hashdb 3.1.0

USERS MANUAL

May 19, 2017

Authored by:
Bruce D. Allen
Jessica R. Bradley
Simson L. Garfinkel

Contents

1 Introduction

1.1 Overview of *hashdb*

hashdb is a tool that can be used to find data in raw media using cryptographic hashes calculated from blocks of data. It is a useful forensic investigation tool for tasks such as malware detection, child exploitation detection or corporate espionage investigations. The tool provides several capabilities that include:

- Creating hash databases of MD5 block hashes.
- Importing block hash values.
- Scanning the hash database for matching hash values.
- Providing source information for hash values.

Using *hashdb*, a forensic investigator can take a known set of blacklisted media and generate a hash database. The investigator can then use the hash database to search against raw media for blacklisted information. For example, given a known set of malware, an investigator can generate a sector hash database representing that malware. The investigator can then search a given corpus for fragments of that malware and identify the specific malware content in the corpus.

hashdb relies on block hashing rather than full file hashing. Block hashing provides an alternative methodology to file hashing with a different capability set. With file hashing, the file must be complete to generate a file hash, although a file carver can be used to pull together a file and generate a valid hash. File hashing also requires the ability to extract files, which requires being able to understand the file system used on a particular storage device. Block hashing, as an alternative, does not need a file system or files. Artifacts are identified at the block scale (usually 512 bytes) rather than at the file scale. While block hashing does not rely on the file system, artifacts do need to be sector-aligned for *hashdb* to find hashes [3].

hashdb provides an advantage when working with hard disks and operating systems that fragment data into discontiguous blocks yet still sector-align media. This is because scans are performed along sector boundaries. Because *hashdb* works at the block resolution, it can find part of a file when the rest of the file is missing, such as with a large video file where only part of the video is on disk. *hashdb* can also be used to analyze network traffic (such as that captured by **tcpflow**). Finally, *hashdb* can identify artifacts that are sub-file, such as embedded content in a `.pdf` document.

hashdb stores cryptographic hashes (along with their source information) that have been calculated from hash blocks. It also provides the capability to scan other media for hash matches. This manual includes uses cases for the *hashdb* tools, including usage with **Autopsy**, *SectorScope*, **bulk_extractor**, and the *hashdb* Python and C++ libraries, and demonstrates how users can take full advantage of all of its capabilities.

1.2 Intended Audience

This Users Manual is intended to be useful to new, intermediate and experienced users of *hashdb*. It provides an in-depth review of the functionality included in *hashdb* and

shows how to access and utilize features through command line operation of the tool. This manual includes working examples with links to the input data used, giving users the opportunity to work through the examples and utilize all aspects of the system. This manual also introduces Forensic tools that use *hashdb*.

For developers, this manual provides in-depth coverage of the data syntax used by *hashdb* and for interfacing with *hashdb* using the *hashdb* **c++** and **Python** interfaces.

1.3 *hashdb* Resources

Users are encouraged to visit the *hashdb* Wiki page at `https://github.com/NPS-DEEP/hashdb/wiki` for quick links to downloads, documentation, and examples.

All *hashdb* users should join the **bulk_extractor** users Google group for more information and help with any issues encountered. To join, send an email to **bulk_extractor-users+subscribe@ googlegroups.com**.

Several articles are available related to block hashing, and its practical and research applications. Some of those articles are specifically cited throughout this manual. Here are some additional references we recommend:

- Michael McCarrin, Bruce Allen. Rapid Recognition of Blacklisted Files and Fragments. Naval Postgraduate School. `http://www.osdfcon.org/presentations/2015/McCarrin-Allen_osdfcon.pdf`.
- Jim Jones, Tahir Khan, Kathryn Laskey, Alex Nelson, Mary Laamanen, Doug White. Inferring Past Activity from Partial Digital Artifacts. George Mason University, National Institute of Standards and Technology. `http://www.osdfcon.org/presentations/2015/Jim-Jones_EtAl-Release.pdf`.
- Simson Garfinkel, Michael McCarrin. Hash-based Carving: Searching media for complete files and file fragments with sector hashing and hashdb. DFRWS 2015 USA. `http://www.sciencedirect.com/science/article/pii/S1742287615000468`
- Joel Young, Kristina Foster, Simson Garfinkel, Kevin Fairbanks. Distinct Sector Hashes for Target File Detection. `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6311397`.
- Garfinkel, Simson, Alex Nelson, Douglas White and Vassil Rousseve. Using purpose-built functions and block hashes to enable small block and sub-file forensics. Digital Investigation. Volume 7. 2010. Page S13–S23. `http://www.dfrws.org/2010/proceedings/2010-302.pdf`.
- Foster, Kristina. Using Distinct Sectors in Media Sampling and Full Media Analysis to Detect Presence of Documents From a Corpus. Naval Postgraduate School Masters Thesis, September 2012. `http://calhoun.nps.edu/public/handle/10945/17365`.

1.4 Conventions Used in this Manual

This manual uses standard formatting conventions to highlight file names, directory names and example commands. The conventions for those specific types are described

in this section.

Names of programs including the post-processing tools native to *hashdb* and third-party tools are shown in **bold**, as in **bulk_extractor**.

File names are displayed in a fixed width font. They will appear as `filename.txt` within the text throughout the manual.

Directory names are displayed in italics. They appear as *directoryname/* within the text. The only exception is for directory names that are part of an example command. Directory names referenced in example commands appear in the example command format.

Database names are denoted with bold, italicized text. They are always specified in lower-case, because that is how they are referred in the options and usage information for *hashdb*. Names will appear as ***databasename***.

This manual contains example commands that should be typed in by the user. A command entered at the terminal is shown like this:

■ `command`

The first character on the line is the terminal prompt, and should not be typed. The black square is used as the standard prompt in this manual, although the prompt shown on a users screen will vary according to the system they are using.

1.5 Changes Over the *hashdb* v3.0.0 Release

hashdb Version 3.1.0 provides a halting bug fix that manifests when building large databases. The halt is a result of heap space exhaustion due to excessive page allocations in LMDB resulting from increasing record sizes on existing records. *hashdb* v2.0.0 does not increase record size and does not manifest this bug. This bug results in program termination. The fix replaces changed records in-place without changing record size. To accomplish this, we change the data store as follows:

- *hashdb* no longer stores source offset values in the hash data store. These values were used to identify where a block hash is located within a source. This information may be obtained by reexamining the source file. As a result of this change, the byte alignment parameter seen when creating a new database is no longer required and is discontinued.

- *hashdb* no longer stores hash suffix values in a list in the hash store. Instead, the hash prefix is longer and is hard-coded to 7 bytes. In a database of one billion hashes, this will result in a false positive rate of one in 72 million. Recall that the hash store is an approximate store and that complete hashes are stored in the hash data store.

1.6 Changes Over the *hashdb* v2.0.1 Release

hashdb Version 3 provides significant functional and performance improvements over v2.0.1:

- False positive block matches may be evaluated because metadata about hashes and sources are now being stored:
  - Block labels and block entropy values indicate characteristics about data blocks.
  - Source type, zero count, and nonprobative count of a source indicate the density of useful blocks within a source.
- Sources are now tracked by source hash rather than by name. This fixes two problems:
  - By not storing duplicates, source relevance and similarity between sources may be weighed.
  - Groups of identical sources are readily identified.
- Bulky output from scans has been significantly reduced:
  - Information is returned in the more condensed JSON format rather than in XML.
  - Source offsets are presented as lists in one record rather than repeating hash and source information for each offset.
  - Additionally, an optimization mode is available where information about matched sources and hashes are returned only once and are not reprinted if a source or hash is matched again.
- A complete *hashdb* API is now available for C++ and Python.
  - A scan interface supports scan functions and functions for reading all hash and source information.
  - An import interface supports functions for importing hash and source information.
  - Additional interfaces support access to settings and higher-layer capabilities.
- The database has been retuned to improve scan and import speed:
  - A compressed hash store has been added for extremely fast and compact approximate scan lookups.
  - The Bloom filter has been removed in favor of the dense hash store.
  - The hash data store contains lists of source offsets for each source rather than one entry per source offset, reducing its size.
  - Several scan modes are available, supporting various levels of verbosity and performance:
    - **expanded** scans for matches and returns complete match information in JSON format.
    - **expanded optimized** scans for matches and returns complete match information in JSON format but matched sources and hashes are cached so that information is not reprinted in other matches.
    - **count** only returns a match count and does not take time to parse match information into a data structure.

    * **approximate count** is fast because it does not read the hash information store when there is a match, but it can have false positives in its matching and in its count.

- *hashdb* can now read media images, scan media images, and ingest sources directly. **bulk_extractor** is no longer required to perform these functions.
- The build process has been restructured to support parallel build trees (VPATH builds). The goal is to support compiling to additional targets such as the ARM processor.

1.7 Licensing

hashdb code is provided with the following notice:

1.8 Obtaining *hashdb*

The *hashdb* tool and API interface library are readily available for Windows systems, Linux flavors, and MacOS. A Windows installer is available for Windows users. A source code distribution is available for Linux and Mac users. Developers may download *hashdb* directly from source available on GitHub.

Steps for installing *hashdb* on Windows and one flavor of Linux are described here. For more installation options, please refer to the installation page on the *hashdb* Wiki at `https://github.com/NPS-DEEP/hashdb/wiki/Installing-hashdb`.

For information on installing *SectorScope* and **bulk_extractor** tools which use *hashdb*, Please see **Section 4**.

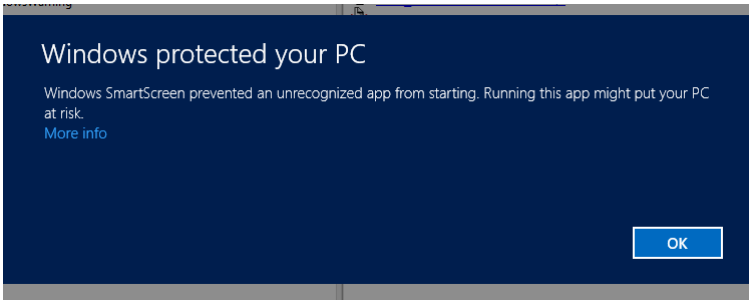

Figure 1: Windows 8 warning when trying to run the installer. Select "More Info" and then "Run Anyway."

1.8.1 Installing on Windows

Windows users should download the Windows Installer for *hashdb*. The file to download is located at `http://digitalcorpora.org/downloads/hashdb` and is called `hashdb-x.y.z-windowsinstaller.exe` where x.y.z is the latest version number.

You should close all Command windows before running the installation executable. Windows will not be able to find the *hashdb* tools in a Command window if any are open during the installation process. If you do not do this before installation, simply close all Command windows after installation. When you re-open, Windows should be able to find *hashdb*.

Next run the `hashdb-x.y.z-windowsinstaller.exe` file. This will automatically install *hashdb* on your machine. Some Windows safeguards may try to prevent you from running it. Figure 1 shows the message Windows 8 displays when trying to run the installer. To run anyway, click on "More info" and then select "Run Anyway."

When the installer file is executed, the installation will begin and show a dialog like the one shown in Figure 2. Users should select all options needed:

- **hashdb tool**
  Installs the *hashdb* tool into the `Program Files` directory and installs the Users Manual shortcut in the `Start` menu.
- **Add to PATH**
  Appends the path to the *hashdb* tool to the System `PATH` variable so that it can be found at the command prompt and by other tools.
- **hashdb Python module**
  Installs the following files onto the desktop at `Users\Public\Desktop`:
  - `hashdb.py`
    The *hashdb* Python interface file.
  - `_hashdb.pyd`
    The `.dll` file needed by `hashdb.py`.

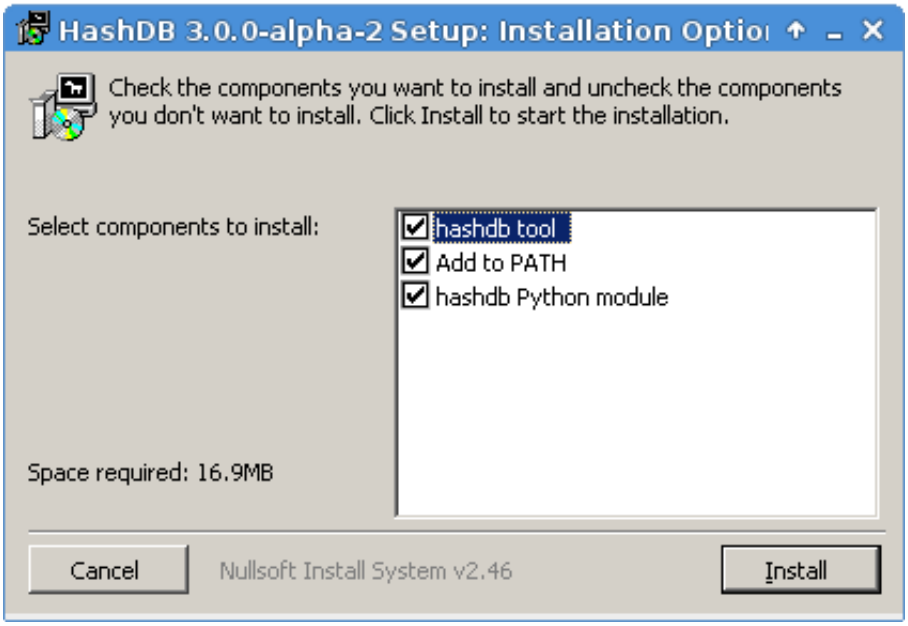

Figure 2: Dialog appears when the user executes the Windows Installer. Select the default configuration to install all components.

- `test_hashdb_module.py`
  A small test program for helping to validate and diagnose the installation of `hashdb.py` and `hashdb.py`. This file may be delted.

Suggestions for managing these files placed on the public desktop include:

- Move them to your working directory so that they can be found by your Python program.
- Move them to another directory and set `PATH` to include the path to `_hashdb.pyd` and set `PYTHONPATH` to include the path to `hashdb.py`.

hashdb is now installed on your system can be run from the command line.

1.8.2 Installing on Linux or Mac

This section describes steps for installing *hashdb* on a Fedora system and is intended to illustrate the installation process. For steps on installing *hashdb* to other flavors of Linux or for MacOS, and for installing specific configurations, please refer to the installation page on the *hashdb* Wiki at `https://github.com/NPS-DEEP/hashdb/wiki/Installing-hashdb`.

Before compiling *hashdb* for your platform, you may need to install other packages on your system which *hashdb* requires to compile cleanly and with a full set of capabilities.

Dependencies
The following commands should add the requisite packages:

- `sudo dnf update`
- `sudo dnf groupinstall development-tools`

- `sudo dnf install gcc-c++`
- `sudo dnf install openssl-devel`
- `sudo dnf install libewf-devel`
- `sudo dnf install bzip2-devel`
- `sudo dnf install swig`
- `sudo dnf install python-devel`

Download and Install *hashdb*
Next, download the latest version of *hashdb*. The software can be downloaded from `http://digitalcorpora.org/downloads/hashdb/`. The file to download is `hashdb-x.y.z.tar.gz` where x.y.z is the latest version.

After downloading the file, un-tar it by either right-clicking on the file and choosing "extract to...' or typing the following at the command line:

- `tar -xvf hashdb-x.y.z.tar.gz`

Then, in the newly created *hashdb-x.y.z* directory, run the following commands to install *hashdb* in */usr/local/bin* (by default):

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

hashdb is now installed on your system and can be run from the command line.

Note: sudo is not required. If you do not wish to use sudo, build and install *hashdb* in your own space at "$HOME/local" using the following commands:

- `./configure --prefix=$HOME/local/ --exec-prefix=$HOME/local CPPFLAGS=-I$HOME/local/include/ LDFLAGS=-L$HOME/local/lib/`
- `make`
- `make install`

Run *hashdb*
When installed as administrator, the *hashdb* tool should automatically be accessible. When installed as a user, the *hashdb* tool can be made available by typing:

- `export PATH=$HOME/local/bin:$PATH`

Import the Python *hashdb* Module
To use the Python *hashdb* module, your shell must have access to the installed `python.py` and `_python.so` resources.

When installed as administrator, the *hashdb* Python interface can be made available by typing:

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

When installed as a user, the *hashdb* Python interface can be made available by typing:

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

1.8.3 Quickstart Guide

The following steps provide a very brief introduction to running your new installation of *hashdb*. Steps include creating a demo database and scanning for matching hashes.

1. Navigate to the directory where you would like to create a hash database. Then, to run *hashdb* from the command line, type the following instructions:

   - `hashdb create demo.hdb`

   In the above instructions, `demo.hdb` is the empty database that will be created with default database settings.

2. Next, import data into the database. In this example, lets import hashes from the Kitty Material demo dataset available at `http://digitalcorpora.org/corpora/scenarios/2009-m57-patents/KittyMaterial`. But rather than downloading these files and ingesting them, lets just import the pre-made `KittyMaterial.json` data available at `http://digitalcorpora.org/downloads/hashdb/demo/KittyMaterial.json`. After downloading this, type the following:

   - `hashdb import demo.hdb KittyMaterial.json`

   This command, if executed successfully, will print processing status followed by statistics indicating changes to the database.

3. Next, scan a media image for matching hashes. In this example, lets scan the demo media image available at `http://digitalcorpora.org/corpora/scenarios/2009-m57-patents/drives-redacted/jo-favorites-usb-2009-12-11.E01` which contains blacklist block hashes from the Kitty demo:

   - `hashdb scan_media demo.hdb jo-favorites-usb-2009-12-11.E01`

   With this media and dataset, the first block hash matched is at offset `2543104` for hash `1d7379fd4d5cf676a9d4de1e48337e71`:


```
2543104        1d7379fd4d5cf676a9d4de1e48337e71        {"block_hash":
"1d7379fd4d5cf676a9d4de1e48337e71","k_entropy":4880,
"block_label":"","count":1,"source_list_id":1193146442,"sources":
[{"file_hash":"1dd00f2e51aeebe7541cea4ade2e20b5","filesize":1549288,
"file_type":"","zero_count":0,"nonprobative_count":10,"name_pairs":
["/home/bdallen/KittyMaterial","/home/bdallen/KittyMaterial/
HighQuality/DSC00003.JPG"]}],"source_sub_counts":
["1dd00f2e51aeebe7541cea4ade2e20b5",1]}
```


2 How *hashdb* Works

The *hashdb* tool provides capabilities to create, edit, access and search databases of cryptographic hashes created from hash blocks. The cryptographic hashes are imported into a database from a directory, another database, **bulk_extractor** or JSON data, or trough the *hashdb* API. Once a databases is created, *hashdb* provides users with the capability to scan the database for matching hash values and identify matching content. Hash databases can be exported, added to, subtracted from and shared.

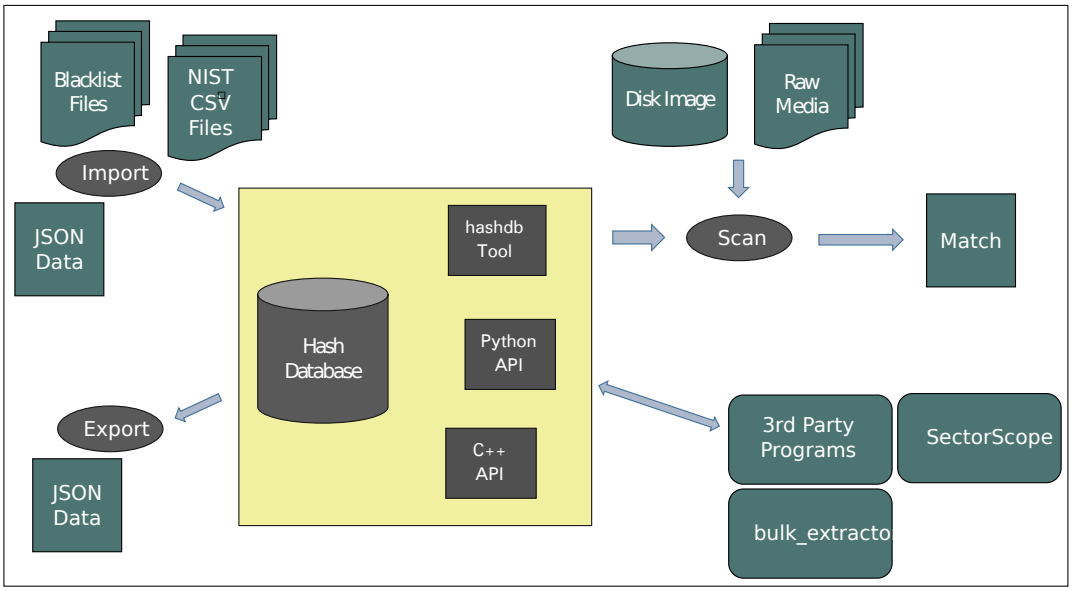

Figure 3: Overview of the *hashdb* system

Figure 3 provides an overview of the capabilities included with the *hashdb* tool. *hashdb* populates databases from whitelist source files or other media provided in JSON format or through the API. Users can add or remove data from the database after it is created. Once the database is populated, *hashdb* can export content from the database in JSON format. It also provides an API that can be used by third party tools (as it is used in the **bulk_extractor** program) to create, populate and access hash databases.

2.1 Block Hash

hashdb works by matching hashes calculated from blocks of data. *hashdb* is different from tools that match files because it can find matches even when part of a file is missing or changed. *hashdb* stores and scans for hashes created from contiguous blocks of data. We call the size of the block hashed the *block size*. *hashdb* stores and scans for hashes in step increments along a hash interval. Blocks hashed at step-sized intervals are illustrated in Figure 4.

As an optimization, *hashdb* provides a byte alignment setting. The byte alignment value must be divisible by the step size. The default configuration with 512 for step size, block size, and byte alignment is shown in Figure 5. Byte alignment is described in **subsection 2.12**.

2.2 Blacklist Data

Blacklist data is the data we scan against to determine whether forensic data contains probative artifact. We build a hash database of blacklist data by importing block hashes from blacklist files, copying from other hash databases, or importing from other sources

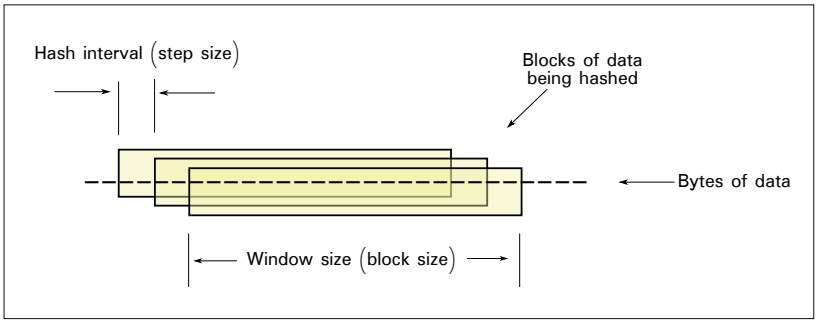

Figure 4: Data blocks are hashed along an interval of bytes

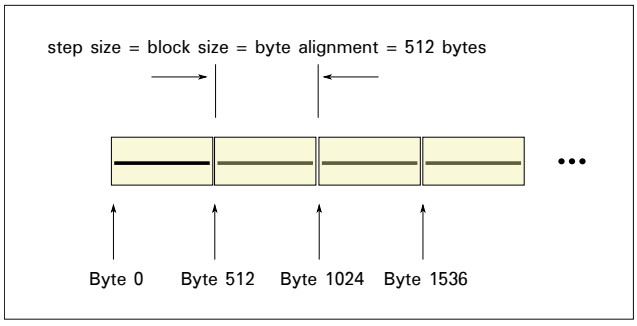

Figure 5: By default, hashes are calculated from 512 byte blocks of data along 512 byte intervals and the database uses a byte alignment of 512

using data prepared in JSON format.

Each block hash in the database includes a total count value of how many times the block has been identified in sources as well as sub-count values indicating how many times the block has been identified by each contributing source. If a block is found several times for a source, then several sub-count values will be recorded for that source. Block hashes associated with many sources tend to contain non-probative data.

2.3 Repository Names

Blacklist data may come from multiple sources called "repositories". *hashdb* tracks repository names in order to know what categories blacklist data belongs to. When importing into a database, users may provide repository names specific to balcklist categories or cases, or allow *hashdb* to select default values. When scanning, hashes may match sources from several repositories.

2.4 Forensic Data

Forensic data is the data we scan to see if it contains artifact matching that in our hash database. Note that just having matches is not sufficient to be considered probative. Some matches are common to many files. *hashdb* tracks entropy and data information to automate the process of eliminating many false positives. Direct analysis such as that provided by the *SectorScope* tool may be used to see the exact content at that location. *SectorScope* is available at `https://github.com/NPS-DEEP/NPS-SectorScope/wiki`.

2.5 Recursive Extraction

The *hashdb* `ingest`, `scan_media`, and `read_media` commands support recursive extraction, meaning that they can recursively decompress compressed content. For `ingest`, the result is that compressed source data is uncompressed and submitted as a new file to be ingested. For `scan_media`, the result is that compressed media is recursively uncompressed and scanned. For `read_media`, the media image offset is recursively interpreted and the uncompressed content is returned. *hashdb* currently decompresses **zip** and **gzip** encodings.

2.6 Recursion Path

Typically, an offset points directly to a byte in a source file or a media image. But when data is decompressed or recursively decompressed, it includes a recursion path to reach the decompressed data. An offset consists of the following:

- The byte offset into the data, specifically, a source file or media image.
- Zero or more recursion path sequences, from out to in, consisting of:
  - A delimiter (**-**).
  - The uncompression algorithm, such as **zip**.
  - A delimiter (**-**).
  - The byte offset into the uncompressed data.

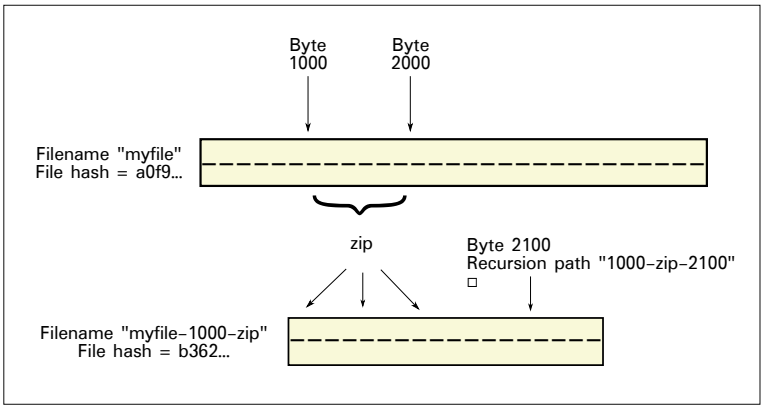

Figure 6: Example of new file `myfile-100-zip` uncompressed from file `myfile`

Example byte offset 2100 at recursion path `1000-zip-2100` within uncompressed data obtained by unzipping data starting at byte 1000 of file `myfile` is shown in Figure 6.

2.7 File Hash

hashdb tracks sources by their file hash rather than by their filename or repository name. This approach provides several benefits:

- The database does not store block hashes from multiple sources when the sources are actually the same file.
- Source filenames and repository names for the same file are grouped together and may be looked up by their file hash value.

2.8 Managing False Positives

A significant problem when scanning for probative blocks is dealing with false positives [1]. False positives arise from data that is easily generated or commonly duplicated such as sparse data or lookup tables. *hashdb* records and uses information about blocks and sources in order to identify blocks as nonprobative. Then, post-processing tools such as *SectorScope* can readily evaluate matched blocks with these false positives removed.

Here we describe the data that *hashdb* stores with hashes and sources. How this data is used to classify blocks as nonprobative is a complex issue. *hashdb* stores this data. It is up to post-processing tools such as *SectorScope* to evaluate it.

Data stored with sources:

- **File Hash**
  Matched source files are indexed by file hash (new to *hashdb* v3). The *SectorScope* tool uses this value to visualize how specific source files are distributed across a media image.
- **File Size**
  The source file size indicates how big a source file is. The *SectorScope* tool uses this value to know what percentage of a source file is matched in a scan.

- **File Type**
  This field stores information about the type of the source file. This field is not used by the *hashdb* Tool but is available through the *hashdb* API interfaces for classifying the file type.

- **File Zero Count**
  The zero count indicates the number of blocks in the source consisting completely of the 0 byte. These blocks are skipped by the *hashdb* `scan_stream` and `ingest` commands and are not imported into the database or scanned for.

- **File Nonprobative Count**
  The nonprobative count indicates how many blocks of the source file are deemed nonprobative. In the **bulk_extractor** *hashdb* scanner import function and in the *hashdb* tool `ingest` function, this value is set to the number of blocks that have been given a block label, indicating that the block is likely nonprobative.

- **Name Pairs**
  The name pairs identify the list of all source repository name and filename pairs associated with a source file as identified by the source file hash. When scanning, this list provides a comprehensive indication of what a hash match is a member of.

Data stored with hashes:

- **Block Hash**
  The block hash is the hash value calculated from a block of data. *hashdb* databases are populated with block hash values from sources. When scanning, block hash values are calculated from media images and are scanned for in a *hashdb* database.

- **Entropy**
  We calculate the entropy of data blocks and use this value to help estimate that the block may be nonprobative. Blocks with a low entropy value are often nonprobative. *hashdb* calculates the Shannon entropy of blocks using an alphabet of 2^{16} values. *hashdb* provides entropy as `k_entropy`, entropy scaled up by $1,000$ so that it can be managed as an integer. Divide `k_entropy` by $1,000$ to obtain actual entropy with three decimal place precision.

  Data in blocks can be a member of many types of alphabets, for example readable text or executable code. For improved results, we recommend considering the type of data along with the calculated entropy when estimating that a block may be nonprobative.

- **Block Label**
  Block labels may be used to hold information about the nature of the block. For example it might be used to indicate that the byte values increment, indicating a homogeneous data structure [1].

- **Count**
  The count indicates the total number of times the block has been seen in sources. High count values are likely to be nonprobative.

- **Source Sub-counts**
  The list of source sub-count information provides information for each source related to the block:

- The **file hash** of the associated block.
- The **sub-count** of offsets contributed by the given source.

2.9 Building a *hashdb* Database

There are several ways to populate a database:

- Using the *hashdb* `import` command.
- Importing from correctly formatted JSON data.
- Importing from another database.
- Using the **bulk_extractor** *hashdb* scanner.
- Using the *hashdb* library through the Python or C++ interface.

A database may contain blacklist hashes from multiple source domains, where a domain is called a *repository*. The repository name indicates the provenance of the dataset. It is its description information, such as "Company X's intellectual property files".

2.10 Scanning

There are multiple ways users can scan for matches in a block hash database:

- Using one of the *hashdb* tool scan commands to scan from a media image, list, stream, or specific hash.
- Using one of the *hashdb* library Python or C++ scan interfaces.
- Using the **bulk_extractor** *hashdb* scanner Scan function.

Additionally, there are several output modes for receiving scan matches. These modes provide varying levels of detail and speed.

2.11 Contents of a Hash Database

Each *hashdb* database is contained in a directory called <*databasename*>*.hdb* and contains a number of files. These files are:

```
lmdb_hash_data_store/data.mdb
lmdb_hash_data_store/lock.mdb
lmdb_hash_store/data.mdb
lmdb_hash_store/lock.mdb
lmdb_source_data_store/data.mdb
lmdb_source_data_store/lock.mdb
lmdb_source_id_store/data.mdb
lmdb_source_id_store/lock.mdb
lmdb_source_name_store/data.mdb
lmdb_source_name_store/lock.mdb
log.txt
settings.json
```

These files include several data store directories and files, a settings file, and a log file:

Listing 1: An example `log.xml` log file showing a database creation entry and a datase ingest entry

```
# command: "hashdb create KittyMaterial.hdb"
# hashdb version: 3.1.0-alpha1
# username: bdallen
# start time 2017-05-18T00:06:54Z
{"name":"begin","delta":"0.000494","total":"0.000494"}
{"name":"end","delta":"0.000014","total":"0.000512"}
# command: "hashdb ingest KittyMaterial.hdb ../KittyMaterial"
# hashdb version: 3.1.0-alpha1, GIT commit: v3.0.0-9-g17ed5eb-dirty
# username: bdallen
# start time 2017-05-18T00:07:13Z
{"name":"begin","delta":"0.000566","total":"0.000566"}
# hashdb changes:
#     hash_data_inserted: 401732
#     hash_inserted: 401598
#     hash_count_changed: 88
#     hash_count_not_changed: 46
#     source_data_inserted: 88
#     source_data_changed: 88
#     source_id_inserted: 88
#     source_id_already_present: 401820
#     source_name_inserted: 88
{"name":"end","delta":"8.453844","total":"8.454414"}
```

- `lmdb store` files
  The `lmdb store` files encode all the block hashes, source files, and related information that are in the database. These filenames start with the prefix `lmdb`.

- `settings.json`
  This file contains the settings requested by the user when the block hash database was created. Database settings are described in **subsection 2.12**. This file also contains the internal *hashdb* settings version used to help *hashdb* identify whether a database is compatible with this version of *hashdb*. The `settings.json` file with the default settings looks like this:

  ```
  {"settings_version":3, "block_size":512}
  ```

- `log.txt`
  Every time a command is run that changes the content of the database, information about the change is appended to this log. Each entry includes the command name, information about *hashdb* including the command typed and how *hashdb* was compiled, information about the operating system *hashdb* was just run on, timestamps indicating how much time the command took, and the specific *hashdb* changes applied.

  Listing 1 shows an example log file containing two entries, one for when the hash database was created, and one for when data was ingested into the database.

- `timestamp.json`
  `timestamp.json` is not formally part of the *hashdb* database. It is created by the *hashdb* tool performance analysis commands described in **subsection 3.6**.

This file is replaced rather than appended to. Timestamp syntax is described in **Section 6**.

2.12 Database Settings

The following database settings are available:

- **Settings Version**
  This hardcoded value identifies the database version.
- **Block size**
  The size of data blocks the database expects to store. Block hashes are calculated from data of this size. The default is 512. *hashdb* does not enforce correct block size when importing using the `import` and `import_tab` commands.

2.13 Maintaining Database Integrity

A *hashdb* hash database can be damaged when operations that modify it are aborted. Re-running the operation may not fully add missing data. Although some data may be lost, the database should remain operational.

A *hashdb* hash database can also be damaged by running a command that should not have been run such as ingesting incorrect files or adding an incorrect database. Some operations can be "rolled out" using database manipulation commands.

Each *hashdb* hash database includes an audit log file that records all commands issued that modify that database. You may inspect this audit log to verify that all issued commands are acceptable and that all issued commands have completed. Audit log files are described in **subsection 2.11**.

Please backup databases that cannot readily be recreated.

3 Running the *hashdb* Tool

The core capabilities provided by *hashdb* involve creating and maintaining a database of hash values and scanning media for those hash values. To perform those tasks, *hashdb* users need to start by building a database (if an existing database is not available for use). Users then import hashes using *hashdb* tool commands, the *hashdb* **bulk_extractor** scanner, or through the *hashdb* library API, and then possibly merge or subtract hashes to obtain the desired set of hashes to scan against. Users then scan for hashes that match. Additional commands are provided to support statistical analysis, performance tuning and performance analysis.

This section describes use of the *hashdb* tool commands, along with examples, for performing these tasks. For more examples of command usage, please see **section 5**. For a *hashdb* quick reference summary, please see **Appendix A**, also available at `http://digitalcorpora.org/downloads/hashdb/hashdb_quick_reference.pdf`.

3.1 Creating a New Hash Database

A hash database must be created before hashes can be added to it. Syntax for creating a hash database is shown in Table 1. Configurable settings associated with the database is shown in Table 2 and described in **Subsection 2.12**.

Table 1: Command for Creating Hash Databases

Command	Usage	Description
create	`create [-b <block size>] <hashdb.hdb>`	Creates a new hash database.

Table 2: Database Settings

Option	Verbose Option	Specification
`-b`	`--block_size=`*block_size*	Specifies the block size in bytes used to generate the hashes that will be stored and scanned against. Default is 512 bytes.

3.1.1 `create`

Create a new hash database configured with provided or default settings.

Example
To create an (empty) hash database named `demo.hdb`, type the following command:

- `hashdb create demo.hdb`

The above command will create a database with all of the default hash database settings. Most users will not need to change these settings. Users can specify either the option and value or the verbose option value for each parameter along with the create command, as in:

- `hashdb create --block_size=4096 demo.hdb`
- `hashdb create -b 4096 demo.hdb`

The above two commands produce identical results, creating the database `demo.hdb` to expect a block size of 4096 bytes.

3.2 Importing and Exporting

Hash databases may be imported to in several ways. Syntax for commands that import and export hashes is shown in Table 3. Import and export options are shown in Table 4.

Note that there are other ways to populate a database besides these listed here, including using other hash databases (discussed in **subsection 5.4**), by using the **bulk_extractor** *hashdb* scanner (discussed in **subsection 5.3**), and through the use of the import capability provided by the *hashdb* library API (discussed in **subsection 5.2**).

Table 3: Commands for Importing into and Exporting Hash Databases

Command	Usage	Description
ingest	`ingest [-r <repository name>] [-w <whitelist.hdb>] [-s <step size>] [-x <rel>] <hashdb.hdb> <source directory>`	Computes and ingests block hashes from files under the source directory into the hash database as directed by options.
import_tab	`import_tab [-r <repository name>] <hashdb.hdb> <tab.txt>`	Imports values from the tab-delimited file into the hash database. This command accepts a dash (-) as a filename to allow terminal streaming from `stdin`.
import	`import <hashdb.hdb> <hashdb.json>`	Imports values from the JSON file into the hash database. This command accepts a dash (-) as a filename to allow terminal streaming from `stdin`.
export	`export [-p <begin:end>] <hashdb.hdb> <hashdb.json>`	Exports the hash database to the JSON file. This command accepts a dash (-) as a filename to allow terminal streaming to `stdout`.

3.2.1 `ingest`

The `ingest` command computes and ingests hashes from files under the source directory, including files in subdirectories. Files with `.E01` extensions are treated as E01 files. If some of the content to be ingested already exists, specifically, if block hashes have already been ingested for a given file hash, it will not be ingested again, but the filename and repository name will be stored to cite the source reference.

Example
To import block hashes from a directory of blacklist sources, type the following command:

- `hashdb ingest -r demo_repository demo.hdb demo_blacklist_dir`

Table 4: Options for Importing and Exporting Hash Databases

Option	Verbose Option	Specification
`-r`	`--repository_name=`*repository name*	Specifies the name to associate the imported hashes with. If not provided, the source filename entered is used as the repository name.
`-w`	`--whitelist_dir=`*whitelist directory*	If a whitelist database is provided, matching hashes are marked with `w` in their block label.
`-s`	`--step_size=`*step size*	The increment to step along for calculating block hashes. The step size must be compatible with the byte alignment defined in the database, specifically the byte alignment must be divisible by the byte alignment.
`-x`	`--disable_processing=rel`	Use this option to disable specific processing, specifically: `r` disables recursively processing embedded data, `e` disables calculating block entropy, and `l` disables calculating block labels.
`-p`	`--part_range=`*begin:end*	Use this option to select a range of block hashes by hexadecimal value rather than selecting all block hashes.

In the above command the option **-r** is used along with the repository name `demo_repository` to indicate the repository source of the block hashes being imported into the database. The repository name is used to keep track of the sources of hashes. By default, the repository name used is the text `repository_` with the filename of the file being imported from appended after it.

The **ingest** command in the above example imports block hashes from files in the `demo_blacklist_dir` directory into the database `demo.hdb`. When the Kitty Material demo dataset available at `http://digitalcorpora.org/corpora/scenarios/2009-m57-patents/KittyMaterial/import` is imported, *hashdb* prints output to the command line to indicate that hashes have been inserted into database `demo.hdb`. Listing 13 shows an

example output of changes from running an ingest command.

Also, database log file `log.txt` is updated to show that a set of hash blocks have just been inserted. The log in Figure 1 was generated from similar **create** and **import** actions. The contents of log files is described in **subsection 2.11**.

Users may prefer to run statistical commands such as this to get information about the contents of the database (and confirm that values were inserted):

- `hashdb size demo.hdb`

3.2.2 `import_tab`

The `import_tab` command imports values from the tab-delimited file into the hash database. Note that tab-delimited files are expected to contain block hashes calculated from 512-byte blocks along 512-byte boundaries. Tab-delimited files are described in **subsection 6.2**.

hashdb checks to see if the source file has already been imported and does not import block hashes from sources imported in previous sessions.

3.2.3 `import`

The `import` command imports values from an exported database. Data is in JSON format as described in **subsection 6.3**. If source information for a block hash is already present, it will not be re-imported.

3.2.4 `export`

The `export` command exports values or a range of values from a *hashdb* block hash database. Data is in JSON format as described in **subsection 6.3**. The following example exports everything in database `demo.hdb` to file, `demo.json`:

- `hashdb export demo.hdb demo.json`

This example exports everything in database `demo.hdb` in two parts:

- `hashdb export -p 00:80 demo.hdb demo_part_1.json`
- `hashdb export -p 80:ffffffffffffffffffffffffffffffff demo.hdb demo_part_2.json`

3.3 Database Manipulation

Databases may need to be merged together or common hash values may need to be subtracted out in order to produce a specific set of blacklist data to scan against. Syntax for commands that manipulate hash databases is shown in Table 5. Destination databases are created if they do not exist yet.

3.3.1 `add`

Add a database to another database.

Table 5: Commands to Manipulate Hash Databases

Command	Usage	Description
add	`add <source db> <destination db>`	Copies all of the hashes from *source db* to *destination db*
add_multiple	`add_multiple <source db1> <source db2> ... <destination db>`	Adds databases *source db1*, *source db2*, etc. to *destination db*
add_repository	`add_repository <source db> <destination db> <repository name>`	Adds *source db* to *destination db* but only when the repository name matches
add_range	`add_range<source db> <destination db> <m:n>`	Copies hash values from *source db* into *destination db* that have source counts within range m and n, inclusive
intersect	`intersect <source db1> <source db2> <destination db>`	Copies hash values common to both *source db1* and *source db2* into *destination db* where sources match
intersect_hash	`intersect_hash <source db1> <source db2> <destination db>`	Copies hash values common to both *source db1* and *source db2* into *destination db* even if their sources are different.
subtract	`subtract <source db1> <source db2> <destination db>`	Copies hash values found in *source db1* but not in *source db2* into *destination db* where sources match
subtract_hash	`subtract <source db1> <source db2> <destination db>`	Copies hash values found in *source db1* but not in *source db2* into *destination db* even if their sources are different.
subtract _repository	`subtract_repository <source db1> <destination db2> <repository namedb>`	Adds *source db1* to *destination db2* unless the repository name matches

3.3.2 `add_multiple`

Add multiple databases into a destination database. This can be faster than using add multiple times because the destination is built in lexicographical order.

3.3.3 `add_repository`

Add a database to another database but only when the repository name matches. Use this to copy everything belonging to a repository to a new database.

3.3.4 `add_range`

Add a database to another database but only when the hash source count falls within the given range. Use this to isolate hashes that appear with a certain frequency or to remove hashes that are too popular.

3.3.5 `intersect`

Add hashes to a destination database when the hash and source are common. Use this to find the intersection between two databases.

3.3.6 `intersect_hash`

Add hashes to a destination database when the hash is common, even if the referenced sources are different. Use this to find hashes that intersect between two databases even if their sources do not intersect.

3.3.7 `subtract`

Add hashes to a destination database when the hash and source is in the first database but not in the second. Use this to ensure that hashes in the second database do not appear in the new destination database.

3.3.8 `subtract_hash`

Add hashes to a destination database when the hash is in the first database but not in the second, even if the referenced sources are different. Use this to ensure that hashes in the second database do not appear in the new destination database even when the sources are different.

3.3.9 `subtract_repository`

Add a database to another database but only when the repository name does not match. Use this to ensure that hashes in the new destination database do not include the repository being subtracted. If information is also contributed from another repository, the information will still be copied but the reference to the removed repository will not be copied.

3.4 Scan Services

hashdb can be used to determine if a file, directory or media image has content that matches previously identified content. This capability can be used, for example, to determine if a set of files contains a specific file excerpt or if a media image contains a video fragment. Forensic investigators can use this feature to search for blacklisted content. Syntax for scan service commands is shown in Table 6. Scan service options are shown in Table 7.

Table 6: Commands that Provide Scan Services

Command	Usage	Description
scan_list	`scan_list [-j e|o|c|a] <hashdb> <hash list file>`	Scans the hashdb for hashes that match hashes in the hash list file and prints out matches
scan_hash	`scan_hash [-j e|o|c|a] <hashdb> <hash value>`	Scans the hashdb for the specified hash value and prints out whether it matches
scan_media	`scan_media [-s <step size>] [-j e|o|c|a] [-x <r>] <hashdb> <media media>`	Scans the hashdb for hashes that match hashes in the media image and prints out matches.

Table 7: Options for Scanning from a Media Image

Option	Verbose Option	Specification
`-s`	`--step_size=`*step size*	The increment to step along for calculating block hashes. The step size must be compatible with the byte alignment defined in the database, specifically the byte alignment must be divisible by the byte alignment.
`-j`	`--json_scan_mode=e|o|c|a`	Select a mode, one of **e**xpanded, expanded **o**ptimized, **c**ount only, **a**pproximate count. Default is **o**.
`-x`	`--disable_processing=r`	Use this option to disable specific processing, specifically: `r` disables recursively processing embedded data.

3.4.1 `scan_list`

Scan for hashes in the list of hashes. List input syntax is described in **subsection 6.7**. Scan output is described in **subsection 6.4**. This command accepts a dash (`-`) as a filename to allow terminal streaming from `stdin`.

3.4.2 `scan_hash`

Scan for the specified hash. The hash to scan for must be provided in hexadecimal format.

3.4.3 `scan_media`

Scan the specified media image for matching hashes.

Example
To scan, first identify the media that you would like to scan. For this example, we download and use the demo media image available at `http://digitalcorpora.org/corpora/scenarios/2009-m57-patents/drives-redacted/jo-favorites-usb-2009-12-11.E01` which contains matching Kitty material.

Then identify the existing hash database that will be used to search for hash value matches. We'll use the database `demo.hdb` that we created from Kitty material in the previous section, containing block hash values calculated from pictures and videos of cats.

Finally, run the *hashdb* scan command to scan for blocks in the media that match block hashes in the database:

■ `hashdb scan_media demo.hdb jo-favorites-usb-2009-12-11.E01 > matches.json`

This command tells *hashdb* to scan media image `jo-favorites-usb-2009-12-11.E01` and try to match the values found in the local database `demo.hdb`, putting match data in file `matches.json`. An example match might look like this:


```
2543104        1d7379fd4d5cf676a9d4de1e48337e71        {"block_hash":
"1d7379fd4d5cf676a9d4de1e48337e71","k_entropy":4880,
"block_label":"","count":1,"source_list_id":1193146442,"sources":
[{"file_hash":"1dd00f2e51aeebe7541cea4ade2e20b5","filesize":1549288,
"file_type":"","zero_count":0,"nonprobative_count":10,"name_pairs":
["/home/bdallen/KittyMaterial","/home/bdallen/KittyMaterial/
HighQuality/DSC00003.JPG"]}],"source_sub_counts":
["1dd00f2e51aeebe7541cea4ade2e20b5",1]}
```


Users may be put off by the quantity of matches incurred by low-entropy data in their databases such as number tables or metadata header blocks from files that are otherwise unique. Database manipulation commands, **subsection 3.3**, can mitigate this, for example:

- Use the "subtract" command to remove known whitelist data created from sources such as "brand new" operating system media images and the NSRL.
- Alternatively, use the "add_range" command to copy all hash values that have been imported some number of times, for example, exactly once.

3.5 Statistics

Various statistics are available about a given hash database including the size of a database, where its hashes were sourced from, a histogram of its hashes, and more. Table 8 shows syntax for the statistics commands. Statistics options are shown in Table 9.

Table 8: Commands that provide Statistics about Hash Databases

Command	Usage	Description
size	`size <hashdb>`	Prints out size information relating to the database.
sources	`sources <hashdb>`	Prints source information for all sources in the database.
histogram	`histogram <hashdb>`	Prints a hash distribution for the hashes in the *hashdb*.
duplicates	`duplicates <hashdb> <number>`	Prints out hashes in the database that are sourced the given number of times.
hash_table	`hash_table <hashdb> <hex file hash>`	Prints hashes associated with the specified source.
read_media	`read_media <media image file> <offset> <count>`	Prints count raw bytes from a media image file starting at the given offset.

Table 9: Options for Commands that Provide Statistics

Option	Verbose Option	Specification
`-j`	`--json_scan_mode=e|o|c|a`	Select a mode, one of **e**xpanded, expanded **o**ptimized, **c**ount only, **a**pproximate count. Default is **o**.

3.5.1 `size`

Prints size information about the given database. Size values are specific to the underlying database storage implementation and indicate how large the parts of the database are.

To find the size of various data stores in hash database `example.hdb`, type the following:

- `hashdb size examle.hdb`

The above command prints the size of various data stores within the database in JSON format.

3.5.2 `sources`

Prints out all source file references that have contributed to this database including repository names and filenames.

3.5.3 `histogram`

Prints a hash distribution of the hashes in the given database, see **subsection 6.10** for output syntax.

3.5.4 `duplicates`

Prints out hashes in the database that are sourced the given number of times.

3.5.5 `hash_table`

Prints out hashes associated with the specified source identified by the source file hexdigest.
To obtain a list of hashes in `example.hdb` associated with the source file identified by hexcode `16d75027533b0a5ab900089a244384a0`, type the following:

- `hashdb hash_table example.hdb 16d75027533b0a5ab900089a244384a0`

3.5.6 `read_media`

Prints raw bytes from the given media image. Note that these bytes are often not printable.

3.6 Performance Analysis

Performance analysis commands for analyzing *hashdb* performance are shown in Table 10. Performance analysis options are shown in Table 11. Timing data is placed in file `timestamp.json`, replacing any previous content.

Table 10: Commands that Support *hashdb* Performance Analysis

Command	Usage	Description
add_random	`add_random -r [<repository name>] <hashdb.hdb> <count>`	Adds count random hashes to the given database, creating timing data in the `log.xml` file.
scan_random	`scan_random [-j e|o|c|a] <hashdb.hdb>`	Scans random hashes in the given database, creating timing data in the `log.xml` file.
add_same	`add_same -r [<repository name>] <hashdb.hdb> <count>`	Adds count same hashes to the given database, creating timing data in the `log.xml` file.
scan_same	`scan_same [-j e|o|c|a] <hashdb.hdb>`	Scans count same hashes in the given database, creating timing data in the `log.xml` file.

Table 11: Options for Commands that Support Performance Analysis

Option	Verbose Option	Specification
`-j`	`--json_scan_mode=e|o|c|a`	Select a mode, one of **e**xpanded, expanded **o**ptimized, **c**ount only, **a**pproximate count. Default is **o**.

3.6.1 `add_random`

Add random hashes, leaving timing data in `log.xml`.

3.6.2 `scan_random`

Scan random hashes, leaving timing data in `log.xml`. Although this command does not produce output, the scan mode used impacts timing.

3.6.3 `add_same`

Add the same hash, leaving timing data in `log.xml`.

3.6.4 `scan_same`

Scan the same hash, leaving timing data in `log.xml`. Although this command does not produce output, the scan mode used impacts timing.

4 Tools that use *hashdb*

SectorScope, the *SectorScope* Autopsy Plug-in, and the **bulk_extractor** *hashdb* scanner use *hashdb*.

4.1 *SectorScope*

The *SectorScope* tool provides a GUI for analyzing data associated with block hash matches found on a media image. An example screenshot of the main window of *SectorScope* showing a histogram of matches on a media image is shown in Figure 7. *SectorScope* also provides interfaces for building and scanning against *hashdb* databases. Please see `https://github.com/NPS-DEEP/NPS-SectorScope/wiki` for more information on *SectorScope*.

4.2 The *SectorScope* Autopsy Plug-in

SectorScope provides an **Autopsy** plug-in for scanning for fragments of previously identified files. **Autopsy** is currently only available on Windows systems. This section describes how to set up the *SectorScope* **Autopsy** plug-in.

4.2.1 Installing the *SectorScope* Plug-in

The *SectorScope* Windows installer installs the requisite `.nbm` Autopsy plug-in module onto the desktop. Please follow these steps to install this module:

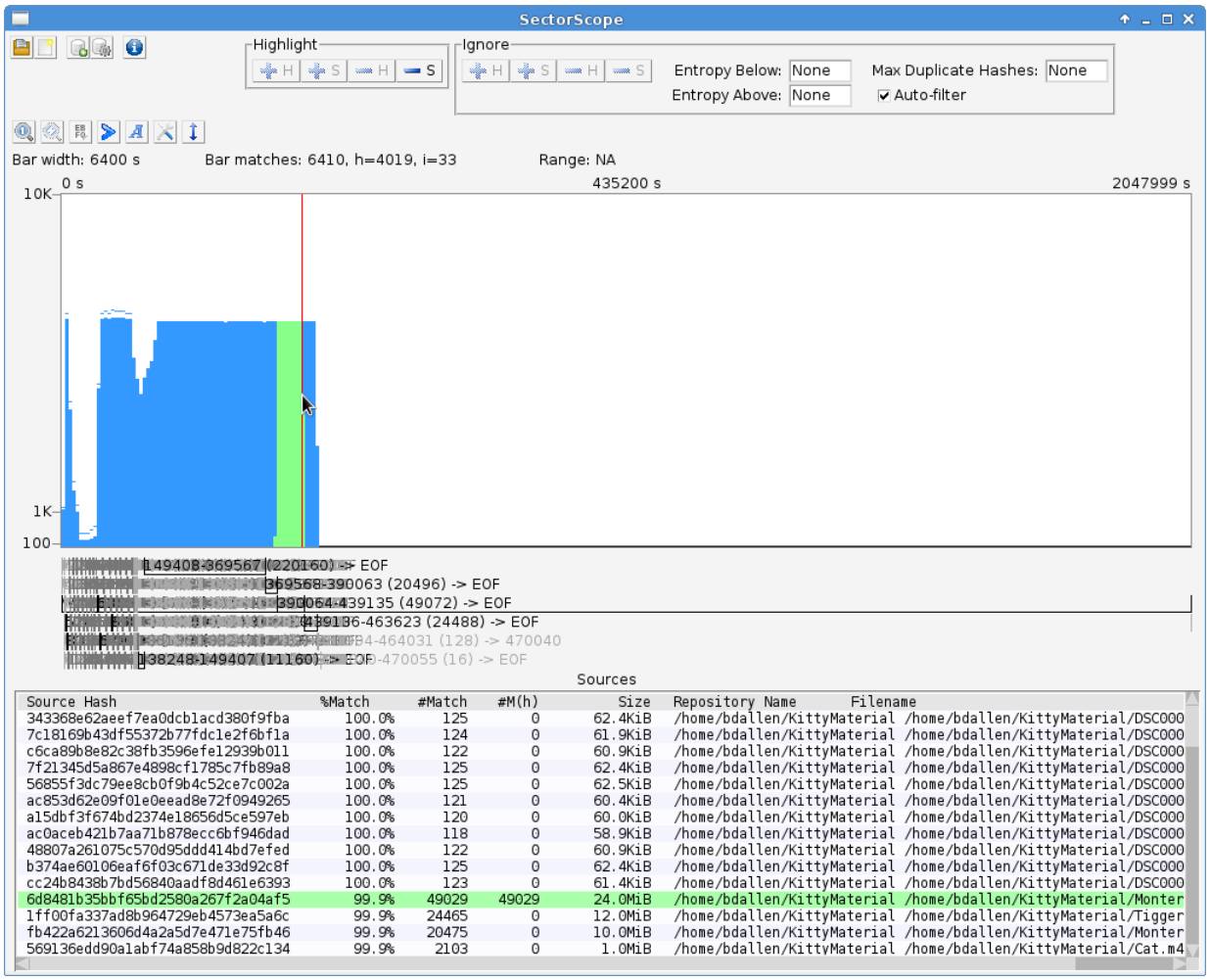

Figure 7: Example screenshot of the *SectorScope* tool

1. Open **Autopsy**. From the Autopsy menu, select `Tools | Plugins`.
2. Open the `Downloaded` tab and click the `Add Plugins...` button.
3. From the `Add Plugins` window, navigate to the `.nbm` module file that was installed onto the desktop, and open it.
4. Click `Install` and follow the wizard. Please note that it may be difficult to replace an old module of NPS-Autopsy-hashdb already installed in Autopsy. In the unlikely case that error `Some plugins require plugin org.jdesktop.beansbinding to be installed` appears, it may be necessary to uninstall and reinstall Autopsy.

4.2.2 Configuring the *SectorScope* Plug-in

The path to the *hashdb* database to scan against must be configured:

1. Start a new case, `File | New Case...`, fill in the Case Information fields, and click `Next`.
2. Fill in Case Information and click `Finish`.
3. For Add Data Source (1 of 3), put in a media image for Autopsy to process and click `Next`.
4. For Add Data Source (2 of 3), select checkboxes as desired, then click on `NPS-SectorScope` text to configure the path to your *hashdb* database to scan against. Currently a file chooser is not available, so please type in the full path, for example: `C:\Users\me\my_hashdb.hdb`. Click Next.
5. For Add Data Source (3 of 3) click `Finish`. When the NPS-SectorScope module begins processing, Autopsy will display "NPS-SectorScope ..." as **bulk_extractor** runs, which may take up to several hours. Unfortunately, **bulk_extractor** progress is not currently indicated. For diagnostics: please see if progress text is appearing in the generated `bulk_extractor\report.xml` file and in the generated log file or try running the scan manually.

4.3 bulk_extractor

bulk_extractor is an open source digital forensics tool that extracts features such as email addresses, credit card numbers, URLs and other types of information from digital evidence files. It operates on disk media images, files or a directory of files and extracts useful information without parsing the file system or file system structures. For more information on how to use **bulk_extractor** for a wide variety of applications, refer to the separate publication *The **bulk_extractor** Users Manual* available at `http://digitalcorpora.org/downloads/bulk_extractor/BEUsersManual.pdf` [2].

In particular, a *hashdb* **bulk_extractor** scanner is available which may be used to import block hashes into a new hash database and to scan for hashes against an existing hash database. Currently, *hashdb* requires a newer build of **bulk_extractor** than is available on the **bulk_extractor** site. Please see the *hashdb* Wiki page at `https://github.com/NPS-DEEP/hashdb/wiki` for information on obtaining a version of **bulk_extractor** that is compatible with the current version of *hashdb*.

Options that control the hashdb scanner are provided to **bulk_extractor** using `"-S name=value"` parameters. Example syntax for the **bulk_extractor** *hashdb* scanner is shown in Table 12. Scanner options are described in Table 13.

When importing, the new database of imported hashes is created in the output directory at `hashdb.hdb`. When scanning, matches are written in the output directory at file `identified_blocks.txt` with one match per line, as described in **section 6**) Listing 8.

Table 12: bulk_extractor *hashdb* Scanner Commands

Goal	Example	Description
import files	`bulk_extractor -E hashdb -S hashdb_mode=import -o outdir1 -R my_directory`	Import hashes from directory into outdir1/hashdb.hdb
import media	`bulk_extractor -E hashdb -S hashdb_mode=import -o outdir1 my_media_image1`	Import hashes from media image into outdir1/hashdb.hdb
scan media	`bulk_extractor -E hashdb -S hashdb_mode=scan -S hashdb_scan_path =outdir1/hashdb.hdb -o outdir2 my_media_image2`	Scan media image for hashes matching hashes in outdir1/hashdb.hdb

Table 13: bulk_extractor *hashdb* Scanner Options

Option	Default	Specification
`hashdb_mode`	`none`	The mode for the scanner, one of `[none|import|scan]`. For "none", the scanner is active but performs no action. For "import", the scanner imports block hashes. For "scan", the scanner scans for matching block hashes.
`hashdb_block_size`	512	Block size, in bytes, used to generate hashes.
`hashdb_step_size`	512	step size, in bytes. Scans and imports along this step value.
`hashdb_scan_path`		The file path to a hash database to scan against. Valid only in scan mode. No default provided. Value must be specified if in scan mode.
`hashdb_repository_name`	`default_ repository`	Selects the repository name to attribute the import to. Valid only in import mode.
`hashdb_max_feature_file_ lines`	0	The maximum number of feature lines to record or 0 for no limit. Valid only in scan mode.

5 Use Cases for *hashdb*

There are many different ways to utilize the functionality provided by the *hashdb* tool. In this section, we highlight some of the most common uses of the system.

5.1 Querying for Source or Database Information

Users can scan a hash database directly using various querying commands. Those commands are outlined in Table 6. The "scan" command allows users to search for hash blocks.

5.2 Writing Software that works with *hashdb*

hashdb provides Python and C++ APIs that can manage all aspects of a hash database including importing and scanning [see **Section 7** for information on using these APIs]. Other software programs can use these APIs to access database capabilities. The file `hashdb.hpp` found in the *src* directory contains the complete specification of the API. That complete file is also contained in Appendix C of this document. The two key features provided by the API include the ability to import values into a hash database and the ability to scan media for any values matching those in a given hash database. The **bulk_extractor** program uses the *hashdb* API to implement both of these capabilities.

5.3 Scanning or Importing to a Database Using bulk_extractor

The **bulk_extractor** *hashdb* scanner allows users to query for fragments of previously encountered hash values and populate a hash database with hash values. Options that control the *hashdb* scanner are provided to **bulk_extractor** using the "-S name=value" command line parameters. When **bulk_extractor** executes, the parameters are sent directly to the scanner.

For example, the following command runs the **bulk_extractor** *hashdb* scanner in import mode and adds hash values calculated from disk media image `my_media_image` to a hash database:

- `bulk_extractor -e hashdb -o outputDir -S hashdb_mode=import my_media_image`

Note, **bulk_extractor** will place feature file and other output not relevant to the *hashdb* application in the "outputDir" directory. When using the import command, the output directory will contain a newly created hash database called `hashdb.hdb`. That database can then be copied or added to a hash database in another location.

5.4 Updating Hash Databases

hashdb provides users with the ability to manipulate the contents of hash databases. The specific command line options for performing these functions are described in Table 5. *hashdb* databases are treated as sets with the add, subtract and intersect commands basically using add, subtract and intersect set operations. For example, the following command will copy all non-duplicate values from `demo.hdb` into `demo_dedup.hdb` by copying all values with a count less than or equal to one:

■ `hashdb add_range demo.hdb demo_dedup.hdb :1`

Whenever a database is created or updated, *hashdb* updates the file `log.xml`, found in the database's directory with information about the actions performed.

After each command that changes a database, statistics are writen in the `log.xml` file and to `stdout`. Table 19 shows all of the changes tracked in the log file along with their meaning. The value of each change statistic is the number of times the event happened during the command.

5.5 Exporting Hash Databases

Users can export hashes from a hash database to a JSON export file using the "export" command [see **Section 6** for information on JSON syntax]. For example, the following command will export the `demo.hdb` database to the file `demo.json`:

■ `hashdb export demo.hdb demo.json`

5.6 Sharding Hash Databases

A block hash database may be sharded into multiple separate databases by using the `-p` option of the "export" command to export parts by block hash range, and then importing each range into individual shard databases.

6 *hashdb* Input/Output Syntax

Many of the *hashdb* commands and API interfaces require or emit data. This section describes the syntax used and required by *hashdb* commands and API interfaces.

6.1 General Output Conventions

- **Expected output**
  Expected output is printed to `stdout`, for example the *hashdb* `create` command will respond with `New database created`.
- **JSON output**
  All JSON output is printed to `stdout`.
- **Status**
  Some commands generate status information. This information is prefixed with a `#` character and a space, and may be treated as a comment. For example the *hashdb* `ingest` command will produce status including files processed, progress, and changes made to the database. The comment identifier separates status from JSON content.
- **Errors**
  Errors are is printed to `stderr`, for example the *hashdb* `create` command might fail with the message `Unable to create new hashdb database at path`.
- **Warnings**
  Warnings are printed to `stderr`. Warnings may result when a command cannot fully complete, for example when JSON input syntax is invalid or when part of an input file cannot be read.

Listing 2: Example tab-delimited import file

```
# tab-delimited import file
# <file hexdigest> <tab> <block hash> <tab> <index>
fac7051447c781b69125994c5d125637    3b6b477d391f73f67c1c01e2141dbb17    1
fac7051447c781b69125994c5d125637    89a170b6b9a948d21d1d6ee1e7cdc467    2
fac7051447c781b69125994c5d125637    f58a09656658c6b41e244b4a6091592c    3
```

Listing 3: Example JSON source data used during import/export

```
{
  "file_hash":"3bf06fd991c312bd852c5f7b84d78174",
  "filesize":5712046,
  "file_type":"",
  "zero_count":3860,
  "nonprobative_count":32,
  "name_pairs":["/home/bdallen/KittyMaterial",
                "/home/bdallen/KittyMaterial/Cat.mov"]}
}
```

6.2 Tab-delimited Import File

The `import_tab` command imports hashes from tab delimited files. The tab-delimited import file consists of hash lines separated by carriage returns, where each line consists of a filename followed by a tab followed by the file hash followed by a 512-byte sector index that starts at 1. Comment lines are allowed by starting them with the # character. An example tab-delimited file is shown in Listing 2.

6.3 Import/Export Syntax

The import and export commands and API interfaces communicate source data and block hash data using JSON syntax.

6.3.1 Source Data

Source data defines information about a source. Source data is identified by the file hash of the source. An example source data line is shown in Listing 3. Fields are described in Table 14.

6.3.2 Block Hash Data

Block hash data is identified by the file block hash. An example block hash line is shown in Listing 4. Fields are described in Table 15.

6.4 Scan Data

When hash matches are found, *hashdb* returns data in JSON format. Due to varying requirements for speed and completeness, several options are available. This section

Table 14: Fields used in JSON source data

Field	Meaning
`file_hash`	The hexdigest of the source file containing the block hash
`filesize`	The size, in bytes, of the source file
`file_type`	A classification of what type of file the source file is
`zero_count`	The number of blocks in the source that have all bytes in the block equal to zero
`nonprobative_count`	The number of blocks in the source that are considered to be nonprobative
`name_pairs`	An array of source name, filename pairs associated with this source

Listing 4: Example JSON block hash data used during import/export

```
{
  "block_hash":"1d7379fd4d5cf676a9d4de1e48337e71",
  "k_entropy":4880,
  "block_label":"",
  "source_sub_counts":["1dd00f2e51aeebe7541cea4ade2e20b5",1]}
}
```

Table 15: Fields used in JSON block hash data

Field	Meaning
`block_hash`	A block hash hexdigest
`k_entropy`	The entropy value calculated for the block, scaled up by $1,000$
`block_label`	A label describing the type of data within the block. The block label may include information that it matched a whitelist database during import. The entropy and block label fields may be used together to estimate that a block might be nonprobative
`source_sub_counts`	An array of pairs of source hash and source sub-count values for each matching source

describes the JSON output options available for hash matches.

6.4.1 Expanded Hash

The returned JSON data contains all the information about a matched hash and the sources containing the hash that matched, even if it has already been returned in a previous scan. An example of expanded JSON output formatted with line breaks added for readability is shown in Listing 5. Fields are described in Table 16.

Listing 5: Example JSON block hash expanded data output from a scan match, with line breaks added for readability

```
{
  "block_hash":"1d7379fd4d5cf676a9d4de1e48337e71",
  "k_entropy":4880,
  "block_label":"",
  "count":1,
  "source_list_id":1193146442,
  "sources":[{
    "file_hash":"1dd00f2e51aeebe7541cea4ade2e20b5",
    "filesize":1549288,
    "file_type":"",
    "zero_count":0,
    "nonprobative_count":10,
    "name_pairs":[
      "/home/bdallen/KittyMaterial",
      "/home/bdallen/KittyMaterial/HighQuality/DSC00003.JPG"]
  }],
  "source_sub_counts":["1dd00f2e51aeebe7541cea4ade2e20b5",1]
}
```

Table 16: Fields used in JSON scan data

Field	Meaning
`block_hash`	The hexdigest hash of the block
`k_entropy`	The entropy value calculated for the block, scaled up by $1,000$
`block_label`	A label describing the type of data within the block
`source_list_id`	A source list ID calculated as a CRC of the source file hashes associated with the block hash
`sources`	An array of source data for each matching source
`file_hash`	The hexdigest hash of a matching source
`filesize`	The size, in bytes, of a matching source
`file_type`	A classification of what type of file a matching source is
`zero_count`	The number of blocks in a matching source that have all bytes in the block equal to zero
`nonprobative_count`	The number of blocks in a matching source that are considered to be nonprobative
`name_pairs`	An array of repository name, filename pairs associated with a matching source
`source_sub_counts`	An array of pairs of source hash and source sub-count values for each matching source

6.4.2 Expanded Hash, Optimized

The returned JSON data contains all the information about a matched hash and the sources containing the hash that matched in the first match, but hash and source metadata is not returned more than once. This optimization reduces the amount of data returned during the scan, but the user must remember associated hash and source metadata as it is returned because it is not returned in subsequent matches. An ex-

Listing 6: Example JSON block hash data count output from a scan match, with line breaks added for readability

```
{
  "block_hash": "3b6b477d391f73f67c101e2141dbb17",
  "count": 501
}
```

Listing 7: Example JSON block hash data approximate count output from a scan match, with line breaks added for readability

```
{
  "block_hash": "3b6b477d391f73f67c101e2141dbb17",
  "approximate_count": 500
}
```

ample output of a subsequent match of the same hash might be: `{"block_hash": "3b6b477d391f73f67c1c01e2141dbb17}`.

6.4.3 Hash Count

Only the count field is returned, indicating the number of sources cited in each match. JSON output contains the hash and the count as shown in example Listing 6. This capability is an optimization provided for users who do not need other hash information.

6.4.4 Approximate Hash Count

Only an approximate count field is returned, and it is possible for the count to be wrong. This capability is an optimization provided for users who do not need other hash information and can accept count values that are not exact. This is the fastest scan option since it only reads the hash store. For information on the hash store, see **Section 8**. JSON output contains the hash and the approximate count as shown in example Listing 7.

6.5 Scan Data Output from Tools

The scan commands provided by the *hashdb* tool and the **bulk_extractor** *hashdb* scanner print one line of output per match. This output consists of the byte offset, which may include a recursion path, a tab, the hash hexcode, a tab, the expanded hash JSON data, and a carriage return. An example of a scan match is shown in Listing 8.

6.6 Scan Stream Interface Data

Scan stream interface data consists of packed binary strings of unscanned input data and packed binary strings of scanned output data. Each contains an array of data as follows:

- **unscanned input data**

Listing 8: Example output from a scan match

```
2543104 1d7379fd4d5cf676a9d4de1e48337e71        {"block_hash":"1d7379fd4d
5cf676a9d4de1e48337e71","k_entropy":4880,"block_label":"","count":1,"sour
ce_list_id":1193146442,"sources":[{"file_hash":"1dd00f2e51aeeb e7541cea4a
de2e20b5","filesize":1549288,"file_type":"","zero_count":0,"nonprobative_
count":10,"name_pairs":["/home/bdallen/KittyMaterial","/home/bdallen/Kitt
yMaterial/HighQuality/DSC00003.JPG"]}],"source_sub_counts":["1dd00f2e51ae
ebe7541cea4ade2e20b5",1]}
```

Listing 9: Example scan list input file

```
# Scan list input file
# <offset> <tab> <block hash hexdigest>
0       3b6b477d391f73f67c1c01e2141dbb17
512     89a170b6b9a948d21d1d6ee1e7cdc467
1024    f58a09656658c6b41e244b4a6091592c
```

  - **hash** A binary hash to scan for, of length `hash_size` bytes.
  - **label length** A 2-byte unsigned integer in native-Endian format indicating the length, in bytes, of the binary label associated with the scan record.
  - **label** A binary label associated with the scan record.

- **scanned output data**
  - **hash** A binary hash that matched, of length `hash_size` in bytes.
  - **label length** A 2-byte unsigned integer in native-Endian format indicating the length, in bytes, of the binary label associated with the hash that matched.
  - **label** A binary label associated with the scan record.
  - **JSON length** A 4-byte unsigned integer in native-Endian format indicating the length, in bytes, of the JSON text associated with the hash that matched.
  - **JSON** The JSON text formatted based on the scan mode selected.

6.7 Scan List Input File

The `scan_list` command scans a list of hashes for matches. Valid lines of input may be:

- Comment lines starting with `#`. Comment lines are forwarded to output.
- Hash lines to scan against, where each line consists of an offset followed by a tab followed by the hash hexcode.

An example scan list input file is shown in Listing 9.

6.8 Size

The *hashdb* `size` command and `size` API interface returns size information about internal data structures in JSON format. The size of the `source_id_store` indicates the number of sources. The size of the `hash_store` is greater than or equal to the number of

Listing 10: Example JSON output of database size values

```
{
  "hash_data_store":401746,
  "hash_store":401598,
  "source_data_store":88,
  "source_id_store":88,
  "source_name_store":88
}
```

Listing 11: Example JSON histogram format

```
{
  "duplicates":2,
  "distinct_hashes":3,
  "total":6
}
```

hashes stored, and is not exact because of how data is stored. Although for internal use, these fields can give some sense of the size of a *hashdb* database. An example output is shown in Listing 10.

6.9 Sources

The `sources` command prints JSON data as shown in Listing 3 and described in Table 14.

6.10 Histogram

The `histogram` command shows the density of hash duplicates across a hash database. Fields are described in Table 17. An example histogram output line is shown in Listing 11.

Table 17: Fields used in JSON histogram output

Field	Meaning
`duplicates`	The total count of file offsets identified for each source for the hash value
`distinct_hashes`	The number of distinct hashes in the database with this duplicates count
`total`	The total number of hashes represented by this entry, specifically, `duplicates` $*$ `distinct_hashes`

6.11 Duplicates

The `duplicates` command prints JSON data associated with hashes with a specified duplicates count as shown in Listing 5 and described in Table 16.

Listing 12: Example JSON timestamp format

```
{
  "name":"begin",
  "delta":"0.000396",
  "total":"0.000396"
}
```

6.12 Hash Table

The `hash_table` command prints JSON data associated with a file hash as shown in Listing 5 and described in Table 16.

6.13 Read Media

The `read_media` command prints raw binary bytes from a media image file. It is intended that this output be consumed by other tools since raw binary data is typically unreadable.

6.14 Timing

hashdb provides timing data in JSON format for use with timing analysis. Python scripts may use this output to produce performance plots. An example timestamp entry is shown in Listing 12. Fields are described in Table 18.

Table 18: Fields used in JSON timing data

Field	Meaning
`name`	The name of the timestamp
`delta`	The delta time since the previous timestamp. In this example, the delta is from the time the timestamping started
`total`	The total time since timestamping started

6.15 Database Changes

Statistics about hash database changes are reported on the console and to the log file inside the hash database. These statistics show specific changes made to stores within the hash database and also changes not made because conditions were not met. An example change report is shown in Listing 13. Changes with a count of zero are not reported. Changes tracked are summarized in Table 19 and discussed further in **section 8**.

Listing 13: Example report of a database change from an import operation

```
# Processing 100000 of ?...
# Processing 200000 of ?...
# Processing 300000 of ?...
# Processing 400000 of ?...
# Processing 401686 of 401686 completed.
# hashdb changes:
#     hash_data_merged: 401713
#     hash_inserted: 401598
#     hash_count_changed: 69
#     hash_count_not_changed: 46
#     source_data_inserted: 88
#     source_data_changed: 88
#     source_id_inserted: 88
#     source_id_already_present: 401801
#     source_name_inserted: 88
```

Table 19: Database changes resulting from commands that manipulate hash databases

Statistic	**Meaning**
`hash_data_inserted`	Number of insert operations issued
`hash_data_merged`	Number of merge operations issued and accepted
`hash_data_merged_same`	Number of merge operations issued but ignored because the data is already there
`hash_data_mismatched_data_detected`	Number of insert or merge operations issued where entropy or label data did not match
`hash_data_mismatched_sub_count_detected`	Number of merge operations issued where the subcount value did not match
`hash_inserted`	Number of new hash values inserted
`hash_count_changed`	Number of hash count changes applied
`hash_count_not_changed`	Number of hash and count changes provided but same
`source_data_inserted`	Number of source data records inserted
`source_data_changed`	Number of source data records changed
`source_data_same`	Number of source data records provided but same
`source_id_inserted`	Number of source ID records inserted
`source_id_already_present`	Number of source ID records provided but already present
`source_name_inserted`	Number of source names inserted
`source_name_already_present`	Number of source names provided but already present

7 Using the *hashdb* Library APIs

hashdb provides C++ and Python interfaces for importing, scanning, and working with block hashes:

- **C++ Interfaces**
  To use C++ interfaces, include interface file `hashdb.hpp` and link *hashdb* library `libhashdb`. *hashdb* interfaces use the *hashdb* namespace. Interfaces can assert on unexpected error.

- **Python Interfaces**
  To use the Python interfaces, load the `hashdb` module.

For information on installing the *hashdb* interfaces, please see **Subsection 1.8**. For further details on syntax and usage, please see *hashdb* header file `hashdb.hpp` in **Appendix C**. Python users may also want to reference the Python interface test module in the source code at `hashdb/python_bindings/test_hashdb.py`.

7.1 Data Types

C++ and Python use the following data type:

- The `scan_mode_t` enumerator defines JSON scan output modes: `EXPANDED`, `EXPANDED_OPTIMIZED`, `COUNT`, and `APPROXIMATE_COUNT`.

Interfaces specific to C++ also use the following data types:

- The `source_sub_count_t` class holds `file_hash` and `sub_count` information for a source.
- `source_sub_counts_t: typedef set<source_sub_count_t> source_sub_counts_t`
- `source_name_t: typedef pair<repository_name, fillename> source_name_t`
- `source_names_t: typedef set<source_name_t> source_names_t`

7.2 Settings

Holds *hashdb* settings.

- `settings = settings_t()`
  Obtain default settings. The configurable setting parameters are: `settings_version, byte_alignment, block_size, hash_prefix_bits, hash_suffix_bytes`.
- `settings_string = settings.settings_string()`
  Return setting values in JSON format.

7.3 Support Functions

Support functions provide miscellaneous support and are not part of a class.

- `version = version()`
  Return the *hashdb* version.
- `version = hashdb_version()`
  Return the *hashdb* version, same as `version`.
- `error_message = create_hashdb(hashdb_dir, settings, command_string)`
  Create a hash database given settings. Return `""` else reason for failure.
- `error_message = read_settings(hashdb_dir, &settings)`
  Query settings else false and reason for failure.
- `binary_string = hex_to_bin(hex_string)`

- `hex_string = bin_to_hex(binary_string)`
- `error_message = ingest(hashdb_dir, ingest_path, step_size, repository_name, whitelist_dir, disable_recursive_processing, disable_calculate_entropy, disable_calculate_labels, command_string)`
  Calculate and import hashes from path to *hashdb*. Can disable recursive processing, calculating entropy, and calculating labels.
- `error_message = scan_media(hashdb_dir, media_image_file, step_size, disable_recursive_processing, scan_mode)`
  Scan the media image for matches, writing match data to `stdout`.
- `error_message = read_media(media_image_file, offset, count, &bytes)`
  C++ syntax. Read bytes at a string offset from a media image file.
- `error_message, bytes_media = read_media(media_image_file, offset, count)`
  Python syntax. Read bytes at a string offset from a media image file, for example `1000` or `1000-zip-0`.
- `error_message = read_media(media_image_file, offset, count, &bytes)`
  C++ syntax. Read bytes at a numeric offset from a media image file.
- `error_message, bytes_read = read_media(media_image_file, offset, count)`
  Python syntax. Read bytes at a numeric offset from a media image file, for example `1000` or `1000-zip-0`.
- `error_message = read_media_size(media_image_file, &size)`
  C++ syntax. Read media image file size.
- `error_message, size = read_media_size(media_image_file)`
  Python syntax. Read media image file size.

7.4 Import

To import hash and source data, open an import manager, for example `{manager = import_manager_t("hashdb.hdb", "create my DB")`. Then use import functions to add data. Information in the log file will be added when the import manager closes. The contents of log files is described in **subsection 2.11**.

- `import_manager = import_manager_t(hashdb_dir, command_string)`
  Open the import manager. `command_string` will be written to the log file.
- `import_manager.insert_source_name(file_hash, repository_name, filename)`
  Register the repository name, filename pair to the file hash.
- `import_manager.insert_source_data(file_hash, filesize, file_type, zero_count, nonprobative_count)`
  Set the source parameters for the file hash.
- `import_manager.insert_hash(block_hash, k_entropy, block_label, file_hash, sub_count)`
  Set hash parameters and add source count information for a new hash.

- `import_manager.merge_hash(block_hash, k_entropy, block_label, file_hash, sub_count)`
  C++ only. Set hash parameters and add source count information for a complete set of source information for a hash.

- `error_message = import_manager.import_json(json_string)`
  Import hash or source, return `error_message` or "" for no error.

- `has_source = import_manager.has_source(file_hash)`
  See if the source is already present.

- `first_file_hash = import_manager.first_source()`
  Access sources that have already been imported.

- `file_hash = import_manager.next_source(file_hash)`
  Access sources that have already been imported.

- `data_sizes = import_manager.size()`
  Return JSON text indicating the number of entries in the LMDB databases.

- `size_t import_manager.size_hashes()`
  Return number of hash data store records in the database, which will be more than the number of different hash values actually imported if duplicate hash values are imported from multiple sources.

- `size_t import_manager.size_sources()`
  Return the number of sources in the database, which can include sources from decompressed content.

7.5 Scan

To scan for hashes, open a scan manager, for example
`manager = scan_manager_t("hashdb.hdb")`. Then use functions to find hash and source information. Functions that return less information run faster than functions that return more. Scan functions provide read-only access to hash and data stores.

- `scan_manager = scan_manager_t(hashdb_dir)`
  Open the scan manager.

- `bool scan_manager.find_hash(block_hash, &k_entropy, &block_label, &count, source_sub_counts)`
  C++ only. Find hash, obtain fields related to hash on match.

- `json_text = scan_manager.export_hash_json(block_hash)`
  Export hash information for the given binary hash else "" if not there.

- `json_text = scan_manager.export_source_json(file_hash)`
  Export source information for the given source else "" if not there.

- `count = scan_manager.find_hash_count(block_hash)`
  Return the total count of offsets associated with the hash.

- `approximate_count = scan_manager.find_approximate_hash_count(block_hash)`
  This is the fastest scan function. It returns an approximate total count of offsets associated with the hash, and can be wrong.

- `has_source_data = scan_manager.find_source_data(file_hash, filesize, file_type, zero_count, nonprobative_count)`
  C++ interface. Return information about the source.

- `has_source_data, filesize, file_type, zero_count, nonprobative_count = scan_manager.find_source_data(file_hash, filesize, file_type, zero_count, nonprobative_count)`
  Python interface. Return information about the source.

- `has_source_names = scan_manager.find_source_names(file_hash, &source_names_t)`
  C++ only. Retrieve the source names for this source or "" on no match.

- `json_text = scan_manager.find_hash_json(scan_mode, block_hash)`
  Find and return JSON text about the match or "" on no match. Text returned depends on the scan mode.

- `first_block_hash = scan_manager.first_hash()`
  Access hashes that have already been imported.

- `next_block_hash = scan_manager.next_hash(block_hash)`
  Access hashes that have already been imported.

- `first_file_hash = scan_manager.first_source()`
  Access sources that have already been imported.

- `file_hash = scan_manager.next_source(file_hash)`
  Access sources that have already been imported.

- `db_sizes = scan_manager.size()`
  Return sizes of internal data stores in JSON format.

- `size_hashes = scan_manager.size_hashes()`
  Return the number of hash data store records in the database, which will be more than the number of different hash values actually imported if duplicate hash values are imported from multiple sources.

- `size_sources = scan_manager.size_sources()`
  Return the number of sources in the database, which can include sources from decompressed content.

7.6 Scan Stream

The scan stream interface is provided to allow rapid multi-threaded scans of lists of hashes. The interface accepts long binary strings of unscanned data and returns long binary strings of scanned data. The user must encode and decode this packed data. The user may wish to embed this stream inside a custom socket layer.

- `scan_stream = scan_stream_t(scan_manager, hash_size, scan_mode)`
  Open a scan stream interface.

- `scan_stream.put(unscanned_data)`
  Submit unscanned data for scanning.

- `scanned_data = scan_stream.get()`
  Retrieve scanned data else "" if data is currently not available.

- `is_empty = scan_stream.empty()`
  Return true if there is no scanned data available to retrieve, no unscanned data scheduled for scanning, and the scanner threads are not busy.

7.7 Timestamp

Provide timestamp support.

- `timestamp = timestamp_t()`
  Create a timestamp object.

- `timestamp_string = stamp(text)`
  Create a named timestamp and provide time and delta from the last stamp time in JSON format.

8 LMDB Data Stores

This section provides details of how LMDB data stores are managed within a *hashdb* database. This technical information is provided to give context behind the optimization settings and options provided by *hashdb* and to explain the meaning of changes reported in the change log.

8.1 LMDB Hash Store

The *LMDB Hash Store* is a highly compressed optimized store of all the block hashes in the database. When scanning for a hash, if it is not in this store, then it is not in the database. Because of the degree of optimization, there can be false positives. To compensate, when a hash is found in the *LMDB Hash Store*, *hashdb* reads the *LMDB Hash Data Store* to be sure the hash actually exists.

The *LMDB Hash Store* is a B-Tree-based store:

- The `key` portion consists of the first 7 bytes of a block hash, in binary. In a database of one billion hashes, this will result in a false positive rate of about one in 72 million.

- The `value` portion consists of an approximate count encoded in one byte.

8.2 LMDB Hash Data Store

The *LMDB Hash Data Store* is a multi-map store of all hashes and their associated data and source information:

- The `key` portion consists of a block hash, in binary.

- The `value` portion contains information about the hash, sources, sub-counts, and total counts of identified blocks. This information is encoded within three types of value records:

- **Type 1** only one entry for this hash:
  `source_id, k_entropy, block_label, sub_count, 0-2 byte padding`
- **Type 2** first line of multi-entry hash:
  `NULL, k_entropy, block_label, count`
- **Type 3** remaining lines of multi-entry hash:
  `source_id, sub_count`

Fields in the `value` portion are:

- `source_id` A source ID integer that maps to a source file hash.
- `k_entropy` The calculated entropy for the block, scaled up by $1,000$.
- `block_label` A label identifying information about the block. Users may wish to examine `k_entropy` and `block_label` together to estimate that a block might be nonprobative.
- `sub_count` The number of times this block has been seen in this source. For Type 1 records, the `sub_count` is also the `count`.
- `count` The total number of times this block has been seen in all the sources. For Type 1 records, the `sub_count` is also the `count`.
- `NULL` A NULL byte distinguishes Type 1 records from Type 2. Note that Type 3 records are distinguished as following Type 1 going forward until the `key` changes.
- `0-2 byte padding` Up to 2 NULL bytes of padding so Type 1 can transition to Type 2 without changing size.

8.3 LMDB Source ID Store

The *LMDB Source ID Store* maps source file hash values to source IDs. Although the user never sees source IDs, we use source IDs in the *LMDB Source ID Store*, *LMDB Source Data Store*, and the *LMDB Hash Data Store* because they are significantly shorter than source file hashes. We wouldn't need source IDs if we didn't make this optimization.

- The `key` is the `file_hash`.
- The `value` is the `source_id`.

8.4 LMDB Source Data Store

The *LMDB Source Data Store* holds all the metadata about sources:

- The `key` is the `source_id`.
- The `value` consists of these fields:
  - `file_hash` The source file hash associated with this source ID, in binary.
  - `filesize` The size of the source file, in bytes.
  - `file_type` A label indicating the type of the file, user defined.
  - `zero_count` The number of blocks in the source that have all bytes in the block equal to zero.
  - `nonprobative_count` The number of block hashes stored for this source which are considered to be nonprobative. Users may wish to set the `nonprobative_count` value based on the `k_entropy` and `block_label` values of each block in the source.

8.5 LMDB Source Name Store

The *LMDB Source Name Store* multimap maps source IDs to source names. This store allows us to not re-import hashes from the same source and also allows us to see the list of source names that are of the same source.

- The `key` is the `source_id`.
- The `value` is a name pair of:
  - `repository_name` A label indicating the source repository.
  - `filename` The path to this source.

8.6 Data Store Changes

The following changes are logged when a *hashdb* operation modifies data stores within a hash database:

- `hash_data_inserted`
  Incremented once for each insert operation issued. All insert operations are accepted.
- `hash_data_merged`
  Incremented once for each new merge issued. Not incremented if already there, specifically, if source information is already present for the hash.
- `hash_data_merged_same`
  Incremented if already there, specifically, if source information is already present for the hash.
- `hash_data_mismatched_data_detected`
  Incremented when entropy or label information provided when values stored are different. Values stored are not changed.
- `hash_data_mismatched_sub_count_detected`
  Incremented when a merge operation is issued and the sub-count value does not match. The stored sub-count value does not change.
- `hash_inserted`
  Incremented each time a new 7-byte hash prefix is inserted. Not incremented if the hash prefix already exists.
- `hash_count_changed`
  Incremented each time an approximate hash count changes from one value to another. The approximate hash count is encoded in one byte. This encoding changes less frequently as the actual hash count value increases.
- `hash_not_changed`
  Incremented each time an insert is attempted but there is no change because the approximate hash count stays at the same value.
- `source_data_inserted`
  Incremented each time a new source data record is created.

- `source_data_changed`
  Incremented each time an existing source data record is changed.
- `source_data_same`
  Incremented each time an existing source data is submitted to be inserted but there is no change because the source data is already there and is the same.
- `source_id_inserted`
  Incremented each time a new source ID record is created.
- `source_id_already_present`
  Incremented each time a source ID record is submitted to be inserted but there is no change because the record is already there.
- `source_name_inserted`
  Incremented each time a new source filename, repository name pair is inserted.
- `source_name_already_present` Incremented each time a source filename, repository name pair is submitted but not stored because the name pair is already present.

9 Alternate Configurations

By default, *hashdb* is compiled to calculate MD5 hashes. *hashdb* can be recompiled to use other encryption algorithms or even other artifacts, please see source code file `hashdb/src_libhashdb/hasher/hash_calculator.hpp`.

- **Alternate Hash Algorithm**
  hashdb calculates block hashes using OpenSSL. If OpenSSL supports your hash algorithm, replace it with yours. For example if you want SHA1, replace `EVP_md5()` with `EVP_sha1()` in source code file `hashdb/src_libhashdb/hasher/hash_calculator.hpp` and recompile.
- **Alternate Artifacts**
  hashdb can be refitted to manage artifacts other than hashes. For example *hashdb* can be refitted to store and search for email addresses. Specifically, replace code that iterates through buffers and calculates block hashes with code that iterates through buffers and finds your artifact.

  For optimal performance, we recommend that you do not store your artifact as-is. Artifact key values should be relatively randomly distributed and not hundreds of bytes long. To achieve this, we recommend hashing your artifact with something like CRC64, and storing and scanning for the CRC hash value of the artifact.

References

[1] Garfinkel, S., and McCarrin, M. Hash-based Carving: Searching media for complete files and file fragments with sector hashing and hashdb, DFRWS 2015 USA. `http://www.sciencedirect.com/science/article/pii/S1742287615000468`

[2] Bradley, J., and Garfinkel, S. bulk_extractor users guide, September 2013. `http://digitalcorpora.org/downloads/bulk_extractor/BEUsersManual.pdf`.

[3] Young, J., Foster, K., Garfinkel, S., and Fairbanks, K. Distinct sector hashes for target file detection. *IEEE Computer* (December 2012). http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6311397.

Appendices

A *hashdb* Quick Reference

New Database

`create [-b <block size>] <hashdb.hdb>`	Create a new hash database.

Import/Export

`ingest [-r <repository name>] [-w <whitelist.hdb>] [-s <step size>] [-x rel] <hashdb.hdb> <import directory>`	Import from path recursively into hash database, labeling hashes in the whitelist and hashes matching entropy traits. Can disable **r**ecursion, **e**ntropy, **l**abels
`import_tab [-r <repository name>] [-w <whitelist.hdb>] <hashdb.hdb> <tab.txt>`	Import from tab file into hash database, labeling hashes in the whitelist.
`import <hashdb.hdb> <hashdb.json>`	Import JSON format data into hash database.
`export [-p <begin:end>] <hashdb.hdb> <hashdb.json>`	Export all or part of hash database in JSON format.

Database Manipulation

`add <A.hdb> <B.hdb>`	$A \rightarrow B$ add A into B
`add_multiple <A.hdb> <B.hdb> ... <C.hdb>`	$A + B + \ldots \rightarrow C$ add A, B, ... into C.
`add_repository <A.hdb> <B.hdb> <repository name>`	$A_r \rightarrow B$ add when repository name matches.
`add_range<A.hdb> <B.hdb> <m:n>`	$A_{m:n} \rightarrow B$ add hashes that have source counts within range, inclusive.
`intersect <A.hdb> <B.hdb> <C.hdb>`	$A \cap B \rightarrow C$ add when hash and source are common.
`intersect_hash <A.hdb> <B.hdb> <C.hdb>`	$A \cap B \rightarrow C$ add when hashes are common.
`subtract <A.hdb> <B.hdb> <C.hdb>`	$A - B \rightarrow C$ add when hash and source not common.
`subtract_hash <A.hdb> <B.hdb> <C.hdb>`	$A - B \rightarrow C$ add when hashes are not common.
`subtract_repository <A.hdb> <B.hdb> <repository name>`	$A_{\overline{r}} \rightarrow B$ add unless repository name matches.

Scan

`scan_list [-j e|o|c|a] <hashdb.hdb> <hashes file>`	Scan hashes file for hash match, return **e**xpanded, expanded **o**ptimized, **c**ount only, or **a**pproximate count.
`scan_hash [-j e|o|c|a] <hashdb.hdb> <hex block hash>`	Scan for hash match, return **e**xpanded, expanded **o**ptimized, **c**ount only, or **a**pproximate count.
`scan_media [-s <step size>] [-j e|o|c|a] [-x r] <hashdb.hdb> <media image file>`	Scan media image for hash match, return **e**xpanded, expanded **o**ptimized, **c**ount only, or **a**pproximate count. Can disable **r**ecursion.

Statistics

`size <hashdb.hdb>`	Print size information for internal database tables.
`sources <hashdb.hdb>`	Print source information.
`histogram <hashdb.hdb>`	Print hash distribution.
`duplicates [-j e|o|c|a] <hashdb.hdb> <number>`	Print hashes sourced the given number of times.
`hash_table [-j e|o|c|a] <hashdb.hdb> <hex file hash>`	Print hashes associated with the source file hash.
`read_media <media image file> <offset> <count>`	Print raw bytes from the media image file.
`read_media_size <media image file>`	Print the size of the media image file.

Performance Analysis

`add_random <hashdb.hdb> <count>`	Add random hashes, log to `timestamp.json`.
`scan_random [-j e|o|c|a] <hashdb.hdb> <count>`	Scan random hashes, log to `timestamp.json`.
`add_same <hashdb.hdb> <count>`	Add same hashes, log to `timestamp.json`.
`scan_same [-j e|o|c|a] <hashdb.hdb> <count>`	Scan same hashes, log to `timestamp.json`.

bulk_extractor Scanner

`bulk_extractor -E hashdb -S hashdb_mode=import -o outdir1 -R my_import_dir`	Import directory.
`bulk_extractor -E hashdb -S hashdb_mode=import -o outdir1 my_media_image`	Import media image.
`bulk_extractor -E hashdb -S hashdb_mode=scan -S hashdb_scan_path= outdir1/hashdb.hdb -o outdir2 my_media_image2`	Scan media image.

B Output of the *hashdb* Help Command

```
hashdb Version 3.1.0
Usage: hashdb [-h|--help|-h all] [-v|-V|--version]
       hashdb [-h <command>]
       hashdb [options] <command> [<args>]

New Database:
  create [-b <block size>] <hashdb>

Import/Export:
  ingest [-r <repository name>] [-w <whitelist.hdb>] [-s <step size>]
         [-x <rel>] <hashdb.hdb> <import directory>
  import_tab [-r <repository name>] [-w <whitelist.hdb>] <hashdb> <tab file>
  import <hashdb> <json file>
  export [-p <begin:end>] <hashdb> <json file>

Database Manipulation:
  add <source hashdb> <destination hashdb>
  add_multiple <source hashdb 1> <source hashdb 2> <destination hashdb>
  add_repository <source hashdb> <destination hashdb> <repository name>
  add_range <source hashdb> <destination hashdb> <m:n>
  intersect <source hashdb 1> <source hashdb 2> <destination hashdb>
  intersect_hash <source hashdb 1> <source hashdb 2> <destination hashdb>
  subtract <source hashdb 1> <source hashdb 2> <destination hashdb>
  subtract_hash <source hashdb 1> <source hashdb 2> <destination hashdb>
  subtract_repository <source hashdb> <destination hashdb> <repository name>

Scan:
  scan_list [-j e|o|c|a] <hashdb> <hash list file>
  scan_hash [-j e|o|c|a] <hashdb> <hex block hash>
  scan_media [-s <step size>] [-j e|o|c|a] [-x <r>] <hashdb> <media image>

Statistics:
  size <hashdb>
  histogram <hashdb>
  duplicates [-j e|o|c|a] <hashdb> <number>
  hash_table [-j e|o|c|a] <hashdb> <hex file hash>
  read_media <media image> <offset> <count>
  read_media_size <media image>

Performance Analysis:
  add_random <hashdb> <count>
  scan_random [-j e|o|c|a] <hashdb> <count>
  add_same <hashdb> <count>
  scan_same [-j e|o|c|a] <hashdb> <count>
  test_scan_stream <hashdb> <count>

New Database:
create [-b <block size>] <hashdb>
  Create a new <hashdb> hash database.

  Options:
  -b, --block_size=<block size>
    <block size>, in bytes, or use 0 for no restriction
    (default 512)

  Parameters:
  <hashdb>   the file path to the new hash database to create

Import/Export:
```

```
ingest [-r <repository name>] [-w <whitelist.hdb>] [-s <step size>]
       [-x <rel>] <hashdb.hdb> <import directory>
  Import hashes recursively from <import directory> into hash database
    <hashdb>.

  Options:
  -r, --repository_name=<repository name>
    The repository name to use for the set of hashes being imported.
    (default is "repository_" followed by the <import directory> path).
  -w, --whitelist_dir
    The path to a whitelist hash database.  Hashes matching this database
    will be marked with a whitelist entropy flag.
  -s, --step_size
    The step size to move along while calculating hashes.
  -x, --disable_processing
    Disable further processing:
      r disables recursively processing embedded data.
      e disables calculating entropy.
      l disables calculating block labels.

  Parameters:
  <import dir>   the directory to recursively import from
  <hashdb>       the hash database to insert the imported hashes into
import_tab [-r <repository name>] [-w <whitelist.hdb>] <hashdb> <tab file>
  Import hashes from file <tab file> into hash database <hashdb>.

  Options:
  -r, --repository_name=<repository name>
    The repository name to use for the set of hashes being imported.
    (default is "repository_" followed by the <import directory> path).
  -w, --whitelist_dir
    The path to a whitelist hash database.  Hashes matching this database
    will be marked with a whitelist entropy flag.

  Parameters:
  <hashdb>       the hash database to insert the imported hashes into
  <NIST file>    the NIST file to import hashes from
import <hashdb> <json file>
  Import hashes from file <json file> into hash database <hashdb>.

  Parameters:
  <hashdb>       the hash database to insert the imported hashes into
  <json file>    the JSON file to import hashes from
export [-p <begin:end>] <hashdb> <json file>
  Export hashes from hash database <hashdb> into file <json file>.

  Options:
  -p, --part_range=<begin:end>
    The part of the hash database to export, from begin hex block hash to
    end hex block hash.  The entire hash database is exported by default.

  Parameters:
  <hashdb>       the hash database to export
  <json file>    the JSON file to export the hash database into

Database Manipulation:
add <source hashdb> <destination hashdb>
  Copy hashes from the <source hashdb> to the <destination hashdb>.

  Parameters:
  <source hashdb>       the source hash database to copy hashes from
```

```
  <destination hashdb>  the destination hash database to copy hashes into
add_multiple <source hashdb 1> <source hashdb 2> <destination hashdb>
  Perform a union add of <source hashdb 1> and <source hashdb 2>
  into the <destination hashdb>.

  Parameters:
  <source hashdb 1>     a hash database to copy hashes from
  <source hashdb 2>     a second hash database to copy hashes from
  <destination hashdb>  the destination hash database to copy hashes into
add_repository <source hashdb> <destination hashdb> <repository name>
  Copy hashes from the <source hashdb> to the <destination hashdb>
  when the <repository name> matches.

  Parameters:
  <source hashdb>       the source hash database to copy hashes from
  <destination hashdb>  the destination hash database to copy hashes into
  <repository name>     the repository name to match when adding hashes
add_range <source hashdb> <destination hashdb> <m:n>
  Copy the hashes from the <source hashdb> to the <destination hashdb>
  that have source reference count values between m and n.

  Parameters:
  <source hashdb>       the hash database to copy hashes from that have a
                        source count within range m and n
  <destination hashdb>  the hash database to copy hashes to when the
                        source count is within range m and n
  <m:n>                 the minimum and maximum count value range in which
                        hashes will be copied
intersect <source hashdb 1> <source hashdb 2> <destination hashdb>
  Copy hashes that are common to both <source hashdb 1> and
  <source hashdb 2> into <destination hashdb>.  Hashes and their sources
  must match.

  Parameters:
  <source hashdb 1>     a hash databases to copy the intersection of
  <source hashdb 2>     a second hash databases to copy the intersection of
  <destination hashdb>  the destination hash database to copy the
                        intersection of exact matches into
intersect_hash <source hashdb 1> <source hashdb 2> <destination hashdb>
  Copy hashes that are common to both <source hashdb 1> and
  <source hashdb 2> into <destination hashdb>.  Hashes match when hash
  values match, even if their associated source repository name and
  filename do not match.

  Parameters:
  <source hashdb 1>     a hash databases to copy the intersection of
  <source hashdb 2>     a second hash databases to copy the intersection of
  <destination hashdb>  the destination hash database to copy the
                        intersection of hashes into
subtract <source hashdb 1> <source hashdb 2> <destination hashdb>
  Copy hashes that are in <souce hashdb 1> and not in <source hashdb 2>
  into <destination hashdb>.  Hashes and their sources must match.

  Parameters:
  <source hashdb 1>     the hash database containing hash values to be
                        added if they are not also in the other database
  <source hashdb 2>     the hash database containing the hash values that
                        will not be added
  <destination hashdb>  the hash database to add the difference of the
                        exact matches into
subtract_hash <source hashdb 1> <source hashdb 2> <destination hashdb>
```

```
  Copy hashes that are in <souce hashdb 1> and not in <source hashdb 2>
  into <destination hashdb>.  Hashes match when hash values match, even if
  their associated source repository name and filename do not match.

  Parameters:
  <source hashdb 1>     the hash database containing hash values to be
                        added if they are not also in the other database
  <source hashdb 2>     the hash database containing the hash values that
                        will not be added
  <destination hashdb>  the hash database to add the difference of the
                        hashes into
subtract_repository <source hashdb> <destination hashdb> <repository name>
  Copy hashes from the <source hashdb> to the <destination hashdb>
  when the <repository name> does not match.

  Parameters:
  <source hashdb>       the source hash database to copy hashes from
  <destination hashdb>  the destination hash database to copy hashes into
  <repository name>     the repository name to exclude when adding hashes

Scan:
scan_list [-j e|o|c|a] <hashdb> <hash list file>
  Scan hash database <hashdb> for hashes in <hash list file> and print out
  matches.

  Options:
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count
  -x, --disable_processing
    Disable further processing:
      r disables recursively processing embedded data.

  Parameters:
  <hashdb>          the file path to the hash database to use as the
                    lookup source
  <hashes file>     the file containing hash values to scan for
scan_hash [-j e|o|c|a] <hashdb> <hex block hash>
  Scan hash database <hashdb> for the specified <hash value> and print
  out matches.

  Options:
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count

  Parameters:
  <hashdb>          the file path to the hash database to use as the
                    lookup source
  <hex block hash>  the hash value to scan for
scan_media [-s <step size>] [-j e|o|c|a] [-x <r>] <hashdb> <media image>
  Scan hash database <hashdb> for hashes in <media image> and print out
  matches.
```

```
  Options:
  -s, --step_size
    The step size to move along while calculating hashes.
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count
  -x, --disable_processing
    Disable further processing:
      r disables recursively processing embedded data.

  Parameters:
  <hashdb>          the file path to the hash database to use as the
                    lookup source
  <media image>     the media image file to scan for matching block hashes

Statistics:
size <hashdb>
  Print the sizes of the database tables inside the given <hashdb> database.

  Parameters:
  <hashdb>       the hash database to print size information for
sources <hashdb>
  Print source information indicating where the hashes in the <hashdb>
  came from.

  Parameters:
  <hashdb>       the hash database to print all the repository name,
                 filename source information for
histogram <hashdb>
  Print the histogram of hashes for the given <hashdb> database.

  Parameters:
  <hashdb>       the hash database to print the histogram of hashes for
duplicates [-j e|o|c|a] <hashdb> <number>
  Print the hashes in the given <hashdb> database that are sourced the
  given <number> of times.

  Options:
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count

  Parameters:
  <hashdb>       the hash database to print duplicate hashes about
  <number>       the requested number of duplicate hashes
hash_table [-j e|o|c|a] <hashdb> <hex file hash>
  Print hashes from the given <hashdb> database that are associated with
  the <source_id> source index.

  Options:
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
```

```
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count

  Parameters:
  <hashdb>              the hash database to print hashes from
  <hex file hash>       the file hash of the source to print hashes for
read_media <media image> <offset> <count>
  Print <count> number of raw bytes starting at the specified <offset> in
  the <media image> file.

  Parameters:
  <media image>  the media image file to print raw bytes from
  <offset>       the offset in the media image file to read from
  <count>        the number of raw bytes to read
read_media_size <media image>
  Print the size, in bytes, of the media image file.

  Parameters:
  <media image>  the media image file to print the size of

Performance Analysis:
add_random <hashdb> <count>
  Add <count> randomly generated hashes into hash database <hashdb>.
  Write performance data in the database's log.txt file.

  Options:
  -r, --repository=<repository name>
    The repository name to use for the set of hashes being added.
    (default is "repository_add_random").

  Parameters:
  <hashdb>       the hash database to add randomly generated hashes into
  <count>        the number of randomly generated hashes to add
scan_random [-j e|o|c|a] <hashdb> <count>
  Scan for random hashes in the <hashdb> database.  Write performance
  data in the database's log.txt file.

  Options:
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count

  Parameters:
  <hashdb>       the hash database to scan
  <count>        the number of randomly generated hashes to scan for
add_same <hashdb> <count>
  Add <count> block hashes of value 0x800000... into hash database <hashdb>.
  Write performance data in the database's log.txt file.

  Options:
  -r, --repository=<repository name>
    The repository name to use for the set of hashes being added.
    (default is "repository_add_same").
```

```
  Parameters:
  <hashdb>       the hash database to add hashes of the same value into
  <count>        the number of hashes of the same value to add
scan_same [-j e|o|c|a] <hashdb> <count>
  Scan for the same hash value in the <hashdb> database.  Write
  performance data in the database's log.txt file.

  Options:
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count

  Parameters:
  <hashdb>       the hash database to scan
  <count>        the number of randomly generated hashes to scan for
test_scan_stream <hashdb> <count>
  Run <count> scan_stream requests, where each request contains 10K block
  hashes of value 0x800000....  Write performance data in the database's
  log.txt file.

  Options:
  -j, --json_scan_mode
    The JSON scan mode selects optimization and output (default is o):
      e return expanded output.
      o return expanded output optimized to not repeat hash and source
        information.
      c return hash duplicates count
      a return approximate hash duplicates count

  Parameters:
  <hashdb>       the hash database to scan
  <count>        the number of scan requests to issue
```

C *hashdb* C++ API: hashdb.hpp

```
// Author:  Bruce Allen
// Created: 2/25/2013
//
// 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.
//
// Released into the public domain on February 25, 2013 by Bruce Allen.

/**
```

```
 * \file
 * Header file for the hashdb library.
 *
 * NOTE: This file includes SWIG preprocessor directives used for
 * building Python bindings.  Specifically:
 *   SWIG is not defined when building C++.
 *   SWIG is defined when building Python bindings.
 */

#ifndef HASHDB_HPP
#define HASHDB_HPP

#include <string>
#include <set>
#include <stdint.h>
#include <sys/time.h>   // timeval* for timestamp_t
#include <pthread.h>    // pthread_t* for scan_stream_t

// ************************************************************
// version of the hashdb library
// ************************************************************
/**
 * Version of the hashdb library, outside hashdb namespace.
 */
extern "C"
const char* hashdb_version();

namespace scan_stream {
  class scan_thread_data_t;
}
namespace hashdb {
  class lmdb_hash_data_manager_t;
  class lmdb_hash_manager_t;
  class lmdb_source_data_manager_t;
  class lmdb_source_id_manager_t;
  class lmdb_source_name_manager_t;
  class lmdb_changes_t;
  class logger_t;
  class locked_member_t;

  // ************************************************************
  // version of the hashdb library
  // ************************************************************
  /**
   * Version of the hashdb library, inside hashdb namespace.
   */
  extern "C"
  const char* version();

  // ************************************************************
  // source sub_counts
  // ************************************************************
#ifndef SWIG
  // source_sub_count information
  struct source_sub_count_t {
    const std::string file_hash;
    const uint64_t sub_count;
    source_sub_count_t(const std::string& p_file_hash,
                       const uint64_t p_sub_count);
    /**
```

```
     * Only the file hash is compared.  If they are the same, the
         sub_count
     * should match.
     */
    bool operator<(const source_sub_count_t& that) const;
  };
  typedef std::set<source_sub_count_t> source_sub_counts_t;

  // pair(repository_name, filename)
  typedef std::pair<std::string, std::string> source_name_t;
  typedef std::set<source_name_t>             source_names_t;
#endif

  // ************************************************************
  // settings
  // ************************************************************
  /**
   * Provides hashdb settings.
   *
   * Attributes:
   *   settings_version - The version of the settings record
   *   block_size - Size, in bytes, of data blocks.
   */
  struct settings_t {
#ifndef SWIG
    static const uint32_t CURRENT_SETTINGS_VERSION = 4;
#endif
    uint32_t settings_version;
    uint32_t block_size;
    settings_t();
    std::string settings_string() const;
  };

  // ************************************************************
  // scan modes
  // ************************************************************
  /**
   * The scan mode controls scan optimization and returned JSON content.
   */
  enum scan_mode_t {EXPANDED,
                    EXPANDED_OPTIMIZED,
                    COUNT,
                    APPROXIMATE_COUNT};

  // ************************************************************
  // misc support interfaces
  // ************************************************************
  /**
   * Create a new hashdb.
   * Return true and "" if hashdb is created, false and reason if not.
   * The current implementation may abort if something worse than a simple
   * path problem happens.
   *
   * Parameters:
   *   hashdb_dir - Path to the database to create.  The path must not
   *     exist yet.
   *   settings - The hashdb settings.
   *   command_string - String to put into the new hashdb log.
   *
   * Returns:
   *   "" if successful else reason if not.
```

```
   */
  std::string create_hashdb(const std::string& hashdb_dir,
                            const hashdb::settings_t& settings,
                            const std::string& command_string);

  /**
   * Return hashdb settings else reason for failure.
   * The current implementation may abort if something worse than a simple
   * path problem happens.
   *
   * Parameters:
   *   hashdb_dir - Path to the database to obtain the settings of.
   *   settings - The hashdb settings.
   *
   * Returns:
   *   True and "" if settings were retrieved, false and reason if not.
   */
  std::string read_settings(const std::string& hashdb_dir,
#ifdef SWIG
                            hashdb::settings_t& OUTPUT
#else
                            hashdb::settings_t& settings
#endif
                           );

  /**
   * Return binary string or empty if hexdigest length is not even
   * or has any invalid digits.
   */
  std::string hex_to_bin(const std::string& hex_string);

  /**
   * Return hexadecimal representation of the binary string.
   */
  std::string bin_to_hex(const std::string& binary_string);

  /**
   * Calculate and ingest hashes from files recursively from a source
   * path.  Files with EWF extensions (.E01 files) will be ingested as
   * media images.
   *
   * Parameters:
   *   hashdb_dir - Path to the hashdb data store to import into.
   *   ingest_path - Path to a source file or directory to recursively
   *     ingest block hashes from.  May include E01 files.
   *   step_size - The step size to move along while calculating hashes.
   *     The step size must be divisible by the byte alignment defined in
   *     the database.
   *   repository_name - A repository name to attribute the sources to.
   *   whitelist_dir - Path to a whitelist hashdb data store.  Hashes
   *     matching these will not be ingested.
   *   disable_recursive_processing - Disable processing embedded data.
   *   disable_calculate_entropy - Disable calculating block entropy
      values.
   *   disable_calculate_labels - Disable calculating block entropy
      labels.
   *   command_string - String to put into the new hashdb log.
   *
   * Returns:
   *   "" if successful else reason if not.
   */
```

```
  std::string ingest(const std::string& hashdb_dir,
                     const std::string& ingest_path,
                     const size_t step_size,
                     const std::string& repository_name,
                     const std::string& whitelist_dir,
                     const bool disable_recursive_processing,
                     const bool disable_calculate_entropy,
                     const bool disable_calculate_labels,
                     const std::string& command_string);

  /**
   * Calculate and scan for hashes from the media image file.  Files with
   * EWF extensions (.E01 files) are recognized as media images.
   *
   * Parameters:
   *   hashdb_dir - Path to the hashdb data store to scan against.
   *   media_image_file - Path to a media image file, which can be a
   *     raw file or an E01 file.
   *   step_size - The step size to move along while calculating hashes.
   *     The step size must be divisible by the byte alignment defined in
   *     the database.
   *   disable_recursive_processing - Disable processing embedded data.
   *   scan_mode - The mode to use for performing the scan.  Controls
   *     scan optimization and returned JSON content.
   *
   * Returns:
   *   "" if successful else reason if not.
   */
  std::string scan_media(const std::string& hashdb_dir,
                     const std::string& media_image_file,
                     const size_t step_size,
                     const bool disable_recursive_processing,
                     const hashdb::scan_mode_t scan_mode);

  /**
   * Read raw bytes at the media offset in the media image file.  Files
   * with EWF extensions (.E01 files) are recognized as media images.
   * Example media offsets are "1000" and "1000-zip-0".
   *
   * Parameters:
   *   media_image_file - Path to a media image file, which can be a
   *     raw file or an E01 file.
   *   media_offset - The offset into the media image file.
   *   count - The number of bytes to read.
   *   bytes - The raw bytes read.
   *
   * Returns:
   *   "" if successful else reason if not.
   */

  std::string read_media(const std::string& media_image_file,
                         const std::string& media_offset,
                         const uint64_t count,
#ifndef SWIG
                         std::string& bytes
#else
                         std::string& OUTPUT // bytes
#endif
                        );

  /**
```

```
    * Read raw bytes at the given offset in the media image file.  Files
    * with EWF extensions (.E01 files) are recognized as media images.
    * Example media offsets are "1000" and "1000-zip-0".
    *
    * Parameters:
    *   media_image_file - Path to a media image file, which can be a
    *     raw file or an E01 file.
    *   offset - The offset into the media image file.
    *   media_offset - The offset into the media image file.
    *   count - The number of bytes to read.
    *   bytes - The raw bytes read.
    *
    * Returns:
    *   "" if successful else reason if not.
    */
   std::string read_media(const std::string& media_image_file,
                          const uint64_t offset,
                          const uint64_t count,
#ifndef SWIG
                          std::string& bytes
#else
                          std::string& OUTPUT // bytes
#endif
                         );

   /**
    * Read the size of the media image file.  Files with EWF extensions
    * (.E01 files) are recognized as media images.
    *
    * Parameters:
    *   media_image_file - Path to a media image file, which can be a
    *     raw file or an E01 file.
    *   size - The size, in bytes, of the media image.
    *
    * Returns:
    *   "" if successful else reason if not.
    */
   std::string read_media_size(const std::string& media_image_file,
#ifndef SWIG
                               uint64_t& size
#else
                               uint64_t& OUTPUT // bytes
#endif
                              );

   // ************************************************************
   // import
   // ************************************************************
   /**
    * Manage all LMDB updates.  All interfaces are locked and threadsafe.
    * A logger is opened for logging the command and for logging
    * timestamps and changes applied during the session.  Upon closure,
    * changes are written to the logger and the logger is closed.
    */
   class import_manager_t {

     private:
     lmdb_hash_data_manager_t* lmdb_hash_data_manager;
     lmdb_hash_manager_t* lmdb_hash_manager;
     lmdb_source_data_manager_t* lmdb_source_data_manager;
     lmdb_source_id_manager_t* lmdb_source_id_manager;
```

```
    lmdb_source_name_manager_t* lmdb_source_name_manager;

    logger_t* logger;
    hashdb::lmdb_changes_t* changes;

    public:
#ifndef SWIG
    // do not allow copy or assignment
    import_manager_t(const import_manager_t&) = delete;
    import_manager_t& operator=(const import_manager_t&) = delete;
#endif

    /**
     * Open hashdb for importing.
     *
     * Parameters:
     *   hashdb_dir - Path to the hashdb data store to import into.
     *   command_string - String to put into the new hashdb log.
     */
    import_manager_t(const std::string& hashdb_dir,
                     const std::string& command_string);

    /**
     * The destructor closes the log file and data store resources.
     */
    ~import_manager_t();

    /**
     * Insert the repository_name, filename pair associated with the
     * source.
     *
     * Parameters:
     *   file_hash - The file hash of the source file in binary form.
     *   repository_name - A repository name to attribute the sources to.
     *   filename - The name of the source file.
     */
    void insert_source_name(const std::string& file_hash,
                            const std::string& repository_name,
                            const std::string& filename);


    /**
     * Insert or change source data.
     *
     * Parameters:
     *   file_hash - The file hash of the source file in binary form.
     *   filesize - The size of the source, in bytes.
     *   file_type - A string representing the type of the file.
     *   zero_count - The count of blocks skipped because they only
     *     contain the zero byte.
     *   nonprobative_count - The count of non-probative hashes
     *     identified for this source.
     */
    void insert_source_data(const std::string& file_hash,
                            const uint64_t filesize,
                            const std::string& file_type,
                            const uint64_t zero_count,
                            const uint64_t nonprobative_count);

    /**
     * Insert or change the hash data associated with the block_hash.
```

```
     * Use this during ingest where the file offset is guaranteed to
     * be new.
     *
     * Parameters:
     *   block_hash - The block hash in binary form.
     *   k_entropy - An entropy value for the associated block, scaled
     *     up by 1,000 for three decimal place precision.
     *   block_label - Text indicating the type of the block or "" for
     *     no label.
     *   file_hash - The file hash of the source file in binary form.
     */
    void insert_hash(const std::string& block_hash,
                     const uint64_t k_entropy,
                     const std::string& block_label,
                     const std::string& file_hash);

#ifndef SWIG
    /**
     * Insert or change the hash data associated with the block_hash.
     * Use this when merging existing sets of file offsets.
     *
     * Parameters:
     *   block_hash - The block hash in binary form.
     *   k_entropy - An entropy value for the associated block, scaled
     *     up by 1,000 for three decimal place precision.
     *   block_label - Text indicating the type of the block or "" for
     *     no label.
     *   file_hash - The file hash of the source file in binary form.
     *   sub_count - The number of file offsets to add for this file hash.
     */
    void merge_hash(const std::string& block_hash,
                    const uint64_t k_entropy,
                    const std::string& block_label,
                    const std::string& file_hash,
                    const uint64_t sub_count);
#endif

    /**
     * Import hash or source information from a JSON record.
     *
     * Parameters:
     *   json_string - Hash or source text in JSON format.
     *
     *   Example hash syntax:
     *     {
     *       "block_hash": "c313ac...",
     *       "k_entropy": 2500,
     *       "block_label": "W",
     *       "source_sub_counts": ["b9e7...", 2]
     *     }
     *
     *   Example source syntax:
     *     {
     *       "file_hash": "b9e7...",
     *       "filesize": 8000,
     *       "file_type": "exe",
     *       "zero_count": 1,
     *       "nonprobative_count": 4,
     *       "name_pairs": ["repository1", "filename1", "repo2", "f2"]
     *     }
     *
```

```
    * Returns:
    *   "" else error message if JSON is invalid.
    */
   std::string import_json(const std::string& json_string);

   /**
    * See if the file hash is in the database.
    *
    * Returns:
    *   true if the file hash is in the database.
    */
   bool has_source(const std::string& file_hash) const;

   /**
    * Return the file_hash of the first source in the database.
    *
    * Returns:
    *   file_hash if a first source is available else "" if DB
    *   is empty.
    */
   std::string first_source() const;

   /**
    * Return the next source in the database.  Error if last_file_hash
    *   does not exist.
    *
    * Parameters:
    *   last_file_hash - The previous source file hash in binary form.
    *
    * Returns:
    *   next file_hash if a next source is available else "" if at end.
    */
   std::string next_source(const std::string& file_hash) const;

   /**
    * Return the sizes of LMDB databases in the data store.
    */
   std::string size() const;

   /**
    * Return the number of records in the hash data store.
    */
   size_t size_hashes() const;

   /**
    * Return the number of sources.
    */
   size_t size_sources() const;
};

// ************************************************************
// scan
// ************************************************************
/**
 * Manage LMDB scans.  All interfaces are locked and threadsafe.
 */
class scan_manager_t {

  private:
   lmdb_hash_data_manager_t* lmdb_hash_data_manager;
   lmdb_hash_manager_t* lmdb_hash_manager;
```

```
    lmdb_source_data_manager_t* lmdb_source_data_manager;
    lmdb_source_id_manager_t* lmdb_source_id_manager;
    lmdb_source_name_manager_t* lmdb_source_name_manager;

    // support find_expanded_hash_json when optimizing
    locked_member_t* hashes;
    locked_member_t* sources;

    // low-level find interfaces
    std::string find_expanded_hash_json(const bool optimizing,
                                        const std::string& block_hash);
    std::string find_hash_count_json(const std::string& block_hash) const;
    std::string find_approximate_hash_count_json(
                                        const std::string& block_hash) const;
    public:
#ifndef SWIG
    // do not allow copy or assignment
    scan_manager_t(const scan_manager_t&) = delete;
    scan_manager_t& operator=(const scan_manager_t&) = delete;
#endif

    /**
     * Open hashdb for scanning.
     *
     * Parameters:
     *   hashdb_dir - Path to the database to scan against.
     */
    scan_manager_t(const std::string& hashdb_dir);

    /**
     * The destructor closes read-only data store resources.
     */
    ~scan_manager_t();

#ifndef SWIG
    /**
     * Find hash, return hash and source information.
     *
     * Parameters:
     *   block_hash - The block hash in binary form.
     *   k_entropy - An entropy value for the associated block, scaled
     *     up by 1,000 for three decimal place precision.
     *   block_label - Text indicating the type of the block or "" for
     *     no label.
     *   count - The total count of file offsets related to this hash.
     *   source_sub_counts - Set of source and source sub-counts for each
     *     source associated with this hash.
     *
     * Returns:
     *   True if the hash is present, false if not.
     */
    bool find_hash(const std::string& block_hash,
                   uint64_t& k_entropy,
                   std::string& block_label,
                   uint64_t& count,
                   source_sub_counts_t& source_sub_counts) const;
#endif

    /**
     * JSON block_hash export text else "" if hash is not there.
     *
```

```
     * Parameters:
     *   block_hash - The block hash in binary form.
     *
     * Returns:
     *   JSON block_hash export string if hash is present, false and ""
     *   if not.  Example syntax:
     *
     *     {
     *       "block_hash": "c313ac...",
     *       "k_entropy": 2500,
     *       "block_label": "W",
     *       "count": 2,
     *       "source_sub_counts": ["b9e7...", 2]
     *     }
     */
    std::string export_hash_json(const std::string& block_hash) const;

    /**
     * JSON file_hash export text else "" if file hash is not there.
     *
     * Parameters:
     *   file_hash - The file hash of the source file in binary form.
     *
     * Returns:
     *   JSON file hash export text if file hash is present, false
     *   and "" if not.  Example syntax:
     *
     *     {
     *       "file_hash": "b9e7...",
     *       "filesize": 8000,
     *       "file_type": "exe",
     *       "zero_count": 1,
     *       "nonprobative_count": 4,
     *       "name_pairs": ["repository1", "filename1", "repo2", "f2"]
     *       }
     */
    std::string export_source_json(const std::string& file_hash) const;

    /**
     * Find hash count.  Faster than find_hash.  Accesses the hash
     * information store.
     *
     * Parameters:
     *   block_hash - The block hash in binary form.
     *
     * Returns:
     *   The total count of file offsets related to this hash.
     */
    size_t find_hash_count(const std::string& block_hash) const;

    /**
     * Find the approximate hash count.  Faster than find_hash, but can
     * be wrong.  Accesses the hash store.
     *
     * Parameters:
     *   block_hash - The block hash in binary form.
     *
     * Returns:
     *   The count of file offset entries expected to be associated
     *   with this hash.  This value can be wrong because there can be
     *   collisions with truncated hash values.
```

```
     */
    size_t find_approximate_hash_count(const std::string& block_hash)
        const;

    /**
     * Find source data for the given source ID, false on no source ID.
     *
     * Parameters:
     *   file_hash - The file hash of the source file in binary form.
     *   filesize - The size of the source, in bytes.
     *   file_type - A string representing the type of the file.
     *   zero_count - The count of blocks skipped because they only
     *     contain the zero byte.
     *   nonprobative_count - The count of non-probative hashes
     *     identified for this source.
     *
     * Returns:
     *   True if file binary hash is present.
     */
    bool find_source_data(const std::string& file_hash,
#ifdef SWIG
                          uint64_t& OUTPUT,       // filesize
                          std::string& OUTPUT,    // file_type
                          uint64_t& OUTPUT,       // zero_count
                          uint64_t& OUTPUT        // nonprobative_count
#else
                          uint64_t& filesize,
                          std::string& file_type,
                          uint64_t& zero_count,
                          uint64_t& nonprobative_count
#endif
                         ) const;

#ifndef SWIG
    /**
     * Find source names for the given source ID, false on no source ID.
     *
     * Parameters:
     *   file_hash - The file hash of the source file in binary form.
     *   source_names - Set of pairs of repository_name, filename
     *     attributed to this source ID.
     *
     * Returns:
     *   True if file binary hash is present.
     */
    bool find_source_names(const std::string& file_hash,
                           source_names_t& source_names) const;
#endif

    /**
     * Find hash, return JSON text else "" if not there.
     *
     * Parameters:
     *   scan_mode - The mode to use for performing the scan.  Controls
     *     scan optimization and returned JSON content.
     *   block_hash - The block hash in binary form.
     *
     * Returns:
     *   JSON text if hash is present, false and "" if not.  Example
        syntax
     *   based on mode:
```

```
 *      EXPANDED - always return all available data.  Example syntax:
 *        {
 *          "block_hash": "c313ac...",
 *          "k_entropy": 2500,
 *          "block_label": "W",
 *          "count": 2,
 *          "source_list_id": 57,
 *          "sources": [{
 *            "file_hash": "f7035a...",
 *            "filesize": 800,
 *            "file_type": "exe",
 *            "zero_count": 1,
 *            "nonprobative_count": 2,
 *            "names": ["repository1", "filename1", "repo2", "f2"]
 *          }],
 *          "source_sub_counts": ["b9e7...", 2]
 *        }
 *      EXPANDED_OPTIMIZED - return all available data the first time
 *        but suppress hash and source data after.  Example syntax
 *        when suppressed:
 *        { "block_hash": "c313ac..." }
 *      COUNT - Return the count of source offsets associated with this
 *        hash.  Example syntax:
 *        { "block_hash": "c313ac...", "count": 1 }
 *      APPROXIMATE_COUNT - Return the approximate count of source
 *        offsets associated with this hash.  The approximate count
 *        is logarithmic and can be wrong because there can be
     collisions
 *        with truncated hash values.  Faster than COUNT because it
 *        accesses the hash_store.  Example syntax:
 *        { "block_hash": "c313ac...", "approximate_count": 1 }
 */
std::string find_hash_json(const scan_mode_t scan_mode,
                           const std::string& block_hash);

/**
 * Return the first block hash in the database.
 *
 * Returns:
 *   block_hash if a first hash is available else "" if DB is empty.
 */
std::string first_hash() const;

/**
 * Return the next block hash in the database.  Error if last hash
 *   does not exist.
 *
 * Parameters:
 *   last_block_hash - The previous block hash in binary form.
 *
 * Returns:
 *   block_hash if a next hash is available else "" if at end.
 */
std::string next_hash(const std::string& block_hash) const;

/**
 * Return the file_hash of the first source in the database.
 *
 * Returns:
 *   file_hash if a first source is available else "" if DB
 *   is empty.
```

```
     */
    std::string first_source() const;

    /**
     * Return the next source in the database.  Error if last_file_hash
     *   does not exist.
     *
     * Parameters:
     *   last_file_hash - The previous source file hash in binary form.
     *
     * Returns:
     *   next file_hash if a next source is available else "" if at end.
     */
    std::string next_source(const std::string& file_hash) const;

    /**
     * Return the sizes of LMDB databases in JSON format.
     */
    std::string size() const;

    /**
     * Return the number of hash records.
     */
    size_t size_hashes() const;

    /**
     * Return the number of sources.
     */
    size_t size_sources() const;
  };

  // ************************************************************
  // scan_stream
  // ************************************************************
  /**
   * Provide a threaded streaming scan interface.  Use put to enqueue
   * arrays of scan input.  Use get to receive arrays of scan output.
   *
   * If a thread cannot properly parse unscanned data, it will emit a
   * warning to stderr.
   */
  class scan_stream_t {
    private:
    const int num_threads;
    ::pthread_t* threads;
    scan_stream::scan_thread_data_t* scan_thread_data;
    bool done;

#ifndef SWIG
    // do not allow copy or assignment
    scan_stream_t(const scan_stream_t&);
    scan_stream_t& operator=(const scan_stream_t&);
#endif

    public:
    /**
     * Create a streaming scan service.
     *
     * Parameters:
     *   scan_manger - The hashdb scan manager to use for scanning.
     *   hash_size - The size, in bytes, of a binary hash, 16 for MD5.
```

```
    *    scan_mode - The mode to use for performing the scan.  Controls
    *      scan optimization and returned JSON content.
    */
  scan_stream_t(hashdb::scan_manager_t* const scan_manager,
                const size_t hash_size,
                const hashdb::scan_mode_t scan_mode);

  /**
   * Release scan_stream resources.
   */
  ~scan_stream_t();

  /**
   * Submit a string containing an array of records to scan.
   *
   * Paramters:
   *    unscanned_data - An array of records to scan, packed without
   *      delimiters.  Each record contains:
   *      - A binary hash to scan for, of length hash_size.
   *      - A 2-byte unsigned integer in native-Endian format indicating
   *        the length, in bytes, of the upcoming binary label associated
   *        with the scan record.
   *      - A binary label associated with the scan record, of the
   *        length just indicated.
   */
  void put(const std::string& unscanned_data);

  /**
   * Receive a string containing an array of records of matched scanned
   * data or "" if no data is available.
   *
   * Returns:
   *    An array of records of matched scanned data or "" if no data
   *    is available.  Each record conatins:
   *    - A binary hash that matched, of length hash_size.
   *    - A 2-byte unsigned integer in native-Endian format indicating
   *      the length, in bytes, of the upcoming binary label associated
   *      with the hash that matched.
   *    - A binary label associated with the scan record, of the
   *      length just indicated.
   *    - A 4-byte unsigned integer in native-Endian format indicating
   *      the length, in bytes, of the upcoming JSON text associated
   *      with the hash that matched.
   *    - JSON text formatted based on the scan mode selected, of the
   *      length just indicated.
   */
  std::string get();

  /**
   * Returns true if scan_stream is empty, meaning that there is no
   * unscanned data left to scan and there is no scanned data left to
   * retrieve.  If not empty, a thread yield is issued so that the
   * caller can busy-wait with less waste.
   *
   * Returns:
   *    true if scan_stream is empty.
   */
  bool empty();
};

// ****************************************************
```

```
  // timestamp
  // ************************************************************
  /**
   * Provide a timestamp service.
   */
  class timestamp_t {

    private:
    struct timeval* t0;
    struct timeval* t_last_timestamp;

    public:

    /**
     * Create a timestamp service.
     */
    timestamp_t();

    /**
     * Release timestamp resources.
     */
    ~timestamp_t();

#ifndef SWIG
    // do not allow copy or assignment
    timestamp_t(const timestamp_t&) = delete;
    timestamp_t& operator=(const timestamp_t&) = delete;
#endif

    /**
     * Create a named timestamp and return a JSON string in format
     * {"name":"name", "delta":delta, "total":total}.
     */
    std::string stamp(const std::string &name);
  };
}

#endif
```