Suspicious contents

M1 project report

Matéo Delerue-Houard, Sylia Bouimedj

UNIVERSITÉ
CAEN
NORMANDIE

Introduction

During a forensic investigation, digital storage medias must be analyzed to find potential evidence of a suspected crime. However, criminals will often try to hide such incriminating files. Sometimes, advanced methods are used such as encryption or complex steganography. But most of the time, suspects don't have a very high technical level and will thus hide evidences by basic means. In our experiments, we mostly focused on file extension tampering, files moved in unexpected locations, and files marked as hidden.

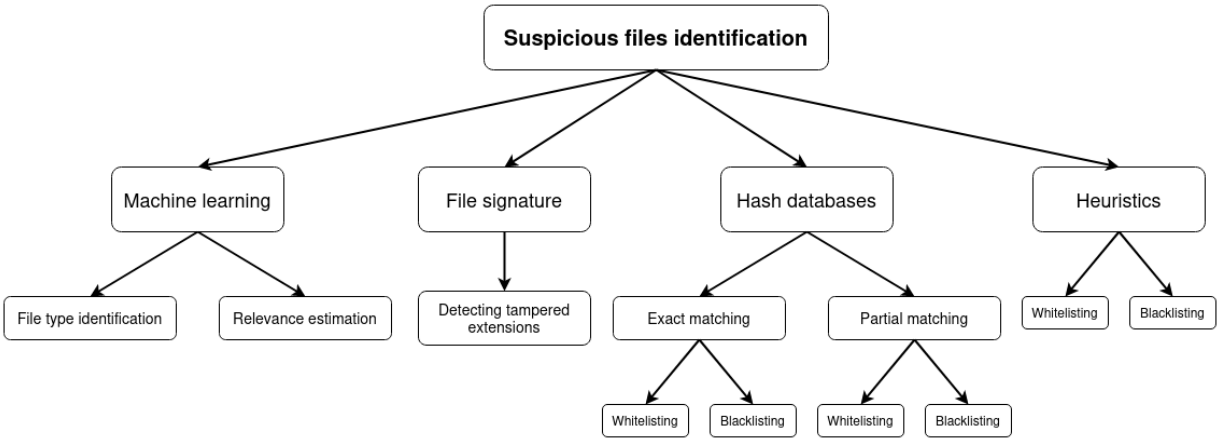
Analysis of whole drives cannot be done manually as modern storage devices can reach terabytes capacities and the time available for investigation is limited. Therefore, we need automated tools to perform a first processing quickly in order to reduce the work for human investigators. In this paper, we'll focus on file triage, namely filter uninteresting files while prioritizing suspicious contents for further analysis.

Our goal is first and foremost to detect intentionally concealed files, but also to filter known benign content, and all that as quickly as possible.

Table of contents

State of the art

Before starting to design our own analysis software, we explored the current state of the art of file triage methods in order to take advantage of the most promising techniques. This led us to establish the following taxonomy:



File signature

For a given file, it's possible to identify its type based on its content and not on its extension. If the type found is different from what the extension suggests, we can deduct that the extension has been intentionally changed in an attempt to conceal the file and so it can be considered suspicious.

Many file formats contain a constant sequence of bytes, usually at the very start of the file, that identify the file type. This sequence is called the *"magic number"*. It's used to identify file types on UNIX systems. A database containing magic numbers in correspondence with their associated file types is usually present at `/usr/share/misc/magic.mgc` on Debian based distributions. Most file identification programs use a similar database in addition to heuristics and format-specific analysis to determine file type.

In a study from 2023[1], it has been shown that we can achieve great results by combining multiple tools together. On a 1M files dataset, Fidentify (a tool using the same database as the famous open source tool PhotoRec) achieves a 98.1% accuracy. Researchers have reached 98.3% by combining it with *ForENSIque*, a software created by ENSICAEN students.

This technique thus works very well to detect files with tampered extension. Depending on the investigation case, we can prioritize the analysis by assigning higher scores to specific types of files. For instance, documents in financial fraud investigations or

multimedia files in a child abuse case. However, this technique is useless if the extension was left intact but the file was hidden by other ways, such as by being placed into an unusual, usually legitimate directory. This is why other methods are needed.

Machine learning

File type identification can also be achieved using machine learning based approaches. This is what Sester et al.[2] did in 2021, by experimenting with support vector machines (SVM) and neural networks. They reached a maximum accuracy of 91.4% using an SVM with a linear kernel.

This approach can be enhanced by directly carrying out the training on the relevance of the files instead of their file type. That simplifies the problem by removing one level of indirection and allows the identification of suspicious files based on other factors. With this technique, Serhal & Le-Khac[3] achieved a 99.8% accuracy score using random forests, K-nearest neighbor and classification and regression trees.

Hash databases

Sometimes, suspects try to hide their files inside benign directories in an attempt to get their data *lost* in the file system hierarchy under legitimate folders. The files can thus be located inside system directories such as `C:\Windows\System32`, in software data directories such as `C:\Program Files\Mozilla Firefox` or in cache folders such as `C:\Users\User\AppData\Local\` subdirectories. Locating hidden files then becomes the needle and haystack problem.

One solution is to compute the hash of all files inside these known folders and compare them with a pre-computed hash database[4]. Such a database contains a list of file hash that are known to be legitimate (system files, applications files, etc.). If the hash of a file is in this database, it can then be considered benign. However, if a file is not found in the database but is located inside a known folder, it could mean it does not belong to this directory and has been placed here intentionally by the user. Therefore, the file can be considered suspicious.

Fortunately, such hash databases already exist such as the National Software Reference Library Reference Data Set or RDS for short, distributed by the US government. This database contains a large set of file hashes belonging to known software and operating systems, along with information on their provenance and metadata such as original file name and size. If a file hash is present in this database, we can conclude with a high level of confidence that this file is not suspicious.

Some programs have already been designed to perform the database lookup for a given set of files. The md5deep suite is a collection of programs that can be used to compute the hash of all files within a directory and display only files for which the hash is not present in a given hash database.

The method can also be used the other way around: comparing file hashes with a pre-computed database of **illegal** content. That way, if a file matches, it can be considered suspicious with a very high degree of certainty. However, this method will only detect already known content such as multimedia files obtained in violation of copyright. It won't be able to detect new or unique content such as personal pictures or confidential corporate documents.

The main limitation with methods based on hash databases is that they only work if files are exactly the same. If at least one byte differs, the hashes will be different and so no match will be found. To overcome this issue, several studies[5][6][7] experimented with sector hashing and piecewise hashing, with the aim to partially match similar files. This way, new content which is similar enough to already known files will be detected or whitelisted (depending on the type of the database). Some existing tools such as ssdeep and sdhash can achieve that to a certain extent.

Additional heuristics

In addition to the previously described methods, some heuristics can be employed while searching the file system to further improve the detection rate and defeat other ways to hide content, but also to filter uninteresting files. The most important heuristics are described here.

Hidden attribute

The NTFS file system, which is the default for Windows systems, allows setting a *hidden* attribute on files and folders, thus making them disappear from the Windows Explorer with default configuration. This feature can be used by the user to conceal data[8]. However, since this technique modifies the file's metadata in a clearly identifiable way, it's rather easy to list all files with the *hidden* attribute set. If some of them are located within user data directories (`Documents`, `Images`, `Desktop`...) we can assume that the attribute has been deliberately set by the user, and thus the file can be considered suspicious.

Timestamp coherence

Most of the time, software files created during the installation are written at the same date and thus have very near timestamps. Therefore, if a software folder contains a file with a timestamp (created or modified) which is more recent than the rest of the directory, it could mean it have been placed here later manually, probably with the aim to hide the file. This can be the sign of a suspicious content.

Size filtering

According to some studies[9][10], interesting files usually have a size which is above a specific threshold, depending on the file type. We can therefore filter all files that are smaller than this size threshold as they are unlikely to contain relevant content.

Filtering groups of files created in a very short time

According to Neil C. Rowe[10:1], files with very near creation timestamps are likely to be uninteresting:

> TM, clustered creation times: Files with the same creation time within a short period as that of many other files on the same drive. Such time clusters suggest automated copying from an external source, particularly if the rate of creation exceeded human
> limits

Thus, groups of file with near creation time can be filtered. However, this rule should be used carefully as it could exclude interesting files as well, for example if the user extracted an archive content.

Similar metadata

A file with similar name or timestamp as another detected suspicious file can be considered suspicious, as it could indicate a group of related files[11][12].

Interesting date and file name

Sometimes, forensic investigators know some details about the evidences they are looking for, such as in which time range they were created, or how they are likely to be named. Thus, the analysis of files with timestamps close to the given date[11:1] or with suspicious keywords in their names can be prioritized.

Timestamp forgery

Suspects can use software to change timestamps of files they want to hide. However, many of these software only change visible timestamps, while leaving other hidden metadata intact. By leveraging the file system design, it's therefore possible to detect timestamp incoherence[13] and thus conclude to an intentional tampering. A file with detected timestamp forgery should be considered as suspicious.

Our implementation

The analysis program has been written in C++. It incorporates an implementation of file type identification, hash database based filtering and detection, plus some other heuristics, including hidden file detection as well as timestamp and filename targeted searches. All of these detection methods that can be switched on and off at runtime, enabling the effectiveness of each technique to be studied individually or in combination. We followed a modular architecture pattern to be easily able to add new detection rules.

File type identification

The type detection has been implemented by leveraging `fidentify`, as it was identified as the most accurate tool according to a study from 2023[1:1]. To speed up the analysis, we decided to statically link the `fidentify` binary into our software instead of invoking the executable for each file. This way, we replace all the overhead of process creation by a simple function call, which make the analysis faster and more efficient. To achieve this, we included the `testdisk` source code as a `git` submodule inside our tree, and modified the `fidentify.c` file to be callable as a library. Then we integrated the `testdisk` build process into our build system (CMake).

As multiple file extensions can actually refer to the same file format, we created a simple extension map which associates generic extensions (i.e. `.zip`) with a set of legitimate extensions which could correspond to this file format (i.e. `.jar`, `.apk`). It's currently implemented as `std::unordered_map<std::string_view, std::unordered_set<std::string_view>>`, allowing to perform lookups in constant time on average.

Hash database

Our file hashing implementation follows the NIST's RDS schema: file are hashed individually and looked up in a SQL database. The hash is either a MD5, SHA1 or SHA256 of the file content. It doesn't include file metadata such as filename, location, or timestamps. After a quick benchmark, we decided to use SHA1 as it turned out to be the fastest of the three.

```
 $ openssl speed md5 sha1 sha256
[...]
The 'numbers' are in 1000s of bytes per second processed.
type     16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
md5      33575.38k   107869.22k   269526.73k   443219.84k   530852.52k
463983.96k
sha1     46415.05k   166869.38k   604677.97k  1285292.37k  1755116.89k
1824964.61k
sha256   67907.21k   236246.27k   658070.61k   892167.02k  1121880.75k
1155674.76k
```

MD5 should theoretically be faster but SHA1 leverages hardware acceleration on modern processors.

Our current implementation uses SQLite as its database system for simplicity. To speed up hash lookups, we created an index over the `sha1` column.

The analysis program computes hashes for each file in the current directory (non-recursively), and lookup them in the database. Recognized files are whitelisted, and thus do not undergo further analysis. If more than half of the files in the current directory are matched in the database, the current directory is considered *known*. Each *unknown* file in a *known* directory or subdirectory (recursively) is considered suspicious.

Heuristics

Hidden files

Our hidden file detector expects the target root directory of the analysis to be the root mountpoint of an operating system. In other words, the analysis must be performed on a whole drive for this detector to be useful.

Currently, hidden files are searched only on Windows, and platforms following the XDG standard. On Windows, files with the hidden attribute placed under `Desktop`, `Downloads`, `Documents`, `Music`, `Pictures` or `Videos` folders are considered suspicious. On other platforms, files located within directories specified in `~/.config/user-dirs.dirs` with names starting with a dot are considered suspicious. Usually, this concerns `DESKTOP`, `DOWNLOADS`, `DOCUMENTS`, `MUSIC`, `PICTURES`, `VIDEOS`, `TEMPLATES` and `PUBLIC` folders.

Since the analysis program is designed to run on POSIX systems, we use the ntfs-3g extended attributes to detect the hidden flag on NTFS partitions. The use of another NTFS driver thus requires it to set compatibles attributes.

Size filtering

All files under 6 bytes are skipped. Raising this threshold will speed up the analysis process and might lower the amount of false positives, but may also miss some suspicious files.

Keyword and timestamp search

When the investigator has some information about the sought evidences, they can provide some additional details at runtime to the program, such as a list of suspicious file name substrings and a target timestamp range. Files matching these criteria will be reported as suspicious.

Performance optimization

In real forensic cases, time is limited. It's crucial to deliver results as quickly as possible. Therefore, we tried to optimize our existing software in order to achieve higher speeds.

Profiling

We started by profiling the analysis program with callgrind to identify the performance bottlenecks of the code.

```
 $ valgrind --tool=callgrind ./analyze ...

$ callgrind_annotate callgrind.out.96
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.96' (creator: callgrind-3.22.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 7471024364
Trigger: Program termination
Profiled target:  ./analyze ... (PID 96, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
126,518,478,058 (100.0%)  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir                      file:function
--------------------------------------------------------------------------------
119,577,351,348 (94.51%)  ???:SHA1Transform [/usr/lib/libmd.so.0.1.0]
  4,961,008,303 ( 3.92%)  ???:0x0000000000157c80 [/usr/lib/libc.so.6]
    734,166,789 ( 0.58%)  ???:SHA1Update [/usr/lib/libmd.so.0.1.0]
```

As shown above, we found that a very high percentage of the global program's instructions are executed within the `SHA1Transform()` function. This function is part of libmd, and is called by the `SHA1File()`, which we directly use in the hash database code. In other words, this means most of the time is spent computing SHA1 digests of file

contents with libmd. Therefore, we decided to try other libraries, with the hope to find a faster SHA1 implementation.

SHA1 implementations benchmark

To compare the speed of different SHA1 implementations, we wrote a simple benchmark code that computes 10,000 iterations of the SHA1 algorithm over a 1MB random buffer with all selected implementations, while measuring overall time spent by each of them.

We selected OpenSSL, an independent C library called racrypt, and a simple implementation leveraging x86 SHA1 native instructions (SHA-Intrinsics). We also included libmd for reference in the benchmark. The test was run on a Debian Linux machine with an x64 processor, with `-O3` and `-march=native` compile options. Here are the results we obtained:

```
 $ ./run.sh
Compiling...
Starting...
Racrypt: 10797ms
Intrinsics: 9131ms
OpenSSL: 8967msms
libmd: 28791ms
```

We can see that libmd is the slowest of these implementations, almost 3 times slower than the others. The fastest are OpenSSL and SHA-Intrinsics, with a ranking changing from run to run. OpenSSL being a very popular and widely available library, we decided to choose it.

Final benchmark

After editing our code to replace libmd function calls by their OpenSSL counterparts, we ran benchmarks using hyperfine to measure the global speedup by comparing the execution times of both versions.

```
 $ hyperfine './analyze_libmd --timestamp 0 --timestamp-threshold 5 -d Testing/
rds.sqlite3 -k Testing/keywords.txt Testing/root' './analyze_openssl --timestamp 0
--timestamp-threshold 5 -d Testing/rds.sqlite3 -k Testing/keywords.txt Testing/
root'
Benchmark 1: ./analyze_libmd --timestamp 0 --timestamp-threshold 5 -d Testing/
rds.sqlite3 -k Testing/keywords.txt Testing/root
  Time (mean ± σ):      2.572 s ±  0.139 s    [User: 1.303 s, System: 1.267 s]
  Range (min … max):    2.423 s …  2.824 s    10 runs

Benchmark 2: ./analyze_openssl --timestamp 0 --timestamp-threshold 5 -d Testing/
rds.sqlite3 -k Testing/keywords.txt Testing/root
  Time (mean ± σ):     791.3 ms ±  21.7 ms    [User: 471.0 ms, System: 319.8 ms]
  Range (min … max):   754.3 ms … 819.0 ms    10 runs

Summary
  ./analyze_openssl --timestamp 0 --timestamp-threshold 5 -d Testing/rds.sqlite3 -
k Testing/keywords.txt Testing/root ran
    3.25 ± 0.20 times faster than ./analyze_libmd --timestamp 0 --timestamp-
threshold 5 -d Testing/rds.sqlite3 -k Testing/keywords.txt Testing/root
```

In conclusion, for the same arguments and the same target directory, our analysis program is now more than 3 times faster. That's an impressive performance gain.

Evaluation protocol

To evaluate the efficiency and accuracy of the technique implementations, we applied the following protocol:

1. Create a legitimate drive by installing an operating system on an empty storage device
2. Hide content from the dataset by:
    - Placing files and directories at random places
    - Altering their extensions according to the chosen probability distribution by:
        - Replacing the extension by a known extension
        - Replacing the extension by a randomly generated extension
        - Removing the extension
        - Keeping the original extension
    - Creating hidden files and directories in user's folders (not implemented yet)
3. Execute each technique on the directory tree and measure their detection rate as well as their false-positive rate

Dataset

To evaluate the file type identification method, we need a set of files of various formats. For the hash database based technique, we only require that the files of the dataset are not part of system or application data, as they could otherwise be whitelisted. The other detection methods exclusively rely on metadata. Therefore, the semantic of the files is not relevant. The dataset can thus be composed of arbitrary user files.

Hash database creation

Due to a very poor internet connection at home, and disk usage restrictions at the university, we haven't been able to download the 106GB of the entire NIST's RDS. And because the NIST doesn't provide a way to download a subset of the database for specific operating systems, we decided to make our own hash database. In order to populate it, we wrote a simple C++ program that explores a directory tree and computes the SHA1 hash of all regular files found inside. The tool follows the RDS database schema, but only stores digest values and filenames to keep the database small. It also handles the index creation. We execute this tool right after the OS installation is complete, before adding suspicious contents on the drive.

Automation

To automate the evaluation process, we wrote a small python script that hides content within the file system. It takes two folders as arguments: the dataset folder containing files and directories to hide, and the root directory tree in which to hide content.

For each source files and directories, the script walk into the destination directory by selecting subdirectories at random. At each level of the hierarchy, the current location is chosen with a probability of 5%. If the script reaches a directory containing no subdirectories, it resumes the walk from the root.

This process is not perfect as it tends to select directories near to the root with a higher probability, but it's more efficient as it avoids exploring the entire file system.

When a destination directory has been selected, the corresponding source file is copied there and its extension may be randomly changed. With a probability of 40%, the extension is picked among a list of known file extensions. With 20% chance, the extension is generated randomly. With a probability of 20%, the extension is simply removed. The remaining 20% just keep the extension unchanged.

With a 20% probability, the file is hidden. On Windows OS roots, it means setting the *hidden* NTFS flag. On others platform roots, the filename is prepended with a dot.

In order to automatically generate an evaluation report, we added a feature to the analysis software that allows to write the detected suspicious files in text files on disk. There is one file per detector, containing the paths of the detected files. Once the analysis is completed, we can use the `stats.py` script to generate a report from these result files, showing the accuracy and the number of false positive of each detector, as well as in combination.

Experiment results

We run the experiment on a freshly installed Linux Mint 21.3 Cinnamon edition inside a Libvirt virtual machine. Just after installation was complete, we ran the database creation tool on the guest directories `/usr`, `/etc`, `/boot`, `/root` and `/home` directories. The other directories `/var`, `/tmp` and `/run` contain changing files. They are unlikely to be identical on different machines. Thus, we don't include them in the database, at the exception of `/var/lib`. Same for `/proc`, `/sys` and `/dev`, which are virtual file systems. `/bin`, `/sbin`, `/lib`, `/lib32`, `/libx32`, and `/lib64` are just symbolic links to `/usr`. `/opt` and `/srv` are empty by default.

To avoid modifying the original disk, we first mounted it as read-only. However, this prevents us from copying new files inside. Therefore, we created a second mountpoint using overlayfs to simulate writes. This enables us to virtually hide content inside the file system without modifying it, while allowing us to undo the changes in order to run new tests without interference.

Here are the results we got from a 100 files set hidden randomly using our `hide.py` script:

Techniques	Detection rate	False-positives
File type identification	73%	1099
Hash database	35%	11
Timestamp search	100%	0
Hidden files detection	3%	0
All combined	100%	1108

Analysis

File type identification gives a huge amount a false-positives. This is mainly due to the fact that GNU/Linux systems tend to use extensions in order to specify a file semantic

rather than a file format. For example, the file `/var/log/Xorg.0.log.old` is just a log file. The extensions `.old` is used to specify that it's an old version of the original file `/var/log/Xorg.0.log`. Similarly, `/usr/sbin/mkfs.ext4` is an executable file. The `.ext4` extension specifies that it operates on EXT4 file systems, unlike `/usr/sbin/mkfs.ext2` which works on EXT2, or `/usr/sbin/mkfs.fat` that's used for FAT file systems. Sometimes, no extension is used at all, as most programs in the `/usr/bin` directory.

To overcome this issue, we could allow the analysis program to filter files according to their detected types. For instance, in the case of a CSAM investigation, we could only show files detected as image or video for which the actual extension do not correspond. This way, all libraries, executable, configuration and text files would be filtered out while still detecting media files with tampered extensions. By reducing the search field, this method would significantly reduce the amount of false-positives, but on the other hand, it would miss not requested file types.

However, on Windows systems, the accuracy of the extension tampering detection method should be higher.

Hash database based detection return a small amount of false-positive. In our experiments, this is due to unknown directories like `/var` containing copies of known files (for example originating from `/etc`). In real cases however, we can expect the number of false-positive to be higher, as the actual state of the system might differ from the database.

The low detection rate of the method is explained by two reasons:

First, files hidden in directories which have not been included in the database populating process won't be detected. In our experiment, files located in `/tmp` or in `/var/log` have not been detected as these directories were excluded from the hashing process. This is unfortunately an expected limitation of this method.

Secondly, files hidden in directory containing no files won't be detected either. Let's take the example of the `/usr` directory:

```
/usr
├── bin
├── include
├── lib
├── local
├── sbin
├── share
└── Nasty hidden file.dat
```

The `/usr` directory originally contains only subdirectories and no files. Therefore, `/usr` won't be recognized as a known directory as it contains no known files, even though

`/usr` subdirectories contain only known files. This issue is due to the way the hashing database has been designed. In its current form, the database only stores hashes of files. Directories are not taken into consideration at all. A flat file system with every files under the root would produce exactly the same database as if the files were dispatched in a complex directory hierarchy. To fix this issue, we have to compute hashes of directories themselves, in the manner of a Merkle tree. In this example, as all subdirectories are known, `/usr` would be marked as known in the database too. Thus, files hidden under `/usr` would be detected.

During our experiment, we set the target timestamp range to the day the files were hidden. Therefore, all of them have been correctly detected. This method didn't report any false positive as the virtual machine has not been started that day. In real cases, we can expect to detect fewer files and have a greater rate of false positives.

The heuristic detecting hidden files only flagged 3 files. This is because only 3 files were marked hidden **and** placed in common users directories.

When we mix all these methods, we can reach a very high detection rate by combining their respective strengths. The amount of false positive may decrease in certain of our experiments, but this is not guaranteed. In the current state of the software, when a detector incorrectly reports a file as suspicious, the error can only be rectified if its hash is present in the hash database, or if it's a too small file. In other words, any false positives larger than the size threshold located in directories non targeted by the hash database won't be avoided by combining the existing detection methods.

Areas for improvement

As explained earlier, redesigning the hash database to store Merkle trees of directories would probably improve the overall accuracy and detection rate of the tool, but would break compatibility with the NIST's RDS. Therefore, we should create a new database format for the Merkle trees based method, while keeping the program compatible with the standard RDS schema.

Additionally, the software integrability could be improved. Even though we added the ability to return the results in a structured form, we'd like to add support for a more standard and robust format such as JSON or TOML, or even allow the results to be stored in a database. That way, the findings of the analysis would be even easier to be processed by other tools, and would help integration with larger frameworks.

The last improvement axis is about performance. Since the time in which forensic investigator must give results is limited, it's essential to make the analysis as fast as possible. First, we could speed up I/O by leveraging memory mapped files instead of standard streams for large contents. This should reduce the time spent to read file

content by avoiding system calls overhead and useless copies. Then, the main exploration and analysis process could be parallelized. A thread pool could be implemented, to which the analysis tasks of each subdirectory would be dispatched. This has the potential to multiply the analysis speed by the number of processor core, which will be a significant performance improvement. Obviously, the individual analysis components can be enhanced too. For instance, the keyword matching part could be speeded up by leveraging the Aho–Corasick algorithm, thus reducing the search complexity, and the DBMS used for hash lookups could be replaced by a high-performance one such as ClickHouse or DuckDB. The hashing algorithm used to match files could also be changed to a faster, possibly non-cryptographic one such as wyhash or XXH3. However, this would imply to recompute the hash database and would also break compatibility with the NIST's RDS.

Conclusion

This paper outlines the progress and findings of our forensic project. With the goal to detect suspicious files hidden in a directory hierarchy, we have successfully implemented several methods to address this objective: extension tampering detection by file type identification, a hash database based approach following the NIST's RDS pattern, suspicious hidden files detection, size based filtering, and a time and filename targeted search. Extension tampering detection achieves a greater detection rate but is less accurate with a very high number of false positives. On the other hand, the hash database technique and hidden files detection discover fewer files but are much more precise. Additionally, timestamp and filename search can turn out to be very powerful tools if some information about the case are known. By combining all of these methods, plus a size-based filter, we were able to achieve very high overall detection rates while keeping the ability to maintain a low number of false positives in certain cases. Moreover, we have other leads to improve the accuracy and detection rate of our tool.

If we had to continue this project, we would have tried to enhance the existing detection methods, and to optimize the software's performance, enabling faster and more efficient detection of concealed files.

Overall, the results obtained so far demonstrate promising progress in our pursuit of developing an investigation tool for detecting suspicious contents within storage devices. We hope that our solution is of sufficiently good quality and is accurate enough to be actually helpful in real-life use cases.

Tasks distribution

This report has been fully written by Matéo. He also performed the code profiling and benchmarks, as well as the experiments, and designed the slides for the two presentations.

Due to Sylia's lack of work (despite initial equal planning of task distribution), all of the code currently used by the program has been completely designed and written by Matéo, including the hashing program, the various python scripts, and the build system configuration. The documentation has also been written by Matéo.

Sylia has written some proof-of-concept C code for heuristics-based detection under the `src/heuristique` directory of the source tree, which cannot be used as-is.

Even though Matéo designed an interface for detectors written in C (`src/analyser/ExternFileBasedDetector.cpp`) to allow Sylia to integrate her code to the project (because she has little experience with C++), explained her how to use it, and provided an example of such integration (`src/analyser/externs/size_filter.h`), Sylia did not add her code to the analysis program. Therefore, Matéo reimplemented it in C++ in the main program.

Bibliography

1. Adrien Dubettier, Tanguy Gernot, Emmanuel Giguet, Christophe Rosenberger, File type identification tools for digital investigations, Forensic Science International: Digital Investigation, Volume 46 (2023) 2666-2817
   DOI: https://doi.org/10.1016/j.fsidi.2023.301574

2. Joachim Sester, Darren Hayes, Mark Scanlon, Nhien-An Le-Khac, A comparative study of support vector machine and neural networks for file type identification using n-gram analysis (2021)
   DOI: https://doi.org/10.1016/j.fsidi.2021.301121

3. Cezar Serhal and Nhien-An Le-Khac, Machine learning based approach to analyze file meta data for smart phone file triage (2021)
   DOI: https://doi.org/10.1016/j.fsidi.2021.301194

4. Xiaodong Lin, File Signature Searching Forensics, Introductory Computer Forensics (2018)
   DOI: https://doi.org/10.1007/978-3-030-00581-8_10

5. Myeong Lim and James Jones, A Digital Media Similarity Measure for Triage of Digital Forensic Evidence, Advances in Digital Forensic (2020)
DOI: https://doi.org/10.1007/978-3-030-56223-6_7

6. Vitor Hugo Galhardo Moia and Marco Aurélio A. Henriques, A comparative analysis about similarity search strategies for digital forensics investigations, Proceedings of the Thirty-Fifth Brazilian Symposium on Telecommunications and Signal Processing (2017)
DOI: http://dx.doi.org/10.14209/sbrt.2017.115
URL: https://www.sbrt.org.br/sbrt2017/anais/1570361754.pdf

7. Jonathan Oliver, Chun Cheng and Yanggui Chen Trend Micro, TLSH - A Locality Sensitive Hash (2013)
DOI: https://doi.org/10.1109/CTC.2013.9

8. Jeremy Davis, Joe MacLean, David Dampier, Methods of Information Hiding and Detection in File Systems, Department of Computer Science and Engineering, Mississippi State University (2010)
DOI: https://doi.org/10.1109/SADFE.2010.17

9. Goldberg, Raymond M., Testing the forensic interestingness of image files based on size and type (2017)
URL: https://upload.wikimedia.org/wikipedia/commons/0/0a/Testing_the_forensic_interestingness_of_image_files_based_on_size_and_type_(IA_testingforensici1094556129).pdf

10. Rowe, N.C., Identifying Forensically Uninteresting Files Using a Large Corpus (2014)
DOI: https://doi.org/10.1007/978-3-319-14289-0_7

11. Anuradha Gupta, Privacy Preserving Efficient Digital Forensic Investigation Framework (2013)
DOI: https://doi.org/10.1109/IC3.2013.6612225

12. Xiaoyu Du, Alleviating the Digital Forensic Backlog: A Methodology for Automated Digital Evidence Processing (2020)
URL: https://www.markscanlon.co/papers/PhDThesis-MethodologyAutomatedDigitalEvidenceProcessing.pdf

13. Gyu-Sang Cho, A computer forensic method for detecting timestamp forgery in NTFS (2013)
DOI: https://doi.org/10.1016/j.cose.2012.11.003