QUANTIFYING UNIFORMITY OF OUTPUT OF OPEN-SOURCE DIGITAL FORENSIC PLATFORMS

LUUK VAN CAMPEN
NEDERLANDS FORENSISCH INSTITUUT

TU Delft

Quantifying Uniformity of Output of Open-source Digital Forensic Platforms

by

L. van Campen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday April 23, 2024 at 9:30 AM.

Student number:	5873738	
Project duration:	October, 2023 – April, 2025	
Thesis committee:	dr. C. Lofi,	TU Delft, supervisor
	Prof. dr. H. van Beek,	Netherlands Forensic Institute
	dr. A. L. D. Latour,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

Dear reader,

This thesis (sort of) marks the end of my time in Delft. Two years ago, I chose to study here, leaving behind an established student-life in Nijmegen. Little did I know that besides a nice curriculum, this move would bring with it to write my Master's thesis during an internship with the Netherlands Forensic Institute. This internship provided me with an insight into the forensic domain of which I knew so little, and the opportunity to write my thesis within the Formats team, of which I felt part from my first day there, and from which I have learnt very much.

I would like to express my gratefulness to everyone who has suppported me over the past months. Firstly, I want to thank dr. Christoph Lofi for being my supervisor and committee chair. Your sharp questions and remarks and positive attitude were very helpful. I am grateful to dr. Harm van Beek. I really enjoyed our weekly meetings, during which you helped me with my thesis but also taught me a lot about the digital forensic domain. You will make an amazing professor at the Open University. Next, I thank Roel van Dijk. Our many conversations have been very helpful to me, and your insights, trust, and positive attitude have been very valuable. I also thank dr. Anna Latour for being part of thesis committee. Lastly, I thank my dear family, Marieke, and my friends for their unconditional support. It would have been more difficult without having you.

Live long and prosper.

Luuk van Campen
The Hague, August 2024

Abstract

In forensic investigations, an increasing amount of evidence is retrieved from digital devices. This evidence is often extracted from devices using digital forensic platforms. The platforms are able to extract digital traces from several types of files, originating from different applications, such as email applications from laptops or chat applications from smartphones. Developing support for a new file format and adding it to a digital forensic platform is time-consuming and difficult. An alternative approach is to integrate an existing digital forensic platform into another platform to extend its capabilities. This is, however, not trivial because firstly, it is difficult to choose a platform to integrate. This difficulty stems firstly from the fact that there exists no overview of open-source digital forensic platforms that can be used to compare advantages and disadvantages of the platforms. The first aim of this thesis is therefore to provide an overview of available open-source digital forensic platforms. This overview is created by means of online research. The second reason that it is difficult to choose a platform to integrate is that it is difficult to assess whether the output of these platforms is accessible and structured such that it can easily be integrated. The second aim of this thesis is therefore to determine what the best method is to quantify the uniformity of the output of digital forensic platforms and to develop a proof of concept implementing this method. Developing the uniformity metric is done by first dividing the concept of uniformity into six sub-forms of uniformity. Conceptual methods to quantify each of those forms are provided, and we present a concrete implementation of a proof-of-concept. The results of this thesis imply that although development of digital forensic platforms is actively ongoing, developers miss out on improving the digital forensic field as a whole by not considering the interoperability of the platforms they develop.

Contents

1 Introduction

Computers and smartphones are becoming more and more intertwined with our daily lives. We use them for the most menial tasks, from checking train departure times to communicating with friends and colleagues. While in use, these digital devices store various kinds of data, ranging from browser history and WhatsApp messages, to the timestamp at which the last connection with a particular WiFi network was made or when a phone was last plugged into a charger. Although this information is generally of little use or even unavailable to regular users, it can be of vital importance in forensic investigations. The time at which a phone last connected to a particular WiFi network, for example, places a phone at a particular location at a particular time, which is information that helps a judge form a picture of what events must have taken place.

The digital forensics process consists of three phases[4], namely acquiring data from a device, gaining insight into and analysing the data in order to get relevant evidence, and lastly, reporting the found evidence to the relevant organisations. The second step, gaining insight into and analysing the data, revolves around the use of digital forensic tools. These tools generally take some sort of unstructured data and produce digital traces, which may serve as evidence. There exist various digital forensic tools, ranging from commercially available to open-source, and from being able to extract trace from a single file, to ones that take entire disk images as input and extract traces from many filetypes. We refer to open-source tools that are able to extract traces from multiple file types, open-source digital forensic platforms.

The Netherlands Forensic Institute has developed the digital forensic platform Hansken. It is able to extract traces from many different file formats and present these traces in a convenient user interface in which forensic investigators can find relevant evidence for their case. From the viewpoint of the case investigator, this can be phrased as Hansken ”supporting” a particular application. For example, the case investigator cares only that he can view emails originating from Apple mail in Hansken, and the name of the Apple mail database does generally not concern them. The process of adding support for a particular application to Hansken consists of several steps, the first of which is finding out what files the application stores its data in. Secondly, the structure and the meaning of the contents of these files has to be understood so the third step, parsing the file, can be done. The results of parsing the file can in turn be represented in Hansken as digital traces. In Hansken, all traces are represented in terms of the Hansken trace model. The trace model lists the concepts that traces found in files can be mapped to in order to be valid Hansken traces. Examples of concepts are ”email”, which is used to represent any email message, or ”GPS”, which is used to represent a set of coordinates. These concepts all have

certain properties; in the case of ”email”, this could be the email address of the sender, the timestamp at which the email was sent, and several more. So, to show found traces in Hansken, a mapping has to be created between the parsed data and the Hansken trace model. This would, for example in the case of Apple mail, consist of mapping the column containing the email address of the sender of an email to the ”email sender” property in Hansken.

To make sure that Hansken is able to gather as many traces from digital devices as possible, the developers of Hansken are continually developing support for the extraction of traces from new or updated applications. One approach to do this is to add support for applications by integrating existing open-source digital forensic platforms that contain support for the application, into Hansken. This has the advantage that it is no longer necessary to research what files an application stores its data in, how the file is structured and what its contents mean, and to write a parser for the file. What remains is only to create a mapping between the output of the integrated digital forensic platform and the Hansken trace model.

Determining whether or not to integrate a particular platform into Hansken is not straightforward for several reasons. Firstly, documentation of open-source digital forensic platforms is often incomplete. The consequence of this is that is often unclear what applications or file types the platform supports, and what the output of the platform means, which is crucial when creating a mapping from the output to the Hansken trace model. Another problem is a lack of uniformity within the output of the digital forensic platforms. To illustrate what this means, we introduce a running example1.1, This figure shows the output of a fictional platform that supports Apple mail and Outlook mail. Although there are two email databases in this example, their names, number of columns, the meaning of the columns, and the formatting of the timestamp, are all different. These differences make creating a mapping from the output of the platform to the Hansken trace model more involved. The Hansken trace model defines one concept ”email” for both emails. Since the output of the platform is not uniform in its naming and formatting, a time-consuming and error-prone mapping has to be made for each supported email application.

The goal of this thesis is twofold. Firstly, we provide an overview of open-source digital forensic platforms that provides the reader with information to help them decide whether or not to integrate the platform into their own platform.
Secondly, we provide a definition of uniformity of the output of open-source digital forensic platforms, as well as functions that can be used to compute the uniformity of such output. We also present a proof-of-concept that is a concrete implementation of these functions.
Using this overview and uniformity metric, developers of Hansken can make a more well-founded decision on whether or not to integrate a platform into Hansken. It also makes it possible to objectively and consistently compare the uniformity of outputs, rather than basing it on intuition.

This thesis also aims to raise awareness within the digital forensic community about the interoperability of the platforms that are developed. If developers would take the uniformity of the output of the open-source digital forensic platforms they develop into consideration more, this would benefit the entire open-source digital forensics community, since digital forensic knowledge can be reused more easily, allowing developers to spend more time on developing support for new file formats or maintaining support for existing formats.

1.1. Research Questions

The goals of this research are achieved by answering the following research questions.

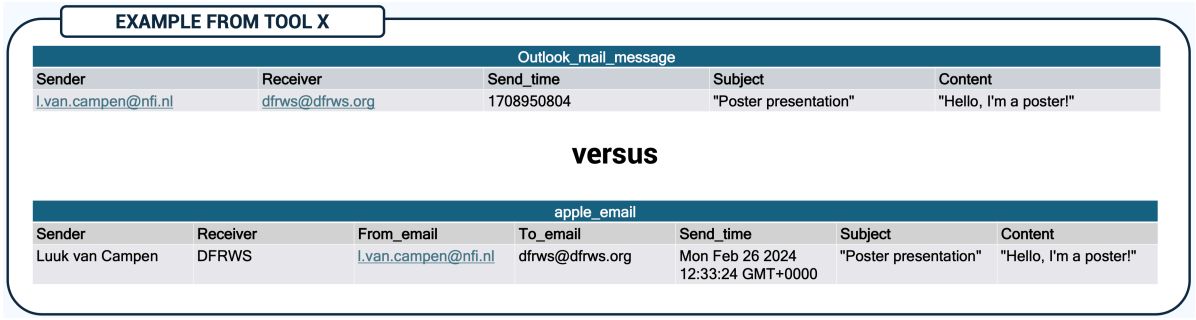

Figure 1.1: Running example displaying a lack of uniformity

RQ-1: What open-source digital forensic platforms are available and how do they compare on certain relevant properties?

In this part of the research we create an overview of the various open-source digital forensic platforms that are available. We discuss each platform based on several properties so we can make a well-founded decision on what platform is suitable for integration into Hansken.

RQ-2: What is the best way to quantify uniformity of the output of open-source digital forensic platforms?

The output of open-source forensic platform consists of sets of traces of many different concepts. With uniformity of the output, we mean that if two traces represent the same concept (according to end users of such platforms), they should be named and formatted similarly. We are not aware of a standardised way to quantify such uniformity of a set of semi-structured data. So, we create a definition of this uniformity, propose an approach for quantifying it, and argue for this currently being the best approach. This quantification can then be taken into account by developers when estimating the time and effort needed to integrate such output into Hansken.

RQ-2.1: How accurate is the uniformity metric developed in this thesis?
We assess the accuracy of our approach to quantifying uniformity by comparing it to uniformity values assigned based on intuition by experts in the digital forensic domain. This tells us whether the approach developed in this thesis is of value to domain experts.

RQ-3: Proof-of-concept

We implement a proof-of-concept which largely automates the process of computing uniformity. The aim of this proof-of-concept is to show that the proposed uniformity metric can be successfully automated and used to evaluate the uniformity of output of digital forensic platforms. We assess the perceived workload of using the proof-of-concept and its accuracy in sub-questions 3.1 and 3.2.

RQ-3.1
How do the perceived workloads of quantifying uniformity based on intuition, the approach introduced in this thesis, and the proof-of-concept compare?

We evaluate the perceived workload of assigning a uniformity score to the output of a digital forensic platform in three different ways, namely based on intuition, using the approach introduced in this thesis, and using the proof-of-concept. By comparing the perceived workload of manually computing uniformity and that of using the proof-of-concept, we can quantify whether the proof-of-concept reduces user effort, thereby increasing the chance of it being adopted by users.

RQ-3.2
How accurate is the proof-of-concept compared to manually computed uniformity scores?

We compare the uniformity scores assigned by experts from the digital forensic domain to those resulting from the proof-of-concept. If we can determine that the results produced using the proof-of-concept are similar to those produced by manually computing uniformity, we show that the proof-of-concept produces accurate results.

1.2. Outline

The rest of this thesis is structured as follows. First, an overview of open-source digital forensic platforms is provided and we present a comparison based on relevant properties. This chapter answers research question 1. Next, in chapter 3, we introduce six types of uniformity, and for each type of uniformity we introduce a function that can be used to compute that type of uniformity. We conclude the chapter with an example uniformity computation. This chapter answers reserach question 2. In chapter 4, we discuss the proof-of-concept. This PoC is a concrete implementation of the uniformity functions introduced in chapter 3. This chapter answers research question 3. In Chapter 5, we discuss the experiment that was conducted to assess the accuracy of the uniformity functions, the accuracy of the PoC, and the perceived workload of computing the uniformity functions by hand and by using the PoC. This chapter yields results that are used to answer research questions 2.1, 3.1, and 3.2 in chapter 7, the discussion. We then proceed to discuss related work, discuss the results presented in this thesis and answer research questions 2.1, 3.1, and 3.2. The last chapter contains a conclusion.

2

Overview of Open-source Digital Forensic Platforms

The goal of this chapter is to answer research question 1, **What open-source digital forensic platforms are available and how do they compare on certain relevant properties?**

We only take into consideration platforms that support multiple file formats. We do not set a specific minimum number of file formats that has to be supported by a program for it to be included here, but rather intend this criterion to filter out programs that only support a single type of file. We start by introducing parameters on which we base the discussion of the platforms. We then proceed with the overview of digital forensic platforms.

2.1. Parameters

In this section, we discuss parameters on which we can base the discussion of open-source digital forensic platforms. These parameters are identified in cooperation with the Hansken developers, who are experts in the digital forensic domain.

Number of supported formats If Hansken supports more file types, it will produce a more complete overview of digital traces found.

Last updated Software, and therefore potentially the traces produced by that software, can change rapidly due to software updates. To make sure that the extraction of traces from files is sound, it is important to update the platform when necessary. A platform not having been updated in a long time might be an indication that it is outdated, which in turn might mean that it is not able to extract traces in a sound and complete way from modern versions of the software for which it was intended.

Modularity The ability to only use required parts of a platform makes its use more flexible. If, for example, Hansken can already extract traces from a subset of the file types supported by a platform, the ability to integrate only a part of the platform might save the developers costly time.

Test data It is important that the results generated by the platform are forensically sound. It is therefore valuable if test data is included in repositories so the functioning of the tool can be verified against that data.

	# Supported Formats	Last Updated	Testdata Included
Plaso	219	9-6-24	Yes
IPED	56	29-6-24	Yes
iLEAPP & Co.	683	25-6-24	No
Dissect	29	27-6-24	Yes
Kuiper	Many	5-6-24	No
mac_apt	48	16-6-24	No
iOS_sysdiagnose_forensic_scripts	14	8-11-19	No
Apollo	247	3-12-20	No

Table 2.1: Summary of results from the overview of open-source digital forensic platforms.

2.2. Platforms

In this section, we provide an overview of open-source digital forensic platforms. We discuss the platforms based on the parameters introduced in the previous section. This overview contains information on when a platform was last updated according to the platforms Github repository. This information was up to date in August of 2024.

2.2.1. Plaso

PLASO is an open-source digital forensic platform that contains several sub-tools, each aimed at a particular task in the digital forensic process. The tool ”log2timeline” can be used to construct a timeline of timestamps found in either a single file, or recursively found in a directory. The result of this process is a PLASO storage file, which can in turn be inspected by using PLASO’s ”pinfo” tool. The PLASO storage file can be post-processed using PLASO’s ”psort” tool. Using this tool, the storage file can be queried and sorted, and it provides the possibility to run automatic analysis on the file.

PLASO is able to extract traces from many different file formats, such as various SQLITE databases, PLIST files, and more. The source code is built up out of parsers for each of the supported file types, and the code of each parser adheres to a similar structure. Since the overall goal of PLASO is to create a timeline, all parsers create instances of ”events” that can go on this timeline. Depending on where the event is created, it is enriched with traces specific to the file type the event was extracted from by the parser. Since all parsers in PLASO adhere to a similar structure, it is relatively easy to extract a single parser from PLASO’s source code and execute that parser standalone, or integrate it into a different digital forensic platform.

What the output of PLASO means exactly is not specified, so this has to be found out by either reading the source code or inspecting the output of PLASO. The consistent structure of the source code does make it relatively easy to find out what traces are extracted from files, since the traces are, in each parser, assigned in a function that has the same name in all parsers. This means the PLASO’s code has good modularity.

The Github repository shows regular commits to PLASO’s source code, which can be an indication that the forensic knowledge within the platform is kept up-to-date.

In conclusion, PLASO has several properties that make it an attractive choice for integration into other digital forensic platforms, whether that be partially, or as a whole. Among these properties are the ability to run parsers standalone of the rest of the tool, the consistent structure of the code of the parsers,

and the large number of parsers included in the platform.

2.2.2. IPED

IPED is an open-source digital forensic platform developed by the Brazilian police. It can be used to extract and analyse traces from images of digital devices. It has an extensive user interface and it contains many features, including data carving, image recognition, and displaying communication links.

The IPED platform is able to extract traces from well over 1000 different file types. The platform is written in Java. Parsers generally implement the interface AbstractParser, which originates from Tika, a framework which can be used to build parsers. Each parser is written in its own class, and the fact that parsers implement this interface implies that they all adhere to a similar structure. This uniform structure makes it easy to extract a single parser from the platform and to execute it standalone.

The IPED platform produces its output in XHTML format, but there is no formal description of what is contained within those files. It is also possible to access IPED's output in a structured manner by means of a Lucene index. This information was kindly provided by the developers of IPED, who responded quickly to questions asked on their GitHub discussion page. In general, IPED's GitHub repository is lively and it shows that commits are regularly made.

In conclusion, IPED is a valuable source of ready-made parsers and digital forensic knowledge in general. The fact that parsers can be executed standalone and therefore integrated into other digital forensic platforms easily makes the knowledge contained within it easily accessible. The fact that the code of all parsers adheres to a similar structure also helps with this. All in all, this platform, whether as a whole or in part, is a suitable candidate for integration into other forensic platforms.

2.2.3. Abrigoni's Parser Collection

The Abrigoni GitHub account contains several digital forensic platforms, among which are iLEAPP and ALEAPP. iLEAPP is a platform intended for parsing and extracting traces from files found on iOS and iPadOS devices. Each parser generates a tsv file containing the parsed data, which makes the output of this platform easily machine-readable.

The documentation on iLEAPP is extensive. It contains an overview of file types that can be parsed by iLEAPP, and the source code generally contains a comment describing the knowledge based on which that particular parser was built.

iLEAPP works by dynamically loading the parsers when it is executed. To allow this dynamic behaviour, all files containing a parser have to specify some information in a structured format specified in the iLEAPP documentation. This format contains, among other things, the name of the parser's entry function, a description of the parser, and what file type it parses. The fact that iLEAPP loads the parsers dynamically implies that the parsers have low coupling with the rest of the iLEAPP code, which in turn means that it is probably relatively uncomplicated to execute the parsers standalone.

The Github repository that iLEAPP belongs to also contains various other open-source digital forensic platforms, one of which is ALEAPP. The platform ALEAPP is intended for use on images retrieved from Android devices. The parsers integrated in ALEAPP adhere to the same structure as those in iLEAPP,

which makes them also easily standalone executable.

Aside from ALEAPP and iLEAPP, the Github account from which these platforms originate also contains a digital forensic platform to investigate the onboard computers of cars, Returns, and Windows artifacts.

In conclusion, the Github account from which these platforms originate contains a wealth of forensic knowledge, and the developer clearly made an effort to standardise the code of the different parsers, even across repositories. Either iLEAPP or ALEAPP are good candidates for integration into other digital forensic platforms, mainly because of their consistent coding practices and ability to execute parsers standalone.

2.2.4. Dissect

Dissect is an open-source digital forensic platform developed by the company Fox-IT. The developers describe it as ”digital forensics and incident response framework”. It consists of various different tools that can all be installed and executed standalone via PyPi, and the source code of the various tools is available on Fox-IT’s Github repository.

The documentation on Dissect is extensive, which might have something to do with the fact that Fox-IT is a commercial party. All parts of Dissect are well described. The documentation states that all libraries are designed such that they are reusable. In combination with the fact that all libraries are intended for standalone use, this means that it is straightforward to integrate the libraries into other digital forensic platforms.

In the documentation, the parsers are called plugins and per plugin it is described what they parse, and often also what is returned. This allows Dissect’s users to know beforehand what traces to expect from certain parsers. Compared to other open-source digital forensic platforms, this level of documentation is exceptional.

The documentation contains guidelines on how to develop plugins that can be used in Dissect, and contributing these plugins is encouraged. Quality of contributed plugins is assured through a review process and guidelines on what tests are expected of a new plugin. The option to export the output as json is also included, which makes it easy to integrate Dissect’s output in other platforms.

The structure of all parser’s source code generally adheres to the same structure. This makes it relatively uncomplicated to execute the parsers standalone or execute them dynamically from different software.

In conclusion, the large number of file types supported by Dissect, their well-structured and clear source code, and ability to execute parsers standalone, make this platform a good candidate for integration into other digital forensic platforms, whether that be as a whole, or in part.

2.2.5. Kuiper

Kuiper is an open-source digital forensic platform that is essentially a free version of the commercially available CyberBomah. At first glance, an interesting property of Kuiper is that it explicitly mentions that it aims to be consistent: ”Depending on different parsers by team members to parse same artefacts might provide inconsistency on the generated results, using tested and trusted parsers increases the accuracy”. Some commits are visible on the Kuiper Github repository, but these seem to be minor bug-

fixes and improvements, rather than adding support for new file types or keeping currently supported file types up to date.

The source code of Kuiper contains a directory containing all parsers. The parsers are stored in subdirectories and are accompanied by a file called ”configuration.json”. This file specifies the entry function of the parser, which allows it to be dynamically loaded and executed by Kuiper. The ”configuration.json” file also contains information on what file the associated parser supports. This is, for example, done by means of a regular expression that matches on the name of the supported file.

The Kuiper documentation includes a guide on how to develop parsers for Kuiper, and from this documentation it becomes clear that the parsers themselves return a list of JSON objects. The fact that they parsers return a list of JSON objects and are executable via a single interface function makes it so that they can be executed standalone relatively easily. The documentation of Kuiper does not specify the meaning of the output of parsers, so the meaning of the output has to be derived from the field names of the JSON objects returned from the parsers.

Running Kuiper can be done via their provided Docker images. If the goal is to integrate Kuiper into another digital forensic platform, however, it is most likely more convenient to pick the desired parsers from the source code and integrate those directly. This is due to the limited API that Kuiper provides and the fact that Kuiper stores its traces on a separate ElasticSearch server, which is included in the Docker image. Running a particular parser and parsing the returned JSON file is less complicated.

In conclusion, the parsers integrated in Kuiper contain valuable digital forensic knowledge which can either be used for development of other parsers, or the parsers can be used entirely and integrated easily into other digital forensic platforms. Running an instance of Kuiper, however, is likely complex and due to their limited API, automating the usage of Kuiper is probably too limited for extensive integration into other digital forensic platforms.

2.2.6. mac_apt

Mac_apt is used for examining iOS and MacOS devices and it is able to extract traces from many interesting files. Contrary to many other platforms discussed in this overview, this platform provides an explanation of its output. Per parser, it is described what the meaning of each field in the output is. It can currently extract traces from 45 different types of files.

The Github repository of mac_apt shows regular commits, which might mean that the forensic knowledge within it is either being expanded or maintained, which ensures good soundness and completeness of the extracted traces.

Mac_apt currently also contains the tool ios_apt, which can be used to extract traces from relevant files on iOS devices. Previously, these tools were available separately.

The parsers included in mac_apt are called plugins. The plugins are all written in Python and each source code file contains properties of that particular plugin, such as a description and information on the author. The plugins can easily be executed standalone by specifying the desired plugin to execute and the file on which to execute it. This makes it uncomplicated to integrate a particular plugin into a different digital forensic platform. The ability to execute the parsers individually makes it easy to integrate only the desired parts of the platform.

2.2.7. iOS_sysdiagnose_forensic_scripts

The Github repository iOS_sysdiagnose_forensic_scripts contains parsers for 14 sysdiagnose files present on iOS devices. These sysdiagnose files are generally log files and they can contain information relevant in digital investigations.

The parsers in this Github repository are all individually executable Python scripts. Contrary to many other platforms in this overview, there is in this case no overarching program that loads and executes parsers dynamically. In this case, all parsers have to be executed standalone. This makes this Github repository arguably not a digital forensic platform, but rather a collection of useful scripts. Since the tools parse relevant files and are structured similarly, it was decided to include it here regardless. The scripts provide JSON output.

The last commit to the Github repository was five years ago. This means that before using these scripts, it has to be verified whether they still produce sound and complete traces for current versions of iOS. The scripts were verified using test data from iOS 12.

If verified properly, or executed on files originating from iOS 12, the scripts in this Github repository might be interesting for integration into other digital forensic platforms. If, after investigation, it turns out that they no longer produce sound and complete traces, it might be decided to use the forensic knowledge within them to build new parser for current versions of iOS, since it is likely that at least part of the forensic knowledge within the parsers is still relevant.

2.2.8. Apollo

Apollo extracts traces from databases present on iOS and MacOS devices. The platform essentially consists of many SQL queries that can be loaded and executed dynamically on the appropriate database. These SQL queries are specified in .txt files. The queries themselves contain a lot of valuable digital forensic information about the meaning of certain databases, which is not always obvious from the databases themselves. The GitHub repository contains 132 queries. The last commit to the repository was four years ago, but since the platform and version on which the query is supposed to be executed are included, it can probably be assumed that the forensic knowledge embedded within them is still relevant for application to files originating from those versions of the operating system. The forensic knowledge has to be verified before adapting the queries or the information within them for use on files originating from modern OS versions.

In conclusion, Apollo is an easy to use tool and the meaning of the results can easily be derived by examining the SQL queries. The fact that the OS versions for which the queries are intended makes it easy to determine whether the queries can be used on retrieved databases.

2.3. Conclusion

This chapter provided an overview of open-source digital forensic platforms, with a focus on their relevance for integration into Hansken. The selection criteria used for evaluating these platforms—number of supported formats, last update, modularity, and availability of test data—were identified in collaboration with Hansken developers to ensure alignment with practical forensic needs. This overview therefore answers research question 1, ”What open-source digital forensic platforms are available and how do they compare on certain relevant properties?”.

The analysis of various platforms revealed significant diversity in terms of supported file formats, modularity, and forensic documentation. Some platforms, such as PLASO and IPED, stand out due to

their large number of parsers, structured codebases, and regular updates, making them strong candidates for integration. Others, like Dissect, offer well-documented forensic tools designed for standalone execution, facilitating their use in other forensic systems. Platforms like Kuiper and iLEAPP/ALEAPP demonstrate structured and reusable parsers but may require additional effort for seamless integration.

Additionally, some platforms, such as Apollo and iOS_sysdiagnose_forensic_scripts, contain valuable forensic knowledge but require verification due to infrequent updates. Despite this, their structured forensic methodologies and documented outputs can still serve as useful resources for developing new or improved parsers.

Overall, this chapter highlights the strengths and limitations of different open-source forensic platforms and provides a structured comparison that can inform decisions regarding their potential integration into Hansken.

3

Uniformity Functions

The goal of this chapter is to answer research question 2, **"What is the best way to quantify uniformity of the output of open-source digital forensic platforms?"**
To answer this question, we present six types of uniformity, and for each of those types, we provide a set of functions that can be used to quantify that type of uniformity. We conclude with a sample computation, in which the introduced functions are applied to a small synthetic output of a fictitious digital forensic platform. In chapter 5, we discuss an experiment conducted in order to gather data that helps us answer research questions 2.1, **"How accurate is the uniformity metric developed in this thesis?"**. An answer to RQ 2.1 is presented in chapter 7, "Discussion".

3.1. General Concepts

We introduce some basic concepts that will be used throughout the following chapter, as well as in the rest of this thesis.

$concept$ A $concept$ is similar to a concept in the real world. In the digital forensic domain, concepts are, for instance, an email with all its properties. In functions that take a $concept$ as input, the type of concept is denoted in the function signature as $\mathcal{C}$.

$property$ A $property$ is a property of a $concept$. To continue with the example, a property of an email would be the sender of an email. In functions that take a $property$ as input, the type of property is denoted in the function signature as $\mathcal{P}$.

$type$ A $type$ can be thought of as a data type as known in programming. All values belonging to all $properties$ as described above are of some type.

Source of general knowledge The functions in this chapter require the possibility to determine whether concepts, properties, and syntactic structures are similar. To do this, we introduce the source of general knowledge. A source of general knowledge is any mechanism that can determine whether concepts, properties, and syntactic structures are similar. Given two things to compare, it should return either "true" or "false". Since things may be similar in one context but not in another, the source of general knowledge makes use of a source of problem-specific knowledge, which is explained in the next paragraph. In functions that require a source of general knowledge as input, it is denoted in the function signature as K_G.

Specifically for the function $abstractionLevel$, the source of general knowledge has to return an integer representing the level of abstraction of a concept.

Source of problem-specific knowledge Things can be similar in one context, but not in another. For instance, an email and a chat message can be similar in one context because they both are some form of digital communication, but they can be not similar in a different context because they are originate from different software. This context is captured in the form of problem-specific knowledge. The previously introduced source of general knowledge makes use of this problem-specific knowledge to make an accurate decision as to whether two things are similar. In functions that require a source of problem-specific knowledge as input, it is denoted in the function signature as K_P
The source of general knowledge and source of problem-specific knowledge are denoted in the signature of every function during the execution of which, a function is called that makes concrete use of the knowledge sources. For clarity, we generally omit passing the knowledge sources down to the similarity functions. The knowledge sources denoted when a similarity function is used.

3.2. High-Level Semantic

High-level semantic uniformity measures the degree to which the level of abstraction of concepts varies. In the running example, both concepts are email messages originating from a specific mail client, and it is valid to say that those concepts are therefore at the same level of abstraction. One is not more concrete than the other. If, for instance, the concept ”apple_email” was changed to ”digital communication”, both concepts would not be on the same level of abstraction, since ”digital communication” is more abstract than ”Outlook_mail_message”.

Intuitively, high-level semantic uniformity is high when all concepts are at the same level of abstraction. In other words, there is only 1 unique level of abstraction in that case. To help us determine the level of abstraction of a concept, we introduce the function $abstractionLevel$. The more unique levels of abstraction, the lower the uniformity.

$$abstractionLevel : C \times K_p \times K_g \mapsto \mathbb{N} \tag{3.1}$$

$$abstractionLevel(c, k_p, k_g) = \text{The level of abstraction of c according to } k_p \text{ and } k_g. \tag{3.2}$$

Now, $abstractionLevel$ can be used to count the number of unique levels of abstractions. For this, we introduce $countAbstractionLevels$.

$$countAbstractionLevels : \mathcal{P}(C) \times K_p \times K_g \mapsto \mathbb{N} \tag{3.3}$$

$$countAbstractionLevels(c, k_p, k_g) = \left|abstractionLevel(a, k_p, k_g) | a \in c\right| \tag{3.4}$$

With $countAbstractionLevels$, we can create a function to compute high-level semantic uniformity. It’s output decreases when there are more than 1 unique levels of abstraction.

$$uniformity_{HLsem} : \mathcal{P}(C) \times K_p \times K_g \mapsto \mathbb{R} \tag{3.5}$$

$$uniformity_{HLsem}(c, k_p, k_g) = \frac{1}{countAbstractionLevels(c, k_p, k_g)} \tag{3.6}$$

3.3. Low-Level Semantic

Low-level semantic uniformity measures the degree to which similar concepts also have similar properties. A lack of low-level semantic uniformity exists in our running example. It can be argued that both

concepts are similar, since they are both some type of email message. They do, however, not have the exact same set of properties, since ”apple_email” has the properties ”Sender” and ”Receiver”, that contain the names of the sender and receiver of the email; ”Outlook_mail_message” has no such properties. So, we note that although both concepts are similar, when comparing ”Outlook_mail_message” and ”apple_email”, there are two properties ”missing” from ”Outlook_mail_message”. This does not hold the other way around, since there are no properties of ”Outlook_mail_message” missing in ”Apple_email”. The observation that properties can be missing from similar concepts forms the base of the function to quantify low-level semantic uniformity, which we build towards in this section. This function, $uniformity_{LLsem}$, has the following signature:

$$uniformity_{LLsem} : \mathcal{P}(C) \times K_p \times K_g \mapsto \mathbb{R}$$

The first helper function we introduce is $propertiesOfConcept$, which takes a concept and returns the properties of that concept.

$$propertiesOfConcept : C \mapsto \mathcal{P}(P) \tag{3.7}$$

$$propertiesOfConcept(c) = \text{properties belonging to concept } c \tag{3.8}$$

Next, we introduce $conceptSim$, which can be used to determine whether two concepts are equal according to a source of general knowledge and a source of problem-specific knowledge.

$$conceptSim : C \times C \times K_p \times K_g \mapsto \mathbb{B} \tag{3.9}$$

$$conceptSim(p_1, p_2, k_p, k_g) = \begin{cases} True & \text{iff } c_1 \text{ and } c_2 \text{ are similar according to } k_p \text{ and } k_g \\ False & Otherwise \end{cases} \tag{3.10}$$

Now we introduce $groupSimilarConcepts$, which takes multiple concepts and returns a set of sets of concepts that are similar according to $conceptSim$.

$$groupSimilarConcepts : \mathcal{P}(C) \times K_p \times K_g \mapsto \mathcal{P}(\mathcal{P}(C)) \tag{3.11}$$

$$groupSimilarConcepts(c, k_p, k_g) = \{z \subseteq c | \forall a, b \in z, conceptSim(a, b, k_p, k_g) \wedge z \neq \emptyset\} \tag{3.12}$$

As mentioned above, the function for computing low-level semantic uniformity relies on counting how many properties are missing when are missing in one concept compared to another. To do this for a particular set of similar concepts, we need to count how many properties are missing from all concepts in the set compared to all other concepts in the set. To facilitate this pairwise comparison, we create a function that given a set of concepts, creates all possible tuples of concepts in that set. We do not want to compare concepts to themselves, so we need the concepts in the tuples to be not equal. We do want to compare concepts in both directions, so given two concepts c_1 and c_2, we want the the function to return (c_1, c_2) and (c_2, c_1). This leads us to the function $tuplesFromSet$:

$$tuplesFromSet : \mathcal{P}(C) \mapsto \mathcal{P}((C \times C)) \tag{3.13}$$

$$tuplesFromSet(c) = \{(c_1, c_2) | c_1 \in c \wedge c_2 \in c \wedge c_1 \neq c_2\} \tag{3.14}$$

To construct a function that counts missing properties from concepts, we first need a function that determines whether properties are similar. We call this function $propertySim$ and it is defined as follows:

$$propertySim : P \times P \times K_p \times K_g \mapsto \mathbb{B} \tag{3.15}$$

$$propertySim(p_1, p_2, K_p, K_g) = \begin{cases} True & \text{iff } p_1 \text{ and } p_2 \text{ are similar according to } K_g \text{ and } K_p \\ False & \text{otherwise} \end{cases} \tag{3.16}$$

Using this $propertySim$, we construct the function $conceptContainsProperty$. This function takes as input a concept and a property, and it returns 1 if and only if the concept contains a property similar to the input property, and 0 otherwise. It is defined as follows:

$$conceptContainsProperty : C \times P \times K_p \times K_g \mapsto \mathbb{N} \tag{3.17}$$

$$conceptContainsProperty(c, p, k_p, k_g) = \begin{cases} 0 & \text{iff } \exists q \in propertiesOfConcept(c) \wedge propertySim(p, q, k_p, k_g) \\ 1 & \text{otherwise} \end{cases} \tag{3.18}$$

Now that we have the ability to create tuples of concepts and to determine whether a property is missing from a concept, we can craft a function that takes a tuple of concepts and counts how many properties of the first concept are missing in the second concept. We call it $countMissingProperties$, and it is defined as follows:

$$countMissingProperties : (C, C) \times K_p \times K_g \mapsto \mathbb{N} \tag{3.19}$$

$$countMissingProperties((c_1, c_2), k_p, k_g) = \sum_{p \in propertiesOfConcept(c_1)} conceptContainsProperty(c_2, p, k_p, k_g) \tag{3.20}$$

As becomes clear from the definition, $countMissingProperties$ works because $conceptContainsProperty$ returns 1 if a property is missing. This allows us to use this relatively simple summation.

The next step is to use $countMissingProperties$ to count how many properties are missing in a set of tuples of similar concepts. This can be done using the function $countMissingPropertiesInSet$, which is defined as follows:

$$countMissingPropertiesInSet : \mathcal{P}((C, C)) \times K_p \times K_g \mapsto \mathbb{N} \tag{3.21}$$

$$countMissingPropertiesInSet(c, k_p, k_g) = \sum_{(c_1, c_2) \in c} countMissingProperties(c_1, c_2), k_p, k_g) \tag{3.22}$$

Next, we need a function that counts the total number of properties of the concepts present in a set of tuples of concepts. For this, we define the function $countPropertiesInTuples$:

$$countPropertiesInTuples : \mathcal{P}((C, C)) \mapsto \mathbb{N} \tag{3.23}$$

$$countPropertiesInTuples(c) = \sum_{(c_1, c_2) \in c} \left|POC(c_1)\right| + \left|POC(c_2)\right| \tag{3.24}$$

In the above definition, POC is an abbreviation of $propertiesOfConcept$.

We can now create the function $similarConceptsUniformity$, which computes the uniformity of a set of similar concepts.

$$similarConceptsUniformity : \mathcal{P}(C) \mapsto \mathbb{R} \tag{3.25}$$

$$similarConceptsUniformity(c) = 1 - \frac{countMissingPropertiesInSet(tuplesFromSet(c))}{countPropertiesInTuples(tuplesFromSet(c))} \tag{3.26}$$

Finally, we can create the function $uniformity_{LLsem}$, which computes the low-level semantic uniformity of a set of concepts.

$$uniformity_{LLsem} : \mathcal{P}(C) \times K_p \times K_g \mapsto \mathbb{R} \tag{3.27}$$

$$uniformity_{LLsem}(c) = \frac{\sum_{t \in groupSimilarConcepts(c)} similarConceptsUniformity(t)}{\left|groupSimilarConcepts(c)\right|} \tag{3.28}$$

3.4. High-Level Syntactic

High-level uniformity measures the degree to which similar concepts are represented syntactically in a similar way. For instance, in the running example, we see that both concepts are similar since they are both emails, but that they have a different name, namely Outlook_mail_message and apple_email. This is an example of a case with low high-level syntactic uniformity. Intuitively, we would have perfect high-level syntactic uniformity if both concepts in the running example would have the same name, or in other words, if there were one unique name. More generally speaking, high-level syntactic uniformity is high when similar concepts have the same name. We quantify high-level syntactic uniformity by creating sets of similar concepts, and counting the the uniquely occurring concept names. The more unique names there are, the lower the uniformity. We can use this notion to create the function $uniformity_{HLSyn}$, which quantifies the high-level syntactic uniformity given a set of concepts.

$$uniformity_{HLSyn} : \mathcal{P}(C) \times K_p \times K_g \mapsto \mathbb{R} \tag{3.29}$$

$$uniformity_{HLSyn}(c, k_p, k_g) = \frac{countSetsOfConcepts(c)}{countUniqueNames(c)} \tag{3.30}$$

This function is defined in terms of the functions $countSetsOfConcepts$ and $countUniqueNames$. The first function, $countSetsOfConcepts$, is used to count how many sets of similar concepts exist in some set of concepts. For instance, given the two concepts from our running example, it would return 1, since both concepts are emails and there thus exists one set of similar concepts. Were we to give $countSetsOfConcepts$ also a ”phone call” as input next to the emails, it would return 2, since emails and phone calls are not similar and therefore there are 2 sets of similar concepts. The function $countSetsOfConcepts$ makes use of the function $conceptSim$, which can be used to determine whether two concepts are similar.

$$countSetsOfConcepts : \mathcal{P}(C) \times K_p \times K_g \mapsto \mathbb{N} \tag{3.31}$$

$$countSetsOfConcepts(c, k_p, k_g) = \left|\{z \subseteq c | \forall a, b \in z, conceptSim(a, b, k_p, k_g) \wedge z \neq \emptyset\}\right| \tag{3.32}$$

The next part of $uniformity_{HLsym}$ is equation 3.33, $countUniqueNames$. This function takes a set of concepts as input and returns the number of uniquely occurring concept names in that set. It creates a set of all unique names and returns the cardinality of that set. In the case of the running example, $countUniqueNames$ would return 2, since the names Outlook_mail_message and apple_email are unique. The function $countUniqueNames$ makes use of the function $conceptName$, which takes a concept as input and returns the name of that concept.

$$countUniqueNames : \mathcal{P}(C) \mapsto \mathbb{N} \tag{3.33}$$

$$countUniqueNames(c) = \left|\{conceptName(c) | c \in C\}\right| \tag{3.34}$$

$$conceptName : C \mapsto string \tag{3.35}$$

$$conceptName(c) = \text{name of concept } c \tag{3.36}$$

3.5. Low-Level Syntactic

Low-level syntactic uniformity measures the degree to which similar properties are represented syntactically in a similar way. For instance, in the running example, we see that Outlook_mail_message has the property ”Sender”, and apple_email has the property ”From_email”. These properties are similar since they both represent the email address of the sender of an email. However, the name of the properties is different, which is an example of a lack of low-level syntactic uniformity. Intuitively, low-level syntactic uniformity would be perfect if similar concepts also had the same syntactic representation. In the case of the running example, this would mean that for both Outlook_mail_message and apple_email, the property containing the email address of the sender would have the same name. This can be rephrased as all properties in a set of similar properties having the same name, thus there being only 1 unique name. We can use this intuition to create a formula for quantifying low-level syntactic uniformity, as shown in equation 1.4.

Given a set s of properties, we first create subsets of s containing properties that are similar according to $property\_sim$. For each of those subsets, we can count the number of uniquely occurring property names. If that number is one, all properties in the subset have the same name, which indicates good low-level syntactic uniformity. By taking the average of this value for each subset of similar properties, we obtain a value for the low-level syntactic uniformity of all properties in the input.

To do this, we construct the function $similarPropertyNameUniformity$. This function takes as input a set of properties. If all properties in the set have the same syntactic representation, it returns 1. Its output gets closer to 0 the more unique syntactic representations the properties in the set have. We define it as follows:

$$similarPropertyNameUniformity : \mathcal{P}(P) \mapsto \mathbb{R} \tag{3.37}$$

$$similarPropertyNameUniformity(p, k_p, k_g) = \frac{1}{1 + (countUniquePropertyNames(p) - 1)} \tag{3.38}$$

The above definition makes use of $countUniquePropertyNames$, which counts the number of uniquely occurring property names within a set of properties. It in turn makes use of the function $propertyName$, which takes a property as input and returns the name of that property. Using $propertyName$, $countUniquePropertyNames$ simply constructs the set of all property names and returns the cardinality of that set.

$$countUniquePropertyNames : \mathcal{P}(P) \mapsto \mathbb{N} \tag{3.39}$$

$$countUniquePropertyNames(P) = \left|\{propertyName(p) | p \in P\}\right| \tag{3.40}$$

We need a function that takes a set of properties, and creates subsets of properties that are similar, according to $propertySim$. To do this, we introduce $groupSimilarProperties$:

$$groupSimilarProperties : \mathcal{P}(P) \times K_p \times K_g \mapsto \mathcal{P}(\mathcal{P}(P)) \tag{3.41}$$

$$groupSimilarProperties(p, k_p, k_g) = \{z \subseteq p | \forall a, b \in z, propertySim(a, b, k_p, k_g) \wedge z \neq \emptyset\} \tag{3.42}$$

We can now construct the function $uniformity_{LLSyn}$, which computes the low-level syntactic uniformity of a set of properties. It takes as input a set of properties which it divides into sets of similar properties. To each of those sets, the function $similarPropertyNameUniformity$ is applied, and the average of each of those applications is returned.

$$uniformity_{LLSyn} : \mathcal{P}(P) \times K_p \times K_g \mapsto \mathbb{R} \tag{3.43}$$

$$uniformity_{LLSyn}(p, k_p, k_g) = \frac{\sum_{s \in groupSimilarConcepts(p, k_p, k_g)} similarPropertyNameUniformity(s)}{\left|groupSimilarConcepts(p, k_p, k_g)\right|} \tag{3.44}$$

$$propertyName : P \mapsto \text{string} \tag{3.45}$$

$$propertyName(p) = \text{The name of property } p \tag{3.46}$$

3.6. Structural Syntactic

Structural syntactic uniformity is a measurement for the degree to which values of the same types, have a similar syntactic structure. Note that this type of uniformity considers the most concrete things, namely values, whereas the low-level and high-level uniformity deal with properties and concepts. An example of a lack of structural syntactic uniformity can be found in the running example. The values in the property ”Send_time” in ”Outlook_mail_message” are Unix timestamps, whereas those in ”Send_time” in ”apple_email” are in a human-readable format. Since both these values are a timestamp, the values are of the same type, namely timestamp, but their syntactic structure is different, which means there is no structural syntactic uniformity. The function that computes structural syntactic uniformity takes as input not a set of values, but a set of properties. This results in more readable functions. To bridge the gap between properties and values within properties, we introduce the function $propType$.

$$propType : P \mapsto Type \tag{3.47}$$

$$propType(p) = \text{Type of the values in property } p \tag{3.48}$$

We also need the ability to determine whether types are similar, so we introduce the function $typeSim$, which can do exactly that. It bases its decision on a source of general knowledge and a source of problem-specific knowledge.

$$typeSim : T \times T \times K_p \times K_g \mapsto \mathbb{B} \tag{3.49}$$

$$typeSim(t_1, t_2, k_p, k_g) = \begin{cases} True & \text{iff } t_1 \text{ and } t_2 \text{ are similar according to } k_p \text{ and } k_g \\ False & \text{otherwise} \end{cases} \tag{3.50}$$

We also need the ability to determine whether the syntactic structure of values of properties are similar or not according to a source of general knowledge and a source of problem-specific knowledge. This function, for instance, would be able to tell us that a human-readable timestamp and a Unix timestamp have a different syntactic structure. We call it $structureSim$, and it is defined as follows:

$$structureSim : P \times P \times K_p \times K_g \mapsto \mathbb{B} \tag{3.51}$$

$$structureSim(p_1, p_2, k_p, k_g) = \begin{cases} True & \text{iff the syntactic structures of } p_1 \text{ and } p_2 \text{ are similar according to } k_p \text{ and } k_g \\ False & \text{otherwise} \end{cases} \tag{3.52}$$

Next, we construct a function that takes a set of properties as input, and returns the number of unique syntactic structures within that set. In the case of the set containing the two ”Send_time” properties, this function would return 2, because the syntactic structures of the values within the properties are not the same (Unix and human-readable) according to the previously introduced $structureSim$. We call this function $countStructuresPerSet$.

$$countStructuresPerSet : \mathcal{P} \mapsto \mathbb{N} \tag{3.53}$$

$$countStructuresPerSet(p) = \left|\{z \subseteq p | \forall a, b \in z, structureSim(a, b)\}\right| \tag{3.54}$$

The next step is to apply the function $countStructuresPerSet$ to sets of properties that have the same type. This is done using the function $countStructuresPerType$. It takes as input a set of properties, creates sets of similarly typed properties according to $typeSim$, applies $countStructuresPerSet$ to each of those sets, and returns the set of the results.

$$countStructuresPerType : \mathcal{P}(P) \mapsto \mathcal{P}(\mathbb{N}) \tag{3.55}$$

$$countStructuresPerType(p) = \{countStructuresPerSet(z) | z \subseteq p \wedge \forall a, b \in z, typeSim(a, b) \wedge z \neq \emptyset\} \tag{3.56}$$

Now that we have $countStructuresPerType$, we can construct the function that computes structural syntactic uniformity. This function is based on the intuition that one unique syntactic structure per set of properties of the same type means high structural syntactic uniformity. If there are more than one unique structures, the uniformity decreases. We compute the structural uniformity for each set of properties of similar types computing 1 over the number of unique syntactic structures, and taking the average. This results in the following formula.

$$uniformity_{StructSyn} : \mathcal{P}(P) \mapsto \mathbb{R} \tag{3.57}$$

$$uniformity_{StructSyn}(p) = \frac{\sum_{x \in countStructuresPerType(p)} \frac{1}{x}}{\left|countStructuresPerType(p)\right|} \tag{3.58}$$

3.7. Structural Semantic

Structural semantic uniformity is about the semantics that are embedded in syntactic structures. It aims to compare whether the same amount of information is embedded within different structures that both represent the same type of data. Take, for example, the columns ”Send_time” in the tables ”Outlook_mail_message” and ”apple_email” from the running example. The value in ”Send_time” in ”Outlook_mail_message” is a Unix timestamp, whereas that in ”apple_mail” is in a human-readable format. The human readable format contains time zone information, which the Unix timestamp does not have. This means that the syntactic structures hold different amounts of information. To quantify structural semantic uniformity, we first need the ability to determine whether the types of the values belonging to two properties are equal. For this, we can reuse the function $typeSim$, which was introduced earlier.

We also need to be able to compare whether syntactic structures of values belonging to properties contain the same information. To do this, we introduce the function $embeddedSim$:

$$embeddedSim : P \times P \times K_g \times K_p \mapsto \mathbb{B} \tag{3.59}$$

$$embeddedSim(p_1, p_2, K_g, K_p) = \begin{cases} True & \text{iff the syntactic structures of the values in } p_1 \text{ and } p_2 \\ & \text{contain the same information} \\ False & \text{otherwise} \end{cases} \tag{3.60}$$

Next, we introduce the function $countEmbeddingsPerSet$. This function takes as input a set of properties, and it returns the number of sets of embedded information. For example, when Outlook_mail_message property Send_time and the apple_email property Send_time are used as input to $countEmbeddingsPerSet$, it would return 2, since the syntactic structure of the Apple_email Send_time contains a day, month, and timezone in human readable format. To construct $countEmbeddingsPerSet$, we make use of $embeddedSim$. It looks as follows:

$$countEmbeddingsPerSet : \mathcal{P}(P) \mapsto \mathbb{N} \tag{3.61}$$

$$countEmbeddingsPerSet(g) = \left|\{z \subseteq g | \forall a, b \in z, embeddedSim(a, b)\}\right| \tag{3.62}$$

The next step is to apply $countEmbeddingsPerSet$ to sets of properties that have the same general type, such as, a set of properties that all contain timestamps. To do this, we construct the function $countSimilarEmbeddingsPerType$. This function takes as input a set of properties. It then proceeds to create sets of properties having same general type, such as timestamp, and then computing how many different embeddings there are in each of those sets. In the definition, we abbreviate $countEmbeddingsPerSet$ as $CEPS$. The function is defined as follows:

$$countSimilarEmbeddingsPerType : \mathcal{P}(P) \mapsto \mathcal{P}(\mathcal{N}) \tag{3.63}$$

$$countSimilarEmbeddingsPerType(p) = \{CEPS(z) | z \subseteq p \wedge \forall a, b \in z, typeSim(a, b) \wedge z \neq \emptyset\} \tag{3.64}$$

We can now use $countSimilarEmbeddingsPerType$ to construct the formula used to compute structure semantic uniformity. This formula, $uniformity_{StructSem}$, is based on the intuition that if a set of similarly typed properties contains different amounts of information embedded in the syntactic structure, the uniformity is lower. The formula is defined as follows:

$$uniformity_{StructSem} : \mathcal{P}(P) \mapsto \mathcal{R} \tag{3.65}$$

$$uniformity_{StructSem}(p) = \frac{\sum_{x \in CSEPT(p)} \frac{1}{x}}{\left|CSEPT(p)\right|} \tag{3.66}$$

3.8. Sample Computation

In this section, we present an example of the usage of the formulas introduced above. For this example, we use an artificial output of some digital forensic platform.

Recall that in many of the functions introduced in the previous subsections, an external source of general knowledge and an external source of problem-specific knowledge were required. In the rest of this section, the author serves as both of those sources of knowledge in all functions that require it.

Table 3.1: Table with data representing the concept ”chat”, containing data on chat messages

Message ID	Chat ID	Sender	Receiver	Send Time	Message Content	Read status
a3bb189e-8bf9-3888-9912-ace4e6543002	98765432101234567 89	+1-202-555-0134	+61-2-9876-5432	August 13, 2024 09:15:30 AM EDT	Hey, can we catch up later?	true
4e8f4c26-cb7f-4d15-9a4c-6e29b5c3e3d2	12345678909876543 21	+1-305-555-0199	+81-3-1234-5678	September 25, 2024 02:45:00 PM PDT	Got it! I’ll send the report soon.	false
e12f11f4-3d64-4b6f-8f65-dad1f249d012	45678901234567890 12	+44-20-7946-0958	+34-91-123-4567	October 8, 2024 06:30:15 PM CST	Thanks for the update!	true
5a2d8e2c-8b79-4f85-8e26-91cbed6a2397	78901234567890123 45	+33-1-70-18-99-00	+55-11-98765-4321	November 20, 2024 10:00:00 PM MST	Are we still on for lunch?	false
d8a6d8f2-53b4-46c4-b2ec-bf616e6a8b24	11223344556677889 99	+49-30-12345678	+27-21-123-4567	December 31, 2024 11:59:59 PM EST	Just a quick reminder about the meeting.	true

Table 3.2: Table with data representing the concept ”email”, containing data on emails

Sender	Receiver	From_email	To_email	Send_time	Subject	Content
Andrew Baker	Grace Wood	andrew.baker@example.com	grace.wood@example.com	1691894400	Action Required: Update Your Profile	Dear Grace, Please take a moment to update your profile information in our system. It’s important to keep your details current. Best regards, HR Team
Chloe Davis	Joshua Clark	chloe.davis@example.com	joshua.clark@example.com	1691980800	Team Outing Scheduled for Next Month	Hi Joshua, We’re excited to announce a team outing next month! Stay tuned for more details. Cheers, Organizing Committee
Ryan Mitchell	Mia Wright	ryan.mitchell@example.com	mia.wright@example.com	1692067200	Proposal Review Meeting Tomorrow	Hello Mia, Just a reminder that we have a proposal review meeting scheduled for tomorrow at 10 AM. Please come prepared. Best, Project Manager
Lucy Evans	Nathan King	lucy.evans@example.com	nathan.king@example.com	1692153600	Congratulations on Your Promotion!	Hi Nathan, Congratulations on your well-deserved promotion! We look forward to seeing you excel in your new role. Best wishes, Your Team
Kevin Harris	Sophie Hall	kevin.harris@example.com	sophie.hall@example.com	1692240000	Urgent: Response Needed by EOD	Dear Sophie, This is an urgent request. Please respond to the attached email by the end of the day. Thank you, Kevin

3.8.1. High-level Semantic Example

We start by computing the high-level semantic uniformity of the sample output introduced earlier. Recall that the high-level semantic uniformity of a digital forensic tool is high when the concepts in the output are all at the same level of abstraction. We can compute it using equation 1.6, which makes use of equation 1.5, $countAbstractionLevels$ to determine the number of unique levels of abstraction of the concepts in the input. First, we use $abstractionLevel$ to determine the level of abstraction of each concept in the input. Recall that this function uses a source of general knowledge and a source of problem-specific knowledge, both of which in this case are the author.

$$abstractionLevel(chat, k_p, k_g) = 1$$

$$abstractionLevel(email, k_p, k_g) = 1$$

$$abstractionLevel(outlook\_mail\_message, k_p, k_g) = 2$$

As seen above, ”chat” and ”email” share the same abstraction level, while ”outlook_mail_message” is at a lower level. Next, applying $countAbstractionLevels$ to the set of concepts yields:

$$countAbstractionLevels(\{chat, mail, outlook\_mail\_message\}, k_p, k_g) = 2$$

Finally, using this result in $uniformity_{HLsem}$, we compute the high-level semantic uniformity:

$$uniformity_{HLsem} = \frac{1}{2} = \frac{1}{2}$$

So, the high-level semantic uniformity of the sample data is $\frac{1}{2}$.

3.8.2. Low-level Semantic Example

We now compute the low-level semantic uniformity of the sample data. Recall that this type of uniformity measures whether similar concepts also have the same properties. In the case of the sample data, one might expect the concepts ”mail” and ”outlook_mail_message” (OMM) to have similar properties, since the concepts both represent some type of email message.

We begin by applying the function $groupSimilarConcepts$ to the set of concepts in the sample output. The function groups the concepts by similarity based on $conceptSim$. Applying this function to all possible pairs of concepts yields the following:

Table 3.3: Table with data representing the concept ”outlook_mail_message”, containing data on emails originating from Outlook

Sender	Receiver	Send_time	Subject	Content
johndoe@example.com	sarah.wilson@example.com	2024-08-13T00:00:00.000	Meeting Reminder: Project Update	Hi Sarah, Please find the latest project update attached. Let me know if you have any questions. Best, John
janedoe@example.com	david.taylor@example.com	2024-08-14T00:00:00.000	Invoice #10234 Attached	Dear David, Your invoice for this month is attached. Please review and let me know if there are any discrepancies. Thank you, Accounts Team
alex.smith@example.com	olivia.martin@example.com	2024-08-15T00:00:00.000	Upcoming Webinar: How to Boost Productivity	Hello, Don’t forget to register for our upcoming webinar on productivity hacks. It’s going to be a great session! Cheers, Webinar Team
emily.jones@example.com	daniel.thomas@example.com	2024-08-16T00:00:00.000	Team Lunch This Friday	Hi Daniel, Let’s gather for a team lunch this Friday at 12 PM. Please RSVP so we can make a reservation. Best, Emialy
michael.brown@example.com	emma.lee@example.com	2024-08-17T00:00:00.000	Feedback on the Latest Design	Hi Emma, I’ve reviewed the latest design and have a few suggestions. Let’s discuss in our next meeting. Best regards, Emily

$$conceptSim(chat, mail, k_p, k_g) = False$$
$$conceptSim(chat, OMM, k_p, k_g) = False$$
$$conceptSim(mail, OMM, k_p, k_g) = True$$

Applying $groupSimilarConcepts$ therefore looks as follows:

$$groupSimilarConcepts\Big(\{chat, mail, OMM\}, k_p, k_g\Big) = \Big\{\{mail, OMM\}\ \{chat\}\Big\}$$

The next step is to determine whether similar concepts have similar properties. To do this, we pairwise compare all similar concepts with each other and determine whether each of the properties of the first concept is present in the second concept. To create the pairs of concepts to compare, we use the function $tuplesFromSet$, as shown in equation 1.13. We apply it to each set in the output of $groupSimilarConcepts$. Since there are no concepts similar to "chat", it has the same properties as the concepts similar to it, vacuously holds. We can exclude it for the rest of the computation.

$$tuplesFromSet\Big(\{mail, OMM\}, k_p, k_g\Big) = \Big\{(mail, OMM),\ \ (OMM, mail)\Big\}$$

We can now apply the function $countMissingProperties$ each of those tuples, which counts the properties that the first concept has, but the second one does not. It uses $conceptContainsProperty$ to determine whether a particular concept has a particular property. Recall that it returns 1 if the property is present and 0 otherwise. For the sake of brevity, all applications of the function $conceptContainsProperty$ are explicitly stated. The function $countMissingProperties$ applied to the tuple $(mail, OMM)$ yields the following:

$$\begin{aligned} countMissingProperties(mail, OMM, k_p, k_g) = {} & conceptContainsProperty(Sender, OMM, k_p, k_g) \\ & + conceptContainsProperty(Receiver, OMM, k_p, k_g) \\ & + conceptContainsProperty(From\_email, OMM, k_p, k_g) \\ & + conceptContainsProperty(To\_email, OMM, k_p, k_g) \\ & + conceptContainsProperty(Send\_time, OMM, k_p, k_g) \\ & + conceptContainsProperty(Subject, OMM, k_p, k_g) \\ & + conceptContainsProperty(Content, OMM, k_p, k_g) \\ & = 2 \end{aligned} \tag{3.67}$$

In the case of the properties $Sender$ and $Receiver$, the function $conceptContainsProperty$ returned 1 because the concept OMM has properties containing the name of the sender and receiver of an email message.

Applying $countMissingProperties$ to the tuple $(OMM, email)$ yields the following:

$$\begin{aligned} countMissingProperties(OMM, email, k_p, k_g) = {} & conceptContainsProperty(Sender, OMM, k_p, k_g) \\ & + conceptContainsProperty(Receiver, email, k_p, k_g) \\ & + conceptContainsProperty(Send\_time, email, k_p, k_g) \\ & + conceptContainsProperty(Subject, email, k_p, k_g) \\ & + conceptContainsProperty(Content, email, k_p, k_g) \\ & = 0 \end{aligned} \tag{3.68}$$

We can now use the function *countMissingPropertiesInSet* to count the total number of missing properties in all tuples returned by *tuplesFromSet*.

$$\begin{aligned} countMissingProperties((email, OMM), (OMM, email)) = \\ countMissingProperties(mail, OMM, k_p, k_g) + \\ countMissingProperties(OMM, email, k_p, k_g) + \\ = 2 \end{aligned} \tag{3.69}$$

Next, we determine the total number of properties of the concepts in the tuples returned by *tuplesFromSet* using *countPropertiesInTuples* (*CPIT*):

$$\begin{aligned} CPIT(\{(email, OMM), (OMM, email)\}) = & 7 + 5 \\ & 5 + 7 = 26 \end{aligned} \tag{3.70}$$

We can now apply the function *similarConceptsUniformity* (*CSU*) to the set $\{mail, OMM\}$ as previously returned by *groupSimilarConcepts*. This looks as follows:

$$SCU(\{email, OMM\}) = 1 - \frac{2}{26} \approx 0,92$$

Finally, we compute the low-level semantic uniformity of the sample data:

$$uniformity_{LLsem}(\{email, OMM\}) = \frac{0,92}{1} = 0,92$$

3.8.3. High-level Syntactic Example

Recall that high-level syntactic uniformity is high when similar concepts have the same name, and it is computed using the following function:

$$uniformity_{HLSyn}(c, k_p, k_g) = \frac{countSetsOfConcepts(c)}{countUniqueNames(c)}$$

The function *countSetsOfConcepts* takes as input a set of concepts and returns the number of sets of similar concepts that exist in it. When applied to the sample data, it returns the following:

$$countSetsOfConcepts(\{outlook\_mail\_message, mail, chat\}) = \left|\{\{outlook\_mail\_message, mail\}, \{chat\}\}\right| = 2$$

The next step is to compute the value of *countUniqueNames* on the sample data:

$$countUniqueNames(\{outlook\_mail\_message, mail, chat\}) = \left|\{\{outlook\_mail\_message\}, \{mail\}, \{chat\}\}\right| = 3$$

We can now compute the high-level syntactic uniformity of our sample data using $uniformity_{HLSyn}$:

$$uniformity_{HLSyn}(\{chat, email, outlook\_mail\_message\}) = \frac{2}{3} = \frac{2}{3}$$

3.8.4. Low-level Syntactic Example

Next, we compute the low-level syntactic uniformity of our sample data. Recall that low-level syntactic uniformity is about similar properties having the same name, and it is computed using the following function:

$$uniformity_{LLSyn}(p, k_p, k_g) = \frac{\sum_{s \in groupSimilarProperties(p,k_p,k_g)} similarPropertyNameUniformity(s)}{|groupSimilarProperties(p, k_p, k_g)|}$$

In this example, p is the set of all properties in our sample data. We start by computing the value of *groupSimilarProperties* with those properties as input. We denote the properties as the name of the property, with the concept from which it originates in subscript. The concept *outlook_mail_message* is abbreviated as *OMM*, and for brevity, not all applications of *propertySim* are explicitly stated.

$$\begin{aligned}
&groupSimilarProperties\Big(\{Message\ ID_{chat}, Chat\ ID_{chat}, Sender_{chat}, Receiver_{chat}, Send\ Time_{chat}\\
&\quad Message\ Content_{chat}, Read\ Status_{chat}, Sender_{email}, Receiver_{email},\\
&\quad Receiver_{email}, From_email_{email}, To_email_{email}, Send_time_{email}\\
&\quad Subject_{email}, Content_{email}, Sender_{OMM},\\
&\quad Receiver_{OMM}, Send_time_{OMM}, Subject_{OMM}, Content_{OMM}\}\Big)\\
&= \big\{\{Message\ ID_{chat}\}, \{Chat\ ID_{chat}\},\\
&\quad \{Sender_{chat}\}, \{Receiver_{chat}\},\\
&\quad \{Send\ Time_{chat}, Send_time_{email}, Send_time_{OMM}\},\\
&\quad \{Message\ Content_{chat}, Content_{email}, Content_{OMM}\}\\
&\quad \{Read\ status_{chat}\}\\
&\quad \{From_email_{email}, Sender_{OMM}\}\\
&\quad \{To_email_{email}, Receiver_{OMM}\}\\
&\quad \{Sender_{email},\}\\
&\quad \{Receiver_{email},\}\\
&\quad \{Subject_{email}, Subject_{OMM}\}\big\}
\end{aligned}$$

We proceed with computing the nominator in the definition of $uniformity_{LLSyn}$,

$$\sum_{s \in groupSimilarProperties(p,k_p,k_g)} similarPropertyNameUniformity(s)$$

$$\begin{aligned}
&similarPropertyNameUniformity(Message\ ID_{chat}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Chat\ ID_{chat}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Sender_{chat}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Receiver_{chat}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Send\ Time_{chat}, Send_time_{email}, Send_time_{OMM}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Message\ Content_{chat}, Content_{email}, Content_{OMM}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Read\ status_{chat}, k_p, k_g)\\
&+ similarPropertyNameUniformity(From_email_{email}, Sender_{OMM}, k_p, k_g)\\
&+ similarPropertyNameUniformity(To_email_{email}, Receiver_{OMM}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Sender_{email}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Receiver_{email}, k_p, k_g)\\
&+ similarPropertyNameUniformity(Subject_{email}, Subject_{OMM}, k_p, k_g)\\
&= 1 + 1 + 1 + 1 + \frac{1}{2} + \frac{1}{2} + 1 + \frac{1}{2} + \frac{1}{2} + 1 + 1 + 1 = 10
\end{aligned}$$

We now have everything we need to compute the low-level syntactic uniformity of the sample data:

$$uniformity_{LLSyn}(p, k_p, k_g) = \frac{10}{12}$$

3.8.5. Structural Syntactic Example

Next, we compute the structural syntactic uniformity of our sample data. Recall that structural syntactic uniformity is high when values in properties of the same type have a similar syntactic structure. An example of this is two timestamps, both denoted as a Unix timestamp. It is computed using the following function:

$$uniformity_{StructSyn}(p) = \frac{\sum_{x \in countStructuresPerType(p)} \frac{1}{x}}{|countStructuresPerType(p)|}$$

We start by applying $countStructuresPerType$ to the sample data. We abbrevaite $outlook\_mail\_message$ as OMM.

$$\begin{aligned}
&countStructuresPerType\Big(\{Message\ ID_{chat}, Chat\ ID_{chat}, Sender_{chat}, Receiver_{chat},\\
&\quad Send\ Time_{chat}, Message\ Content_{chat}, Read\ Status_{chat}, Sender_{email}\\
&\quad Receiver_{email}, From_email_{email}, To_email_{email}\\
&\quad Send_time_{email}, Subject_{email}, Content_{email},\\
&\quad Sender_{OMM}, Receiver_{OMM}, Send_time_{OMM}, Subject_{OMM}, Content_{OMM}\}\Big)\\
&= \{countStructuresPerSet(\{Message\ ID_{chat}\}),\\
&\quad countStructuresPerSet(\{Chat\ ID_{chat}\}),\\
&\quad countStructuresPerSet(\{Sender_{chat}, Receiver_{chat}\}),\\
&\quad countStructuresPerSet(\{Send\ Time_{chat}, Send_time_{email},\\
&\qquad Send_time_{OMM}\})\\
&\quad countStructuresPerSet(\{Message\ Content_{chat}, Content_{email},\\
&\qquad Content_{OMM}, Subject_{email}, Subject_{OMM}\}),\\
&\quad countStructuresPerSet(\{Read\ status_{chat}\}),\\
&\quad countStructuresPerSet(\{Sender_{chat}, Receiver_{chat}\}),\\
&\quad countStructuresPerSet(\{From_email_{email}, To_email_{email})\\
&\qquad Sender_{OMM}, Receiver_{OMM}\})\}\\
&= \{1, 1, 1, 3, 1, 1, 1, 1\}
\end{aligned}$$

In the above calculation, we can see that there is only a single set containing properties with values have the same type as well as different syntactic structures, namely $\{Send\ Time_{chat}, Send\_time_{email}, Send\_time_{OMM}\}$. The values of property $Send\ Time_{chat}$ are in a human-readable format, the values in $Send\_time_{email}$ are Unix timestamps, and those in $Send\_time_{OMM}$ are formatted according to ISO8601. This has a negative impact on the structural syntactic uniformity.

We proceed with using the output of the above computation in the function $uniformity_{StructSyn}$, where we denote the set of all properties in the sample data as p:

$$uniformity_{StructSyn(p)} = \frac{\frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{3} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} +}{8} \approx 0.92$$

So, the structural syntactic uniformity of our sample data is approximately 0.92.

3.8.6. Structural Semantic Example

We proceed with the last sample computation, namely that of the structural semantic uniformity of our sample data. The structural semantic uniformity of our sample data is high when values of the same type have syntactic structures that allow the embedding of the same information.

We start by computing the value of $countSimilarEmbeddingsPerType$ applied to the sample data. This function creates sets of properties that have values of the same type, and for each of those sets, it counts how many unique information embeddings are present in that set.

$$
\begin{aligned}
&countSimilarEmbeddingsPerType\Big(\{Message\ ID_{chat}, Chat\ ID_{chat}, Sender_{chat}, Receiver_{chat},\\
&\quad Send\ Time_{chat}, Message\ Content_{chat}, Read\ Status_{chat}, Sender_{email}\\
&\quad Receiver_{email}, From\_email_{email}, To\_email_{email}\\
&\quad Send\_time_{email}, Subject_{email}, Content_{email},\\
&\quad Sender_{OMM}, Receiver_{OMM}, Send\_time_{OMM}, Subject_{OMM}, Content_{OMM}\}\Big)\\
&= \{countEmbeddingsPerSet(\{Message\ ID_{chat}\}),\\
&\quad countEmbeddingsPerSet(\{Chat\ ID_{chat}\}),\\
&\quad countEmbeddingsPerSet(\{Sender_{chat}, Receiver_{chat}\}),\\
&\quad countEmbeddingsPerSet(\{Send\ Time_{chat}, Send\_time_{email}, Send\_time_{OMM}\}),\\
&\quad countEmbeddingsPerSet(\{Message\ Content_{chat}, Content_{email}, Content_{OMM},\\
&\qquad Subject_{email}, Subject_{OMM}\}),\\
&\quad countEmbeddingsPerSet(\{Read\ status_{chat}\}),\\
&\quad countEmbeddingsPerSet(\{Sender_{chat}, Receiver_{chat}\}),\\
&\quad countEmbeddingsPerSet(\{From\_email_{email}, To\_email_{email},\\
&\qquad Sender_{OMM}, Receiver_{OMM}\})\\
&\}\\
&= \{1, 1, 1, 2, 1, 1, 1, 1\}
\end{aligned}
\tag{3.71}
$$

In the above calculation, we can see that there is only a single set of properties of which the syntactic structures of its values are capable of embedding different amount of information, namely the set $\{Send\ Time_{chat},\ Send\_time_{email},\ Send\_time_{OMM}\}$. The difference lies in the fact the syntactic structure of the values stored in $Send\ Time_{chat}$ contains an explicit time zone, whereas the syntactic structures of the values in the other properties do not.

Using the result of the above computation, we can compute the value of $uniformity_{StructSem}$ on the sample data:

$$
uniformity_{StructSyn(p)} = \frac{\frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} +}{8} \approx 0.94
$$

3.9. Conclusion

In this chapter, we presented six ways in which output of a digital forensic tool can manifest a lack of uniformity, namely high-level semantic, low-level semantic, structural semantic, high-level syntactic, low-level syntactic, and structural syntactic. For each of those types of uniformity, we presented a function to quantify it, and we have shown how to apply those functions in an example. We have also discussed the need that these functions have for sources of problem-specific knowledge and general knowledge. These functions together serve as a way to quantify the uniformity of the output of open-

source digital forensic tools, and therefore this chapter answers research question 2, **”What is the best way to quantify uniformity of the output of open-source digital forensic platforms?”**.

4

Proof-of-Concept

In this chapter, we answer research question 3 by creating a proof of concepts that implements the uniformity functions introduced in chapter 2. Recall that the uniformity functions often make use of a source of general knowledge and a source of problem-specific knowledge. Because we need concrete instances of both these knowledge sources in an implementation of the uniformity functions, we start by outlining several potential sources of general knowledge and problem-specific knowledge. We compare advantages and disadvantages and choose the most suitable knowledge sources to be used in the proof-of-concept. Next, we discuss the implementation, architecture, and workflow of the proof-of-concept.

4.1. Knowledge Sources

In this section, we go over the various options that can serve as a source of general knowledge, problem-specific knowledge, or both. We first discuss properties that these sources should adhere to in order to be a suitable option for use in an implementation of the functions introduced in previous chapter. Next, we discuss various options and argue whether they do or do not adhere to these properties, and we conclude with choosing the most suitable sources of knowledge for our usecase.

4.1.1. Requirements

Firstly, we want the knowledge sources to be usable for never-before-seen words. This means that if the input data contains, for example, a column name that has never before occurred in the world, the source of knowledge can still make meaningful decisions about it. In other words, we want the source of general knowledge to be complete.
Secondly, we want the source of general knowledge to be sound. High soundness results in more accurate quantification of uniformity.
Thirdly, we want the knowledge sources to be able to answer multiple types of queries. This would allow for the use of a single source of domain knowledge for multiple functions, rather than needing a separate one for each of them. This would mean, for example, having a source of general knowledge that determine similarity of high-level concepts, as well as similarity of properties.

In the remainder of this section, we discuss various potential candidates for serving as a source of general knowledge.

4.1.2. Thesaurus

A thesaurus maps words to their synonyms, and possibly other relations such as broader or narrower terms and antonyms. Thesauri are commonly available for natural language in many languages and some are purpose built for domain-specific applications.

In the case of our running example, the high-level concepts ”Outlook_mail_message” and ”apple_email” can be deemed similar by a thesaurus if it contains an entry that maps these words to each other. Conversely, if a thesaurus does not contain an entry for those words, it can not map them to each other. The fact that a thesaurus can only operate on a fixed domain means that all terms for which it is queried have to be known at the time the thesaurus is created. This in turn means that applying the thesaurus to new data might not work because this data contains terms unknown at the time the thesaurus was created.
An advantage of thesauri is that they can serve as a source of general knowledge and problem-specific knowledge simultaneously because problem-specific knowledge is implicitly integrated in the general knowledge when creating a thesaurus.

4.1.3. Ontology

An ontology describes objects and the relationships between them. An ontology can contain various kinds of relations, such as hierarchical relationships that contain relationships between narrower and broader terms, or relationships that specify that something is an instance of something else, or a synonym. Despite these capabilities, an ontology shares limitations with thesauri, mainly the fixed domain and the inherent inability to deal with never before seen words as input. Therefore, ontologies are not a valid option for use as an external source of domain knowledge.

4.1.4. D4

The authors of [13] describe D4, a method for domain discovery. The aim of D4 is grouping columns that are likely to draw their values from the same domain. Assuming that columns that draw their values from the same domain have similar semantics, an attempt could be made to use D4 as a source of general knowledge.

D4 relies heavily on the co-occurrence of values across columns to determine whether they draw their values from the same domain. In experiments done with output of digital-forensic tools, D4 correctly identifies the domain of WiFi network names, but not that of timestamps. This seems to be due to the fact that several overlapping values exist between multiple columns containing WiFi network names, but none between columns containing timestamps. Although this is inherent to the workings of D4, it is counter-intuitive; WiFi network names words that are often not even present in natural language and that are close to meaningless without context, but timestamps that are all formatted exactly the same are missed.

It should be noted that discovering whether columns draw their values from the same domain is by itself inadequate for properly determining whether columns have similar semantics; columns that contain values from the same domain can have very different semantics, as is the case in, for example, the ”Sender” and ”Receiver” columns from the ”Outlook_mail_message” table from the running example.
In conclusion, D4’s heavy reliance on value co-occurrence makes it unsuitable as a source of general knowledge. knowledge.

4.1.5. Word Embeddings

Word embeddings map words to a vector space, where proximity indicates similarity. This approach could be used to determine semantic similarity between column and table names. Traditional word embedding techniques, however, struggle with out-of-vocabulary words. This is due to the nature of these methods; if a word is not in the vocabulary during training, the method can not map it to a vector. Therefore, similar to approaches discussed earlier, word embeddings are not suitable as a source of general knowledge due to their inability to deal with out-of-vocabulary words. A correctly trained model could have the advantage that it can serve as a source of general knowledge and problem-specific knowledge simultaniously, since the desired level of abstraction can be embedded in the training data.

4.1.6. Thesaurus of Regular Expressions

Regular expressions can be used to compare the structures of values. The use of regular expressions does come with some inherent potential flaws. For one, regular expressions can be either too strict, or not strict enough. For example, when crafting a regular expressions that matches a set of timestamps in a human readable format, such as 01-05-2024T10:51:15UTC, it is possible to create a regular expression that matches exactly the set of the provided timestamps, or create a regular expression that matches all values in the universe. Crafting the correct regular expression therefore requires knowledge about the meaning of the values to know what parts of the values are variable or fixed, whether values are optional, or whether they can vary in length.

Regular expressions can be used to compare the syntactic structure of values. One way in which this can be done is by creating a thesaurus of regular expressions. In such a thesaurus, regular expressions that match values of a particular type will map to regular expressions that match values of the same type, but of a different syntactic structure. For example, the thesaurus can contain a mapping from a regular expression that matches the values in ”Send_time” of ”Outlook_mail_message” in the running example, to a regular expression that matches the human-readable timestamp format present in the ”Send_time” in ”apple_email”. The thesaurus could also be implemented such that it maps regular expressions based on the embedded semantics of the values. In that case, the ”Send_time” values from the running example would not be mapped to each other because they do not contain the exact same amount of information.

Due to the extensive domain knowledge that this thesaurus contains, it can not be created in an automated way, but only by humans with problem-specific knowledge. It can therefore serve as both general knowledge and problem-specific knowledge for $struct\_sim$ and $embedded\_sim$. An advantage of the use of the regular expression thesaurus is that it can be of high soundness, depending on the care with which it was constructed. This quality can represent itself both in the quality of the regular expressions, i.e. they match exactly the desired set of values, as well as it containing the knowledge that values with a particular structure do or do not contain the same information.

A disadvantage is the inherent lack of flexibility of such a thesaurus. Since it functions as a mapping, any values that are not matched by a regular expression in it will not be recognised by it. If it can be assumed that the data the thesaurus will be applied to always contains values of a similar nature, this problem would not exist but the nature of the data considered in this thesis is inherently unpredictable. Therefore, the thesaurus of regular expressions is not a valid for use as an external source of domain knowledge.

4.1.7. Learning Regular Expressions

Instead of creating a predefined set of regular expressions, regular expressions can be learned from the data as well. This process is called induction of a regular language. Its goal is to learn a formal description of a regular language based on a set of example strings. Current methods, however, fail to generalise beyond literal matches of values. For example, a Python package exists that promises to learn regular expressions given a set of example strings, but its result is simply a big regular expression that matches all values literally.

If an accurate method to learn regular expressions from a set of data would exist that generalises well, it could be very helpful when reasoning about the structures of sets of values. To what degree the learning method should generalise is also dependent on the problem-specific knowledge, but presently such a method does not exist. Therefore, induction of regular languages can now not be used as a source of domain knowledge for $struct\_sim$.

4.1.8. Large Language Model

A recurring issue with the solutions proposed earlier is their inability to handle out-of-vocabulary words, making them unsuitable for dealing with never-before-seen data. Large language models such as GPT-4 mitigate this problem; these models generally derive the meaning of unseen words well. Experiments in the web interface of GPT3.5 show that it is able to answer queries such as ”Do columns with names ”email_sender” and ”emailSentBy” have the same meaning?”, despite it being likely that those column names have never been seen before.

Another advantage of the use of LLMs is that they can take the data present in columns into account. This would allow an LLM to conclude that although column names might be semantically close, based on their contents they might have a different meaning. Looking at both data and column name simultaneously is something that the previously mentioned candidate-solutions are not capable of. LLMs are also capable of answering queries related to the syntactic structure of values, which makes usable as a source of general knowledge for the $struct\_sim$ function.

An advantage of the use of large language models as a source of general knowledge is also that by means of prompt engineering, it can take problem-specific knowledge into account. If, for example, the LLM is instructed to consider all things resembling emails to be similar, this is taken into account by the LLM when asked whether the concepts ”Outlook_mail_message” and ”apple_email” are similar.

Although large language models are powerful, they do have several properties that complicate their use compared to the solutions earlier described.

Firstly, there is the nondeterministic nature of large language models; an LLM might provide two different answers to the same query, which in turn might result in inconsistent uniformity results.

Secondly, the output of large language models is not necessarily well-structured. Whereas the output of comparing word embeddings is a clear number and that of querying a thesaurus simply a set of synonyms, it is fundamentally unclear every time what the output of a large language model will look like. This adds a layer of complexity to the use of large language models.

Thirdly, a large language model can make mistakes. Mistakes caused by erroneous prompt engineering can sometimes be resolved, but mistakes caused by hallucination of the LLM are more difficult to both detect and resolve.

Despite these challenges, the ability of large language models to handle out-of-vocabulary words, compare both table and column names and data values, and compare the structures of values, makes them a good candidate for serving as a source of general knowledge and problem-specific knowledge in our use case.

4.1.9. Domain Expert

A human domain expert can function as source of general knowledge and problem-specific knowledge. The answers a domain expert provides to queries of similarity are likely to be very accurate for several reasons. Firstly, a human can also deal with out-of-vocabulary words better than a thesaurus, ontology, or word embedding. Secondly, a domain expert is likely to have the ability to accurately determine semantic similarity despite a lack of syntactic similarity, both in the case of data values and property and concept names. An example of this is similarity between the property ”Sender” from ”Outlook_mail_message” and ”From_email” from ”apple_email”. Lastly, the domain expert per definition has problem-specific knowledge which makes him a suitable candidate for being both a source of general knowledge as well as problem-specific knowledge.

A disadvantage to this approach is the time it takes the human to answer the queries. The use of a human as source of domain knowledge is promising, and we explore this option further in the next subsection.

4.1.10. Domain Expert in the loop and LLM

Finally, we propose combining the human domain-expert as source of general knowledge and problem-specific knowledge, with a large language model as source of general knowledge and explicit problem-specific knowledge. This approach combines the advantages of both approaches and mitigates disadvantages, such as the time consumption of querying a human domain expert, and the mistakes the large language model might make. For example, the prompts to the large language model can be engineered such that the LLM provides a specific output when it is unsure of the answer it returns. The human domain expert can in turn be queried to provide a definitive answer to the query. Bundling the possibility to automate querying the LLM and its broad knowledge and ability to understand explicit problem-specific knowledge, with the expert knowledge and context-awareness of a human domain expert is the best option for use as sources of knowledge in the introduced formulas.

4.1.11. Abstraction

Earlier, we introduced the function $abstracttion\_Level$ that takes a concept, a source of problem-specific knowledge, and a source of general knowledge and outputs and integer representing the level of abstraction of this concept. This function can be implemented using a domain expert in combination with a large language model as sources of knowledge, but as becomes clear from the source code of the proof of concept, we choose to implement this function with only the domain expert as source of knowledge. This is due to the fact that assigning an integer to a certain degree of abstraction requires vast problem-specific knowledge that is hard to convey to a large language model in a prompt.

4.2. Motivation behind Proof-of-Concept

The proof-of-concept is a software tool that implements the uniformity functions introduced in chapter 2. Its aim is to automate the computations required for evaluating the uniformity of the output of a digital forensic tool. As could be seen in the example uniformity computation in chapter 3, using the functions could be a time consuming and error-prone endeavour. The PoC mitigates these problems. By extension, the aim of the PoC is also to show that it is possible to implement the uniformity functions in software, and that they can exist outside of their formal definitions.

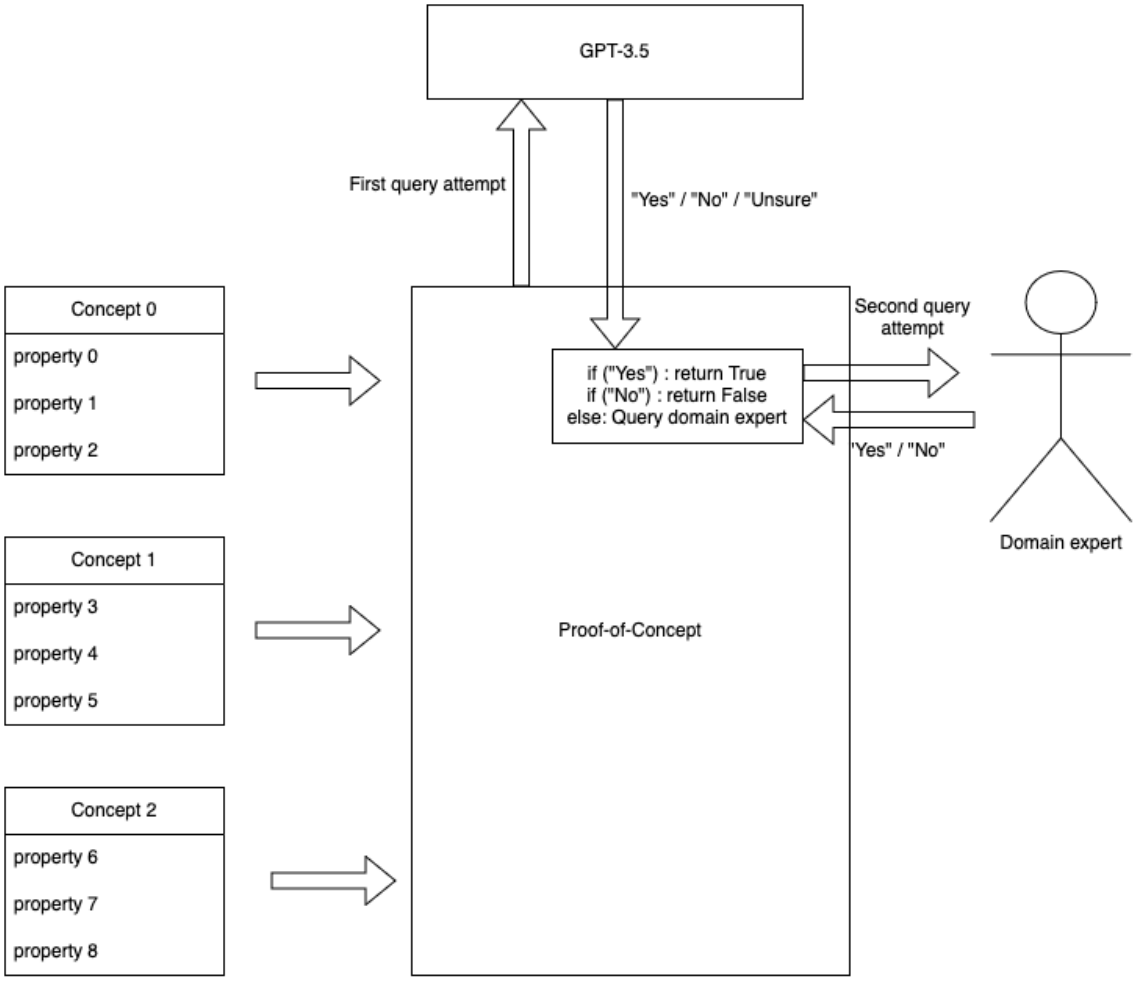

Figure 4.1: Workflow demonstrating querying mechanics

4.3. Architecture and Workflow

The proof-of-concepts is a software tool written in Python. It takes as input the output of the digital forensic tool of which the uniformity is to be computed. This output is to be provided in separate `.tsv` or `.csv` files, where each file represents a *concept*, the name of the file is the name of the *concept*, the column names are the *property* names, and the values in the columns are the values belonging to those properties.

4.3.1. Workflow

The PoC starts by reading the provided tool output and converting it to an internal representation of concepts, properties, and values. This conversion is specific per tool output format, so if support for a different format, for example SQLite databases, is required, the conversion capabilities need to be extended. Next, computation of the values of the uniformity functions begin. If required, the domain expert is asked. When all required queries to either GPT or the domain expert are finished, values for each type of uniformity are returned. Figure 4.1 shows a high-level schematic of this process.

4.3.2. Knowledge Sources

Recall that the uniformity functions rely on external sources of problem-specific knowledge and general knowledge, and that earlier it was concluded that a large language model in conjunction with a human domain-expert can serve as an effective source of both these types of knowledge. In the uniformity functions, these knowledge sources are used to determine whether concepts, properties, syntactic structures or embeddings are similar or not. Recall that all similarity functions, for example $ConceptSim$, return a boolean.

All similarity functions are implemented in a similar way, and only differ in two aspects, namely the input they take and the prompt they use to query GPT. For example, in the case of the implementation of $propertySim$, the prompt contains instructions to GPT about how to interpret properties and how to compare properties. The objects that are to be compared by GPT are appended to the prompt. The format in which this is done depends on which similarity function is implemented. For example, when comparing concepts, the prompt includes the name of the concept and the names of the properties, whereas when comparing types of values, only values are appended.

The prompts also contain instructions that specify how the output should be formatted. In all prompts, the GPT is instructed to return ”yes” in case the compared objects are deemed equal, ”no” in case the objects are deemed not equal, and ”unsure” in case GPT can not make a confident decision. In case ”yes” or ”no” are returned, the similarity function simply returns either the boolean value $True$ or $False$. In case GPT outputs ”unsure”, the human domain expert is prompted to make a decision about the similarity of the objects being compared. The domain expert is then presented with some representation of the objects that are being compared, and they can simply answer ”yes” or ”no” in response. This answer is converted to either $True$ or $False$ and returned by the similarity function.

The subtle difference between this implementation and the formal definition of the similarity functions introduced in chapter 2, is that the knowledge sources are presented as explicit parameters of the similarity functions in the formal definitions, whereas the similarity functions in the proof-of-concept have access to the knowledge sources directly from within their implementation. In other words, the similarity functions in the PoC do not need to be passed the knowledge sources as explicit parameters.

In the case of the proof of concept, we opted to use OpenAI’s `gpt-3.5-instruct` model. Advantages of this model are its high rate-limits and the fact that it is trained to be good at following particular instructions rather than chat-completion. It is, however, not as advanced as OpenAI’s newer models, such as GPT-4o.

4.3.3. Key Challenge: Overcoming High Response Time

One challenge in implementing the proof of concept is the time required for a request to OpenAI’s API, which takes approximately 0.3 seconds. While this is relatively fast, it becomes problematic when the tool is applied to large datasets requiring numerous comparisons.

A potential solution is to make simultaneous API requests using multiple threads, thus increasing the number of requests that can be processed per second. However, the proof-of-concept is inherently single-threaded. It heavily relies on grouping similar items - such as concepts or properties - by assigning them to lists based on similarity functions. Since these groupings are constantly updated, multi-threading is not feasible. To address this, we constructed a mapping that stores the results of pre-computing the output of the similarity function for each possible comparison in parallel. This mapping acts as a cache, allowing for rapid similarity assessments by storing API responses for all potential queries.

While this approach may lead to more API requests than strictly necessary, we believe the resulting speedup justifies this trade-off. Another drawback is the potential to hit rate limits with OpenAI's advanced models due to the frequent queries. However, the high rate limits of gpt-3.5-instruct help mitigate this issue. Therefore, to successfully implement these functions using external knowledge sources, the PoC needs to use the knowledge sources in such a way that their output can be converted to a boolean.

4.3.4. Source Code

The source code of the proof-of-concept can be found in the following Github repository: https://github.com/luukvancampen/uniformity.

To run the tool, the following steps have to be taken:

- Create an OpenAI account and create an API key. This key is used to query GPT-3.5.
- In the users virtual environment, the environment variable `OPENAI_API_KEY` needs to the OpenAI api-key
- (Optional) Change the prompts in `prompts.py` so they accurately reflect the users opinion on when objects should be deemed similar
- Execute the `Main()` function in `Main.py`
- When the user is asked whether two objects are similar, provide an answer

The `README` of the repository also contains instructions on how to run the tool.

5

Experiment: Accuracy of Uniformity Functions and PoC

The goal of this chapter is to elaborate on the conducted experiment and the results that followed. The results are discussed in the Discussion chapter, and they are used in the conclusion chapter to answer the following research questions:

- **RQ-2.1** How accurate is the uniformity metric defined in this thesis?
- **RQ-3.1** How do the perceived workloads of quantifying uniformity based on intuition, the functions introduced in this thesis, and using the proof-of-concept compare?
- **RQ-3.2** How accurate is the proof-of-concept compared to manually computed uniformity scores?

In order to answer these questions, an experiment was conducted, consisting of three steps. First, a group of experts from the digital forensics domain are asked to intuitively assign uniformity values to a synthetic output of a fictitious digital forensic tool. The uniformity scores assigned during this step serve as a baseline to compare the scores assigned in the second step to. Secondly, the experts are asked to compute uniformity values for the same output using the introduced uniformity functions. Comparing the values resulting from this step to the values resulting from the first step allows us to answer research question 2.1. Lastly, the experts use the proof-of-concept to assign a uniformity value to that same output. Comparing the uniformity scores resulting from this step to those resulting from step 2 allows us to answer research question 3.2. After each step, the experts are asked to fill out the NASA-TLX, which serves as an indication of the effort required to do the task. This allows us to answer research question 3.1.

In remainder of this chapter, we further elaborate on the design of this experiment and the rationale behind it. We also discuss potential shortcomings.

5.1. Experiment Design

In this section, the design and rationale behind the conducted experiment is elaborated on.

5.1.1. Fictitious Digital Forensic Tool Output

The fictitious digital forensic tool output was crafted such that it displays a lack of five of the six types of uniformity introduced in this thesis. The data does not show a lack of high-level syntactic uniformity. It consists of five concepts (”chat”, ”email”, ”outlook_mail_message”, ”wifi”, and ”wifi”), with properties

that fit these types of concepts. There are five rows of data in each concept, which in this experimental setting is enough to draw conclusions about the all relevant aspects of the data. A lack of each type of uniformity is represented in the data in the following ways:

- **High-level semantic:** the concepts ”email” and ”outlook_mail_message” are not at the same level of abstraction, because ”email” is more abstract. This is therefore a lack of high-level semantic uniformity.
- **Low-level semantic:** the concepts both named ”wifi” are similar concepts since they both represent connections to wifi networks, but they do not have the same properties. This can therefore be considered a lack of low-level semantic uniformity.
- **Structural semantic:** the timestamps in the ”Send Time” property of the concept ”chat” has a resolution of seconds, whereas the timestamp ”Last Connected” in (one of) the ”wifi” concepts, has a resolution of nanoseconds. This displays a lack of structural semantic uniformity, because both these properties contain values of the type ”timestamp”, but their structure does not allow storing the same amount of information. This can therefore be considered a lack of structural semantic uniformity.
- **Low-level syntactic**: both ”wifi” concepts have properties that contain the name of the wifi network to which was connected. In one ”wifi” concept, this property is called ”SSID”, in the other it is called ”Name”. This therefore constitutes a lack of low-level syntactic uniformity.
- **Structural syntactic:** the ”email” concept has the property ”Send_time”, which contains values that are timestamps formatted as Unix timestamps. A property that also contains timestamps is ”Send Time” of the concept ”chat”, but those timestamps are formatted in a human-readable way. This displays a lack structural syntactic uniformity because values of the same type are represented using a different syntactic structure.

The fictitious data can be found in the Github repository linked in the chapter on the proof-of-concept.

It is important to remark that these lacks of uniformity are in the end solely lacks of uniformity based on the opinion of the author. This is due to the fact that the uniformity functions all rely on an external source of knowledge to judge whether concepts, properties and syntactic structures are similar. One person might state that the sample data is perfectly uniform because all concepts represent things in the universe, and the syntactic structures of the timestamps are all similar because they are shorter than 100 character. However, because the author is familiar with the digital forensic domain and the level of abstraction at which the experts in the digital forensic domain operate, it is valid to assume that the participants in the experiment consider these instances of lacks of uniformity to be judged as such as well. This fictitious digital forensic tool output is therefore valid for use in the experiment.

5.1.2. Participants

The participants in this experiment are software developers working to maintain the forensic knowledge contained in Hansken. These software developers have extensive experience in creating mappings between various types of data and the Hansken trace model. While creating these mappings, they have come across various problems regarding uniformity of data that has to be mapped. Due to this experience, it can be stated that they are experts in this domain.

5.1.3. NASA-TLX

To evaluate the subjective workload of computing the uniformity value of a dataset, the NASA-TLX[8] was used. The NASA-TLX is widely used to compute the subjective workload of doing a particular

task. In the first part of the questionnaire, participants are asked to rate the perceived workload of a task based on six scales. In the second part, they are asked to pairwise compare the six scales to determine which scales are the most important to the participant. This weighting is then applied to the results of the first part to compute a subjective workload score. The use of the NASA-TLX allows us to answer research question RQ-3.1.

5.2. Process

In this section, the individual steps conducted during the experiment are discussed.

5.2.1. Preparing the Experts

The digital forensic experts we first introduced to the six types of uniformity introduced in this thesis, by means of a presentation and examples of each type of uniformity. The experts were also thought the meaning of the terms ”concept”, ”property” and ”type” in the context of this experiment. The experts having an understanding of these types of uniformity is vital for the first task in the experiment, namely assigning uniformity values based on intuition.

5.2.2. Assigning Uniformity Scores Based on Intuition

As the first step in this experiment, the digital forensic experts were asked to assign uniformity score to the fictitious tool output based on their intuition. The values intuitively assigned by the experts serves as a baseline for assessing the quality of the uniformity functions introduced in this thesis. This step was concluded with the participants filling out the NASA-TLX for this task.

5.2.3. Assigning Uniformity Scores Based on the Created Functions

As the second step in this experiment, the experts were asked to compute uniformity scores using the functions introduced in this thesis. The values resulting from this can, in turn, be compared to the values resulting from the previous step. This provides an insight into the differences between the intuition of the experts and the results they obtain when using the introduced functions. Since there is nothing else to compare the output of the uniformity functions to, this is also the best metric regarding the correctness of the created functions. This step in the experiment allows us to answer research question 2.1. The closer the values resulting from this step are to the values resulting from the previous step, the more support there is that the functions created in this thesis are accurate. This step was concluded with the participants filling out the NASA-TLX.

5.2.4. Assigning Uniformity Scores Using the Proof-of-Concept

As the final step in the experiment, the experts were asked to use the proof-of-concept to compute uniformity scores. The resulting values can be compared to the values resulting from the previous step. The closer the values resulting from the proof-of-concept are to the values resulting from the previous step, the more support there is that the results produced by the proof-of-concept are accurate. With this comparison, we can answer research question 3.2. Finally, this step was concluded with the participants filling out the NASA-TLX.

5.3. Results

In this section, we present the results obtained from executing the experiments described in the previous chapter. Recall that the participants of the experiment performed the following three tasks:

- Assign uniformity scores to a synthesized digital forensic tool output based on their intuition, and fill out NASA-TLX

Table 5.1: Results obtained by digital forensic expert 1.

Expert 1	Intuition	Developed metric	Proof-of-concept
High-level Semantic Uniformity	0,9	0,5	0,5
High-level Syntactic Uniformity	0,5	0,75	1,0
Low-level Semantic Uniformity	0,3	0,79	1,0
Low-level Syntactic Uniformity	0,3	0,86	0,14
Structural Semantic Uniformity	0,3	0,81	0,97
Structural Syntactic Uniformity	0,2	0,68	0,96
Average Uniformity	0,42	0,73	0,76
Perceived workload	39,7	59,7	43,3

- Assign uniformity scores to a synthesized digital forensic tool output using the uniformity functions, and fill out NASA-TLX
- Assign uniformity scores to a synthesized digital forensic tool output using the proof-of-concept, and fill out NASA-TLX

The results of these steps are gathered in tables 5.1, 5.2, and 5.3. Each table contains the results obtained by a single participant, and each column contains the results obtained in a single step of the experiment. The last row in each table shows the perceived workload for each task.

5.3.1. Uniformity Based on Intuition

The values in the ”Intuition” columns of tables 5.1, 5.2, and 5.3 show that the digital forensic experts assigned varying levels of uniformity when basing this value on intuition alone. For two of the experts, the perceived workload when assigning uniformity values based on intuition, is relatively high when compared to that of using the proof of concept. For one expert, this task incurs a slightly lower perceived workload compared to that of using the proof-of-concept.

5.3.2. Uniformity Based on Developed Metric By Hand

For participants 2 and 3, the uniformity values obtained using the developed metric by hand are more consistent than those obtained when assigning uniformity scores based on intuition. The uniformity value assigned by participant 1 is significantly higher than the one assigned by them when basing the value on intuition. Using the developed metric by hand to compute uniformity incurs the highest perceived workload for two of the participants. One participant found it to incur a lower perceived workload than assigning uniformity scores based on intuition.

5.3.3. Uniformity Using the Proof-of-concept

The uniformity results obtained by the experts when using the proof-of-concept are close together. For expert 1, the obtained score is close to that obtained when expert 1 assigned a uniformity value based on the developed metric. For participant 2, the value obtained using the proof-of-concept is closer to that obtained when basing the value on intuition, rather than using the developed metric by hand. The use of the proof-of-concept incurs a significantly lower perceived workload for all participants compared to the perceived workload incurred by using the developed metric by hand. For experts 2 and 3, the perceived workload is also significantly lower than that incurred by assigning uniformity scores based on intuition.

Table 5.2: Results obtained by digital forensic expert 2.

Expert 2	Intuition	Developed metric	Proof-of-concept
High-level Semantic Uniformity	0,8	0,5	0,33
High-level Syntactic Uniformity	0,7	0,6	1,0
Low-level Semantic Uniformity	0,5	0,56	1,0
Low-level Syntactic Uniformity	0,6	0,85	0,51
Structural Semantic Uniformity	0,8	0,95	0,96
Structural Syntactic Uniformity	0,8	0,86	0,97
Average Uniformity	0,70	0,52	0,80
Perceived workload	64,7	57,7	39,0

Table 5.3: Results obtained by digital forensic expert 3.

Expert 3	Intuition	Developed metric	Proof-of-concept
High-level Semantic Uniformity	0,4	0,33	0,5
High-level Syntactic Uniformity	0,4	0,75	1,0
Low-level Semantic Uniformity	0,4	0,45	1,0
Low-level Syntactic Uniformity	0,4	0,5	0,5
Structural Semantic Uniformity	0,2	0,75	0,97
Structural Syntactic Uniformity	0,6	0,35	0,97
Average Uniformity	0,40	0,52	0,82
Perceived workload	65,3	77,0	25,7

6

Related Work

In the following subsections, we will present an overview of work done in fields related to data integration and other topics related to this thesis.

6.1. Schema Document Complexity

Basci and Misra[2] propose a metric for determining the complexity of an XML schema document by means of entropy. This complexity mostly relates to the structure of the schemas, and it does not consider the semantics of elements in a schema. Pušnik et al. propose a novel quality measuring approach and also define six aspects of schema quality. As argued earlier, the goal is using uniformity of output of digital forensic tools as guideline for the time required to integrate that tool into Hansken. This required time can be regarded as difficulty as well; a non-uniform output is more difficult to map to the Hansken trace model. The ability to quantify the complexity of an XML schema could provide similar information, especially because relations between objects in the schema are also regarded when determining complexity. Therefore, if digital forensic tools would provide XML schemas of their output, quantifying their complexity could prove to be a useful guideline when determining the complexity of integrating the tool. However, since we do not have XML schemas describing the output of tools, this approach is not a valid solution to our problem.

6.2. Semantic Integration

Semantic integration is an umbrella term related to data integration and schema matching; more specifically, Noy et al.[12] specify the problem as solving many semantic-heterogeneity problems, such as ”matching ontologies or schemas, detecting duplicate tuples, reconciling inconsistent data values, modelling complex relations between concepts in different sources, and reasoning with semantic mappings”. This thesis touches on the topic of semantic heterogeneity in the uniformity functions. We essentially aim to quantify the degree of semantic heterogeneity. It is therefore worthwhile to explore this topic further in the remainder of this section.

Bergamaschi et al.[3] propose an approach to the integration and query of multiple heterogeneous information sources, containing structured and semi-structured data. Their approach, MOMIS, does information integration based on the conceptual schema, or metadata, of the information sources, and on certain architectural elements. The first of these architectural elements is a common object-oriented data model, which is used to describe source schemas for integrating purposes. The second architectural element consists of one or more wrappers to translate schema descriptions into the common

object description language, and the final element is a mediator and a query processing component. The approach in the paper makes use of a common thesaurus which is constructed from the ODL descriptions. It is made clear that the creation of the common thesaurus is a semi-automatic process. The relations that the thesaurus is supposed to capture are more complicated than simple semantic relatedness; the thesaurus is supposed to tell whether words are synonyms, broader terms, or related terms. The first step when integrating the semi-structured data is associating an object pattern with each set of objects having the same label in the source graph. This means making groups of objects where the objects have the same attributes. The objects from the structured schema and the object patterns are translated to an object description language. The process also includes the building of the mediator integrated global schema. In this step, Object Description Language objects that have a semantic relationship in different sources are identified. For all possible pairs of objects, a numerical value between 0 and 1 is determined, the affinity coefficient. This is based on the terminological relationships in the common thesaurus. Affinity coefficients determine the degree of semantic relationship of two classes based on their names and attributes. The global affinity coefficient is a linear combination of the name affinity coefficient and the structural affinity coefficient. A hierarchical clustering algorithm is used to classify ODL classes according to their degree of affinity. The affinity coefficients determined in this process could be a valid technique for use in the uniformity functions introduced in this thesis. The function $conceptSim$ could be implemented using this approach by choosing a certain limit for the affinity value, above which the function would return $true$, and below which it would return $false$. The technique proposed in this paper relies on meta-information about the data. In the context of this thesis, where the input data can be of vastly different formats, we have no guarantees about the availability of such data, which means we can not be sure that it is possible to construct the common object-oriented data model, let alone in an automated manner.

Castano and De Antonellis[5] propose a computer-aided, affinity-based schema unification method. Their method works by means of analysis of schema elements in order to identify elements with affinity in different source schemas. Affinity is the level of semantic relationship between elements in different schemas. This approach assumes the existence of an explicit and well defined schema. For structural affinity, names, domains, and cardinalities are taken into consideration. For name affinity, the authors rely on the semantic contents of names of schema elements. To capture the semantic relatedness of words, a thesaurus is used. A thesaurus in this context is a sort of dictionary that specifies the relations between words, such as whether they are synonyms, bigger terms, or narrower terms. The affinity function is capable of considering the strength and length of paths between objects in the thesaurus. An example from the paper is the path between Individual and Professor; Individual is synonym of Person, and Person is a bigger term of Professor. This results in a path length of two. For analysing the structure of objects, the concept of semantic correspondence is introduced. Semantic correspondence between objects can be determined when the name affinity between objects is higher than 0. Semantic correspondence between properties can be done in two ways, namely by means of name affinity or analysing the domain of properties (domain compatibility). Properties that have both name and domain compatibility have stronger correspondence than is the case when there is only name compatibility. The different relations described in the thesaurus used in this approach could be of use in the uniformity functions in this approach, specifically in that of high-level semantic uniformity. If two concepts can be compared in such a thesaurus, it could become clear instantly whether concepts are at the same level of abstraction, since a bigger term can be considered more abstract and vice versa, or concepts are synonyms. This would remove the need for the function $abstractionLevel$, since the level of abstraction of individual concepts no longer has to be determined. At the same time, the need for a thesaurus is a limitation, since it would have to be constructed for digital forensic tool of which the uniformity is to be determined, which makes this approach inflexible. This limitation was also discussed in the chapter on possible knowledge sources. It is also not always straightforward to determine the

domain of properties. For instance, it required problem-specific knowledge to realise that an integer is a Unix timestamp, and that therefore the domain of a property containing such values is vastly different than that of a property containing values that are actual integers. Since the approach described by Castano and De Antonellis assumes the existence of a schema describing the data, it is a valid approach in their case, but since we can not assume to have a schema, we can not use it.

6.2.1. Domain Incompatibility Problem

The domain incompatibility problem is described by Czejdo et al.[7]. This problem can manifest itself in multiple forms. First, there could be the issue of semantically similar entities having a different name. Secondly, the domain of semantically similar attributes might be incompatible; an example of this could be one table defining ID as a 9-digit string and another as an 11-digit string. Thirdly, there may not be a one-to-one mapping available between attributes and a many-to-one or vice versa mapping will have to be created. An example of this could be one database schema splitting addresses into a street, house number, and zip code, whereas this might all be a single column in a different schema. The authors propose to solve this problem by using extended abstract data types in schema definitions and by augmenting the relational model ”connectors”. Note that the provided example on the difference in how addresses are stored, can be a case of a lack of low-level semantic uniformity when the concepts to which these columns belong are similar. For example, if there is a ”subscriber” table with the columns ”street”, ”house number”, and ”zip code”, and another ”subscriber” table with the column ”address”, the concepts represented by those tables can be semantically similar, but they do not have the same properties. Therefore, the amount of effort or complexity it takes to reconcile this difference using the approach proposed by Czejdo et al., could serve as an indication of the degree of low-level semantic uniformity of data. A disadvantage of this is that it would require an objective way to measure effort or difficulty or complexity of employing their approach, which is not straightforward. The approach of Czejdo et al. also relies on the availability of well-defined schema that describes the data, which is not always available in the case of the output of digital forensic tools. Therefore, although the techniques described in by Czejdo et al. are interesting, it is not possible to employ their solutions in quantifying uniformity of the output of digital forensic tools.

Sheth et al.[14] explore the topic of semantic proximity. They define semantic proximity of two objects to be a four tuple consisting of a context, abstraction, the domains of the two objects, and the states of the objects. The context is considered because two objects can be semantically similar in one context and not in another. In case the objects do not have semantic proximity in either all or no cases, the contexts in which the objects have semantic proximity has to be specified. The abstraction is used to refer to a mechanism used to map the domains of the object either to each other, or a common third domain. Domains refer to sets of values from which the objects can take their values. The state of an object can be thought of as a record of an object in a database. The notion that objects can be semantically near in one context and not in another, captured in this thesis in the form of problem-specific knowledge. The role of the problem-specific knowledge is to work in conjunction with the source of general knowledge to make decisions on whether object are semantically near. The authors also describe problems that arise in schema integration that can be related to types of uniformity introduced in this thesis. For example, the authors mention naming conflicts, which are semantically similar attributes (properties in this thesis), having different names. This relates to low-level syntactic uniformity. Union compatibility conflicts, which is similar concepts having different attributes, and schema-isomorphism conflicts, which is concepts having a different number of attributes, are also mentioned. These types of conflict are quantified in this thesis as low-level semantic uniformity. Furthermore, the authors mention data representation conflicts, data scaling conflicts, and data precision conflicts, which can be related to the two types of structural uniformity. An interesting direction for future work could be to explore whether it is valuable to further refine the types of uniformity so these three types of conflicts can be

quantified on their own. The authors also mention conflicts that can arise when an explicit database schema is present, namely default-value conflicts and attribute integrity constraint conflicts. Because the the output of digital forensic platforms generally does not include a schema, these conflicts are not relevant for this thesis.

6.2.2. Schema Conflicts and Data Conflicts

Kim et al.[9] make the distinction between schematic and data conflicts in a multi database system. It is argued that a database is defined by its schema and its data. Schema conflicts result from the use of different schema definitions in the multiple databases, and data conflicts are due to inconsistent data in the absence of schema conflicts. Although generally no explicit schema is available in the data considered for this thesis, one might be derived, and therefore we can still have schema conflicts. The authors state that there are two causes for schema conflicts, namely the use of different structures to represent a semantically similar entity type, the second cause is the use of different specifications for the same structure. This could be, for example, the use of different names and data types for semantically equivalent tables or attributes. The paper also discusses table-versus-attribute conflicts, causes by data being stored as attributes in one schema and in a table in another. A similar situation could probably occur in the data considered in this project.

Data conflicts can arise due to two situations: conflicting data constraints and different representations of the same data. In the context of this thesis, the first of these problems can per definition not occur, since an explicit schema is generally not present, and therefore neither are constraints. The latter of the problems is highly relevant to this thesis. The paper notes that it is also very well possible to have a combination of the problems mentioned above.

6.3. Semi-structured Data and Similarity

Recall that the uniformity functions introduced in chapter 2 all rely on functions that determine whether either concepts, properties, or syntactic structures are similar. These functions work by accessing sources of general knowledge and problem-specific knowledge to make decisions about similarity. In literature, other techniques are presented to determine similarity of concepts and properties. In this section, we discuss some of these approaches. We start out by providing discussing the concept of semi-structured data, and argue why the output of open-source digital forensic platforms is generally semi-structured. This serves as a foundation for the discussion of techniques from literature to determine similarity of concepts and properties.

6.3.1. Properties of Semi-Structured Data

Abiteboul et al.[1] outline several properties of semi-structured data. These properties are the data being irregular, the data having an implicit structure, the structure being partially present, the structure being only indicative, the data having an a-posteriori data guide, a rapidly evolving schema, and the data being self-describing.

The first property of semi-structured data is that it is irregular, which means that some records omit fields, whereas others record additional information. The output of digital forensic platforms generally seems to adhere to this property; the fields of objects that originate from the same place might not be the same. For example, one picture might have exif data, whereas another might not. If we consider this from a more general view, we can of course have the situation that entities that have similar semantics do not all have the same fields, but this is more related to the uniformity and it is not so much what is meant in this context with irregularity. This principle can be seen in the running example. We

can generally expect to see similar fields in all instances of ”Outlook_mail_message”, but we see that ”Apple_email” and ”Outlook_mail_message” have different fields.

The second property of semi-structured data is that its structure is implicit, which is the case with the output of digital forensic platforms. For instance, wha the data shown in the running example means can be derived from examining the column names and column contents, but a specification of what the data means is missing. Intuitively, one could consider the data to have an explicit structure, since generally the field names are provided. This is, however, not what makes data structured, since parsing is required to obtain the actual data and the correspondence between the parse-tree and the logical representation of the data is not always immediate.

The third property of semi-structured data is that the structure could only be partially present. The running example shows a case of this in the ”Content” column. The text within that column can be unstructured, whereas the data in the other columns is structured.

The fourth property of semi-structured data is that it generally uses an indicative structure rather than a constraining structure. A standard relational database generally follows a strict schema that can specify what data is allowed in it, including types and possibly constraints. In situations where such a schema is considered too restrictive, this approach might not work, and an indicative schema might be used, which simply indicates the current type of the data.

The fifth discussed property of semi-structured data is the usage of an a-priori schema versus an a-posteriori data guide. In the case of forensic data, the notion of a schema is often posterior to the existence of data; this simply stems from the fact that forensic activities are always done after a fact. The data in this thesis adheres well to this property, since the tools that generate the data are usually purpose-built to gather information on a particular type of file that is preexisting.

The next relevant property of semi-structured data discussed is the rapid evolving of the schema. To be able to find as much forensically valuable data as possible, changes in forensically valuable files have to be implemented in forensic tools as well, thereby possibly changing the output and possibly the schema.
Furthermore, Want et al.[15] state that semi-structured data can be self describing. This means that each object can contain its own schema, and the distinction between schema and data is blurred.

In conclusion, the output of the digital forensic platforms can be considered semi-structured. The biggest consequence of this is that there is no explicit schema available and we can not make any assumptions about the values present in the data. The approach to quantifying uniformity that we introduce later, revolves around this property.

6.3.2. Determining Similarity

The previous subsection concluded with the notion that the output of open-source digital forensic platforms is generally semi-structured. In this subsection, we explore methods used in schema matching that can be used to determine the similarity of concepts and properties when the data is not semi-structured. These methods might be relevant in case the uniformity functions are applied to data that is not semi-structured, and a different implementation of the similarity functions is desired.

Similarity in Schema Matching
Schema matching is the task of finding semantic correspondences between elements of two schemas. In this section, we provide an overview of methods from schema matching used to find these semantic

correspondences.

Melnik et al.[10] propose an algorithm that can be used for matching of diverse datastructures, such as graphs. The authors propose converting database schemas to graphs and in turn using the matching algorithm to identify corresponding elements between those graphs. The approach to matching the graph makes use of string similarity and it uses a human domain expert as external source of knowledge, since the correct match often depends on the information only available or understandable by humans.

The fact that this approach involves the use of a database schema makes it unsuitable for application to semi-structured data. However, the principle that two objects are similar if they have similar neighbours, which is how the graph is used to determine semantic similarity, is relevant in the case of semi-structured data. It might, for example, be possible to determine that tables are similar if they have columns with similar names. The required problem-specific knowledge to determine whether high-level concepts are similar can in this case come from the human domain expert. It is explicitly mentioned that in some cases the knowledge whether concepts are semantically similar can only come from a human domain-expert, which in essence is the same as requiring problem-specific knowledge.
Scalability of this approach might be problematic due to the human domain expert having to be queried often.

Fernandez et al.[6] propose the building of a hypergraph that captures the syntactic relationships between data from different sources. The construction of this graph relies on the generation of signatures of columns, which are ”summaries” that contain information about a column such as, content sketches, cardinality, and data distribution. Relationships in the hypergraph are then constructed based on the similarity of column signatures.

This approach leans heavily on syntactic similarity between data sources. This likely yields incomplete results in situations where there is semantic similarity, but no syntactic similarity, for instance in the situation where timestamps are represented in different formats. This approach lacks the possibility to take problem-specific knowledge into account. For example, this approach applied to the running example might lead to inaccurate result due to low syntactic similarity, and there is no way to provide the approach with problem-specific knowledge to solve this problem.

Zhang et al.[16], propose a way to cluster columns into attributes. In this context, attributes are strong relationships between columns based on the common properties and characteristics of the values they contain. The authors provide the example of distinguishing two columns containing values of the same primitive datatype, namely integer, of which one contains phone numbers and the other social security numbers. From this primitive datatype there is no indication that these columns might be semantically related, but in the case of two columns that contain similar data, such as phone numbers, this semantic relatedness can be discovered. This approach therefore determines semantic similarity between columns that contain similar types of data based solely on characteristics of that data, but it disregards other indications of semantic similarity, such as column names or table names. The authors explicitly state that their approach does not make use of any external source of information, which might mean that cases where semantic similarity is less obvious or not derivable from the data, are missed. This also means that problem-specific knowledge can not be used to determine similarity of high-level concepts in this approach.

Nargesian et al.[11] define the table union search problem and present a probabilistic solution for finding tables that are unionable with a query table within massive repositories. The authors aim this

approach on semi-structured data, despite not using this exact term; they note the absence of a strict schema, which results in the need to consider attribute values to determine unionability. Contrary to other approaches that use only value overlap or class overlap heuristically to find unionable attributes, this paper defines attribute unionability using three statistical tests based on value overlap, annotation of attributes with classes from an ontology and class overlap (semantic unionability), and natural language similarity. To measure semantic unionability, attributes are mapped to classes by means of an ontology. Natural language similarity is determined by means of word embeddings; each attribute is assigned a set of word embeddings that can be compared to that of another attribute. The paper presents a test that can be used to determine which of those metrics can be used best to determine whether attributes are unionable.

As seen earlier, some knowledge about whether or not to match particular attributes can only be provided by a human who functions as an external source of knowledge. This approach attempts to capture this external knowledge using an ontology and word embeddings, but this approach likely suffers from a lack of performance in the case of out-of-vocabulary words. If a class is not present in an ontology or in the training data of the word embedding, this approach might yield incomplete results. An ontology is, however, in principle well suited to capture problem-specific knowledge. Concepts that are deemed similar by the problem-specific knowledge can be recorded as such in the ontology. Training word-embedding such that concepts deemed similar by problem-specific knowledge is also possible, but this process is probably very costly.

7

Discussion

7.1. Overview of Open-source Digital Forensic Platforms

Several lists of open-source digital forensic platforms can be found online, but these lists generally do not provide the reader with additional information that can be used as a guideline to compare platforms and gain more in-depth information about them. The overview presented in this thesis does provide the reader with a brief analysis of the found platforms, both in terms of quantifiable metrics, such as the number of file types the platform is able to extract traces from, and more subjective matters, such as how well the code is structured. From this overview we can conclude that development of open-source digital forensic platforms is actively ongoing, with several large platforms being the result of this.

Upon analysing the code structure of several of digital forensic platforms, we see that they are generally structured in a very modular way. This way, a platform can be extended to support extracting traces from a new type of file, without having to modify much existing code. Instead, if code adheres to a particular structure, such as implementing a particular interface, the platform is able to utilise it without much additional configuration. This has the benefit that is is generally very clear how the individual parts of the code work, which makes it easy to extract the forensic knowledge implicitly embedded in the code. Forensic knowledge in this context is the knowledge of where in a particular file certain information can be found and what forensically relevant parts of a file mean. Therefore, if an open-source digital forensic platform is able to extract traces from a type of file for which support is to be added to a different platform, it might be worthwhile to consider integrating that specific part of the code, or extracting the forensic knowledge and using that in a new implementation.

The overview presented in this thesis does not inform the reader about what specific file formats the platform can extract traces. If the reader is interested to know whether a platform can extract traces from a particular file, they have to resort to either the documentation of the platform or the source code.

To make the presented overview more complete, a list of files from which the platforms can extract traces can be included. This will make it more obvious to the reader whether the platform contains support for the files in which the reader is interested.

7.2. Uniformity Functions

We developed a definition of uniformity for the output of open-source digital forensic platforms. This definition consists of six sub-forms of uniformity, for each of which we provide formal definitions of func-

tions that can be used to compute them. We also provided an example of a computation on fictitious data.

The six types of uniformity together with the formal definitions of functions to compute them serve as an answer to research question 2, ”what is the best way to quantify uniformity of the output of open-source digital forensic platforms?”

The field of digital forensics lacks a method to assess the uniformity of output of digital forensic platforms. More generally, to the best of our knowledge, there does not exist an approach to assess the uniformity of sets of semi-structured data. The uniformity functions can be a valuable addition to the field of digital forensics in multiple ways. Firstly, the uniformity definitions can be taken into account during development of digital forensic platforms. This might result in developers taking the uniformity of their platforms output into consideration more, which in turn may lead to better interoperability of platforms.

7.3. Proof-of-Concept

The uniformity functions introduced in chapter 2 are formal definitions, and the presented example computation was done by hand, which was error-prone and much work, especially for large outputs of digital forensic platforms. To show that the uniformity functions can be automated and that it is possible to create concrete implementations of the formally defined functions, a proof-of-concept was created. It was also shown that it is possible to find concrete instances for the sources of general knowledge and problem-specific knowledge, which were left largely abstract in the definitions of the uniformity functions.

7.4. Experiment: Accuracy and Perceived Workload of Uniformity Functions and PoC

An experiment was conducted to obtain data that can be used to answer research questions 2.1 (How accurate is the uniformity metric developed in this thesis), 3.1 (How do the perceived workloads of quantifying uniformity based on intuition, the approach introduced in this thesis, and the PoC compare?), and 3.2 (How accurate is the PoC compared to manually computed uniformity scores).

7.4.1. RQ-2.1: How accurate is the uniformity metric developed in this thesis?

To answer research question 2.1, ”how accurate is the uniformity metric developed in this thesis?”, the experiment involved digital forensic experts assigning uniformity values to a set of data based on their intuition. The idea behind this step is that the values assigned in this step can serve as a baseline to which the values resulting from the next step in the experiment, the experts computing uniformity values using the functions introduced in chapter 3, be compared. The results in figures 5.1, 5.2, and 5.3 show that the values obtained by the experts in the intuition step vary rather significantly for expert 1, but less so for experts 2 and 3. It is therefore hard to answer the research question based on this data alone.

A possible explanation for the deviating results might be that the experts were not given any direction on how to assign uniformity values based on intuition, apart from that they could range from 0 to 1, and that 1 is the highest possible score. The inherent subjectivity involved with assigning the values based on intuition can be a possible cause for the deviation.

7.4.2. RQ-3.1: How do the perceived workloads compare?

The data required to answer research question 3.1, ”How do the perceived workloads of quantifying uniformity based on intuition, the approach introduced in this thesis, and the proof-of-concept compare?” was obtained by having the experiments’ participants fill out the NASA-TLX after each step of the experiment. The results show that that experts 2 and 3 assign the lowest perceived workload to the use of the proof-of-concept, whereas expert 1 assigns the lowest perceived workload to assigning uniformity values based on intuition, but this perceived workload value is not significantly lower than that resulting from the use of the proof-of-concept. For experts 2 and 3, there is no significant patter to discern between assigning uniformity values based on intuition and computing them using the functions introduced in chapter 3. Using these values, we can carefully conclude that the use of the proof-of-concept incurs the lowest perceived workload out of the three tasks the experts performed. It must be noted, however, that the sample size is small, and more conclusive results can only be drawn by repeating the experiment with more participants.

7.4.3. RQ-3.2: How accurate is the PoC compared to manually computed uniformity scores?

To obtain the data required to answer research question 3.2, ”How accurate is the PoC compared to manually computed uniformity scores?”, the experiments’ participants used the proof-of-concept developed in chapter 4 to compute uniformity scores. The results show that the uniformity values computed by the experts when using the proof-of-concept are relatively similar, contrary to the values computed manually by using the functions. In the case of experts 2 and 3, the difference between the uniformity values obtained using the proof of concept and those obtained by manually computing them is nearly the same. For expert 1, the values resulting from manually computing them and those obtained by use of the PoC are nearly the same. The results obtained by expert 1 therefore support that the PoC is accurate compared to manual use of the uniformity functions, whereas the values obtained by experts 2 and 3 support the contrary. It is therefore hard to give a uniquivocal answer to the research question.

An interesting direction for future work in this case is to have the participants of the experiment write their own prompt to the large language model. In the current experiment setup, the prompts provided to the LLM were written by the author, and they therefore reflect the authors intuition on whether certain concepts, properties, and syntactic structures are similar. If the participants would write their own prompts, the results obtained by manually using the uniformity functions and using the PoC might be more in line, and it might be possible to answer the research question using those results.

7.4.4. Limitation: Lack of Participants

In this subsection, we discuss some limitations of the research performed for this thesis.

Lack of Experiment Participants

A clear limitation of the conducted experiment is the lack of digital forensic experts available for this experiment. Conducting the experiment with more participants would likely result in more unequivocal results about the accuracy of the functions developed in chapter 2.

Incompleteness of Tool Output

A limitation of the way in which uniformity of open-source digital forensic platforms was determined in this thesis is that it relies on the output of using the platform with a specific input. This means that if the input to the platform is incomplete, it might result in a uniformity value that is not in accordance with the value that would have been obtained had the tool been executed on a more complete input. Imagine the case of the running example; if the input that resulted in the output shown in the running

example would not have contained an instance of an Apple Mail database, this output would only have contained traces found for Windows mail, which in turn makes the platform perfectly uniform.

Subjectivity
The uniformity metric relies heavily on the ability to determine whether concepts, properties, and syntactic structures are similar, and we have shown that this in turn relies on problem-specific knowledge. Since both these things can be subjective, the uniformity value resulting from the proof-of-concept is inherently subjective as well, and it is difficult to establish a ground-truth about the accuracy of the developed methods.

8

Conclusion

This thesis aimed to provide the reader with useful information that can help them decide what digital forensic tool is best suited for integration into another digital forensic platform. This was done by addressing three main research questions, and two sub-questions.

8.1. Research Questions

The first research question (RQ-1), ”What open-source digital forensic tools are available?” was addressed in chapter 2. We conducted review of various open-source digital forensic platforms found online. This overview provides an insight into how suitable the platforms are for integration into another platform, by considering both measurable properties such as the number of file types the platform extracts traces from, or when it was last updated, as well as assessments of how well the platforms are documented or how their code is structured.

Our second research question (RQ-2), ”What is the best way to quantify uniformity of a set of semi-structured data?” was explored in chapter 3. We introduced six types of uniformity, and provided formal definitions of functions that can be used to quantify them. We concluded the chapter with an example computation.

The answer to the third research question was provided in the form of a proof-of-concept, which is a software tool that is a concrete implementation of the uniformity functions introduced in chapter 3. We discuss various potential sources of general knowledge and problem-specific knowledge, and conclude that a large language model operating in conjunction with a domain expert is a valid choice for these sources of knowledge in the PoC. Further design choices were discussed, and its usage was described.

The necessary data to answer research questions 2.1, 3.1, and 3.2 is created using an experiment described in chapter 5. In that chapter, the conducted experiment is discussed, and the results are presented. In chapter 7, the results are interpreted and discussed, but it is not possible to draw unequivocal conclusions due to various reasons, among which is a small sample size.

8.2. Future Work

In this section, we briefly discuss potential future work that could be done to further improve the process of quantifying uniformity of semi-structured data.

8.2.1. Prompt Engineering

Although the prompts utilised in the implementation of the PoC have proven to be effective, an interesting direction for further research and experimentation is to have participants in the experiment write their own prompts. This ensures that the problem-specific knowledge of the experts is better conveyed to the large language model, and the results obtained by use of these prompts in the PoC might be more in line with the results obtained when computing the uniformity values manually. The participants could experiment with the following:

Phrasing of Problem-Specific Knowledge As discussed, the prompts incorporate problem-specific knowledge which is used as a frame of reference from which it can be ruled whether concepts are similar. How this knowledge is phrased might be of influence on the quality of the prompt result.

Question Formulation After reading this thesis, the reader probably has a grasp of how vague or subjective the concept of similarity is. Consequently, stating the question of whether things are similar in a way that conveys the domain-experts intention, is difficult. Therefore, it is worth exploring whether phrasing the question of whether things are similar to the large language model can be improved.

8.2.2. Optimising the Number of Queries in the PoC

The proof-of-concept in its current state makes many queries to either the OpenAI API or the domain expert, namely almost quadratic with respect to the number of tables or number of columns. Because rate-limiting is in place in the API, running the proof of concept generally takes a considerable amount of time. Developing a smarter clustering algorithm or using a heuristic to limit the number of queries made, will result in a lower running-time of the proof-of-concept.

8.2.3. Optimising Involvement of Domain-Expert

In the proof-of-concept, a choice has to be made on when to query the domain-expert. This can either be when the large-language non-similarity, similarity, uncertainty, or a combination of those options. Balancing the involvement of the domain-expert with the accuracy of the of the results of the proof-of-concept is important. Investigating the optimal balance between minimising the number of queries for the domain expert and maximising the accuracy of the result can result in an improved proof-of-concept. In this process, it is important to keep in mind that the optimal balance differs per use-case.

Bibliography

[1] Serge Abiteboul. “Querying Semi-Structured Data”. In: *Proc. ICDT 97* (Feb. 1970). DOI: 10.1007/3-540-62222-5_33.

[2] Dilek Basci and Sanjay Misra. “Entropy as a measure of quality of XML schema document.” In: *Int. Arab J. Inf. Technol.* 8.1 (2011), pp. 75–83.

[3] S. Bergamaschi, S. Castano, and M. Vincini. “Semantic integration of semistructured and structured data sources”. In: *SIGMOD Rec.* 28.1 (Mar. 1999), pp. 54–59. ISSN: 0163-5808. DOI: 10.1145/309844.309897. URL: https://doi.org/10.1145/309844.309897.

[4] Brian Carrier. “Defining digital forensic examination and analysis tools”. In: *Digital Investigation* (2002).

[5] S. Castano and V. De Antonellis. “Global viewing of heterogeneous data sources”. In: *IEEE Transactions on Knowledge and Data Engineering* 13.2 (2001), pp. 277–297. DOI: 10.1109/69.917566.

[6] Raul Castro Fernandez et al. “Aurum: A Data Discovery System”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1001–1012. DOI: 10.1109/ICDE.2018.00094.

[7] Bogdan Czejdo, Marek Rusinkiewicz, and David W. Embley. “An approach to schema integration and query formulation in federated database systems”. In: *1987 IEEE Third International Conference on Data Engineering*. 1987, pp. 477–484. DOI: 10.1109/ICDE.1987.7272414.

[8] Sandra G. Hart and Lowell E. Staveland. “Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research”. In: *Human Mental Workload*. Ed. by Peter A. Hancock and Najmedin Meshkati. Vol. 52. Advances in Psychology. North-Holland, 1988, pp. 139–183. DOI: https://doi.org/10.1016/S0166-4115(08)62386-9. URL: https://www.sciencedirect.com/science/article/pii/S0166411508623869.

[9] W. Kim and J. Seo. “Classifying schematic and data heterogeneity in multidatabase systems”. In: *Computer* 24.12 (1991), pp. 12–18. DOI: 10.1109/2.116884.

[10] S. Melnik, H. Garcia-Molina, and E. Rahm. “Similarity flooding: a versatile graph matching algorithm and its application to schema matching”. In: *Proceedings 18th International Conference on Data Engineering*. 2002, pp. 117–128. DOI: 10.1109/ICDE.2002.994702.

[11] Fatemeh Nargesian et al. “Table union search on open data”. In: *Proc. VLDB Endow.* 11.7 (Mar. 2018), pp. 813–825. ISSN: 2150-8097. DOI: 10.14778/3192965.3192973. URL: https://doi.org/10.14778/3192965.3192973.

[12] Natalya F. Noy, AnHai Doan, and Alon Y. Halevy. “Semantic Integration”. In: *AI Magazine* 26.1 (Mar. 2005), p. 7. DOI: 10.1609/aimag.v26i1.1794. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1794.

[13] Masayo Ota et al. “Data-driven domain discovery for structured datasets”. In: *Proc. VLDB Endow.* 13.7 (Mar. 2020), pp. 953–967. ISSN: 2150-8097. DOI: 10.14778/3384345.3384346. URL: https://doi.org/10.14778/3384345.3384346.

[14] Amit Sheth and Vipul Kashyap. “So Far (Schematically) yet So Near (Semantically)”. In: *Interoperable Database Systems (Ds-5)*. Ed. by DAVID K. HSIAO, ERICH J. NEUHOLD, and RON SACKS-DAVIS. IFIP Transactions A: Computer Science and Technology. Amsterdam: North-Holland, 1993, pp. 283–312. ISBN: 978-0-444-89879-1. DOI: https://doi.org/10.1016/B978-0-444-89879-1.50022-1. URL: https://www.sciencedirect.com/science/article/pii/B9780444898791500221.

[15] Ke Wang and Huiqing Liu. “Schema Discovery for Semistructured Data”. In: *Knowledge Discovery and Data Mining*. 1997. URL: https://api.semanticscholar.org/CorpusID:52857898.

[16] Meihui Zhang et al. “Automatic discovery of attributes in relational databases”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’11. Athens, Greece: Association for Computing Machinery, 2011, pp. 109–120. ISBN: 9781450306614. DOI: 10.1145/1989323.1989336. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1989323.1989336.