

```
<filename><space><artifact class><space><forensic path><tab>
<artifact text><tab><context>
```

For example input file `testfile` might produce an `email` artifact at offset `100` for artifact `aaa@bbb.com` embedded in context `aaa@bbb.com` as follows:

```
testfile email 100    aaa@bbb.com                    aaa@bbb.com
```

And input file `testfile` might produce a `zip` artifact at offset `200`, where the calculated MD5 of the uncompressed data is `8ef6ce99fa6e5bbfac040cc51c98f0a9` and the uncompressed data is `94432` bytes in size as follows:

```
testfile zip 200    8ef6ce99fa6e5bbfac040cc51c98f0a9        94432
```

Note that the *artscan.py Tool* calculates the **MD5** of the uncompressed region and inserts it into the artifact record printed, along with the length of the uncompressed region. The artifact returned by the *be_scan Library* does not provide useful information for the artifact field or the context field.

This next example shows an email artifact 50 bytes into an uncompressed zip region located 200 bytes into input file `testfile`:

```
testfile email 200-ZIP-50    ccc@ddd.com                        ccc@ddd.com
```

5.4 Usage

Here is the usage from the *artscan.py Tool* as seen by typing `artscan.py -h`.

```
usage: artscan.py [-h] [-e ENABLED_SCANNERS] [-d {1,2,3,4,5,6,7}] [-v {0,1,2}]
                  [-c CUSTOM_REGEX_FILE]
                  filename

Scan a file for forensic artifacts using multiple processes and recursion.

positional arguments:
  filename              the file to scan

optional arguments:
  -h, --help            show this help message and exit
  -e ENABLED_SCANNERS, --enabled_scanners ENABLED_SCANNERS
                        enable specific scanners, default: 'email
                        uncompression custom_regex'
  -d {1,2,3,4,5,6,7}, --recursion_depth {1,2,3,4,5,6,7}
                        recursively scan into decompressed regions
  -v {0,1,2}, --verbose {0,1,2}
                        show runtime status, 0=none, 1=scan status, 2=scan
                        status plus all the bytes being scanned
  -c CUSTOM_REGEX_FILE, --custom_regex_file CUSTOM_REGEX_FILE
                        a file containing custom regular expressions to scan
                        for
```

5.5 Under the Hood

This section describes the internal operations of the *artscan.py Tool*. In summary:

- `artscan.py` calculates media image regions and asynchronously runs processes that recursively scan their assigned region.
- Overflow bytes just beyond the assigned region are used to catch artifacts and compressed data that extend into the next region.
- You can modify `artscan.py` to skip unwanted artifacts or to customize `artifact` or `context` fields before reporting them.

5.5.1 The `main` Function

The `main` entry to `artscan.py` works as follows:

1. Parse input including the filename to scan and the file of custom regular expressions, if provided.
2. If a custom regex file is provided: add each regex to `be_scan` via `add_custom_regex_pattern regex_pattern`.
3. Build the scan engine. It is immutable, so it is available as a global resource for scanner processes.
4. Divide the work into pages plus some margin to catch artifacts that cross page boundaries. Specifically, divide the work into pages of 128MiB with a 1MiB margin allowance.
5. For each page: create a future object to be run as a process on a CPU. The future object consists of the `scan_range` function along with a tuple of parameters about the scan range, the input file, and depth and verbosity options.
6. `artscan.py` then asynchronously runs each future object, printing the artifacts returned by each future object when it completes.

5.5.2 The `scan_range` function

As mentioned in the `main` function for `artscan.py`, each future object runs function `scan_range` on one range of the media image's data. Here is the flow of the `scan_range` function:

1. Open the media image file and seek to the offset containing the range to scan, then read the range and the margin content into buffers to scan.
2. Create an empty list of `formatted_artifacts`. Artifacts will be formatted as strings and put in this list as they are found.
3. Create a scanner instance for scanning this range. Provide the immutable scan engine so the scanner can access the scan program to use, and provide and a fresh artifacts object so the scanner instance can have a place to put found artifacts.

4. Scan the range using `scan_stream`.
5. Scan the margin using `scan_fence_final`. This captures artifacts that start in the range but extend into the margin.
6. Iterate over each artifact and process it:
   (a) See if it is a compression artifact, specifically, *zip* or *gzip*. If it is:
      i. Prepare an uncompressor so we can uncompress artifact data and recursively scan into it. The uncompressor takes up lots of memory ($256*1024*1024 = 256MiB$) so we make it process-global and never prepare it if we do not need it.
      ii. Identify the buffer region to uncompress, and uncompress it. If it is less than 1MiB away from the end of the buffer, prepare a region with the buffer appended and uncompress that.
      iii. If there is nothing to uncompress, disregard this artifact and do not report it.
      iv. Since something does uncompress, we improve the artifact record so that it contains information about the uncompressed data: we put the MD5 hexdigest of the uncompressed data into the `artifact` field and we put the length of the uncompressed data in the `context` field.
      v. We disregard this artifact and do not report it if it is obviously useless. Specifically, we skip it if its MD5 is `8da7a0b0144fc58332b03e90aaf7ba25`.
   (b) Here we can check and massage artifact records for other artifact classes as we did for *zip* and *gzip*. For example we can skip specific email addresses. Currently, we have nothing else to prepare for, but this is where it would go if we did.
   (c) Now the artifact is as we want and we want it, so we add it to our list of `formatted_artifacts`.
   (d) If the artifact generated uncompressed data and we want recursion then we call the `recurse` function to process the uncompressed buffer. Parameters for `recurse` are the uncompressed buffer, the forensic path prefix leading up to the uncompressed buffer, the recursion depth which is one, the uncompressor in case it might be used agin, a reference to `formatted_artifacts` so that the `recurse` function can store artifacts that it finds, `max_depth` to protect `recurse` from recursive compression bombs, and `verbose` so that `recurse` can know how much runtime status to emit.
7. Now all artifacts are processed and all recursion has completed, so we return the pickled list of formatted artifacts back to the main program.

5.5.3 The `recurse` Function

As mentioned in the `scan_range` function, above, the `recurse` function scans uncompressed data and, if it finds compressed data, recursively calls `recurse` with the next,

deeper, forensic path prefix. Here is the flow of the `recurse` function:

1. Open a scanner just for this buffer containing this uncompressed region. Scanner objects are lightweight, they are basically structures with pointers, so it is cheap to have them come and go.
2. Scan the buffer using `scan_final` to scan and finalize scanning at the same time. This is simpler than is done in `scan_range` which scans into overflow space.
3. For each artifact found:
   (a) If the artifact is a compression artifact then uncompress its region, prepare it, and maybe skip it, as is done for `scan_range`.
   (b) Add the artifact to `formatted_artifacts`.
   (c) If the artifact is a compression artifact and recursion has not reached maximum depth: calculate the next recursion prefix and recurse to the `recurse` function providing the newly uncompressed buffer, the next recursion prefix, and the next recursion depth value.

6 *spark_be_scan Infrastructure*

The latest version of spark_be_scan has not been tested and as such is available as a reference only.

The *spark_be_scan Infrastructure* provides bulk-scan capability in a big-data environment. It uses the **Java** bindings of *be_scan Library* to perform scanner functions.

The *spark_be_scan Infrastructure* is run on a compute node cluster using Gradle by typing `gradle run`. This starts function `main` in file https://github.com/NPS-DEEP/spark_be_scan/blob/master/src/main/java/edu/nps/deep/spark_be_scan/SparkBEScan.java which allocates cluster resources, makes shared object resources available on each node, sets up the list of media files in **HDFS** to process, then sets the executors running, scanning partitions of files. As executors find artifacts, they obtain the string representation of the artifact using the artifact's `toString` interface, then put the artifact string into an RDD for collection and export when all scanning is done.

Class `BEScanSplitReader` reads a partition of a media image file from **HDFS** into a byte array and calls `scan` in class `ScanBuffer`. The `scan` function gets a scanner object and scans the byte array, recursively scanning uncompressed regions. The process of recursion, substituting the **MD5** in the artifact field, etc. is the same as the process in the *artscan.py Tool*, but is written in Java.

7 Installing the *BE Scan Toolset*

This section describes how to install components of the *BE Scan Toolset*. In addition to these components, *Liblightgrep* and other packages are required. Links to online instructions are also provided. These instructions are tailored for installing as administrator on a Fedora system. As an additional reference, instructions are presented for installing these components in user space on a CentOS 6.8 system in Section 7.6.

7.1 Installing *Liblightgrep*

Liblightgrep requires GCC 4.6+ or Clang 3.1+, Boost headers 1.49+, and ICU 1.48+. Once these dependencies are met, download and install *Liblightgrep*:

- `git clone --recursive git://github.com/LightboxTech/liblightgrep.git`
- `cd liblightgrep`
- `autoreconf -fi`
- `./configure`
- `make -j4`
- `sudo make install`

For more details on installing liblightgrep please see https://github.com/strozfriedberg/liblightgrep/blob/master/BUILD.md.

7.2 Installing *Lightgrep Wrapper*

Download *Lightgrep Wrapper* from GitHub then build and install it:

- `git clone https://github.com/NPS-DEEP/lightgrep_wrapper.git`
- `cd lightgrep_wrapper`
- `./bootstrap.sh`
- `mkdir build`
- `cd build`
- `../configure`
- `make`
- `sudo make install`

You may test your *Lightgrep Wrapper* installation:

- `make check`

7.3 Installing *be_scan Library*

Obtain *Lightgrep Wrapper* from GitHub:

- `git clone https://github.com/NPS-DEEP/lightgrep_wrapper.git`
- `cd lightgrep_wrapper`

Build:

- `./bootstrap.sh`
- `configure`
- `make`
- `sudo make install`

You may test your *Lightgrep Wrapper* installation:

- `make check`

Currently, `make install` must be run before `make check` for Python tests to pass because Python tests require packages from `PYTHONPATH`.

Additional setup is required for your programs to be able to access the *be_scan Library* based on the language you want to use:

7.3.1 Java

To use the Java bindings your Java program must:

- Include the `be_scan_jni.jar` file when you compile your Java code. This file is located under the `share` directory set by `configure`, for example `$home/local/share`.
- Load the `libbe_scan_jni.so` shared object file when your Java code starts to run. This file is located under the `lib` directory set by `configure`, for example `$home/local/lib`.

7.3.2 Python

The `PYTHONPATH` system variable may be set to allow the *be_scan Library* interface files `be_scan.py` and `_be_scan.so` to be visible at the command prompt. The path to these resources depends on the prefix path set by `configure`. We recommend that you add your `PYTHONPATH` settings to your `~/.bashrc` file for easy access to these resources. Here is an example user setting:

- `export PYTHONPATH=$HOME/local/lib/python2.7/site-packages:`
- `        $HOME/local/lib64/python2.7/site-packages:$PYTHONPATH`

Here is an example administrator setting:

- `export PYTHONPATH=/usr/local/lib/python2.7/site-packages:`
- `        /usr/local/lib64/python2.7/site-packages:$PYTHONPATH`

7.3.3 C++

Compile against header file `be_scan.hpp`. Link against shared object library file `libbe_scan.so`. These files are installed by running `make install` into directories set by `configure`, for example `$HOME/local/include` and `$HOME/local/lib`.

7.4 Installing the *artscan.py Tool*

The *artscan.py Tool* is installed along with the *be_scan Library* when installing *be_scan Library*. Be sure to set up the `PYTHONPATH` as described in Section 7.3.2.

7.5 Installing the *spark_be_scan Infrastructure*

Install the *spark_be_scan Infrastructure* by typing the following:

- `git clone https://github.com/NPS-DEEP/spark_be_scan.git`

At the time of writing, the latest version of *spark_be_scan Infrastructure* has not been tested and as such is available as a reference only.

7.6 Installing the *BE Scan Toolset* on CentOS 6.8

Liblightgrep requires a newer GCC, BOOST, and ICU than is available by default on CentOS 6.8. These must be installed first.

Build a newer compiler

Ref. https://gcc.gnu.org/install/index.html.

Note: liblightgrep recommends GCC 4.6+, but code uses `set::emplace` which requires 4.8.0+ so use GCC 4.9.4.

Also note: 4.9.4 does not yet support constructor that calls another constructor, so change liblightgrep code in thread.h:28 and 30 to not do this.

Get GCC:

- **wget http://www.netgull.com/gcc/releases/gcc-4.9.4/gcc-4.9.4.tar.bz2**
- **tar jxf gcc-4.9.4.tar.bz2**
- **cd gcc-4.9.4**

Get requisite packages, ref. https://gcc.gnu.org/wiki/FAQ#configure:

- **./contrib/download_prerequisites**

Make the build directory, build with 32-bit configuration disabled:

- **mkdir gcc_build**
- **cd gcc_build**
- **../configure --prefix=/home/bdallen/work/local --disable-multilib**
- **make**
- **make install**

Get and install Boost 1.4.9+:

- **wget https://sourceforge.net/projects/boost/files/boost/1.49.0/boost_1_49_0.tar.b**
- **tar jxf boost_1_49_0.tar.bz2**
- **./bootstrap.sh --prefix=/home/work/local_boost**
- **./b2**
- **./b2 install**
- **export BOOST_ROOT=/home/bdallen/work/local_boost**
- **export LD_LIBRARY_PATH=/home/bdallen/work/local_boost/lib // for liblightgrep'**

install autoconf:

liblightgrep command `autoreconf -fi` requires autoconf 2.64+. Also, the beginning of configure.ac needs hacked to accept the unused default version of autoconf:

- mkdir autotools; cd autotools
- wget ftp://ftp.gnu.org/pub/gnu/autoconf/autoconf-2.64.tar.gz
- tar -xvzf autoconf-2.64.tar.gz
- cd autoconf-2.64
- mkdir ~/work/autotools
- ./configure --prefix=/home/bdallen/work/autotools
- make
- make install

Install *Liblightgrep*

Ref. https://github.com/strozfriedberg/liblightgrep/blob/master/BUILD.md

- git clone --recursive git://github.com/LightboxTech/liblightgrep.git
- cd liblightgrep
- // Hack line 3 of configure.ac to accept 2.63 instead of 2.64.
- ~/work/autotools/bin/autoreconf -fi   // still use the newly generated 2.64
- // Also change thread.h:38 and 40 to not call constructor from constructor.
- export PKG_CONFIG_PATH=/home/bdallen/work/local/lib/pkgconfig
- ../configure CC=/home/bdallen/work/local/bin/gcc
- CXX=/home/bdallen/work/local/bin/c++
- LDFLAGS=-L/home/bdallen/work/local/lib
- --prefix=/home/bdallen/work/local

Install *Lightgrep Wrapper*

- git clone https://github.com/NPS-DEEP/lightgrep_wrapper.git
- ./bootstrap.sh
- mkdir build
- cd build
- ../configure --prefix=/home/bdallen/work/local
- make
- make install

Lightgrep Wrapper is now installed.

Install *be_scan Library* To install the *be_scan Library*, which uses *Lightgrep Wrapper*, on CentOS 6.8:

- git clone https://github.com/NPS-DEEP/be_scan.git
- cd be_scan
- ./bootstrap.sh
- mkdir build; cd build
- ../configure --prefix=/home/bdallen/work/local
- make
- make install // before make check for python tests
- export LD_LIBRARY_PATH=/home/bdallen/work/local/lib64   // so
- libbe_scan.so can find GCC's libstdc++.so.6
- make check

7.7 Building *bulk_extractor* on CentOS 6.8

This section is about building *bulk_extractor* on CentOS 6.8. *bulk_extractor* does not use *Lightgrep Wrapper*. This section is included solely to assist users in building *bulk_extractor* so that the *be_scan Library* can be validated against *bulk_extractor* output in this environment.

First Install TRE:

- mkdir tre
- cd tre
- wget http://laurikari.net/tre/tre-0.8.0.tar.bz2
- tar jxf tre-0.8.0.tar.bz2
- cd tre-0.8.0
- mkdir build; cd build
- ../configure --prefix=/home/bdallen/work/local

Then configure *bulk_extractor*. Ensure that environment variables are set up:

- export BOOST_ROOT=/home/bdallen/work/local_boost
- export LD_LIBRARY_PATH=/home/bdallen/work/local/lib:
- /home/bdallen/work/local/lib64
- export PKG_CONFIG_PATH=/home/bdallen/work/local/lib/pkgconfig

Fix `bulk_extractor/configure.ac` to find libraries that lightgrep needs by placing new line

```
LIBS="-licuuc -licudata $LIBS"
```

before line

```
if test x"$lightgrep" == x"yes"; then
```

Build *bulk_extractor* with *Liblightgrep* instead of *flex*:

```
■ sh bootstrap.sh
■ mkdir build; cd build
■ ../configure --prefix=/home/bdallen/work/local
■              --disable-flexscanners --enable-lightgrep
■ make
```

At this point, you might run `bulk_extractor` from `build/src/bulk_extractor` rather than installing it with `make install`, for example:

```
■ ~/work/gits/bulk_extractor/build/src/bulk_extractor -o z170731
■                                   AE1001-1004.E01_1470083869.raw
```

8 Licence

Code in the *BE Scan Toolset* is provided with the following notice:

The software provided here is released by the Naval Postgraduate School, an agency of the U.S. Department of Navy. The software bears no warranty, either expressed or implied. NPS does not assume legal liability nor responsibility for a User's use of the software or the results of such use.

Please note that within the United States, copyright protection, under Section 105 of the United States Code, Title 17, is not available for any work of the United States Government and/or for any works created by United States Government employees. User acknowledges that this software contains work which was created by NPS government employees and is therefore in the public domain and not subject to copyright.

However, because hashdb includes source modules, the compiled hashdb executable may be covered under a different copyright.

`liblightgrep` is Copyright (C) 2010-2015, Stroz Friedberg, LLC under version 3 of the GNU Public License.

9 Conclusions

The *BE Scan Toolset* provides interfaces suitable for recursively scanning for digital forensic artifacts by processing buffer streams. The *artscan.py Tool* and *spark_be_scan Infrastructure* validate usability of these interfaces by producing forensic artifact output.

10 Future Work

Future work is in two areas:

- **Performance timing**
  Formal performance measurments need to be made and presented.
- **More capability**
  More scanners and decompressors should be added in order to increase the set of artifacts captured.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California