Disk Encryption Analyzer

A tool for the idenfication of encrypted containers in a live system or an evidence file.

Software Project – CSE2000

Delft University of Technology

Authors: Group 16-D
Kaan Altinay, Mihnea Bernevig, Yigit Colakoglu,
Matej Tomasek, Konstantin-Asen Yordanov
Client: Rotterdam Police
TU Coach: Thomas Durieux, TA: Ruben Backx
Project Coordinators: Martin Skrodzki, Thomas Overklift, Otto Visser, Huijuan Wang

June 2023

Preface

This report was written by a group of five BSc Computer Science students at the Delft University of Technology. In the last quarter of our second year, we applied for a project offered by Team Bestrijding Kinderporno en Kindersekstoerisme of the Rotterdam Police involving identifying encrypted containers in evidence files and live systems.

While writing this report, we assumed the reader has a basic understanding of software development and is knowledgeable of digital forensics and Windows-supported file systems on disk drives. A list of abbreviations was added for further clarity of the terms used, together with footnotes for external references and more detailed explanations of concepts.

Readers with experience in software development who will maintain the tool in the future would find Chapter 3 and Chapter 4 the most relevant. For readers interested in the problem's scope and its ethical implications, such information can be found in Chapter 2 and Chapter 6.

We would like to thank our clients Jan-Willem van Lottum, Mark van Ochten, and Leo Kerkhof for their continued support and oversight, as well as our coach Thomas Durieux for his supervision and advice for Python development. We are also very thankful to the hardworking authors of Digital Forensics Virtual File System, a Python package widely used in our application, and its back-ends.

Delft, 25th of June 2023
Kaan, Mihnea, Yigit, Matej, and Konstantin-Asen

Summary

In the field of forensic investigation, identifying artifacts of encryption software is a common task carried out by analysts whose main objective is to find evidence of criminal activity. While encryption may be used solely for security purposes with no ill intentions, it also presents an opportunity to hide such incriminating files for people involved with blackmail, data theft, or organized crime. In the scope of this project, images and footage of child pornography are often similarly hidden in encrypted containers, in many cases being stockpiled by hoarders and distributors on their personal devices.

Given the sensitive nature of the data, investigators need to reliably detect these volumes and quickly find the perpetrators responsible. However, analyzing an entire drive or disk image is not always computationally feasible. The main problems arise from the variety of commercially-available cryptographic software as well as the typically large size of the drives under-inspection. Most tools used by forensic analysts can in practice only handle certain file types, and they fail to efficiently traverse the targeted filesystem, which costs valuable time during investigation and produces inconclusive results.

The main objective of this report is to elaborate on our group's approach and design choices throughout the development of the Disk Encryption Analyzer (**DEA**) tool. Team Bestrijding Kinderporno en Kindersekstoerisme (TBKK) of the Rotterdam Police requested a maintainable software application that offers high performance for live inspections at a suspect's home and provides greater accuracy when run on evidence files at the police station. The tool's codebase should also be structured in an extensible way, allowing developers to easily add support for new analysis methods and other file formats in the future.

To improve the accuracy of flagging suspicious containers, we implemented three analysis methods checking file headers and extensions (signature analysis), common Windows directories and registry entries (artifacts analysis), as well as the randomness of a file's raw byte-data (statistical analysis). They interleave in a way that continuously reduces the search space, adhering to the performance requirement for the tool. We provided support for filesystems recognized by the Windows operating system and incorporated multiprocessing to parallelize the detection of encryption-related artifacts. To test our functionality, we performed unit testing, mocking, and system testing with virtual machines, achieving line coverage of over 75%. We also wrote code documentation for individual classes and methods, as well as an installation, a developer, and a user guide.

In conclusion, **DEA** offers an efficient solution for detecting encrypted containers in both live systems and forensic evidence files. The extensible class structure would allow future developers of the tool to add more analysis techniques that reduce the number of flagged false positives. Support for the macOS and Linux operating systems could also be added, so that **DEA** is applicable in even more circumstances. Additionally, translating the tool's documentation as well as the error-handling messages into Dutch would make **DEA** available to investigators who do not speak English, further improving transparency and explainability.

List of Abbreviations

DEA	Disk Encryption Analyzer
TUD	Delft University of Technology
TBKK	Team Bestrijding Kinderporno en Kindersekstoerisme
IDE	Integrated Development Environment
CLI	Command Line Interface
GUI	Graphical User Interface
MBR	Master Boot Record
VHD	Virtual Hard Disk
DTO	Data Transfer Object
GDPR	General Data Protection Regulation
LED	Law Enforcement Directive
NSRL	National Software Reference Library

Contents

1 Introduction

Detecting encrypted containers in physical and virtual drives remains a difficult task in the field of forensic investigation. One challenge comes from the variety of encryption software that is currently available, like BitLocker[1] and VeraCrypt[2]. The large number of disk-image formats exacerbates this issue as tools used by forensic analysts can in practice only specialize in a set amount of file types (common ones include E01, L01, DD, and AD1). Consequently, they generalize poorly to other forms of encryption and fail to offer a universal solution. Another problem arises from the size of a given drive or disk image. Because it can vary from 50 GB to more than 14 TB, the complete analysis of a targeted system is often computationally infeasible. Such large search spaces cannot be traversed efficiently during inspection [1].

Both problems are important to consider since forensic investigators need to reliably detect and process encrypted files regardless of what software produced them. People engaging in immoral or criminal behavior often hide the evidence in such volumes, precisely to avoid police intervention. Consequently, forensic analysts regularly perform on-site inspections at a suspect's home – in such scenarios, the speed of analysis becomes the greatest priority, which in turn relates to the problem of computational infeasibility. For these reasons, Team Bestrijding Kinderporno en Kindersekstoerisme (TBKK) and Team Zeden of the Rotterdam Police asked us to develop a software tool that identifies encrypted containers in both live systems and already-collected evidence files, offering high performance during analysis yet also remaining maintainable as the technological stack changes in the future.

The objective of this report is to provide justification for the design choices our group made while developing the Disk Encryption Analyzer (**DEA**) tool, our proposed solution that identifies encrypted containers in a system. A literature study was performed in connection with detecting the outputs of cryptographic algorithms and the installations of commercially-available encryption tools. In order to extract the requirements for the product's operation, interviews were conducted with the clients – two of Team TBKK's forensic investigators. The desired functionality was then prioritized according to the MoSCoW model, placing a larger emphasis on the critical features with reference to the time available for development. Throughout the design process, we strove to make **DEA** both extensible and efficient. To achieve the former, we used abstractions for disks, filesystems, and analysis techniques, providing a universal standard for each of the tool's components. In order to satisfy the speed requirements without neglecting the tool's accuracy, we implemented two operation modes (live and complete), leaving the choice of which version to run to the operator conducting the investigation – either in the field or at the police station.

[1]Overview of Windows BitLocker: `https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/`

[2]VeraCrypt: `https://veracrypt.fr/en/Home.html`

The report will be presented in the following structure. Chapter 2 will provide background information and outline the stakeholders involved. Chapter 3 will demonstrate the team's major architectural and design decisions, as well as the different analysis methods that were implemented. Chapter 4 will then elaborate on the concrete implementation of **DEA** and discuss the critical parts of the application. Afterwards, Chapter 5 will outline our group's development methodology, the code quality and testing frameworks we used, as well as the list of requirements for the tool's operation, prioritizing some as necessary for the minimum-viable product over any extra functionality. Finally, Chapter 6 will address some of the ethical implications of **DEA**'s future use in forensic investigation, followed by our team's conclusions and recommendations for the tool's maintenance.

2 Establishing the Task's Scope

Elaborating on the intricacies of the issue at hand, this chapter focuses on a comprehensive evaluation of the identified problem, the stakeholders involved, the applicable use cases. It provides an overview of the currently-available products and the expertise incorporated throughout the development process. With this chapter of the report, we aim to deliver a robust understanding of the problem, crucial for grasping the rationale behind our design choices.

Section 2.1 discusses the difficulties forensic investigators face when searching for cryptographic output, further justifying the need for **DEA**. Afterwards, Section 2.2 outlines the relevant stakeholders, and Section 2.3 elaborates on the potential use cases for the product our group developed. Section 2.4 then provides information on already-existing tools for encryption detection, their benefits and drawbacks. Lastly, Section 2.5 lists the areas in which the two experts we consulted, Jan-Willem van Lottum and Mark van Ochten, assisted us throughout the project.

2.1 The Need for Encryption Identification

Detecting encryption presents a significant challenge. Suspects often conceal illegal files in encrypted containers, making detection and access difficult for investigators. While we can access these containers through various means, these containers are usually hidden deep within the filesystem. This makes it more costly and harder to detect their presence and location, leading to investigators missing valuable evidence.

One difficulty in identifying encrypted containers arises from the wide variety of cryptographic software and disk-image formats which are prevalent today. Forensic analysts' tools are specialized, which limits their ability to recognize and support all encryption methods and file types.

Another problem are the immense (and often infeasible) time requirements of analyzing drives of **large sizes** (usually, Terabytes-worth of data), which are particularly difficult to meet during on-site inspections, where the time window is short, and speed is paramount. Therefore, as encryption of illegal files becomes more prevalent in crime, a tool that could perform encryption detection in a fast and thorough manner is crucial.

To mitigate these challenges, forensic investigation teams Zeden and TBKK seek a high-performance software tool. This product should be competent in recognizing encrypted containers within live systems and pre-collected evidence files, and also offer an adaptable structure, facilitating the smooth integration of new elements.

2.2 Stakeholders Involved

This product's success hinges on addressing the needs and expectations of various stakeholders, each bringing a unique perspective to the functionality, usability, and effectiveness of the proposed tool. From forensic analysts striving for accuracy to suspects concerned about the integrity of their devices, the stakeholders involved and their roles and requirements are outlined as follows:

Forensic Analyst The forensic analyst, working with evidence files in an office environment, should be able to use the tool to analyze such files and check whether they contain encrypted containers. For this stakeholder, accuracy and thoroughness are more important than speed.

Field Officer The field officer would usually rely on the tool to quickly analyze a running machine for potentially encrypted files, as well as any suspicious artifacts which would indicate whether an encryption tool was used in the system under-inspection.

Developer The developer is going to maintain the tool we create and add new evidence file types, artifacts, and filesystems as the scope of the tool widens due to the needs of the customer. The process of adding new functionality, especially support for new evidence files and filesystems, should therefore require minimal interaction with the previously written code, adhering to the **open-closed**[3] principle for extensibility.

Decision-Maker The tool's output is going to be used by decision-makers in law enforcement, and possibly in judicial environments. This means that it should be concise and readable to be processed in a quick manner by this stakeholder. Moreover, the tool should make it clear why it has deemed an artifact as suspicious to allow this stakeholder to justify their decision.

Suspect The main priority for the suspect is that their device is not damaged and their stored files remain untouched in the event that they are innocent. To that end, the analysis tool should not alter the files it interacts with while running. The findings report generated by the tool ought to contain the paths to the likely-encrypted together with the benign files from the system under-inspection. Flagged files will then be provided as input to cracking software used by forensic analysts, their contents being revealed. While decryption is not part of the **DEA** pipeline, the presence of both benign and suspicious file paths in the tool's output would provide more justification for the decision-making that ultimately affects the suspect.

2.3 Use Cases

The analysis performed by the **DEA** tool has two major applications that would benefit Team TBKK and Team Zeden during forensic investigation:

- A suspect's system needs to be inspected on-the-spot during a live police operation: the drive(s) are scanned for the presence or traces of encryption tools (artifact analysis), and the files are divided into likely encrypted vs.

[3]The Open-Closed Principle states that software entities (classes, modules, functions) should be open for extension but closed for modification, allowing for new behavior to be added without altering existing code.

benign based on the header information and file types (signature analysis). The tool is run on the target system from an **external** thumb drive. The finding report is created and stored on the thumb drive, in order to avoid leaving unnecessary traces on the suspect's machine.

- Collected evidence files now need to be analyzed at the police station using the larger amount of computational resources available there: in addition to the artifact and signature analysis, statistical checks are carried out to show the presence of pseudorandom[4] data (highly indicative of encryption) and achieve a greater detection accuracy with fewer false positives.

2.4 Survey of Existing Tools

There are several tools in the industry that are able to detect encrypted volumes, files, and search for the artifacts of an encryption tool in a filesystem. Analyzing these existing tools in the search for an already-applicable solution to this issue gave us some relevant insight into the functionality our tool should provide.

2.4.1 MAGNET Encrypted Disk Detector (EDD)

The tool's official description from the MAGNET forensics website [2] states:

> Encrypted Disk Detector checks the local physical drives on a system for TrueCrypt, PGP, VeraCrypt, Check Point processes, SafeBoot, or Bitlocker encrypted volumes. If no disk encryption signatures are found in the MBR (Master Boot Record), EDD also displays the OEM ID and, where applicable, the Volume Label for partitions on that drive, checking for Bitlocker volumes.

The primary purpose of the MAGNET forensics tool is to assist analysts in identifying encrypted disks on a live system, providing valuable information to support decision-making processes. However, it is important to note that EDD does not detect encrypted partitions within evidence files, nor does it scan the filesystem for encrypted containers or encryption artifacts.

2.4.2 Autopsy Encryption Detection Module

This[5] commonly-used forensics analysis platform comes bundled with a module for detecting encrypted containers in evidence files. It mainly relies on statistical analysis methods like entropy calculation, and filters based on file sizes. It lacks support for analyzing a whole disk dump for encrypted partitions and detecting artifacts of encryption tools in a system.

[4]A sequence that appears to be random, but was generated by a deterministic process.

[5]For more information, `https://sleuthkit.org/autopsy/docs/user-docs/4.19.3/encryption_page.html`

2.4.3 Survey Conclusion – Insight Gained

After analyzing the available tools, we have concluded there are various solutions to detecting encrypted containers on live systems and evidence files, with the caveat, however, that none of them provide a comprehensive package that can perform the complete analysis our client is looking for. Although the tools we have discovered are not enough to outright solve the problem we were presented with, some valuable insight was gained from the analysis. The user interface of the Autopsy Encryption Detection Module, both when configuring the settings of the application and when viewing the output, provided us with inspiration for the design of our own tool's GUI. As for MAGNET forensics' solution, the non-intrusiveness and ease-of-use of the tool gave us a standard to strive for.

2.5 Consulted Experts

The product owners we are developing the **DEA** tool for are Mark van Ochten and Jan-Willem van Lottum, both of whom are forensic investigators for Team TBKK of the Rotterdam Police. We discussed the functionality requirements with them early on in the project timeline, and they remained the main point of contact regarding any modifications to the product throughout the quarter. We also consistently held weekly Monday meetings to report our team's development progress and ask for clarifications as needed.

They also provided us with forensic tools used in-practice (FTK[6], EnCase[7], and TrueCrypt[8]) and demonstrated how they operate, allowing us to create and inspect our own encrypted containers for testing purposes. In this manner, we also had the opportunity to direct any questions related to these tools and the way they handle evidence files to our product owners.

For technical questions regarding the files our tool processes, the future use of the tool after deployment, and the most suitable formats of the findings report that should be generated, we regularly contacted Jan-Willem and Mark. Since they are experts in the field of forensic analysis, they immensely facilitated the development process by providing relevant and swift answers.

[6]FTK Imager: `https://www.exterro.com/ftk-imager`
[7]OpenText EnCase Forensics: `https://www.opentext.com/products/encase-forensic`
[8]Discontinued, superseded by VeraCrypt

3 Project Approach

This chapter of the report expands upon our team's design choices, providing a discussion of our reasoning and a high-level overview of the product's internal structure. Section 3.1 presents an analysis of the architectures we considered for the tool and then describes how the pipe-and-filter architecture we ultimately chose. Section 3.2 explains supporting design patterns through the use of UML diagrams. Finally, Section 3.3 demonstrates the analysis workflow, describing the functioning of the tool step-by-step.

3.1 Settling on the Pipe-and-Filter Architecture

In preparation for the development process, we carefully assess the different software architecture options. A right choice of an architectural pattern would greatly simplify our development process since it provides a general solution to commonly-occurring problems, which have relevance in our case. Additionally, it serves as a set of guidelines we adhere to when extending our code structure. The most important observations about architectural patterns are the trade-offs involved with their implementation. Choosing a non-fitting one would lead to wasted effort and lost time spent on an inefficient code base. We have considered the following architectural patterns:

Layered Architecture was our first choice of architecture due to the manner in which the items the tool should be able to analyze stack up into a layered hierarchy. If we choose this architecture, then the disks would be on the bottom layer, files on the top, with file systems staying in-between those two. Each layer would contain methods to analyze the artifacts that belong to that layer, and the resulting artifacts would be passed on only to the layer that is one above the current one. This would provide us with a convenient categorization of our artifacts. Because of this, the layered architecture was the one that we initially considered using. However, it proved to be too constricting of how the analysis methods interact with each other and how artifacts are passed between the internal components.

Monolithic Architecture was considered as the application would become a self-contained unit where its components rely on coupling and complement their functionality. For a project of our scope, opting for this pattern was partially fitting, considering how the analyzable items interact with the different analysis techniques. However, it provides no solution for streamlining the workflow in an efficient and extensible manner that at the same time remains explainable and is unlikely to become obfuscated.

Pipe-and-Filter Architecture being a viable choice was not immediately obvious to us. After we established the components' structure, we looked into the speed requirement of our application. Maintaining a queue (or pipe) of items on which each analyzer acts as a filter, possibly generating more items and adding them to the queue, seemed the most viable to us. Using multiple processes, we could then further boost the performance.

Following our analysis of potential architectures, we came to the conclusion that the pipe-and-filter architecture is the most appropriate for our application. First, it complements the modularity requirement we had from the beginning of the project. Adhering to this architecture, we could achieve a structure where the artifacts generated would simply be piped into their corresponding analysis methods, and the work done by analyzers could be decreased by adding filter components. Moreover, different components could be chained together without having to modify the existing ones, allowing future developers to add features while adhering to the open-closed principle.

3.2 Supporting Design Patterns

Due to the extensibility requirements of the application, the selection of design patterns played a crucial role in the tool's development. Furthermore, because of the limited time span allocated for development (6–7 weeks), choosing design patterns that would meet the modularity requirements of the tool was of high importance since starting off with a wrong set of design patterns in-mind would have cost-valuable development time. Therefore, the final product would ideally facilitate adding new functionality with **minimal modifications** to the existing codebase and maintain a class structure with **low cohesion**, separating classes whose behavior is unrelated into different modules, exposing only some methods to other parts of the application, and keeping the rest private.

3.2.1 Strategy

Given the modularity requirements of the **DEA** tool, it was crucial to design it in a manner that facilitates adding support for other evidence file types and new file systems. Even though the tool will initially operate only on **live systems** and **evidence files**, adding other operation media should also be easy to make the tool future-proof. Therefore, obeying the open-closed principle while developing the tool was of utmost importance, both for extensibility and code reuse.

Another part of the tool that requires a high level of extensibility is the analysis techniques that are being applied. Therefore, one of our main priorities was to give the possibility of adding new analysis methods in the future as further research into cryptography is conducted, new encryption algorithms are discovered, or other encrypted containers become widely used.

Using a *strategy* design pattern[9] for the parts of the **DEA** tool that requires high modularity (i.e. should remain loosely coupled, interacting only through an interface of exposed functionality) goes hand-in-hand with our need to achieve

[9]The strategy design pattern allows for interchangeable algorithms to be encapsulated and used dynamically within an object, enabling flexibility and runtime selection of different behaviors.

flexibility in the data types being worked with. By abstracting away the lower-level details of accessing an evidence file and reading a filesystem, the **strategy** pattern allowed us to focus on the analysis instead. A simplified UML diagram describing how this pattern was implemented can be found in *Figure 1*.

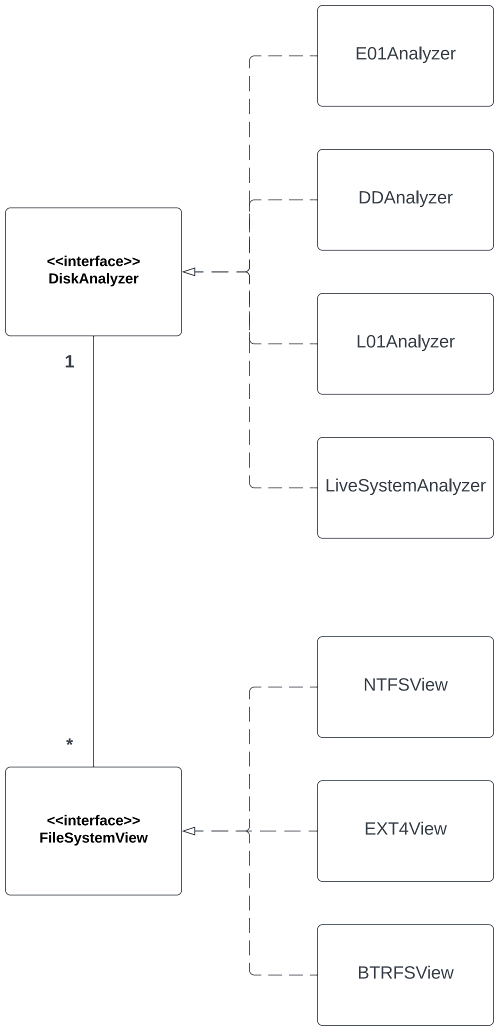


Figure 1: UML Diagram of the DiskAnalyzer and FileSystemView Strategies

The strategy pattern was also applicable when we implemented the analysis techniques for flagged file paths. Despite their differing internal operation, these methods can be categorized based on **the common type of input** they expect and **the identical result-objects** they produce. Taking advantage of this, the strategy pattern allows the core system to run different analysis techniques to

abstract away from implementation details. As long as the standardized class and method signatures for the analysis modules are maintained, users will only need to provide the correct input type to obtain a result. This design facilitates extensibility as a developer can add another analysis method by simply creating a new strategy class without having to interact with any other part of the tool. An example of how this was implemented is shown in *Figure 2*.

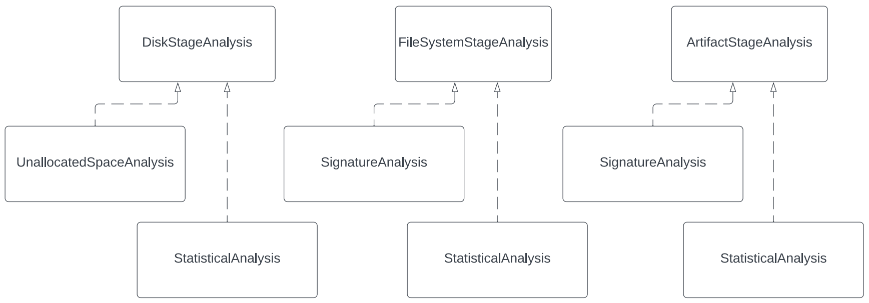


Figure 2: Class Diagram of the Analysis Strategies per Stage

3.2.2 Data Transfer Object (DTO)

Because of the fact that all analysis methods have similar outputs and all of the aggregated results will eventually be serialized into a file, it is helpful to use a common DTO for all analysis strategies to use as their output. This final DTO would consist of all the relevant information that should be reported to the user once the analysis has ended. These data would be a list of containers flagged as encrypted that should be passed through cracking software, a list of benign files that can undergo basic contents analysis, and optionally an instance of a *Chart* object that stores information on how to visualize the DTO when serializing to a visual format like HTML, so that the graph is included in the findings report.

3.2.3 Factory Method

Because we support several reporting techniques in the tool, we needed a way to serialize the output DTOs into another format. Moreover, with customizability being another major focus of the tool, it is necessary to allow the user to modify the data which is serialized. This, of course, necessitates that the user is able to modify the serializer instance that the tool uses – in such a scenario, a *factory method*[10] design pattern is applicable. Having a serializer factory be maintained for each output-format, a serializer of the corresponding type could quickly be created whenever a findings report is to be generated, regardless of the concrete analysis stage. Several formats are supported for the findings report generated

[10]The factory method design pattern is a creational pattern that provides an interface for creating objects, but allows subclasses to decide which class to instantiate.

by the tool, including JSON, HTML, TSV, and TXT. A UML diagram outlining the factory method pattern is shown in *Figure 3*.

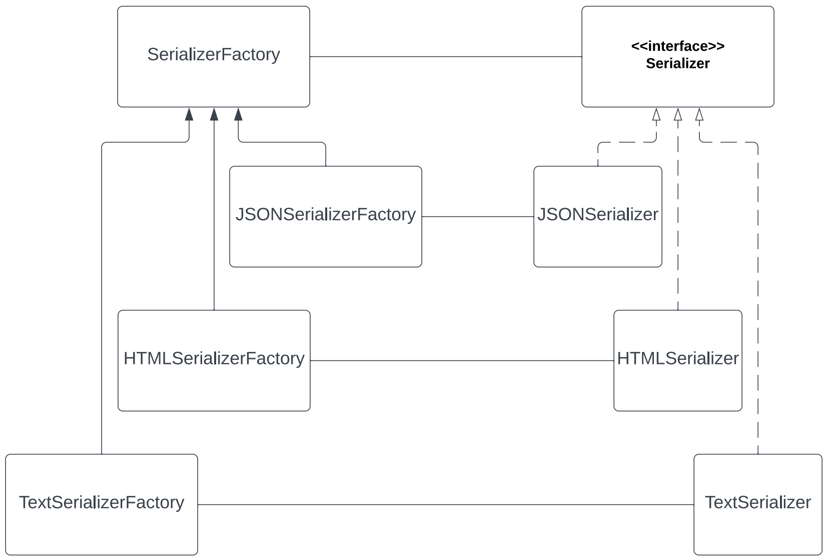


Figure 3: UML Diagram Representing the FindingsReport Factory

3.3 Outlining the Analysis Workflow

The analysis performed by the **DEA** tool constitutes a sequence of checks, each narrowing down the search space occupied by encrypted containers that need to be inspected while also improving the accuracy of the flagging carried out by the tool. Hence, checks that are costly in terms of their time requirements (like the statistical ones) will be run on a **maximally-reduced** search space for the purpose of reducing the number of false positives in the tool's output.

First, the tool reads the partition table of the provided forensic image in order to view the partitions' locations and sizes, as well as to check whether there is disk space that is unpartitioned and should therefore be inspected further. Having access to the partitions and their corresponding filesystems, the tool then performs **signature** analysis, checking for mismatches between a file's header and its explicit extension. During this step, files smaller than 50 Megabytes can be omitted and immediately labeled as benign since the output of any available encryption tool would be too large to fit within that size.

In parallel to that, the tool searches the filesystem under-inspection for file types and keywords usually indicative of encryption tools (such as BitLocker, TrueCrypt, and VeraCrypt). It also parses the registry files and event logs that it encounters for any references to the former and traces of their use, typically in the form of root keys or file names (**artifact** analysis) [3].

The most accurate yet time-consuming step are the **statistical** checks that allow for the detection of pseudorandom data – another indicator of encryption. For our tool, we implemented four such algorithms (Entropy, Chi-Square Test, Frequency Test, Monte Carlo for Pi), all of which provide an expected threshold for encrypted data the tool's results can be compared against [4].

The generated **findings report** contains a list of benign files and a collection of likely-encrypted containers flagged by the **signature** and **statistical** analysis steps. It also shows whether flags were raised for the presence of unpartitioned space, encryption tools, or virtual machines from **artifact** analysis.

4 Implementation Details

Expanding on the architectural choices from the previous chapter, this chapter of the report elaborates on the concrete implementation of our product and the way we decided to structure its internals. The different sections explore certain algorithms and techniques we have used that play a central role in **DEA**'s operation. First, Section 4.1 expands upon the user interface for the application. Afterwards, Section 4.2 discusses the functionality of the engine, the benefits of multiprocessing, and the use of lazy-loading. Lastly, Section 4.3 demonstrates the analyzer graph, outlining the interactions between the analysis techniques.

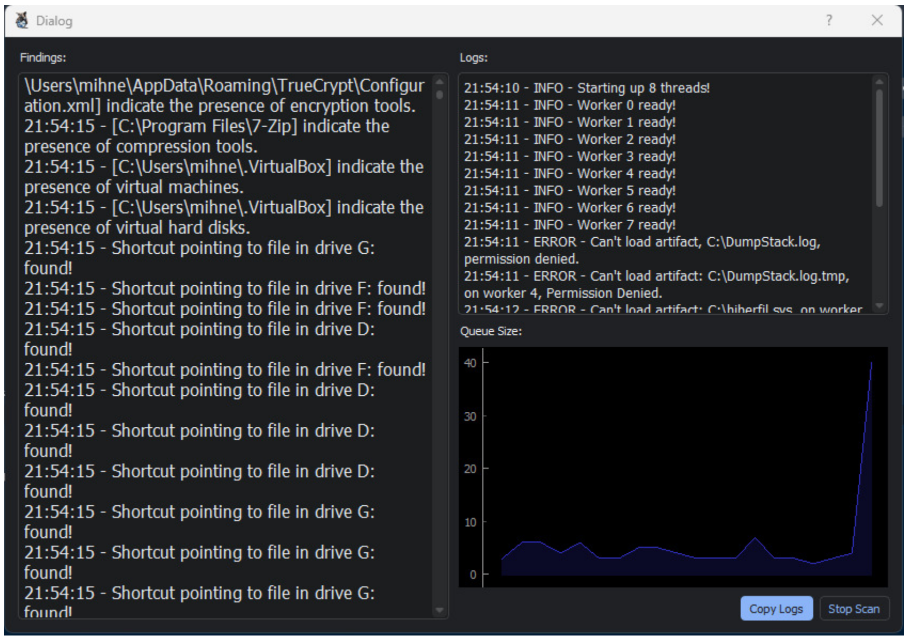
4.1 Design of the User Interface

The user interface has three main features: extensibility, portability and variety. It allows the addition of new parameters to analysis techniques without having to edit the existing code. Moreover, it provides the ability to transport a certain configuration so that search results can be reproduced. Finally, it supports usage through multiple media, namely a command line interface (CLI) and a graphical user interface (GUI).

First, in order to allow the addition of analysis techniques that require arguments, we implemented a custom method where techniques specify what arguments they need. This information is then used by the tool to configure the user interface dynamically so that the user can customize the internal state of any analysis method.

The requirements analysis period has shown us that the GUI and the CLI are both equally valuable to the client when they are using **DEA**. The GUI is crucial when operating in the field while the CLI is very important for the integration of the tool to the analysis pipeline of the client. Because of this, we have integrated both features into the tool and made it so that the CLI and the GUI are each other's complement. Thus, any operation that can be executed on the GUI can also be done by running an equivalent command in the terminal.

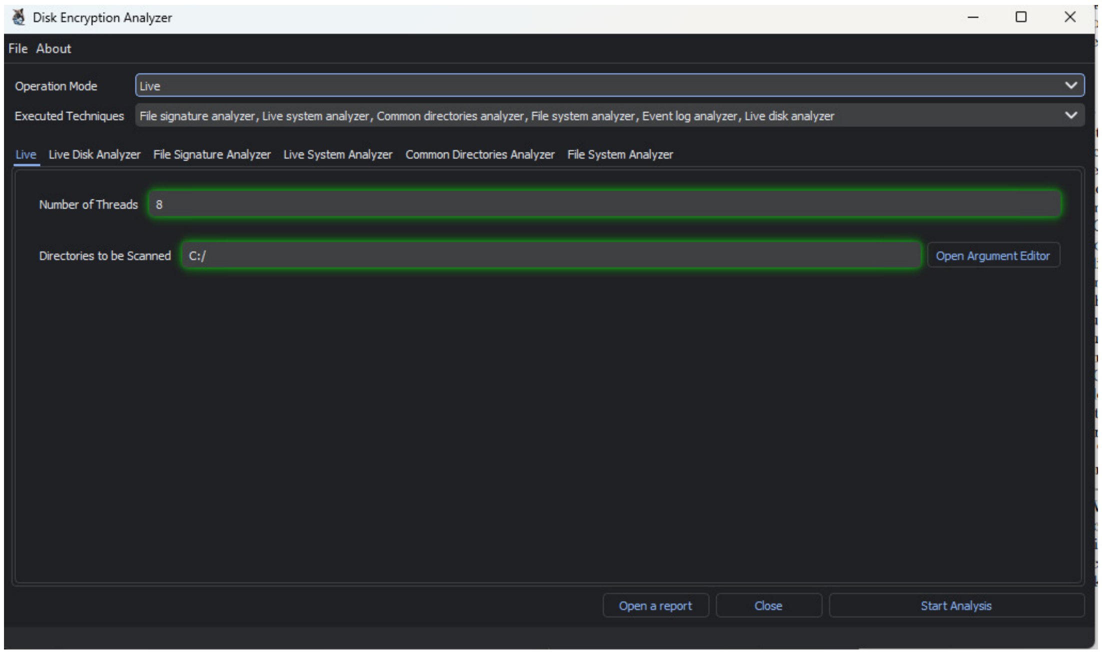
Finally, because of the portability requirement, the tool had to provide the option to import and export a scan's configuration so that it can be reused in the field or in the forensics lab. Because of this, we have integrated the ability to pass a configuration file to the tool when running it, which is parsed by the same central argument parser, and thus allows users to export and import any possible configuration that **DEA** supports.



```
\Users\mihne\AppData\Roaming\TrueCrypt\Configur
ation.xml] indicate the presence of encryption tools.
21:54:15 - [C:\Program Files\7-Zip] indicate the
presence of compression tools.
21:54:15 - [C:\Users\mihne\.VirtualBox] indicate the
presence of virtual machines.
21:54:15 - [C:\Users\mihne\.VirtualBox] indicate the
presence of virtual hard disks.
21:54:15 - Shortcut pointing to file in drive G:
found!
21:54:15 - Shortcut pointing to file in drive F: found!
21:54:15 - Shortcut pointing to file in drive F: found!
21:54:15 - Shortcut pointing to file in drive D:
found!
21:54:15 - Shortcut pointing to file in drive F: found!
21:54:15 - Shortcut pointing to file in drive D:
found!
21:54:15 - Shortcut pointing to file in drive D:
found!
21:54:15 - Shortcut pointing to file in drive D:
found!
21:54:15 - Shortcut pointing to file in drive G:
found!
21:54:15 - Shortcut pointing to file in drive G:
found!
21:54:15 - Shortcut pointing to file in drive G:
found!
```

```
21:54:10 - INFO - Starting up 8 threads!
21:54:11 - INFO - Worker 0 ready!
21:54:11 - INFO - Worker 1 ready!
21:54:11 - INFO - Worker 2 ready!
21:54:11 - INFO - Worker 3 ready!
21:54:11 - INFO - Worker 4 ready!
21:54:11 - INFO - Worker 5 ready!
21:54:11 - INFO - Worker 6 ready!
21:54:11 - INFO - Worker 7 ready!
21:54:11 - ERROR - Can't load artifact, C:\DumpStack.log,
permission denied.
21:54:11 - ERROR - Can't load artifact: C:\DumpStack.log.tmp,
on worker 4, Permission Denied.
21:54:12 - ERROR - Can't load artifact: C:\hiberfil.sys, on worker
```

Figure 4: Screenshot from the analysis GUI



Operation Mode: Live

Executed Techniques: File signature analyzer, Live system analyzer, Common directories analyzer, File system analyzer, Event log analyzer, Live disk analyzer

Number of Threads: 8

Directories to be Scanned: C:/

Figure 5: Screenshot from the analyzer settings GUI

4.2 Engine

The engine is a component of **DEA** that is designated to handle the management of the analysis process, keep track of the artifacts that have been queued up for analysis, and spawn the worker processes. To implement all this, it makes use of several techniques, namely multithreading, multiprocessing, and lazy-loading.

4.2.1 Multiprocessing and Multithreading

The engine makes use of multiprocessing to spawn up to n processes, specified by the user when it is first initialized. It spawns subprocesses instead of threads in order to ensure that the concurrency is not impacted by the Python GIL. The GIL is the Python Global Interpreter Lock, it is a lock within the Python interpreter used to ensure that only one thread is able to run using the interpreter instance. When creating the subprocesses, it provides them with a latch, a counter, and a queue which it uses to manage the processes.

When a worker is started up, it enters the cycle of polling the queue for artifacts and analyzing them. Once it retrieves an artifact, it increments the shared worker counter and analyzes the artifact. Once it is done analyzing, it decrements the counter and iterates over the loop one more time. It keeps looping until one of the following three conditions are met: the latch has been set, the queue is empty and there are no other workers running analyzing an artifact, or an error has been encountered. The pseudocode for the worker's analysis loop is provided in Algorithm 1:

Algorithm 1 Pseudocode for a worker process' main loop

```
1: $latch \leftarrow False$
2: $worker\_count \leftarrow 0$
3: while $\neg latch$ do
4:     $artifact \leftarrow queue.pop()$
5:     if $artifact = None$ then
6:         break
7:     end if
8:     $worker\_count \leftarrow worker\_count + 1$
9:     $load(artifact)$
10:    $analyze(artifact)$
11:    $worker\_count \leftarrow worker\_count - 1$
12:    break
13:    $worker\_count \leftarrow worker\_count - 1$
14:    if $worker\_count = 0 \wedge queue.isEmpty()$ then
15:        break
16:    end if
17: end while
18: $queue.push(None)$
19: $latch \leftarrow True$
```

The engine also starts up 3 watchdog threads, which are run under the same main engine thread and share its GIL. These threads are responsible for handling I/O operations such as waiting for a graceful exit input if we are running in the CLI, saving the report every n minutes to ensure minimal data-loss in case of failure, and finally notifying the user of the current status of the queue so that the operator can get a general idea on the tool's process.

4.2.2 Lazy-Loading

When the engine is started up, it initializes the data structures necessary for **DEA**'s operation and initializes proxies for each of them, which are then handed over to every worker process that is spawned. These proxies allow GILs running in different processes to communicate with the main process where the engine is running. When communicating with these proxies, the workers need to serialize/deserialize objects using Pickle[11].

The use of proxies for communication comes with the benefit of achieving true multiprocessing, which is very important for the performance of the tool. However, it introduces the complication that not every object is serializable using Pickle. This is especially valid in **DEA**'s case, as it interacts with file objects. Because of this, we have chosen to make every artifact object that **DEA** analyzes lazily-loaded. This allows every artifact to initially be treated simply as metadata on what needs to be analyzed. Once they are loaded, artifacts allow access to the actual data they contain.

The lazy-loading of artifacts also comes with the extra benefit of caching. If the objects were not lazily loaded, the only option would be to implement a central cache data structure in the main engine thread, but this cache's benefits would be severely hindered by the inter-process communication required. By lazily loading the objects in workers, however, the workers are able to maintain their own local cache which they use when loading objects. This is especially beneficial when analyzing large evidence files since they require caching to be analyzed efficiently.

4.3 Analyzer Graph

The entire analysis process heavily relies on all sequences, or paths, of analyzers that the disk under inspection takes. Passing through the chain, the disk is split by the analyzers into smaller analyzable artifacts, which are then processed by their corresponding analyzers along the path. This arrangement of consecutive analyzers creates a directed graph, where the nodes are the analyzers. Such a graph is shown in *Figure 6.*

[11]Pickle is a standard python library which is used to serialize and deserialize objects. https://docs.python.org/3/library/pickle.html

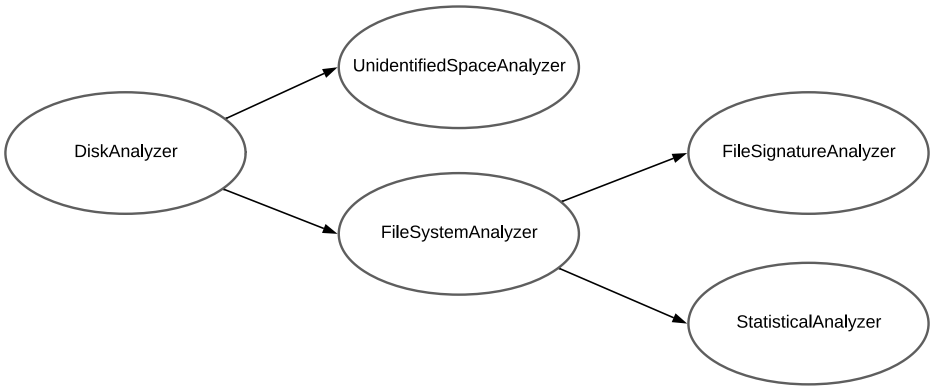


Figure 6: A graph of analyzers

To enforce such an ordering, every analyzer has to provide its input type and all possible output types of artifacts, such that it is possible for us to match different analyzers to each other. When an analyzer has more than one output type, it is called a generator. Generators are usually at the beginning of each path, enqueueing smaller analyzables in the engine, such as file systems or files. They are also prioritized by the engine to ensure that analyzables are generated first.

5 Development Methodology

This section offers a detailed overview of the project's development life cycle. Beginning with "Our Approach to Development" (5.1), we expand upon the use of the Scrum framework, focusing on its impact on team communication, task management, and client feedback incorporation. "Code Quality, Testing, and Style" (5.2) explores our strategies for maintaining high-quality code, from our dual-pronged testing approach to the use of linting tools and Python's type hints, while "Reflection on our Testing Strategy" explains the changes we have made to our initial testing approach, throughout the project. The importance of clear and accessible documentation is emphasized in "Documentation" (5.3). Lastly, "Requirements" (5.6) lays out our non-functional and functional requirements, explaining our prioritization process via the MoSCoW model.

5.1 Our Approach to Development

The development methodology we adopted for this project was based on the Agile approach, specifically employing the Scrum framework. This provided us with the flexibility and adaptability necessary to respond effectively to changes and feedback throughout the development process.

Our choice to utilize the Scrum framework was influenced by several factors. Firstly, we drew on our past experiences with Scrum from previous projects, which allowed us to take full advantage of the framework's strengths and avoid potential pitfalls. Secondly, we valued Scrum's iterative, incremental approach to development, which aligned well with our project goals and the nature of our team.

Given our small team size, frequent face-to-face interactions — a cornerstone of Scrum — were practical and beneficial. Additionally, the Scrum methodology facilitated a continuous feedback loop between the client and the development team. End-of-sprint meetings served as opportunities to showcase our progress to the client, obtain feedback on implemented features, and discuss areas that required further development. This process greatly enhanced transparency, ensured mutual understanding of the project's status, and helped to align the client's expectations with our deliverables.

We operated within a sprint duration of 7 days, which offered a balance between ample working time and frequent feedback cycles. Each sprint started and ended with a comprehensive team meeting that included the product owner. These meetings served as platforms for us to evaluate the accomplishments and challenges of the past sprint, and also to establish the objectives for the subsequent one.

To ensure consistent progress, the development team conducted an additional weekly meeting halfway through each sprint. This mid-sprint check-in provided an opportunity for us to assess each team member's progress and make any necessary adjustments.

Our team members actively communicated and arranged meetings using a blend of platforms, including Mattermost, WhatsApp, and Discord. This multi-platform approach enabled us to stay connected and ensured efficient information flow within the team. For interactions between the team and the TUD Teaching Assistant, we relied on Mattermost and weekly meetings, whereas client-developer communication was conducted via email and weekly face-to-face meetings.

We utilized GitLab's built-in 'Issues' module to manage and track tasks, which were derived from each of the requirements outlined in subsection 5.5. The ability to label tasks in GitLab not only promoted an open-to-client development process but also enabled us to keep the workflow organized. To ensure fair workload distribution and effective time management, we further divided each task into subtasks. This granular approach to task management allowed us to maintain a clear and structured view of our project's progress.

5.2 Code Quality, Testing, and Style

In this section, we discuss the implemented measures that uphold the superior quality and reliability of our software tool. Throughout the project, we focused on conducting rigorous testing and ensuring stringent standards of code quality and style.

Our testing strategy was both comprehensive and multilayered, incorporating *unit tests and system tests*. Each **unit test** targeted individual components of the tool in isolation, created concurrently with each new method. This methodology allowed us to proactively detect and address bugs and issues, ensuring the correct functioning of every component before integrating it into the system. Our **system tests**, on the other hand, validated the tool's functionality as an interconnected system, guaranteeing that methods interacted as expected when combined [5].

We used virtual machines and actual evidence files to conduct system testing. In the project's initial week, we set up various virtual machines with different settings, such as an encrypted Windows installation and an unencrypted Windows installation with unallocated space, among others. These were designed to enable us to thoroughly examine the tool's necessary functionality. Our client also significantly contributed to the system testing. They used our tool on their machines, testing it with real-world data files we would otherwise have not had access to. We were able to detect and fix many tricky bugs, thanks to our client's access to a vast amount of data.

In addition to our exhaustive testing strategy, we employed several tools to uphold high standards of code quality and style. Our use of **linting** tools, specifically *pylint* and *flake8*, enabled automated checks on our code's syntax and style, facilitating a uniform and highly readable code structure. We also leveraged *radon*, a comprehensive tool for collecting **code metrics**, to gain insights into our code quality. This aided us in enhancing the maintainability of our code and reducing its complexity, a crucial step in our software development process. Python's support for type hints was another feature we exploited, which considerably reduced the likelihood of introducing type-related bugs and aided linters and Integrated Development Environments (IDEs) in style checking.

In summary, the measures implemented in testing and maintaining code quality highlight the robust approach taken in the development of our software tool. The dual-pronged testing strategy, consisting of both unit and system tests, has ensured thorough validation of the software at both component and system levels. The involvement of our client in testing the tool with actual data files has provided additional reliability. To maintain code quality, automated linting tools and code metric collectors have been utilized, delivering insights into the maintainability and complexity of the code. The use of Python's type hints feature has further contributed to reducing potential errors and assisting in style checking.

5.3 Reflection on Our Testing Strategy

This section narrates the progression and adaptations in our testing methodologies throughout the project. These changes, crucial in the development and refinement of our software tool, were driven by the unique challenges and necessities presented by the project.

Our initial plan to conduct unit tests for each feature before its integration into the system did not prove viable for this particular project. This approach, while systematic, was overly time-consuming and offered limited value, as we predominantly encountered bugs during system testing. The tool's intricate interaction with the system, coupled with the use of multiprocessing, necessitated a more robust focus on integration testing. The latter requirement also led to the occurrence of elusive 'Heisenbugs', bugs that are particularly challenging to reproduce due to their reliance on a highly specific set of concurrent or parallel conditions.

This shift towards system testing and subsequent relegation of complete unit testing to the project's final weeks may lead to perceptions that we deviated from the Scrum development methodology. Typically, Scrum dictates thorough testing before a feature is considered 'implemented', and a separate testing phase aligns more with the Waterfall model. However, elements of our development pipeline, such as our iterative process and weekly decision-making meetings, remain consistent with Agile and Scrum principles. We endeavored to adhere to the Scrum methodology to the best of our abilities, while modifying our approach to testing in response to the ongoing need for intensive system testing.

Initially, we incorporated mutation testing into our development pipeline. However, after the first couple of weeks, we encountered a significant issue: the number of mutants escalated dramatically. As a result of this rapid increase in mutants, our development pipeline's efficiency began to suffer. The situation escalated to a point where the sheer volume of mutants was monopolizing our limited pipeline execution time, thereby stalling other vital tasks in the pipeline. After a thorough assessment of the situation, we made the strategic decision to remove mutation testing from our pipeline. This move was imperative to ensure that our pipeline resources could be more effectively utilized and our development process would not be hampered by unnecessary delays.

5.4 Documentation

Our codebase was documented, every process specified, enabling stakeholders, users, and other developers to efficiently comprehend and utilize the tool. To this extent, we have created written documentation intended for the tool's users, which encompassed:

- A complete description of the tool's functionality, including our chosen analysis methods, their strengths and their limitations.
- A step-by-step installation and quick-start guide.
- A comprehensive Usage Guide, describing anything that an investigator would need to know about the tool, including:
  - A guide on how to use the Command Line Interface.
  - A guide on how to use the Graphical User Interface.
  - An explanation of the report structure and tool configuration files.
  - Examples of possible analysis commands and configurations.
- An in-depth documentation of classes and methods that encapsulated:
  - The structure of each method.
  - The interconnections between methods.
  - The rationale behind each method's existence.
  - The proper usage of the methods, including examples of use cases, inputs, and outputs.

Additionally, the tool was supplemented with a README file, with a brief description of the tool and its functionalities, installation instructions, links to further documentation, and the list of contributors. For the benefit of future developers of the tool, we leveraged autogenerating documentation tools, specifically *pydoc* and *Sphinx*. The generated HTML documentation was then integrated into GitLab Pages, making it easily accessible for reference and further development.

5.5 Requirements

After conducting thorough research ourselves and then gathering insights from the stakeholders, both the non-functional and functional requirements have been extracted from the client meeting on 25/04/2023, as well as from the detailed information provided in the accompanying PowerPoint presentation.

Requirements are listed according to the MoSCoW[12] model for prioritization. We have chosen this method as it provides a clear and structured framework for categorizing features based on their necessity for the project's success, allowing

[12]In order of importance: Must-, Should-, Could-, Won't-Haves

us to create a realistic development schedule and guarantee the minimum-viable product's delivery. This way, adding extra functionality to the final product does not carry the risk of omitting a crucial feature due to time constraints, instead putting the latter requirement earlier in the timeline, to be implemented first. The MoSCoW method thus allows the development team to have a greater buffer to develop the Minimum-Viable Product in case the difficulty of implementing some features has been underestimated, at the expense of the features in the lower-importance categories [6].

5.5.1 Non-Functional Requirements

In this section, we delineate the non-functional requirements, that is, the criteria that judge the operation of a system, rather than specific behaviors. These requirements encompass the behavior of the tool under certain conditions, its capacity, and its limitations.

1. The tool will be written for the less technically-experienced examiners to be able to use in their everyday workflow, enabling them to choose between ***live*** and ***complete*** analysis.
2. The system will be usable via a CLI (Command Line Interface) as well as a GUI (Graphical User Interface).
3. The ***live*** version will use minimum resources to prevent system crashes in the process of detecting encrypted containers, and the tool will generate a findings report within the timeframe of the on-site police inspection.
4. The ***complete*** version will make use of an optimal amount of resources available to achieve the highest performance possible, and it will prioritize a greater accuracy using statistical analysis methods.
5. The tool will be written in the Python v3.8 programming language.
6. Once completed, **DEA** will fit within and be runnable from a commercially available thumb drive.
7. The test suite of the tool, when completed, will have 70% line coverage.
8. The ***live*** version of the tool will only create and write to files present on the thumb drive it was run from, in order to avoid leaving traces on the target system.
9. The tool will inspect the target-system without requiring or accessing an active Internet connection.
10. The tool's internals will only use open-source tools and frameworks.

5.5.2 Functional Requirements

The functional requirements define specific behaviors (or functionalities) that the system should possess. This section is further divided into 4 categories: Must-Haves, Should-Haves, Could-Haves, and Won't-Haves, each reflecting the relative importance and priority of the features in the tool's development.

Must-Haves These are the indispensable features for the proposed tool. The absence of these would drastically impair the tool's ability to fulfil its core functionality and purpose.

1. The system must search for and identify encrypted containers in the medium under-inspection, outputting a findings report that contains the paths to the files flagged as potentially encrypted, so that these suspicious files are then provided as input to cracking software.
2. The system's functionality must be fully documented and made available to the users, and the tool must be extensible, allowing the recognition of additional file types and file systems to easily be integrated in the future.
3. The system must work with E01, DD (raw), and L01 evidence files.
4. The system must recognize and work with the Windows file system.
5. Two versions of the system's pipeline must be implemented: a ***live*** version and a ***complete*** version.
6. The ***live*** version of the tool must operate stealthily, confined solely to the system under-inspection, without producing changes in the files that were inspected, and interacting with the system in a minimum viable manner.
7. The ***live*** version must provide a greater speed of analysis for on-site inspection by only scanning the system for the presence of encryption software, filtering the partitions for containers that are likely encrypted, and storing the paths of both the files flagged as potentially encrypted and those considered benign.
8. The ***complete*** version must provide better accuracy by running statistical checks for pseudorandom data – in addition to the scanning and filtering done by the ***live*** version.
9. Four probabilistic tests must be implemented as the statistical checks for the ***complete*** version: Entropy Test, Chi-Square Test, Monte Carlo Value for Pi Estimation, and Frequency Mean Test[13].
10. The system must be able to access remote files on a network using UNC (Universal Naming Convention) file paths.

[13] These four statistical methods allow for the detection of pseudorandom data in the raw byte-contents of the files they are ran on. The following academic paper provides further information on the mathematical foundation behind these checks: *Theoretical and Practical Aspects of Encrypted Containers Detection – Digital Forensics Approach* [7].

11. The tool must be able to detect the presence and traces (through registry keys, logs, and artifact files) of WinRAR, WinZIP, 7Zip, BitLocker, and VeraCrypt/TrueCrypt installation on a device.
12. The tool must analyze the partition table and report the presence of any unpartitioned sections that contain data.
13. The tool must be able to detect the presence and traces of VMware and VirtualBox software.
14. The tool must enable the user to specify the target partition which the tool will analyze.

Should-Haves This category encompasses the features that, while not strictly necessary for the tool's operation, add significant value and enhance the overall functionality.

1. The system should allow users to select which detection methods to apply when running the tool, enabling the forensic analyst to prioritize between either accuracy or speed.
2. The system should allow users to queue up the analysis of several evidence files, by supplying a list containing the paths of the files to-be-processed.
3. The tool should be able to produce a findings report at each major stage of the inspection, i.e disk, filesystem, and artifacts.
4. The tool running through the GUI should continually display each finding produced during the execution of the analysis (in both ***live*** and ***complete*** versions of the system's pipeline).
5. The system should produce a visual representation of the encrypted disks and files in the findings report that is generated.
6. The tool should also be able to generate a finding report midway through execution if the analyst manually terminates it, even if the analysis is not fully complete.
7. The tool should support the generation of finding reports in three formats: JSON, HTML, and TXT.

Could-Haves Features falling under the Could-Haves section are desirable but not necessary. Their implementation would provide additional convenience or enhancement to the tool, but their absence will not hinder the tool's primary functionality.

1. The tool could work on macOS machines.
2. The tool could detect traces of MacOS Parallels.
3. The tool could work on Linux machines.
4. The system could work with AD1 evidence files and IMG/DMG files found on collected hard drives.

5. The system could detect encrypted files and archives created by WinRAR and WinZIP applications on the device.
6. The system could perform hash analysis of files found within the evidence disk forensic images, applying NSRL hash lists to filter out false positives incorrectly flagged by the tool.
7. The tool could work with encrypted virtual hard disks (VHD's).

Won't-Haves These are features that are identified but will not be implemented in this phase of the project. Although they might be considered for future development, they are not a priority for the current scope of the project.

1. The tool will **not** attempt to decrypt the files it flags as likely encrypted.
2. The system will **not** extract any encrypted containers or files, it is responsible only for storing the paths to said files.
3. The tool will **not** generate forensic images as part of its functionality.

6 Ethical Considerations – From Development to Deployment

The tool our team has developed has clear ethical concerns that we need to take into consideration before deploying the product. As the product will remain in use at the Rotterdam Police for much longer than the Software Project's duration, a forward-thinking values assessment will be conducted in this chapter. The latter will focus mainly on the management of private data, transparency, and trade-offs made that are relevant to ethical implications. First, Section 6.1 discusses the trade-off between privacy and transparency, outlining the relevant accountability concerns. Next, Section 6.2 elaborates on the possibility of false positives and false negatives in the tool's output. Lastly, Section 6.3 considers additional risks, such as the **DEA** becoming more widely used.

6.1 Privacy and Transparency as Conflicting Priorities

DEA is planned to work on suspects' personal devices or forensic images of these devices. The team is not supplied with data from the Police database for testing and uses simulated data for this purpose. Even though this means team members will not interact with private data during the project, the tool will be used on such data when deployed.

Furthermore, we should clarify that the tool does not display private data from a device. **DEA** does not provide functionality to open the files stored on the device under inspection, but rather to identify what is on it. Any encrypted containers found by the tool cannot be viewed or decrypted without the use of another tool built for this purpose or through using the password given by the suspect themselves. However, the tool does provide the investigator the freedom of choosing which file extensions to flag. Therefore, enabling flagging for non-encrypted, yet still interesting files – such as **IMG** or **DMG** files – is possible.

Flagged files can be viewed by navigating to the address provided by the tool, which is necessary to assess their importance to the case. Given that the nature of the crimes relies on the production of visual media of various types, this functionality has to be preserved. In essence, the tool is not introducing a new inspection process, but making the existing process quicker. Privacy issues about the usage of the data on the device under inspection are delegated to the **General Data Protection Regulation (GDPR)** and **Law Enforcement Directive (LED)** published by the EU, and are beyond the responsibility scope of the team [8].

A significant concern with a forensic tool is whether the tool functions only towards its stated goal, or if a second malicious agenda is at play in the background. Claims could be made that the tool has a feature that forges evidence and not being able to prove otherwise could lead to the tool being detracted on wrong grounds. To this end, **DEA** code source is provided to the police. If the police deemed necessary, the source code could be provided to a third party in order to assess its behavior. This will allow investigators to negate potential allegations regarding the tool's intent in a court of law.

When drafting the project requirements, the client was adamant about the tool not creating or modifying files on the system-under-inspection. Therefore, the team designed the product to leave minimal traces behind after operation. This approach curtails the possibility of the suspect's defense accusing forensic investigators of planting or skewing evidence, thereby upholding the clarity of the investigation process and the tool's integrity.

To further strengthen the accountability of the tool, we have added the functionality to include a date and time of analysis in the findings report. In addition to this, the report also includes the name of the inspector, the name of the device, where the device was located (encoded in a format stating which room of the house the device was found), which version of the tool was run, and which analysis techniques were used. In this way, the tool is transparent about how, when, and where it was used.

6.2 Consequences of Inaccurate Analysis

Compromises in accuracy had to be made in keeping with the requirements that were met, and these are acknowledged in this section. For the live version of our tool, speed was specified as the priority. To this end, timewise-costly analysis methods are not used in this version. However, this has increased the probability of *false negatives* (such as when quicker methods cannot detect traces that are there), and *false positives* (such as corrupted/compressed files that are mistaken to be encrypted) occurring.

False positives could cause forensic investigators to use the tool to chase a dead-end, and thus waste valuable time. Beyond that, false positives can mean a suspect's device is confiscated for no good reason, potentially causing difficulties until the police returns the device. On the other hand, false negatives could cause the premature dismissal of potentially important pieces of evidence for the investigator's case. This could result in the suspect subsequently removing the evidence, and thus greatly reducing the possibility of catching the suspect in a second investigation. As we understand these risks are created by this necessary trade-off, choosing to run the live version of the tool raises a warning that explicitly states this potential. Thus, by making the end-user aware of that, we improve the transparency and explainability of the tool's internal operation.

6.3 Additional Risks and Their Implications

As the project objective serves to benefit the justice system where we all live, we feel it is necessary to touch on the intended ethical results and the group's motivation in developing the tool. Firstly, the tool is designed to speed up and automate the already-existing process of searching for evidence. This implies that cases can be supported or refuted with the truth much sooner than when no such digital forensic tools are available [9]. Furthermore, our supervisor at the Police mentioned that if the tool is found useful and reliable within the

Rotterdam force, it could be distributed to and used by units in other regions of the Netherlands in the future.

The prospect of our work being distributed and used to fight crime at a larger scale increased motivation within the group, urging us to pay closer attention to writing explainable and well-documented code, as well as making the final product maximally extensible and adaptable to different forensic-image formats, filesystems, and investigation circumstances (both on-site at a suspect's place of residence and at a police station).

With the potential of the tool's usage being widened comes the possibility that is used by other police divisions than TBKK. For example, what happens if the tool is used by other divisions to investigate devices of potential terrorists, tax evaders, or drug traffickers? Although these crimes sound more severe and thus give the impression that the ethical implications of the tool would be widened, the way suspects and their devices are treated remains the same in essence. Suspects still have their privacy rights and can still be convicted or evicted with the support of evidence discovered using **DEA**. Therefore, the arguments presented in this section still hold.

One possibility that deserves explicit recognition is the tool escaping (either knowingly or not) into the hands of parties outside the police force. If these parties are still organizations of a government, the aforementioned legislation **GDPR** applies and there is no cause for concern. But if they are not, then we should consider how the police handles any of the powerful tools it has at its disposal. Like **DEA**, leakage of any software tool to third parties is undesirable and dangerous. In the wrong hands, these tools can be modified to serve malicious intent. Therefore, their owners should be taking appropriate precautions to prevent such situations.

7 Recommendations

As the team had limited time during the development of **DEA**, this section introduces *four* points of recommendation regarding the further development and the use of the tool. These recommendations focus on giving advice about how to increase the tool's effectiveness through extensions, and how to best utilize the tool's capabilities.

First, the addition of Dutch language support to the tool is the first extension that is recommended considering that it will primarily be used in the Netherlands. This will enable users to interact with the tool in the comfort of their native language. However, the translation process should be conducted with caution. The tool makes use of frequent and informative warnings which should be maintained and extended in the Dutch version. Preserving transparency and ease-of-use of the tool in both languages will increase usability, and thus allow it to be in use for a longer period of time.

Second, extending the tool to support analysis of Linux and MacOS devices is recommended to make the tool universally usable. Even though Windows is the predominant operating system found on devices, Linux and MacOS devices are intermittently encountered. Extending the tool to handle these operating systems' partition tables and file systems is convenient for future developers thanks to the extensible design in place.

Third, if new analysis techniques are to be added to the tool, these should focus on reducing the number of false positives. The current logic implemented by the tool dictates that flagging a potentially suspicious file is preferable to allowing it to pass unnoticed, but more robust techniques can be added to reduce these to a minimum.

One such technique is Hash analysis. This analysis method involves hashing programs found on the device under inspection, and comparing these to the hashes of known programs in the National Software Reference Library (NSRL) [10]. In addition, newly found suspicious files can be hashed and compared to hashes of previously discovered files in the Police database.

Fourth, for the end users of the tool, it is recommended to prepare DEA configuration files for use in specific situations. To achieve optimal utilization, analysis techniques can be dynamically enabled and disabled depending on the context. The most frequently encountered situations can be made easier to handle by keeping a configuration file specifically suited for that circumstance. Techniques that are quicker and more effective at finding large artifacts can be enabled for a field configuration, while time-consuming yet accurate techniques can be enabled for a configuration set to run overnight.

8 Conclusions

DEA is our group's solution to the challenges that forensic investigators face, namely the large variety of encryption tools available and the very large sizes of the drives and disk images under investigation. This report set out to provide justification for the design choices our group made while developing the tool. To provide this justification, we considered 3 possible architectures: Layered, Monolithic, and Pipe-and-Filter. The advantages and drawbacks of each were discussed, and after careful consideration, support was given for the eventual choice of proceeding with the Pipe-and-Filter architecture. Supporting design patterns were also described, elaborating on **DEA**'s inner functionality.

An approach that narrows down the search space after each stage was deemed appropriate, given the large containers that are analyzed with the tool. Thus the Pipe-and-Filter architecture was chosen for the implementation of the **DEA**. With this architecture, each analysis technique acts as a filter reducing the search space. An *artifact queue* together with the *predecessors* attribute of each analysis technique functions as a pipe between the methods.

Furthermore, the Pipe-and-Filter architecture increases the modularity and reduces the complexity of interactions between analysis methods in comparison to a layered architecture. New analysis techniques are easy to add to the pipeline by defining the *predecessors* of each technique. Furthermore, passing an artifact to another method is as simple as enqueueing it to the *artifact queue* as a pair with the method of analysis.

The project process has shown that further development of **DEA** is crucial to increase the efficiency of forensic analysis. To continue development and ensure efficient use of the tool, this report recommends *four* items.

First, it is recommended to add Dutch language support while maintaining and extending the tool's warnings. The addition of Dutch support and the upkeep of transparency will increase trust in the tool for new users. This will subsequently allow it to help more investigators for a longer time.

Second, a team of developers could add Linux and MacOS support to **DEA** in order to expand the use cases of the tool. Even though Windows is the predominant operating system encountered in suspect devices, support for Linux and MacOS would make the tool universally usable.

Third, the new analysis techniques added can focus on reducing the amount of false positives through more vigorous filtration and analysis. It is recommended to hash found programs for comparison with the NSRL database, and to hash found files to compare against files in the Police database.

Fourth, it is recommended to prepare configuration files and to dynamically enable/disable certain analysis techniques depending on the context. This will enable the efficient use of the tool in situations demanding swiftness, or conversely in situations that require thorough analysis.

References

[1] B. Lyons, “Disk Image Content Model and Metadata Analysis,” Harvard University Libraries, Brooklyn, New York, Tech. Rep., 2016.

[2] Magnet Forensics, “Encrypted Disk Detector,” 2020. [Online]. Available: https://www.magnetforensics.com/resources/encrypted-disk-detector/

[3] A. Davies and A. Tomlinson, “Detecting the Use of TrueCrypt,” Royal Holloway, London, United Kingdom, Tech. Rep., 2010.

[4] A. L. Rukhin, “Random Number Generation Tests,” in *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. U.S. Dept. of Commerce, National Institute of Standards and Technology, 2000, pp. 23–62.

[5] M. Aniche, A. v. Deursen, and S. Freeman, *Effective Software Testing: A Developer’s Guide*, 1st ed. Manning Publications, 2022.

[6] E. Miranda, “Moscow Rules: A Quantitative Exposé,” in *Agile Processes in Software Engineering and Extreme Programming*, V. Stray, K.-J. Stol, M. Paasivaara, and P. Kruchten, Eds. Cham: Springer International Publishing, 2022, pp. 19–34.

[7] I. Jozwiak, M. Kedziora, and A. Melinska, “Theoretical and Practical Aspects of Encrypted Containers Detection – Digital Forensics Approach,” in *Dependable Computer Systems*, W. Zamojski, J. Kacprzyk, J. Mazurkiewicz, J. Sugier, and T. Walkowiak, Eds. Springer Berlin Heidelberg, 2011, pp. 75–85.

[8] B. Custers and L. Stevens, “The Use of Data as Evidence in Dutch Criminal Courts,” *European Journal of Crime, Criminal Law and Criminal Justice*, pp. 25–46, 2021.

[9] M. B. Seyyar and Z. Geradts, “Privacy Impact Assessment in Large-Scale Digital Forensic Investigations,” *Forensic Science International: Digital Investigation*, 2020.

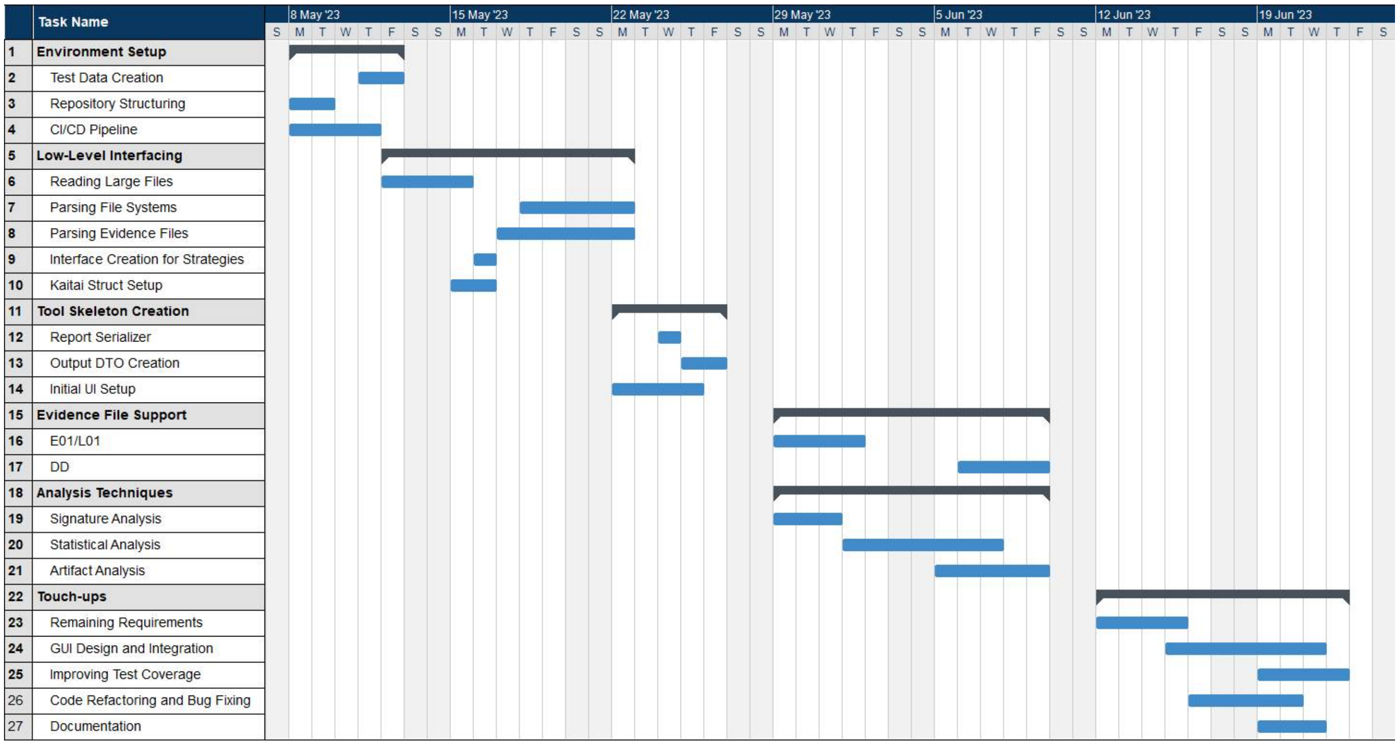
[10] N. Rowe, “Testing the National Software Library,” *The Proceedings of the Twelfth Annual DFRWS Conference: Digital Investigation*, 2012.

A Work Distribution among Team Members

Task	Description	Group Members				
		Kaan Altinay	Mihnea Bernevig	Yigit Colakoglu	Matej Tomasek	Konstantin-Asen Yordanov
Implementing concurrency and the Engine	The engine allows the CLI to run, parses arguments, manages the queue of tasks and handles multithreading.			🟩	🟩	
File Object Creation and File System Handling	File systems need to be parsed and files be extracted from them. This is crutial for analysis techniques to be conducted.				🟩	
File Signature Analysis	File signature analysis compares the headers of files with their extentions, and marks files with mismatches. It additionally marks files with interesting types.		🟩			
Statistical Analysis	Statistical analysis uses mathematical techniques to determine the randomness of data in a given container. If the data is overly random, it is possibly encrypted.	🟩				
File Size Filtering	File size filteration reduces the search space for other analysis techniques, greatly reducing the runtime for the tool.		🟩			
Creating Reports and corresponding Serializers	The tool supports writing of finding reports in multiple useful formats. These get their data from the corresponding serializers which aggregate data from result DTO's.					🟩
Registry Analysis	Analyses the Windows Registry files for traces of encryption software and suspicious activity on the user's behalf (such as recently used applications and accessed documents)	🟩				
Common Directory Analysis	Analyses Program Files, Users, AppData, and other common Windows directories for encryption artifacts.					🟩
Event Log Analysis	Parses the Event logs of the Windows System, checking for unusual devices, or to devices mounted to unusual letters.		🟩			
Result Data Transfer Object	The Result DTO is an essential component that facilitates data transfer between components of the DEA.					🟩
Visual Creation	Multiple visual formats such as Histograms, Tree Plots and Pie Charts are used to visualize data available in the findings reports for easier understanding.	🟩				
GUI Creation	The graphical user interface translates the CLI to allow users with less technical knowledge to interact with the tool efficiently.		🟩	🟩		
CLI Creation	The command line interface is the main method of interacting with the system. It allows users to choose which methods to use and which files to conduct the analysis on.			🟩		
Analysis Graph	The analysis graph is for the management of different analysis technique and connecting them to each other using predecessor logic so that the pipe and filter architecture can be achieved.			🟩	🟩	

B Project Timeline

Figure 7 presents the Gantt chart, which provides a comprehensive overview of the anticipated timeline for the **DEA** tool's development. The priority and significance of each task were determined based on its dependency relationships with other tasks. This chart offers a precise depiction of the project's actual timeline, with some notable deviation: the shift of Signature Analysis to the project's second week and the focus on unit testing in the last weeks.



	Task Name
1	**Environment Setup**
2	Test Data Creation
3	Repository Structuring
4	CI/CD Pipeline
5	**Low-Level Interfacing**
6	Reading Large Files
7	Parsing File Systems
8	Parsing Evidence Files
9	Interface Creation for Strategies
10	Kaitai Struct Setup
11	**Tool Skeleton Creation**
12	Report Serializer
13	Output DTO Creation
14	Initial UI Setup
15	**Evidence File Support**
16	E01/L01
17	DD
18	**Analysis Techniques**
19	Signature Analysis
20	Statistical Analysis
21	Artifact Analysis
22	**Touch-ups**
23	Remaining Requirements
24	GUI Design and Integration
25	Improving Test Coverage
26	Code Refactoring and Bug Fixing
27	Documentation

Figure 7: Gantt Chart of the Project Timeline