THE UNIVERSITY
of EDINBURGH

Analysing system behaviour by automatic benchmarking of system-level provenance

Sheung Chi Chan

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2020

Abstract

Provenance is a term originating from the work of art. It aims to provide a chain of information of a piece of arts from its creation to the current status. It records all the historic information relating to this piece of art, including the storage locations, ownership, buying prices, etc. until the current status. It has a very similar definition in data processing and computer science. It is used as the lineage of data in computer science to provide either reproducibility or tracing of activities happening in runtime for a different purpose. Similar to the provenance used in art, provenance used in computer science and data processing field describes how a piece of data was created, passed around, modified, and reached the current state. Also, it provides information on who is responsible for certain activities and other related information. It acts as metadata on components in a computer environment.

As the concept of provenance is to record all related information of some data, the size of provenance itself is generally proportional to the amount of data processing that took place. It generally tends to be a large set of data and is hard to analyse. Also, in the provenance collecting process, not all information is useful for all purposes. For example, if we just want to trace all previous owners of a file, then all the storage location information may be ignored. To capture useful information and without needing to handle a large amount of information, researchers and developers develop different provenance recording tools that only record information needed by particular applications with different means and mechanisms throughout the systems. This action allows a lighter set of information for analysis but it results in non-standard provenance information and general users may not have a clear view on which tools are better for some purposes. For example, if we want to identify if certain action sequences have been performed in a process and who is accountable for these actions for security analysis, we have no idea which tools should be trusted to provide the correct set of information. Also, it is hard to compare the tools as there is not much common standard around.

With the above need in mind, this thesis concentrate on providing an automated system ***ProvMark*** to benchmark the tools. This helps to show the strengths and weaknesses of their provenance results in different scenarios. It also allows tool developers to verify their tools and allows end-users to compare the tools at the same level to choose a suitable one for the purpose. As a whole, the benchmarking based on the expressiveness of the tools on different scenarios shows us the right choice of provenance tools on specific usage.

Acknowledgements

I am extremely grateful to many people for their help in completing my PhD. In particular, I would like to thank the following people:

Thank you to my supervisors James Cheney, David Aspinall and Pramod Bhatotia for their advice, criticism, help, patience, and giving me the chance to work as a PhD under their supervision. Thank you to Ashish Gehani, Hassaan Irshad and researchers in SRI International for the helpful discussions in early PhD. Another thank you to Ashish Gehani and Hassaan Irshad, together with Thomas Pasquier, Margo Seltzer, Lucian Carata, Ripduman Sohan and researchers of SPADE, OPUS and CamFlow for helping me to test the ProvMark tools on their provenance systems and support me in the publication for ProvMark. Thank you to Myrto Arapinis and Ajitha Rajan for providing valuable opinions in the year review meeting. Thank you to Ajitha Rajan and Adam Bates, examiners of my viva, for providing valuable comments and suggestions on my PhD project and thesis.

Thank you to the two Daniel, Marcin, Joseph, Rui, Ghita, Wilmer, Panos, Weili, Pau, Emanuel and the rest of IF 5.24 for helpful discussions in both academic and leisure topics over the years. Thank you to Zui Tao for his opinion on future career path development. A second thank you to David Aspinall for allowing me to gain much teaching experience by working as a teaching assistant under his course each year. And another thank you to Volker Seeker for allowing me to work as a tutor under his Java course.

Thank you to my friends and family members for supporting me over the past years, especially those who spent their holiday coming to Scotland to visit me.

Finally, another thank you to James Cheney, AFOSR and DARPA for choosing me to work in this PhD programme and providing me with scholarship and stipend for the whole period.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Part of the material used for the contributions made by my thesis has been published in the papers listed below. For the ProvMark work, the co-authors other than my supervisors, are the original developers and researchers of the provenance systems that we adopted as the testing target for ProvMark. Their contribution is limited to helping us to test and comment ProvMark behaviour on their tools respectively.

1. **Chan, S. C., Gehani, A., Cheney, J., Sohan, R., & Irshad, H.** (2017). *Expressiveness benchmarking for system-level provenance*. In *Proceedings of the 9th USENIX Workshop on the Theory and Practice of Provenance*. USENIX, 2017.

2. **Chan, S. C., Cheney, J., Bhatotia, P., Pasquier, T., Gehani, A., Irshad, H., Carata, L., & Seltzer, M.** (2019). *ProvMark: A Provenance Expressiveness Benchmarking System*. In *Proceedings of the 20th International Middleware Conference*. ACM, 2019.

3. **Chan, S. C.** (2019). *Analysing system behaviour by automatic benchmarking of system-level provenance*. In *Proceedings of the 20th International Middleware Doctoral Symposium*. ACM, 2019.

4. **Chan, S. C., Cheney, J** (2020). *Flexible graph matching and graph edit distance using answer set programming*. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, Cham, 2020.

5. **Chan, S. C., Cheney, J., Gehani, A., & Irshad, H.** (2020). *Integrity checking and abnormality detection of provenance records*. In *Proceedings of the 12th USENIX Workshop on the Theory and Practice of Provenance*. USENIX, 2020.

(*Sheung Chi Chan*)

To my grandfather.

Table of Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

Provenance is a term originating from the world of art and soon spread to a wide range of usage in different domains. The fundamental concept of provenance is to record all actors' identity and actions on specific artefacts from their generation to the current state. The artefacts can be different when we are referring to different domains. For example, the artefacts can be pieces of artwork where the provenance records the creator, intermediate owners, ownership transfer and selling transactions. The artefacts can also be digital files generated by users and the provenance records all the users who have modified the files and the timestamps of the actions. We can generalize that the content of the provenance is dependent on its field of usage. In this thesis, we concentrate on the field and usage of provenance in data science and its extension to security.

Data provenance is one of the big topics discussed in the research field of data science, especially in some fields that need to handle big data analysis like database and system analysis. The existence of data provenance can help researchers with traceability, accountability, and reproducibility of some runtime actions and activities because it can record all things that happened at runtime and is easily accessible. For example, in Pasquier et al. [130], the authors mention the usage of reproducing and interpreting what is happening in runtime by observing and analysing provenance collected in a runtime session. Besides, the provenance showing the runtime behaviour can help us ensure things are executing as expected, which provides some level of dependability checking for the high-level activities, as suggested by Alvaro and Tymon [8]. In short, data provenance is describing the complete history of the tracing target. It always needs to be collected at runtime to record all the information. Some early provenance system like PASS [117] is designed to do this job in the background transparently and bundled

with existing storage systems. This can ensure provenance is already collected when it is needed later for problem identification or reproducing incidents. In the context of this work, we are motivated by using provenance for security auditing and forensics application. We aim to provide an alternative way for intrusion detection, accountability tracing and fault recovery. Thus we are concentrating on provenance information and systems that are focusing on system-level provenance. Other usages of provenance information like database or data archival are not in our consideration.

To achieve the advantages of data provenance, it is important to ensure the quality of provenance information and decrease the cost for generating, storing and querying it. The quality of provenance is interpreted as the provenance expressiveness while the cost is interpreted by the effectiveness of provenance related processes. The effectiveness of the generation process depends on how the provenance is collected and what information is collected. This topic has already been researched deeply and different automated provenance collecting systems have been produced. There is also research on the effectiveness of the storing and querying of provenance by filtering, organizing and limiting the amount of provenance information stored to speed up the query process. For example, these publications [6, 42, 126] consider provenance result management and automated filtering to limit the amount of provenance stored to the necessary level to increase efficiency.

On the other hand, the problems of provenance expressiveness are related to the goals of provenance collection and are hard to compare across different automated tools with different perspectives. With the effectiveness consideration in mind, a system may filter out some information which may only be necessary for some other uses. For a general end-user, it is hard to understand the expressiveness of the provenance data generated by tools for specific applications. This direction has received very little attention currently and thus this work aims to provide an automated framework on top of some provenance recording tools to compare their expressiveness in different scenarios. We aim to use an automated framework to provide expressiveness benchmarking for the tools in different scenarios to allow users to compare the tools when they need them for specific applications.

The original idea for the automated framework came from an information security perspective. When we try to research some new method of accountability tracking and intrusion detection for forensic and security usage, we have been introduced to the idea of data provenance, which is rather new for security analysis. In the preliminary study, we observed that there are many automated recording tools for provenance and

it is hard to determine which of them will be adequate for a given purpose. This is an important factor because in security applications, the expressiveness of provenance information, together with the integrity of the generated provenance, is needed to provide a chain of evidence for security and forensic usage. Thus we need to have a way to compare and assess the expressiveness of the generated provenance and thus it has become the main focus of this work.

1.1 The ecosystem of data provenance

Since the introduction of the conceptual idea of provenance into the data science field, the research community has started to engage in research of the practical application of data provenance. Researchers defined data provenance as a meta-data that contains information relating to the origin of an artefact (mostly data) and how, why and by whose actions the data attained the present state and location. This collected meta-data provides additional evidence about the operation and processing of data including the accountable parties and source of origin. These data become important when the usage of a data system is scaling up. The existence of provenance data allows users to trace accountable parties, diverts deviation for error searching or re-usability of data. These users benefit from the traceable and reproducible properties of provenance. From reading or querying on result provenance, the same data processing flow can be either traced or reproduced to provide the above information.

As there are increasing needs for data provenance to aid the processing of systems and large scale data sets, researchers are starting to focus on ways to collect this information. In general, collecting provenance is almost the same as monitoring all actions in a system and recording the necessary information as meta-data for future querying of the action. In a current system, many processes are executing at the same time and combined to form high-level actions. Recording these actions and all the related information becomes a troublesome and time/space consuming task. And one question is, how to determine which information is necessary for an application? There is no general agreement to determine what should be kept as the provenance information. Researchers tended to develop provenance recording tools when they had a specific need for a certain type of data or information. For example, the SPADE tool [64] originally developed for provenance auditing in distributed systems and CamFlow tool [129] focuses on security and system auditing in a single system and thus it requires whole system provenance capture which includes all possible activity records. It con-

centrates on the provenance management from distributed hosts to a central server for auditing purposes. As a result, the tools are developed solely for their designated purpose. In general, there is no common standard or agreement on what a piece of data provenance should contain or what information should be kept or dropped in the provenance collecting process.

One of the main problems affecting the efficiency of data provenance applications is the size of the provenance data. The general definition of provenance aims to collect all information that allows complete traceable and reproducible abilities of the same set of data operations. In reality, some research directions did not need this complete set of information. Some of them only take advantage of a subset of information recorded in full provenance trace and does not need the remaining information at all. In a survey paper, Herschel et el. [77] classify some of the provenance systems into different groups based on their applications and source of information. By studying the initiatives of different groups, it demonstrates that some of the groups are targeting on different applications and types of provenance information, thus they are collecting different subsets of the full provenance trace from different components and levels. Some of the systems collect information from the user level and some relay information from the kernel. As a result, the current ecosystem of the data provenance includes many specific tools developed for single or limited purposes and new tools are developed when new needs arise. Although some provenance tools are starting to consider general application targets, the development of provenance systems is still receiving only a little concern in the data science field.

1.2 Challenges and standards

Although there is no general agreement on how provenance is collected and what should be included in the provenance result, there are still some loose standards in the research field. Based on the increasing need to understand and develop a standard of provenance, Moreau [110] summarizes some common applications and definitions of data provenance on the web. The paper gives an overview and foundations of provenance, including some basic definition of provenance and possible workflow for provenance collection and application in databases. It also provides some key properties of data provenance by analysing some of the informal uses of provenance in the field. Moreover, it generalizes the idea and provides an open vision and understanding of how the provenance data should be formalized for use in the web environment.

The authors also influence and contribute to the development of the famous model for data provenance, the Open Provenance Model (OPM) 1.0 [112] and its later updated core specification for version 1.1 [111]. The model provides a great base for the foundational definition of provenance. It also provides a general understanding of how provenance can be used directly or extended to provide different properties and characteristics in different areas.

The Open Provenance Model (OPM) is the first research community-driven standard that aims to propose a formal specification for how provenance should be represented. It provides a set of basic design principles for provenance systems and data provenance researchers to follow. Also, the OPM aims to provide a way of displaying and representing provenance data in a common standard format. After the introduction of the OPM, provenance researchers worked towards a common unified standard for provenance. The unified standard is targeted to support widespread use of provenance that allows different tools to interchange provenance information and understand provenance information generated by other tools. The common standard was developed and named as PROV [156], standardized under W3C and includes multiple documents defining different aspects of the interchangeable and interoperable characteristics of provenance. For example, PROV-DM [22] defines the data model for the PROV standard. With the standard, it is expected that provenance collecting processes can follow and produce interchangeable provenance information that should be readable by all parties that understand the PROV standard. The standard aims not only at the traditional goals of versioning and reproducibility, but it also provides formalizations of other aspects. For example, it defines how to identify an object entity and represent it in the resulting graph. Although the standard provides guidelines on many aspects of provenance collecting, it does not define how a given operating system's behaviour should be recorded.

The existence of the PROV standard provides a general and formal way for provenance collecting and representation. But there is no guarantee that tools or users must follow the standard at all. There are still tools in the wild not fully compatible with the PROV standard. At the end of the day, it is just another community-driven standard and there is no force to push it to mandatory. Thus there are still challenging problems to identify and compare provenance results generated and collected from tools not following the standard. To compare the strengths and weaknesses of different provenance tools in different application scenarios, it is necessary to have an objective benchmarking and comparison. This is one of the major goals of this work.

1.3 Research contribution

The preliminary initiative of this work is the research on new and efficient methods for identifying the existence of sensitive behaviour at runtime and to provide a chain of evidence for the behaviour and to identify the accountable parties of the behaviour. The need for chains of evidence and related accountable parties match closely to what data provenance is recorded. Thus we focused on studying existing provenance tools to see if any of them collect provenance information sufficient for the security and forensic goals. Also, one of the requirements for provenance usage in security is to ensure the integrity of the provenance information collected. This relates to the security of the provenance itself and verifying that the provenance systems did the correct job to collect provenance relating to an event. One of the identification standards is the correctness and completeness of the provenance collected by the tools. If the process is confirmed to be correct and complete, then the provenance collecting process should be able to reproduce the repeating process result with the same set of provenance information if we are executing the same set of activities in the same environment.

Following the above initiative and the necessity to prove the reproducibility of provenance processing, we need to have a standard to measure if the tools match the requirements in security settings. As mentioned, we aim to measure this by identifying the correctness and completeness properties of the provenance information. One of the challenges is how to measure the correctness and completeness and how to compare the provenance results and collecting/generating processes of different tools to each other. This is a big question as we mentioned, although there is a W3C standard of PROV that provides a unified requirement to collect and represent provenance information, it is not a mandatory standard and tools can have a choice to follow it or not. If the tools follow the same standard, it is easy to identify and compare their strength and weakness in terms of correctness and completeness in different applications and environment. Although the common standard does help to provide a more unified result, it sometimes underspecifies some relevant runtime behaviour which makes the resulting provenance incomplete. Also, currently there lack a common agreement that the tools must follow the standard, thus there is a possibility that the target tools are not comparable at all.

This thesis is motivated by the need to have an objective comparison of provenance collecting tools to assess their correctness and completeness. This allows end-users to identify the strengths and weaknesses of different tools for different purposes and in different environments. The thesis concentrate on introducing a fully automated system to provide expressiveness benchmarking. The expressiveness benchmarking is targeting different provenance collecting tools working for different purposes in different environments. The reason for the introduction of the automated approach is because of the tediousness of the manual benchmarking process. Benchmarks in terms of provenance entities are given to show how these tools describe an activity sequence in a specific environment. End-users can compare the benchmarks of each tool with the same set of action sequences. The comparison result demonstrates which of the tools gives the closest and richest information that they need for a specific purpose. This thesis also provides handling of non-deterministic events. The following list shows the research contribution of this work.

Contribution 1: General benchmarking of provenance tools and system calls The automated system mainly concentrates on providing a benchmark in terms of provenance model entities. When we provide a set of action sequences in a given environment, the automated system will monitor different provenance collecting tools to collect provenance on the execution of this action sequence and further process their provenance outputs and generate a benchmark for each of the tools for this specific action sequence. These benchmarks represent what information is collected by those provenance tools for this specific action. End-users can compare the generated benchmark results directly to see which of the tools produce provenance that forms the closest match to their needs. For example, providing a set of actions as input and the resulting benchmark shows the capabilities of the tools in describing different aspects of the actions. Some tools may be describing more about which channels an artefact passed through. Other tools may have more detailed descriptions of the processes handling the artefact and the information of the processes' owners. These examples show the strengths and weaknesses of the tools for different purposes. Besides, as different provenance tools have different purposes and applications, they may source the information from different layers of the operating system and software stack. Some of them may source from the user library in the user layer, others may choose to go down in the stack and source directly from some of

the kernel modules like the Linux Audit System or the Linux Security Modules. Sometimes, it is possible for a provenance system to source from multiple locations. The benchmarking approach of different provenance systems can help to reason about the pros and cons of different information sources and collecting approach and provide an objective analysis of the information gathered from different layers in the operating system.

Contribution 2: General analysis of provenance tools Apart from the consideration of the perspectives of end-users, this automated system can also benefit the tool developers. As we understand, data provenance contains a large amount of information. In general, recording provenance for a few simple actions already generates a very large set of provenance entities and relations. This is because most of the high-level operations in current computers contain a large set of kernel activities including privilege checking, memory exchange, and other calculations. These activities all contribute to the large provenance result. Provenance systems have to handle, generate and process these large amounts information and it is hard to check for bugs and errors in these results. Errors could exist in these systems that may affect the correctness and completeness of the tools. One of the uses of our proposed automated system aims to eliminate some of the unimportant and volatile information (coming from background activities unrelated to the target action sequence) from the provenance and generate a much-simplified benchmark to describe the key action sequence. This smaller size of results is easier for the tools' researchers and developers to use for checking their tools to identify errors. It can also reassure us that the tools meet their expectations in provenance generation. As a whole, the automated approach helps provenance tool developers to evaluate their tools.

Contribution 3: Approximate isomorphic graph comparison One of the major obstacles that automated systems faces are the comparison of graphs. One of the steps for the benchmark generation requires filtering the different graph structure between two provenance graphs. One of the provenance graphs represents the provenance generated for a set of the target execution activities while the other graph represents the provenance generated for a subset of the first set of activities. In this situation, the provenance graph structure representing the common activities in both sets should be isomorphic to each other. The additional graph structure in the first graph can be filtered by first matching the isomorphic struc-

ture among the two graphs. This makes the two graphs an isomorphic sub-graph pair. As an on-going discussion in graph theory, it is generally believed that graph isomorphism testing is not known to be tractable nor NP-complete, while isomorphic sub-graph matching is an NP-complete problem. Indeed, László Babai has produced proofs in [14] to conclude that graph isomorphism has a quasi-polynomial solution. The proofs have not been refuted since then. Thus it shows that these are hard and complex problems to be solved by the automated system. We adopt the edit-distance algorithm to retrieve an approximate set of result to shorten the time needed and decrease the complexity of this problem into an acceptable range. We also make use of Answer Set Programming (which is a form of logic programming) to help to solve the problems.

1.4 Thesis outline

This work mainly concentrates on developing the expressiveness benchmarking framework for provenance generated by different provenance systems. It starts with manual classification and then develops a fully automated framework. It also includes some obstacles overcome when switching the approach from manual to fully automated process. We also add in some additional features to make it more realistic towards live usage. At last, we provide some self-evaluation and testing on the automated framework. The thesis will focus on discussing the full development process and evaluation done on the automated framework and some statistics and data for real testing results. We also include some evaluation of the target provenance recording tools, where the developers have provided us with feedback on the expressiveness benchmarking results. The feedback helps to improve the framework and to make it more useful not only for the end-users but also for the tool developers to evaluate their tools. The following is the structure of the remaining chapters of this thesis.

Chapter 2: Background This chapter includes state-of-the-art for all related research directions. These directions include provenance, security, formal methods, automated graph comparison, answer set programming and analysis of non-determinism. These research directions include technology related to the motivation of our ideas and the design of the automated framework. This information provides a basic understanding and shows the foundations of this work

Chapter 3: Manual analysis of basic system-calls This chapter contains the introduction of System-level provenance benchmarking. It includes details presented in the workshop paper published in TaPP 2017. The context includes the discussion of the preliminary goal and the motivation of the benchmarking approach. It also discusses the importance of the existence of such a benchmark approach for the developers of different provenance collecting mechanisms and how the benchmark may be used in security analysis. Besides, the original idea for unit testing and expressiveness analysis of the different benchmark will be discussed. This chapter explains how the idea of benchmarking is built and how it contributes to provenance and system tracing research. Lastly, the need to make this approach automatic will be discussed.

Chapter 4: Graph isomorphism with answer set programming This chapter introduces property graph comparison by Answer Set Programming (ASP) with the edit distance approach. This is one of the key research questions to solve before we can fully automate the benchmarking approach mentioned in the last chapter. Part of the work has been presented in a workshop paper published in PADL 2020. This chapter illustrates using ASP to support the edit distance and isomorphic graph comparison on property graphs, which are the major types of results generated by provenance recording. The description will stress the importance of this comparison in the full automation of the benchmarking process. Finally, the conversion between ASP graph types to other graph types used by different provenance collecting mechanisms will be discussed. This chapter also shows some ASP code to solve the isomorphic graph matching problem on attributed multigraphs, which directly benefit provenance graph comparison and benchmarking.

Chapter 5: ProvMark: the automated system This chapter includes the description of ProvMark, which is the automated system for provenance expressiveness benchmarking. The context includes the discussion of the full system design and evaluation. It fulfils the needs mentioned in Chapter 3 and demonstrating the need for the graph comparison approach mentioned in Chapter 4. In addition to the system design and evaluation, some of the real provenance benchmark examples and the illustration of each processing subsystem will be included. The

content of this chapter is organized around ProvMark, the automated system for provenance benchmarking, which has been presented in the Middleware Conference 2019. Lastly, some limitations of the ProvMark system and possible extensions of the resulting benchmarks are discussed.

Chapter 6: Non-determinism This chapter discusses how ProvMark extends to benchmark non-deterministic events which have closer proximity to real-life applications. After extending the benchmarking to non-deterministic events, ProvMark can, in theory, identify the unique patterns for each of the system call activities, or system call activity sequences (assuming the non-deterministic events only have limited number of execution combinations). The development and extension allow ProvMark to generate sets of provenance benchmarks describing the same activities from a deterministic or non-deterministic program. The groups of benchmarks can then be compared and analysed to understand the completeness and correctness of each provenance tool. Also, it can discover how non-determinism can affect some of the actions in different execution order. This chapter contains the updated implementation of ProvMark to handle non-deterministic benchmark programs and a more complete evaluation of the ProvMark system and its result regardless of the determinism of an input program.

Chapter 7: Future work and extensions This chapter provides some suggestions of possible directions extended from this thesis, including those future work proposed but not able to complete within the PhD scope. This includes how to use the benchmark for pattern discovery in runtime trace for security purposes, or how to effectively distinguish obfuscation of an event sequence.

Chapter 8: Conclusion This chapter concludes the thesis and summarizes the contribution of this thesis to different fields of research communities.

Chapter 2

Background

This chapter provides a detailed background that describes the foundation for this thesis. Section 2.1 provides a summary of the data provenance research and the state of the art of this area. It includes the usage of provenance and some existing provenance systems. This knowledge provides a basic foundation and overview of the provenance research, supporting further enhancement and contribution in this direction. Section 2.2 describes some existing security-related research, including analysis, modelling and intrusion detection. Also, this section summarizes some of the existing provenance applications in security research. This knowledge aims to motivate the uses of more unified and structural data provenance. As not all provenance systems are following the same set of standards, section 2.3 describes some of the formal modellings of provenance. It aims to build up the foundation for self-evaluation of the completeness and correctness of provenance data. As a result, it helps to describe the activity sequences, and thus motivates the need to have a uniform way to compare the expressiveness of the provenance systems. Section 2.4 describes some hard questions on isomorphic graph comparison in terms of complexity. These questions form the major obstacle on provenance data comparison because most of them use a graph-based description of system event sequences. Section 2.5 describes some existing algorithms or heuristics to solve and approximate hard graph comparison questions, which provides a foundation for the graph comparison component of our automated approach. Finally, section 2.6 describes non-deterministic execution in runtime activity sequences which may affect the provenance collection stage. This also poses one of the obstacles we faced.

2.1 Provenance

2.1.1 Data provenance

Provenance is a broad topic in Information Science. Although it has already been used in different settings such as databases, distributed web, and business auditing, there is not yet a general and unified definition and foundation for provenance. So it is worthwhile to review some of the literature discussing different definitions of provenance and their semantic characteristics, including the details of how to record provenance data, what it is used for, how it is kept and what data should be recorded. Reviewing these ideas around foundations of provenance helps to build some general understanding of provenance and help to develop ideas on how it can link up with the research field of information security or formal modelling.

Provenance is a term used in fine art before applying to Computer and Information Science. It is mainly used to describe the history, ownership details and other annotations of some art objects. In the article Buneman et al. [27] dated back to 2000, the term data provenance is introduced to refer to the tracing and recording process of information flows and origins in database environments. It is the first publication to mention the term "data provenance" to describe the data lineage and data flow in the information science field. Since then, the research community has started to use the term provenance to refer to electronic metadata which records the information flow and data origin in different areas, including the semantic web, information auditing, forensics, etc.

By the late 2000s, there are already many applications of data provenance in the research community. In Ram and Liu [136], the authors summarize a conceptual model, the W7 model, to represent the seven key components of data provenance. This model is the first paper defining a general semantics and structure for data provenance. It considers what data provenance should record to achieve the original purpose of reproducibility and traceability. After that, the two articles Moreau et al. [113] and Cheney et al. [35] summarize the state of art for provenance research. The two papers concentrate on the importance of provenance data in different areas as there are significant increases in electronic data handling. Many of the traditional businesses transfer their physical data handling to electronic systems. This increases the importance of provenance data because it acts as a possible solution to keep track of the origin and change history of the electronic documents and data to provide change logs and to fulfil some security requirements. Cheney et al. [35] also mention that most of the provenance

usage is ad-hoc and it is better to define some formal structure and foundations of provenance to make it more suitable for the growing environment of information and data exchange.

Base on the increasing need to understand and develop foundations of provenance, Moreau [110] summarizes some common applications and definitions of data provenance on the web. The paper gives an overview and foundations of provenance. It proposes some key properties of data provenance by analysing some of the informal uses of provenance in the field. It also generalizes the idea and provides an open vision of how the provenance data should be formalized for use in web environments. The author also influences and contributes to the development of the famous model for data provenance, the Open Provenance Model (OPM) [112, 111], which provides a foundation and understanding of provenance. It also provides a general understanding of how provenance can be used or extended to provide different properties and characteristics in different areas.

The term provenance was initially introduced to database and data storage environment to keep track of the change and flow of data. In Cohen et al. [38], they proposed a formalized way to keep track of the data changes in scientific workflows, which results in an easy understanding of the data change progress in some intermediate steps. This article provides us with a better understanding of the foundational usage and the pros and cons of provenance. In Anand et al. [9] and Chavan et al. [33], they consider provenance in databases with more concentration on the querying and storage of provenance data, including the versioning of provenance and data. When the database is increasing in size and complexity, the provenance data recording the trace and workflow becomes hybrid and nested (recursive) because of increased versioning information for data. These papers provide an understanding of solutions for querying and storing those relational provenance data to let users have a chance to review the data flow easily by using some hybrid queries on the provenance.

Apart from database settings, there is also some usage in other areas. For example, Factor et al. [49] considered the usage of provenance for the long term digital preservation. The authors proposed ways to preserve the provenance and extra information of some digital data storage in the case that technology, stakeholders, formats, and communities are all changed on a large scale. The major task is keeping the data understandable by new groups of people and technology and also to provide trust transfer from the original owner to the new handler which may not have the chance to meet. This direction of provenance application is similar to the one used in fine art which tries

to keep the creator, past owners and many details of the masterpieces to let the current owner and audience understand it more thoroughly. It also helps to identify possible ways of extending provenance to security auditing and forensics which need evidence of traces to prove some sensitive actions occurred or to understand the behaviour of some sensitive actions. Apart from digital preservation purposes, Pasquier et el. [132] proposed other applications of data provenance. They suggest using data provenance for system and firmware auditing which maintains and checks if certain systems or frameworks fulfil the requirements and guideline. This can help to maintain security and specific company regulations to protect the company from data leakage, insider threats or making sure that the company handles data with care and follows the legal requirements like privacy protection.

2.1.2 Provenance systems and tools

Provenance is a set of meta-data as mentioned above. This setting makes the construction and simplifying of provenance data essential before we can analyse or process it. Also, it is not possible to record this information manually by an outsider. For this reason, an automatic tool is required for the recording, transforming and simplifying steps. One example of such a provenance system is mentioned in Gehani and Tariq [64]. The authors proposed a tool named as SPADE for the collection and simplification of provenance data. The tools are proposed based on earlier work by Gehani et al. [65] which shows that policies can help to limit the overhead for collecting and analysing meta-data that allows reproducibility. This tool monitors the necessary system-calls and other important system communication data and tries to transform and plot those collected provenance data as a graph following the Open Provenance Model (OPM). It runs as a black box in the underlying system and provides a provenance graph describing the trace and behaviour of the target systems or applications. It is designed to work in a distributed platform which allows combining and relating the provenance data collected from different devices.

SPADE mentioned above is just one example of existing provenance systems and tools. As we mentioned above, there are many developers that design and implement specific tools and systems for collecting, managing and analysing provenance to support needs such as process reproduction or tracing events. Herschel et al. [77] provides a systematic survey of already published or released provenance systems until the paper publication. It provides a landscape comparison for the characteristics of many

provenance systems in the provenance research communities. They also classify those provenance systems into different categories based on their storage types, source of information and result analysing approach. This paper gives an analysis of the state-of-art of provenance collecting and existing provenance systems and tools. It provides a basic analysis of the tools and shows that different provenance systems have different focus and applications. Thus some of them may adopt different sources of information and resulting provenance format. This survey demonstrates one of the limitations on comparing and analysing provenance systems as some of the provenance results from different systems may not be directly comparable because they are following different provenance standard. Also, because of the different initiatives, these provenance systems may collect different subsets of system execution. Because of this, the result of certain provenance systems may not be suitable to use in other applications. This finding supports our initiative for the expressiveness benchmarking approach to compare the capabilities of provenance systems and the quality of resulting provenance data for different applications. It also supports our choice of candidates with different characteristics as our targets for expressiveness benchmarking.

Data provenance is a broad topic in the research field. In general, the meta-data format and its characteristics allow users to capture, trace and analyse many activities and events for different purposes. As mentioned above, the amount of information involved in system execution is huge, therefore provenance systems always filter information and only collect what they need. It is sometimes not enough to only control the information in the collection stage. The storage and querying of provenance data are also necessary if large amounts of information are needed for some purpose. In this situation, how to manage the collected provenance information to preserve efficiency and correctness becomes another important topic, thus some provenance systems concentrate on provenance storage mechanisms. Lastly, the analysis of provenance results may also require great effort and manual tracing of large data sets is non-effective and error-prone. It is even worse when the data set size is continually increasing. Thus some provenance analysis tools aim not on collections and storage of provenance but concentrate on either auto analysis or visualization of results for easier manual analysis. In Pérez et al. [133], the authors give a systematic review of most of the provenance tools and summary of surveys about provenance and provenance usage. This paper provides a general overview of the usage of data provenance in the research field.

From the above understanding, we can roughly separate existing provenance systems into three main groups. The first group is provenance collecting and recording

tools, which concentrate on the collection, filtering, and recording of provenance information. These tools generally execute at runtime by monitoring real-time events and collect and process provenance information at the same time. The source of information can vary, depending on the usage of the resulting provenance information. In general, they also allow some configuration to filter out necessary provenance information for further analysis or storage purposes. Some example of this kind of provenance tools including PASS [117], Hi-Fi [134], SPADE [64], OPUS [17], LPM [20], Inspector [155] and CamFlow [129], covering a variety of operating systems from Linux and BSD to Android and Windows. These tools collect provenance at different levels, some from the user level and some from kernel modules. Most of them allow simple configuration for filtering and limiting the range of the resulting provenance through some criteria.

The second group of provenance systems is provenance management systems and tools which allow easy and efficient management of provenance data including systematic storage and fast querying and tracing of information from provenance collected. In general, these tools have some specific design to allow more efficient handling of the large amounts of provenance. Some of them are specially designed storage systems and some of them are libraries that can be added on top of the provenance collecting system to allow further processing of the provenance outcome from those tools in the first group. Examples of this group of provenance systems include ProvStore [84], the first systems developed for storing and publishing provenance information online following the W3C PROV standard [156], Core Provenance Library (CPL) [102] and Dataverse [43]. To provide a more standardized and efficient way to allow users to query and post-process provenance, these tools generally follow some existing provenance standard like PROV-DM [22] under the W3C PROV family [156].

The last group of provenance systems is automatic analysers and visualizers. These tools are important because manual effort for analysing a large amount of provenance data is time-consuming and error-prone. Automatic analysis and visualization can help to automate some of the analysis processes or to visualize provenance as a graph to allow easier comparison and analysis. Although it is still a rather new research topic and generating a fully automated analyser is still a future target, these tools already provide great help in the research field. Some notable system in this field includes Prov-O-Viz [78] and Orbiter [101]. Some systems are not limited to the usage in the data provenance field, there include some tools for chemical and biological structure visualization [67] which also suitable to visualize and analyse provenance data.

2.1.3 The three candidates

As mentioned in the above sections, the provenance systems can be divided into three main groups. This work is concentrating on the provenance recording system only, which are the first group mentioned in the last subsection. These provenance recording tools are software systems that provide a broad-spectrum recording service, separate from the monitored applications, and can have their bugs or idiosyncrasies. To rely on them for critical applications such as reproducible research, compliance monitoring, or intrusion detection, we need to understand and validate their behaviour. The strongest form of validation would consist of verifying that the provenance records produced by a system are accurate representations of the actual execution history of the system. However, while there is now some work on formalizing operating system kernels, such as seL4 [89] and HyperKernel [120], there are as yet no complete formal models of mainstream operating systems such as Linux. Developing such a model seems a pre-requisite to fully formalize the accuracy, correctness, or completeness of provenance systems.

This subsection summarizes some background of three of the provenance recording systems (SPADE [64], OPUS [17], and CamFlow [129]) which have been chosen to be candidates for testing our system. Table 2.1 shows the version number for the three provenance systems that we are using in this work. There exist some newer versions of the provenance systems that may behave differently because all the three tools are still expanding and updating after our work has been summarized.

Provenance System	Version Number
SPADE	v2 (Git tag tc-e3)
OPUS	v0.1.0.26 (For Unix)
CamFlow	v0.4.5 (dnf 0.7.6-1)

Table 2.1: Version of Provenance Systems

SPADE's intended use is synthesizing provenance from machines in a distributed system, so it emphasizes relationships between processes and digital objects across distributed hosts. Our analysis uses SPADEv2 (tag *tc-e3*) with the Linux Audit recorder [70], which constructs a provenance graph using information from the Linux audit system (including the Audit service, daemon, and dispatcher). In addition to the Linux Audit recorder, there are also multiple sources of information supported by SPADE, such as Strace reporter that monitors interactions between processes and kernels, Linux

Fuse reporter monitors operations on the file system and LLVM reporter monitors the compilation process. All of these reporters are presented as modules and SPADE allows switching modules to support provenance collection from different sources. We choose to use the Linux Audit recorder which provides detailed and thorough information about all the event and process interactions. SPADE runs primarily in user space and provides several alternative configurations, including filtering and transforming the data, for example, to enable versioning or finer-grained tracking of I/O or network events, or to make use of information in `procfs` to obtain information about processes that were started before SPADE.

As discussed, SPADE provides multiple ways to record provenance by treating different reporters as modules. The modularization of SPADE is not only limited to the reporters. SPADE also allows inserting other types of modules including filters, transformers or storage handlers. All of these modules affect the processing of SPADE and allow users to choose their combinations to meet their needs for collecting provenance. Users can also write custom modules for some specific applications such as targeting sources of information to a certain range or outputting the provenance in a customized format. But in this work, we are using the baseline configuration to demonstrate and test our automated benchmarking, so the resulting benchmark of provenance generated by SPADE should only represent the baseline of SPADE and the behaviour may be different if other modules are chosen in the SPADE configuration.

OPUS focuses on file system operations, attempting to abstract such operations and make the provenance collection process portable. There are two versions of OPUS which collect provenance from two different sources. For older version of OPUS, it works in the user level only. It wraps standard C library calls and adds activities to record the call before following the general library execution flow. Most C library calls perform a combination of kernel system-calls. The developers of OPUS have mapped most of the C library calls to system-call combinations, based on the PVM model[1,2]. When a C library call passes through the additional provenance collection layer, OPUS can record the corresponding system-calls in the resulting provenance graph without the need to monitor the underlying kernel operations. OPUS makes use of the ability offered in some operating systems (such as Linux) to alter the dynamic library linking process (i.e. `LD_PRELOAD`), which overrides the Global Offset Table (`GOT`) in the binary memory and points it to wrapped versions of library calls. Each wrapped call keeps

[1] http://www.cl.cam.ac.uk/research/dtg/fresco/opus/pvm.pdf

[2] http://www.cl.cam.ac.uk/research/dtg/fresco/opus/posix-pvm.pdf

the original call untouched and unaltered. It only provides an additional layer to record information of the original call before passing it to the lower level. Thus, in a sense, the additional layer acts as a 'man in the middle' able to intercept C library calls and pass on the requests to the original library. Because it needs to wrap around libraries and make use of the dynamic library linking process to hook those wrapped libraries to binaries, it will not work on statically linked binaries which skip the dynamic library linking process.

The authors and developers of OPUS discussed the advantages and disadvantages of collecting provenance through system and library call interception in a later publication [16]. Although it is lightweight to collect provenance in the user-space because no additional privileges are needed, it also introduces vulnerabilities for adversaries. One of the big disadvantages the authors mentioned is that adversaries have the same level of privilege to alter the provenance information and the target binaries. This makes the provenance information less useful in a security perspective because an attacker can alter both the binaries and the provenance information to hide its existence. In this kind of situation, to show the existence of the adversaries, the provenance itself needs to be secured and separated from the target execution. They first propose the use of sandboxing for the separation. Later, they develop a new version of OPUS that uses a different source of information for provenance collection. In the new version of OPUS, the intrusive way for provenance collection is dropped. Instead, it relies on the kernel module `dtrace` for collecting provenance information. This new version makes OPUS source information directly in the kernel and allow it to work under both statically and dynamically linked binaries.

The newer version of OPUS is only released after the major work of this thesis has been completed. Thus, all of the experiments and work presented in later chapters are based on the old version which collects provenance from the extra layers inserted in wrapped C libraries. The OPUS system is especially concerned with versioning support and proposes a Provenance Versioning Model, analogous to models previously introduced in the context of PASS [116] and later SPADE [65].

CamFlow's emphasis is sustainability through modularity, interfacing with the kernel via the Linux Security Module (LSM). The LSM hooks capture provenance, but then dispatch it to userspace, via `relayfs` [164] (a module relaying collected provenance from LSM hooks to the CamFlow daemon running in user space), for further processing. It strives for completeness and has its root in Information Flow Control systems [131]. By default, CamFlow captures all system activity visible to LSM and

relates different executions to form a single provenance graph; as we shall see, this leads to some complications for repeatable benchmarking. SPADE can also be configured to support similar behaviour, while CamFlow can also be used as a reporter module for SPADE. Compared to SPADE and OPUS, which both run primarily in user space, CamFlow [129] monitors activity and generates the provenance graph from inside the kernel, via LSM and NetFilter hooks. This means the correctness of the provenance data depends on the LSM operation. As the rules are set directly on the LSM hooks themselves, which are already intended to monitor all security-sensitive operations, CamFlow can monitor and/or record all sensitive operations. CamFlow allows users to set filtering rules when collecting provenance. CamFlow falls into the observed provenance [25] category, incurs minimal overhead and is not intrusive to any userspace object and linking like OPUS is. CamFlow's provenance capture does not require trust in user space applications like SPADE trusting on the Audit Daemon, and therefore provides stronger security guarantees. The provenance captured in the kernel is made available to user-space applications through `relayfs` [164] pseudo files. The default CamFlow installation comes with `camflowd`, a service that performs serialisation to a W3C PROV format (PROV-JSON), and other services can be implemented, for example, to perform intrusion detection [73].

2.2 Provenance and security

2.2.1 Security analysis and intrusion detection

One of the objectives of this work is to provide new alternatives for security analysis and intrusion detection. Thus, it is necessary to go through some of the work that has been done in this direction and also summarizes the state of the art for this security research to provide an overview and understanding how provenance can help in security analysis. Besides, it also presents overviews on how we could identify if provenance generated by these provenance systems is sufficient for security usage. In the book Chess and West [36], the authors summarize foundational requirements for security property analysis, which gives a broad overview of the target for this research field.

Security analysis is an important research topic. Researchers are working on different parts of the analysis. Some of them try to analyse the problems in existing security mechanisms. Some of them try to analyse the properties, behaviours, and characteristics of adversaries, such as hackers or malware attack patterns. Some of the

researchers even concentrate on analysing malicious behaviour at a lower level, such as kernel analysis. Some of these research directions relate to patterns and behaviours, which is similar to the reproducibility, traceability and accountability properties that are provided by provenance. This shows the need to study some of the existing security analyses to understand the pros and cons of existing methods.

Providing security to different systems is not an easy task. Research in this direction is an ongoing work. In Sabelfeld and Myers [146], the authors provide a survey on the security challenges and existing solutions for information flow policies. They also discuss the problems and open challenges on using language-based information flow policies to secure the end-to-end communications and provide assurance on top of some security properties like confidentiality. Nowadays, systems are starting to rely on some of the cloud services, like distributed systems and Software-as-a-Service (SaaS) consideration. In Ali et al. [5], the author studies the security challenges and vulnerabilities in the cloud platform, together with some existing security measures and the problems on those measures. These surveys and summary provide a brief explanation on some early security analysis and intrusion detection work before adopting provenance to security analysis and further help justifying the need for an alternative way for security research with provenance support.

One of the goals of this work is to vet the existence of certain malicious or sensitive behaviour and trace its accountable parties under different circumstances. Adversaries are always enhancing the technique in the war between security defenders. They continue to develop malware with different kind of evasion and obfuscation techniques, trying to get past the security boundaries. Thus studying some of the existing malware can help to consolidate possible defences and come up with ideas on how to discover their existence and provide evidence for later forensic usage. By studying the attack patterns and techniques of malware, it helps to provide understanding on which specific actions or activity sequences should be noticed in the security analysis. In You and Kim [163], Marpaung et al. [104] and Saeed et al. [147], the authors summarize some of the popular attack and evasion techniques used by certain malware to avoid being noticed by users. This knowledge also provides a reason for using provenance in vetting malicious and obfuscated actions and tracing the initiator and accountable parties. The main reason for that is provenance information traces all the changes back to the originator and some obfuscation in the middle is also included in the activity trace and will not stop or affect what is already recorded. Also, linkage of activities and actions are still preserved to allow reproducibility and traceability, thus it once again

provides a rationale for introducing provenance into security analysis as an alternative.

Lastly, the study of the state of the art for existing malware and security analysis research provides an understanding of the pros and cons of current approaches. This helps to motivate why this work proposes a way for users to analyse which provenance systems can help in security analysis, chain of evidence generation and accountability tracing. One of the hard problems in the analysis and discovery of malware or malicious action is how to determine if a certain action is malicious. In Yan and Yin [161], Nari and Ghorbani [119] and Egele et al. [48], the authors summarize different kinds of identification and analysis that discover malicious actions and identify malware in different platforms. They analyse and summarize the identification methods for common malicious actions and provide a set of attack patterns and signatures for further analysis and identification. The suggestion of using provenance to vet malicious activities and trace their initiator is an example of behavioural pattern matching through some machine learning techniques.

There are also some other automated malware analysis approaches using the same technique but are not obtaining the patterns from provenance. They instead obtain and learn the patterns by other means. In Forrest et al. [52], the authors introduce an approach to source information from the audit stream to generate a database of normal system call activities. Then they use those normal records for abnormal activities and intrusion detection purposes. Later paper Wager and Soto [157] studies a very similar approach and point out that the short time window is a big limitation for doing real-time abnormal and intrusion detection as the introspection with the normal behaviour may not process fast enough in the short time required by a real-time intrusion detection system. These approach analyses use certain intermediate data storage to analyse live stream data which may be enhanced by provenance-based intrusion detection which has better management of data and information flow to allow efficient analysis of the stream for abnormal behaviour. This supports the need and reasoning for using provenance-based intrusion and abnormal detection. On the other hand, Firdausi et al. [51] analyse some of the machine learning techniques for behavioural pattern matching malware detection and compare different techniques and sources of information for this approach. They claimed that most of the existing approaches have a high rate in detecting malware, but they did not mention providing a chain of evidence to prove the existence of attacks nor the trace for accountable parties and initiators of such attacks or malware. Thus it left a gap for us to present provenance as an alternative way for accountability and traceability purposes.

With different existing machine learning techniques and behavioural analyses, there are some automated systems for malware analysis, identification, and classification. In Aafer et al. [1], they propose a machine learning approach on the Java API running on top of Java virtual machine, while Isohara et al. [85] propose a kernel-level analysis which does machine learning and behaviour pattern matching on low-level activities in the kernel. In Mohaisen et al. [108], the authors propose a full system named as AMAL that consists of two components that use automated behavioural analysis to identify malware and label it for further investigation and classification later. Their system generates the behavioural patterns for malware by comparing malware sample binaries in the same family and deduce similar patterns by binary analysis. Apart from obtaining behavioural patterns of malware by static comparison of binaries, Reina et al. [140] and Hay et al. [76] propose alternative ways to obtain the behavioural patterns. They suggest monitoring the system-calls and process communications to receive information about the behaviour patterns of certain malicious action sequences. These patterns are deduced by real malware execution in sandboxes. Summarizing most of the analysis research of malware, there is already much research providing a very high accuracy on identifying the existence of malware or a piece of malicious code. There is less research considering the chain of evidence and tracing for accountability, which again provides motivation for using provenance to support this work and thus initiating the need to benchmark the provenance system to identify which of them are more suitable for this job, which is the main contribution of this work.

2.2.2 Security application of provenance

Data provenance is useful in collecting meta-data from runtime execution for reproducibility and traceability purposes. These characteristics are useful for debugging and incident handling. As one of the motivations for this work, we want to find a way to identify the existence of certain activity sequences at runtime and to provide a chain of evidence for them and identify the accountable parties for the sequences. In recent years, researchers have started to make use of data provenance in aid of protection and analysis of security properties on different platforms, especially in traditional systems, distributed and mobile environments. In general, data provenance can help to identify the existence of certain malicious actions, providing referencing patterns for certain actions for later static and dynamic analysis and also detecting integrity violations of underlying data by recording all alterations of the data and related accountable par-

ties for that actions. In some cases, the provenance of provenance information may be required to ensure the integrity of the underlying provenance describing runtime executions.

Existing papers in recent years start to consider adopting data provenance into the above-mentioned security related usage. King and Chen [88] is one of the first papers mentioning the use of the provenance-based approach for security and intrusion detection. They propose a tool named *BackTracker* to automatically read the sequences of system call activities and detect an abnormal sequence of activities. Although they did not mention the term provenance directly, their application of those system call sequences can be classified as a kind of provenance information which make their approach as one of the early research and foundation on provenance-based security. Later in Husted et al. [82], the authors consider using provenance on the developer side as a debugging reference for some security features. They propose additional actions in the mobile application profiling process to collect traces describing the system resource usage and accountable parties. This information assists the developer to identify misuse of system-calls that may open memory leakage, disorders or security loopholes for other adversaries. In Backes et al. [46] and Dietz et al. [15], the authors consider using provenance to replace the faulty Android Binder in handling permission decisions for inter-component communications. In older versions of Android, the Android Binder just recorded the caller of certain actions and thus can only verify if this caller is permitted for certain sensitive action. If this call is instead initiated by adversaries through a privileged caller, the security is broken. Introducing provenance in this scenario allows Android Binder to trace back to the initiator of certain actions to avoid privilege transfer through a faulty Android Binder. In Yang et al. [162], the authors first provided a provenance pattern library that contains mappings between provenance patterns and their related sensitive actions and system-calls in the kernel. They also proposed an automated system that collects provenance patterns from a target mobile application and uses static analysis to determine if it contains any pre-identified sensitive actions by comparing the pattern of the mobile application with the library.

The use of data provenance for security is not limited to the mobile platform. There are also applications in general operating systems. For example, in Han et al. [72], the authors suggested using data provenance to detect application anomalies and to perform intrusion detection. Their system makes use of data provenance information to model and records legitimate provenance patterns by executing some assumed normal training examples of activity sequences. Then these legitimate models are used as a

control to compare unknown executions to detect abnormal actions. This is an example of data provenance usage on security or fault detection over cloud service platforms. Similar concepts for intrusion detection in other platforms using data provenance have been discussed by the same author in Han et al. [73] which also emphasize the challenges for such an approach. Authors of Hassan et al. [75] also suggested some analysis of provenance information for malicious action detection and auditing purposes. They suggest adopting the Deterministic Finite Automata (DFA) approach to build-up models for legitimate and malicious activities for later process. The authors continue to work on this approach and build up a tool NoDoze [74] to generate abnormal scores for the build-up provenance models on abnormal activities. This approach makes use of a novel network diffusions model and aims to decrease false alarm rate for filtering out similar but normal activities sequences. This help to decrease the overhead for abnormality detection. Besides, authors of Berrada and Cheney [24] suggests some ranking techniques for identifying provenance graphs which are better in describing the abnormality, which also helps to increase the performance for malicious action detection. In later publication Han et al. [71], the authors introduce a tool UNICORN for Advanced Persistent Threats which are difficult to detect because of its minimized and slow attack patterns that are possible to span over a long period. Some existing system may already run out of memory and execution time before the full attack has been completed. The authors adopt a graph sketching technique with the support of a novel modelling approach to aid the runtime discovery of these attacks with high accuracy and performance. Comparatively, earlier literature Manzoor et al. [103] also suggests another kind of heterogeneous graph steaming to allow fast and memory-efficient abnormality detection for advanced persistent threats. Their proposed tool SteamSpot are mostly concerning in the analysis of heterogeneous graphs by introducing a new similarity function to vet the existence of certain low and slow abnormality, their approach is to discover possibly abnormality offline rather than the Han et al. [71] runtime detection approach. Also, the literature Milajerdi et al. [107] also suggests a tool HOLMES for real-time advance persistent threats detection. They use another approach by generating a simplified digital signal to represent activities happening in real-time and compare suspicious information flows with known threats signal to discover possible abnormal information flow with close correlation to the known malicious flow. To aid the analysts, their tool can also generate high-level graphs summarizing the malicious flow of activities. The above-mentioned intrusion detection and advance persistent threats discovery make use of different kind of provenance graphs for history and ac-

tivity flow tracking for security analysis and intrusion detection purposes. They summarize the state-of-the-art of recent research on provenance-based intrusion detection and security analysis.

In Tan et al. [153], the authors discuss the need for complete and fool-proof provenance if it is used in systems and network security. They derive a set of minimal requirements for identifying the existence of security problems and to trace accountable parties across a distributed environment. It is an important consideration in the security and forensics field which needs a complete chain of evidence for some incidents and all accountable parties for some security violations. From the viewpoint of security incident handling and forensics, it is important to keep the whole trace from the initiator to the actual point of security violation as any broken steps in the evidence could fail to prove the accountability of certain parties in court or act as a model intrusion detection source for future attacks. In Alrajeh et al. [7], the authors provide minimum evidence preservation requirements for further digital forensic investigation. The requirements consider what data should be collected for such purposes in both a local and distributed environment. This can be translated to requirements for the data provenance for forensic and security usage. The usage of data provenance can apply to security settings and provide evidence in digital forensic investigation. More examples for providing reference and evidence traces for auditing and forensic usage are summarized in Wang and Daniels [158], Pasquale et al. [128] and Zhu et al. [166] which also show that the traceability, dependability and reproducibility characteristics of data provenance can greatly support forensic usage by providing a chain of evidence and accountable tracing functionality. In Zhou et al. [165] and Lee et al. [93], the authors also summarized some uses of data provenance for tracing what has happened and who is accountable for certain actions in a network and distributed environment. Anderson and Cheney [10] propose the use of data provenance to ensure the correctness and completeness of configuration languages across distributed platforms. As configuration languages are used for mass configuration of many machines and components across a distributed platform, there exists a chance that the configuration fails or errors in some of the hosts. These problems may lead to broken access control or opening security problems in separate hosts. The authors propose using data provenance to query and validate the configuration process in the hosts to ensure correctness and completeness of the process. This may limit the security problems resulting from accidental misconfigurations. As a whole, these papers show how data provenance helps to answer and prove what is happening in runtime across the network and distributed environments,

to find potential problems and provide a chain of evidence for the existence of certain events, together with the accountable parties of certain events.

There are some other security applications making use of data provenance. In Bates et al. [18] and Ramane et al. [137] the authors proposed using data provenance as a policy for access control purposes in cloud environments. They build up policies of legitimate actions and user data access using provenance. These policies match the actions of the user to identify abnormal behaviour, enforce access control, and deny abnormal behaviour. It also asks users to provide a provenance trace of an action signed by an authority before completing the actions. The authorities are responsible to verify if the activity sequences recorded by the provenance information are legitimate following the predefined policies and sign it only if those actions are allowed. This can help provide policy-based access control through signature verification on signed provenance information. In this case, data provenance is used as part of a digital certificate for access control or data usage purposes. Thus, the security of provenance itself also needs to be protected to ensure the integrity of the whole access control model. These applications treat provenance as a kind of secure token and policies for protecting security in the underlying layers.

2.2.3 Provenance for algorithm validation

One of the initiatives of this work is to provide the chain of evidence and accountability tracing for forensic and auditing applications. In the research field for digital forensics, different researchers have proposed different kind of approaches and algorithms to retrieve activities trace and runtime properties from the operating system. To make use of those retrieved information for forensic and audit usage, those data are required to be complete and correct to main the validity and authority of the chain of evidence. As the full trace of runtime activities and all its related properties and meta-data is very large in size and may include a lot of non-related information, a certain level of abstraction and optimization is required to retrieve the necessary information for the application. Provenance post-processing by some of the provenance system may help in the reduction of information by capturing only necessary information from a specific source like the Linux Audit System. Other provenance systems may do some optimization and reduction on the information retrieved and generate provenance graphs to tell the story of runtime execution sessions. One of our initiatives is to provide a comparison of provenance generated by different provenance system in terms of completeness and

correctness, it is also possible to compare the result processed by different optimization and reduction algorithm which is used for handling of large scale provenance information and system information. These provenance and system information may be used for forensic and audit application which requires a certain level of completeness and correctness, or in other words, unambiguous and accurate data for further forensic purpose. In this subsection, we are summarizing certain algorithms and techniques that are used for provenance and system information optimization and reduction that are requiring completeness and correctness validity and act as a possible target user group for the work done in this thesis.

System-level audit logs system like Linux Audit System captures the interaction between applications, communication channels and the runtime environment. It provides a good source of information for both provenance tracking and forensic auditing because it could help to trace the accountable parties and the root cause of certain activity sequences. By analysing its integrity, it is also possible to discover possible removal and coverage of malicious activities which the adversaries purposely remove certain log record to cover their traces. One of the common problems for the direct analysis of the audit log has been mentioned above, that is the sheer size of log record generated. It can go up to Gigabytes per day for a busy system. To reduce the amount of size, certain provenance system configures some filters and choose only part of the information for provenance generation. Indeed, the operating system itself has also consist of certain garbage collection features to remove the oldest record to save spaces for new log records. The garbage collection may remove certain old records and may reduce the backtrace period of historic events. In Lee et al. [94], the authors proposed LogGC, an audit log system with special garbage collection service which improves the effectiveness of the audit system by minimizing certain log record to allow more space for newer record and increase the length validity of the log record. This allows possible audit log user to have the ability to trace back to more historic execution trace for forensics and security analysis purposes. Their approach makes use of certain data reduction by analysing certain false dependencies in the system calls relationship. This is a kind of practical application of certain big data reduction techniques.

In later literature Xu et al. [160], Tang et al. [154], Ben et al. [23] and Hossain et al. [80], the authors also propose certain algorithms to reduce the size of the big data such as audit logs and system information traces for forensics and security analysis. The authors of Xu et al. [160] proposes an aggregation algorithm to preserve data dependency during the aggressive data reduction process. They claim that their approach

can significantly reduce the size of the data log records while preserving high-quality forensics values. The authors of Tang et al. [154] proposes a different approach by using access pattern templates to identify certain data for removal to reduce the data size. Their template is built to identify some patterns which are unrelated to certain sensitive activities. They combine their findings of the template-based data reduction and propose a NodeMerge tool for the automation of the process. Nonetheless, the authors of Ben et al. [23] proposes a tool T-Tracker which compress the audit log for intrusion detection. With an assumption that all intrusion is coming from external sources, the T-Tracker builds up a taint model with all external communicating activities and system calls. Then it follows the path of execution from those sources to extend the taint model. In theory, when the whole taint model is built, all activities related to the external source should be covered and those are the only audit log to be analysed for the intrusion detection process. Thus the remaining activities audit log is considered as unrelated and removed to reduce the size of audit logs for analysis. This approach aims to reduce the size of the target audit log needed to be analysed by filtering out mostly unrelated audit log. Lastly, the authors of Hossain et al. [80] formalize the notion for data reduction while preserving data dependency. Their formalization provides certain proves for their novel algorithms to optimize and reduce the size of the audit log. The above approaches introduce different kinds of data reduction algorithms and techniques to allow better performance and larger data storage size for audit log system. As these approaches have done certain post-process to the raw audit data, validation may be required before trusting the result information for forensics and security analysis which require high-quality data.

In addition to certain reductions on the audit log records generated by the operating system audit system, there are also other ways to limit the size and overhead of the audit log record. Ma et al. [99] proposes an in-kernel log reduction system that aims to reduce redundant events and log record generated by the audit system. It makes use of a multi kernel-layer caching scheme to reduce the overhead of log transferring and processing activities, it also makes use of the cache to reduce the need for certain cache writing processes. Their cache system reduces the storage and runtime overhead as a whole for the audit system and its related process. The authors of Milajerdi et al. [107] use a different approach to optimize the information flow record by generating high-level graphs to summarise adversaries action. Their approach aims to decrease the false alarm rate by reducing and ignoring some of the redundant events by comparing each event flow with a correlation of known adversary activity flow patterns. This approach

aims to decrease the false-positive rate of similar activity flow, it also decreases the need to handle large amounts of similar but unrelated false events.

Besides the direct handling of the system audit logging features, there are also some researchers working on the use of provenance for the handling of large storage and execution overhead of audit log recording process. In Hassan et al. [75], the authors propose the tool Winnower that aims to eliminate the needs to transfer redundant audit log generated from different hosts across a distributed environment. They make use of provenance graphs to describe communications and other related data of individual nodes and generate a behaviour model of the host by a novel grammatical inference learning technique to reduce the size and overhead of information between distributed hosts and the central monitoring unit. The central monitoring unit can interpret those grammatical inferences back into a provenance graph and generate a full provenance graph describing different host activities. This approach helps to decrease the need for large audit log record transfer between distributed hosts. On the contrary, the author of Bates et al. [19] introduces a notion of minimal completeness and implements a system with this property by exclusively collecting provenance for a target application. They apply a policy-based provenance filtering to allow the exclusive collection of provenance. Their approach aims to identify and isolate sub-domains to limit the provenance collection to the domains of the target application which allows limiting unrelated information of a target application. This approach can help to reduce the size of the provenance generated for analysis and maintain the completeness of the provenance data for a certain additional application like forensics auditing. Lastly, Ma et al. [100] propose a tool ProTracer which also aims to reduce space and runtime overhead of the audit log and provenance tracing activities. Their tool ProTracer works as a lightweight provenance tracing system that alternate between the audit logging processes and the unit level taint propagation. It will help limit the number of audit logging processes needed by studying the actual tainting propagation and only record system log following the propagation and ignore other non-related log recordings. This kind of audit logging reduction reduces the size and runtime overhead of audit logging system and still preserve the record of the tainted activities relating the adversary attacks and possible intrusion activities.

The above-mentioned literature is some state-of-art algorithms and techniques that aim to reduce and optimize the audit log recording processes to reduce the overhead in storage size and execution time. These approaches do need a certain level of validation and evaluation because they modify the original audit log system or the raw data

generated by them. The validation and evaluation are mainly used to preserve the high quality of data, in terms of the completeness and correctness of the data in describing the real system activities. These requirements of high quality of data are necessary if that information is used for forensics or security analysis which require a high level of quality and chain of evidence. The possible provenance benchmarking approach proposed in this thesis can provide help in the validation and comparison of the quality of those data, which make those optimization and reduction approach become potential users of our approach.

2.3 Provenance and formal model

2.3.1 Formal modelling

Process calculus is a formal modelling approach commonly used in data science, programming language and systems research. It provides an automated symbolic way to define process communication and message or resources exchange in a concurrent or distributed environment. When we are mostly concentrating on provenance systems that are working in different operating systems with many process communications, it fulfils the characteristics of process calculus that make it a suitable formal modelling approach in analysing the behaviour of systems, which is one of the major purposes of most of the provenance systems that we are concerned with. Besides, formal modelling provides references for analysing and verifying if certain security requirements have been enforced in different systems and environment. This additional use provides an alternative for security analysis and testing statically.

The book Sangiorgi and Walker [148] summarizes the method and transformation from raw system behaviour of mobile platforms to a formal modelling language. It also provides some foundational semantic definitions of different scenarios and some reasoning and verification technique on the language. This can help to provide formal modelling on applications and binaries for security analysis. The analysis includes identifying and discovering of target malicious actions by comparing malicious patterns in process calculus with new execution provenance trace. Also, it helps to identify and trace the accountable parties of some target actions. More examples can be found in Chaudhuri [32], Shin et al. [149] and Armando et al. [11] in which the authors make use of formal modelling on the mobile platform (especially Android) to formally define the security properties and working behaviour of some of the components, in-

cluding security sandbox, Android Security Framework, and manifest privilege handling mechanisms. These formal models help to provide a basis for static analysis on the mobile platforms to determine if possible security violations or loopholes exist and if some sensitive or malicious actions can pass without notice.

Process calculus is also a major approach used for the formal modelling of system behaviour in a distributed and cloud environment. Reviewing works of literature in this topic can provide a foundational understanding of process calculus. It also helps to summarize what qualities are needed for provenance information uses in different purpose which provides a reference for the analysing criteria for the expressiveness benchmarking of the provenance information and the provenance systems working in the background.

2.3.2 Formal modelling of provenance

The main direction of our research concentrates on the foundations of provenance and the analysis and comparison of provenance to determine if some provenance tools are suitable for a specific task, especially in security analysis and intrusion detection which requires completeness and correctness of data provenance. This part describes some existing formal modelling of provenance which helps to consolidate the needed criteria for deciding if the provenance and the provenance systems are suitable for security usage. It also helps to understand how some properties of the resulting provenance can be analysed formally in the resulting benchmark and self-evaluation of the result.

Zhou et al. [165] proposed a formal modelling technique for provenance to explain the trace of data changes in a network environment, or in other words, why the data is in its current state and what process the data has gone through to reach the current state. In this case, the provenance is used as evidence for the data alteration history in a network environment with adversaries. So the integrity of the provenance needs to be secured to provide a correct trace to explain the needed information about the data. This is a good example of how formal modelling techniques can help to protect the security properties of provenance itself to prove some security properties of data traces.

While Zhou et al. [165] proposed a formal modelling technique for provenance data in a real network environment, their work shows the need for a formal modelling and reasoning technique to prove the security properties of the provenance data itself, because of the benefit provided by a correct trace of provenance, and also because of

the possible leakage of original data or information through provenance, which is the privacy issue.

Cheney [34] and Acar et al. [4] propose a different formal framework and modelling technique for provenance data. They are aiming to use mathematical approaches to prove some security properties of the provenance data, like partial confidentiality and availability to check if the provenance data leaks some secret information about the original data, or hides some public data that should be available. This direction defines a better understanding and consolidation of meaningful provenance-tracking mechanisms which may help to formalize what, where and how the provenance should be recorded and what security properties are needed for protecting provenance in different situations. Also, the mathematical approach helps to provide ways to formally analyse if the mechanisms are good enough to record provenance data correctly and securely for further use, without raising new security problems. Before these two papers, Souilah et al. [151] also provides a formal model of provenance for distributed systems. This approach uses Pi-calculus as the base to understand and analyse the system data flow with a formal tracking semantics. It aims to ensure and prove that all data changes and data flows have been recorded for all computational processes, including communication among distributed servers. This literature describes and illustrates some semantic properties of provenance data which help to build up understanding and formal foundations of provenance used later in the research community.

2.4 Graph comparison

As mentioned above, provenance is a kind of meta-data to record how the current state of the systems or files was reached and who is responsible for it. It records all the information about actions and actors throughout the execution period and allows a user to trace back what is happening and who is responsible for certain runtime activities. As provenance allows reproducibility and traceability of the execution, the order of the execution trace events must be presented. The most suitable way to describe this information with precedence and order requirement is using a directed graph with attribution. That is also the reason why most of the provenance systems support graph representations for the provenance results. Graph representations of provenance information not only allows preserving event order and properties, but it also allows easy interpretations by users. These graphs show and link the orders of events and actors' attributes by directed edges between nodes which represent the state of certain files or

processes. As graphs are an important representation for provenance, we need to have some understanding of basic graph matching and comparison. We want to match and compare provenance graphs generated by different provenance systems to benchmark the expressiveness of the system itself and the provenance generated by these systems. This subsection aims to study some existing graph comparison problems and algorithms that can be used or extended in the expressiveness benchmarking of provenance and their generating systems.

2.4.1 Definition of graph isomorphism problems

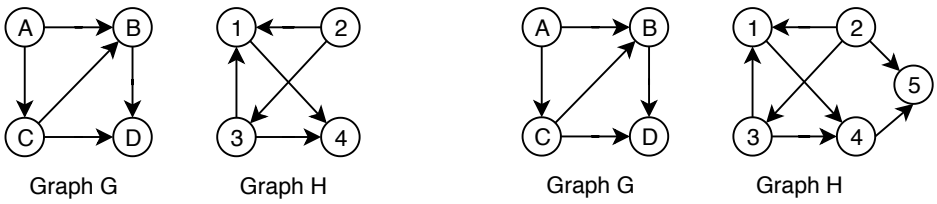
In most operating systems, if we start executing some of the binaries, there are always some kernel activities done before and after the target executions which may or may not be related to the binaries or applications. In most cases, provenance systems do not have the ability to cleanly identify a portion of a process execution which refers to non-related activities from the activities of the target executions. These non-related activities may be related to certain preparation and clean-up work is done before and after the target activities by the same process. Thus graph comparison can be applied to filter out those non-related activities to avoid obfuscating the target execution patterns. Another important use of provenance graph comparisons is to identify similarities and differences among the generated provenance graphs. This can help to debug and profile the provenance systems which generate the graphs. Also, it can help to understand the capabilities of the provenance systems for different applications. Besides, we may also classify and cluster the systems with similar capabilities by clustering their generated graphs. The classification and clustering are also based on the similarity of the graph which is also related to graph comparison problems. In general, these problems consist of matching and comparing graphs with partially similar structure, which is named as (sub)graph isomorphism matching and comparing problem.

For most provenance systems, the generated provenance graph is a directed multigraph which allows multiple directed edges from the same source and destination vertexes. This allows the provenance system to efficiently represents continuing communication between processes or artefacts. All the graphs discussed in this thesis is assumed to be a multigraph unless otherwise specified. Also, there is a label for each vertex and edge to identify the type of items that the graph elements represent. In general, the vertexes and edges in a directed multigraph are identified by a set of unique identifiers and labels. The definition of a simple directed multigraph is shown below.

If G is a simple directed multigraph, then
$G = (V, E, src, tgt, lab)$ satisfying:

- V is the set of vertex identifiers of G
- E is the set of edge identifiers of G
- $V \cap E = \emptyset$
- $src : E \to V$ is a function identifying the source vertex of each edge
- $tgt : E \to V$ is a function identifying the destination vertex of each edge
- $lab : V \cup E \to \Sigma$ is a function assigning each vertex/edge a label from some set Σ

The concept of isomorphic (sub)graph relationships for simple directed multigraphs forms the base for our need to compare and match provenance graphs. Two examples for illustrating the isomorphic definition are shown in Figure 2.1. Figure 2.1a contains an isomorphic multigraph pair and Figure 2.1b contains a pair of directed multigraphs with isomorphic sub-graph relationship. The matching node set of the two graphs in both figures are shown in Table 2.2. The non-matching node on graph H of Figure 2.1b makes that two multigraphs form an isomorphic sub-graph relationship.



(a) Isomorphic graph pair (b) Isomorphic sub-graph pair

Figure 2.1: Example of a pair of isomorphic (sub)graph

Graph G Node	A	B	C	D
Graph H Node	2	1	3	4

Table 2.2: Mapping node set of Graph G and Graph H in Figure 2.1

In general, the graph isomorphic relationship of two simple directed multigraphs can be defined as follows.
Let G and H be two simple directed multigraphs.
$G = (V_G, E_G, src_G, tgt_G, lab_G)$
$H = (V_H, E_H, src_H, tgt_H, lab_H)$

If G and H is an isomorphic pair, represented by $G \cong H$,
then there is two bijection functions f and g where
$f : V_G \to V_H$ and $g : E_G \to E_H$ such that

- $\forall_{v \in V_G} lab_H(f(v)) = lab_G(v)$
- $\forall_{e \in E_G} lab_H(g(e)) = lab_G(e)$
- $\forall_{e \in E_G} src_H(g(e)) = f(src_G(e))$
- $\forall_{e \in E_G} tgt_H(g(e)) = f(tgt_G(e))$

Apart from the general graph isomorphism relationship, the sub-graph isomorphism relationship can be defined as follows.
Let G and H be two simple directed multigraphs.
$G = (V_G, E_G, src_G, tgt_G, lab_G)$
$H = (V_H, E_H, src_H, tgt_H, lab_H)$

If G and H contain isomorphic sub-graph relationship, represent by $G \lesssim H$ then
$G \cong H'$ satisfying:

- $H' = (V_{H'}, E_{H'}, src_{H'}, tgt_{H'}, lab_{H'})$
- $V_{H'} \subseteq V_H$
- $E_{H'} \subseteq E_H$
- $\forall_{e \in E_{H'}, src_{H'} \in V_{H'}} src_{H'}(e) = src_H(e)$
- $\forall_{e \in E_{H'}, tgt_{H'} \in V_{H'}} tgt_{H'}(e) = tgt_H(e)$
- $\forall_{x \in (E_{H'} \cup V_{H'})} lab_{H'}(x) = lab_H(x)$

As we mentioned above, the graph matching problems are unavoidable in provenance analysis. Although the graph comparison needed in provenance is a (sub)graph isomorphism problem, there exist some slightly different requirements. First of all, provenance graphs generally record many different property values with the executions. These properties include timestamps, process identifiers and other information that is useful for tracing back for accountability or reproducibility reasons. In most operating system environments, these data are constantly changing. It is expected to get a different set of these data for the same set of actions while the remaining data should remain the same. This property makes it different from a general graph isomorphism matching problem. In general, an attributed directed multigraph contains additional property key-value pairs attached to the vertexes and directed edges comparing to a

simple directed multigraph. The additional requirement is defined as follow.
Let G be an attributed directed multigraph, then
$G = (V, E, src, tgt, lab, prop)$

where $prop : (V \cup E) \times \Gamma \rightarrow \Delta$ is an extra function

where Γ is the set of property keys and Δ is the set of property data values

The isomorphic graph relationship of two attributed directed multigraphs are defined as follows.

Let G and H be two attributed directed multigraphs.
$G = (V_G, E_G, src_G, tgt_G, lab_G, prop_G)$
$H = (V_H, E_H, src_H, tgt_H, lab_H, prop_H)$

If G and H is isomorphic pair, represented by $G \cong H$,
then the two bijection functions f and g have the following additional requirements:

- $\forall_{v \in V_G, k \in \Gamma} prop_H(f(v), k) \sqsubseteq prop_G(v, k)$
- $\forall_{e \in E_G, k \in \Gamma} prop_H(g(e), k) \sqsubseteq prop_G(e, k)$

Apart from the general graph isomorphism relationship, the sub-graph isomorphism relationship can be defined as follows.
Let G and H be two attributed directed multigraphs.
$G = (V_G, E_G, src_G, tgt_G, lab_G, prop_G)$
$H = (V_H, E_H, src_H, tgt_H, lab_H, prop_H)$

If G and H contain isomorphic sub-graph relationship, represent by $G \lesssim H$ then
$G \cong H'$ satisfy the additional requirement as follows:

- $\forall_{x \in (E_{H'} \cup V_{H'})} prop_{H'}(x) \sqsubseteq prop_H(x)$

Above is the definition for isomorphic relationship for attributed directed multigraphs. It is an extension from the general isomorphic relationship for simple directed multigraphs. For isomorphic sub-graph mapping, the only thing needed is to add in additional consideration of the attributed labels to identify the optimal matching solution, the remaining non-match labels should be kept to a minimum for the optimal solution as we want to map as much as we could for all elements in the graphs to identify the maximum sub-graph in graph H that are isomorphic to Graph G.

2.4.2 Existing algorithms

There exist some algorithms proposed by different researchers, aiming to either speed up the (sub)graph isomorphism problem or solve a subset of the problem.

In Gallagher [55], the author surveys existing pattern matching techniques for graphs. It mentions the sub-graph isomorphism problem and the hardness of it and provides two options. One option is to use approximation, which may result in non-optimal solutions. The other option is to find optimal solutions but with smaller data size by picking a subset of the data. They take the second route and present ways to filter out part of the graph which is less related to the problem. This also inspires the need to filter out part of the unrelated trace in the provenance graph. The major reason why we need to choose between the two options is that sub-graph isomorphic matching problem is known to be NP-Complete, as mentioned in Washio and Motoda [159] where the authors discussing the graph isomorphism using for graph mining.

As solving graph isomorphism in polynomial time is a known open problem in the research field, there exist some researchers proposing some algorithm to simplify the problem or to solve partial problems through approximation or other means to keep the process within polynomial time. In Cordella et al. [40], the author suggests an algorithm VF2 to solve the graph and sub-graph isomorphism problems for large graphs. While in Sun et al. [152], the authors suggest alternative algorithms to match graph structures with billions of nodes. In McKay and Piperno [105], the authors summarize the state of art for the graph isomorphism problem and provide some surveys of existing algorithms. These publications are suggesting ways to handle graph comparison and matching more efficiently, but the graph isomorphism problems still require either intensive resources or certain levels of approximations or partial comparison to decrease the problem by the size of the graph.

In Gamkrelidze et al. [56], the authors suggest an algorithm to compute invariants of graphs through randomized walk-throughs in polynomial time. They did not prove successful for large and full graphs. Another paper by Mendivelso and Pinzón [106] also provided a new way to solve the (sub)graph isomorphism problem, they are aiming to solve graph isomorphism problem in attributed graphs. They wanted to determine how many combinations of attributes exist when the edges and vertexes of the attributed graphs form an isomorphic pair. Finally, Lee et al. [92] summarize five of the algorithms for the (sub)graph isomorphism problem that tries to make the problem easier and faster to solve. They also provide state of the art for the research on

this problem. These existing algorithms are all proposing ways for some shorthand or partial handling of graph isomorphism problems in different types of graphs.

Our need is different from traditional graph isomorphism as provenance graphs may contain volatile data where the property key is the same but the property value of the attribute can be ever-changing. If we adopt traditional (sub)graph isomorphism problem to our case, it will either give no solutions (If we consider property values as matching criteria) or multiple solutions (If we just consider vertexes and edges only) and randomly assign one as the answer. But, we want the graph to be as close as possible because the only difference should be in the value of attributes and all other parts should be preserved. To achieve this, we need to have some metric to measure how close the two graphs are when their edges and vertexes match in a certain way. Then we can choose the closest one as the matching result. There is some research on algorithms for this problem.

In Bunke and Shearer [29] and Hoffmann et al. [79], the authors propose the need to have a metric to aid the discovery of maximum common graphs. The metric can be used in finding the closest graph in a case where the underlying simple graph is part of an isomorphic pair. Some researchers have proposed edit distance approaches as the metric to determine how close the two graphs are. This approach has been mentioned in Riesen and Bunke [142], Bunke [28], Gao et al. [57] and Abu-Aisheh et al. [3]. Also the book [141] mentions different kinds of edit distance approaches for attributed graphs, including SimRank [31], Probability learning [122], Self-organizing maps [123], Binary linear programming [86], Convolution graph kernel [124] and Sub-graph and super-graph matching [50].

Similar to isomorphic sub-graph matching, computing the graph edit distance between two graphs (even without labels or properties) is an NP-complete problem. In general, the metric using edit distance is to determine the differential level between two graphs. Given a set of basic edit operations (e.g., insertions, deletions or in-place modifications), we write $op(G)$ for the result of op acting on G where op is any basic edit operations. More generally, if ops is an ordered list of operations op, then we write $ops(G)$ for the result of applying all the operations in ops to G in order. Given two graphs G_1, G_2 we define the term *graph edit distance* between G_1 and G_2 as $GED(G_1, G_2) = \min\{|ops| \mid ops(G_1) = G_2\}$, that is, the shortest length of an edit script modifying G_1 to G_2. The associated costs for each operation may be different depending on the need to highlight the importance of certain operations. As a result, the graph pairs with the closest proximity should have a lower *graph edit distance*

value. We aim to take the graph edit distance approach for the user to identify matching vertexes, edges and property keys where the property values are not checked. This allows us to filter out volatile property fields that are not related to the activities in the key part of the executions. It does not affect the isomorphism on the underlying simple graph as the bijection relationship would still hold for the edge and vertex sets.

2.5 Answer Set Programming

2.5.1 Solving hard search problems

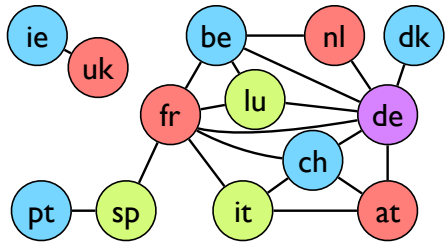
Answer Set Programming (ASP) is a kind of declarative programming. Its primary aim is to solve difficult search problems, like problems in the NP-hard range. It wants to provide a fast and efficient solver for problems specified in logic by using some logic programming semantics. In Lifschitz [95] and Brewka et al. [26], the authors give a summary of Answer Set Programming, introducing the semantics, complexity, applications and other characteristics of Answer Set Programming. They give a general idea of what problems the Answer Set Programming can solve. In our case, we are aiming to match isomorphic graph pairs with random property values together. This requires an intensive search for all possible matches of the vertexes, edges and property keys and calculating its edit distance in terms of the change needed for the property values. The problem is NP-hard and the structures of the graphs allow us to transfer them into logical statements and make use of Answer Set Programming to search for an optimal choice for the matching solution with least edit distance, or in other words, maximum common sub-graphs among the pairs.

2.5.2 Potassco framework and Clingo

It is not an easy thing to solve a large set of logical statements with Answer Set Programming manually as the scale of proving may be large. There is a need to have some tools to automate the proving and solving steps. In Gebser et al. [62], the authors proposed *Clasp*, an answer set solver that aims to solve variable free logical statements and provide suitable non-monotonic reasoning on top of the solutions. The authors later summarize the challenge for the solver in Gebser et al. [60], which also mentioned some possible applications of Answer Set Programming. The authors also proposed some possible extensions to make their tools and technology more complete. The authors later develop the *Potassco framework* [61] that bundles various tools that apply

Answer Set Programming to solve problems. For example, it contains a tool to solve and handle the package configuration and installation in Linux environments. Besides, the framework includes a tool named *Clingo* which is a combination of the grounding tools and answer set solver provided in the framework. In general, the grounding process takes in ASP programs and maps them to propositional logic statements. In other words, the process transfer general logical statements into variable free logical statements. The grounding tools *Gringo* (which is formalized in Gebser et al. [58]) provides the grounding process and the solver *Clasp* mentioned above solves those variable free logical statements and provide solutions and reasoning accordingly as results.

The major programming language support by Clingo (or Gringo the grounder) is Datalog. In general, Datalog is a subset of the Prolog language, which makes it also a declarative logical language which describes each of the logic criteria one by one. In traditional usage, Datalog is more likely used in a database as a kind of query language, which is why it is adopted in Clingo to solve hard search problems. As mentioned in the book Abiteboul et al. [2], David Maier is credited for the naming of Datalog. Since then, Datalog is widely used by logic and database field. Until recent years, it spreads into some newer field like provenance, programming and security analysis, and cloud computing because of its high expressiveness for representing data in different circumstances and environments. In Lifschitz [96], the author provides a stable model for Datalog, aiming to provide a semantic definition for Datalog to adopt its usage in software development which may include parallel computing. In later paper Huang et al. [81], the authors provide a basic description of the state of art for Datalog development and application. They also provide a summary of the semantics of Datalog and a tutorial on who should and how to use it. Lastly, in Deutch et al. [44], the authors even provide an algorithm for querying stored data provenance which is represented by Datalog statements. This once again shows that the ability of declarative languages and answer set programming solvers can help to solve some hard search problems including the provenance graph matching problem required by our work. In our work, we transfer the graph and our matching requirements into logical statements and use Clingo to help us determine the maximum common sub-graph between graphs. The process helps us to identify the matching elements in the graphs and to filter out volatile information which is not needed in the benchmarking process. The Potassco framework is under continuing development and the newest version [59] is published when this thesis is written. Also, some researchers are summarizing their experience and use in a tutorial of Clingo in Kaminski et al. [87]. It provides an overview of how to apply



e(uk,ie). e(sp,pt). e(it,at).
e(fr,sp). e(fr,de). e(fr,ch). e(fr,it).
e(fr,be). e(fr,lu). e(be,lu). e(be,nl).
e(ch,it). e(ch,at). e(de,be). e(de,nl).
e(de,at). e(de,dk). e(de,ch). e(de,lu).

Figure 2.2: Graph coloring example

Clingo in different cases that require solving problems relating to extensive searching or reasoning.

A standard example of answer set programming is given in this subsection. The example code in this subsection (and those in later chapters) can be run verbatim using the Clingo interactive online demo[3]. This example aims to use Clingo to solve the graph 3-colouring problem, which aims to colour the nodes of an undirected graph with the minimum number of colours. Figure 2.2 shows an example graph where edge relationships correspond to land borders between some countries.

Code Snippet 2.1 defines graph 3-coloring declaratively. The first line states that the edge relation is symmetric and the second defines the node relation to consist of all sources (and by symmetry targets) of edges. Line 3 defines a relation color/1 to hold for values 1,2,3. Lines 4–5 define when a graph is 3-colourable, by defining when a relation c/2 is a valid 3-colouring. Line 4 says that c/2 represents a (total) function from nodes to colours, i.e. for every node there is exactly one associated colour. Line 5 says that for each edge, the associated colours of the source and target must be different. Here, we are using the not operator solely to illustrate its use, but we could have done without it, writing C = D instead. Lastly, the operator :− and , stands for *if* and *and* respectively. If the statement is started the :− operator, it simply means that the given condition is met by nothing, that means the given condition should not exist in all statements, like our definition in Line 5.

```
1 e(X,Y) :- e(Y,X).
2 n(X) :- e(X,_).
3 color(1..3).
4 {c(X,Y) : color(Y)} = 1 :- n(X).
5 :- e(X,Y), c(X,C), c(Y,D), not C <> D.
```

Code Snippet 2.1: Graph 3-coloring

[3]https://potassco.org/clingo/run/

Code Snippet 2.1 is a complete program that can be used with Figure 2.2 to determine that the example graph is not 3-colourable. What if we want to find the least k such that a graph is k-colourable? We cannot leave the number of colours undefined since ASP requires a finite search space, but we could manually change the '3' on line 5 to various values of k, starting with the maximum $k = |V|$ and decreasing until the minimum possible k is found.

```
1 color(X) :- n(X).
2 cost(C,1) :- c(_,C).
3 #minimize { Cost,C : cost(C,Cost) }.
```

Code Snippet 2.2: Minimal k-coloring (extending Listing 2.1)

Instead, using *minimization constraints*, we can modify the 3-colouring program above to instead compute a minimal k-colouring (that is, find a colouring minimizing the number of colours) purely declaratively by adding the clauses shown in Code Snippet 2.2. Line 1 defines the set of colours simply to be the set of node identifiers (plus the three colours we already had, but this is harmless). Line 2 associates a cost of 1 with each used colour. Finally, line 3 imposes a minimization constraint: to minimize the sum of the costs of the colours. Thus, using a single Clingo specification we can automatically find the minimum number of colours needed for this (or any) undirected graph. The 4-colouring shown in Figure 2.2 was found this way.

The above example tries to illustrate the basic usage for the answer set programming solver. It defines the basic operation of the solver. In general, the data set for the target problem should be attached to the end of the program for the Clingo solver to find an optimal solution for the given set of data. The dataset format is defined according to the actual usage. Thus we will define our kind of graph representation on attributed multigraphs isomorphic matching in later chapters.

2.6 Non-determinism

2.6.1 Non-deterministic events

In most operating systems and environments, the concept of non-determinism is widespread. It refers to some uncertainty in the choice of execution processes and the order of some of the execution statements. The reason for the existence of non-deterministic events is various, for example, to support efficient execution or provide a more rational use of

resources without affecting the original execution result. In Mohindra et al. [109], the authors proposed a way to make use of non-determinism in mobile environments to capture state change and the computation traces for fault recovery. This paper demonstrates the advantages of non-deterministic events. In Potop-Butucaru et al. [135], the authors analyse concurrency issues in synchronous systems which have certain levels of non-deterministic events which give another example of non-determinism. Later in Armoni and Gal-Ezer [12], the authors summarize the abstract concept of non-determinism and provide some additional information about the characteristics and applications for non-deterministic events. They also provide some description of the state-of-art of the research on non-determinism. Finally, in Okech et al. [125], the authors study the behaviour of certain system-calls in the kernel level. Their work shows that even some deterministic events on the user level behave in a non-deterministic way in the kernel level and this is also a motivation for why we need to include the support of non-deterministic events in our automated benchmarking approach.

2.6.2 Symbolic execution

In many programming environments, symbolic execution is being used for debugging and generating test cases automatically. In a general application, it is sometimes not possible to have one execution path. Most of them contain multiple execution paths depending on many criteria including user interaction, environments, configurations, etc. Debugging of these applications is required for each of the paths as we never know whether a certain path will be executed. Symbolic execution aims to search for all possible paths in applications and binaries. It can also be used in some executions containing non-deterministic events. In Ma et al. [98], the authors propose using symbolic execution to aid in the discovery of a certain execution path that reaches a target end. This is also related to what we want for provenance analysis to trace the initiators of certain events in multiple execution path situations.

Symbolic execution is no doubt a powerful technique which can help to determine all possible paths of execution. But it is not scalable enough because extensive searching for execution paths is a complex problem and requires lots of time and resources. Thus symbolic execution is not feasible on large programs that contain many execution paths. There are some researchers proposing ways to make it easier to handle large cases. For example, in Chipounov et al. [37], the authors propose a selective symbolic execution which only discovers execution paths on a chosen subset of an application

or binary execution. This reduces the exhaustive searching from the whole application to the limited part of execution which the users are interested in. But this does not have much use if the user is interested in retrieving execution paths for the whole application. In Kuznetsov et al. [91], the authors propose ways to dynamically merge some of the similar paths to decrease the total number of results for the work. This may help to increase the efficiency of the work.

Manual effort is almost impossible for symbolic execution because it requires tracing different paths accordingly. Developers are working on tools for automating this work. KLEE, proposed in Cadar et al. [30], is a tool aiming for automatic symbolic execution. In Corin and Manzano [41], the authors propose ways to use KLEE with additional taint analysis to trace data flow in execution and seek to discover security problems along the path discovery process. This shows one of the similarities with our applications to discover provenance in all possible execution paths to cover full patterns of execution for provenance benchmarking. The characteristics of symbolic execution are useful for the provenance benchmarking as the target activities may contain non-deterministic events and with the support of symbolic execution, most of the possible execution paths can be covered in the automatic benchmarking process and hopefully allow capturing all possible patterns for certain activity sequences.

2.6.3 Fingerprinting and grouping

In our usage of provenance tracing for non-deterministic events, we apply a scheme of fingerprinting to collect and classify graphs by their execution traces. One way of fingerprinting is to trace some of the essential information in the kernel which remains the same for the execution of the same path and different for the execution of a different path. One possible solution fulfilling this requirement is using the tool ftrace which is an existing functional tracer in the Linux kernel. In Restedt [143, 144] and Edge [47], the authors summarize the usage of ftrace and provide some examples of applications of ftrace. In general, we can distinguish the different execution paths by examining the order in which the real system-call execution happens in the kernel, which is captured by ftrace. This order of system-call execution can be used as a fingerprint to identify and group the provenance graphs of different execution paths for further processing.

2.7 Summary

This chapter covers most of the related work, background information and knowledge and some of the building blocks for this thesis. The automated benchmarking system developed for this work is based on the needs mentioned in this chapter which originate from different literature around this research field. This chapter also summarizes some of the techniques that are adopted in the automated benchmarking system, ProvMark, for the analysing of data provenance and the patterns for certain sensitive actions. The main work from the concept generation to the full implementation of ProvMark is introduced starting from the next chapter.

Chapter 3

Manual analysis of basic system-calls

3.1 Motivation

The major target for this project is to evaluate the capabilities of different provenance collecting tools to distinguish different system-call action sequences. In general, we do not have a unified formal definition of provenance. Different provenance collecting tools collect provenance information as different structures to suit their own needs. So before analysing the capabilities of each of the provenance tools, we need to first identify how they describe the same set of activities. We start this task by doing some experiments to understand how different provenance tools describe the basic system-calls, which are the foundational building blocks for large provenance graphs.

The goal is to identify provenance patterns for a universal unit across different Unix-based operating system: system-calls operating on the OS kernel. Those system-calls combine in different permutations to form high-level processes. When we understand the explicit patterns of provenance for each system-call, we can map each system-call to its effect on the resulting provenance graph for specific provenance collecting tools. These mappings can help us identify what system-calls have been involved in a runtime process by a simple analysis of its provenance graph generated by a specific tool. Besides, these mappings can also be used for unit testing by the tool developers to identify if the result provenance generated match their expected result and confirm the correctness of their generation process.

Before considering how the system-calls patterns can help in the process, we aim to develop some building blocks manually for proof of concept. In this first stage of work, we aim to manually feed some provenance collecting tools with suitable input and observe the provenance information generated by them. And we will do some

manual identification, classification and comparison to retrieve the provenance patterns for those system-calls, generated by different tools. This manual work can help us to build up and consolidate the foundational work, which provides a basic understanding of how those system-calls are displayed as provenance patterns. It also helps to demonstrate the feasibility of this study on a bigger scale.

3.2 Definitions

3.2.1 Expressiveness micro-benchmarking

One of the motivations of this work is to compare the capabilities of the tools in security, auditing and forensic usage. In general, the term benchmark is used to define the point of references which can represent the object at a similar level for general comparison. In our project, we also aim to generate some mapping for different provenance collecting tools to identify their provenance result for describing the same system-calls. For easy understanding, we define the system-calls to provenance mapping of each tool as the *micro-benchmark* of that tool. These mapping act as the point of references for the same level general comparison of the tools.

We further define the process of generating *micro-benchmark* and the same level general comparison as *micro-benchmarking*. This process aims to provide a formal way to describe the foundational building blocks for the provenance data graph generated by each of the provenance collecting tools. All resulting provenance graphs should be built by combining these foundational units because they describe the smallest meaningful unit: system-calls. Each action sequence produced in the resulting provenance graph is simply a combination of different system-calls in some order. By comparing the micro-benchmark of different tools, we already cover provenance patterns for almost all possible action sequences. Thus, the process of micro-benchmarking helps us to evaluate the capabilities of the tools in different settings.

After that, we define the term micro-benchmark as a set of provenance information. It describes the kernel action sequence relating to a target system-call which is generated by a chosen provenance collecting tool. In other words, the micro-benchmark should have a one to one mapping to a specific high-level system-call which allows a viewer of the micro-benchmark uniquely identify the existence of the system-call in a larger action sequence with multiple combinations of system-call execution. The micro-benchmark is generally presented as a subset of elements in a directed graph.

Thus the micro-benchmark may not be a completed graph. From above, we further define the term *micro-benchmarking* as the process of evaluating the capabilities of the provenance collecting tools in different settings by comparing the *micro-benchmark*. In this context, we further define the term *expressiveness benchmarking* as a more specific type of *micro-benchmarking* that concentrate on the expressiveness of the provenance tools only. We further define the term *expressiveness* as the ability for provenance collecting tools to correctly and completely describe an activity sequence as a provenance graph. As a whole, the *expressiveness benchmarking* process aims to use the *micro-benchmark* to compare the capabilities of provenance collecting tools specifically identifying whether certain system-call or activity sequences have occurred at runtime by studying the provenance graph generated for the session.

We further define the terms *correctness* and *completeness* as the two main targets for the evaluation of *expressiveness* in the *expressiveness benchmarking* process. *Correctness* of provenance refers to the quality of the provenance data of being free from error. Or in other words, the accuracy of the provenance data describing what is happening during the runtime execution. The *correctness* requirement may be different when analysing the provenance data for different applications. Some applications may allow a certain level of error while still correctly identifying the needed information. For example, if someone just wants to understand what system call activities exist during the runtime execution, then the other data like timestamp and execution order is not important in this application. In this situation, possible errors in this meta-data will not affect the identification of existing system call activities and do not affect the *correctness* factors of the provenance data. Thus our definition of *correctness* focuses on the accuracy of data for a specific application and can vary across different applications which require different levels of preciseness for the provenance data. On the other hand, *completeness* of provenance refers to the quality of the provenance data having all the necessary data or components. It can also be identified as unambiguity of the provenance data. Similar to the *correcteness* factor, the *completeness* factor may also be different when analysing the provenance data for different applications. The level of *completeness* is determined by analysing if the provenance information contains enough information for identifying different events correctly. In different applications of provenance, this requirement may be different. For example, if one application just wants to understand if a read system call has been executed, then the provenance data can be less complete because it just needs to contains the system call trace for this application. If other application wants to know that if there is a read system call ex-

ecuted before a write system call, then it will require timestamps and execution order to distinguish the different cases. In the latter situation, it requires a higher level of *completeness* because it requires more information recorded in the provenance data to distinguish ambiguous cases. From the above example, it shows that the *correcteness* and *completeness* quantifiers for provenance expressiveness evaluation can vary across different applications. Our *expressiveness benchmarking* approach aims to provide benchmarks in terms of provenance graph structure for a provenance system. Further comparison of those benchmarks from different provenance systems describing the same runtime execution for the same application are needed to understand which provenance systems have a higher *completeness* or *correctness* level in a specific application. Thus, we can also related the *completeness* quantifier to the false-negative error rate and the *correctness* factor to the false-positive error rate. The linkage between the qualifier and error rate may not be true in some of the application. We only make use of this two items *correctness* and *completeness* as the main quantifiers in the *expressiveness benchmarking* process. The actual analysing and comparing of these qualifiers and possible error rates requires a further comparison of the benchmark and may be varied for different applications and may relate to the user expectation on what information should be included to correctly and completely identify the needed information and distinguish between similar cases.

3.2.2 Operating System terminology

Many resources can be managed by a user via kernel activities or system-calls, including the file system, input or output stream, hardware drivers or even network protocols. Many sensitive system-calls can modify these resources and sometimes may touch multiple of them to finish a job. In general, the term artefact is defined as some man-made object of historical significance. When we are analysing the provenance graph, we are studying what is happening to some resources in the past. So, the resources defined in the provenance graph are just artefacts. For easy understanding and grouping, we borrow this noun and define *artefact* as all of those resources which can be controlled, used or accessed by system-calls and processes in runtime.

Last but not least, we further define the term *noise* as a subset of the provenance graph result which is not related to the target system-call actions, including random heartbeat message or mapping of libraries and memories. These data are transmitted randomly and repetitively in the kernel. Thus the value of these *noise* are ever-changing

and can be *volatile* across different runs. They are the disturbing factor in the result which we want to extract and avoid. Thus to identify target action sequence by graphs comparison, we need to generalize the graph across different runs to filter out these *volatile values or property labels*. With this understanding, we can use the patterns to generate rules to identify the existence of high-level actions (containing different permutations of system-calls) in a large system by analysing the provenance graph describing its runtime behaviour. This can help to trace the existence and initiator of certain actions for security and forensic analysis.

3.3 Micro-benchmarking

To create a micro-benchmark for system-calls in the kernel, we choose to create some sample programs that use minimal system-calls to see those patterns without a large amount of noise. Although not all system-calls in Linux environment have a one to one mapping to C library calls, we do prepare these micro-benchmark programs in C for easier analysis. We do not test all 300+ system-calls in the Linux operating system, instead, we are using a subset of them which are generally believed to relate to sensitive actions. This subset is mainly gathering from two places. Firstly, we summarize different pieces of literature in the security research field which we mentioned in the background section to understand which system-calls are commonly related to sensitive or malicious activities. Then we study the capturing target for some commonly used provenance collecting tools and summarized their choice. Interestingly, not all of them are collecting the whole set of system-calls and their choice is also very similar to those in the security research field. After consulting the literature and tools from the security and provenance research field, we summarize out a list of system-calls to act as our starting point which is shown in Table 3.1. It is always extensible if we need to handle more system-calls in the future.

These preparation works already contribute to the system-call activity in the kernel. Most of them are memory read/write and memory mapping. To avoid and filter out these kinds of noise from the executing binary and capture the real provenance patterns for our target system-call action, we provide two types of benchmark program mentioned above. First, we develop an empty C program which does nothing to act as the background program. Then we develop a set of C programs for each of the system-call, each of them contains only the target system-calls statement in the main function without any additional actions. When we need to do the test, we compile the

two binaries from the control program and the C program of the target system-call (we name it as the foreground program). The only difference between the two binaries will be the execution of the target system-call that we are interested in. By executing the two binaries and comparing the resulting provenance graph, it is easy to identify in the trial execution, which part in the graph represents the target system-call action. After a manual filter, we can easily identify the additional edges and nodes that describe the additional actions in the two trials, which directly represent the target system-call action. Because the only difference between the two compiled binaries is the system-call executed in the benchmark program, those additional graph elements represent the basic unit of provenance for a specific provenance collecting tools to describe the target system-call.

A sample background program is shown in Code Snippet 3.1, while two benchmark programs for ***creat*** and ***chmod*** are shown in Code Snippet 3.2 and Code Snippet 3.3 respectively. Those two provenance graphs generated by SPADE on the two benchmark programs (Figure 3.2 and Figure 3.4) are compared to the provenance graph (Figure 3.1) generated by SPADE on the background program separately. The additional subset from the two comparison describing only the ***creat***/***chmod*** system-call provenance patterns, which is shown in Figure 3.3 and Figure 3.5. As there are too many properties label in the graphs and it is impossible to read, we only show the structure with certain text labels to emphasize the important nodes in the background and foreground graphs. In the graphs and benchmarks, rectangle nodes are referring to processes and oval nodes are referring to artefacts, including files or pipes.

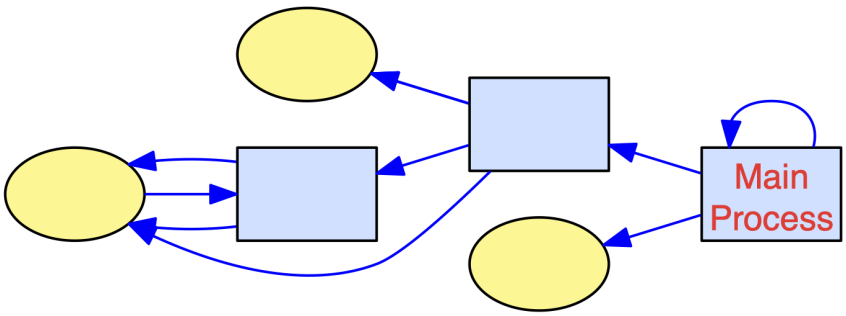


Figure 3.1: Abstract background provenance graph generated by SPADE

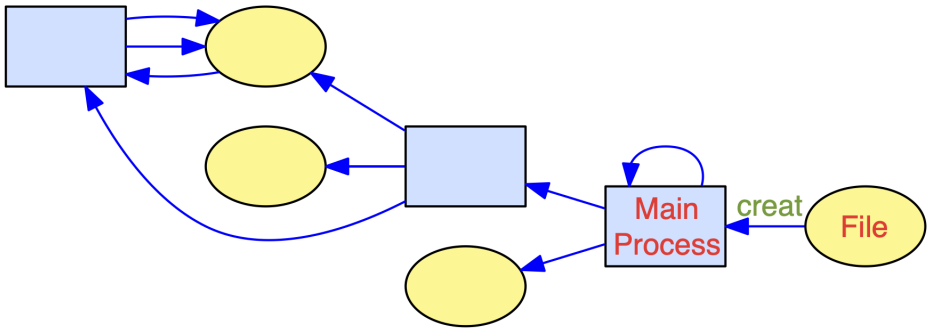


Figure 3.2: Abstract Provenance graph for the system-call ***creat*** generated by SPADE

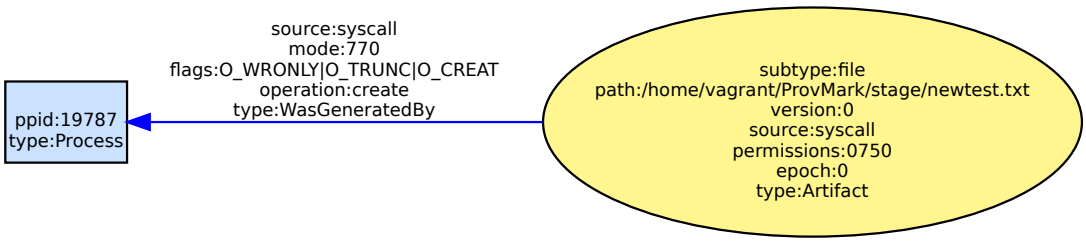


Figure 3.3: Benchmark graph for the system-call ***creat*** generated by SPADE

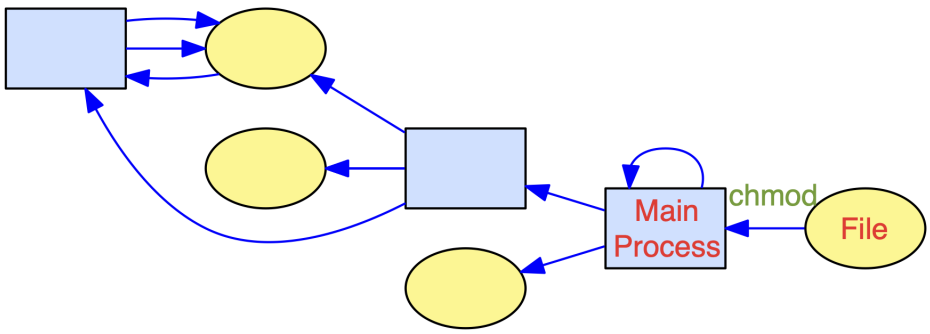


Figure 3.4: Abstract Provenance graph for the system-call ***chmod*** generated by SPADE

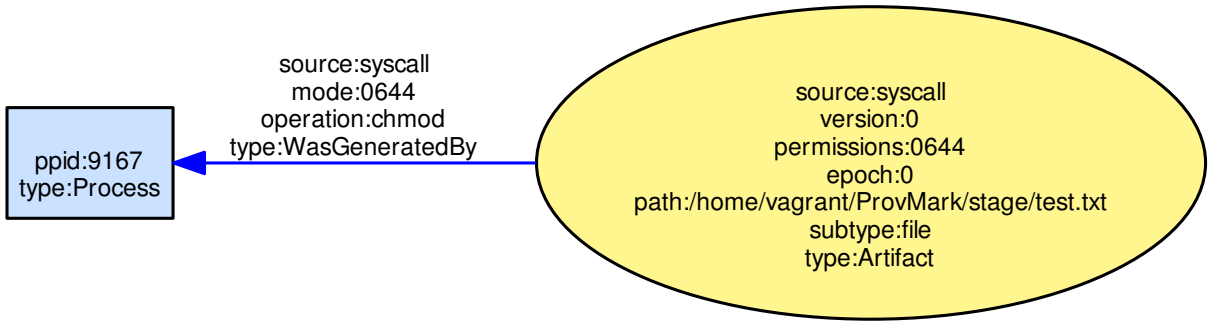


Figure 3.5: Benchmark graph for the system-call ***chmod*** generated by SPADE

```
1 void main() {
2         //Empty Program
3 }
```

Code Snippet 3.1: Sample control program

```
1 #include<fcntl.h>
2 #include<unistd.h>
3 void main() {
4     creat("txt.txt", S_IRWXU|S_IRWXG);
5 }
```

Code Snippet 3.2: Sample benchmark program for ***creat*** system-call

```
1 #include <sys/stat.h>
2 void main() {
3     chmod("txt.txt", S_IRUSR|S_IWUSR);
4 }
```

Code Snippet 3.3: Sample benchmark program for ***chmod*** system-call

From the above program and resulting provenance graph comparison, we can easily identify the additional part that describes the minimum real patterns for a specific system-call. But we have also made certain assumptions. First of all, most provenance collecting tools will provide as much information as they can when generating the provenance information describing what is happening in runtime. In most cases, some volatile values like timestamps, process ids or identifiers provided by the OS are recorded. This information may create part of the noise when we do the sample comparison because it is possible that the two trials running on the background program and foreground program can have differences that make them harder to compare. In our manual approach, it is easy to identify these changing values and manually ignore

them when doing the graph matching and comparison. In this case, we define the graph generated by the background program as background graph BG and define the graph generated by the foreground program of the target system-call as foreground graph FG. We always assume that BG is an isomorphic sub-graph of FG, which ignores the changing elements that we consider noise. We further assume that after ignoring this noise, BG and FG will each remain the same even if the corresponding program is executed multiple times. It can be illustrated as follows.

$BG_1 = BG_2 = ... = BG_N = BG$

$FG_1 = FG_2 = ... = FG_N = FG$

where FG_n and BG_n represent the foreground/background graph generated for the *nth* trial of execution of the control program and the benchmark program for the target system-call respectively. In this context, we are only considering isomorphism relationships between the nodes and edges of the graphs. With the existence of noise and volatile information across different runs, taking into account the property values in the isomorphism is too strong. Thus the isomorphism considering in our context is just limited to the skeleton of the graphs, which contains the edges and vertices only. This makes our work more flexible and less complex in handling the graph generalization and comparison. Also, the above-assumed equality is preserved as we are only considering deterministic events in our manual approach, thus the execution order and events should be preserved and the generated graphs should be equal (if we ignore the noise) and isomorphic (on edges and vertices only). In our manual approach, as we are comparing the graphs and generalizing them manually, it is possible to achieve the assumed equality. But this may not be possible once the generalization steps are done by automated approach.

In addition, our manual efforts also ignore the volatile values when comparing the foreground graph and background graph. With this assumption, we can assume there is a sub-graph FG' of the foreground graph FG which is always isomorphic to the background graph BG. Following the graph definition in Section 2.4, the following defines the isomorphic sub-graph relationship for the background graph BG and foreground graph FG.

Let FG and BG be two attributed directed multigraphs,then

$BG = (V_{BG}, E_{BG}, src_{BG}, tgt_{BG}, lab_{BG}, prop_{BG})$

$FG = (V_{FG}, E_{FG}, src_{FG}, tgt_{FG}, lab_{FG}, prop_{FG})$

If BG and FG are isomorphic sub-graph pairs,
then $BG \lesssim FG$ and $BG \cong FG'$ satisfying

- $FG' = (V_{FG'}, E_{FG'}, src_{FG'}, tgt_{FG'}, lab_{FG'})$
- $V_{FG'} \subseteq V_{FG}$
- $E_{FG'} \subseteq E_{FG}$
- $\forall_{e \in E_{FG'}, src_{FG'} \in V_{FG'}} src_{FG'}(e) = src_{FG}(e)$
- $\forall_{e \in E_{FG'}, tgt_{FG'} \in V_{FG'}} tgt_{FG'}(e) = tgt_{FG}(e)$
- $\forall_{x \in (E_{FG'} \cup V_{FG'})} lab_{FG'}(x) = lab_{FG}(x)$
- $\forall_{x \in (E_{FG'} \cup V_{FG'})} prop_{FG'}(x) \sqsubseteq prop_{FG}(x)$

$SP = FG - FG' = \{V_{FG} \setminus V_{FG'}, E_{FG} \setminus E_{FG'}, src_{SP}, tgt_{SP}, lab_{SP}, prop_{SP}\}$
where

- $src_{SP} = \forall_{e \in E_{FG'}} \ src_{FG}[e := \bot]$
- $tgt_{SP} = \forall_{e \in E_{FG'}} \ tgt_{FG}[e := \bot]$
- $lab_{SP} = \forall_{x \in (E_{FG'} \cup V_{FG'})} \ lab_{FG}[x := \bot]$
- $prop_{SP} = \forall_{x \in (E_{FG'} \cup V_{FG'}), k \in \Gamma} \ prop_{FG}[x, k := \bot]$

where SP is the minimum system-call pattern generated by retrieving the additional part in the foreground graph, which is the micro-benchmark we are looking for. In other words, we are aiming the find the set difference between the node-set, edge-set and property-set of the two graphs. Some provenance collecting tools may return an empty result for some system-calls because they did not capture them in their generating process or some of them has been filtered out by settings.

3.4 Results from Micro-benchmarking

For the manual effort of micro-benchmarking, we chose two existing provenance collecting tools, SPADE and OPUS, as the testing targets. Both of them focus on a similar target on recording the call trace history for Unix-based system calls which allow us to compare their resulting provenance graphs easily. Also, both of the tools do not concentrate much on artefact versioning thus produce comparatively smaller provenance graphs than other provenance systems. Because we are focusing on demonstrating the feasibility of our proposed provenance benchmarking approach with manual comparison, thus it is better to take some smaller and simpler examples as the preliminary target. This is the major reason we are choosing SPADE and OPUS as our preliminary candidates for the manual benchmarking experiment. Besides, both of them can output the provenance result in Neo4J graph database format [121] that can be queried using the Cypher query language. This allows easy access for manual expressiveness benchmarking. The unique provenance patterns should identify each of the system-call

actions. Those results should be enough to generate rules to determine the existence of certain activities. The resulting provenance patterns are expected to behave the same when the system-call activity happens in different environments. Last but not least, it is also important to mention that although both SPADE and OPUS can generate provenance in the same output format, they collect information from different sources and preserve different levels of detail for different system-call activities. So their results are quite different. This difference demonstrates the importance of a systematic way to compare their expressiveness towards malicious patterns identification.

Group 1				Group 2	Group 3		Group 4
creat	dup	dup2	dup3	clone	chmod	fchmod	pipe
link	linkat	symlink	symlinkat	execve	fchmodat	chown	pipe2
mknod	mknodat	open	openat	exit	fchown	fchownat	tee
read	pread	rename	renameat	fork	setgid	setregid	
truncate	ftruncate	unlink	unlinkat	vfork	setresgid	setuid	
write	pwrite	close		kill	setreuid	setresuid	

Table 3.1: System-calls considered in the manual analysis

Table 3.1 displays the system-calls that we are using again. We divided the system-calls into groups for easy understanding. Group 1 contains system-calls that are related to file and artefact management. Group 2 contains system-calls that are used for process management. Group 3 contains system-calls that are used to manage permissions and privileges of processes of some artefacts. Last but not least, group 4 contains system-calls related to inter-process communication. We operate SPADE and OPUS to monitor and filter background information to generate minimum provenance data to describe these system-calls separately and we classify the results into multiple categories according to the resulting provenance structure. We have prepared an empty background program and a batch of C programs (foreground programs) for each of the system-calls mentioned in Table 3.1.

3.4.1 Micro-benchmarking on SPADE

As mentioned above, we classify the results into multiple categories according to the resulting provenance structure of the target system-call. The detailed description will be given in subsection 3.4.3. The classification for SPADE is shown in Table 3.2. When we are compiling the C programs (control and benchmark programs) to executable binaries, we choose to statically link the library at compile time to avoid more noise from dynamic linking of the library in the real-time execution. Both Figure 3.6 and Figure 3.7 shows the abstract provenance graphs

(property labels are omitted because of readability problem) generated by SPADE on the same program (with one ***creat*** system-call only) with different library linking options. Figure 3.6 shows the provenance graph for the static-linked binary while Figure 3.7 shows the provenance graph for the dynamic-linked binary. We can see that the graph for dynamic-linked binary contains more structure than the static-linked binary. Most of the extra structure is referring to those dynamic linking operations and is not related to the actual execution of the ***creat*** system-call. Thus it justifies that using the static linking option for program compile can help reduce the noise and allow a better generalization and comparison process.

Cat 1		Cat 2	Cat 2a	Cat 3	Cat 4	Cat 5
dup	dup2	creat	fchmod	execve	link	pipe
dup3	kill	close	truncate	fork	linkat	pipe2
mknod	mknodat	open	ftruncate	vfork	symlink	tee
setgid	setregid	openat		clone		
setresgid	chown	unlink		setuid		
fchown	fchownat	unlinkat		setreuid		
		chmod		setresuid		
		fchmodat		exit		

Table 3.2: System-calls classification for SPADE

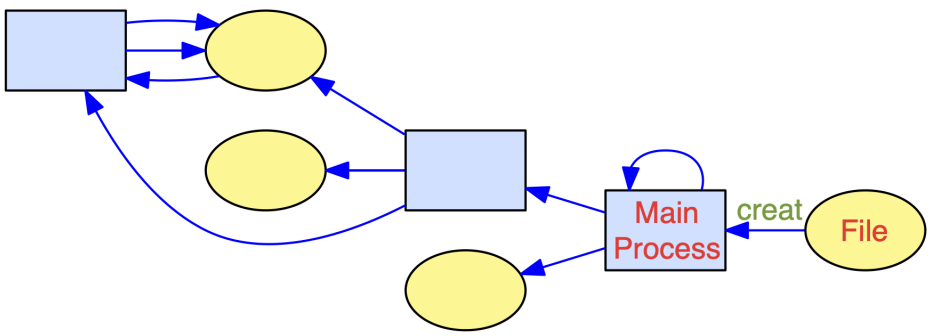


Figure 3.6: Abstract provenance graph for the static-linked binary generated by SPADE

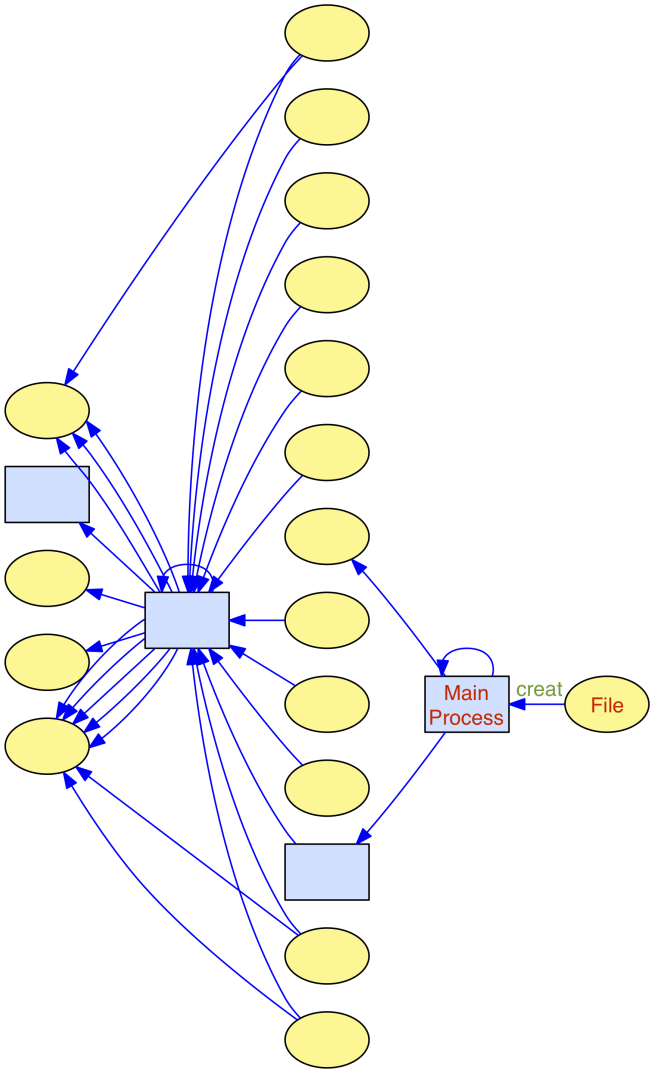


Figure 3.7: Abstract provenance graph for the dynamic-linked binary generated by SPADE

As mentioned in Chapter 2, SPADE retrieves the system activity from many different channels. As some of them are still not well implemented, we choose the most completely implemented reporter module provided by SPADE, which is the audit reporter which aims to retrieve kernel activities from the audit daemon (`auditd`) and dispatcher provided by the operating system. The audit daemon acts as a log to record all the events and activities that have happened in the kernel for system administrators to monitor. SPADE uses this information to form a result provenance graph as output to describe what has happened in runtime. One of the important points to mention is, SPADE does not record explicit information about some actions that do not touch the content of the artefacts, like ***dup*** or ***mknod*** system-calls. Instead, they just create or clone the internal path mapping elements. In the viewpoint of SPADE, these actions do not

affect the content of the artefacts and so they are not recorded explicitly. Nevertheless, they do indirectly affect the behaviour of subsequent system-calls. For example, ***dup*** creates a new file descriptor of an artefact. This action may have indirect effects on how subsequent system-calls are recorded since processes can communicate with the same artefact with different file descriptors. This example shows that provenance may not completely reflect all execution at runtime (completeness problem), thus demonstrating the need for expressiveness benchmarking.

As mentioned in Chapter 2, SPADE provides some features, including partial version information and filtering mechanisms. For system-calls related to output and operation of files, SPADE provides additional descriptions in the resulting provenance data that showcase the version updates of a files or output stream target. This provides additional information about the version changes and accountable process of the action. On the other hand, the filtering mechanisms of SPADE are provided at two levels. The lower level is on the Audit reporter itself. The audit daemon and dispatcher from Unix-like operating system will automatically record all system activities, but the reporter hook to the audit dispatcher can be configured with audit rules to filter those dispatcher log records and only receive part of them. This is the first level of filtering provided by SPADE. The second level of filtering is provided when SPADE is processing the log from the audit reporter. This level of filtering can allow SPADE to ignore part of the received audit log according to the group of system-calls configured by the user of SPADE. For example, SPADE can purposely ignore all I/O related system-call action, or only concentrate on process communications. This filtering helps to narrow down the noise level when we are interested in the micro-benchmarking of specific system-call activities only. We discuss the basic filtering at this point because in our manual approach, we are trying to prove the feasibility of such approach and we purposely turn on filtering for I/O even though those I/O related system-calls always contain large amounts of noise because the kernel is reading and writing at most of the time. This makes our assumption of keeping the same set of results across different trials more difficult to achieve. For easy testing, we use the filtering mechanism to ignore those I/O related system-calls at this manual comparison stages.

3.4.2 Micro-benchmarking on OPUS

As mentioned above, we classify the results into multiple categories according to the resulting provenance structure of the target system-call. The detailed description will be given in subsection 3.4.3. The classifications of system-calls for OPUS is shown in Table 3.3. As mentioned in Chapter 2, OPUS currently cannot operate on statically linked programs because OPUS works by wrapping dynamically linked libraries. Therefore, we ran the benchmarks using dynamically linked binaries, but this yields larger and noisier provenance graphs. We have manually inspected the graphs to discern common patterns from the result set. (This is related to the old version of OPUS when this work is done. When this text is written, there is already a new version of OPUS which also supports static linking of the library)

Cat 1	Cat 2	Cat 2a	Cat 3	Cat 4	Cat 5
dup	tee	chmod	setuid	symlink	rename
dup2	read	chown	setreuid	symlinkat	renameat
dup3	write	truncate	setresuid		mknod
fork	pread	ftruncate	setgid		mknodat
vfork	pwrite	creat	setregid		link
clone	fchmod	close	setresgid		linkat
kill	fchown	open			unlink
exit	fchmodat	openat			unlnkat
pipe	fchownat				
pipe2					

Table 3.3: System-calls classification for OPUS

Unlike SPADE, which tries to identify accountability and traceable provenance across distributed host, OPUS concentrates on the completeness of the provenance information describing runtime context and version of items. To provide more understanding about the state changes of objects in the operating system, OPUS maintains a framework for recording object version changes. When an object has changed and a new epoch of that object is created, the provenance information of that object splits into two series. The Provenance Versioning Model (PVM) acts as the backbone for the OPUS tools when collecting provenance in runtime. This makes OPUS more capable to collect provenance for version changes of an object, providing detailed information about the history of an object. OPUS groups the provenance around an artefact and its related epoch.

Consider a situation in which multiple processes access the same artefact concurrently. If a new system-call action is executed on this artefact, it may affect the artefact's status from the

perspective of other processes working on it. If so, then a new version of the artefact should be created for subsequent actions. Also, the PVM makes it possible to determine ordering relationships among actions because system-calls executing on the older version of an artefact always execute earlier than system-calls executing on a newer version. On the other hand, OPUS does not follow a heavyweight version-on-write model; new versions are only created when necessary to reflect changes visible to other processes.

3.4.3 Comparison of SPADE and OPUS

The classification is based on the patterns the tools produced in the resulting graph. The patterns of OPUS are similar to SPADE in general but there are two differences. Firstly, the result of OPUS contains additional elements describing version information. Secondly, both the system-call activities and general artefacts are described inside vertices for OPUS results and linked by edges. On the contrary, SPADE only describes artefacts inside vertices and records the system call information in edges. These two differences make the benchmark graph structure different in some examples, but they are generally classified following similar rules as follows. The detailed description of each category is shown in the list below.

Category 1 This category includes all those system-calls with no observable entities from SPADE or OPUS when they are tested alone.

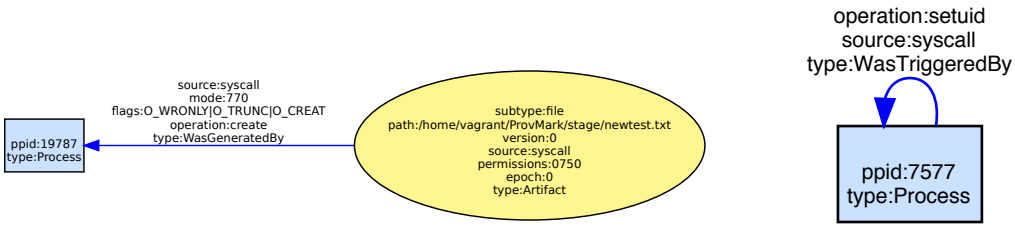
Category 2 This category includes system-calls involving one artefact (or 1 additional process). Example graphs are shown in Figure 3.8a.

Category 2a This category contains special cases of category 2 system-calls that involve version information. Example graphs are shown in Figure 3.8c.

Category 3 This category includes system-calls that involve process and inter-process communication only. Example graphs are shown in Figure 3.8b.

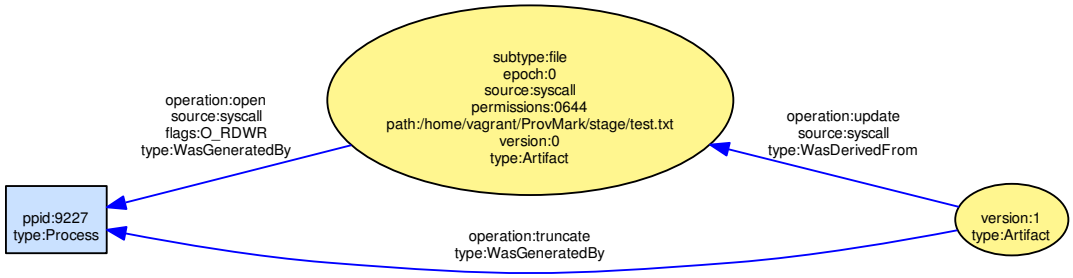
Category 4 This category includes system-calls involving one process and multiple artefacts. Example graphs are shown in Figure 3.8d.

Category 5 This category includes system-calls with irregular provenance patterns that do not fit any of the other categories. Example graphs are shown in Figure 3.8e.

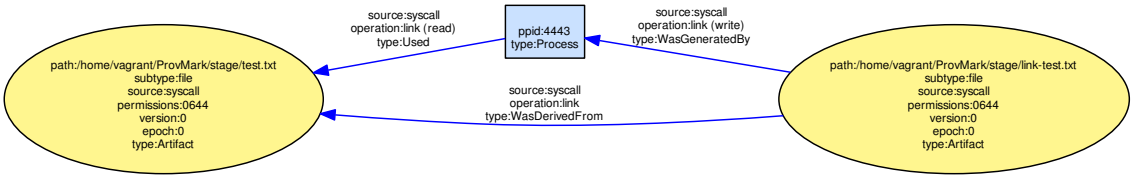


(a) Category 2

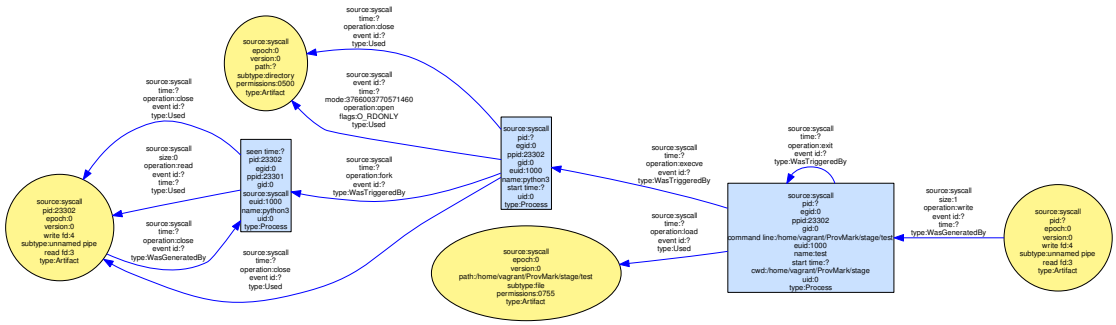
(b) Category 3



(c) Category 2a



(d) Category 4



(e) Category 5

Figure 3.8: Example graphs for each graph classification category

We have also checked the classification results with the developers of SPADE and OPUS and the results mostly match the expectation of their tools e some exceptions, which turn out to showcase that some problems exist in the tools themselves (which we will discuss later in this chapter). Last but not least, we also need to discuss the differences between the two tools in handling and preserving version information. In the classification for SPADE, only system-calls in category 2a and all the I/O system-calls related to changing of files preserve version information, while in the classification for OPUS, version information is preserved for categories 2a, 3, 4, 5. Categories 1 and 2 do not have version information preserved because they do not change the version of any artefacts. This difference in version handling also demonstrates the different perspectives of the two tools.

These classifications lay out the foundation for the fully automated expressiveness benchmarking and pattern discovery for activities sequences. It is also worthwhile to mention when we are classifying the system-calls into categories, we make use of some cypher queries to help the process. For example, if a result exists when executing the Cypher query in Code Snippet 3.5 on a system-call provenance pattern, then this system-call must belong to category 2. More sample cypher queries for category 3 and 4 are shown in Code Snippet 3.4 and Code Snippet 3.6 respectively. The above mentioned Cypher query is generally targeted to the result of SPADE. As mentioned above, OPUS treat the nodes and edges differently, thus slightly modification of the condition matching the nodes and edges will need to be changed, including the type which is representing by different wording by the tools.

From the classification results, we can see that the provenance generated by SPADE and OPUS are different because they aim to explain different perspectives of runtime behaviour. Sample benchmarks for ***creat***, ***open***, ***chmod*** and ***setuid*** are shown in Table 3.4. When we look closer to the benchmark structure of the ***creat***, ***open*** and ***chmod*** which all belongs to category 2 for SPADE result and category 2a for OPUS result. It clearly shows the different perspective of the tools. SPADE records the process (in blue rectangles) responsible for performing the system-call action. Then the specific system-call action has been recorded at the edge which links to the target artefact (which is the target file) in yellow ellipses. This match the category 2 structure we mentioned above. While OPUS result record both the system-call action and artefact as nodes in blue rectangles. Notice that these system-call has somehow touching the file and thus updating the version of the file. Thus there is additional version information exists in the OPUS result which makes it classified as category 2a. Another example is the ***setuid***, we can see that in the result of SPADE, the system-call is a circular edge point back to the same process, representing a system-call initiated by a process and performed on the process itself, which matches the behaviour of the ***setuid*** system-call to change the effective user id of the process. On the contrary, the result of OPUS contains two sets of separate events, this is mainly because both the process and system-call activities are recorded as graph

nodes by OPUS, thus it contains more than one node. Besides, as the process effective user id has been changed after ***setuid*** has been executed and the version information is preserved. OPUS treat them as two separate processes as they do not have the same user id nor version number. As a result, the separate structure in the benchmark represents the same process before and after ***setuid*** has been executed and are owned by a different user.

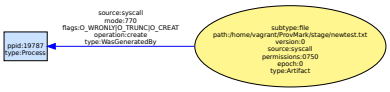
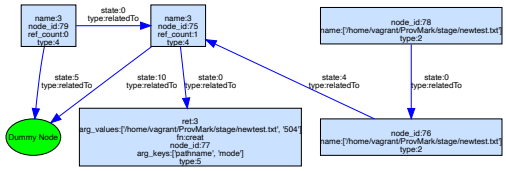
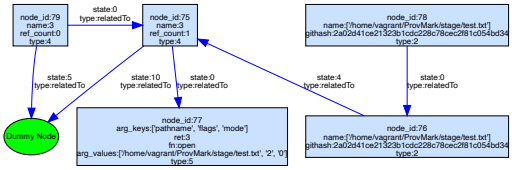
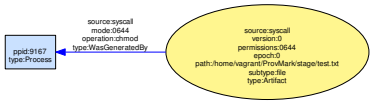
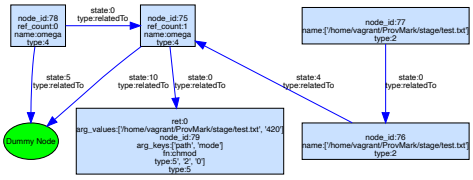
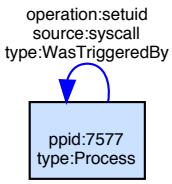
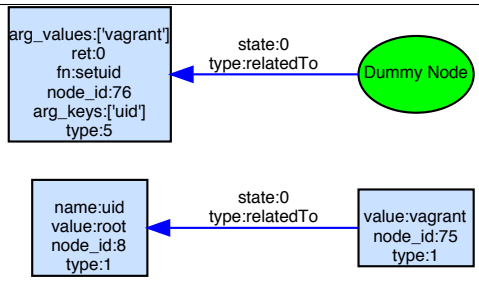
	SPADE	OPUS
creat		
open		
chmod		
setuid		

Table 3.4: Example benchmark results for SPADE and OPUS

```
MATCH
    (n1:VERTEX)-[r:EDGE]->(n2:VERTEX)
WHERE
    n1.type='Process'
    AND n2.type='Process'
    AND r.operation='<syscall>'
RETURN n1,n2,r
```

Code Snippet 3.4: Cypher query for Category 3

```
MATCH
    (n1:VERTEX)-[r:EDGE]->(n2:VERTEX)
WHERE
    (
    (n1.type='Process' AND n2.type='Artifact')
    OR
    (n1.type='Artifact' AND n2.type='Process')
    )
    AND r.operation='<syscall>'
RETURN n1,n2,r
```

Code Snippet 3.5: Cypher query for Category 2

```
MATCH
    (n1:VERTEX)-[r1:EDGE]->(n2:VERTEX),
    (n3:VERTEX)-[r2:EDGE]->(n1:VERTEX),
    (n3:VERTEX)-[r3:EDGE]->(n2:VERTEX)
WHERE
    n1.type='Process'
    AND n2.type='Artifact'
    AND n3.type='Artifact'
    AND r1.operation='<syscall>_read'
    AND r2.operation='<syscall>_write'
    AND r3.operation='<syscall>'
RETURN n1,n2,n3,r1,r2,r3
```

Code Snippet 3.6: Cypher query for Category 4

OPUS concentrates on changes in the file system. It maps multiple library calls into lower-level operations as specified by the PVM. This helps to minimise information since actions with no observable effect on the resulting provenance graph will be ignored. On the contrary, SPADE concentrates on communication between processes and artefacts across distributed hosts, and by default captures these operations without version updates. This setting illustrates the different goals of the two provenance collection tools, and the resulting provenance graphs show different information. SPADE's provenance graph shows processes communications and actions on artefacts, while OPUS's provenance graph shows more details on which process has contributed to which version of each artefact and can answer more detailed questions on the ordering of actions or which process contributed to each change. On the contrary, OPUS lacks distributed environment handling and initiator recording so cannot answer questions regarding accountability of actions if those requests are from another trusted domain.

In addition to general functionality, SPADE and OPUS also provide filtering functionality that may affect the collecting of provenance information and thus affecting the final provenance benchmark result. In general, the inclusion of all events from the audit log (SPADE) or intercepting library calls (OPUS) will make the resulting provenance graph large and hard to analyse. These large set of events always include some information such as background activities, memory exchanges and some process identifiers and timing elements. This information may be useful for certain events, but not for us to identify the existence of certain high-level action. This is because these events are ever-changing in each trial run and are generally not affected by the target system-calls that we are analysing. Both SPADE and OPUS provides different ways to handle these kinds of unrelated activities.

SPADE provides two approaches to filter results and narrow down the resulting provenance graph to a suitable size for analysis. These approaches are named as *filters* as discussed in Gehani et al. [65] and *transformers* as mentioned in Gehani et al. [63]. Similar to the two levels of filters, transformers are also operating in different stages within SPADE. The two filters work at collection time which allows the audit log filter the data sent to SPADE or provide the full information to SPADE and let SPADE ignore part of them while generating the provenance graph. Transformers work at the querying stage after the provenance graph has been generated, and only the parts concerned will be returned when querying for the result. On the contrary, OPUS handles this volatile information (background activities, memory exchange, identifiers, etc.) differently. OPUS tries to abstract application behaviour from the underlying operating system and provides a set of transformations that define every operation. The definition aims to identify the key parts of each operation which contribute to its version behaviour. Activities with no effect on the artefact version are ignored. However, the version from PVM may be inflexible and may lead to false alarms and result in missing information in the resulting provenance graph.

We can see the different approaches used by SPADE and OPUS in filtering result in different resulting provenance. It shows that difference provenance tools see the world differently and generate provenance on different perspectives for the same runtime behaviour. This phenomenon demonstrates the need to have a formalized way to compare their effectiveness to understand and compare the feasibility towards different practical usage, like identification of the existence of certain malicious activities for intrusion detection, or tracing the accountable artefacts for certain actions in forensic or auditing usage. In our manual approach, we are aiming to prove the feasibility of the expressiveness benchmarking approach and thus we want to build up the consolidation with an easier example, so we also make use of the above-mentioned filtering functionality of SPADE and OPUS to filter out background activities and other unrelated actions and some of the complicated system-calls (I/O and heavy version related) to make the test case easier to compute by manual approach. The existence of the filtering mechanism decreases the work needed for the manual comparison and micro-benchmark generation process and thus we can use these simple test case to prove the feasibility of the approach before building the automated system for the expressiveness benchmarking.

3.5 Discussion

The above resulting micro-benchmarks are generated by manual effort, which requires a comparison of foreground and background graphs of provenance data generated by the two tools for all the system-calls mentioned above. Apart from understanding the provenance data patterns for each system-call when they are processed by different provenance collecting tools, it can also provide a reference for the tools' developers and researchers to cross-check the correctness of their generation process. The easiest way is to check if the micro-benchmark generated by our manual approach is fulfilling their expectation. It is worth mentioning that our manual approach does help the developers discover problems in their provenance generation process (which they have fixed in a later version). One example is, the ***setuid*** system-call group and ***setgid*** system-call group are similar; the only difference between them is that ***setuid*** calls target users and the ***setgid*** calls target the user groups. Intuitively, they should be treated similarly because users and groups are both treated as a single integer identifier in Unix operating system, but SPADE only records explicit provenance for ***setuid*** calls when we are testing our manual approach in the SPADE version as of early 2017. The documentations of SPADE mentioned that this is a known limitation and will be implemented later. The developer has now implemented it when this text is written, and we have confirmed the new implementation when we run the experiment to test the functionality of our automated system for expressiveness benchmarking, *ProvMark*.

The above is an initial step towards identifying the provenance patterns for basic system-calls. For more advanced usage, the benchmark programs can contain more than one system-call or a series of system call actions that form an action sequence. The same logic can help to identify the minimum patterns for a specific provenance collecting tools to describe such a sequence. This advanced application will be discussed in later chapters. One major observation is, by comparing the graph manually one system-call at a time is already time-consuming, just comparing the foreground graph (Figure 3.2 and background graph (Figure 3.1) for the **`creat`** system-call to generate the corresponding provenance benchmark (Figure 3.5) which contains very few elements comparing to other system-calls already need to take more than an hour. Needless to say, it is surely a big project to consider if we need to make the approach for identifying longer activity sequences or even non-deterministic events. From the example shown in this chapter, a background graph generated from SPADE from the control program already contains around 20 edges, 20 nodes, and a hundred properties; while the foreground graph will only contain more of them because it is generated from the foreground program which does an additional system-call comparing to the background program. Although we could manually ignore those volatile values of properties, we still need to compare the graphs one by one and retrieve the micro-benchmark. A simple manual comparison for the generation of benchmark of SPADE for **`creat`** already takes roughly a few hours to do so, even if we ignore the time to configure SPADE and launch it to collect the data on the execution of the background program and foreground program. Although the time need for comparison may increase a little bit after we get used to it, it does not affect much as it is still a hard job to compare a large set of properties that are very similar to each other. Last but not least, the manual comparison may also bear a large error rate which is common in human processes. These reasons make the manual process very troublesome and are not effective to handle the whole expressiveness benchmarking given the number of system-calls and tools exist in the wild. Thus we urgently need to make use of these foundational building blocks to build up a fully automated system to do the work and extend the system to other provenance collecting mechanisms and tools beyond OPUS and SPADE.

The above manual effort illustrates the basic foundations for the understanding of different provenance patterns generated by different tools describing the same system-calls. These foundational blocks help us to consolidate the steps needed in a fully automated process and to demonstrate the feasibility of the expressiveness benchmarking approach. Before trying to build a whole automated system for expressiveness benchmarking, we manually collected provenance data for describing system-calls generated by two commonly used provenance tools, SPADE and OPUS. Also, we provided a simple comparison and classification for those system-calls to understand the differences between tools describing the same system-call activity. This simple comparison demonstrates the needs of creating a tractable approach to

formalizing correct and complete behaviour for provenance recording systems. Also, this manual effort aims to validate and demonstrate the flexibility of our work before building a fully automated system for the whole expressiveness benchmark action and self-evaluation.

As a conclusion, this manual work aims to provide qualitative answers to demonstrate the non-unified standard of different provenance tools and the feasibility of our expressiveness benchmarking approach. In our manual approach, we first launch the provenance collecting tools, then execute the control program and benchmark program separately. After that, we compare the two resulting provenance graph to retrieve the additional part from the foreground graph as the micro-benchmark for the target system-call patterns generated by these tools. When we are transforming these steps into an automated approach, there are some challenging parts, we will consider them one by one.

Launching and configuring the provenance collecting tools seems to be easy at first, but it also requires efforts to make them function correctly. Some tools like SPADE has provided a configured and reproducible virtual machine environment through vagrant (which is a utility that allows a developer to auto-build a virtual machine and configure it to a predefined state according to a set of scripts defined). But not all of the tools provide this kind of pre-configurations and setting up and configuring those tools is not a trivial task. Part of the time in the manual process has been spent to correctly configure those tools by finding the correct combination of commands and arguments for our testing. These steps done can also help us to understand the tools more thoroughly and this can reduce the time needed to build up the automated modules to handle these tools later in the automated approach.

Another challenging part is the multiple output format of the tools. When we are comparing the result of SPADE and OPUS, it is easy because both of them can output as the same Neo4J graph database format. But in the wild, there is more type of output format used by different tools. So one of the challenges for the automation process is to provide a unified provenance format and transferring different type of result into the same format for comparisons. The other challenge is related to the removal of volatile and unrelated information. As mentioned above, when we are using the manual approach, we can ignore those volatile background information and identifiers manually when we read through the nodes, edges and properties. But it is not a trivial task for an automated process to identify them because the system has no idea how to match the nodes and edges together and which of them should be filtered. To handle this problem, we need to add in a generalization process and remove those volatile values which are constantly changing in multiple executions of the same binary. This additional step can help to decrease the amount of noise from background activities and identifiers to a minimum. Last but not least, the most challenging task is the comparison of the background and foreground graph to identify the additional patterns in the foreground graph. It may seem easy to do it manually because we can easily spot the differences in the graph structure. But this is only

limited to identifying the additional edges and nodes, we still need a long time to compare the property values. In an automated approach, it needs to be done by first matching the nodes, edges and properties of the two graph together before doing the comparison and identification. As we are assuming the two graphs must have at least some graph pattern (after ignoring the noises), we need to find the optimal mapping solution to correctly map the two different graphs and identify the additional patterns. This is related to graph comparison which is a hard and complex problem to solve. This becomes the most challenging problem to be solved for the automation of the expressiveness benchmarking approach. We will discuss this graph comparison problem in the next chapters.

3.6 Conclusion

Provenance is increasingly being used as a foundation for security analysis and forensics. System-level provenance can help us trace activities at the level of libraries or system-calls, which offers great potential for detecting subtle malicious activities that can otherwise go undetected. However, analysing the raw provenance trace is challenging, due to scale and to differences in data representation among system-level provenance recorders: for example, common queries to identify malicious patterns need to be formulated in different ways on different systems. As a first step toward understanding the similarities and differences among approaches, this chapter describes a manual approach to expressiveness benchmarking. This helps us to understand the provenance patterns for the smallest meaningful units. system-calls are interpreted differently by tools, which provide a reason for formalizing them.

In this chapter, we discuss the basic approach and the manual effort taken to build up the foundation and consolidate the need for a fully automated system to compare the provenance collecting tools. We also summarize the steps and the lessons learnt from the manual approach. This helps us to identify some challenges when we need to automate the whole expressiveness benchmarking approach. Starting from the next chapter, we will be discussing some necessary bits on graph comparison which is one of the key obstacles in the automation of the framework.

Chapter 4

Graph isomorphism using answer set programming

This chapter discusses the general isomorphic graph comparison and matching problem, and how we solve the problem instances using answer set programming. This problem is a key obstacle in automating the expressiveness benchmarking. Besides, the edit distance approach for identification of graph similarity is also discussed in this chapter. The graph similarity is an important factor for the sub-graph isomorphism problem as it helps to find the optimal solution for the sub-graph mapping. We apply the edit distance approach for solving the graph similarity problem with the support of answer set programming. This can allow easy modification which is suitable for rapid prototyping situation. The proof of equivalence between the general edit distance algorithm and our adopted version using answer set programming is given later in this chapter. Lastly, some simple evaluations of our graph comparison and matching under different configuration, scales and settings are described at the end of the chapter.

4.1 Motivation

4.1.1 From manual to automated

From the manual approach, we can observe that it is feasible to retrieve provenance patterns for the minimum meaningful units, namely system-calls, by comparing the provenance graphs generated. The result can be used for monitoring two slightly different program executions. By comparing the resulting provenance graph generated from the execution of the background program and the target foreground program, we can easily identify the additional patterns from the foreground graph and extract it as the micro-benchmark. But if we consider the number of provenance collecting tools that exist in the field and the number of system-calls supported in a modern operating system, it is not practical to launch the tools one by one and compare them

manually. If we want to build up a whole framework for the expressiveness benchmarking for different collecting tools, we need to make the whole benchmarking process automated and extensible. Considering the scale of the work, it is time-consuming to verify the provenance systems when some upgrades and enhancement has been made. This step is important to ensure the new changes do not affect other components and preserve the completeness and correctness of the resulting provenance graphs. It is valuable to have an overview of changes and allows the developer of the provenance system to verify if the tools are still functioning as expected. This is one of the benefits provided by automating the benchmarking process. Also, changes in the provenance tools may affect how we use and configure them, thus we aim to make the whole automatic approach modular, allowing switching and modifying modules to support changes or newly introducing provenance tools and provenance result types.

One of the most important steps we describe in the manual approach is the graph comparison. We generate two sets of graphs, foreground graph and background graph from the control program and the target benchmark program. Two separate provenance graphs will be generated. In this case, the only difference between the two graphs should be the additional provenance data which describes the target system-call action. Although the programs are running in the same environment, there are volatile data in the graph that we defined as noise which will affect the graph comparison result. In general, the noise is provenance information that represents background information or identifiers that are not useful for identifying the existence of specific actions. It can either be some information that exists in almost all system-call actions, or information that is not related to the specific actions itself. This noise may include process IDs or timestamps which are assigned to each process and keep changing across different executions of the same programs. Other examples of noise include memory exchange information which is background activities unrelated to the specific system-call actions and is constantly changing across different program executions. If we can ignore this noise, the two graphs should be isomorphic sub-graph pairs. For the manual comparison of the graphs, we can easily ignore the noise manually and identify the provenance data sub-graph describing the target system-call. But this is not an easy process for the automated approach, because the system needs to first match the nodes and edges together one by one to find the matches before identifying the additional part, there might be multiple possible mappings and the mapping process may be subject to interference due to noise as it has no way to distinguish between noise and useful data effectively. The complexity of (sub)graph isomorphism is a hard problem in graph theory. This is one of the biggest obstacles when approaching a fully automated system. So we aim to develop a set of algorithms to make the graph isomorphism problems easier to solve by some assumption, approximation, and optimization.

4.1.2 Graph / Sub-graph isomorphism comparison

Graph Isomorphism is a hard problem which is neither known to be tractable nor NP-complete. On the contrary, sub-graph isomorphism is known to be NP-complete. Both of them are considered as hard search problems. The hardness appears to be mapping the nodes and edges together with multiple possibilities and to check if they have the same structure in general. There may exist some noise in the graph disturbing the comparison which needs to be avoided by the settings. In our automated approach, we also need something similar. First, we need to execute the same process multiple times and generalize the resulting provenance graphs into a single graph. For deterministic input, we assume that the graphs generated by repeating the same process should be isomorphic to each other. This assumption should be true when we ignore those volatile variables that distinguish the different executions. To remove these kinds of variables, we aim to compare these graphs and make a generalization to filter out those volatile variables to retrieve only the necessary bit representing certain action sequences. This requires graph comparison to discover similarities and differences between the graphs to retrieve isomorphic pairs of graphs.

In addition to the traditional graph isomorphism matching problem, we also require sub-graph isomorphism matching. When we compare the background graph and foreground graph, we assume the graphs should only differ by the additional system-calls actions in the foreground graph which are the provenance patterns describing the target system action. With this assumption, we first need to map the remaining part together to understand what to filter out to identify the resulting benchmark for that target system-call action. The isomorphism relationship exists between the background graph and a sub-graph of the foreground graph and the remaining elements from the foreground graph which does not match are the resulting benchmark. For example, Figure 3.1 is a background graph generated by SPADE while Figure 3.2 is a foreground graph generated by SPADE. SPADE monitors the execution of an empty binary to generate the background graph. On the other hand, SPADE monitors the execution of a foreground program which only contains a single ***creat*** system-call to generate the foreground graph. By matching the background graph to a sub-graph of the foreground graph, we filter out the matching elements and the result is the provenance benchmark. Sample provenance benchmark generated by SPADE to describe the provenance patterns of the ***creat*** system-call is shown in Figure 3.3.

4.1.3 Application of the graph / sub-graph isomorphism

In general, there are many real business processes requires graph representation and process. For example, a scientific database requires images to have associated provenance graphs to record alteration history. If we have these (perhaps large) collections of graphs, there are

several natural questions we might ask that require a notion of "similarity" of graphs. For example, given graph G of interest, we might want to find the k^{th} most similar graphs G' with a similar structure. Given two graphs, for example, provenance graphs tracing different runs of a process, we might want to highlight the differences, to aid debugging or performance profiling. Given a collection of graphs, we might want to automatically cluster them so that each cluster consists of similar graphs. Last but not least, the automation of our expressiveness benchmark also requires matching some similar graphs to extract general content and additional patterns as benchmark mapping for high-level action sequences. These usages all require to have a way to quantify the similarity before we can find the optimal mapping automatically.

These problems all involve computing a suitable "distance" metric between pairs of graphs, and computing an associated partial matching between the graphs that aligns the common parts. The graph edit distance is the least-cost sequence of editing steps needed to transform one graph into the other. Finding the graph edit distance is an NP-complete problem. There are (approximate or exact) algorithms for this problem, but they do not consider property graphs, in which nodes and edges may have associated key-value pairs in addition to atomic labels. Property graphs are in wide-spread use, to represent provenance, business processes, and as a data model in graph databases such as Neo4j. So we want to extend this similarity comparison approach to graph structure which the properties on nodes and edges also contribute to the similarity metrics. This means the graph comparison problems not only consider the structure of the graph, but also the properties of the graph.

In addition to the above usage, the similarity comparison approach using a suitable "distance" metric can also help to automate and compare graph for evaluation purposes. One of the goals of the automated system is to give some end-users an easy answer of which provenance generated by provenance systems provided the most complete and correct information for a specific application. Also, another capability of the automated approach is to provide provenance system developers with a reference to the differences between the provenance graph results from two trial runs before and after some changes to the provenance system. This usage requires automatic comparisons of graphs and reporting the similarity level across graphs, representing by some metric that is possible for users to understand. The usage of the similarity comparison can provide a suitable "distance" metric that allows automated evaluation of provenance graphs generated by different provenance systems or different trial executions of the same provenance systems.

4.1.4 Comparing provenance graphs in general

In general graph theory, it is possible to have two graphs appearing as different shapes in visual layout but are still an isomorphic pair of graphs which logically have the same graph structure. An example can be found in Figure 2.1a where Graph G and H is an isomorphic

pair but appearing differently. They should have a matching pair of nodes which preserve the node adjacency relationship. That is when any node a and b from graph G are adjacent to each other and node a and b map to node u and v from graph H respectively, then u and v must also be adjacent to each other. One possible combination of matching nodes that make Graph G and H an isomorphic pair can be found in Table 2.2. The problem to identify if two graphs are isomorphic pair is believed to be hard. And the problem that requires to find the matching pairs of nodes between two isomorphic graphs is also another hard question to solve. In this project, we aim to use some approximation and support from answer set programming to help solve the isomorphic graph matching problem. In this case, we assume the target given must be an isomorphic pair if all edges, vertexes, and labels relating to background information and identifiers are filtered out. Also, we assume that the property labels attached to the nodes and edges of the graph are also part of the matching criteria on top of the original isomorphic (sub)graph matching problems. This assumption allows the isomorphic (sub)graph matching problem to be *edge-preserving* and *label-preserving* as mentioned in the last section.

The method we provide in this chapter requires some level of approximation on the matching process that we try to minimize the number of mismatches through the comparison of the *edit distance* metric. In this process, we need to match two graphs which are similar but are not an isomorphic pair because they contain information that is changing across different execution of the same program. We consider these kinds of volatile information as noise and one of our targets is to filter the noise from the graphs. As we assume that the two graphs only contain a limited amount of differences, thus we try to find the exact solution of mapping by a certain level of approximation on those unmatched structures and aim to discover the optimal solution with minimum difference. As a result, the two graphs are still an isomorphic pair after those unmatched parts are removed from the graph. In this situation, we assume the unmatched structure in the optimal solutions to be noise and this is filtered out after the matching process.

In addition to the general graph isomorphism problem, we also try to solve the more specific problem, which is the isomorphic sub-graph matching problem. This is useful to match a sub-graph to a larger graph and identify the additional elements in the larger graph. This is another of the necessary components for our benchmark automation process. As mentioned in the last chapter, one of the key processes is to identify the additional elements when comparing the foreground and background graph. The major reason is we try to identify the provenance representing the additional actions executed in the trial which generated the foreground graph. The additional action executed should correspond to the target system-call. Identifying that additional part means that we can successfully filter out the provenance for the background action (which are represented by the isomorphic sub-graph matched between the two graphs) and retrieve the key provenance graph elements that represent the target system-call action. Additional uses include comparisons of structural formulas in chem-informatics for identifying

the similar chemical structure of two chemical compounds [139, 53], matching and modelling of social network or people [138], or even used for some pattern discovery for machine learning [141]. As a whole, we aim to provide a lightweight and minimum effort to benefit graph comparison with sets of properties and thus provide a way to analyse the effectiveness of provenance collecting mechanisms and the provenance graph information generated by them. This may help to fill the gap for traditional solutions on graph comparison in terms of the additional consideration of property set attached to the graph elements.

Graph comparison is not limited to comparing isomorphic (sub)graphs. It is possible that the two comparing targets do not contain an isomorphic pair. The general case is to identify the difference between the two graphs. By setting an acceptable range of difference, we can determine if we consider the two compared graphs are indeed an isomorphic pair or containing isomorphic sub-graph relationship. In addition to finding the most optimal mapping of nodes and edges by getting the least *edit distance* value, the edit distance algorithm can also act as an index to showcase how different the two graphs are. The discussion for the edit distance proposal can be found in Section 2.4. In general data science and engineering field, understanding the level of difference between two property graphs can have many uses. For example, we can classify and cluster similar operations into groups, or we could easily evaluate the efficiency of two similar actions using different approaches of intermediate steps. As a whole, *edit distance* provides reference information to show the level of difference between two directed property graphs for further processing in general.

4.2 Definitions

4.2.1 Graph definition

In general, the graph isomorphism problem can be used on both labelled and non-labelled graphs. If an isomorphism exists between two graphs G and H, then the two graphs form an isomorphic pair. In this case, the isomorphism of the two graphs is a *edge-preserving vertex bijection* no matter if they are labelled graphs or not. If indeed the isomorphic pair is labelled graphs, then this pair of graphs is also *label-preserving* in their bijection relationship. We are concerned with labelled graphs which contain properties attached to vertices and edges. This is something more deep in the graph isomorphism problems which need to fulfil mapping of properties, either in the same class of variables or even values of variables. So in our project, we are more aiming for both *label-preserving and edge-preserving isomorphism* which concentrate on label classes (which is the volatile noise that we want to avoid) and values. Lastly, the sub-graph isomorphism forms a special case which has a complexity of NP-complete. The sub-graph isomorphism defines a similar relationship to the general graph isomorphism but the

goal is different. The graphs' relationships and all the invariants of the (sub)graph isomorphism problems that are used in our automated approach are formally defined in Section 2.4.

As mentioned above, the generated provenance graphs always contain some noise because of background activity or random identifiers. If we ignore this information, the remainder of the graph should be the same across different trial executions. When working on the automating process, we want to filter out this information as much as possible to make the resulting benchmark as accurate as possible for describing the target activity sequences. Thus we introduce graph generalization steps before comparing and generating the final benchmark. By execution the foreground and background program multiple times, we get a set of *background graphs* (or *BG* in short) and a set of *foreground graphs* (or *FG* in short). The member graph of the two sets should be similar to each other and should only be different in those background activities and random identifiers which are considered as noise. Thus, comparing them and filtering out the difference should help us deduce a common sub-graph from every member of the set and produce a generalized graph which filters out most of the noise. This process will be done on both sets of graphs which are generated from multiple executions of the respective program. The definitions below show the generalization process of the two graph sets.

$BG = \{BG_1, BG_2, ..., BG_{n-1}, BG_n\}$
$FG = \{FG_1, FG_2, ..., FG_{n-1}, FG_n\}$
where BG_n and FG_n is the provenance graph collected and generated by the chosen provenance system on the n^{th} trial execution of the *background or foreground program*.

We assume that if we ignore those volatile variables which we call *noise*, like time-stamp or process ID which try to distinguish each operation, the graphs should by almost the same as each other in terms of vertices, edges, and properties. The assumption is defined for the whole graph comparison problems.

$BG_1/noise_1 = BG_2/noise_2 = ... = BG_{n-1}/noise_{n-1} = BG_n/noise_n$
$FG_1/noise_1 = FG_2/noise_2 = ... = FG_{n-1}/noise_{n-1} = FG_n/noise_n$

In general, provenance graphs always contain attributes and properties that are volatile. So it is possible that the property values defined are not the same in different graphs representing the same execution. Thus, our invariants for the isomorphic (sub)graph problems need to be a little bit relaxed in the property values, but the property keys still need to match in general. The looser definition fits the actual behaviour of provenance collection across real executions. It is a generalization of the existing isomorphic (sub)graph problem as there exist cases where even if some of the property values do not match, we ignore them and still count them as structural

matching. For handling this problem, we cannot simply compare the structure, we instead look for the closest match in terms of property value because we cannot predict if the property value is the same or different in some compared candidates. Also, most of the existing algorithms concentrate on attributed graphs with a single attribute exists in edges. In our case for handling provenance graphs, there always exist a large set of property values attached to both vertexes and edges. This is one of the problems that are not much discussed in the research field.

4.2.2 Answer Set Programming

In this work, we make use of Answer Set Programming (ASP) [97, 95] to help solve the isomorphic (sub)graph comparison and matching problems. Its primary aim is to solve difficult search problems, like problems in the NP-hard range fast and efficiently by logic programming. As mentioned in Section 2.5, we are aiming to match isomorphic graph pairs with random property values together. This action requires an intensive search for all possible matches of the vertexes, edges and property keys. Besides, calculating the edit distance in terms of the change needed for the property values is another example of a hard search problem. With the help of Answer Set Programming, we turn the graph and the matching criteria into a set of *Datalog* logical statements and use *Clingo*, the ASP solver from the *Potsdam* Answer Set Solving Collection (*Potassco*), to help us determine the optimal solutions for the matching more efficiently. The solutions to the problem then correspond to models of the combined theory and facts. One of the deciding factors for the graph matching is optimizing the matching of vertices and edges with properties. In real execution, there exists a lot of noise and unpredictable values such as timestamps that will affect the optimization results. So, in addition to the consideration of the graph structure itself, the *closest proximity* of the properties becomes another optimization factor that needs to be considered. The *closest proximity* approach choose the solution with the least amount of mismatches as the matching solution, which gradually decrease the time to solve this complex problem by cross-matching all solutions.

Datalog is a *Prolog*-like syntax used in *Clingo*, which is often used to represent relational data or multiple set structure in logic programming (as well as databases [2] and networking [68]). We need to convert the graph into *Datalog* syntax before we can provide it to *Clingo*. This is mainly because most provenance systems use a different format to represent and store provenance graphs. To make those provenance graphs able to work in *Clingo* which are only compatible with *Datalog*, the process to transform them to *Datalog* is necessary. *Datalog* defines the graph structure by defining the nodes (vertexes), edges and labels set of a graph into multiple lines of logical propositions. Each element from the three sets is represented by one proposition statement. Thus it is easy and straightforward to transform from common graph type into *Datalog* format by identifying the three sets of elements in the graph. Illustration of the format for those statements is shown in Code Snippet 4.1.

```
1  Node n<gid>(<nodeID>,<label>)
2  Edge e<gid>(<edgeID>,<srcID>,<tgtID>,<label>)
3  Label l<gid>(<nodeID/edgeID>,<key>,<value>)
```

Code Snippet 4.1: *Datalog* graph format

We assume a fixed string `gid` used to uniquely identifies a given graph. Each node n is represented as a fact $n_{gid}(n_{id}, lab(n_{id}))$ where $n_{id} \in V$ is a vertex identifier. Likewise, each edge $e = (n_1, n_2)$ is represented as a fact $e_{gid}(e_{id}, src(e_{id}), tgt(e_{id}), lab(e_{id}))$ where $e_{id} \in E$ is an edge identifier (corresponding to the pair (n_1, n_2)). Finally, if a node or edge x has property key-value pair p with value s, we represent this with the fact $l_{gid}(x_{id}, p, s)$ where x_{id} is the identifier of the vertex or edge x. Code Snippet 4.2 contains a *Datalog* representation for Figure 4.1 as an example.

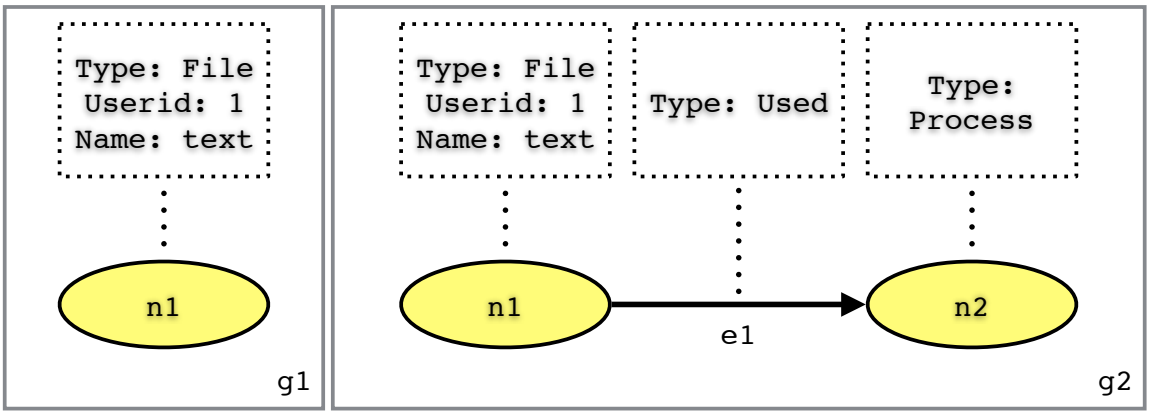


Figure 4.1: Sample Graphs for *Datalog* demonstration

```
1  ng1(n1,"File").
2  pg1(n1,"Userid","1").
3  pg1(n1,"Name","text").
4
5  ng2(n1,"File").
6  ng2(n2,"Process").
7  pg2(n1,"Userid","1").
8  eg2(e1,n1,n2,"Used").
9  pg2(n1,"Name","text").
```

Code Snippet 4.2: *Datalog* representation for Figure 4.1

As mentioned above, we are focusing on two types of (sub)graph isomorphism problems. One is the generalization problem, which compares members of provenance graph sets generated from multiple executions of the same program. In this case, we are assuming that these graphs should be isomorphic when all the noise in the graph has been ignored. So the optimal solution to match those very similar graphs should be chosen with the *closest proximity* so that the two graphs only differ in a few property values and all other parts in the graphs should be the same. This approach requires a measurement of proximity which is related to the *graph edit distance problem*, which is a general-purpose way to compare two graphs. The *graph edit distance problem* is also mentioned in Section 2.4. The calculation of the *closest proximity* is a necessary step to map those volatile variables because the graphs generated from multiple trials should be isomorphic except for the noise. So the target graph pairs should only differ from the values of those volatile variables which can be mapped with a single in-place modification. This modification operation should be the only contribution to the cost and it should be minimized when mapping two correct node/edge pairs. In particular, the generalization problem is a special case of graph edit distance where deletion or in-place modifications of properties have cost 1, insertions have cost 0, node or edge insertions have cost 0, and node or edge deletions are not allowed (i.e. have infinite cost). When we limit the associated cost of those properties to 1, we are calculating how many different values exist in the graph pair, which also shows how close the two graphs are. As defined above, the graph members from the same set should be as close as possible and only differ in a small number of property values from noises.

The other problem is the discovery of a sub-graph which is isomorphic to another graph. As the compared graphs are not the same or having that *close proximity* as the above case, it is necessary to have some approximation on top of the matching process. The requirement for this second problem is to find a sub-graph which forms an isomorphic relationship with the other graph. As this problem is known to be complex (NP-Complete), we apply some level of approximation when searching for optimal solutions. We aim to find the matching that has the fewest mismatches and adopt the result directly. This is another application of the *edit distance* metric to identify the closest match between sub-graph of the first graph and the second graph in the graph pair, which should be n*edge-preserving* and *label-preserving*.

4.2.3 Edit distance operations

As mentioned in Section 2.4, the set of basic edit operations includes insertions, deletions or in-place modifications. As we are mostly considering attributed multigraphs for the graph comparison, matching and edit distance calculation problems, the edit operations can further defined for different target elements in the graphs. The set of insertion operations inserting a node/vertex ($\mathsf{addV}(v,l)$), edge ($\mathsf{addE}(e,v_1,v_2,l)$), or property key-value pair ($\mathsf{addP}(x,k,d)$). The set of deletion operations are simliar to the insertion operations, which includes, deleting

$op \in ops$	$V_{G'}$	$E_{G'}$	$src_{G'}$	$tgt_{G'}$	$lab_{G'}$	$prop_{G'}$
$\mathsf{addV}(n,l)$	$V \uplus \{v\}$	E	src	tgt	$lab[v := l]$	$prop$
$\mathsf{addE}(e,v,w,l)$	V	$E \uplus \{e\}$	$src[e := v]$	$tgt[e := w]$	$lab[e := l]$	$prop$
$\mathsf{addP}(x,k,d)$	V	E	src	tgt	lab	$prop[x,k := d]$
$\mathsf{delV}(v)$	$V - \{v\}$	E	src	tgt	$lab[v := \bot]$	$prop$
$\mathsf{delE}(e)$	V	$E - \{e\}$	$src[e := \bot]$	$tgt[e := \bot]$	$lab[e := \bot]$	$prop$
$\mathsf{delP}(x,k)$	V	E	src	tgt	lab	$prop[x,k := \bot]$
$\mathsf{altP}(x,k,d)$	V	E	src	tgt	lab	$prop[x,k := d]$

Table 4.1: Edit operation semantics

a node/vertex ($\mathsf{delV}(v)$), edge ($\mathsf{delE}(e)$), or property key-value pair ($\mathsf{delP}(x,k)$). Lastly, the set of in-place operations only contains one operation which is the alteration of the property key-value pair ($\mathsf{altP}(x,k,d)$) because both edges and vertexes are either match or not match and thus can only be deleted or inserted if they are not matched in the two graphs. Following the graph definition in Section 2.4, we further define the the effect of each of the above operations which is the only legal operations to be included in ops. Table 4.1 shows the meaning and effect of each operation $op \in ops$ on a graph. We write $G = (V_G, E_G, src_G, tgt_G, lab_G, prop_G)$ for the graph before the edit and $G' = (V_{G'}, E_{G'}, src_{G'}, tgt_{G'}, lab_{G'}, prop_{G'})$ for the updated graph after applying any operation to G ($op(G)$).

4.3 Isomorphic graph matching by Clingo

In general graph isomorphism problems, the key requirement is to make sure that the two graphs are isomorphic pairs. But our work includes some approximation by allowing some unmatched graph structure. The optimal solutions for the mapping should have the least mismatched structure among the graphs. After finding the optimal solutions, we consider all the remaining mismatch part to be noise and will filter it out. In this section, we aim to provide works on the *Clingo* coding, trying to find an optimal solution for the pairs of graphs which have minimum mismatch among them. This can be solved by calculating the edit distance among all the possible matchings and find the optimal solution by choosing the matching with the least edit distance value.

4.3.1 Simple isomorphic graph matching

We first demonstrate the simplest problem with the least constraints. We ignore the property labels and just consider the nodes and edges of the graph candidates in this problem. The *Clingo* code shown in Code Snippet 4.3 demonstrates how to turn the simple isomorphic graph matching problem into an answer set programming specification.

Consider the code in Code Snippet 4.3, we are simply comparing the nodes and edges of the graph and try to match them together and preserving the edges. If a solution is found, it means that the graphs are indeed isomorphic. If no solution is found, it means that the graphs do not form an isomorphic pair. As a whole, this program checks if an isomorphic relationship exists among the graphs, ignoring the property labels.

This is a complete program in the *Clingo* input format; if we combine it with *Datalog* serializations of the two graphs and send it to *Clingo*, it will search for an isomorphism between G_1 into G_2. The serialization definition can be found in Code Snippet 4.1 and an example can be found in Code Snippet 4.2 which contains a sample *Datalog* serialization for Figure 4.1. These rules define graph isomorphism matching as a decision problem over graphs with atomic labels on vertices and edges. The labels have to match exactly, ignoring all other property labels for all vertices and edges. If a solution is found, the solver prints out the matching relation that witnesses the solution.

```
1  { match(X,Y) : n2(Y,_)} = 1 :- n1(X,_).
2  { match(X,Y) : e2(Y,_,_,_)} = 1 :- e1(X,_,_,_).
3
4  { match(X,Y) : n1(Y,_)} = 1 :- n2(X,_).
5  { match(X,Y) : e1(Y,_,_,_)} = 1 :- e2(X,_,_,_).
6
7  :- X <> Y, match(X,Z), match(Y,Z).
8  :- X <> Y, match(Z,Y), match(Z,X).
9
10 :- n1(X,L), match(X,Y), not n2(Y,L).
11 :- e1(E1,_,_,L), match(E1,E2), not e2(E2,_,_,L).
12
13 :- n2(X,L), match(X,Y), not n1(Y,L).
14 :- e2(E1,_,_,L), match(E1,E2), not e1(E2,_,_,L).
15
16 :- e1(E1,X1,_,_), match(E1,E2),
17      e2(E2,Y1,_,_), not match(X1,Y1).
18 :- e1(E1,_,X2,_), match(E1,E2),
```

```
19      e2(E2,_,Y2,_), not match(X2,Y2).
20
21  :- e2(E1,X1,_,_), match(E1,E2),
22      e1(E2,Y1,_,_), not match(X1,Y1).
23  :- e2(E1,_,X2,_), match(E1,E2),
24      e1(E2,_,Y2,_), not match(X2,Y2).
25
26  #show match/2.
27
28  %Datalog serialization of the two graphs goes here
```

Code Snippet 4.3: Clingo code for simple isomorphic graph matching

The list below shows more detailed descriptions of the *Clingo* code:

- Lines 1 and 2 states that for each node and edge in G_1, there should be exactly one node in G_2 linked to it by the `match` predicate. These rules have the effect of defining a search space for possible solutions: all possible relations matching each node and edge in G_1 to a corresponding node and edge in G_2. The underscores are wild-cards, indicating that the additional arguments of the predicates are ignored here.

- Lines 4 and 5 are the opposite of lines 1 and 2, which state that for each node and edge in G_2, there should be exactly one node in G_1 linked to it by the `match` predicate. These two lines complete the two-way definition to ensure G_1 and G_2 are isomorphic to each other.

- Lines 7 and 8 states that it should be impossible to have two different nodes (or edges) X and Y such that both X and Y are mapped to Z, or vice versa. That is, the matching relation needs to be a one-to-one function from nodes and edges of G_1 to those of G_2.

- Lines 10 and 11 say that each node/edge in G_1 needs to be matched to a node/edge in G_2 with the same label. That is, line 10 says it is impossible that node X in G_1 has label L and is matched to a node Y and Y does not have label L.

- Lines 13 and 14 are the opposite of lines 10 and 11, which say that each node/edge in G_2 needs to be matched to a node/edge in G_1. Similar to lines 10 and 11. These two lines complete the two-way definition to ensure G_1 and G_2 are isomorphic to each other.

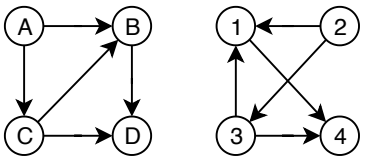
- Lines 16–19 say that if an edge E_1 in G_1 is matched by another edge E_2 in G_2, then the source and target of E_1 should be matched to the source and respectively target of E_2. Again, this has to be formulated as a constraint.

- Lines 21–24 is the opposite of lines 16–19, which say that if an edge E_2 in G_2 is matched by another edge E_1 in G_1, then the source and target of E_2 should be matched to the source and respectively target of E_1. Again, these lines complete the bijection relationship among two directions of the graph isomorphism.

- Line 26 is a directive instructing the solver to print out the definition of the `match` relation whenever a solution is found. This is what we are interested in, not just whether such matching exists.

- Finally, line 28 is the end of the main program. The *Datalog* serialization of the two target graphs is expected to be inserted here. This is an important step because *Clingo* needs the graph data to do the real logical isomorphic matching. This statement exists in line 28 as just a comment denoting the location for inserting the real graph candidates.

Figure 4.2 shows a sample isomorphic graph pair with their Datalog representation (which is reusing the example graphs and mappings shown in Section 2.4. The type V and E are dummy type labels representing Vertex and Edge respectively as we assume all directed multigraph to have a label for each element.



Graph G Graph H

```
n1("A","V"). n1("B","V"). n1("C","V").
n1("D","V"). e1(e1,"A","B","E").
e1(e2,"A","C","E"). e1(e3,"B","D","E").
e1(e4,"C","B","E"). e1(e5,"C","D","E").

n2(1,"V"). n2(2,"V"). n2(3,"V"). n2(4,"V").
e2(e6,1,4,"E"). e2(e7,2,1,"E"). e2(e8,2,3,"E").
e2(e10,3,1,"E"). e2(e9,3,4,"E").
```

Figure 4.2: Example of isomorphic graph pair

We try to demonstrate the real execution for Code Snippet 4.3 on the graphs shown in Figure 4.2. We first attached the Datalog representation of the graph pair to the end of Code Snippet 4.3 and execute it in Clingo (Clingo code can be executed by Clingo application in the host machine or through the Clingo online demo webpage[1]). The result is shown in Figure 4.3 and are summarized in Table 4.2. We can see that the mapping is in both directions as isomorphic graph pair means that there is two bijection functions mapping for vertexes and edges for the two graph involved. The result demonstrates that the given Code Snippet 4.3 does successfully identify the matching pairs of vertexes and edges in the two graphs and thus is confirmed that the two graphs G and H are indeed an isomorphic pair with a set of matching

[1]https://potassco.org/clingo/run/

elements. This proves the correctness of the sample mapping given in Table 2.2 which is shown in Section 2.4.

```
match(1,"B") match(3,"C") match(4,"D") match(2,"A")
match("D",4) match("B",1) match("C",3) match("A",2)

match(e7,e1) match(e8,e2) match(e6,e3)  match(e10,e4) match(e9,e5)
match(e3,e6) match(e1,e7) match(e2,e8)  match(e4,e10) match(e5,e9)
```

Code Snippet 4.4: Clingo code for simple isomorphic graph matching

Figure 4.3: Execution result for Code Snippet 4.3 with graph shown in Figure 4.2

Graph G Node	Graph H Node	Graph G Edge	Graph H Edge
A	2	e1 (A → B)	e7 (2 → 1)
B	1	e2 (A → C)	e8 (2 → 3)
C	3	e3 (B → D)	e6 (1 → 4)
D	4	e4 (C → B)	e10 (3 → 1)
		e5 (C → D)	e9 (3 → 4)

Table 4.2: Mapping element set of Graph G and Graph H in Figure 4.2

4.3.2 Filtering non-isomorphic graphs

We mentioned that we always assume that the two graphs are isomorphic pair after noise filtering because the provenance graphs are describing the same action sequences. This assumption is true in most cases. But there exist some scenarios which we want to identify if the two graphs are isomorphic before the matching process. One example of this can refer to our original motivation which is the expressiveness benchmarking of the provenance collecting tools. To reduce the amount of noise in the graph, we are aiming to provide some generalization process before doing the real benchmarking process to identify the additional patterns representing the target action sequences. But there are some additional considerations needed in the execution process of multiple provenance collecting sessions. Most of the provenance collecting tools aim to be started together with the system and record all activities across the active session until the system is shut down. Some of them are not meant to collect provenance on a process by process basis. For this kind of provenance collecting tool, it is sometimes hard to filter out and divide the collected provenance into chunks to represent executions of the same binary files.

Forced chunking of this provenance information or continuous turning on and off of the provenance collecting tool may result in some amount of errors for the graph candidates. If there are some problematic graphs included in the candidates, the assumption of isomorphic pairs for all graphs cannot be preserved. This may result in wrong processing or waste of a large amount of time to match graph pairs which are not isomorphic at all. To handle this problem in general, we provide a way to check if the graph pair is indeed isomorphic and try to filter out as many error graphs as we could. Although we are aiming to filter out all of those erroneous graphs, it is not always possible because the isomorphic graph mapping problem is also very complex. We also add in some rules to filter out those obvious problematic graphs and reduce the chance for matching pairs of graphs which are not isomorphic and waste time and resources to find a non-existing optimal solution.

In the above-mentioned case, the correct graph should always preserve the same amount of edges, nodes, and properties because they are referring to the same execution in a different trial. The only difference should be some values in the properties, but the whole structure should keep the same. With this knowledge, we can do some basic filtering before putting them to the mapping. For example, we could count their numbers of nodes, edges, and properties and only proceed if their element numbers are matched. The code in Code Snippet 4.5 aims to map the two target graph together after some basic filtering to check if the two graphs have a chance of being an isomorphic pair.

The code in Code Snippet 4.5 is very similar to the code shown in Code Snippet 4.3 for the simple isomorphic graph matching problem. This code can be considered as a more specific version of the simple isomorphic graph problem. In the original code, it can also help to solve the isomorphic sub-graph problem in addition to the general isomorphic graph matching problem because, in the scenario of isomorphic sub-graph matching, G_1 may have a set of matching nodes and edges to G_2 but not the reverse. Then it is still possible that G_1 is isomorphic to a sub-graph of G_2 which forms a pair of isomorphic sub-graphs. But in our proposed scenario, the graphs should be almost identical without considering the values in the property labels. Thus their structure should be the same and their isomorphic relationship should be a bijection relationship. For this reason, the mapping should be working in both directions to make the constraint tighter. Also, we add in the mapping for the property label (ignoring the value) to increase the accuracy of the identification of those erroneous graphs which should be non-isomorphic to other graphs. As a result, ProvMark makes use of the slightly modified code from Code Snippet 4.3 to identify possible isomorphic pairs within the set of generated provenance graphs with edge-preserving and label preserving properties. The graphs that are not isomorphic to other graphs in the set have a high chance to be erroneous graphs that exist randomly. These erroneous graphs are filtered out to increase the correctness and completeness of the generalization process.

```
1  { match(X,Y) : n2(Y,_)} = 1 :- n1(X,_).
2  { match(X,Y) : n1(X,_)} = 1 :- n2(Y,_).
3
4  { match(X,Y) : e2(Y,_,_,_)} = 1 :- e1(X,_,_,_).
5  { match(X,Y) : e1(X,_,_,_)} = 1 :- e2(Y,_,_,_).
6
7  :- X <> Y, match(X,Z), match(Y,Z).
8  :- X <> Y, match(Z,Y), match(Z,X).
9
10 :- n1(X,L), match(X,Y), not n2(Y,L).
11 :- n2(Y,L), match(X,Y), not n1(X,L).
12
13 :- e1(E1,_,_,L), match(E1,E2), not e2(E2,_,_,L).
14 :- e2(E2,_,_,L), match(E1,E2), not e1(E1,_,_,L).
15
16 :- e1(E1,X1,_,_), match(E1,E2),
17         e2(E2,Y1,_,_), not match(X1,Y1).
18 :- e1(E1,_,X2,_), match(E1,E2),
19         e2(E2,_,Y2,_), not match(X2,Y2).
20
21 :- l1(X,K,_), match(X,Y), not l2(Y,K,_).
22 :- l2(Y,K,_), match(X,Y), not l1(X,K,_).
23
24 % Display
25 #show match/2.
26
27 %Datalog serialization of the two graphs goes here
```

Code Snippet 4.5: Clingo code to check if two graphs are pair of isomorphic sub-graphs

The list below shows more detailed descriptions of the *Clingo* code:

- Lines 1, 4, 7–10, 13, 16–19, 24–27 are the same set of code in Code Snippet 4.3 which is a general mapping for both the isomorphic graph / sub-graph problem.
- Line 2 and line 5 are the reverse mappings for line 1 and line 4, which state that for each node and edge in G_2, there should be exactly one node and edge in G_1 linked to it by the `match` predicate. This rule has the effect of defining a search space for possible solutions: all possible relations matching each node and edge in G_2 to a corresponding

node in G_1.

- Line 11 and line 14 are the reverse mappings for line 10 and line 13, which say that each node/edge in G_2 needs to be matched to a node/edge in G_1 with the same label. That is, line 11 says it is impossible that node Y in G_2 has label L and is matched to a node X and X does not have label L.

- Line 14 is the reverse mapping for line 13, which say that each edge in G_2 needs to be matched to an edge in G_1 with the same label.

- Finally, Lines 21–22 is the additional mapping for property labels, which say that each label in G_1 needs to be matched to a label in G_2 which should be attached to the same node/edge which is mapped together in G_1 and G_2, and vice versa. In other words, if G_1 is isomorphic to G_2 with optimal matching given by isomorphism relationship function f, then the specification is satisfiable using $match(x,y)$ defined as $f(x) = y$, and if G_1 is not isomorphic to G_2 then there will be no optimal solution and the result is "unsatisfiable"

4.3.3 Isomorphic sub-graph matching

As we mentioned above, apart from the simple isomorphic graph matching problem, we also aim to solve the more complicated isomorphic sub-graph matching problem. It is known that the simple isomorphic sub-graph matching is NP-complete [39, 45]. One of the key points is, even if one graph is isomorphic to a sub-graph of another, it is still very hard to find the matching pairs as the search space for the matching could be huge. This makes this specific problem seem to be even harder than the general isomorphic graph matching problem. Here we try to make use of some approximation, together with the additional consideration of the key-value property labels attached to the nodes and edges to help us to find an optimal matching solution. An optimal solution should have the least edit distance, following the cost definition back in Section 2.4. The approximation we applied in this situation concentrates on the key-value property labels as we assume that the amount of volatile variables is minimum and so it provides the least noise which will affect the calculation of edit distance. We will keep the requirement that labels of vertexes or edges need to match exactly, at the same time we allow vertices and edges to have key-value properties that can match approximately. As in most scenarios, property values in provenance graphs also show information about the operations which should be almost the same except for some volatile variables. That matches our need in handling the large set of provenance properties attached to vertices and edges in a real scenario.

```
1  { match(X,Y) : n2(Y,_)} = 1 :- n1(X,_).
2  { match(X,Y) : e2(Y,_,_,_)} = 1 :- e1(X,_,_,_).
3
4  :- X <> Y, match(X,Z), match(Y,Z).
5  :- X <> Y, match(Z,Y), match(Z,X).
6
7  :- n1(X,L), match(X,Y), not n2(Y,L).
8  :- e1(E1,_,_,L), match(E1,E2), not e2(E2,_,_,L).
9
10 :- e1(E1,X1,_,_), match(E1,E2),
11      e2(E2,Y1,_,_), not match(X1,Y1).
12 :- e1(E1,_,X2,_), match(E1,E2),
13      e2(E2,_,Y2,_), not match(X2,Y2).
14
15 #minimize { LC,X,K : label_cost(X,K,LC) }.
16 label_cost(X,K,0) :- l1(X,K,V), match(X,Y),
17                                  l2(Y,K,V).
18 label_cost(X,K,1) :- l1(X,K,V1), match(X,Y),
19                                  l2(Y,K,V2), V1 <> V2.
20 label_cost(X,K,1) :- l1(X,K,V), match(X,Y),
21                                  not l2(Y,K,_).
22
23 #show match/2.
24
25 %Datalog serialization of the two graphs goes here
```

Code Snippet 4.6: Clingo code for simple isomorphic sub-graph matching with minimum mismatch

The code specified in Code Snippet 4.6 is the code considering the property values attached to nodes and edges when we are handling the graph isomorphism comparison and matching. As it is possible to have multiple solutions due to the uncertainty of some non-deciding factors in the provenance graph, this new approach provides an optimization constraint, essentially it says to search over all of the possible solutions to the first specification and finds a least-cost one (closest proximity in general). This is accomplished using a general branch-and-bound algorithm and we are assuming each property value carries an equal weight of importance in the provenance graph.

The list below shows more detail description of the *Clingo* code:

- Lines 1–13 and lines 23-25 are the same set of code in Code Snippet 4.3 which aim to find the matching nodes and edges. The additional lines 15–21 is the additional code to provide constraints to match the key-value property labels attached to the nodes and edges for the calculation of an optimal isomorphic graph and sub-graph matching. It is worth mentioning that in this code, we did not include the lines for the opposite definition because the problem we are solving here is a sub-graph isomorphism problem which the isomorphic relationship is not bi-directional between two graphs.

- line 15 introduces an optimization constraint. It says that we wish to minimize the sum of all costs C such that `cost(X,K,C)` holds.

- lines 16–21 define the cost predicate. The cost associated with identifier X and key K is zero if X has K-value V in G_1 and is matched with an identifier Y in G_2 that has the same K-value V. If the keys both exist in the two matched nodes but are different, then the cost is 1, and likewise, if the key exists in the first graph but not the second, the cost is 1. As additional property keys may refer to important information related to the target activity sequence, thus keys that occur only in the second graph do not incur a cost; that is, it is OK for the second graph to have more key-value pairs (just like it can have more nodes or edges) as long as it doesn't conflict with corresponding key-value pairs in the sub-graph.

4.3.4 Edit distance calculation

The isomorphic (sub)graph matching problem we studied above is related to the *graph edit distance problem*, which is a general-purpose way to compare two graphs. Given two graphs, and a set of basic edit operations (e.g., insertions, deletions or in-place modifications) with associated costs, the edit distance is the minimum cost of a sequence of edit operations that transforms the first graph into the second. By definition, if two graphs are isomorphic and they only differ in some values of the volatile property labels, the optimal solution for the graph matching should have the minimum mismatch between the graphs. In particular, the isomorphic sub-graph matching problem is a special case where deletion or in-place modifications of properties have cost 1, insertions have cost 0, node or edge insertions have cost 0, and node or edge deletions are not allowed (i.e. have infinite cost). These special properties help us get the optimal sub-graph matching and allows us to identify the correct additional elements from the bigger graph (foreground graph in our approach). Although edit distance can be used for identifying the optimal solution (with minimum edit distance value) for the isomorphic (sub)graph matching problem, it can also be used as a metric for showing the level of difference between

any two graphs. In this subsection, we provide a more general edit distance calculation that is not limited to the graph isomorphism and matching problem.

In general, graph comparison is not limited to isomorphic (sub)graph pairs. It is possible that the two targets are not isomorphic. We could identify this when the code from Code Snippet 4.3 and 4.6 returns nothing as result. But it may need to take a long time to try matching all possibility before answering. In this case, we can simply use edit distance as an index to show how the two graphs differ. A clear example for the usage of edit distance as a metric can refer to one of our original motivations for the expressiveness benchmarking. One of the reasons we want to initiate an expressiveness benchmarking is we want to provide a unified way to compare the capabilities of different provenance collecting tools in different scenarios. Most of the provenance generated by those tools are displayed in graph format. By comparing those final provenance patterns (which are not guaranteed to be isomorphic to each other) generated from different tools, we can quantify on the level of difference among the tools when it is handling the same set of action sequences. Besides, if we already know that some resulting provenance is the closest to what we want, then the edit distance quantifiers can help us identify how the other provenance collecting tools behave compared to the most capable result. In general, edit distance acts as a numeric quantity to showcase the level of difference of the graphs. It can also help to classify and cluster graphs generated from the same provenance systems for different action sequences. This can help to understand how the provenance systems treat different kinds of system-calls and actions and how they group the actions.

Computing the graph edit distance (and the associated alignment of the two graphs) is a non-trivial search problem; even though isomorphic sub-graph matching is NP-complete, in practice it makes sense to use the more specialized code above to solve the (sub)graph isomorphic matching problems because it imposes further constraints on the search space that a smart solver can use to improve performance in many cases. However, we also sometimes want to compare two arbitrary graphs where we do not have any reason to believe that one is structurally isomorphic to a sub-graph of another. An example is the case mentioned above when we want to compare the benchmark result patterns generated from different provenance collecting tools for the same set of action sequences. Another example is the case which we want to classify and group the benchmark result patterns for different system-calls generated from the same provenance collecting tools, like the sample classification mentioned in Chapter 3.

Code Snippet 4.7 shows our code for getting the minimum edit distance value itself. As mentioned above, the general edit distance problem may be used for quantifying the difference between the two graphs. We do not use any `#show` directive to display the optimal matching result; however, it is possible to do so by adding some predicates like the last two examples. By showing the optimal matching solution like the one mentioned in the last two subsections,

we could understand which (sub) graph matches the graph pair. By showing the operations (insertion, modification, and deletion), we can know what action has been done and which parts are different between the graphs.

```
1  { match(X,Y) } <= 1 :- n1(X,_), n2(Y,_).
2  { match(X,Y) } <= 1 :- e1(X,_,_,_), e2(Y,_,_,_).
3
4  :- X <> Y, match(X,Z), match(Y,Z).
5  :- X <> Y, match(Z,Y), match(Z,X).
6
7  :- n1(X,L), match(X,Y), not n2(Y,L).
8  :- e1(E1,_,_,L), match(E1,E2), not e2(E2,_,_,L).
9
10 :- e1(E1,X1,_,_), match(E1,E2),
11     e2(E2,Y1,_,_), not match(X1,Y1).
12 :- e1(E1,_,X2,_), match(E1,E2),
13     e2(E2,_,Y2,_), not match(X2,Y2).
14
15 add_node(Y,L)   :- n2(Y,L), not match(_,Y).
16 add_edge(Y,S,T,L) :- e2(Y,S,T,L), not match(_,Y).
17 add_key(Y,K,V) :- l2(Y,K,V), add_node(Y,_).
18 add_key(Y,K,V) :- l2(Y,K,V), add_edge(Y,_,_,_).
19 add_key(Y,K,V) :- l2(Y,K,V), match(X,Y), not l1(X,K,_).
20
21 remove_node(X) :- n1(X,_), not match(X,_).
22 remove_edge(X) :- e1(X,_,_,_), not match(X,_).
23 remove_key(X,K) :- l1(X,K,_), remove_node(X).
24 remove_key(X,K) :- l1(X,K,_), remove_edge(X).
25 remove_key(X,K) :- l1(X,K,_), match(X,Y), not l2(Y,K,_).
26
27 update_value(X,K,V1,V2) :- l1(X,K,V1), match(X,Y),
28                             l2(Y,K,V2), V1 <> V2.
29
30 node_cost(Y,1) :- add_node(Y,_).
31 node_cost(X,1) :- remove_node(X).
32 node_cost(X,0) :- n1(X,_), match(X,Y), n2(Y,_).
33 edge_cost(Y,1) :- add_edge(Y,_,_,_).
34 edge_cost(X,1) :- remove_edge(X).
```

```
35 edge_cost(X,0) :- e1(X,_,_,_), match(X,Y), e2(Y,_,_,_).
36 label_cost(X,K,1) :- update_value(X,K,V1,V2).
37 label_cost(X,K,1) :- remove_key(X,K).
38 label_cost(Y,K,1) :- add_key(Y,K,V).
39 label_cost(X,K,0) :- l1(X,K,V), match(X,Y), l2(Y,K,V).
40
41 #minimize { NC,X : node_cost(X,NC);
42             EC,X : edge_cost(X,EC);
43             LC,X,K : label_cost(X,K,LC)}.
44
45 %Datalog serialization of the two graphs goes here
```

Code Snippet 4.7: Clingo code for the edit distance calculation

The major idea is shown in Code Snippet 4.7 is that the final edit distance result comparing for a different group of graphs can showcase how different it is from the others. If the two graphs have an edit distance lower than a threshold, then it can be classified into the same group. One example of this usage can be referred to our expressiveness benchmarking of different provenance collecting tools. If the final provenance benchmark patterns of the two tools have an edit distance which is lower than a pre-set threshold, we can say that these two tools behave almost the same in handling certain action sequence or system-call action. We can then say that the tools similarly handle these two system-calls.

The list below shows more detailed descriptions of the *Clingo* code:

- Lines 1–2 states that between each pair of nodes (or pair of edges) there can be at most one matching relationship. This relaxes the constraint that each node/edge in G_1 needs to be matched to a node/edge in G_2 which we used in the specifications for isomorphic sub-graph matching problem.
- Lines 4–13 are exactly as in the previous specification, and ensure that the matching is compatible with the labels and matching preserves edge relationships.
- Lines 15–19 define when an element is considered to be inserted for a given matching; Line 15 is related to node insertion, line 16 is related to edge insertion and lines 17–18 are related to property key-value pair labels insertion.
- Lines 21–25 define when an element is considered to be deleted for a given matching; Line 21 is related to node deletion, line 22 is related to edge deletion and lines 23–25 are related to property key-value pair labels deletion.
- Lines 27–28 define when a property key-value pair label is considered to be updated in-place.

- Lines 30–39 define the costs associated with each element, depending on whether the element is preserved, inserted, deleted or updated; Lines 30–32 are related to node actions, lines 33–35 are related to edge actions and lines 36–39 are related to label actions. Preserved elements attract no cost, while updates, inserts, and deletes all have cost 1.
- Lines 41–43 contain a directive that specifies that we wish to minimize the total cost, resulting from summing the costs of all nodes, edges, and properties induced by the given matching.
- Finally, similar to the last two problems, the *Datalog* serialization of the two target graphs are expected to be inserted at the end of the program marked in line 45.

Formulating edit distance as an answer set programming problem must be possible in principle (since it is NP-complete) but is tricky since we cannot easily represent edit scripts of arbitrary length, and it is non-trivial to compute an upper bound on the maximum length of an edit script linking two graphs. Instead, the approach we take is to search for partial matching of the nodes and edges of G_1 to those of G_2. The matching induces two temporary transition graphs G_{T1} and G_{T2} which represents the common substructure of G_1 and G_2. The two transition graphs should share the common set of nodes, edges, and key-value property labels. The transition between G_{T1} and G_{T2} only includes in-place updates as they share a common set of elements. In other word, the edit script must be factorizable into a sequence of deletions transforming G_1 into G_{T1}, followed by in-place updates transforming G_{T1} into G_{T2} and insertions transforming G_{T2} into G_2. The process is visualized in Figure 4.4.

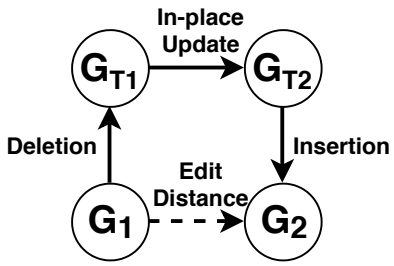


Figure 4.4: Illustration of our edit script factorization

The edit distance approach aims to find the least cost required to transform one graph into another. If all insertions, in-place updates, and deletions have the same cost, the edit distance approach can help to discover the least operations needed for the graph transformation. There are no constraints to limit the order of the operations when calculating the edit distance approach. The resulting operations path for the transformation with least edit distance can have a different order of insertion, deletion, and updates. As a result, there may exist a large set of results with the same set of operations in different orders. In our scenario, we formulate edit distance as an answer set programming problems. As it is not trivial to compute an upper bound

for the length of the edit scripts with uncertain order, we factorize and group the operations as mentioned above. Following the visualization in Figure 4.4, there are two paths to transform G_1 into G_2. The path displayed in the dotted arrow represents the edit script for transforming the graph using the general edit distance approach with no constraints of the orders of operations. The path in solid arrow represents the edit script factorized specifically for our approach to formulate with answer set programming. The two transition graphs G_{T1} and G_{T2} must have the same structure and the transformation between them should not insert or delete any elements. This assumption makes G_{T1} and G_{T2} an isomorphic pair when ignoring all the values of the property labels.

Following the defined order visualized in Figure 4.4, we further define a canonical form which has specific order requirements for each set of factorized operations in ops. The definition assists the reasoning on showing our adoption of edit distance approach using answer set programming does find the optimal solution with minimum edit distance value.

Definition 1. ***Edit script canonical form***. *An edit script is in* canonical form *if it is of the form* $del_p;del_e;del_v;alt_p;add_v;add_e;add_p$*, where:*

– del_p*,* del_e *and* del_v *are sequences of property deletions(*$\mathsf{delP}(x,k)$*), edge deletions(*$\mathsf{delE}(e)$*), and node deletions(*$\mathsf{delV}(v)$*) respectively;*

– alt_p *is a sequence of property updates (*$\mathsf{altP}(x,k,d)$*);*

– add_v*,* add_e*, and* add_p *are sequences of node insertions(*$\mathsf{addV}(v,l)$*), edge insertions(*$\mathsf{addE}(e,v_1,v_2,l)$*), and property insertions(*$\mathsf{addP}(x,k,d)$*), respectively.*

We argue that any valid edit script can be converted to a canonical one by applying a set of rewrite rules, as shown in Figure 4.5-4.7 for each of the basic operations. We first consider *marked* versions op^* of each edit operation, for example writing $\mathsf{delP}^*(x,k)$ for the marked version of $\mathsf{delP}(x,k)$. A marked operation op^* has the same effect as op when applied to a graph; the mark is only to indicate which operation is actively being rewritten. The idea here is that if we have a canonical edit script ops and wish to add a new edit operation, we use the rewrite rules to "canonicalize" $op^*;ops$. The rules are applied in order and at each step, the first matching rule is applied. Essentially, the rewrite rules consider all of the possible pairs of adjacent operations that can appear in a non-canonical form, with the first element marked. In each case, they show how to simplify the edit script by either moving the marked operation closer to the end or removing the mark. Removal can happen as a result of either cancellation of the marked operation by another operation (e.g. a delete undoing an insert), or by removing the mark once it has reached an appropriate place for it in the canonical form.

In general, the rewrite rules semantics mentioned in Figure 4.5-4.7 aims to convert an edit script into the canonical form which each of the seven supported basic operations is grouped

and performed in the predefined order. During the rewrite process, each of the operations is first marked and added to the beginning of the edit script that is already in canonical form. The marked operator will keep moving towards the end of the script until one of the two conditions is met. First of all, there is a catch-all rule (as mentioned above) to simply remove the marks from the incoming operation and confirm that the current edit script is in canonical form. This rule only applies when none of the rewrite rules is applied. In other words, it means that the current edit script is already in canonical form and did not need a further rewrite. In the general case, it represents the marked operation is already reached the designated location where all edit operations in the edit script are grouped. The rewrite will also stop when a cancellation occurs. It is quite obvious that when an insertion and deletion operations on the same elements are adjacent to each other, it is safe to cancel each other out to decrease the value of edit distance because the graph does not change after a set of insertion/deletion operation of the same elements. The result of the cancellation is an empty string which does not affect the edit script, thus the edit script remains in canonical form. Apart from cancellation and exchanging the order of operations to move it towards the end, there are two special rewrite rules for the update operation of the property key-value pairs. There are no update operations for vertexes or edges because the mapping result of vertexes or edges is either match or not match and this limit the operations to insertion and deletion only. When an update operation and a delete operation of the same property is adjacent to each other, only the delete operation will remain. This is because if a delete operation of property exists, it means that either the resulting graph does not have this property or the resulting graph does not have the annotated elements for this property. Thus the update of the property values is redundant. On the other hand, when an update operation and an insert operation of the same property is adjacent to each other, only the insert operation will remain. This is because if an insert operation of property exists, it means that either the original graph does not have this property or the original graph does not have the

$$\mathsf{delP}^*(x,k);ops \longrightarrow \mathsf{delP}(x,k);ops$$

(a) Operation $\mathsf{delP}(x,k)$

$$\begin{aligned}
\mathsf{delE}^*(e);\mathsf{delP}(x,k) &\longrightarrow \mathsf{delP}(x,k);\mathsf{delE}^*(e) \\
\mathsf{delE}^*(e);ops &\longrightarrow \mathsf{delE}(e);ops \qquad \text{if no earlier rule applies}
\end{aligned}$$

(b) Operation $\mathsf{delE}(e)$

$$\begin{aligned}
\mathsf{delV}^*(v);\mathsf{delP}(x,k) &\longrightarrow \mathsf{delP}(x,k);\mathsf{delV}^*(v) \\
\mathsf{delV}^*(v);\mathsf{delE}(e) &\longrightarrow \mathsf{delE}(e);\mathsf{delV}^*(v) \\
\mathsf{delV}^*(v);ops &\longrightarrow \mathsf{delV}(v);ops \qquad \text{if no earlier rule applies}
\end{aligned}$$

(c) Operation $\mathsf{delV}(v)$

Figure 4.5: Edit script rewrite rules for deletion operations

$$\begin{aligned}
\mathsf{altP}^*(x,k,d);\mathsf{delP}(y,k') &\longrightarrow \begin{cases} \mathsf{delP}(y,k') & \text{if } x=y, k=k' \\ \mathsf{delP}(y,k');\mathsf{altP}^*(x,k,d) & \text{otherwise} \end{cases} \\
\mathsf{altP}^*(x,k,d);\mathsf{delE}(e) &\longrightarrow \mathsf{delE}(e);\mathsf{altP}^*(x,k,d) \\
\mathsf{altP}^*(x,k,d);\mathsf{delV}(v) &\longrightarrow \mathsf{delV}(v);\mathsf{altP}^*(x,k,d) \\
\mathsf{altP}^*(x,k,d);ops &\longrightarrow \mathsf{altP}(x,k,d);ops \quad \text{if no earlier rule applies}
\end{aligned}$$

Figure 4.6: Edit script rewrite rules for operation $\mathsf{altP}(x,k,d)$

$$\begin{aligned}
\mathsf{addV}^*(v,l);\mathsf{delP}(x,k) &\longrightarrow \mathsf{delP}(x,k);\mathsf{addV}^*(v,l) \\
\mathsf{addV}^*(v,l);\mathsf{delE}(e) &\longrightarrow \mathsf{delE}(e);\mathsf{addV}^*(v) \\
\mathsf{addV}^*(v,l);\mathsf{delV}(v') &\longrightarrow \begin{cases} \varepsilon & \text{if } v=v' \\ \mathsf{delV}(v');\mathsf{addV}^*(v,l) & \text{otherwise} \end{cases} \\
\mathsf{addV}^*(v,l);\mathsf{altP}(x,k,d) &\longrightarrow \mathsf{altP}(x,k,d);\mathsf{addV}^*(v,l) \\
\mathsf{addV}^*(v,l);ops &\longrightarrow \mathsf{addV}(v,l);ops \quad \text{if no earlier rule applies}
\end{aligned}$$

(a) Operation $\mathsf{addV}(v,l)$

$$\begin{aligned}
\mathsf{addE}^*(e,v_1,v_2,l);\mathsf{delP}(x,k) &\longrightarrow \mathsf{delP}(x,k);\mathsf{addE}^*(e,v_1,v_2,l) \\
\mathsf{addE}^*(e,v_1,v_2,l);\mathsf{delE}(e') &\longrightarrow \begin{cases} \varepsilon & \text{if } e=e' \\ \mathsf{delE}(e');\mathsf{addE}^*(e,v_1,v_2,l) & \text{otherwise} \end{cases} \\
\mathsf{addE}^*(e,v_1,v_2,l);\mathsf{delV}(v') &\longrightarrow \mathsf{delV}(v');\mathsf{addE}^*(e,v_1,v_2,l) \\
\mathsf{addE}^*(e,v_1,v_2,l);\mathsf{altP}(x,k,d) &\longrightarrow \mathsf{altP}(x,k,d);\mathsf{addE}^*(e,v_1,v_2,l) \\
\mathsf{addE}^*(e,v_1,v_2,l);\mathsf{addV}(v',l) &\longrightarrow \mathsf{addV}(v',l);\mathsf{addE}^*(e,v_1,e_2,l) \\
\mathsf{addE}^*(e,v_1,v_2,l);ops &\longrightarrow addE(e,v_1,v_2,l);ops \quad \text{if no earlier rule applies}
\end{aligned}$$

(b) Operation $\mathsf{addE}(x,v_1,v_2,l)$

$$\begin{aligned}
\mathsf{addP}^*(x,k,d);\mathsf{delP}(y,k') &\longrightarrow \begin{cases} \varepsilon & \text{if } x=y, k=k' \\ \mathsf{delP}(y,k');\mathsf{addP}^*(x,k,d) & \text{otherwise} \end{cases} \\
\mathsf{addP}^*(x,k,d);\mathsf{delE}(e) &\longrightarrow \mathsf{delE}(e);\mathsf{addP}^*(x,k,d) \\
\mathsf{addP}^*(x,k,d);\mathsf{delV}(v) &\longrightarrow \mathsf{delV}(v);\mathsf{addP}^*(x,k,d) \\
\mathsf{addP}^*(x,k,d);\mathsf{altP}(y,k',d') &\longrightarrow \begin{cases} \mathsf{addP}^*(x,y,d') & \text{if } x=y, k=k' \\ \mathsf{altP}(y,k',d');\mathsf{addP}^*(x,k,d) & \text{otherwise} \end{cases} \\
\mathsf{addP}^*(x,k,d);\mathsf{addV}(v,l) &\longrightarrow \mathsf{addV}(v',l);\mathsf{addP}^*(x,k,d) \\
\mathsf{addP}^*(x,k,d);\mathsf{addE}(e,v,w,l) &\longrightarrow \mathsf{addE}(e,v_1,v_2,l);\mathsf{addP}^*(x,k,d) \\
\mathsf{addP}^*(x,k,d);ops &\longrightarrow \mathsf{addP}(x,k,d);ops \quad \text{if no earlier rule applies}
\end{aligned}$$

(c) Operation $\mathsf{addP}(x,k,d)$

Figure 4.7: Edit script rewrite rules for insertion operations

annotated elements for this property. Thus the update operation does not change the value and become a redundant operation. In both cases there exists a redundant operation that increases the edit distance value, thus they are simplified and decrease the number of operations.

Lemma 1. *1. If ops maps G_1 to G_2 and $ops \longrightarrow ops'$ then ops' maps G_1 to G_2 and $|ops'| \leq |ops|$.*

2. If ops is a canonical edit script mapping G_2 to G_3 and op^* is an edit operation mapping G_1 to G_2 then $op^*;ops$ rewrites to a canonical edit script ops' mapping G_1 to G_3 with $|op;ops| \leq |ops'|$.

3. If ops is an edit script mapping G_1 to G_2, then there is a canonical edit script ops' mapping G_1 to G_2 such that $|ops'| \leq |ops|$.

Proof. 1. The proof is straightforward for each rule; in most cases, the two operations commute. The interesting cases are:

– $\mathsf{altP}^*/\mathsf{delP}$: If the updated property is immediately deleted, it has the same effect as just deleting.

– $\mathsf{addP}^*/\mathsf{altP}$: If the inserted property is immediately updated, it has the same effect as just inserting the updated value.

– $\mathsf{addV}^*/\mathsf{delV}$, $\mathsf{addE}^*/\mathsf{delE}$, $\mathsf{addP}^*/\mathsf{delP}$: If a node, edge, or property is inserted and immediately deleted, the two operations cancel out.

2. Let ops be a canonical edit script mapping G_2 to G_3, and op^* a marked edit operation mapping G_1 to G_2. Given an edit script with at most one marked operation, define the *-length of an edit script to be 0 if it contains no marked operation and $|op_1^*;ops_2|$ if it is of the form $ops_0;op_1^*;ops_2$. That is, the *-length is the length of the marked suffix of the edit script, or 0 if there is no mark. All of the rules in Fig. 4.5-4.7 decrease the *-length of the edit script, as well as preserving or decreasing the length, so we can rewrite $op^*;ops$ to a normal form. Moreover, clearly the rewrite rules preserve the order of the operations aside from the marked one, and in the cases where the mark is removed, the edit operation is in a position that is allowed in a canonical edit script (because all of the cases where a marked operation violates canonical form are covered by other rules). Thus, after rewriting to a normal form, $op^*;ops$ is a canonical edit script.

3. We proceed by induction on the length of ops. If it is empty, there is nothing to prove. Otherwise, suppose it is of the form op, ops_0. By induction, there must exist ops_0' equivalent to ops_0 with $|ops_0'| \leq |ops_0|$. Consider the marked edit script $op^*;ops_0$. By part 1, this normalizes to an unmarked edit script ops' that is equivalent to $op;ops_0'$ with $|ops'| \leq |op^*;ops_0'| = |op;ops_0'| \leq |op;ops_0| = |ops|$.

□

4.4 Configurations and evaluations

In this chapter, we have proposed using ASP and existing edit distance approach to help identify isomorphic relationships among the graphs. The purpose of the isomorphic matching includes generalizing graphs, identifying additional graph structure in a sub-graph isomorphic pair and filtering out erroneous graphs that are non-isomorphic to others which are not expected. These approaches are all based on using answer set programming to find the least cost for transforming one graph to another. In this section, we provide an evaluation of our approach using answer set programming and some additional description of the configuration of the answer set programming tool, Clingo. The purpose for the evaluation is to demonstrate the effectiveness of the approach and to compare and evaluate our approach on different kinds of graphs to justify the motivation and usefulness of the approach on solving isomorphic (sub)graph matching and discovering process on scalable graphs with a large number of property labels. For all of the test cases mentioned in this section, we are running on a virtual machine of 2 CPU, 8GB ram and 50GB of free hard disk storage.

4.4.1 Clingo configurations

As mentioned in Section 2.5, *Clingo* is a combined tool that includes the grounding tools *Gringo* and answer set solver *Clasp* and provides an abstract layer between them to pass the information internally. With this understanding, the configuration settings of Clingo are passed to the two underlying working tools. Thus the configuration settings for both the Gringo and Clasp are allowed for the Clingo tools. The major work for Gringo the grounder is to transfer those Datalog statements into propositional logic statements accepted by the Clasp the solver. It is a one to one mapping and not much could be done except for general configurations for input/output format. On the other hand, Clasp the solver provides many configurations for different strategies on the searching and look-back process to handle the isomorphic graphs matching and edit distance calculation. In this subsection, we summarize the different configuration strategies and how they affect the solving of the problems we mentioned in the last section.

4.4.1.1 Search options

There are three kinds of search options available for the Clasp solver, including the look-ahead strategy, heuristic solving strategy and random probing strategy. All of these three kinds of search option aims to provide alternative or additional techniques for the solving process. The look-ahead strategy considers the usage of failed-literal detection mentioned in Freeman [54]. It proposes to look a few more steps ahead and eliminate paths that are worse than others. This can eliminate some blocked paths in the result searching process and increase the performance

of the solver. On the other hand, the random probing strategy and the heuristic solving strategy are mutually exclusive options. Random probing introduces random decision by a certain fixed probability to increase the performance, but it will also decrease the probability of finding an optimal solution. If no random probing strategy is chosen, Clasp solver will choose to use the default static variable ordering which is a baseline decision heuristic. The solver also support some different kind of decision heuristic, including BerkMin-like [66], Siege-like [145], Chaff-like [115] and Smodels-like [150] decision heuristic.

4.4.1.2 Look-back options

The look-back technique is a kind of conflict-driven learning. It can just revert to an earlier clause when a certain level of conflict occurs and restart from the jump back point to continue the learning. Using look-back to solve the satisfiability problem (SAT) is once mentioned in Bayardo and Schrag [21]. For our usage of Clingo, we also aim to solve the SAT problem which we want to find matching pairs of nodes from the two graphs that make them an isomorphic pair (isomorphic matching) or with the closest proximity (edit distance calculation). Clingo provides certain look-back options which allow the users to apply different look back strategy to the solver when a certain level of conflict is found. The options include restarting the whole learning after a predefined amount of conflicts, shuffling the data structure before restarting or even relaxing some constraints temporarily after every restart. These options enable clasp solver to achieve a better-optimized result with a trade-off of longer processing for some of the look-back actions.

4.4.1.3 Evaluation settings

In this subsection, we are evaluating the options in solving the two problems of isomorphic graph matching and edit distance calculation. To have a fair comparison, the graph and Clingo code used for each test case of the same problem is the same. For the test cases relating to edit distance calculation, we use two graphs generated by SPADE for monitoring two different executions of the same binary file. The binary file is generated from a C program which opens a file, then writes a character to the file 10 times and closes the file afterwards. The resulting graphs contain around 25 nodes, 30 edges and almost 300 properties. At last, we transform the two graphs into Datalog format for Clingo to process. For the isomorphic graph matching test cases, we use the same graph as the edit distance calculation problem. We then transform the single graph into two Datalog descriptions by renaming the same graphs elements into different identifiers and randomly rearrange the orders of the Datalog statements to add randomization for the isomorphic graph matching problem. Basically, the two graphs are exactly the same. The expected results for all test cases are the exact matching pairs of nodes and edges for the

isomorphic graph matching problem and the edit distance values between the two graphs for the edit distance calculation problem.

4.4.1.4 Configuration evaluation

In our proposed uses in previous subsections, it is clear that our target is to compare graphs with large numbers of property labels. In the ASP solving process, all elements are grouped and forming a constraint separately. Large numbers of property labels contribute to a large set of raw data which requires more efforts to match and reach a satisfiable state with the matching nodes of the isomorphic graph pairs. In this subsection, we are evaluating if the different search or look-back options can help to increase the performance and accuracy for solving our isomorphic (sub)graph matching and edit distance calculation problem. The evaluation aims to compare if these options are helpful when they are configured on top of the default searching scheme, which provides better effectiveness in solving our hard searching problems. We will evaluate and compare the two kind of options one by one.

First of all, we compare and evaluate the search options. The test case with different options is shown in the Table 4.3 and Table 4.4.

Test Case	Look-ahead	Decision Learning	Average Time (in ms)
#S1	No	Default static ordering	56
#S3	Yes	Default static ordering	117
#S5	No	BerkMin-like [66]	60
#S7	No	Siege-like [145]	54
#S9	No	Chaff-like [115]	55
#S11	No	Smodels-like [150]	61
#S13	No	Random probing	46

Table 4.3: Test cases for different search options on isomorphic graph matching problem

Table 4.3 and Table 4.4 list all 14 test cases that we have done on the search options. The test cases in both tables are the same set of configuration on solving different problems. The first four test cases are using default static variable ordering as the decision heuristic. The only different is test cases S3 and S4 have turned on the look-ahead options. In general, the look-ahead option requires to search a few steps ahead and eliminate some bad paths. It can benefit the performance in some cases but not in our proposed problems. The main reason is the look-ahead searching will check all branches a few steps ahead and it creates a large

Test Case	Look-ahead	Decision Learning	Average Time (in ms)
#S2	No	Default static ordering	112
#S4	Yes	Default static ordering	187
#S6	No	BerkMin-like [66]	110
#S8	No	Siege-like [145]	104
#S10	No	Chaff-like [115]	106
#S12	No	Smodels-like [150]	122
#S14	No	Random probing	98

Table 4.4: Test cases for different search options on edit distance calculation problem

bottleneck for our problems. When we match the two graphs with large numbers of labels, we need to consider the labels attached to each node and edge and each of them creates a branch for checking. This results in a large number of branches in the search space for the look-ahead process which creates a bottleneck and delays the whole searching process with partially repeating actions. In general, look-ahead is more suitable for those with fewer branches to effectively eliminate worse branches. This scenario is illustrated by the result of the test cases. The remaining test cases in Table 4.3 and Table 4.4 each choose a different decision learning heuristic and compare their performance. Each test case is repeated with the same set of data for 10 times, the average time for getting the satisfiable result for each test case is shown in the last column.

Apart from the clear bottleneck exists for test cases S3 and S4 due to the inclusion of the look-ahead option, the resulting times for each of the test cases are quite close when choosing different decision heuristic. As mentioned by the developer of Clingo in their documentations, the Siege-like, Chaff-like, and random probing learning approach is better for large branches thus should be more suitable to solve the graph matching with a large set of property labels that affect the isomorphic graph matching. To show a more obvious difference for the different configurations, we have used some large sample provenance graph for the execution of the test cases. We can still see the average time using the above mentioned three approaches is shorter than the others. When we are processing for the edit distance calculation which is more complicated as it is an isomorphic sub-graph matching problem, we can see that the average time needed is longer than the time needed for solving isomorphic graph matching problem.

Next, we compare and evaluate the look-back options. The test cases with different options are shown in the Table 4.5 and Table 4.6. We are using the default static variable ordering for the decision learning option for all test cases to have a fair comparison and evaluation.

Table 4.5 and Table 4.6 list all 12 test cases that we have done on the look-back options.

Test Case	Look-back options	Average Time (in ms)
#L1	No	68
#L3	Restart 100,1.5 and Shuffle 0,0	69
#L5	Restart 100,1.5 and Shuffle 1,1	63
#L7	Restart 100,1.5 and Shuffle 5,0	67
#L9	Restart 100,1.5 and Shuffle 5,5	61
#L11	Restart 100,1.5 and Reduce-on-restart	62

Table 4.5: Test cases for different look-back options on isomorphic graph matching problem

Test Case	Look-back options	Average Time (in ms)
#L2	No	105
#L4	Restart 100,1.5 and Shuffle 0,0	102
#L6	Restart 100,1.5 and Shuffle 1,1	96
#L8	Restart 100,1.5 and Shuffle 5,0	101
#L10	Restart 100,1.5 and Shuffle 5,5	91
#L12	Restart 100,1.5 and Reduce-on-restart	93

Table 4.6: Test cases for different look-back options on edit distance calculation problem

Similar to the search options, the test cases in both tables are the same set of configuration on solving different problems. To have a fair comparison, the graph and Clingo code used for each test case of the same problem is the same. The first two test cases are a plain run with no look-back options enabled. All the remaining test cases have the restart options turned on. Restart is the most important choice for the look-back options. It enables restart and look-back when a certain level of conflict occurs in the searching and decision heuristic process. The two numbers (separated by comma) following the restart option configure the requirements for a restart. The restart occurs when $n_1 + n_2{}^i$ conflicts are found where n_1 is the first number, n_2 is the second number and i is the number of restart actions already performed for this process. Test cases L3 to L10 include shuffle options that shuffle the data structure before some restart of the process. This process introduces randomness into the data structure to avoid the restart look-back process to go into the same dead-end again. The two numbers (separated by comma) following the shuffle command configure how often the shuffling is done. The shuffle will start on the nth restart indicating by the first number. A zero means no reshuffle, a one means

reshuffle on the first restart, a two means reshuffle on the second restart, and so on. The second number indicating the frequency of reshuffle after the first reshuffle has been done. Similar to the first number, zero means no reshuffle is scheduled after the first reshuffle, otherwise, the reshuffle will repeat every nth restart indicating by the second number. As a result, test cases L3 and L4 will not reshuffle while test cases L5 and L6 will reshuffle on every restart, starting from the first one. Test cases L7 and L8 will only reshuffle once after the fifth restart and test cases L9 and L10 will reshuffle every 5 restarts, starting from the fifth restart. Test cases L3 and L4 are the default settings for Clingo and Clasp solver where no reshuffle is enabled. The remaining test cases L11 and L12 loosen some of the learned constraints temporary after every restart, which try to find a satisfiable decision with less constraint and map back to tighter constraint later on to avoid a large number of conflict on the decision path. Similar to the search options, each test case is repeated with the same set of data for 10 repetitions and the average timing for getting the satisfiable result is shown in the last column of Table 4.5 and Table 4.6. When comparing the result, there is a clear difference with or without look-back options. Although the time difference between different look-back option is not obvious, we can still observe that infrequent shuffle on restart and relaxing criteria on restart do help to decrease processing time.

4.4.2 Scalability evaluation

In the last section, we proposed using edit distance approach by answer set programming to find the optimal matches between two graphs. If the optimal match does exist, the comparing graphs form an isomorphic (sub)graph relationship. In most of the cases, the graphs involved in the edit distance approach are labelled graphs. One of the properties of the labelled graph is each of its elements can be labelled with key-value pairs. What we are proposing in this chapter and the whole provenance benchmarking automation is the ability to extend the general edit distance problem to graphs that contain a large number of property labels. Each node and edge in the graphs can be annotated with multiple labels. The resulting set of graph elements are larger than the general case. In this subsection, we are going to evaluate the scalability for our proposal of using edit distance approach by answer set programming to solve graph proximity calculation for labelled graphs with large numbers of property labels. The major target for this evaluation is to show this approach using answer set programming solver can scale up to much larger target graphs without seriously affecting the performance of the process. We separate the scalability test into two different part. The first part aims to evaluate the approach with an increasing number of properties, while the second part aims to evaluate the approach with an increasing number of nodes and edges. Table 4.7 shows test cases for the first part of our scalability test on using edit distance approach by answer set programming solver, while test cases for the second part are shown in Table 4.8.

Test Case	# of Property Labels (Graph A)	# of Property Labels (Graph B)	Average Time (in ms)
#P1	0	0	6
#P2	50	0	8
#P3	50	50	8
#P4	500	0	11
#P5	500	50	12
#P6	500	500	14
#P7	1000	0	12
#P8	1000	50	14
#P9	1000	500	17
#P10	1000	1000	19

Table 4.7: Test cases for scalability test of increasing property labels

Table 4.7 contains the 10 test cases for the scalability test of increasing property labels. To focus on the differences and scalability on the property labels, the two testing graphs have the same graph structure with 4 nodes and 10 edges. The graph pair scales from no property labels to a thousand property labels. In each test case, the average time needed to find the optimal isomorphic matching solution is recorded to demonstrate our proposed approach is possible to scale up to graphs contain a large set of property labels. The graphs are isomorphic pair when ignoring all property labels. Thus the evaluation shows how the performance of our approach is affected by an increasing number of property labels. The result is shown in the last column which each test case is repeated for 10 times. From the average time result, we can see that not only the total number of properties in the graphs will affect the time needed for the answer set programming solver to find an optimal solution, the differences in the number of properties also affect the processing time. For example, comparing test cases P6 and P7. The total number of property labels are 1000 for both cases, but the average time needed for test case P7 is much shorter. The major reason is there are no property labels in Graph B for test case 7, which make the answer set programming almost not necessary to compare property labels when mapping the isomorphic pair. This is because the only possible operation for transforming Graph A to Graph B in terms of those property labels is to delete them. On the contrary, the answer set programming will need much more effort to match the 500 property labels of Graph A and Graph B in test case P6 which requires slightly longer processing time. Although the time needed is increasing in these test cases, it is still in a very good shape which requires less than a second for processing graphs matching for a thousand property labels in each graph. This

shows that our approach using answer set programming solver to map isomorphic graphs is possible to scale up to graphs with a large set of property labels and other elements.

Test Case	# of Nodes	# of Edges	Average Time	# of Success
#P11	10	10	21 ms	10
#P12	50	50	476 ms	10
#P13	100	100	5.6 s	10
#P14	250	250	62 s	7
#P15	500	500	19 min	2
#P16	750	750	Out of memory	0
#P17	1000	1000	Out of memory	0

Table 4.8: Test cases for scalability test of increasing nodes and edges

Table 4.8 contains the 7 test cases for the scalability test of increasing nodes and edges. The test uses two graphs in each test case and scale from 10 nodes and 10 edges for each graph to 500 nodes and 500 edges for each graph. Each of the testing graphs has 1000 property labels attached randomly to its nodes and edges to have a fair comparison and evaluation. The test calculates the minimum edit distance value between the two randomly generated graphs and returns the matching pairs of elements. Again, the average time needed is recorded to demonstrate our proposed approach is also possible to scale up to graphs contain large elements set. The average time and number of successful trials of each test case are shown in the last two columns which each test case is repeated for 10 times. From the average time result, the effect on performance is more clear when increasing the number of nodes and edges comparing to the increasing of the number of property labels. The major reason is the additional number of nodes and edges make the possible mapping option increase exponentially and thus the search space also increase exponentially and requires much more time to search for an optimal solution with minimum edit distance values. When comparing the two different scalability evaluation, it is clear that the number of nodes and edges affect the performance the most, and even using our approach with answer set programming solver, it still not effective when the graph scale up to 500 nodes and edges. It is worth mentioning that the running of test cases P14 to P17 have all experience out of memory exception. As mentioned above, we have repeat each of the test cases for 10 times, test cases P14 and P15 success in some of the cases and receive an out of memory exception in other trials, while test cases P16 and P17 receive an out of memory exception for every trial, thus we cannot get a timing result. The average time for P14 and P15 do not count the fail attempts. This evaluation demonstrates that although our approach

using answer set programming solver can help to solve the isomorphic matching for attributed graphs with a large set of property labels, it still cannot scale up to graphs with too many nodes and edges. In reality, nodes in provenance graphs represent artefacts and processes. If we are just focusing on certain process behaviour and not focus on the analysis of whole system provenance, the resulting provenance graph should not contain a large set of nodes as many of the artefacts and processes are reused and do not require generating new nodes to represent them. As a whole, the numbers of artefacts and processes related to certain system behaviour are limited and will not be required to reach the extreme cases we have been evaluating here in this subsection. Thus our approach can still help and scale up to graphs with a limited amount of nodes and edges with a large set of property labels.

4.5 Discussion

Isomorphic (sub)graph comparison and matching is a complex question. Although the complexity of the general isomorphic graph comparison and the matching problem is not known, it is already proved that the subgraph isomorphism problem is an NP-complete problem [39, 45]. Thus it is even harder when the complexity of the directed graph increases and contains a large set of properties attached to each vertex and edge which form part of the deciding factors for the isomorphism condition. Recalling the motivation of this chapter, one of the major applications of this complex isomorphic graph comparison and matching belongs to the field of data provenance. In most of the cases, due to the need of details and complete information, different provenance collecting tools will capture as much information as they can and generate detailed graphs which result in high graph complexity and large numbers of properties. If we need to extract patterns from the graph for security or benchmarking, we cannot avoid direct comparison of the graphs to find isomorphic sub-graphs and to filter out unique patterns. This is the main motivation that pushes us to find a fast way to handle the problem.

By making use of the edit distance approach to find the matching with a minimum mismatch, we aim to develop a fast way to handle the problems and identify an optimal matching solution. Our original objective is to handle the provenance graph generalization and benchmarking where we can assume that the comparison and matching target must preserve the isomorphic (sub)graph relationship. Although these provenance graphs may not be very large, they may contain large numbers of attached key-value property labels in the nodes and edges. The main usage of those labels is to keep information relating to the action performed in runtime to achieve the history of tracing characteristics later on. In general graph comparison, many ways are proposed for a fast comparison of graphs but very few of them are concerned about graphs with large numbers of property labels. This allows us to provide a slightly modified edit distance approach, with the support of the answer set programming and certain levels

of approximation to help to solve the graph / sub-graph comparison and matching problems on graphs with large numbers of properties. In addition to the need of graph / sub-graph comparison and matching, we also need to have a unified way to identify the level of difference between the final provenance benchmark for different provenance collecting tool, this allows us to further extend the modified edit distance algorithm and get a unified quantifier for classifying and grouping the graphs according to their similarity. This approach not only provides us with a way to compare the results from different provenance collecting tools, but it also allows us to classify and group the results from the same provenance collecting tool. This allows us to understand how the tools treat and group different system-call actions.

In our approach, we actually treat every element in the graph as uniformly important. We weight every elements modification, insertion and deletion actions as the same. This approximation does help us decrease the processing time and overhead. Also, it does help to maximize the number of matches in the calculation. But it also creates one of the limitations. Some of the property values and other graph components may have more significant meanings in the executions which require higher attention for the analysis. This makes some of the graph components have a higher weighting in the calculation because they need to exist and certain deletion actions may not be allowed during the edit distance calculation. Out approach treats all of them the same and may accidentally perform a deletion action during the edit distance calculation because it requires lesser actions. This may result in wrong calculations and risk the fact that certain important activities in the execution will be lost. An example of that includes certain command-line arguments for the execution of the program for an interpreted language. Hassan et al. [75] has proposed a different approach named ”Graph Abstraction” which encode some semantic meaning of certain properties with a set of manual heuristics. This helps to treat graph elements with different levels of importance during the graph matching actions. Although their approach does not provide maximum edit distance faster than ours, they do handle the realistic problem of the existence of different importance level of the graph elements describing the execution of processes. Last but not least, our approach may be good to solve graph matching problems and edit distance calculation problems when we are facing the problems for comparing provenance graphs for the similar executions which we can treat all graphs elements as equal weight as most of the elements should be preserved in multiple executions. Unfortunately, this consideration may become a bigger problem when this approach is used in some generic problems which the comparing targets are no longer guaranteed to be similar and this creates one of the limitations of our approach.

4.6 Conclusion

In this chapter, we discuss one of the obstacles in transforming the manual expressiveness benchmarking to a fully automated system. The obstacle mainly related to the isomorphic (sub)graph comparison and matching problem which is a very complex problem to be solved. We summarize the way we propose to identify optimal matching among similar graph for our automated system. We also provide effort done in introducing ways to quantify the level of difference by slightly modified edit distance approach. Lastly, we evaluate the proposal with different graphs and existing approaches to show its capabilities. The contributions in this chapter not only benefit our automation of the expressiveness benchmarking, but it can also be used in general graph comparison and matching problems. As the major obstacle has been solved, we will be discussing the basic design architecture and working logic of the automated system for the expressiveness benchmarking in the next chapter.

Chapter 5

ProvMark: the automated system

This chapter will discuss the basic design architecture and working logic of the automated system, ProvMark, for expressiveness benchmarking.

5.1 Motivation

5.1.1 Provenance benchmark and formalization

One of the major research contributions of this thesis is providing expressiveness benchmarking for different provenance recording tools. The expressiveness benchmarking aims to analyse the provenance graphs generated by each of the tools when monitoring the same action sequence. As there are no unified standards, different provenance systems may generate provenance graphs in different formats. This setting increases the difficulty of comparing and analysing of the provenance graphs generated from different provenance systems. By viewing and analysing the benchmark in the same format and structure, we can decrease the difficulty of identifying the expressiveness of each of the tools in different use case scenarios. The expressiveness is defined as how correct and complete is the given provenance record in telling the story of what is happening in runtime. In certain cases, like in security forensics, the audience is more interested in the full evidence link to the originators and accountable parties for certain actions. In this case, the identity of the actors across the lifetime of an artefact is mandatory in the provenance information. This can be analysed by the completeness criteria of the expressiveness benchmarking. For example, if we provide a known action sequence to the provenance systems as testing input, we summarize the resulting provenance information and compare it with the known fact for checking. The checking can show us if all information, including but not limited to relating processes, artefacts and actors, are completely captured. This can demonstrate if the provenance collecting tools and its resulting provenance information fulfil the completeness criteria. The provenance collecting tools capable of this forensic usage must be completed on

actor identity information to provide a full chain of evidence. This example usage identifies the need to have a unified way to analyse the expressiveness of the generated provenance and also the capabilities of each of the provenance recording tools in different scenarios. As mentioned in the last two chapters, although it is possible to do the benchmarking manually, it may be error-prone and needs a lot of human effort to do so. That is why we aim to develop a fully automated system for handling the whole benchmarking process. This should benefit both the provenance recording tools' developers in understanding, cross-checking and enhancing their tools and the end-users, including researchers on data science or database analysis, who can use ProvMark to research which tools are best suited for their use case scenario.

5.1.2 Size of system-level provenance

The error-proneness of the human manual benchmarking process is not the only reason for developing an automated tool. The other major reason is the size of the system/kernel level provenance makes it costly to analyse them manually. We understand from the example work in chapter 3, a simple provenance graph like Figure 3.7 representing the execution of a binary with a single system call already results in a big provenance graph with around 40 nodes and edges, together with 100+ property values. This problem will become more serious if we move to handle real-world program execution which consists of a large set of system calls. If we continue to use manual effort for handling provenance graphs of increasing size from real-world programs, the error rate and the resources needed will also be increased. The error rate and resources needed to handle the graph manually is directly proportional to the size of the provenance graph. The size of the provenance graph is also directly linked to the number of system calls involved in the program execution. In this case, if we are aiming to analyse the correctness of the provenance graph describing a real-world program, we need to face provenance graphs with a large set of vertexes, edges and property labels. It is almost impossible to validate and check every element and analyse if every piece of information exists and correctly represents the real execution in a reasonable time. Also, checking large groups of elements manually increases the chance for errors. Indeed, multiple errors has been raised and corrected during the manual approach mentioned in earlier chapters. It shows that the manual benchmarking approach is not scalable to handle large sets of data. This is a clear weakness because in general, system-level provenance will always contain a very large set of elements in describing what is happening in runtime. This necessitates developing a fully automated tool to take over the manual approach and provide automatic analysis and comparison of those provenance graphs. This automatic transformation can provide automatic expressiveness benchmarking on real-world programs which make it more useful among the tools' developers and end-users. In chapter 4, we also discuss the hard problems on graph comparison which are obstacles to handle large graphs automatically. After providing ways to go around the problem, the automated

expressiveness benchmarking should be able to complete in a reasonable time and also maintain correctness of the result at a suitable level. This can be measured by evaluating ProvMark result with some known training set from some security research.

5.1.3 Expressiveness comparison of provenance

In addition to the error rate and the scalability problems of the manual benchmarking approach, there are also some other minor motivations for adopting the fully automated system to the expressiveness benchmarking process. As mentioned in chapter 3, the two target audiences of ProvMark are those provenance recording tools' developers and the end-users of those tools. In the viewpoint of the developers, they know how their tools work, but they normally do not know the working logic of other tools; while the end-users of those tools do not care how the tools work, they only care about if the tools can help them in accomplishing their tasks. To allow them to achieve their own goals without knowing every detail and working logic, we provide an additional automated layer on top of ProvMark and help them to compare the capabilities of the tools in different situations without the need to understand the provenance results of the tools and how is it collected. The users can use our automated system as a black box and the automated system will show the expressiveness comparison of the provenance benchmark generated by each of the tools for the same action sequences. The user can then judge directly on how complete and correct is the resulting benchmark by viewing the comparison result. This layer avoids the need for a user to understand how the tools work in general.

5.1.4 Discovery of unexpected behaviour

In addition to the normal end-users which only need to compare tools capabilities without the need to know the running logic of each tool, there is another group of target users for the benchmarking. They are the developers of the provenance collecting tools. The provenance benchmark results are still generated by different provenance collecting tools. All those tool developers should have a clear expectation of how the resulting provenance benchmark should be. With the automated approach of the expressiveness benchmarking, it can provide the tool developers with a chance to provide large amounts of different benchmark programs with different target action sequences and get a set of provenance benchmarks. By examining the set of benchmarks, those developers can verify if their tools indeed work as their original expectation. This can help them to discover possible problems, bugs, and idiosyncrasies from their tools because our automated approach result is based on what the tool originally generates. The developers can verify those output by comparing some predefined result of their own. In some of the cases, if the developers aim to verify their tools after some code changing or bug fixing, they can match the new set of benchmarks with the set generated with the older tools and see if

there are unexpected changes. This case is similar to regression testing done in systems, with the support of graph comparison and static analysis. The ability to generate and handle large amounts of input and generate large sets of provenance benchmarks is one of the reasons for us to automate the benchmarking approach because it is almost impossible to provide the same features when it is done manually. Indeed, during the development, testing and evaluation of our automated approach, we did find some unexpected results and confirmed some were indeed bugs in the tools after discussing with the original developers.

5.1.5 Practical usage of precise provenance

Last but not least, in some usage of provenance, like auditing or forensic usage, it is important to have very precise information describing runtime events. For example, the owner change in an artefact needs to be described clearly. One of the points we mentioned earlier is the fact that different provenance collecting tools will collect and filter system call event details according to some rules. This is because it is not possible to capture all the information at once. What data are collected or filtered out are determined by the design of the tool itself which follows one or more original purpose of the tool. So in some of the tools, it may not collect enough information for some precise usage. Some precise usage requires comparing and analysing each of the elements of the graph, include vertexes, edges and property labels. One missing or wrong comparison may result in the wrong conclusion. For example, if we mistakenly miss one of the actor identifications in the provenance results, we may miss an accountable party for certain action and ruin the analysis result. This precision is not possible to do by manual approach because of the large error rate and the scalability problems. With the needed precision in the automated approach, it can help to identify if the benchmark provenance for a specific tool is *complete* enough to describe an event or to distinguish it from a similar event. It also helps to identify if it *correctly* identifies the real action that is happening at runtime with the correct accountable parties and other properties. This need for precision in the practical real-life example shows again the necessity for building up an automated approach of the expressiveness benchmarking.

5.2 Definitions

5.2.1 The automated system

We name our fully automated system for the expressiveness benchmarking as *ProvMark*. *ProvMark* automates all the steps mentioned in chapter 3 and works as a black box. The user just needs to provide the *benchmark programs* and choose the target *provenance collecting tools*. Then *ProvMark* will automatically compute and generate provenance benchmark as an output.

In this report, we sometimes use the term *provenance recording tools* interchangeably with the term *provenance collecting tools*. Both of them refer to the same group of provenance systems developed by different researchers in the provenance field, aiming to collect provenance information in different scenarios and operating systems. Examples of tools are introduced back in Section 2.1.

In the manual approach, the *benchmark programs* refer to a pair of *background and foreground programs* which are very similar to each other. Their only difference is the target action sequence that we aim to analyse. In general, both the background and foreground program contains the same code for all the background actions, while the foreground program contains some additional actions which are the target actions that we want to analyse. When we provide both of them to a provenance collecting tools and compare their provenance graph result, the additional part in the provenance graph representing the foreground program describes the trace for the target actions. This comparison and additional pattern retrieval are included in the automation process of *ProvMark* and we defined this resulting graph pattern as *provenance benchmark* which demonstrate the expressiveness for the chosen provenance collecting tools on the target actions.

One of the uses of the provenance benchmark is understanding the patterns of certain action sequences described in the basic unit of system calls in the kernel. For this reason, we choose the C language for writing the *benchmark programs* because it contains libraries that can map one to one to the kernel system calls which let us easily analyse action sequence differences down to the base level.

In the manual approach, we separated the benchmark programs into two similar foreground and background programs. When we move to the automated approach, we want to decrease the error rate and execution complexity. One of the possible problems in the automated approach is, it is hard to confirm if the foreground and background programs are a pair of programs that are very similar and only differ in the target system calls. We have no guarantee that the input follows the rule which is one of the requirements of the input program if we want some reasonable benchmark result. To solve this problem, we only accept one benchmark program as input to *ProvMark*. We instead use the CPP directive to help identify the target action sequence. The input benchmark program will be the original *foreground program* from the manual approach, with an additional `#ifdef TARGET` CPP directive statement to enclose the target action sequences. When *ProvMark* receives the program, it will compile it into two versions of binaries which are controlled by the definition of the macro keyword. For the compilation without the macro keyword defined, the circled target action sequences will be ignored, and the resulting binary is the same as compiling the background program, while the other way will result in the same binary as compiling the foreground program. This setting keeps *ProvMark* with the pairs of benchmark program to work on and easier to maintain.

In *ProvMark*, we aim to automate everything from the collecting of provenance and graph comparison to generation of the resulting provenance benchmark. As we are collecting system-level provenance from different provenance collecting tools for further processing, we need to automate the process for starting the tools and managing their input and output. We defined the term *tool handler* as a module in *ProvMark* which starts and monitors a specific provenance recording tool. Tool handler also manages settings, configuration and the corresponding input/output of the chosen provenance recording tools throughout the whole automated process. As different tools operate in different ways, there are separate tool handlers for each of the supporting tools. Currently, we are only supporting the three target tools SPADE, OPUS and CamFlow. To make *ProvMark* easy to expand and handle other provenance recording tools, the tool handler follows the same set of skeleton interface. All the tool handlers follow the same input/output format and the only difference is the different configuration values, how to start the tools and where to retrieve the resulting provenance graph. The choice of tool handler and corresponding special configuration values are changing in the global configuration files of *ProvMark*.

As mentioned above, one major reason for the expressiveness benchmarking is to provide a more general way to compare the provenance result for different provenance collecting tools which does not have a formalized standard and tends to generate different type of provenance result. Some of them generate in *Prov-JSON* or other *Prov* format, some of them generate graph in *DOT* format. Some of them even generate graphs which are stored in a graph database like *Neo4j*. To provide a general way for the comparison, we define the term *graph handler* as another module in *ProvMark*. There are multiple copies of graph handlers but in each execution of *ProvMark*, only one of them will be chosen. This is configured by the same global configuration file mentioned above. Each graph handler turns one format of provenance graph output into *Datalog* format which is the language supported by the *Clingo* which is chosen as the graph comparison tools used in *ProvMark* as mentioned in chapter 4. The existence of graph handlers makes *ProvMark* able to handle provenance graph results generated by different tools. Currently, most of the known provenance graph types have been covered by the existing group of graph handler modules.

5.2.2 Provenance collecting modules

In general, *ProvMark* works on the user level. If it needs access to a specific service in the kernel level, it will also call through an existing user-level wrapper that does the job. But, this is not the case for some of the provenance recording tools. As different tools aim to collect different types of information which may be recorded in different locations, this may include some components working in the kernel level. In this subsection, we clarify some of the working components used by some of the provenance recording tools which sometimes the

tool handler of *ProvMark* may need to communicate with.

One of the common targets for provenance recording tools to collect information about a runtime action sequence is the *Linux Security Module* or *LSM* in short. The LSM is a base framework provided in the kernel level for applying security policies to action sequence request. In other words, LSM allows or denies action sequence request by a set of *mandatory access control rules*. This framework is a standard component since version 2.6 of the kernel and has different variants in the field. There are different contributed LSM modules which are accepted by the kernel. Some provenance collecting tools capture information from the LSM to understand what action sequence has been performed or banned in runtime. To do that, they implement their *LSM hook* to register themselves to the notification list of the LSM. Once new decisions have been made by the LSM, the registered LSM hook will be notified and relay this information back to the user level where a daemon should be kept waiting and listening to this information and process them when new information arrives.

Another common target is the audit framework in the operating system. The audit framework captures target action sequence and related information in the kernels. The framework will then relay the information back to the user level which is received by the *audit daemon* module. The audit daemon will distribute the captured information through the *audit dispatcher*. Components or processes can register to the audit dispatcher to receive notification and an updated copy of the captured record once some new audit records are relayed to the audit daemon. This allows a provenance collecting tool to retrieve audit events for their provenance generation purposes.

5.2.3 Operating System coverage

Although we understand there are many different operating systems in the field, *ProvMark* currently only supports Unix-like operating system which is supported by most of the provenance collecting tools in the field as we understand. Although we only test our system in some Linux distribution, the system call definitions and C compiler used in our development and test should work in most of the Unix-like operating system, except that some modifications may need for BSD as they have a slightly different bash script model. The major development language of ProvMark is Python (version 3) which should be supported by all Unix-like operating system after installing the correct interpretation modules. The testing and evaluation of *ProvMark* in other Unix-like operating systems may be done later as a future work because as far as we understand, most of the provenance collecting tools already have at least one variant or version supporting Linux operating system.

5.2.4 Provenance and graph terminology

In the working of *ProvMark*, it should take in the benchmark program and the choice of provenance collecting tools and output a set of provenance benchmarks. This process needs to go through many intermediate steps. In this subsection, we define the terminologies used in the intermediate steps.

To avoid noises from volatile information in the provenance graph generation, we will need the generalization of graphs before comparing the background and foreground graph. For this reason, *ProvMark* generates a set of background graphs and a set of foreground graphs. The two sets of graphs will be compared to each other to filter out volatile variables and result in one graph for each set. The resulting graph is named generalized graph. There should be one *generalized foreground graph* and one *generalized background graph* after the process.

After the above process, the two generalized graphs will be compared to identify the additional components in the generalized foreground graph as the generalized background graph should be isomorphic to a sub-graph of the generalized foreground graph. The remaining components in the generalized foreground graph will be the final result, which is the provenance benchmark. It is worth noticing that as we are trying to compare and remove the isomorphic part of the two graphs, the resulting provenance benchmark should be a subset of elements (nodes, edges and properties) of the original graph. It is possible that the provenance benchmark itself is not a complete graph at all as some of the components may be filtered out. For example, the resulting provenance benchmark may contain an edge that only connects to one of the nodes and the other end connects to nothing because the node in the other end has been filtered out because it exists in both the foreground and background graph. In order to make it displayable as a graph, *ProvMark* purposely adds some *dummy elements* which make the provenance benchmark become a full graph again. These *dummy elements* will be clearly marked and it should be ignored when viewing because they are not part of the provenance benchmark. They exist because of the need to make the provenance benchmark displayable as a graph. An example of provenance benchmark with a dummy node is shown in Figure 5.1. It is the provenance benchmark for the **`close`** system call generated by the provenance collecting tool, SPADE.

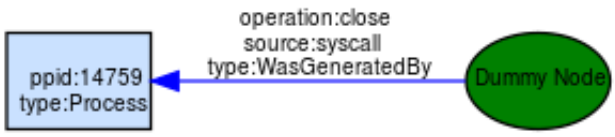


Figure 5.1: Example of dummy node in provenance benchmark

5.2.5 Assumptions

In this chapter, we show the basic design and architecture of *ProvMark*. We aim to use *ProvMark* as a black box. We only need to feed in a benchmark program and a choice of the provenance collecting tool to *ProvMark* and it will execute on its own and generate the provenance benchmark. In this initial design, we assume that all the benchmark program only contains deterministic events which keep the same in every execution. This should make the resulting provenance benchmark the same even if we execute *ProvMark* multiple times with the same set of input. This is one of the big assumptions we defined in the basic architecture of *ProvMark* because non-deterministic input will create multiple different foreground graph and it will be more challenging when comparing them to find a generalized foreground graph. The handling of the non-deterministic input is an enhancement to *ProvMark* which is described in the next chapter. In this chapter, we will concentrate on the basic design and architecture of *ProvMark* in the deterministic setting first.

5.3 ProvMark design

In this section, we will discuss the design and architecture of *ProvMark*.

5.3.1 The preliminaries

ProvMark is intended to automatically identify the sub-graph of a provenance graph that is recorded for the given target activities. Target activities could consist of individual system calls (like ***open*** or ***close***), sequences of system calls, or more general (concurrent) processes. In this chapter, the examples provided will only cover the simplest case of a single system call for easy illustration. But *ProvMark* has the ability and techniques to handle longer target action sequences of deterministic system calls. Handling concurrency and non-determinism is an enhancement of *ProvMark* which will be discussed in the next chapter.

As defined above, one of the required inputs for *ProvMark* is a benchmark program which contains `#ifdef TARGET` CPP directive to enclose the target system calls and action sequence for analysis. It compiles into two binaries, the background program contains only the background activities and the foreground program contains the target system calls on top of the background activities. But what are the background activities? They are activities performed in the kernel before and after starting an empty binary or a binary with some system call events. Indeed, in a Unix-like environment, before the execution of a binary, some preparation work, including memory mapping, is done. And at the end of the binary's execution, there is also some finalizing work to clear up the memory. These starting and ending processes such as (***fork*** or ***execve***), as well as other preparation activities like access to resources (program

files and libraries) or allocates memory location will create considerable "boilerplate" data provenance. These are the major contributions to the resulting background graph. The existence of this noise makes it necessary to distinguish the *foreground and background binaries* to help us filter the noise and understand which part in the resulting provenance is describing the target action sequences.

Furthermore, some target action sequences may depend on some prerequisite calls. For example, the ***read***, ***write*** or ***close*** system calls depend on a resources handle. We need to call the ***open*** system call to initialize a file or stream handle before executing these system calls. Without the corresponding handle, it does not make sense to execute them. If we are only interested in analysing the ***read***, ***write*** or ***close*** alone, we need to also filter out the ***open*** system call. In this case, the `#ifdef TARGET` CPP directive should not be covering the ***open*** system call and leave it as a background activity. As a whole, the above-mentioned process start, process end and all prerequisite calls form the background activity that we would like to elide and thus shows the need to use `#ifdef TARGET` CPP directives to distinguish the background and foreground binaries. As a result, the two binaries are almost identical; the difference between the resulting graphs should precisely capture the behaviour of the target system-call action sequence.

ProvMark includes a script for each system call that generates and compiles the appropriate C executables and prepares a staging directory with any needed setup for the execution. For example, the script will create and link a file in the stage directory before executing the ***unlink*** system call because a successful ***unlink*** system call requests a linked file. We constructed these scripts manually since different system calls require different settings. Code snippet 5.1 shows the script to prepare the stage and settings for the ***unlink*** system call and to generate the benchmark program with `#ifdef` CPP directive surrounding the target system call. We can see the script also compiles the binary and creates/links file which aim to prepare the environment for the ***unlink*** system call execution. *ProvMark* will pass in the directive and other compile options accordingly in different stages of the collecting period to make this script compile the benchmark program into the foreground and background binaries.

```
#!/bin/bash
if [ "$#" -le 1 ]
then
        echo "Usage: "$0" <Stage Path> [GCC_MACRO]"
        echo "Sample: "$0" ./stage -DTARGET --static"
        exit 1
fi
cd "$1"

#Clean directory
ls | grep -v 'prepare' | xargs rm -f

#Prepare Benchmark Program
CODE=$(cat <<EOF
#include <unistd.h>
int main() {int rn=0;
#ifdef TARGET
rn=unlink("`realpath $1`/link-test.txt");
#endif
return (rn==-1);}
EOF)
TMPFILE=$(mktemp -t tmp.XXXXXX.c)
echo "$CODE">$TMPFILE
gcc -o test ${*:2} $TMPFILE
rm "$TMPFILE"

touch test.txt
echo "TEST" > test.txt
link test.txt link-test.txt
chown 1000:1000 *
```

Code Snippet 5.1: Staging environment preparation script for ***unlink*** system call

5.3.2 The four subsystems

When comparing to the manual approach of the expressiveness benchmarking in chapter 3, we understand that there are multiple steps to handle in the automated approach. First, we need to start the chosen provenance collecting tools with certain settings, then we need to execute the background and foreground binaries. As mentioned above, we want to use a unified format to analyse the provenance graph, so we need to convert them from different provenance graph format to the Datalog format. After that, we also need to use generalization to filter out volatile

variables. Last but not least, we need to compare the generalized foreground and background graph to filter out the duplicate part and the remaining part in the foreground graph is the resulting provenance benchmark which will need to pass a check and fixing to include some dummy elements to make it a displayable graph. This is an abstraction of what the automated system should be doing.

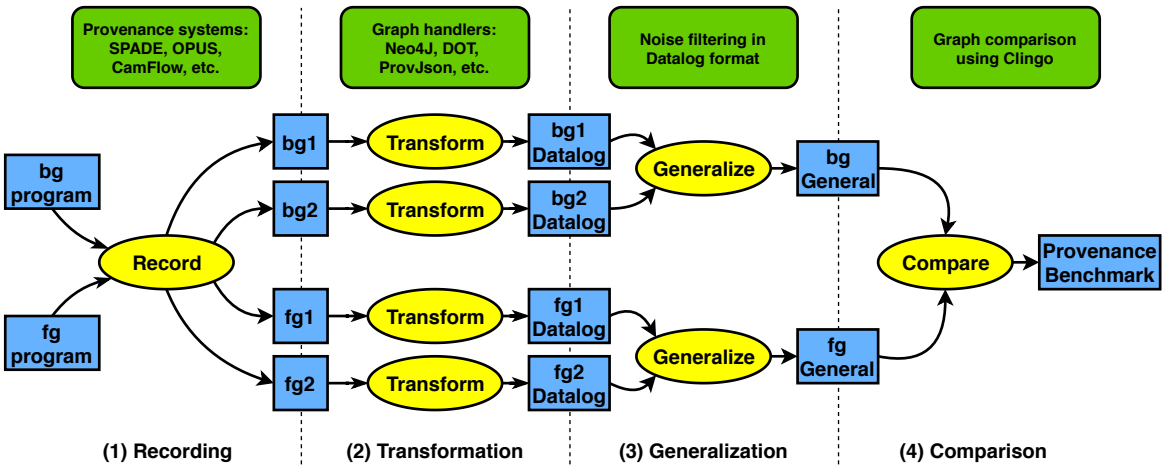


Figure 5.2: *ProvMark* system overview

ProvMark groups the above necessary steps into four ranges and is divided into four subsystems for handling different ranges of activities. The four subsystems are Recording Subsystem, Transformation Subsystem, Generalization Subsystem and Comparison Subsystem. The detailed design of the four subsystems will be described in this section. For a better understanding of the whole system design, there is a brief system diagram shown in Figure 5.2. The recording subsystem (1) uses one of the provenance recorders to compute multiple background graphs $bg_1, bg_2, ..., bg_n$ and multiple foreground graphs $fg_1, fg_2, ..., fg_n$. The transformation subsystem (2) maps these graphs to a unified *Datalog* format. The generalization subsystem (3) identifies the common structure in $bg_1, bg_2, ..., bg_n$, resulting in bg, and likewise $fg_1, fg_2, ..., fg_n$ are generalized to fg. Finally, bg and fg are compared in comparison subsystem (4); structure corresponding to bg in fg is removed, yielding the benchmark result.

In general, *ProvMark* allows a user to choose the target provenance collecting tools, the benchmark programs that are used for generating benchmarks and other minor configuration settings such as the number of trials and path for the output result. Most of the settings are done by either command-line arguments or by modifying the configuration file.

5.3.2.1 Recording Subsystem

The *recording subsystem* is the first subsystem in the execution life cycle of *ProvMark*. It interacts with the corresponding provenance recording tools to retrieve provenance results for a specific execution of the binaries. This subsystem first prepares a staging directory that provides a consistent environment for test execution. The recording subsystem then starts the provenance capture tool with appropriate settings, captures the provenance generated by the tool, and stops the tool afterwards.

The recording subsystem is the only one to interact directly with the provenance recording tools. It prepares the execution environments and passes the required inputs to the target provenance tools and retrieves the result from those tools for further processing. For each provenance tool, we implemented a script module that configures the tool to capture the provenance of specific processes (and child processes) relating to the execution of the target binary, rather than recording all contemporaneous system events. For each trial execution of *ProvMark*, only one of the modules corresponding to the chosen provenance recording tool will execute as the recording subsystem. The recording subsystem is used to record provenance graphs for the foreground and background variants of each target action sequence. Provenance can include transient data that varies across runs, such as timestamps, so we typically record multiple runs of each program and filter out the transient information in a later generalization subsystem, described below. Thus, the same script module may be executed multiple times with similar or different input to record those set of provenance graphs for generalization.

Recording multiple runs of the same set of processes using CamFlow was challenging in earlier versions because CamFlow only serialized nodes and edges once when first seen. The current version (0.4.5) provides a workaround to this problem that re-serializes the needed structures when they are referenced later. The original working logic of CamFlow is to retrieve a provenance graph from the starting of the operating systems to the shutting down of it, which covers the whole operating session. To avoid confusion, details of an artefact will only be recorded when it first appeared. All the later processes and executions related to this artefact may refer to the earlier record. If we just take a snapshot of the execution, we may miss the detailed information for the artefact as it may be recorded before the real execution of the target activities. As a result, we may lose track of this artefact. In the current version, the developers of CamFlow provide an option for repeat record of the artefact information when it is

used again to allow the snapshot provenance information complete and contains all the details of the needed artefacts. This eliminates the need to trace back the provenance information before execution for certain details and identifiers of the artefact itself. We also modified its configuration slightly to avoid tracking ProvMark's behaviour. We otherwise use the default configuration for each system; we refer to these configurations as the *baseline* configurations. As we discuss later, ProvMark can also be used to compare different configurations of the same tool.

This stage may sound straightforward: in fact, it has been the most delicate, especially for systems that work at the kernel level, because the needs of benchmarking differ significantly from the expected uses of these systems. We usually obtain repeatable results from SPADE by inserting time-outs to wait for successful completion of SPADE's graph generation process; even so, we sometimes stop too early and obtain inconsistent results leading to mismatched graphs. Similarly, using CamFlow, we sometimes experience small variations in the size or structure of the results for reasons we have not been able to determine. In both cases, we deal with this by running a larger number of trials and retaining the two smallest consistent results (as discussed later). For OPUS, any two runs are usually consistent, but starting OPUS and loading data into or out of Neo4j are time-consuming operations. During the implementation process, we discussed with the tool developers to check if our benchmark result fulfils what they are expecting as the output for their tools. Because of the need for generalization to filter out the noise and volatile information, we need to execute the same binaries multiple times, it does somehow affect the result provenance information as most of these provenance collecting tools are not designed to repeat the starting and stopping process frequently. In general, most of them aim to capture whole system provenance and analyse the execution for the whole section. But starting and stopping them and retrieving snapshots of provenance does cause some problems in the real implementation. In addition to the one we mentioned above on the non-repetitive information of artefacts in the case of CamFlow, we also face some cache flushing delay in SPADE. This affected the consistency of the result. After some additional discussion with the tool developers and multiple trials, we manage to find a suitable way to minimize the effect for the repeat execution, including adding a random delay in between each execution, or completing more trials and filtering out erroneous results with a high mismatch with other graphs in the same set.

The inputs of this subsystem are the target binaries needed for provenance collection. We do not need to pass on the choice for provenance collecting tools nor the

number of execution needed for each execution. The choice of tools and the multiple executions are controlled by the backbone process of *ProvMark* which managed the life cycle for the whole execution. The output of this subsystem is two sets of provenance graphs generated by the chosen provenance recording tools on the target background and foreground binaries that are freshly compiled just before passing to the subsystem. The two sets of provenance graph result are passed on to the transformation subsystem to handle.

5.3.2.2 Transformation Subsystem

This is the second subsystem of *ProvMark*. It takes in the two sets of provenance graphs generated from the recording subsystem (the set of foreground graphs and background graphs) and transforms them into a unified format. This subsystem also consists of multiple modules for handling different graph transformation processes.

Different provenance recording tools have different end usages of their provenance, thus they output their provenance graphs in different formats which is most suitable for their end usages. For example, SPADE supports Graphviz DOT format and Neo4j storage (among others) output, OPUS only supports Neo4j [121] storage but not the others at the point of writing, and CamFlow supports W3C PROV-JSON [83] as well as several other artefacts or streams processing back-end engines. CamFlow also can be used instead of Linux Audit as a reporter to SPADE, though we have not yet experimented with this configuration. There are several standard formats for provenance, but the lack of common formats used is one of our motivations for providing the expressiveness benchmarking in an automated system. This setting allows different tools to have a unified way to represent their data for benchmarking and further analysis or other real-life applications. Thus, this subsystem is one of the major parts of *ProvMark* that helps to transform different types of provenance graph format into a common format. Unlike the recording stage, this stage is straightforward. We describe the common format in detail below.

The common target format is a logical representation of property graphs, in which nodes and edges can have labels as well as associated properties (key-value dictionaries). Specifically, given a set Σ of node and edge labels, Γ of properties, and D of data values, we consider property graphs $G = (V, E, src, tgt, lab, prop)$ where V is a set of vertex identifiers and E a set of edge identifiers; these must be disjoint ($V \cap E = \emptyset$). Further, $src, tgt : E \to V$ maps each edge e to its source and target nodes, respectively, $lab : V \cup E \to \Sigma$ maps each node or edge x to its type $lab(x) \in \Sigma$, and

$prop : (V \cup E) \times \Gamma \to \Delta$ is a partial function such that for a given node or edge x, then $prop(x, p)$ (if defined) is the value for property $p \in \Gamma$. In practice, Σ, Γ and Δ are each sets of strings.

For provenance graphs, following the W3C PROV vocabulary, the node labels includes but are not limited to *entity*, *activity* and *agent*, edge labels are typically relations such as *wasGeneratedBy*, *used* and some less common relation like *wasDerivedBy*, and properties are either PROV-specific property names or domain-specific ones, and their values are typically strings. However, our representation does not assume the labels and properties are known in advance; it works with those produced by the tested system.

We represent property graphs as sets of logical facts using a Prolog-like syntax called *Datalog* [2], which is often used to represent relational data in logic programming (as well as databases [2] and networking [68]). The details of the Datalog format has been defined in chapter 4 where we consider the isomorphic graph comparison as a standalone problem to be solved. It can also be used in *ProvMark* as the generalization subsystem and comparison subsystem also requires isomorphic graph / sub-graph matching to get the result. We want to make use of the Answer Set Programming to help solve the automatic expressiveness benchmarking problem, thus we adopt Datalog as our unified format for provenance representation.

All the remaining stages, including the final result, work on the Datalog graph representation, so these stages are independent of the provenance recording tools and their output format. The Datalog representation can easily be visualized in different graph formats.

5.3.2.3 Generalization Subsystem

The third subsystem performs *graph generalization*. Recall that the recording stage produces several graphs for a given test program and results in two sets of similar graphs, the foreground graphs set and the background graphs set. We wish to identify a single, representative and general graph for each test program. We recall the notion of isomorphic graph matching problem mentioned in Section 2.4 below:

Let G and H be two attributed directed multigraphs.
$G = (V_G, E_G, src_G, tgt_G, lab_G, prop_G)$
$H = (V_H, E_H, src_H, tgt_H, lab_H, prop_H)$

If G and H is an isomorphic pair, represented by $G \cong H$,
then there is two bijection functions f and g where
$f : V_G \to V_H$ and $g : E_G \to E_H$ such that

- $\forall_{v \in V_G} lab_H(f(v)) = lab_G(v)$
- $\forall_{e \in E_G} lab_H(g(e)) = lab_G(e)$
- $\forall_{e \in E_G} src_H(g(e)) = f(src_G(e))$
- $\forall_{e \in E_G} tgt_H(g(e)) = f(tgt_G(e))$
- $\forall_{v \in V_G, k \in \Gamma} prop_H(f(v),k) \sqsubseteq prop_G(v,k)$
- $\forall_{e \in E_G, k \in \Gamma} prop_H(g(e),k) \sqsubseteq prop_G(e,k)$

Moreover, we say that G and H are *similar* if only the first four conditions hold, that is, if G and H have the same structure (vertexes and edges) but possibly different properties. This is the case for our two sets of graphs as they represent the same execution in multiple trials. They should differ only by those volatile variables which are recorded as property values. These volatile values are informative, but at the same time, this detail may be considered noise when we wish to identify the invariant provenance patterns for target actions. Also, they may differ in each execution because some of them are process-related and are not shared across multiple trial execution. Keeping these volatile data may affect the correctness of the benchmark generation because they are not unified across similar graphs. There is a possibility that with extra noise in the resulting benchmark, there may exist another similar action sequence which is a closer match to the wrong benchmark. This makes it hard to identify the existence of the correct pattern because they also contribute to the edit distance cost mentioned in chapter 4 and increases the difficulty to solve the isomorphic sub-graph matching problem. That is why we try to minimize the provenance recorded by ignoring them. The generalization process aims to filter them and leave behind the elements which are necessary for describing a runtime action sequence.

Before the generalization process, we need to handle and avoid some of the incomplete or incorrect graphs because of the uncertainty in the recording stage. We assume that over sufficiently many trials, there will be at least two *representative* ones, which are similar to each other after dropping all those incomplete or incorrect graphs. Identifying two representative runs is complicated by the fact that there might be multiple pairs of similar graphs. We discuss a strategy for identifying an appropriate pair below. To obtain two representative graphs, we first consider all the trial runs and par-

tition them into similarity classes. We first discard all graphs that are only similar to themselves and consider these to be failed runs because they do not match any other graph results. Among the remaining similarity classes, we first choose the similarity class with the most number of graphs. We assume that most of the complete and correct provenance collecting process should result in a similar result, and the incomplete and incorrect results are generally leading to different graphs. Thus the similarity class with the largest number of graphs should contain the best candidates for post-processing. After that, we choose a pair of graphs whose size are the smallest within the chosen similarity class. The main reason for choosing the pair of graphs with the smallest size is because there is a large set of noise and unrelated components exist in the graphs and we assume that the smallest pair of graphs contains the cleanest result among the others in the same similarity class. Thus it provides a better source of graphs for the post-processing of ProvMark.

Empirically, we observed that the probability of failure for graphs resulting from the recording stage could be as high as 50-65%. There are multiple reasons for the high failure. One of the reason is believed to be related to the uncontrollable flushing of the cache. One of the main pitfalls is that we can not control when the cache flushing is happening, this affects the completeness of the result. If we stop the provenance tools before the flush has been done, then those data that are not been flushed will be lost and result in incomplete graphs. From our manual approach mentioned back in Chapter 3, the success rate is much higher when we wait until the cache has been flushed before further processing the provenance result. This may not be an efficient way for the automated system because the flushing happens randomly. We choose to restart the tools and filter out those failure graphs to have higher efficiency. Also, as CamFlow is retrieving information through LSM hook and this information is passed to the user level through the *relayfs* components, there may exist a certain level of delay. As CamFlow originally aims to record whole system provenance for the full operating session, it keeps running in the kernel. We can only control a daemon in the user level to process the provenance received. Some data may be delayed and send to the CamFlow daemon after we turn off the recorder when we finish a trial execution. This delay may cause missing of information during the snapshot period and result in incomplete provenance graph which we considered as failure. Besides, some provenance source used by those provenance systems, including audit reporter are not too reliable on the integrity of data. There is some loss experienced when there is a large number of events happening in the kernel at a similar time which will overflow

the buffer queue and cause some of the records dropped. To decrease the overhead of the logging process, the Linux Audit System allows configuration using *audit rules*. From the original Linux Audit Daemon configuration manual [69], it has mentioned a *max_log_file* configuration which allows the system administrator to limit the size of the log file and also use the *max_log_file_action* to configure how will the Linux Audit Daemon handle the additional log when the log file is full. The key problem here is how the administrator handles situations in which storage space or communication buffer space is exhausted. Although the default action of *max_log_file_action* is to keep the additional log in buffer, it is possible to configure the Daemon to drop oldest log record entries or simply ignore new audit log record. Also, the buffer may be full and fail to store additional log records. These possible settings may result in loss of log records and make the Linux Audit Daemon an unreliable (and incomplete) source for system and kernel activities. Morrison [114] provides an analysis of the whole Linux Audit System and summarizes some standards and mechanisms for the prevention of audit data loss and the protection of the integrity of those data. Although these methods do help to reduce the loss rate by different configuration enforcement including the maximum buffer size settings, some of them still require certain tradeoffs. These tradeoffs include immediate halting of system applications or denial of later activities. Thus a certain level of integrity detection is still required to detect random loss or altered audit log records. Although some provenance systems like SPADE do have repairing mechanisms, there are still some data which cannot be recovered due to random loss. These different factors affect the error rate in the provenance graph generation process.

As we are aiming to retrieve some similar graphs to find the pair of graphs which are closest to each other, we need to provide more candidates, the better balance is to provide double of the graphs needed. When we consider the failing percentage of around 50-65%, the success case is limited to one-third of chance. Different provenance systems depend on different provenance sources, thus the error rate is different for each of the tools. We have done some simple experiments to summarize the failing percentage and how many graphs are adequate to retrieve enough graphs in good shape. Each of the provenance systems is managed to collect provenance for a sample benchmark program with only one system call **`rename`**. The collection is repeated for a different number of trials ranging from one trial to twelve trials. The experiments are repeated for 10 times and an average number of accepted graphs are calculated from the same number of trials for the same provenance systems. The result of the experiments is shown in Figure 5.3. From these results, it shows the error rate in CamFlow is

quite high due to the uncertain delay and halts on ProvMark, while SPADE also results in some level of errors because of the random loss occurring in the queue buffer of the audit reporter. To ensure we could get at least 4 graph candidates, it is necessary to multiply the number by 3. So we calculated, observed and tested that 11-12 trials are sufficed to raise the probability of finding two representative graphs above 99%. This is the worst-case scenario. Usually, fewer trials suffice. This observation is done by running multiple trials of testing on the three provenance collecting tools we have chosen for our testing, which is SPADE, OPUS and CamFlow. The handling of non-determinism with more combination of graphs in the two sets are discussed in the next chapter.

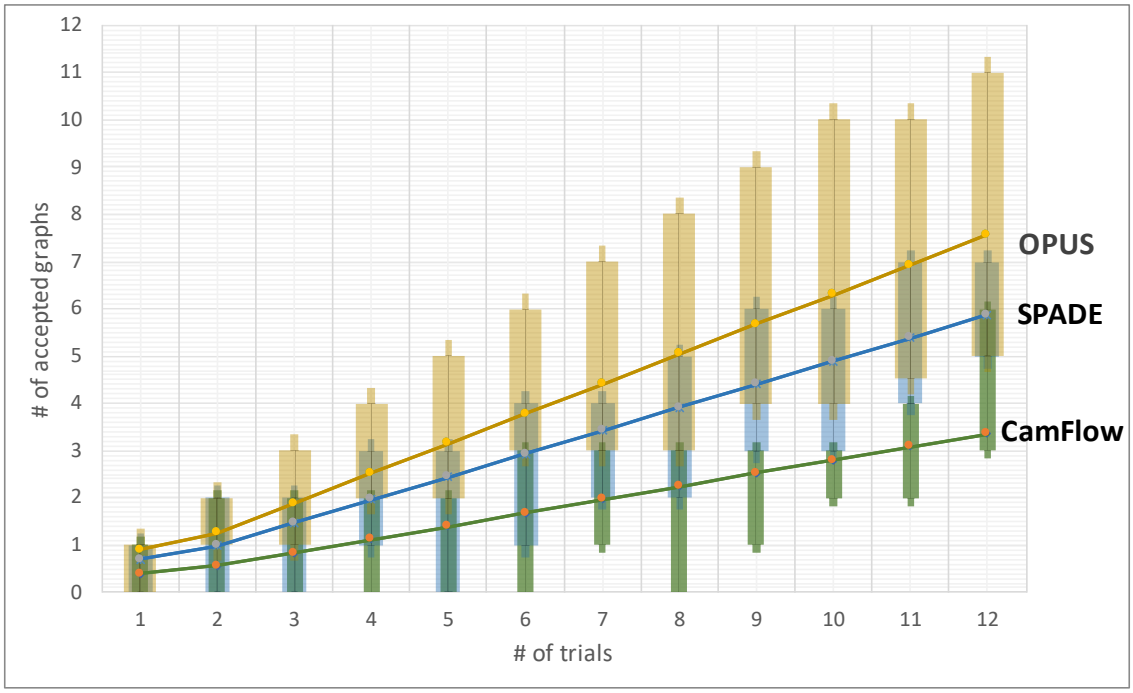


Figure 5.3: Number of accepted graphs for the three provenance systems

Given two similar graphs, the generalization subsystem identifies the property values that are consistent across the two graphs and removes the transient ones. It searches for a matching list of nodes and edges from the two graphs that minimize the number of different properties. We assume that the differences are transient data and discard them. We would like to minimize the number of differences, that is, match as many properties as possible to avoid accidentally removing important properties.

Similarity and generalization are instances of the *graph isomorphism* problem, whose status (in P or NP-complete) is unknown as mentioned in Arvind et al. [13]. We have discussed this problem and how we go around it in a separate chapter 4. We

solve these problems using an Answer Set Programming (ASP) [61] specification that defines the desired matching between the two graphs. The problem specification is a logic program defining when a binary relation forms an isomorphism between the graphs.

As mentioned in chapter 4, the similarity and generalization problem can be solved using `clingo`, an efficient ASP solver [61]. We use the resulting matching to determine which properties are common across both graphs and discard the others. We perform generalization independently on the foreground and background graphs. Thus the inputs to this subsystem are the sets of foreground graphs and background graphs. The final output is the two generalized graphs representing the invariant activity of the foreground and background programs respectively. It is also worth mentioning that as this stage is already independent of what provenance collecting tools have been used, there is only one module for this subsystem to handle the whole generalization process. It is also possible to provide an additional module for this subsystem if we later discover another better way to solve the generalization and isomorphic graph matching problem.

5.3.2.4 Comparison Subsystem

The fourth and last subsystem is *graph comparison*. Its purpose is to match the background graph to a sub-graph of the foreground graph; the unmatched part of the foreground graph corresponds to the target activity which is output as the provenance benchmark.

In our definition, we defined our foreground and background binaries are compiled from the same C program with a simple `#ifdef` CPP directives enclosing the target action sequence. The foreground binary is compiled with the directives defined while the background binaries are compiled without the directives defined. As a result, the background binary performs a subset of the behaviour of the foreground binary. The remaining action in the foreground binary representing the behavioural pattern of the target action sequence. It is worth mentioning that the more actions performed by the two programs, the larger the resulting graphs will be and the more expensive it will be to compare the graphs. Thus, it is always better to narrow down the testing program to a minimum, that is only doing the necessary actions in the background graph and add in the target action on top of it to form the foreground graph. Besides, programs that exhibit non-determinism or I/O activities might not yield the same provenance graph shape on repeated trials, which would invalidate our earlier assumption that the test

program executions are deterministic. The handling of non-determinism is indeed an enhancement for *ProvMark* and is separately discussed in the next chapter.

When we are compiling the two binaries from the same program with or without the CPP directives, we are expecting the foreground binary only contains additional actions enclosed by the directives when comparing to the background binary which represents the target action to be analysed. Thus, the generalized provenance graph for the background binary is a sub-graph of the generalized provenance graph for the foreground binary. As a result, there should be a one-to-one matching from the nodes and edges in the background graph to the foreground graph. This graph matching problem is a variant of the isomorphic sub-graph matching problem mentioned in chapter 4. We again solve these problems by ASP. The details of the problems are in chapter 4.

Given two graphs, G_1 and G_2, the solver finds possible matches between nodes and edges of G_1 and a sub-graph of G_2 and choose the matches with minimum mismatch for properties. It is similar to the general isomorphic graph matching problem, the only difference is the matching target are not the pair of graphs themselves. Instead, the matching is done between one graph and the sub-graph of the other graph. This makes the isomorphic sub-graph matching problem more specific than the general problem. And when we comparing the problem to the real scenario of our benchmark process, we can recognize their similarity. We are solving the same problem when we aim to match the background graph to part of the foreground graph and retrieve the unmatched part from the foreground graph as a result.

This comparison subsystem is similar to the last generalization subsystem, where the comparison subsystem is aiming to solve a more specific problem. They are both independent of what provenance collecting tools have been used. Thus only one module is developed for this subsystem and it still could be extendible similar to the generalization subsystem. This subsystem will take the generalized foreground graph and generalized background graph and compare them to generate the provenance benchmark for the target action sequence and the chosen provenance collecting tools. The result is passed back to the backbone process and it will decide on how to post-process the result and return to the user. The returned provenance benchmark should remain in the Datalog format.

5.3.3 Modular design

The automated system *ProvMark* automates the expressiveness benchmarking process mentioned in chapter 3. The target provenance collecting tools are all existing tools in the field. As mentioned in Section 2.1, there are a large group of tools in the field collecting provenance information in different levels via different components of the operating system. They also generate provenance in different formats which are most suitable for the usage of their target user group. So, to allow easy extension of *ProvMark* to handle additional tools and provenance format, different components and subsystems of *ProvMark* are designed as modules. For example, the recording subsystem has several modules to handle different kinds of provenance recording tools. Although each of them has customized handling for the tools, they follow the same interface patterns for input and output. To support new tools, the tool developer just needs to implement the provided interface and add in some starting logic for their tools. It works as an adaptor to connect the recording subsystem to the provenance collecting tools. Users of *ProvMark* only need to specify the tools they choose and the recording subsystem will automatically find the needed adaptor module to connect to the related provenance recording tool. This also applies to the handling of different provenance result formats in the transformation subsystem. This setting demonstrates how *ProvMark* makes use of the modular design to achieve *extensibility*. More details will be discussed in the evaluation section of this chapter.

As each of the components and subsystems is working as modules, it is necessary to have a backbone control for *ProvMark*. The backbone controls the overall execution of *ProvMark*, including but not limited to the reading of the user configuration and input, managing the intermediate result and output the final provenance benchmark result. The backbone control for *ProvMark* is the centrepiece that arranges and handles the full execution life cycle. It will follow the basic runtime as mentioned in Algorithm 1.

ProvMark backbone receives the choice for benchmark program, provenance collecting tools and other user settings from the configuration file and command-line input. It will interpret them and arrange for the correct modules for each subsystem. It is also responsible for relaying intermediate results between subsystems or modules and provides some final process after the last subsystem has completed. At last, it will return the provenance benchmark in the format chosen by the user. The whole backbone process act as a central control for the execution of *ProvMark* and support the modularized design that allows *ProvMark* to be easily extended to more tools and

Algorithm 1: ProvMark life cycle

Input : Benchmark Program

Input : Choice of Provenance Collecting Tool

Input : User Settings from Configuration File

```
Read configuration and arrange modules
for i ← 0 to NumberofTrials do
    Run Recording Module
end
for i ← 0 to NumberofGraphs do
    Run Transformation Module
end
Filter failure graphs in the background and foreground graphs
 set
Run Generalization Module
Run Comparison Module
Final process of Provenance Benchmark
```

Result: Provenance Benchmark in user-chosen format

provenance formats.

5.3.4 Usage and further applications

This subsection describes some further applications of the provenance benchmark result generated by *ProvMark*. Note that the applications we mentioned here are possible extensions of *ProvMark* which are not implemented and can be either a future working direction of *ProvMark* or possible extensions to the application for the current benchmark by other research area or tools for the designated purpose.

As mentioned earlier, the different designs of provenance recording tools affect whether (and how) they can observe certain activities. We consider a representative example, calling ***rename*** on a file name that does not exist. By default, SPADE installs Linux Audit rules that only report on successful system calls, so SPADE records no information in this case. OPUS monitors system calls via intercepting C library calls, so it knows whether a call is being attempted, and typically generates some graph structure even for unsuccessful calls. For example, the result of a failed ***rename*** call has the same structure as the successful one, except that the return value property is

-1 instead of 0. Finally, CamFlow can in principle monitor failed system calls, particularly involving permission checks, but does not do so in the case of ***rename*** failing due to a missing file.

We do not take a position regarding whether it is desirable to record provenance for failed calls: a failed call is part of the history of the system, but it may not result in new dependency information that belongs in the provenance graph. The ability to record failed system calls also demonstrate how different *provenance collecting tools* choose to interpret what information should be collected regarding system action sequences. Comparison of provenance benchmark results can help to distinguish this difference.

5.3.4.1 Configuration validation

All of the systems we considered have several configuration parameters, to allow greater or lesser levels of detail, coalescing similar operations (such as repeated reads or writes), or enabling/disabling versioning which distinguishes the modified artefacts as separated nodes. Each of these configuration settings may have (potentially surprising) effects on what provenance is recorded.

We have done some preliminary experimentation with benchmarking alternative configurations of SPADE. SPADE provides a flag `simplify` that is enabled by default. Disabling `simplify` causes ***setresgid*** and ***setresuid*** (among others) to be explicitly monitored. However, enabling this flag also uncovered a minor bug: when `simplify` is disabled, one of the properties of a background edge is initialized to a random value, which shows up in the benchmark as a disconnected sub-graph. This has been fixed in the current version.

SPADE also provides various *filters* and *transformers* which perform pre-processing or post-processing of the stored provenance respectively. We experimented with one of SPADE's filters, `IORuns`, which controls whether runs of similar read or write operations are coalesced into a single edge. We found that in the benchmarked version of SPADE, enabling this filter had no effect. This turned out to be due to an inconsistency in the property names used by the filter and those generated by SPADE. This has also now been fixed by the SPADE developers.

It is worth mentioning that this usage of configuration validation provides wider options to provenance system developers for validating their tools. Currently, the tool developers can use *ProvMark* to generate test case results for some features of their tools and determine if the provenance benchmark result fulfils their expectations. This usage was considered useful for some of the tool developers because it does help them

to identify processing problems in their tools. The configuration validation mentioned in here not only considers the general execution of the tools but also considers configuration and specific execution criteria which covers more parts of the tools and thus provide a stronger validation and wider options for the tool developers.

5.3.4.2 Regression testing

Although *ProvMark*'s major goal has been to make the system call-level behaviour of provenance recording tools more transparent to their users, it can also act as an ingredient in regression testing for the tool developers. Specifically, we can retain the benchmark graphs (as Datalog) from previous runs, and when a change is made to the system, the tool developers can rerun all of the benchmarks and compare the previous results to the new ones. The results should be isomorphic; if not, the differences can be visualized to help understand how the behaviour has changed and possibly provide a source of information for bug finding in the new implementation. This is another usage on top of the tools and configuration validation.

5.3.4.3 Sensitive activity detection

A more ambitious application which is left for future work is activity detection, which is one of the motivations for this project. If we have an example of a kind of activity we would like to detect, e.g. a C source code or shell script snippet that illustrates how to perform an attack, we could use *ProvMark* to record the attack-specific activity, and then use the extracted sub-graph to identify patterns that are distinctive to that activity. However, there are several obstacles to making this practical. A realistic attack (for example against a web server) might correspond to hundreds or thousands of provenance graph nodes, and it is not yet clear whether our approach scales to sufficiently large graphs. So far we have considered benchmarking deterministic activity only, whereas realistic attacks may involve concurrency. Finally, even if we successfully identify the sub-graph corresponding to an attack, it might not match future similar attacks exactly. Thus, additional work may need to be done on extracting the key features from the attack graph(s). The future development of *ProvMark* is aiming to work towards this direction on identifying large and non-deterministic patterns for future identification of similar activities. Part of the work on the enhancement list like handling of non-determinism will be discussed in a later chapter. Other challenges on top of that are considered as future work and are beyond the scope of this project.

5.4 ProvMark result analysis

This section discusses the testing result of *ProvMark* and some preliminary analysis and comparison of the tools and system call candidates.

5.4.1 Tools and system call candidates

For the manual approach of the expressiveness benchmarking mentioned in chapter 3, we choose two of the provenance collecting tools OPUS and SPADE for testing because they create smaller provenance graphs which allow us to analyse them manually. We only choose two of them because it is not possible to do a large amount of analysis and adding another tool will be very inefficient. When moving to the automated approach, we aim to keep the original choice for the manual approach which we are already familiar with while adding in one more candidate which collects provenance using a different methodology to widen our experiment and allow us to access broader result sets for the comparison, analysis and the evaluation of the whole *ProvMark* system. This choice decreases the time needed to study the usage of a brand new set of tools but still allow us to gather our test candidates working in different methodology and applications. To verify and evaluate the performance of ProvMark, we do need some minor help by the original provenance tools' developers. We have successfully connected with the developers of SPADE, OPUS and CamFlow also support our choice for using these three candidates as our preliminary testing target to evaluate and analyse the feasibility and application of ProvMark.

One of the similar characteristics of SPADE and OPUS is their source of information. They rely on some existing system components to relay information back to them. SPADE adopts the audit reporter (or some other modules like *Strace* or log record) which captures the information in kernel level and passes it to components that have registered a receiver. The information can also be filtered manually before sending to the recipient under certain configurations. On the other hand, the initial version of OPUS intercepts dynamic library calls to receive notification of actions happening in real-time for provenance recording. Both of them rely on some user-level components, thus they may miss some information that is not handled by the underlying components. Although both of them rely on some existing system components, they source information from different levels. SPADE retrieves information in the user level which is relayed from the kernel while OPUS retrieves information by intercepting library calls which are performed in the user level.

We have chosen CamFlow as our additional testing candidate. It retrieves provenance information in a different location by a different methodology. It builds up a customized LSM hook that registers with the LSM directly and monitors every decision of the LSM and passes it back to the user level by a self-defined relaying components. As it is adding functionality in the kernel, CamFlow requires a recompilation of its custom kernel to include the functionality in the base level around the LSM. Thus running CamFlow has two prerequisites, the first one is root privilege and the second one is that capturing must be done in the operating system based on the customized kernel. On the other hand, it provides a trace for everything happening in the system since system booting. This is because every system call activity needs to pass LSM for authorization. The provenance trace collected by the CamFlow LSM hook includes all the information explaining everything happening in a process or the system itself that requires LSM authorization or monitoring. The three candidates have different provenance collecting methodology and source of information. Also, they have different use case scenarios in mind and thus they have done some customization, filtering and transforming of information before generating the provenance. Their significant differences make them a good candidate for the automated *ProvMark* to evaluate its ability for expressiveness benchmarking and also helps to solve problems and enhance the tools for those provenance collecting tools developers. The three different approaches cover most of the range of working for provenance collecting tools, thus we choose them as our testing candidates to cover more possible cases.

In addition to provenance recording tools, we also need to choose the target system calls for testing. The chosen target system calls are shown in Table 5.1, similar to the choice mentioned back in Chapter 3. The chosen group is commonly used for security and auditing usage and most of them are believed to be related to sensitive action sequences [90, 118]. Note that the same system call may display different behaviour using different parameters and system states; we focus on common-case behaviour here and discuss failure cases and other call contexts later. Writing benchmark programs is currently a manual (but not difficult) process as mentioned above. The following is a comparison of the system calls in the four groups, file (artefact) management, process management, permission management and pipe/stream management.

Last but not least, there is also an additional group of system calls which are related to memory, sockets, multiple threads and processes which are all non-deterministic events. These will be discussed further in the next chapter when we enhance *ProvMark* to handle non-determinism.

5.4.2 Analysis of results

This subsection discusses some of the similarities and differences in the resulting provenance generated by the three provenance recording tools candidates. The results are grouped according to the grouping shown in Table 5.1, which is similar to the grouping we used in the manual approach. In theory, ProvMark should be able to identify the benchmark for any system calls in the provenance graphs. But we limit our evaluation test cases to the system calls shown in Table 5.1 because these are the system calls handled by the three provenance tools. Other system calls are either ignored or filtered by one or more of the three provenance tool candidates and those system calls are not shown in their resulting provenance graphs. The empty result makes the testing and evaluation useless and thus we ignore those system-calls in our testing and evaluation. In this evaluation, we are focusing on the effect of each system calls and are not concentrating on the communication and data exchange between multiple system-calls in order to have a better understanding on how different provenance tools on handling a single system call event.

Group 1				Group 2	Group 3		Group 4
creat	dup	dup2	dup3	clone	chmod	fchmod	pipe
link	linkat	symlink	symlinkat	execve	fchmodat	chown	pipe2
mknod	mknodat	open	openat	exit	fchown	fchownat	tee
read	pread	rename	renameat	fork	setgid	setregid	
truncate	ftruncate	unlink	unlinkat	vfork	setresgid	setuid	
write	pwrite	close		kill	setreuid	setresuid	

Table 5.1: System calls used for the ProvMark evaluation

5.4.2.1 Artefact management

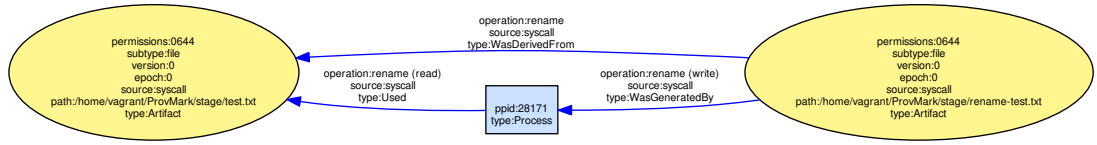
Group 1 includes system calls which manage artefacts and files. Examples such as ***creat***, ***open***, ***close*** are generally covered well by all of the tools, but they each take somewhat different approaches to represent even the simplest file open behaviour. For example, for the ***open*** call, SPADE result contains a single node and edge, while CamFlow creates a node for the file object, a node for its path, and several edges linking them to each other and the opening process, and OPUS creates four new nodes including two nodes corresponding to the file. On the other hand, reads and writes ap-

pear similar in both SPADE and CamFlow, while by default, OPUS does not record file reads or writes. For **`close`**, CamFlow records the underlying kernel data structures eventually being freed, but *ProvMark* does not reliably observe this.

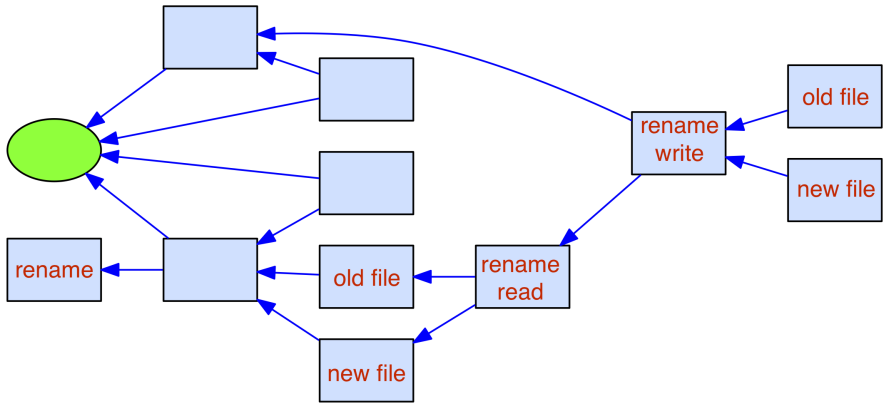
The **`dup`**, **`dup2`** and **`dup3`** system calls duplicate a file descriptor so that the process can use multiple descriptors to access the same open artefact. SPADE and CamFlow do not record this call directly, but the changes to the process's artefact descriptor state can affect future file system calls that are monitored. OPUS does record **`dup`**, **`dup2`** and **`dup3`** actions specifically. The two added nodes are not directly connected but are connected to the same process node in the underlying graph. That means that in the viewpoint of OPUS, the original artefact descriptor and the copied artefact descriptor is treated as two completely different recording targets. Thus, one component records the system call itself (status of the original artefact descriptor) and the other one records the creation of a new resource (the duplicated artefact descriptor). This is also similar to the case of the **`mknod`** system call family which has similar usage to the **`dup`** system call family. The **`mknod`** system call is only recorded by OPUS, and **`mknodat`** system call is not recorded by any of the systems.

Lastly, the **`rename`** and **`renameat`** system call illustrates how the three systems record different graph structure for this operation. SPADE represents a rename using two nodes for the new and old file names, with edges linking them to each other and to the process that performed the rename. OPUS creates around a dozen nodes, including nodes corresponding to the rename call itself and the new and old file name. CamFlow represents a rename as adding a new path associated with the file object (treat the rename action as a version update of the file itself); the old path does not appear.

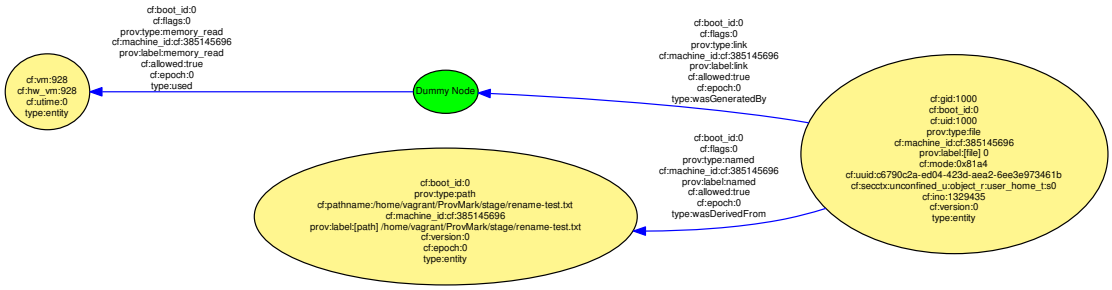
An example provenance benchmark results with **`rename`** system call are shown in Figure 5.4. The property labels of the OPUS result has been removed because of the readability problem.



(a) SPADE



(b) OPUS (Abstract)



(c) CamFlow

Figure 5.4: Benchmark for ***rename*** by SPADE/OPUS/CamFlow

5.4.2.2 Process management

Group 2 includes process management system calls. The process management operations start processes (***fork***, ***vfork***, ***clone***), replace a process's memory space with a new executable and execute it (***execve***), or terminate a process (***kill***, ***exit***). These system calls may result in multi-processing or multiple-threading execution which is non-determinism. We are only focusing on the deterministic part of process management in this chapter. The target action sequences in the chapter include the testing of the creation and termination of process or process management within a single process. Other action sequences out of this range may result in non-deterministic

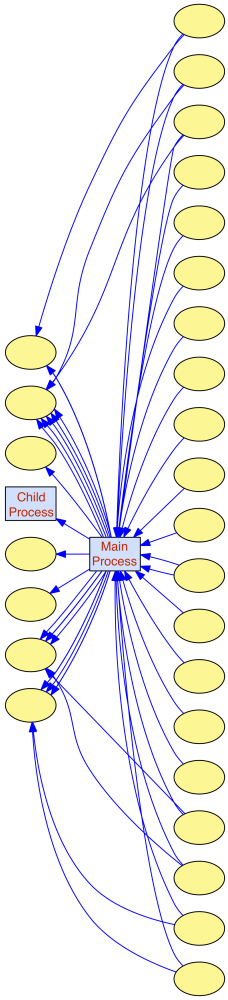
events which are handled after the enhancing of *ProvMark*. We will discuss them in the next chapter.

Process creation and initialization calls are generally monitored and captured by all three tools, except that OPUS does not appear to monitor the ***clone*** system call. Inspecting the results manually shows significant differences, however. For example, the SPADE benchmark graph for ***execve*** is large but has just a few nodes for OPUS and CamFlow. On the other hand, the ***fork*** and ***vfork*** graphs are small for SPADE and CamFlow and large for OPUS. This may indicate the different perspectives of these tools: SPADE relies on the activity reported by Linux Audit, while OPUS relies on intercepting C library calls outside the kernel.

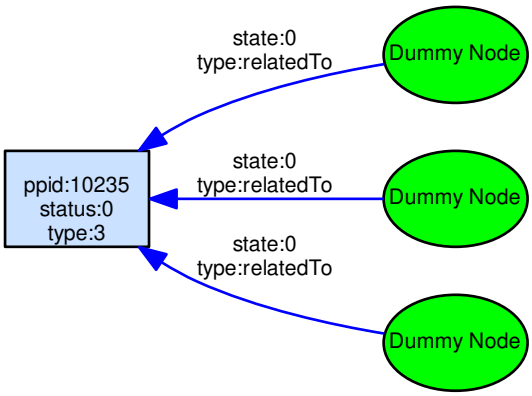
Another interesting observation is that the SPADE benchmark results for ***fork*** and ***vfork*** differ. Specifically, for ***vfork***, SPADE represents the forked process as a disconnected activity node, i.e. there is no graph structure connecting the parent and child process. This is because Linux Audit reports system calls on exit, while the parent process that called ***vfork*** is suspended until the child process exits. SPADE sees the ***vfork*** child process in the log executing its system calls before it sees the ***vfork*** that created the child process. This may relate to the order handling of the audit manager in the kernel which only records an event when the process finishes. If process A starts earlier than process B and ends after the termination of process B, then the audit reporter will give a smaller event id to process B and relay process B information before process A. Thus, it may affect the ordering. But it may be fine-tuned by ordering by timestamps of process start.

The ***exit*** and ***kill*** calls are not detected because they deviate from the assumptions our approach is based on. A process always has an implicit ***exit*** at the end, while killing a process means that it does not terminate normally. Thus, the ***exit*** and ***kill*** benchmarks are all empty.

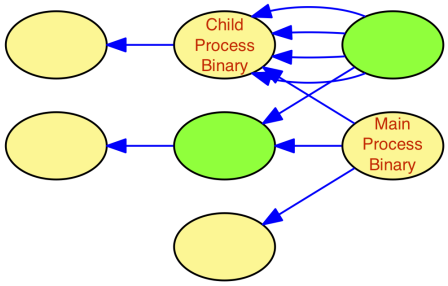
Figure 5.5 shows provenance benchmark results for ***execve*** system call.



(a) SPADE (Abstract)



(b) OPUS



(c) CamFlow (Abstract)

Figure 5.5: Benchmark for ***execve*** by SPADE/OPUS/CamFlow

5.4.2.3 Permission management

Group 3 includes permission related system calls. These system calls aim to change artefact or process owners or permissions (***chown***, ***fchown***, ***fchownat***, ***chmod***, ***fchmod***, ***fchmodat***). According to its documentation, SPADE currently records the ***chmod*** family but not ***chown*** family. OPUS does not monitor ***fchmod*** or ***fchown*** because from its perspective these calls only perform read/write activity and do not affect the process's file descriptor state, so as for other read/write activity OPUS does not record anything by default. CamFlow records all of these operations.

Apart from general owners or permissions management, there are also a set of system calls managing the (effective) identity of an user executing a binary (***setuid***,

setreuid, ***setresuid***, ***setgid***, ***setregid***, ***setresgid***). In its default configuration, SPADE does not explicitly record ***setresuid*** or ***setresgid***, but it does monitor changes to these process attributes and records observed changes. Our benchmark result for ***setresuid*** is non-empty, reflecting an actual change of effective user id, while our benchmark for ***setresgid*** just sets the effective group id attribute to its current value, and so this activity is not noticed by SPADE. OPUS also does not track these two calls, while CamFlow again tracks all of them.

Figure 5.6 shows provenance benchmark results for ***setreuid*** system call.

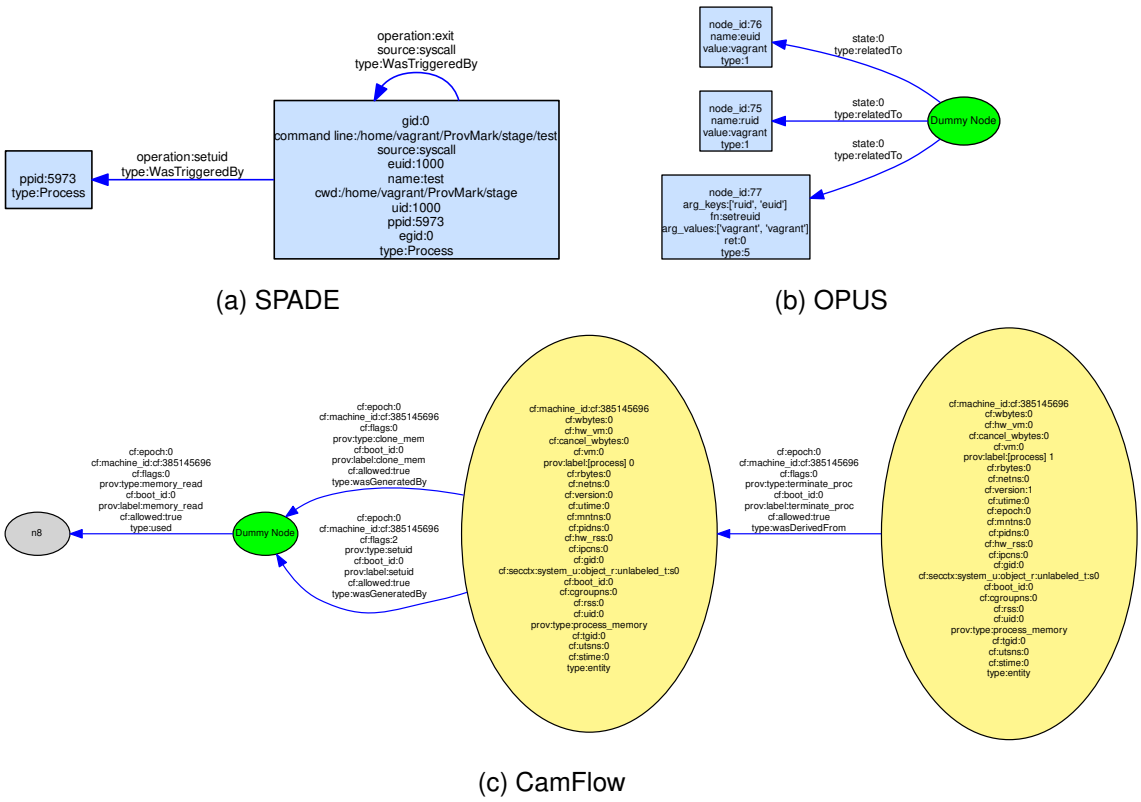


Figure 5.6: Benchmark for ***setreuid*** by SPADE/OPUS/CamFlow

These permission management system calls are important for accountability because they keep track of the permission changes of artefacts and processes and also the changes of privilege level of a user in runtime. The resulting provenance of these system calls shows the detailed process of privilege escalation of a user for accessing a specific resource. These details help to identify the processes or users who are responsible for certain actions. This may help to fix problems in those processes which accidentally give the user the chance to escalate privileges. As a result, the benchmark in these system calls can identify if the candidate tools are capable of this security and auditing usage.

5.4.2.4 Pipe Management

Last but not least, group 4 includes system call management piping communication. Similar to the system calls managing processes, piping system calls are related to interprocess communication which may also involve non-determinism. We are still focusing on the deterministic part which involves pipe creation and passing information through two processes. The remaining action sequences are also left to discuss later in the next chapter.

An example provenance benchmark result with ***pipe*** system call recorded by OPUS is shown in Figure 5.7. An additional example provenance result with ***tee*** system call recorded by CamFlow is shown in Figure 5.8.

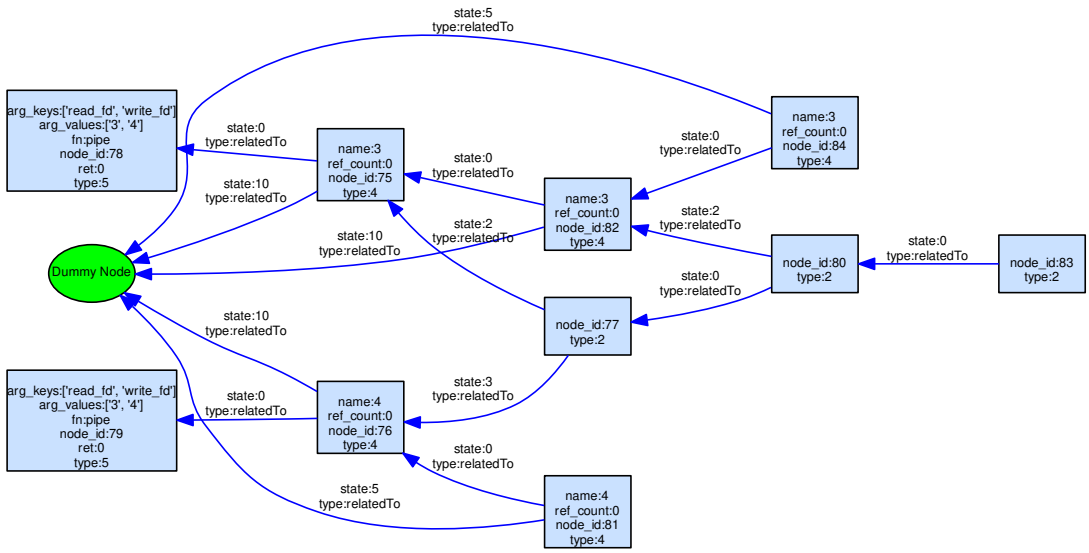


Figure 5.7: Benchmark for ***pipe*** by OPUS

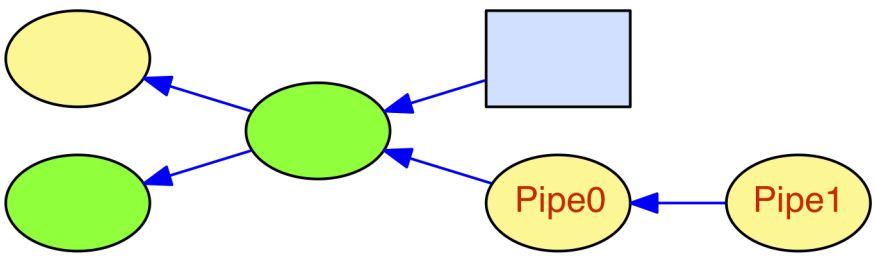


Figure 5.8: Benchmark for ***tee*** by CamFlow (Abstract)

Pipes provide efficient interprocess communication. The system calls (***pipe*** and ***pipe2***) creates and opens a pipe in different ways, which can be read or written using standard file system calls, and ***tee*** duplicates information from one pipe into another without consuming it. That is, it passes output from one pipe as input to another pipe. Of the three systems, only OPUS records the ***pipe*** family (***pipe*** and ***pipe2***), while only CamFlow currently records ***tee***.

5.4.2.5 Summary

This subsection discusses comparisons of the benchmark results generated by *ProvMark* with the chosen system calls and provenance collecting tools candidates. It shows the differences and similarities in the resulting benchmarks of different tools on the same system call actions. It demonstrates large differences in some of the cases which once again show the necessity of building up the automated approach for the expressiveness benchmarking for helping tools developers and end-user to have a standardized and unified way to analyse the capabilities of different tools in different circumstances. Table 5.9 summarize the result status of the provenance generation process of ProvMark, working with each of the three provenance systems on all system calls mentioned in Table 5.1. Table 5.9 gives an overview of whether a system call is explicitly handled by a provenance system and representing in the resulting provenance graph and benchmark. It also gives an overview of the expressiveness of the provenance benchmark, which is verified by the original developers of the provenance systems. It is worth mentioning that the summary given in Table 5.9 is recorded by using specific version of the provenance systems mentioned in Table 2.1, back in Section 2.1.3.

System Call	SPADE	OPUS	CamFlow
close	OK	OK	LP
creat	OK	OK	OK
dup	SC	OK	NR
dup2	SC	OK	NR
dup3	SC	OK	NR
link	OK	OK	OK
linkat	OK	OK	OK
symlink	OK	OK	NR
symlinkat	OK	OK	NR
mknod	NR	OK	NR
mknodat	NR	NR	NR
open	OK	OK	OK
openat	OK	OK	OK
read	OK	NR	OK
pread	OK	NR	OK
rename	OK	OK	OK
renameat	OK	OK	OK
truncate	OK	OK	OK
ftruncate	OK	OK	OK
unlink	OK	OK	OK
unlinkat	OK	OK	OK
write	OK	NR	OK
pwrite	OK	NR	OK

Group 1

Note	Meaning
NR	Behavior not recorded (by default configuration)
SC	Only state changes monitored
LP	Limitation in ProvMark
OK	Validated by developer

Key definition

System Call	SPADE	OPUS	CamFlow
clone	OK	NR	OK
execve	OK	OK	OK
exit	LP	LP	LP
fork	OK	OK	OK
kill	LP	LP	LP
vfork	OK	OK	OK

Group 2

System Call	SPADE	OPUS	CamFlow
chmod	OK	OK	OK
fchmod	OK	NR	OK
fchmodat	OK	OK	OK
chown	NR	OK	OK
fchown	NR	NR	OK
fchownat	NR	OK	OK
setgid	OK	OK	OK
setregid	OK	OK	OK
setresgid	SC	NR	OK
setuid	OK	OK	OK
setreuid	OK	OK	OK
setresuid	OK	NR	OK

Group 3

System Call	SPADE	OPUS	CamFlow
pipe	NR	OK	NR
pipe2	NR	OK	NR
tee	NR	NR	OK

Group 4

Figure 5.9: Summary of validation results

5.5 ProvMark evaluation

This section evaluates the basic functionality of *ProvMark*. We first consider the actual overhead of the *ProvMark* components and subsystems, then we have provided a simple self-evaluation of the correctness and completeness of the benchmark. Full scale automated evaluation is considered as future works. As we are claiming *ProvMark* as an all-around expressiveness benchmarking automation system for provenance collecting tools, we also demonstrate how easy it is to extend *ProvMark* to other tools, system calls and provenance formats which are not included in our candidate list. For a fair comparison, all the experiments are done in a virtual machine with 1 virtual CPU and 4GB of virtual memory. The common operating system for the virtual machine is Ubuntu 16.04 which is auto set-up by a Vagrant file for each of the tools. Dependencies for all three provenance collecting tools, including Neo4j and some needed Python libraries, are also installed automatically by the Vagrant engine according to the configuration in the Vagrant files. Vagrant [127] is an open-source engine/tool for building and managing virtual environments and virtual machines. Users can build up their vagrant files which instruct the Vagrant engine to build up customized virtual environments. The Vagrant files include but not limited to the choice of operating systems, virtual hardware configuration, user management, software package management and initialization script after the building of the virtual systems. Also, the Vagrant engine can help the user connect to the virtual machine and provide an abstract layer for the user to manage the environments through direct or network connections. In our evaluation, all virtual machines were hosted on an Intel i5-4570 3.2GHz with 8GB RAM running Scientific Linux 7.5.

5.5.1 Performance

We first discuss the performance of the *ProvMark* system. Some performance overhead is acceptable for a benchmarking or testing tool; however, we need to establish that *ProvMark's* strategy for solving NP-complete sub-graph isomorphism matching problems is effective in practice (i.e. takes minutes rather than days). In this section, we will show that the extra work done by *ProvMark* to generate benchmarks from the original provenance result is acceptable.

We first give some sample sizes of the graphs produced by different stages, to aid in interpreting the timing results. Table 5.2 presents the sizes of the graphs resulting from different stages for the ***open*** and ***setuid*** system calls. The table shows the size

of the generalized background graph, generalized foreground graph and the resulting provenance benchmark of the two sample system calls. Each cell contains $|V|$ / $|E|$ / $|P|$ where $|V|$ is the number of vertices, $|E|$ is the number of edges, and $|P|$ is the number of properties in the graph.Thus, we can see that for SPADE, the background graph has around 20 nodes and 30 edges, with around 260 properties, while the foreground graphs for these two calls have just a few additional nodes and edges. For OPUS the graphs are larger and sparser (around 80 nodes and edges) but with many more properties; the difference is mostly due to OPUS creating many nodes to record initial environment variables. Finally, for CamFlow, the graphs have fewer nodes and edges than SPADE, but more properties (over 300). Again, the target graphs include only a few nodes and edges but over 40 properties.

Graph	SPADE	OPUS	CamFlow
Background (open)	20 / 28 / 268	78 / 78 / 461	12 / 15 / 337
Background (setuid)	20 / 28 / 270	77 / 78 / 457	12 / 15 / 337
Foreground (open)	21 / 30 / 270	79 / 79 / 471	14 / 19 / 396
Foreground (setuid)	20 / 30 / 273	80 / 79 / 477	14 / 18 / 424
Target (open)	1 / 1 / 10	1 / 1 / 7	3 / 4 / 45
Target (setuid)	1 / 1 / 3	3 / 2 / 13	2 / 2 / 45

Table 5.2: Number of nodes/edges/properties in provenance graph of tools/system calls

Next, we report measurements of the recording time. Since the number of trials varies, we report the average time needed per trial. For SPADE, recording took approximately 20s per trial. For OPUS, the recording time was around 28s per trial, and for CamFlow each trial took around 10s. We wish to emphasize that the recording time results are *not* representative of the recording times of these systems in normal operation — they include repeatedly starting, stopping and waiting for output to be flushed, and the waiting times are intentionally conservative to avoid garbled results. No conclusions about their relative performance, when used as intended, should be drawn. As a result, we only consider the overhead from *ProvMark*, so we only evaluate the time performance on the transformation subsystem, generalization subsystem and comparison subsystem.

In Figures 5.10–5.12, we summarize the time needed for ProvMark to execute each of our system call candidates using SPADE, OPUS, and CamFlow respectively. The bars are divided into three portions, representing the time needed for the three subsys-

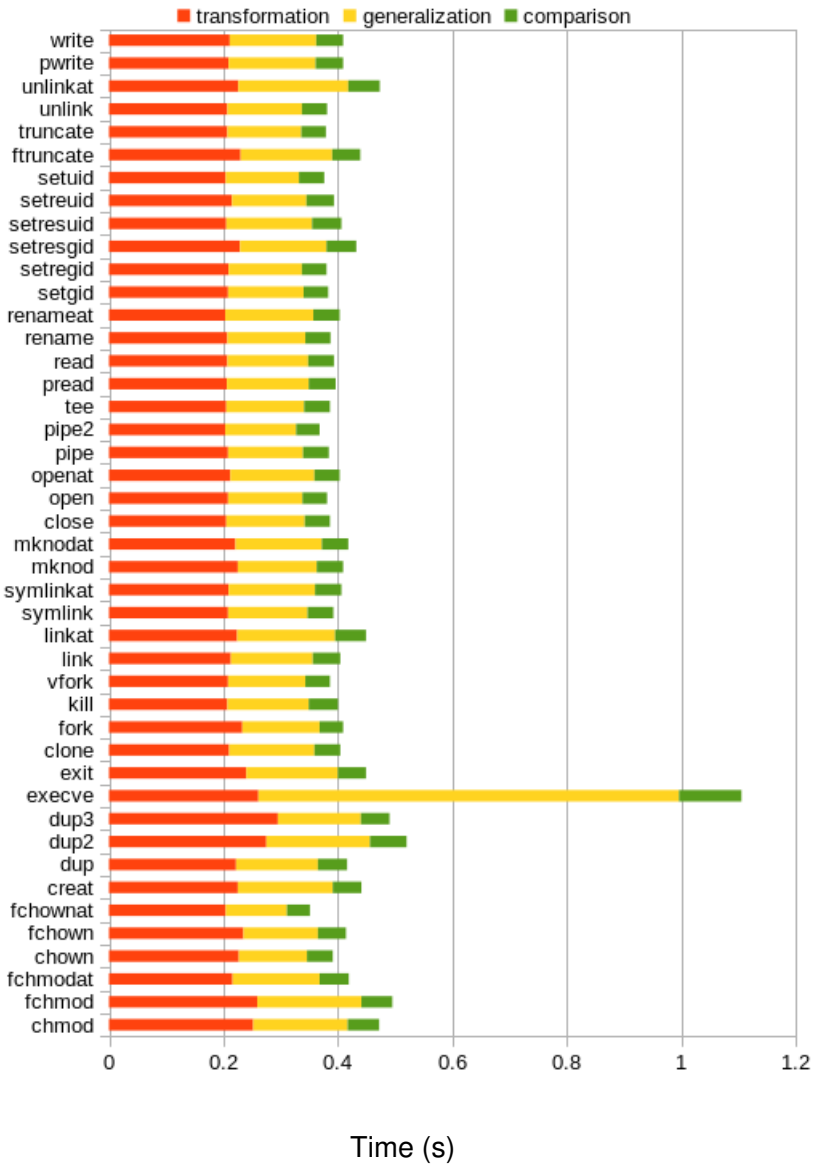


Figure 5.10: Processing time for system calls by SPADE

tems. Note that the x-axes are not all to the same scale: in particular, the transformation, generalization and comparison times for OPUS are much higher than for the other tools. It is because of the large overhead for transforming Neo4j graph data structure to Datalog format.

From the data in Figures 5.10–5.12, we can see that the transformation stage is typically the most time-consuming part. The transformation stage maps the different provenance output formats to Datalog format. The transformation time for OPUS is much longer than for the other two systems. This appears to be because OPUS produces larger graphs (including recording environment variables), and extracting them

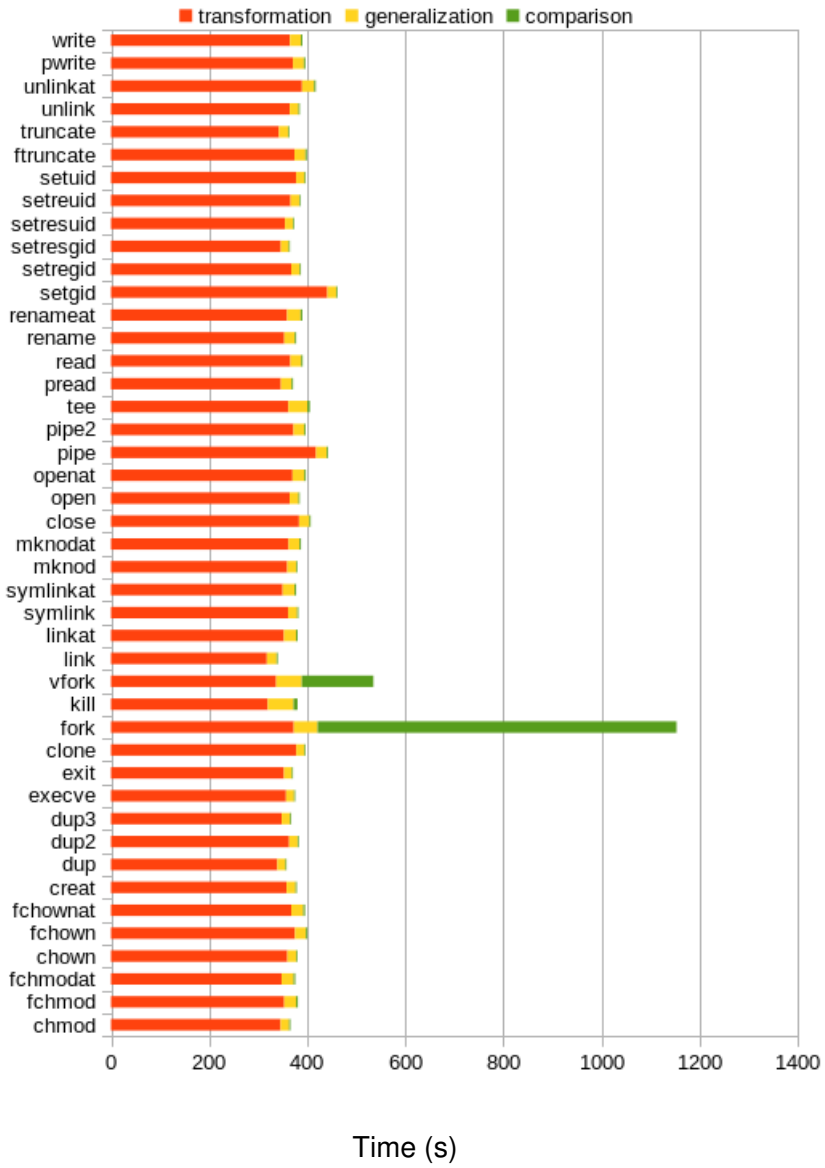


Figure 5.11: Processing time for system calls by OPUS

involves running Neo4j queries, which also has a JVM warm-up and database initialization cost which generates large time overhead.

The time required for the generalization stage depends mainly on the number of elements (nodes, edges and properties) in the graph since this stage maps elements in pairs of graphs to find an isomorphic matching pair. As we can see from the results, the generalization of OPUS graphs again takes significantly longer than for SPADE and CamFlow, probably because the graphs are larger. For SPADE, the generalization of the ***execve*** benchmark takes much longer than for other calls (though still only a few seconds) because this system call generates much larger graphs then other system

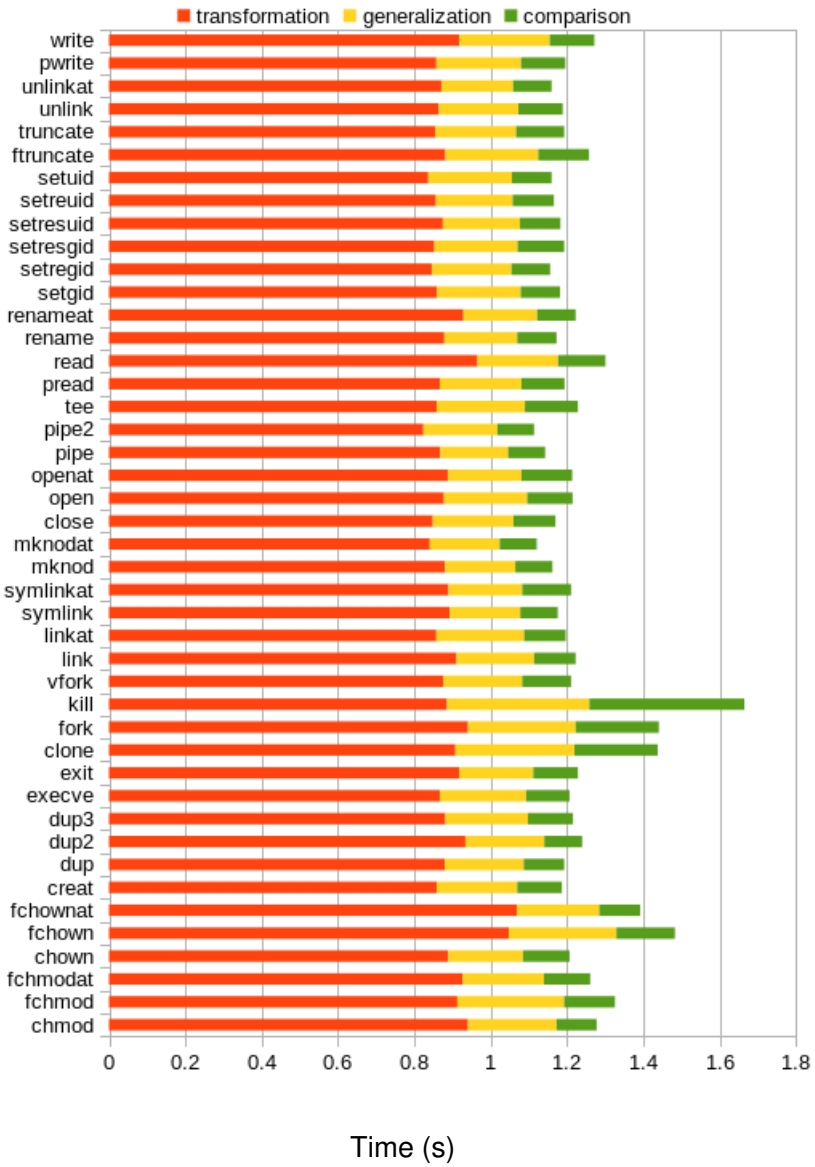


Figure 5.12: Processing time for system calls by CamFlow

call candidates.

The time required for the comparison stage is usually less than for generalization (recall that in generalization we process two pairs of graphs whereas in comparison we just compare the background and foreground graph). Comparison of OPUS graphs again takes longest, while the comparison of CamFlow graphs again takes slightly longer than for SPADE, perhaps because of the larger number of properties.

In general, we can conclude that the time needed for ProvMark's transformation, generalization, and comparison stages are relatively shorter compared with the time needed for recording. Most benchmarks complete within a few minutes, though some

that produce larger target graphs take considerably longer; thus, at this point running all of the benchmarks is an overnight activity. This seems like an acceptable price to pay for increased confidence in these systems, and while there is room for improvement, this compares favourably with manual analysis, which is tedious and would take a skilled user several hours. Besides, we have monitored the memory usage and found that *ProvMark* never used more than 75% of the 4GB virtual memory on a virtual machine, indicating memory was not a bottleneck. As a whole, the general overhead for *ProvMark* in both time and memory overhead is acceptable.

5.5.2 Scalability

Our experiments so far concentrated on benchmarking one system call at a time. The design of *ProvMark* allows arbitrary activities as the target action, including sequences of system calls. As mentioned above, we can simply identify a target action sequence using the `#ifdef TARGET` CPP directive statement to let *ProvMark* generate background and foreground programs respectively.

Of course, if the target activity consists of multiple system calls, the time needed for *ProvMark* to process results and solve the resulting sub-graph isomorphism problems will surely increase. The generated provenance graph will also increase in size and number of elements. The NP-complete complexity of sub-graph isomorphism means that in the worst case, the time needed to solve larger problem instances could increase exponentially. This section investigates the scalability of *ProvMark* when the size of the target action increases. We have included a set of benchmark programs with the same set of system calls repeating for different times. The resulting time performance demonstrates the scalability of *ProvMark* in handling a bigger set of target action sequences that are more realistic and closer to those real-world applications. In reality, if we repeat a certain system call for multiple times, the resulting provenance graph is more likely contains fewer vertices by reusing certain vertices for representing the same processes or artefacts. This helps to create a smaller graph result for analysing the scalability of ProvMark on processing graphs with larger numbers of components.

In figure 5.13-5.15, the charts show the time needed for the three processing subsystems of *ProvMark* in handling some scalability test cases on SPADE, OPUS and CamFlow respectively. Note that the x-axes are not all to the same scale: in particular, the transformation, generalization and comparison times for OPUS are much higher than for the other tools. It is because of the large overhead for transforming Neo4j

graph data structure to Datalog format. The scalability test cases range from scale1 to scale8. In test case scale1, the target action sequence is simply a creation of a file and another deletion of the newly created file. In test case scale2, scale4 and scale8, the same target action is repeated twice, four times, and eight times respectively. The results show that the time needed initially grows slowly for SPADE, but by scale8 the time almost doubles compared to scale1. For OPUS, the time increases are dwarfed by the high overhead of transformation, which includes the one-time Neo4j startup and access costs as discussed above. For CamFlow, the time needed approximately doubles at each scale factor. These experiments do demonstrate that *ProvMark* can currently handle short sequences of system calls without problems.

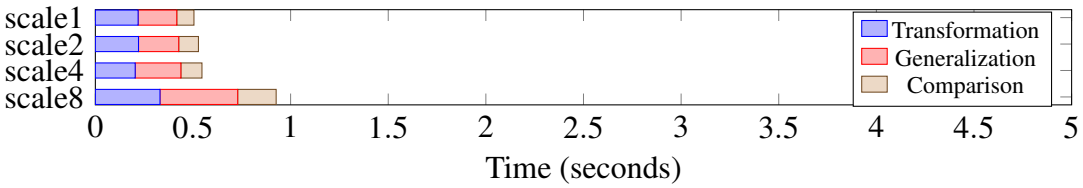


Figure 5.13: Scalability results: SPADE+Graphviz

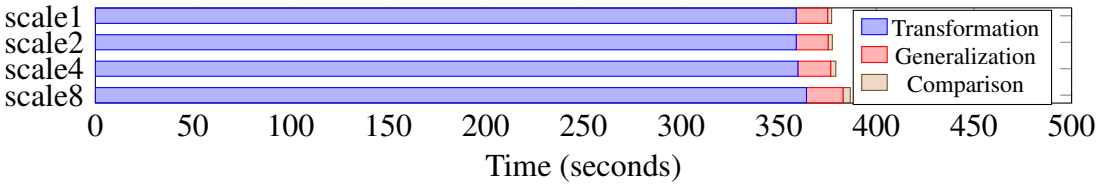


Figure 5.14: Scalability results: OPUS+Neo4J

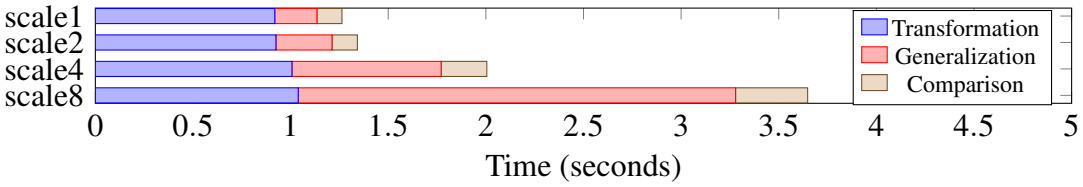


Figure 5.15: Scalability results: CamFlow+ProvJson

5.5.3 Modularity and extensibility

ProvMark was designed with extensibility in mind. Only the first two stages (recording and generalization) depend on the details of the provenance recording tools being benchmarked. To support a new tool, it suffices to implement an additional recording module that uses the new tools to record provenance for a test executable, and (if

Module (Format)	SPADE (DOT)	OPUS (Neo4j)	CamFlow (PROV-JSON)
Recording	171	118	192
Transformation	74	122	128

Table 5.3: Recording and transformation module sizes (Python lines of code)

needed) an additional translation module that maps its provenance output format to Datalog. ProvMark provides a configuration file in which one can select the needed recording and transformation modules.

As discussed earlier, it was non-trivial to develop recording modules for the three provenance recording tools that produce reliable results. In particular, supporting CamFlow required several iterations and discussion with its developers, which have resulted in changes to CamFlow to accommodate the needs of benchmarking. This architecture has been refined through experience with multiple versions of the SPADE and CamFlow systems, and we have generally found the changes needed to ProvMark to maintain compatibility with new versions to be minor. Developing new recording modules ought to be straightforward for systems that work in a similar way to one of the three we considered. It is also worth mentioning during the development of *ProvMark*, the three candidate tools have been updated. When their respective recording modules are done, it is easy to update to suit the new version of the tools. This also shows the extensibility for supporting provenance collecting tools candidates.

If a provenance tool generates data in a format not currently supported by *ProvMark*, an additional module for handling this type of provenance data is needed. But the need for new transformations should decrease over time as more tools are introduced to *ProvMark* since there are a limited number of common provenance formats. Developing a new module for supporting a new provenance format should be easy. There are already a set of templates. The only necessary thing to do is to identify the edges, vertices and properties of data in the new format and the transformation code should be easily done on top of that. Some provenance recording tools support generating provenance in multiple formats, for example, SPADE can generate provenance in both Graphviz graph and Neo4j graph database format. The user of *ProvMark* only needs to specify which tools module and graph format modules they want to use and specify it in the command line arguments or configuration file, then *ProvMark* will automatically use the needed module for each subsystem.

As shown in Table 5.3, none of the three recording or transformation modules required more than 200 lines of code (Python 3 using only standard library imports). We initially developed *ProvMark* with support for SPADE/DOT, SPADE/Neo4j and OPUS/Neo4j combinations; thus, adding support for CamFlow and PROV-JSON only required around 330 lines of code.

5.5.4 Simple expressiveness evaluation

After demonstrating the performance and different properties of ProvMark, we understand that ProvMark is theoretically capable of scaling up to some real system behaviour analysis. One of the key points for the expressiveness benchmarking proposed in Chapter 3 is to provide a unified way to analyse the expressiveness of a specific provenance system on several system call action sequences. The expressiveness is composed of two factors, completeness and correctness. Completeness of provenance ensures all the core elements for identifying an activity sequence have been included in the result which minimizes the false-negative rate. On the other hand, the correctness of provenance ensures no ambiguous or wrong elements that affect the correct mapping between provenance and action sequence have been included in the result which minimizes the false positive rate. As a result, completeness and correctness factors ensure a one to one mapping between a specific action sequence and its corresponding provenance benchmark without ambiguity.

Apart from the basic performance, evaluation of the expressiveness of the provenance benchmark result is also necessary to ensure ProvMark is possible to handle real system behaviour and activity sequence analysis. It is the most important property to be evaluated as the key motivation provided by ProvMark is the expressiveness benchmarking which acts as a base for different applications of the resulting provenance and most of the use cases we suggested in early chapters. In the development of ProvMark, there is an additional module implemented for a simple expressiveness evaluation to ensure ProvMark does complete its job correctly and the resulting provenance benchmark does correctly and completely describe the specific action sequence for further application and analysis. This subsection describes the details of the self-evaluation module of ProvMark, together with some evaluation process and result of provenance benchmark by the self-evaluation module.

5.5.4.1 Expressiveness evaluation module

One of the simple ways for the expressiveness evaluation of the generated provenance benchmark is to put them into real use, for example, by feeding them together with some random binary files to see if it can help recognize the existence of the corresponding action sequence. To test both completeness and correctness factors, the self-evaluation should be executed multiple times with different kinds of testing binaries. The expressiveness of the provenance benchmark (and the generating provenance systems) are evaluated by the number of correct judgments on the simple evaluation. The correct judgment should be positive if the specific activity sequence is found in the binary and a negative if the specific activity sequence is not found in the binary. A correct and complete provenance benchmark should be able to help discover the existence of the specific activity sequence and distinguish between similar activity sequences. This follows the correctness and completeness definition in the earlier text. Some general test cases with different binaries are shown in Table 5.4. These test cases are designed based on different numbers of activity sequences included in the binary for all marginal cases. The major idea is to demonstrate and evaluate if the correctness and completeness factors discussed in earlier chapters can be observed from the benchmark generated by ProvMark. These test cases cover different combinations to evaluate false-positive, false-negative and duplicate events which should demonstrate the effectiveness of ProvMark result.

The expressiveness evaluation module of ProvMark provides an interface for feeding the provenance benchmark and an executable binary file to test the existence of the specific activity sequence represented by the provenance benchmark. It goes through a slightly different process compared to the general provenance benchmark generation. A brief description of the expressiveness evaluation process is shown in Figure 5.16.

As shown in Figure 5.16, the testing binary first goes through the recording subsystem. In this stage, the testing binary will be executed multiple times and monitored by the corresponding provenance systems. The recording subsystem of ProvMark manages both the execution of the testing binary and the recording process of the provenance system. The result of this stage is a set of provenance graphs from multiple execution trials. These graphs are then passed to the transformation subsystem where each of them will be transformed into Datalog format for further processing. Different formats of graphs are transformed by different handlers. The resulting graphs in Dat-

Test Case	Testing binary description
EV#1	Binary that contains one set of the specific activity sequence
EV#2	Binary that contains multiple sets of the specific activity sequence
EV#3	Binary that contains a set of activity sequence that is slightly different with the specific activity sequence (For example, ***open*** and ***openat*** which opens an artefact by path or file handler)
EV#4	Binary that contains one set of specific activity sequence and one set of activity sequence slightly different from the specific activity sequence
EV#5	Binary that contains a completely different set of activity sequence (For example, ***open*** and ***rename*** which does the completely different actions)

Table 5.4: Test cases for expressiveness evaluation

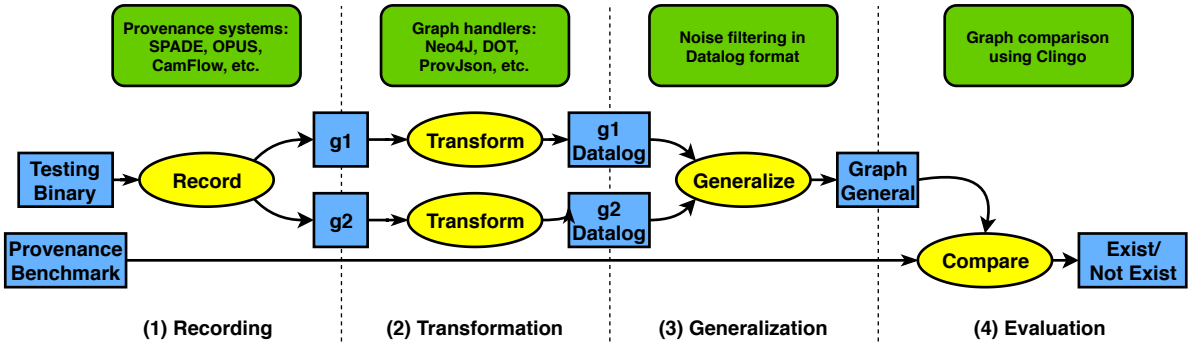


Figure 5.16: *ProvMark* expressiveness evaluation overview

alog format are compared to each other and volatile and background property labels will be filtered out as noise by the generalization subsystem. The result of this stage is a single generalized graph representing the activity sequence of the testing binary. The testing binary goes through the same process as the normal provenance benchmark generation process in the first three subsystems. Although generalization is not necessary for the simple evaluation as we did not need to filter noise, the graphs will go through the generalization subsystem to decrease the interference caused by the noise and increasing the success matching rate. The only difference happens in the last comparison subsystem as there is only one generalized graph in this process. The generalized graph for the testing binary will instead be compared to the given provenance benchmark in a new *evaluation subsystem*. The evaluation subsystem will try to iden-

tify if the activity represented by the provenance benchmark P does exist in the testing binary. It is an isomorphic sub-graph matching problem. If the activity sequence represented by P does exist in the testing binary, the same structure should appear in the generalized provenance graph G of the testing binary. As P only contains the specific activity sequence and G may contain more activities, the numbers of elements in G must be larger than or equal to the numbers of elements in P if the testing binary does contains the specific activity sequence. Thus, if the activity sequence represented does exist in the testing binary, P must form an isomorphic sub-graph relationship with G, that is P must be isomorphic to G' where G' is a sub-graph of G. We use the same isomorphic sub-graph code mentioned in the last chapter to discover if the isomorphic sub-graph relationship does exist between P and G. The evaluation subsystem will provide judgment according to the result of the answer set programming. If *unsatisfiable* is returned, it means that the isomorphic sub-graph relationship does not exist between P and G and thus the evaluation subsystem concludes that the specific activity sequence does not exist in the testing binary, and vice versa. The final result is a positive or negative judgment showing the existence of the specific activity sequence in the testing binary.

There are two assumptions in this simple evaluation process. The first assumption is the correct provenance system is chosen for the simple evaluation process. As different provenance systems generated different provenance graphs and result in different provenance benchmark, the simple expressiveness evaluation needs to know the correct provenance systems that generated the given provenance benchmark to have a correct generalized graph for the evaluation subsystem to compare and judge on the existence of the specific activity sequence. Wrong input may result in an undefined result as there is no guarantee that the same set of provenance graph elements mean the same thing for different provenance systems. The second assumption is the absence of obscure events. The simple expressiveness benchmarking evaluation needs to match the same activity sequence. Obscure or redundant activities that make the activity sequence an invariant which still performs the same effect cannot be distinguished by the evaluation module. Only the activity sequence with the same set of system call event can be detected. The application for vetting sensitive behaviour from obscurity or redundancy is considered as future works.

There is a special configuration threshold for the expressiveness evaluation process. The threshold defaults to be zero and it represents the maximum edit distance value between the provenance benchmark P and a sub-graph G'' of a provenance graph G

generalized from the multiple provenance graph generated by chosen provenance systems for the execution of the test binary. This threshold exists for error tolerance in the generalization and comparison subsystem in the original provenance benchmark generation process. In the generalization process, multiple graphs are compared together and volatile information is filtered out as noise. In this process, it is possible that some noise did not change in several trials and remain in the generalized graph and become impurity in the resulting benchmark. Thus, some level of impurity may exist in both the generalized graph of the test binary or the benchmark itself. This impurity may result in either an unsatisfiable result or an ambiguity in the evaluation process in terms of failing to distinguish activity sequences that are similar to the target activity sequence. Turning the threshold up allows a certain level of "similar" match but at the same time decreases the accuracy of the match. User needs to find a balance between unsatisfiable and ambiguous results by tuning the threshold to a suitable level. By increasing the threshold, the flexibility on similar matching is increased while the level of accuracy is decreased. This favours the completeness factor. On the other hand, decreasing the threshold increases the accuracy but decreases the flexibility and allow less impurity in the benchmark and the provenance graph generation for the testing binary which favours the correctness factor.

As shown in Table 5.1, ProvMark has been tested with a limited set of system calls on three chosen provenance systems, SPADE, OPUS, and CamFlow. All this testing is done by generating a set of provenance benchmarks. These provenance benchmarks are further feedback to the expressiveness evaluation module to see if they fulfil the completeness and correctness factors at a basic level following the five different test cases mentioned above. For test case EV#1, we are feeding the provenance benchmark together with the original foreground program. For test case EV#2, we slightly modify the original foreground program to repeat the same system call for 3 times. For the remaining test cases, the slightly different set of specific activity sequence is obtained by interchanging the system call with the other system call in the same family, for example, interchanging ***open*** with ***openat*** or interchanging ***link*** with ***symlink***. For test case EV#3, we directly feed the original foreground program of the interchangeable system call with the provenance benchmark. That is, we are feeding the provenance benchmark for ***open*** together with the original foreground program for ***openat***, etc. For test case EV#4, the testing binary is a combination of the chosen system call and its interchangeable counterpart. That is, we are feeding the provenance benchmark for ***link*** together with a foreground program that contains both the

link and ***symlink*** system calls. Lastly, for test case EV#5, the foreground program contains another system call that is not interchangeable. For example, we feed the provenance benchmark of the ***open*** system call with the foreground program of the ***rename*** system call.

We try to execute the test on some same provenance benchmark in each group for each test case. Each test is repeated for five times and the numbers of correct judgments are summarized in Table 5.5 to Table 5.7. We did not include the test for group 4 system calls using SPADE because SPADE did not have obvious provenance graph generated for any system calls in group 4. From the result, we can see most of the case give correct judgment where there is some random error exists in some of the cases. One possible reason for those random failures is because of the random loss of information in the provenance graph generation of the testing binary. Other possible reasons includes random graph matching errors during the generalization stage or comparison stage for the benchmark generation process or the evaluation process.. In general, this test result give a basic overview of the completeness and correctness of the provenance benchmark and their corresponding provenance systems.

In our test cases, we are still using testing binaries with a very limited number of system calls. Currently, we are only testing with small scale programs and aim to demonstrate the expressiveness of the provenance benchmark and show the feasibility of the ProvMark system. It is expected that the current expressiveness evaluation module does not handle well for realistic binary or application for doing some real sensitive activity vetting. In theory, some enhancement is needed to scale it up to realistic binary or application execution which results in a much larger provenance graph for the evaluation. One possible enhancement is to formalize the patterns or transform the patterns as query format to allow auto-discovery of the patterns. But this is left as one of the future work and applications of ProvMark and its provenance benchmark results. Currently, the ProvMark implementation mentioned in this thesis is still concentrating on the automation of the provenance benchmark generation and to provide a basic level of expressiveness evaluation of the result. Further applications of the result, including formal evaluations and validations of the provenance benchmark or automation testing modules, are considered as future enhancement work for ProvMark. Some of them are discussed in Section 5.3.4 and in later chapters.

System call group	System call chosen	# of correct judgment for test cases				
		EV#1	EV#2	EV#3	EV#4	EV#5
Group 1	open	5	5	5	5	5
Group 2	fork	5	5	4	4	5
Group 3	chmod	5	5	5	5	5

Table 5.5: Result for Test Cases in Table 5.4 (SPADE)

System call group	System call chosen	# of correct judgment for test cases				
		EV#1	EV#2	EV#3	EV#4	EV#5
Group 1	open	5	5	4	4	5
Group 2	fork	4	5	5	5	5
Group 3	chmod	5	5	5	5	5
Group 4	pipe	5	4	5	5	5

Table 5.6: Result for Test Cases in Table 5.4 (OPUS)

System call group	System call chosen	# of correct judgment for test cases				
		EV#1	EV#2	EV#3	EV#4	EV#5
Group 1	open	5	5	5	5	5
Group 2	fork	5	5	4	4	5
Group 3	chmod	5	5	5	5	5
Group 4	tee	5	4	5	4	5

Table 5.7: Result for Test Cases in Table 5.4 (CamFlow)

5.5.5 Summary

The main hypothesis of the *ProvMark* system is providing an automatic expressiveness benchmarking which is easy to use, fast, extensible, flexible, useful and effective to compare provenance generated by different tools and mechanisms. The comparison helps to understand the capabilities of each tool in different scenarios and helps to assess completeness and correctness problems of those tools. In the evaluation above, we provide evidence for the hypothesis in multiple directions.

5.6 Discussion

During the basic design stage of *ProvMark*, we have contacted and discussed with the developers of the three provenance recording tools candidates, SPADE, OPUS and CamFlow. We also continued to feedback our generated provenance benchmark results to them to check if the results met their expectations. One of the motivations we have mentioned at the beginning is that the expressiveness benchmarking can help the tool developers to check for problems or bugs in their tools when we provide some provenance benchmark results that are out of their expectations. It did happen several times in our early development stage that we located some of the problematic parts in the tools themselves and notified the developers to fix them. Also, from the basic performance evaluation of *ProvMark*, we can see that the additional processing of *ProvMark* is very lightweight. It shows acceptable performance despite the need to solve multiple NP-complete isomorphic graph / sub-graph matching problems. This demonstrates one of the achievements of *ProvMark* which shows that *ProvMark* provides a significant step towards the validation of such systems and should be a useful tool for developing correctness or completeness criteria for them.

This basic design and methodology mentioned in this chapter is the first stage development of *ProvMark* which aims to transform the basic functionality from the manual approach to automatic. So we only focus on deterministic input in this stage of development. We understand that most of the real usage is indeed non-deterministic, thus we put the handling of non-determinism on our enhancement list. Non-determinism (for example through concurrency or socket) introduces additional challenges: both the foreground and background binaries might have several graph structures corresponding to different schedules, and there may be a large number of different possible combinations. We may need to run larger numbers of trials and develop new ways to align the different structures to obtain reliable results or result sets. This is the top item on our enhancement list.

Also, one of the important functions for the automated approach is the auto comparison and self-evaluation of the resulting provenance benchmark. We aim to provide *ProvMark* as an automated tool which the end-user and tool developers can use as a black box. Thus we also want to have an automatic comparison of the provenance benchmark to generate results which should be easily understandable by the normal end-users. In other ways, we should also ensure the result of *ProvMark* is indeed accurate through self-evaluation to act as a cross-reference for tool developers to cross-

check their tools. We have provided a basic form of self-evaluation (as mentioned in subsection 5.5.4). We include some basic features for evaluating the generated benchmark to check for its correctness and completeness. The process checks if the benchmark results can help to identify the same pattern of system calls in other applications. The full automatic comparison (extended from current self-evaluation function) for both the developers and end-users are included in our enhancement list and are considered as future works.

5.7 Conclusion

In this chapter, we discuss the basic design and methodology of *ProvMark*, the automated approach for the expressiveness benchmarking mentioned in chapter 3. We choose three representative provenance recording tools which operate and gather information of runtime processes at a different level and make them generate provenance graph results on a set of target system call action sequences. Then we handle the results and produce benchmarks as output. The whole process works as a black box. Besides, we have provided some basic evaluation of *ProvMark* in this chapter. The two major enhancements of *ProvMark* mentioned above are the work we have proposed after the first batch of development. The handling of non-determinism is done and included in the next chapter, while the automated comparison and evaluation extended to form the current simple self-evaluation is included in the future worklist.

Chapter 6

Non-determinism

In the last chapter, we presented the basic design, development and methodology of *ProvMark*, the automated approach for the expressiveness benchmarking. This is only the first stage of development. One of the enhancements mentioned is non-deterministic event handling which aims to make *ProvMark* able to handle more different action sequences which are closer to real-world examples. In this chapter, we will discuss non-determinism and how *ProvMark* handle *non-deterministic* sources of information. We will discuss the design and methodology on top of the first stage of development to handle non-determinism.

6.1 Motivation

6.1.1 Non-determinism in real-world example

Starting from chapter 3, we have continuously talked about expressiveness benchmarking of provenance graph and provenance recording tools. We study how to go from a manual approach to an automated approach and how to get an optimal solution for graph comparison and matching by Answer Set Programming. All the experiments and contents in our test case are focused on the smallest meaningful unit, system calls. In the manual approach and the first stage development of *ProvMark*, we only considered deterministic events of system calls because they have less uncertainty and more similar results in each trial run and require less effort to handle. But when we take *ProvMark* into real-world usage, especially for some of our proposed uses in security forensics or auditing which require to identify the existence of accountable components of some sensitive action sequences, it is hard to avoid the non-deterministic character-

istics in the action sequences. In current computing age, most of the real-world system and action sequences adopt synchronized processing which increases the efficiency and speed of the operations. This concurrent processing is an example of non-deterministic events in which the process owner has no control over the interleaving between processes or the execution order of system calls coming from different processes. This is only one of the examples of non-deterministic events, there are many more different types of non-determinism in real operating system processes. To cover these cases and provide all-around automated expressiveness benchmarking for real-world examples, we need to enhance *ProvMark* to handle non-determinism. To make *ProvMark* more useful out of the experimental stage, one of the basic criteria is to add non-deterministic monitoring support to *ProvMark* to allow it to handle more realistic cases on top of the experimental cases. This creates a strong motivation to enhance *ProvMark* to make it useful in real-world scenarios and provide more automation in the topic of expressiveness benchmarking. This enhancement makes *ProvMark* closer to what the tool developers and end-users need.

6.1.2 Scale of non-deterministic event

In chapter 5, we mentioned that one of the motivations for building up the fully automated system *ProvMark* is the scale of the provenance graph and the labour intensiveness and the error rate of the manual approach. This is also one of the strong motivations for handling non-deterministic events. In the deterministic situation, the foreground graph and background graph generated in the intermediate stage should be almost isomorphic to each other, this is one of the assumptions we made when we performed the generalization stage to eliminate volatile data. So, as a result, there will be only one generalized foreground graph and background graph after the generalization step. The major reason is that deterministic events will always perform the system calls involved in the same order. For non-deterministic situations, this is not the case. For example, if our target action sequence involves one thread executing a ***`read`*** system call and another thread executing a ***`write`*** system call in parallel, there may be two versions of foreground graphs generated and the graphs in different versions are not isomorphic to each other and as a result, the generalization process should generate two generalized foreground graphs. The number of versions (or generalized graphs) depends on the size of the non-deterministic portion in the benchmark program and increases exponentially. For example, if the two threads execute two system calls each,

the non-deterministic combination has six different versions. What is worse than that is, it is hard to predict the distribution of the versions. This increasing size and scale and uncertainty in non-deterministic event graph generation makes it hard to handle by manual effort. As mentioned above, we cannot avoid these non-deterministic events as they are very common in real-world execution. To make *ProvMark* more realistic and useful, we have to add in the automated handling of non-deterministic events in *ProvMark* to withstand the exponentially increasing scale of non-deterministic events.

6.1.3 Obfuscation and non-determinism

Another top motivation of the non-determinism handling enhancement comes from the consideration of practical usage of *ProvMark*. As mentioned in the earlier chapters, one of the practical uses of *ProvMark* for end-users is to identify the existence of certain action sequences and to trace back the accountability of parties responsible for certain actions. These may be related to some sensitive or malicious activities. Sometimes the authors in these cases may try to add in some noise in these actions to obfuscate some analysing and detection tools based on signatures. These obfuscated action sequences may also involve non-deterministic events which may come from random morphing, repeating actions or out of order execution. To capture the patterns and benchmarks for these activities and to identify the existence of certain actions, it is necessary to capture all combinations of the obfuscated events which means to capture all possible variants of the non-determinism that either come from manual change of action sequence or automated morphing. As it is not certain what will be fed to *ProvMark* in each execution, we have no idea if the input contains non-deterministic events at all. The best way is to treat all input as a potential non-determinism source to capture all possible sequences. This motivates the need for *ProvMark* to handle non-determinism.

6.2 Definitions

6.2.1 Determinism and non-determinism

In chapter 3-5, we are talking about the expressiveness benchmarking from the manual approach to the automated approach, with the description of the obstacles of isomorphic (sub)graph comparison and the automated *ProvMark*. Along the way, we give multiple examples of each of the processes, including sample benchmark graphs, sets

of generalized foreground and background graphs and the sample benchmark results for different Provenance Recording Tools. To limit the size and complexity of those tests, we limited the benchmark programs to contain only minimum meaningful system calls. Also, all of them are deterministic events. All non-deterministic system call events or sequences have been filtered out in the testing. But, as mentioned in the motivation above, the exclusion of non-deterministic events is not as realistic as most of the real-world execution includes some level of non-deterministic events to increase process efficiency and time. Thus we are enhancing *ProvMark* to handle non-deterministic events in this chapter. But first, we need to define the difference between determinism and non-determinism.

The expressiveness benchmarking process requires comparisons of background graphs and foreground graphs to identify the additional elements that exist in the foreground graph which represent the target action sequences enclosed in the benchmark program by the `#ifdef TARGET` CPP directive statement. In our context, we are only considering the non-determinism within the target action sequences. So the range of *determinism* and *non-determinism* only applies within the `#ifdef TARGET` CPP directive boundaries. We define *determinism* or *deterministic events* within the `#ifdef TARGET` CPP directive statement as the combination of system calls that are always returning the same set of kernel action sequences with the same order in different runs, except for the volatile information including those process id and timestamps which could change across runs. We further define *non-determinism* or *non-deterministic events* within the `#ifdef TARGET` CPP directive statement as the combination of system calls that are returning either different sets of kernel actions or same set of kernel actions with different execution order across different runs. There is s possibility that the non-deterministic events return the same set of kernel action sequences with the same order in different runs, but in general, if we execute the same set of non-deterministic events multiple times, we are likely to observe several different combinations or orders of the kernel action sequences. The repeating action sequences may be used to determine the likeliness of the appearance of certain action sequences in the non-determinism environment.

6.2.2 Non-deterministic events

As mentioned in the last subsection, non-deterministic events refer to some system call combinations that may return different kernel action sequences across different

runs. The major reason for the unpredictability either comes from the system calls themselves or combinations of multiple system calls. There are multiple types of non-deterministic system calls that can result in non-deterministic events they are mainly classified into different categories. *Concurrent system calls* can handle multiple threads and processes but do not have much control over the execution order of the multiple threads and processes. *Socket system calls* are those system calls handling communication through network ports. *Streaming system calls* are another kind of *non-deterministic events* that handle data transfer across different artefacts or even in a distributed environment through networks. *I/O system calls* control the input and output events and also data buffering which also belongs to the non-determinism family. Last but not least, there are some adversaries make use of some *randomized system calls* to create obfuscation to avoid showing the patterns of their attack or some other sensitive activities. These system calls also belong to the family of non-deterministic events. The detailed description of each type of non-deterministic event and their behaviour in *ProvMark* are discussed in the later section in this chapter.

6.2.3 Fingerprinting and activity tracing

In our usage of provenance tracing, one of the work is to identify the kernel execution patterns for certain action sequences. The same set of action sequences may behave in a non-deterministic manner in the kernel throughout multiple executions. Thus it is not possible to generalize and compare the graphs directly because graphs generated for different execution path may not contain isomorphic sub-graph between them. To allow further processes and covering most of the patterns or benchmark for a specific execution sequence, we need to distinguish the provenance graph and collect them into groups of same execution paths before moving on to the later steps. To distinguish the multiple sets of non-deterministic events in the provenance graph format, we need to trace the activities involved in the kernel action sequences represented by the resulting provenance graph. In a Unix-like environment, there are many tracing tools at the kernel level. We could also customize our module to do the job like CamFlow but that would require the system to run on top of the customized kernel module and requires kernel access to do so. We choose an easier approach to make use of the existing activity tracer *Ftrace* [143, 47, 144]. Ftrace is a tracing utility built and residing in the kernel. It is derived from two well-known tools, the **latency tracer** and the **logdev** utility. They combine to help us monitor the activities and events (based on functions)

happening in the kernel and return debugging information of all executions and action sequences. In addition to tracing activity sequences, it can also help to analyse latencies and performance issues which are out of our scope. Similar to the LSM hook of the CamFlow, Ftrace is a kernel utility which requires some module to pass the result back to the user level for processing. There are some user-level front-end tools available to help the user communicate with Ftrace and receive the result from it. We choose *trace-cmd* to act as the front-end of Ftrace which is a tool shipped with many Linux distribution. It can configure, start and stop Ftrace event and function tracing and retrieve results from the kernel. It will also process and filter the results according to the configuration.

As we mentioned above, non-deterministic events will generate multiple sets of kernel action sequences or similar sets with different orders. For example in Code Snippet 6.1, we put two system calls in separate threads, both of the system calls will be executed eventually, but the executing orders are non-deterministic for each trial runs. Another example in Code Snippet 6.2, we apply randomization to conditional branches, different system calls set are executed determined by the result of the randomize source. Different system calls may be executed for each trial runs. It is also possible for the same kernel action sequence with the same order to appear more than one time across multiple runs. To handle different provenance graphs representing different possible executions and to preserve the generalization features which aim to remove volatile information, we need to distinguish graphs and groups the similar graph together. We make use of the *event tracing* features of Ftrace to help us identify and group the generated provenance graphs. As we know, Ftrace also aims to record the actions sequences, events and functions that happened in the kernel level, so if the same schedule happened twice, their Ftrace result should also be similar. In this case, the Ftrace result can act as the *fingerprint* for a certain combination of kernel action sequences and the related provenance graph generated for this combination. We only need to match the fingerprints to group the generated provenance graphs. The process to distinguish the generated provenance graphs by comparing their fingerprints (Ftrace results) is defined as *fingerprinting* and it is the additional step added to *ProvMark* between the recording subsystem and generalization subsystem. Although we can not guarantee that all paths are executed in the trial executions, we at least can make sure that the process of generalization, comparison and benchmark generation is only done between provenance graphs representing the same execution paths. This action avoids polluting the patterns and benchmarks with non-isomorphic (sub)graph pairs.

```
#include<fcntl.h>
#include<unistd.h>
void main() {
        if(fork()) {
                chmod("txt1.txt", S_IRUSR|S_IWUSR);
        } else {
                chmod("txt2.txt", S_IRUSR|S_IWUSR);
        }
}
```

Code Snippet 6.1: Example for non-determinism with different orders

```
#include<fcntl.h>
#include<unistd.h>
#include<time.h>
void main() {
        srand(time(0));
        if (rand()%2 == 0) {
                chmod("txt1.txt", S_IRUSR|S_IWUSR);
        } else {
                chmod("txt2.txt", S_IRUSR|S_IWUSR);
        }
}
```

Code Snippet 6.2: Example for non-determinism with different system calls

6.2.4 Assumptions

As we aim to retrieve most of the possible schedule of the non-deterministic events, those events should be our target actions. Thus, all of the non-deterministic events should be enclosed by `#ifdef TARGET` CPP directive statement inside the benchmark program. As a result, the non-determinism should only exist in the foreground graph generation process because the background program used for generated background graph does not contain parts enclosed by the `#ifdef TARGET` CPP directive statement and thus does not contain non-determinism at all. Thus all the generated background graph should have the same fingerprint. So we are only doing the fingerprinting on the generated foreground graph. And because foreground graphs in different groups are generalized separately, the resulting generalized foreground graphs are also compared separately and as a result, we will receive multiple benchmarks which each of them represent one of the possible action sequence combinations of the non-determinism.

Although symbolic execution provides ways for seeking all possible execution paths for non-deterministic inputs, it does not fit in most of the scenarios for our automated expressiveness benchmarking approach. One of the major motivations for our automated approach is to identify the patterns for certain activity sequences and use them as a reference for later detections. In this scenario, the later detection process includes the execution of a new binary and compares the provenance trace with the referencing patterns to identify if the given activity sequences exist in the new input binary. This process aims to identify the patterns in some random input which the system has never seen before. In most of the symbolic execution approach, the target execution needs to go through some static analysis and certain levels of code intrusion to hook some of the interpreters to the execution. These steps may need to be repeated multiple times to retrieve all possible paths and the time and resources needed is directly proportional to the number of possible paths for the execution. These extra requirements make it unsuitable for the automated provenance benchmarking approach for two reasons. The first reason is the unpredictable time and resources needed for the discovery of all possible paths. The second reason is we can never be certain that we can access the source for the incoming binary executions thus it is no guarantee that we can use the symbolic execution approach on every source and testing target. Thus we are not using this approach in our automated benchmarking system, ProvMark.

In some of the cases, the execution of non-deterministic input will result in limited combinations of possible outcomes. For example, if our target actions only contain a read system call in one thread and a write system call in another thread, then the resulting provenance graph can only have two possible combinations. They either contain read then write or write then read. There are no additional executions that can be deduced from this input. But, the counting of all possible combinations may not be possible if the number of system calls increase or more complex non-deterministic conditions are introduced. Also, as the non-deterministic event execution and schedules are out of our control, there is a possibility that some combinations have far lower chance to be triggered and we can never guarantee to execute all combinations of the non-deterministic input. For easy processing of *ProvMark*, we assume that all groups of graphs collected from the recording stage have covered all possible combinations of the input. We do that by running multiple times more than the total number of combinations. For example, we will execute $12\tilde{1}6$ trial runs for non-deterministic events that have 4 combinations to covers all of the possible combinations. We then process the graph generalization and benchmarking processing on top of this assumption.

6.3 Sources of non-deterministic events

In this section, we classify possible non-deterministic events.

6.3.1 Concurrent and interleaved events

In the advancement of computer science, researchers are always aiming to increase execution efficiency. One way to achieve the goal is the adoption of parallel computing for concurrent units. This concurrency property allows different portions of the programs or framework to be executed in a partial order. The key point here is although the portions may work out of order, the final result is still satisfying the original intention. So concurrency allows decomposing a single process into multiple processes and execute them in parallel following only a partial order and result in the same outcome.

To achieve the outcome from concurrency, the sliced portions of the process must interleave events. It means that the order of events in different portions of the process in the parallel execution environments should not affect each other. If some events must preserve some sort of order, they should be grouped into the same execution thread or coordinated using semaphores or locks. Besides, concurrency allows processes to share processor resources. It allows a single processor to handle more than one process by continuously interleaving events from different processes. This flexibility increases the number of active process in a limited amount of processors. The setting is related to the property on concurrent execution. While the execution order in the same portion of the program will be preserved, the process itself has no control of the interleaving of different concurrent events. There is no guarantee of execution order between events in different threads or processes in each run. It is completely up to the processor mechanism to decide when to switch to another process and only the execution of the events in the same thread are kept. This uncertainty of the execution order of interleaving events creates non-determinism because in each run, although the events in the same process may decompose into the same sets of execution threads, we never have control of the execution order of events across different threads. This may result in different provenance results. These provenance results may have the same set of system calls captured, but they may be in a different order and this may affect the structure of the resulting provenance graph. When we consider one of the use cases of *ProvMark* is aiming to discover the existence of certain patterns in runtime, it requires not only identifying certain system calls. The execution order of the system calls can also be a key factor and thus we need to handle the non-determinism of events.

6.3.2 Socket and network communication

In the last subsection, we mentioned about the most common kind of non-deterministic events, which is the concurrent event. Another important type of event is distributed processing, which includes socket and network communication. This kind of events generally can communicate externally, like from the Internet, Intranet or distribution environment. For example, the ***send*** system call family and the ***recv*** system call family are aiming to communicate with external devices through the socket protocol. In general, these system calls depend not only on the execution in the local environment, but they also depend on feedback from an external system. One of the major considerations of communicating with external parties is there is no guarantee when the communicating target will send its responses back to the local environment. And it is also inconceivable to halt local execution and passively wait for the response. Thus these socket and network communications are always working in asynchronous mode and passively wait for the reply from external communicating partners. These system calls will block themselves in the background after their core work in the local environment, then they will reactivate when a response is received from external communicating partners and complete the remaining tasks.

These socket and network-related system calls work in asynchronous mode and the waiting time is non-deterministic because the local system has no way to control artefacts outside of the system which can respond at any time in any order. They are similar to concurrent execution. They do not have control over the execution order when working in parallel with other activities, including other socket related system calls. The only difference is, in general, concurrent execution, the programmer can assign tasks to different execution threads. They cannot control the execution order between the execution threads, but they still can control the order of event execution in the same thread or process. In the case of socket system calls, even the owner has no control over the moment that the reply returns from external partners, thus the socket system call is forced to work concurrently and thus are guaranteed to be non-deterministic by its property.

6.3.3 Piping and buffering

The next group relates to process communication and input/output events. This group of events can be considered as a special case for the concurrent execution in a single system, mentioned above in Section 6.3.1. The different is, buffering and piping are

related to the resource management of the operating systems. The programmers or system users can only suggest the operating systems to perform the actions but the systems are not guaranteed to perform these actions immediately. In other words, the programmers or system users have not much control over this group of events. In general, the operating systems will perform this group of events when necessary. The necessity of the events is determined by the optimal plan. For example in current operating systems, reading and writing to storage devices is expensive, thus many of the input and output events (like ***read*** or ***write*** system calls) are buffered in the cache and will only flush the data intermittently when the buffer is full or the processor is free to perform the synchronization to a storage device. If the binaries do not have execution order controlling statements like semaphore flagging that force some events to perform before other events, the users generally have no control on how the input/output executes in the system, the execution sequences depend on the buffering settings and the resource optimization strategy of the operating systems. Thus this property creates a non-deterministic ordering of events which relates to the buffering mechanism and resource management of the operating systems and affects the resulting provenance graph.

In addition to buffering, in Unix-like operating systems, processes communicate and pass information to each other during execution. It is possible to redirect input and output through the ***pipe*** system call family. As mentioned above, processes can be executed in parallel and thus the input and output redirection or process communication may not operate at the same time. It is possible that receiving processes are inactive during the communication process and thus the order of execution is non-deterministic as once again, the process itself has no control of the execution order across processes. The different execution order of system calls in a different process creates non-determinism in the resulting provenance graph. Considering processes communication and data passing, it also makes use of the buffering mechanism to temporarily store the intermediate results between the processes which add another source for non-deterministic events.

6.3.4 Obfuscation and randomization

Last but not least, some man-made situations may cause non-deterministic events. One case is process obfuscation. As mentioned at the very beginning, one of the use cases of identifying certain action sequences is to detect the existence of malicious or sensitive

actions. This is a kind of security analysis of the attackers' behaviour by past activity tracing. Those attackers may want to cover their traces and make it hard to identify the existence of themselves and their actions. Thus they may include some dummy activities or system calls manually in their activity traces to make their action sequence traces different on each run. This is possible by adding some randomized obfuscation into their actions. They can also add in some combinations of action sequences that cancel each other out logically and achieve the same result. This can also be some live modifying of binary and code which create non-determinism in the execution and make the analysis harder.

Apart from purposeful obfuscation by attackers, randomization in legitimate programs may also create non-determinism. If some logic or action makes execution choices based on randomized values, and the randomized source or seed is changed in different trial runs, it is possible to create non-determinism, thus there is no guarantee the same branches will be executed next time. This kind of non-determinism can be minimized by minimizing the uses of the pseudo-randomized source. It can limit the number of possible non-deterministic states but can never eliminate it as sometimes the randomized source is uncontrollable. In general, randomization and obfuscation create non-determinism which is controlled and originated from the program owner itself, not the properties of some system calls.

6.4 ProvMark non-deterministic handling

In this section, the non-determinism handling mechanism is described in detail. In general, non-determinism handling involves additional components added to the four subsystems of *ProvMark*. The design aims to be general enough to allow *ProvMark* to handle both deterministic and non-deterministic inputs using the same logic.

6.4.1 Tracing kernel actions

In the recording subsystem, *ProvMark* aims to use the chosen module to control the Provenance Recording Tools to collect provenance for the background and foreground program execution. The process will execute multiple times to collect a set of background graphs and a set of foreground graphs for the generalization process, which aims to filter out volatile information like process ids and timestamps. As mentioned above, the generalization process in Chapter 5 assumes the input is deterministic. If

ProvMark is fed with non-deterministic input, there may be non-isomorphic pairs in the set of foreground graphs because execution is non-deterministic and thus the resulting provenance for each trial run may result in different provenance graph structures. This makes it impossible to generalize directly because we assumed the graphs are similar to each other to identify the volatile information which is the varying part of the similar graphs. To solve this problem, we need to first group similar sets of graphs and generalize each of the groups separately.

The first thing we need to do is to identify which of the graphs represent the same schedules of non-deterministic events. As we notice, if two trial runs follow the same schedule, then they will have the same set of system calls and execution order. Thus we can identify them by matching their system call order list. This could be retrieved by monitoring the Linux Security Enhancement (Linux SE) in the kernel as each of the system call execution need to go through Linux SE for permission checking before execution. To decrease dependencies from writing customized kernel modules to do the monitoring directly, we instead choose to use an existing kernel framework Ftrace to handle the monitoring work. As mentioned in Section 6.2.3, Ftrace is a framework shipped with the Linux kernel, aiming to monitor and trace functions and activities at the kernel level. There is also a tool named trace-cmd that allows users to configure Ftrace from user level and it will also pass back the result of Ftrace from the kernel to user level. With this understanding, in each trial run of the foreground program, we also use trace-cmd to start Ftrace alongside the provenance collecting tools to capture the list of system calls for the execution of the foreground program. We also configure the provenance collecting tools to ignore the system calls generated by the process of controlling trace-cmd and any of its child processes to avoid additional provenance from the trace-cmd utility and the Ftrace framework. After each trial run, we get a system call schedule to match with each of the foreground provenance graphs. As we assumed that all of the non-deterministic input is enclosed in ***`CPP directives`***, thus background graph is always deterministic and so all of them should have a very similar set of system calls received from the *trace-cmd* components. This statement should also be true for foreground graphs of deterministic input.

6.4.2 Fingerprinting action sequence

As mentioned above, each of the provenance graphs generated from the recording subsystem will have a schedule attached to them which is collected from the Ftrace

framework and passes to the user by the trace-cmd utility. Although in our testing, we are considering small sets of system calls initially, we eventually intend *ProvMark* for real use on larger target action sequences. Large target action sequences not only produce larger provenance graphs for analysis but also contain a large number of system calls to be executed. Thus, it will result in a very long schedule captured by Ftrace and returned to the user. These schedules are matched to each of the graphs and we are grouping the resulting provenance graphs in terms of the schedule. To minimize time and increase efficiency, it is important to control the size of these schedules. It is not good to compare the schedules directly as their size is directly proportional to the number of system calls that exist in the execution. Besides, labelling graphs with a long schedule is hard to read as it is also used manually to distinguish different benchmark schedules. To limit the size of the comparator, we generate a hash value for each of the schedules. We first concatenate all the system call names in the schedule to form a long string, then we generate a hash value for this long string to form the fingerprint which is guaranteed to be a fixed size. This fingerprint is used directly as the key for the graph it is attached to and replacing the long schedule. After this step, we get a set of foreground provenance graphs and a set of background provenance graphs, each with its fingerprint attached to it. As we assume that background program is always deterministic, all the background provenance graphs should have the same fingerprint. This can also help to filter out the minority with a different fingerprint which may represent errors in the graph generation process for the background graphs set.

6.4.3 Group action sequences

The next step is to group the foreground graphs into similar groups and generalize the graph sets separately. From the last step, we have already labelled each graph with a fingerprint, which is the hash value generated from the system calls schedule for the graph it represents. In this process, graphs with the same fingerprint will be grouped because it means that they executed the same system calls in the same order. As we are assuming all of the possible combinations of the non-deterministic events should be executed, the total number of groups should be equal to the number of the possible combinations. This statement is also true for deterministic events because the number of possible combinations for deterministic input should be one. After the grouping of the graphs, each group is passed on to the generalization subsystem to continue the process, the fingerprint for each of the group should also be preserved.

After the additional fingerprinting and group of graph done in the recording subsystem, it needs to go through the remaining three subsystems. It will go through the transformation subsystem first and transform each of the graphs into Datalog format for the generalization and benchmark comparison steps. This step is not changed as the transforming of graphs from one graph type to another graph type is always a one to one mapping. The only thing that needs additional care is that the groups of the graphs and the attached fingerprints should be preserved. The result for the first two subsystems should return a set of background graphs and one to many sets of foreground graphs.

6.4.4 Generate multiple benchmarks

The clear difference from the handling of deterministic events is that there are multiple groups of graphs that need to go through the generalization and comparison process. In the case of deterministic inputs, we assume that all graphs should be similar to each other and thus can be generalized directly and compare the generalized background and foreground graph directly to retrieve the provenance benchmark. In the case of non-deterministic inputs, there are multiple sets of foreground graphs, only the graphs in each set are similar to each other. Thus only the graphs in the same set will be generalized together. As a result, there will also be multiple generalized foreground graphs. Each generalized foreground graph is associated with a unique group fingerprint and is compared to the background graph one by one to retrieve a provenance benchmark pattern. At last, *ProvMark* generates a set of provenance benchmark graphs after processing non-deterministic program input and each of the provenance benchmark results are labelled by their fingerprint.

Recalling the usage and purpose of *ProvMark* and the expressiveness benchmarking, we aim to identify the key elements of a provenance graph that represent the target action sequences correctly and completely which can map the graph back to a set of action sequences with one to one relationship. The resulting provenance benchmark represents the key elements to describe a certain action sequence for that specific provenance recording tools, thus it acts as a benchmark for the tools. It can also be used as a pattern to identify the existence of the action sequence in a runtime environment if the same provenance collecting tools is used in the provenance collecting phase. If we have these considerations in mind, it is not hard to understand why the provenance benchmark generated for non-deterministic input programs are a set of

benchmarks rather than a single benchmark. For non-deterministic input program, we do not know which combination will execute in each trial. Thus we need to collect all possible combinations of executions and map them one by one to the current trial. If any of the benchmarks matched or exists in the new trials, we can still confirm the existence of the matching of non-deterministic action sequences and we can even label by their fingerprints. If the benchmark process covers all possible combinations, the generation of the set of provenance patterns for each of the combinations should be complete in describing all possible behaviour of the non-deterministic input program. Thus it should cover all possible future runs and guarantee the identification of the existence of execution for either of the combinations. This, however, is not guaranteed in the current implementation because we still have no way to ensure all combinations are covered. This remains a future enhancement for *ProvMark*.

6.5 ProvMark evaluation

This section provides a basic evaluation and comparison of *ProvMark* for working on deterministic and non-deterministic action sequences. We compare the actual overhead of the *ProvMark* components and subsystems for handling deterministic and non-deterministic inputs. We also provide the result demonstration for non-deterministic events to show a reference of how the result differs from handling deterministic events, thus how has the performance been affected. Again, all the experiments are done in a virtual machine with 1 virtual CPU and 4GB of virtual memory. The common operating system for the virtual machine is Ubuntu 16.04 which is set-up by a vagrant file for each of the tools. Dependencies for all three provenance collecting tools, including Neo4j and some needed Python libraries, are also installed automatically by the vagrant engine according to the configuration in the vagrant files. All virtual machines were hosted on an Intel i5-4570 3.2GHz with 8GB RAM running Scientific Linux 7.5. These settings are the same as the performance evaluation for deterministic inputs.

6.5.1 Result for non-deterministic event

As described above, the major difference in handling deterministic events and non-deterministic events is the number of benchmarks generated as the final result. In general, for deterministic events, there is only one structure for each of the foreground graphs and background graphs because all the activities and system calls should be

the same for every trial run. Even the execution order should be the same. Thus the resulting provenance always follows the same set of system calls in the same order and results in the same graph structure. Then the generalization for the set of foreground graphs and the set of background graphs can work directly and be compared to each other to generate a single provenance benchmark result. In the presence of non-determinism, there will be not much difference in the recording subsystem. Ftrace works in parallel with the provenance recording tools to capture the system call lists which are then hashed to form the fingerprints and mapped to the generated foreground graphs. This action does not affect either the original process or the transformation process because the number of graphs captured is similar, the only difference is that the graphs will be divided into groups for separate generalization and benchmarking processes. So the main overhead and performance difference will be in the generalization and comparison subsystems. In general, the number of trials required for non-determinism is much larger than determinism. This is mainly because there is only one possible combination for determinism. In general non-deterministic input, there are multiple paths for the execution and thus will have multiple groups of graphs that need to be generalized separately. To perform generalization to filter out noises for each of the groups, we need to retrieve multiple graphs for each of the possible executions. For non-deterministic input, there are at least two different execution schedules. To cover all schedules, we need at least double or triple number of trials compared to deterministic input which always contains one execution schedule. As a result, the number of trials required for non-deterministic inputs is generally much larger than the number of trials for determinism input. Although the number of trials required for non-deterministic events is much larger, the performance of the first two subsystems will not have much difference if they are comparing the same numbers of trial executions.

Code Snippet 6.3 shows a benchmark program with non-deterministic input. It contains two threads, one thread performs two read events and the other thread performs two write events. For this non-deterministic input, the total possible number of schedules is 6. We labelled the two write events as W_1 and W_2 in order and the two read events as R_1 and R_2 in order. Because the two write events and the two read events are in the same thread respectively, W_1 must be performed before W_2 and R_1 must be performed before R_2. Thus the only possible combinations are shown in Table 6.1.

```
1  #include <time.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  int main() {
6  int id=open("test.txt", O_RDWR);
7  char buf[1];
8  #ifdef PROGRAM
9  if (fork()) {
10     read(id, buf, 1); // R1
11     read(id, buf, 1); // R2
12 } else {
13     write(id, "TEST", 1); // W1
14     write(id, "TEST", 1); // W2
15 }
16 #endif
17 close(id);
18 }
```

Code Snippet 6.3: Sample benchmark program for *non-deterministic input*

Path #1	$W_1 \rightarrow W_2 \rightarrow R_1 \rightarrow R_2$	Path #2	$W_1 \rightarrow R_1 \rightarrow W_2 \rightarrow R_2$
Path #3	$W_1 \rightarrow R_1 \rightarrow R_2 \rightarrow W_2$	Path #4	$R_1 \rightarrow R_2 \rightarrow W_1 \rightarrow W_2$
Path #5	$R_1 \rightarrow W_1 \rightarrow R_2 \rightarrow W_2$	Path #6	$R_1 \rightarrow W_1 \rightarrow W_2 \rightarrow R_2$

Table 6.1: Possible execution path combinations of Code Snippet 6.3

If we execute the trial for enough trials, most of the combinations from the non-deterministic input should be covered, but there is no any guarantee that all combinations are covered because there may exist some less frequent combinations. From the above example non-deterministic benchmark program, we can see that there are at most 6 possible combinations, and Figure 6.1-6.6 shows the six different benchmarks (generated by SPADE). Some of them are very similar to each other with a similar structure of edges and vertices but demonstrate system call events in different orders. As ***write*** system call will trigger version update of the files, thus different order of the ***write*** system call does affect some edges as they are pointing out from a different version of the same file. When we compare each of them to the deterministic benchmarks shown in the previous chapters, they are very similar. For conclusion, the difference in non-deterministic input handling is the number of benchmark results.

Each of the separate results represents a combination of the non-deterministic input and can be treated as similar to six different (and separate) deterministic inputs in general.

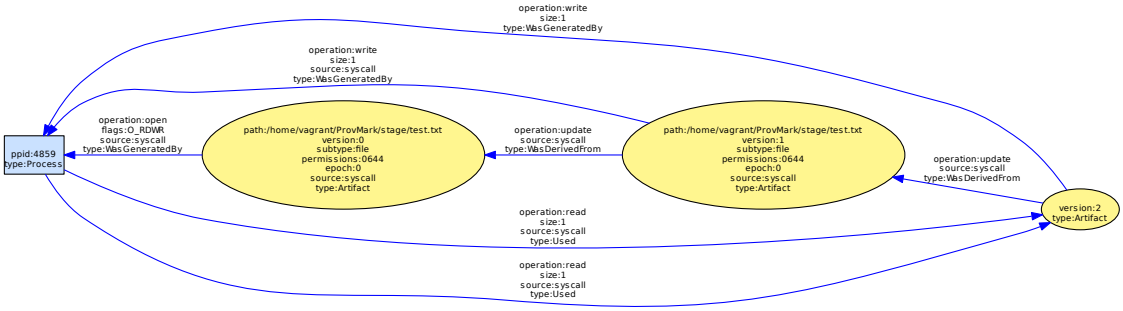


Figure 6.1: Provenance benchmarks for Code Snippet 6.3 (Path #1)

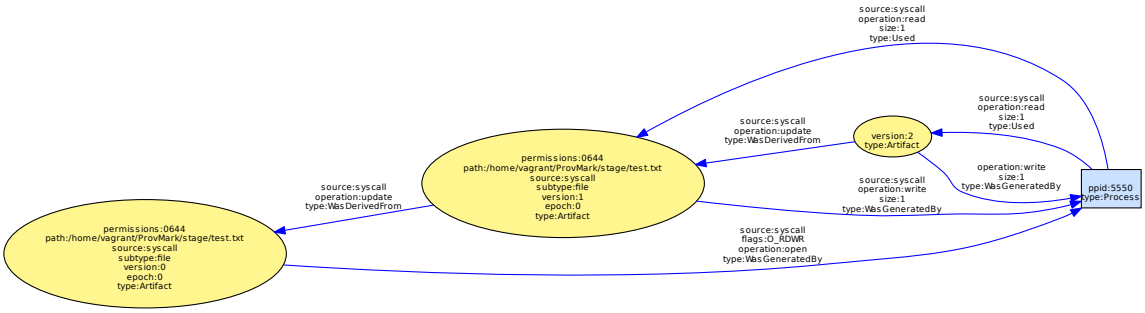


Figure 6.2: Provenance benchmarks for Code Snippet 6.3 (Path #2)

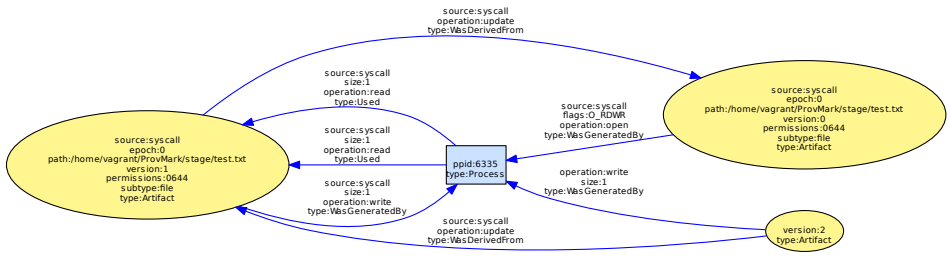


Figure 6.3: Provenance benchmarks for Code Snippet 6.3 (Path #3)

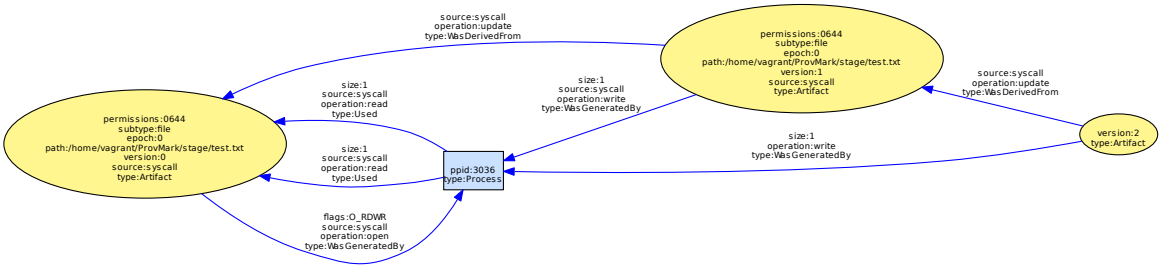


Figure 6.4: Provenance benchmarks for Code Snippet 6.3 (Path #4)

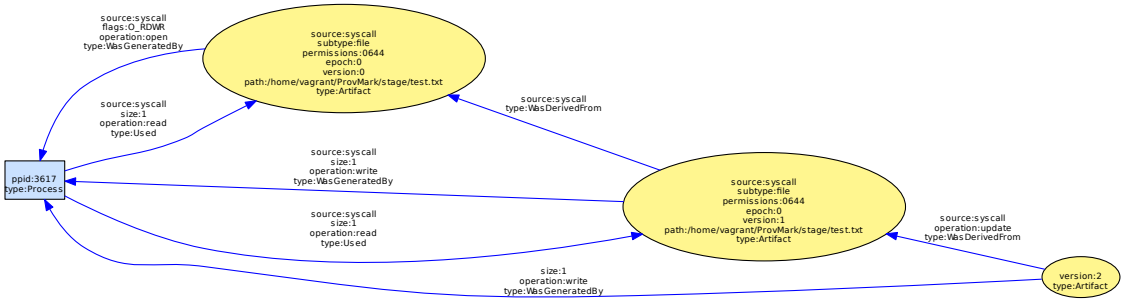


Figure 6.5: Provenance benchmarks for Code Snippet 6.3 (Path #5)

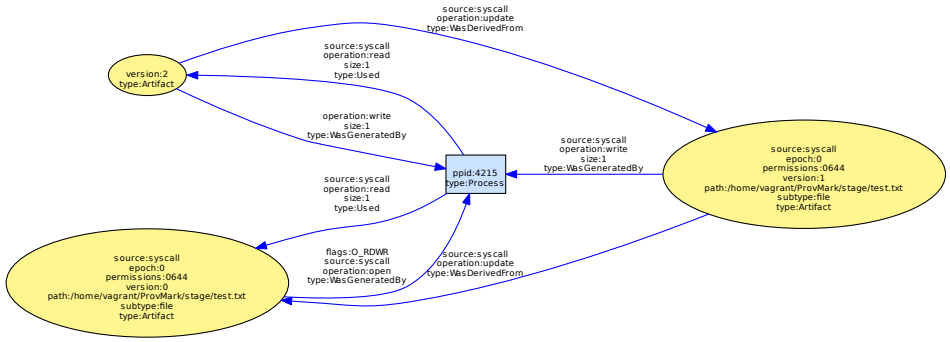


Figure 6.6: Provenance benchmarks for Code Snippet 6.3 (Path #6)

6.5.2 Performance with non-deterministic event

Next, we report measurements of the recording time. This section aims to compare the time performance difference when handling deterministic and non-deterministic input. For the same reason mentioned in the last chapter, we only consider the overhead from *ProvMark*, so we only evaluate the time performance on the transformation subsystem, generalization subsystem and comparison subsystem.

In Figure 6.7–6.9, we summarize the time needed for ProvMark to execute three sample benchmark programs using SPADE, OPUS, and CamFlow respectively. The first two sample benchmark programs contain only system call ***`read`*** and ***`write`*** respectively, which are deterministic input. The third sample benchmark program is the one shown above in Code Snippet 6.3 which is a non-deterministic input. Note that the x-axes are not all to the same scale: in particular, the transformation, generalization and comparison times for OPUS are much higher than for the other tools. It is because of the large overhead for transforming Neo4j graph data structure to Datalog format. We purposely compare these three candidates to show the difference in performance for deterministic and non-deterministic input with a similar set of system calls. The bars in the graph are divided into three portions, representing the time needed for the

three subsystems. Note that the x-axes are not all to the same scale: in particular, the transformation, generalization and comparison times for OPUS are again much higher than the other tools because of the large overhead for transforming Neo4j graph data structure to Datalog format. Once again we are not considering the time for the recording subsystem as it relies too much on the performance and settings for the provenance recording tools themselves.

From the data in Figure 6.7–6.9, we can see that the time needed for the transformation stage is almost the same for *deterministic and non-deterministic input*. As we mentioned above, if we run enough trial executions, most of the combinations should be covered (but are not guaranteed). To maximize the coverage of all combinations, we execute each of the candidates for 15 times, which is more than a double for the possible path in the third non-determinism input. And thus they are only generating 15 provenance graphs as a total for either deterministic or non-deterministic input. Although the generalization target for the non-deterministic input in each group will be less, the total number of graph needed for transformation still equals to the number of trial runs, thus it is expected that the time difference in this subsystem is almost negligible.

The time required for the generalization stage depends mainly on the number of elements (nodes, edges and properties) in the graph since this stage matches elements in pairs of graphs to find an isomorphic pair. For deterministic input, the difficulty and time required for each of the graph comparisons are constant as we are assuming all the candidate graphs are isomorphic to each other (except for the process id and timestamps information to be removed). It is not quite the same for non-deterministic inputs because there are multiple groups of graphs and we only assume the element graphs of each group are isomorphic to each other, there are no guarantees that graphs from different groups will be similar. But this is acceptable because we are only comparing and generalizing graphs in the same group. The hardness and the time required may not be constant because the sizes of the groups may vary. Also, additional time is needed to fingerprint and group the graphs, thus, in general, the time required for the generalization for non-deterministic input will be longer. This difference in the time is more significant if the number of groups is larger.

The time required for the comparison stage is usually less than for generalization in deterministic input because it only needs to compare two graphs (generalized foreground and background graph) in any case. But this is not the case for non-deterministic input. The number of comparisons needed in this last subsystem relates

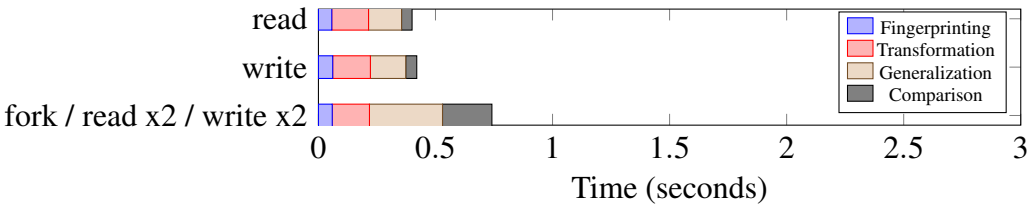


Figure 6.7: Timing results: SPADE+Graphviz

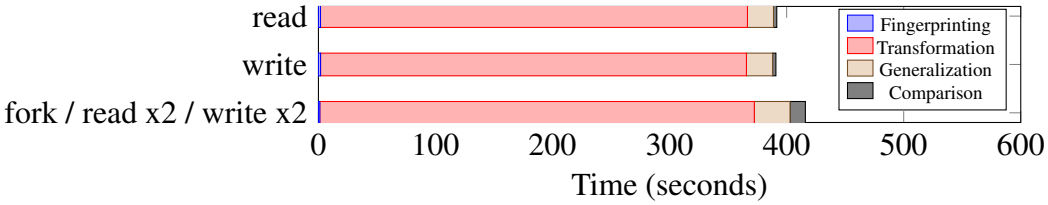


Figure 6.8: Timing results: OPUS+Neo4J

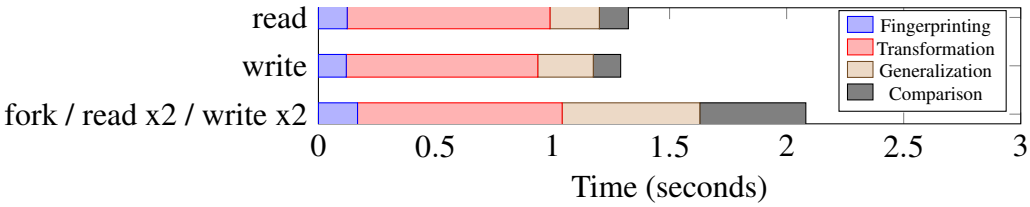


Figure 6.9: Timing results: CamFlow+ProvJson

to the number of possible schedules of the non-deterministic input program. For the example above, we have 6 possible schedules and thus it requires 6 graph comparisons in the comparison subsystem to generate 6 different benchmarks representing all of the possible execution traces of this non-deterministic program. As it is far less than the 15 trial runs we executed on the recording subsystem, we can see that the time needed for this subsystem is shorter than the time needed for the generalization step. But if we are handling non-deterministic programs with many possible schedules, it is expected to see the time required for the comparison subsystem be much longer than the time needed for the generalization subsystems.

In general, we can conclude that the time needed for ProvMark's transformation, generalization, and comparison stages is acceptable compared with the time needed for recording. The extra overhead for handling non-deterministic input programs is still in an acceptable range. But the time needed depends on the number of possible schedules of the non-deterministic program itself. The time overhead is exponentially proportional to the number of possible combinations since we also need to increase

the number of trial executions to have high confidence that all possible combinations are covered. As mentioned in Section 2.6, some researchers propose some symbolic execution method for finding all possible execution paths and the minimum threshold for covering all schedules. Besides, we have once again monitored the memory usage and found that *ProvMark* keeps its memory usage less than 75% on a 4GB virtual machine, indicating once again memory was not a bottleneck even when handling non-deterministic events.

6.5.3 Non-deterministic schedules coverage

One of the important consideration for the provenance benchmark generation of non-deterministic input is the coverage of possible schedules. In general, if there are N different schedules exists for a program binaries, it is not likely that all N schedules will be covered by just executing N trials. It generally takes more execution trials to cover all the schedules. We have done some basic evaluation of the coverage using the program described in Code Snippet 6.3. As mentioned in Table 6.1, there are six possible schedules for this program. We defined twenty test cases, each of the test cases has different numbers of trial executions range from one trial to twenty trials. Each of the test cases is executed for 10 times and result in 10 numbers of coverage which are used to calculate the average coverage of schedules for each test cases. The result is shown in Figure 6.10.

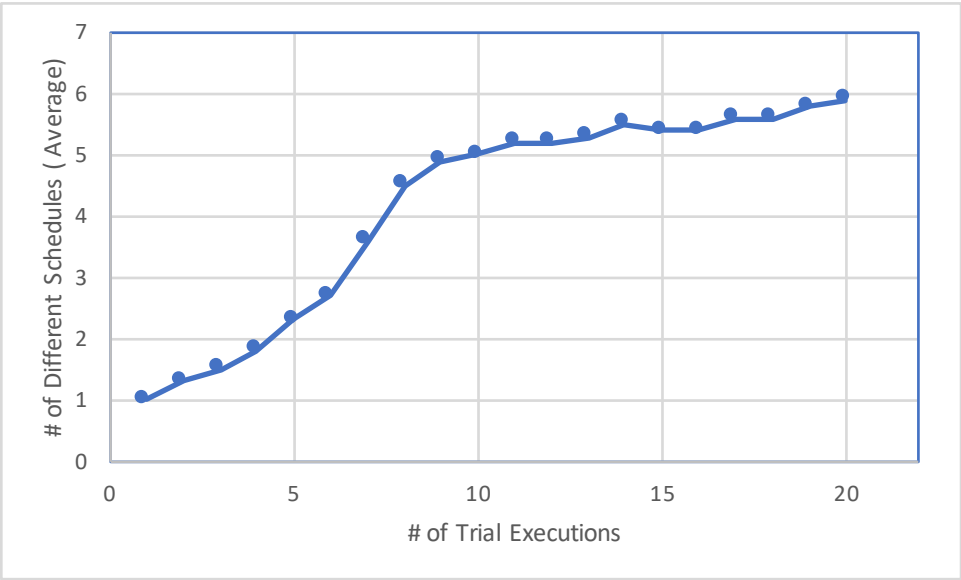


Figure 6.10: Average schedule covered for executing Code Snippet 6.3

From the chart shown in Figure 6.10, we can see the average schedules coverage for executing for a different number of trials. The experiment is done by observing the audit log to determine the order of the read/write event. This is possible because the audit daemon will automatically assign event IDs to system calls in execution order, thus observing the event IDs for the read/write event can determine the schedule for this run. Although in the 10 repeat executions of 6 trials can sometimes cover all 6 schedules, it does not succeed in most of the case and makes the average coverage lower than 3. In our experiment, even if we execute 20 trials, we still cannot guarantee that all 6 trials have been covered. During our experiment, we success to cover all 6 schedules in 9 runs of the 10 trial executions test case, but there is 1 run that only cover 5 schedules for the 20 trial executions test case which make it not 100% coverage. Even if the experiment does success covering all of the schedules, it is still not guarantee that in the next execution, all schedules will be covered. We just assume in most of the cases, most schedules should be covered and thus there is a provenance benchmark generated for each of the possible schedules. Repeating the provenance generation process or increasing the number of trials do help to increase the full coverage rate but it has no guarantee for full coverage.

From the experiment above, it is understood that although in the worst-case scenario, executing 20 trials still did not cover all schedules, it is still possible to cover them with fewer trials in general. In average cases from the experiment, executing two times the number of all possible schedules covers all possible schedules and executing three times the number of all possible schedules result in at least two runs for each schedule. This refers to 12 trial executions and 18 trial executions in the experiments. It is important to get at least two runs for each schedule to generalize the graphs for each schedule to filter out noise in the resulting provenance benchmarks. The result above is just intended to show an average experiment. There may be some non-deterministic input which has branches that are taken less often than others. Although on average, these kinds of unbalanced schedules should also be covered, it is never guaranteed. Besides, the above experiment and result are assuming that the number of possible schedules is known before the process. There do exist some cases that the non-deterministic input is not possible to analyse and thus we do not know the number of possible schedules in advance. For example, if we try to process binaries with socket or distributed system calls, we have no idea when will the response from remote components returns. In this case, we have no idea how many possible schedules exist. Also, if the user feeds random binaries to ProvMark for provenance benchmark gener-

ation or expressiveness evaluation, we do not have an idea of the number of possible schedules. In these scenarios, we have no guarantee on full coverage of all schedules, the only way to ensure more schedules are covered is to execute as much trial as we could and at the same time without executing too long and affecting the performance of the whole provenance benchmark generation process. The balance in performance and schedules coverage is considered as one of the future enhancement of ProvMark. Besides, using symbolic execution and other formal methods to ensure full schedule execution is also considered as part of the future enhancement of ProvMark for better handling of non-determinism.

6.5.4 Simple expressiveness evaluation for non-deterministic event

Following the discussion of the simple expressiveness evaluation in Section 5.5.4, we continue to discuss the module in this section. We have already described the design and implementation of the expressiveness evaluation module in Setion 5.5.4, we also present some test and description on how to perform simple expressiveness evaluation on provenance benchmark generated from deterministic events. In this chapter, we discussed and implement an enhancement to *ProvMark* for handling non-deterministic input. The expressiveness evaluation module has also been altered to allow simple expressiveness testing for non-deterministic provenance benchmark. A brief description of the enhanced expressiveness evaluation process is shown in Figure 6.11.

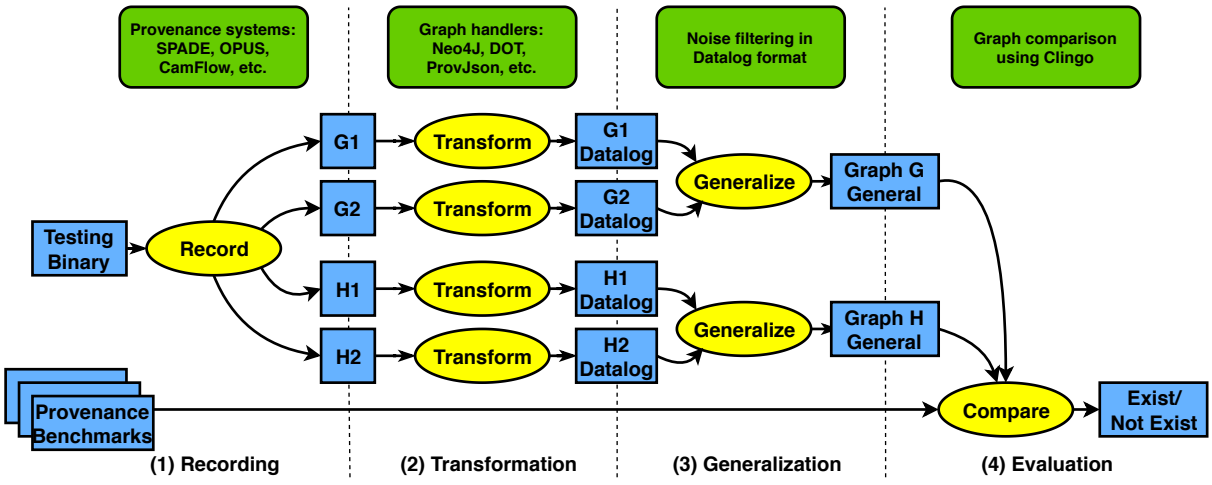


Figure 6.11: Enhanced *ProvMark* expressiveness evaluation overview

As shown in Figure 6.11, the enhanced process is very similar to the original design for determinism. The testing binary once again goes through the recording subsystem first. In this stage, the testing binary will be executed multiple times and monitored by the corresponding provenance systems. The recording subsystem of ProvMark manages both the execution of the testing binary and the recording process of the provenance system. The result of this stage is a set of provenance graphs from multiple execution trials. One difference for non-deterministic input is, those resulting provenance graphs are grouped by fingerprints which follows what we mentioned above. These graphs will then passed through the transformation subsystem and generalization subsystem. The result will be multiple generalized graphs, one for each of the groups. This process still works even if the provided testing binary is deterministic which will only have one group of results (one generalized graph). Basically, similar to the original design of the module, the testing binary goes through the same process as the non-determinism provenance benchmark generation process for the first three subsystems. Then all the generalized provenance graph in Datalog format will pass to the same *evaluation subsystem* for the judgment. Also, the enhanced process will not only take one provenance benchmark as input, but it will also take a directory which contains all the provenance benchmarks generated. At the evaluation subsystem, instead of judging by a one to one mapping and comparison, it will repeat multiple times by cross comparing each of the generalized provenance graphs of the testing binary and the set of provenance benchmarks. If only one part of the provenance benchmark identifies the same action sequence in one of the generalized graphs, we can judge that the testing binary does contain activity sequence that is matching some of the paths of a known non-deterministic activity sequence. Currently, this is the only judgment possible for the evaluation module in handling non-deterministic input. We cannot determine if the testing binary has the same non-deterministic activity sequence described by the provenance benchmark. The major reason for this uncertainty is because we currently have no guarantee that all path in a non-deterministic input will be executed accordingly. Thus we cannot prove that if the generalized graph set from the testing binary or the provenance benchmark itself already covers all possibilities. This is one of the shorthand for the simple expressiveness evaluation module on handling non-deterministic input. The enhancement for the module remains a future work after we could guarantee that all path in a non-deterministic input is covered.

We use a similar approach to test the simple expressiveness evaluation module mentioned in Section 5.5.4. The only difference is we try to put in random system calls

inside non-deterministic branches to allow us easily distinguish the possible difference. For example, in a program with a fork command, we put in a ***open*** system call in the parent process and a ***rename*** in the child process. For test case EV#1 and EV#2, it is the same as deterministic input, only that the system call instead of the process generation is repeating for EV#2. For the remaining test cases, it is altered with the same logic, that is altering the content within a process without touching the outermost process forking that introduces non-determinism.

Once again we try to execute different combinations of system calls inside a multiple process environment. The only difference is that we provide multiple benchmarks for each of the non-deterministic schedules. As mentioned above, we can consider the different non-deterministic schedules as deterministic path separately. A set of provenance benchmark for a non-deterministic input only generates multiple sets of provenance benchmark, each of them representing a schedule and the action sequence for the same schedule is deterministic and remain the same across different trials. For this reason, it does not matter whether the testing binary is deterministic or not. The only thing needs to do is compare the benchmark to the provenance graph of the testing binary one by one and determine if the testing binary match at least one benchmark. If the testing binary is a deterministic input and a match is found, it means that the action sequence in testing binary match one of the non-deterministic schedules. If the testing binary is a non-deterministic input and at least one matches are found, it means that at least one schedules of the testing binaries match the non-deterministic schedule. Both cases provide a positive judgment for the existence of the target activity sequences. For this reason, the expressiveness evaluation module for non-deterministic benchmark set is no different when comparing to deterministic benchmark set except the numbers of provenance benchmark used. The most important point is the above conclusion is correct only when all schedules have been covered by the provenance benchmark set, which we assumed.

6.6 Discussion

At the end of Chapter 5, we mentioned the enhancements needed to make *ProvMark* more realistic to be used. These enhancements include handling of non-determinism which is a challenge because most of the non-deterministic source may produce different provenance graph with different combinations of nodes, edges and properties or similar elements with different orders. In this chapter, we discuss our work on the enhancement of *ProvMark* on non-determinism.

One major difference between deterministic input and non-deterministic input is uncertainty. In most of the non-deterministic program, we can still deduce multiple static paths of execution. Each path of execution will only generate one provenance graph if the same path is executed multiple times. The uncertainty in non-determinism is that we can never predict which schedule will be executed for each trial run. With this understanding, we give up on the prediction and instead we generate a provenance benchmark for each of the observed schedule. We group them by matching their fingerprints that are generated by hashing their execution schedules from Ftrace which should be static for each group. Thus, the problem of non-determinism has been simplified to multiple deterministic problems after we group different provenance graphs according to the actual schedules. As a result, we treat a non-deterministic input as multiple deterministic inputs that generated from the same non-deterministic input source. Thus the final result contains multiple provenance benchmarks, each of them representing a schedule of the non-deterministic input.

With this understanding, the enhancement only needs to extract the schedule of each provenance recording stage. Then *ProvMark* will use the schedule to fingerprint and group the provenance graphs together. When comparing the steps for handling deterministic and non-deterministic input, the only extra steps done are the additional fingerprinting actions and the additional generalization and comparison for each of the combinations. These additional steps in handling non-deterministic input allow us to ensure enough samples of schedules are collected from the trials. This is important when we need to ensure the completeness of the provenance benchmark for covering all possible schedules for a specific non-deterministic target action sequence. The overhead for non-deterministic inputs is mostly determined by the number of possible combinations of the non-deterministic source itself. This has been illustrated in the above performance evaluation and comparison of the determinism and non-determinism.

Although it seems easy at first sight to cover all possible execution path schedules,

it is not as easy as it seems to be. There are two challenges exist in the handling of non-determinism and we have put them into future enhancement works. One of the challenges is the coverage for all possible execution path schedules. Although on average all possible path schedules are covered if we repeat the execution for enough times, we never have a guarantee about the full coverage as the execution is non-deterministic and there could exist some paths that are never executed during the limited execution trials. No matter how high the chance of full coverage, there still some probability that some paths are not covered. Another challenge is the number of possible path schedules may not be known during execution thus we have no idea if the result has already covered all paths. These two challenges are currently not handled by ProvMark and are considered as future work enhancement. One possible way to handle these problems is to make use of symbolic execution to assist in executing all possible paths to ensure full coverage of path schedules.

In reality, the handling of non-determinism is one of the biggest obstacles to using ProvMark for security analysis and intrusion detection. Although our work does not guarantee to cover all non-deterministic paths for a random execution, it does help to generate benchmark patterns for identifying most of the possible paths of execution. This can help to understand and discover possible paths of execution and analyse them to understand what activity sequence has been executed in a runtime section. This work provides an alternative for identifying malicious behaviour in certain execution and provides ways.

6.7 Conclusion

In this chapter, we discuss the enhancement to *ProvMark*. This enhancement aims to extend the functionality of *ProvMark* to handle more realistic situations by capturing the action sequence behaviour for non-deterministic sources. We also provided some demonstration of results for handling non-deterministic input and compare the performance of *ProvMark* between deterministic and non-deterministic input. The provenance recording process is still executed as a black box. This enhancement completes one of the extensions of *ProvMark* proposed in Chapter 5.

Chapter 7

Future extension and work

This chapter discusses possible future work extending ProvMark and also identifies some future work for applying ProvMark.

7.1 ProvMark overview

7.1.1 Current features

In general, the basic features for ProvMark are complete, all the source code is stored in an open-sourced GitHub repository. ProvMark has four subsystems to handle different part of the benchmarking process. We currently have three modules for the first subsystem which support SPADE, OPUS and CamFlow. We have three modules for the second subsystem to transfer Neo4J, Graphviz graph, and Prov-JSON data into Datalog format. Our whole work is tested and evaluated by these current modules. Additional modules can be developed by modifying the given templates.

Using Answer Set Programming to solve our case of the graph isomorphism problem is one of the contributions of our work. We need to handle two different cases of the graph isomorphism problem in the provenance benchmark generation process. These two cases are slightly modified versions of (sub)graph isomorphism matching. Currently, there are not many graph comparison algorithms proposed for graphs with large numbers of property values. For this reason, we contribute to the literature by applying existing edit distance algorithms and making use of an Answer Set Programming solver to match small provenance graphs optimally. We have not yet expanded the experiments to larger graphs, but our current usage of edit distance and an Answer Set Programming solver is already a novel contribution to the graph communities.

One of the main applications of ProvMark is providing ways for end-users to compare the expressiveness of both the provenance tools and the generated provenance information. We currently do not have support for comparing the benchmarks. The only feature we have provided on top of the resulting benchmark is a simple expressiveness evaluation of the benchmark, which we discussed in both Section 5.5.4 and Section 6.5.4. This feature allows users to provide the resulting benchmark back to the system with another fresh binary and ProvMark will identify the existence of the actions mentioned by the benchmark in that binary. This is a basic version for behavioural analysis and intrusion detection approach that motivates this whole work. Other usage and applications are discussed in Section 5.3.4.

7.1.2 Limitations

The current version of ProvMark only supports three provenance systems. Although it should be easy to extend to more tools, the progress is not included in this work. Besides, the resulting benchmarks are raw in shape. It may be hard for users with no system experience to understand some of the components included in the resulting benchmarking. Also, although the development of ProvMark allows it to be automated in most cases, it still requires some settings and correct preparation of input to get results. There are already some batch tools implemented to decrease manual configuration, but there is still some level of manual operations needed for ProvMark.

Besides, the full coverage for all non-determinism execution path schedules is not guaranteed. In the current version of ProvMark, no matter how many trials of execution has been included in the recording subsystem, there is no guarantee that all execution path schedules have been covered because the execution is random. Also, as ProvMark is working as a black box, there is sometimes not possible to determine how many possible path schedules exist for a non-deterministic input source. Thus it is no way to guarantee that all possible path schedules have been covered. This result in possible non-complete coverage of provenance benchmark for all possible activity sequences exists in a binary. The insurance of complete path schedules coverage is proposed as a future work which could be implementation and application of symbolic execution approach.

7.2 Enhancement

7.2.1 Current features extension

In chapter 3-5, the basic features of the expressiveness benchmarking have been reported. The text includes the preliminary trial of the manual benchmarking approach, the obstacle of the (sub)graph isomorphism problems that need to be overcome, and the system design and implementation of the automated system, ProvMark. Following the discussion in Chapter 5, the basic features aim to solve the easy problems for demonstrating the feasibility of such an approach. We suggest several enhancements at the end of that chapter where some of them are already implemented. These enhancements include the handling of non-deterministic events mentioned in Chapter 6, which makes ProvMark more realistic. The main reason is most of the executions and operations nowadays include a certain level of concurrency and non-determinism.

Another enhancement is the simple expressiveness evaluation of those provenance benchmark generated by ProvMark. This has been mentioned in both Section 5.5.4 and Section 6.5.4. The major work done is providing a simple evaluation for the ProvMark operation on both deterministic and non-deterministic source. In this evaluation, we aim to provide the user with a certain level of automation for understanding the expressiveness of those generated benchmark and the systems behind them. By feeding the generated provenance benchmark patterns of a specific activity sequence with some random application or binaries that does or does not contain that activity sequences, ProvMark will try to use the benchmark to match the activities in the binaries. The result can show the correctness and completeness of the generated provenance benchmark. An acceptable level of correctness and completeness should make ProvMark possible to identify the existence of those activities correctly. This can test both the capabilities of ProvMark and help end-users to identify the expressiveness of the provenance systems which are responsible for generating that benchmark. In general, the enhancement provides consolidation for a fully automated evaluation of ProvMark and the generated benchmark and act as a base for further applications and usages we mentioned in Section 5.3.4.

7.2.2 Future Work

One possible enhancement is increasing automation of ProvMark. Currently, the design of ProvMark still requires a certain level of manual input and configuration. Also,

it requires work to make use of the provenance benchmark result at last. Although there are already a simple evaluation and automatic testing of the existence of certain activity sequences, there are still many locations that required manual effort. The disadvantages of manual effort include the need for the user to understand the tools, inputs, and configuration to use it. It may be obvious for tool developers but complicated for end-users. Our tool aims to provide an easy comparison of the capabilities of provenance systems on different uses, these comparisons are done by judging the generated provenance benchmarks. The problem is that some of the users do not have the skillset for understanding and comparing kernel events and activities that exist in the benchmark. Thus one possible future direction is to extend ProvMark to additional use cases by abstracting the input and result. ProvMark can provide a user interface allowing users to choose their desired applications and configurations they want. As a result, ProvMark will automatically choose suitable training data set and provenance system to generate benchmarks, then it will also compare benchmarks according to pre-defined criteria and rank the provenance systems for the user to identify which tools are more capable for their specific applications. This enhancement of ProvMark lowers the entrance requirement for using ProvMark, which makes it more useful for all groups of end-users. Work on this enhancement could include collecting the specifications and requirements for different uses as result analysis criteria. Also, it may include collecting suitable training data for benchmark generation.

Another possible enhancement is improving support for the tool developers. As suggested, one of the expected applications for ProvMark is the support of regression testing and debugging for those provenance systems developers. By viewing the benchmark result, the developer can identify some unexpected behaviour and trace back to the place of bugs or errors. Sometimes the developer may try to check if a code modification affects other components accidentally. It may be time-consuming for checking all of the benchmark results because the resulting set of benchmarks may cover a large set of different activities sequences. For this reason, ProvMark can be enhanced by storing the benchmark results for full testing of the tools as a snapshot. Then next time when the developers require another full testing, they can simply choose the options for regression testing. ProvMark will then rerun the full testing set and generate a new set of provenance benchmark results. The new set of benchmark results will be automatically compared with the stored snapshot from the last execution and identify any differences in the two benchmark sets. The developers can then view the differences and determine whether there are unexpected modifications of the behaviour of

their tools. This additional regression testing option is a possible future work that allows easy testing for provenance systems developers and could make ProvMark more automatic and useful.

Lastly, one of the improvement and enhancement lies in the insurance of complete path schedules coverage of non-deterministic source input. In the current settings of ProvMark, there is no way to guarantee that all possible execution path schedules are covered by the execution trials and thus the generation provenance benchmarks set is not guaranteed to identify all possible activity sequences of a non-deterministic input. The enhancement of this direction aims to look in method, including symbolic execution approach to discover both the number of possible execution path schedules exist in a binary and to ensure that all possible path schedules have been executed for at least twice within the limited number of trial executions. The two goals make it possible for ProvMark to generate provenance benchmarks that cover all possible activity sequences exist in different execution path schedules from a non-deterministic source.

The above are possible suggestions for future enhancements of ProvMark to aid the provenance research communities. They allow users to choose a suitable provenance system. It also allows developers to fix and enhance their provenance systems with fewer errors. For conclusion, these enhancements make ProvMark more automatic and user-friendly by abstracting certain layers and provide user interfaces for users and developers for those steps that currently require manual efforts.

Chapter 8

Conclusion

This thesis surveyed the state of art for data provenance and provenance systems and analysed the capabilities of provenance systems in different settings. We analysed provenance systems and their collection of data provenance in terms of graphs. These data provenance graphs help users to understand how the current state of components was attained. Besides, they also answer the question of who is responsible for each of the changes and where did it go through. These applications of data provenance make it useful in many situations that require understanding and analysing what is happening in the system for certain executions.

Our work aims to provide a unified way of comparing the expressiveness of provenance systems and the data provenance generated by them. The expressiveness is measured based on the completeness and correctness of the resulting provenance information. In other words, we aim to provide an analysis of the quality of the data provenance to see if they provide sufficient information to identify activities and to distinguish two different activities. There are many applications for data provenance and each of them requires different levels of information, thus the same set of provenance benchmark may have different expressiveness measurement for different applications. Our work provides a mean of comparing the data provenance and its source provenance system.

From the study, we see that the provenance benchmark results of the same system calls are described differently by different provenance tools. It shows the different perspectives of the provenance tools on viewing the same activities. Some of the tools provide a more complete result describing the event and this allows greater usability in tracing the source and accountable parties for the action. Other tools provide more details about the version changes of an artefact which allows tracing the modification his-

tory of an artefact to search for problematic moves. The provenance benchmarks help the user to identify the usability of the provenance systems by showing their strength in their provenance representation. Also, they demonstrate the completeness of details in the provenance for specific needs. In general, by analysing and comparing the provenance benchmarks of provenance systems, the compatibility of the systems on different identifiable applications can be understood. This is one of the preliminary motivation and contribution by our automatic expressiveness benchmarking approach.

Our work contributes to the literature by providing a fully automated system, ProvMark, that generates expressiveness benchmarks for provenance systems. The user can treat ProvMark as a black box. They just need to provide source code with certain annotations showing the target actions, together with the configurations to ProvMark. ProvMark will then generate a set of benchmarks for the specific activities using the chosen provenance systems. If additional provenance systems need to be benchmarked which are not supported by ProvMark, the tool developers can modify the templates and insert them into ProvMark as modules. This helps to make ProvMark more flexible in supporting benchmarking for more provenance systems.

One important user group for our benchmarking is the developers of provenance systems. As we understand, developing provenance systems is hard work. It is even harder to verify if the system works correctly in all situations. This is because in most cases, the data sets and graphs handled by a provenance system are large and it is impossible to check the details one by one. Also, if a developer wants to have a fast check after an update to confirm the changes did not affect other parts of the system, they may need to check for many possibilities and it is inefficient. ProvMark can help to generate benchmarks for their systems to check if they function as expected because the benchmark only contains the key activities without background and unrelated information. Also, the benchmark results can be stored and compared with the new benchmark results when updates to the provenance system are done to check for changes. This is a form of providing regression testing functionality for provenance systems.

Another important user group is the users of those provenance systems. Sometimes, it is daunting to see many choices of provenance systems that claim they can solve a variety of problems. As a result, it is hard to choose which one is most suitable for a certain application. It is inefficient to try out the tools one by one and ProvMark can provide a simpler overview of the quality of the provenance generation process for specific applications. By comparing the expressiveness of those benchmarks generated by ProvMark, the users can have a more intuitive understanding of the capabilities of

the tools for any specific purpose.

Lastly, an important motivation for ProvMark is to study the behaviour patterns for sensitive or malicious actions. Using ProvMark to generate benchmarks of specific malicious activity sequences can help to understand the key patterns for specific malicious actions. As most of the unrelated operations are removed in the resulting benchmark, it helps us to eliminate some of the obfuscation in those malicious activities and provide a key pattern for what this activity looks like. This helps to vet the existence of this activity if the pattern is discovered once again in later execution. Also, the characteristics of data provenance allow tracing back to the origin of a piece of data, which provides a complete chain of evidence for proving certain actions and activities that are being executed at runtime. Besides, the accountable parties and components are also captured through the process and will also exist in the final benchmark. This provides an alternative way for behavioural analysis and intrusion detection based on patterns. Also, it provides ways to identify accountable parties for malicious or sensitive actions. In the current implementation of ProvMark, we provide the basis for the expressiveness evaluation. It can currently identify if the certain activity sequences exist in a random testing binary by comparing the provenance benchmark to the provenance graph of the testing binary directly. The current feature is limited to identify the same sequence without obfuscation and randomization. Future enhancement of ProvMark aims to provide more automation and tracing support to the evaluation

In conclusion, this work provides a fully automated system for benchmarking the expressiveness of provenance systems and their generated data provenance, which contributes to research in the field of data provenance and security analysis. It also allows general end-users to take advantage of the characteristics of data provenance by choosing a suitable provenance system for their uses.

Bibliography

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer, 2013.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] Z. Abu-Aisheh, B. Gaüzere, S. Bougleux, J.-Y. Ramel, L. Brun, R. Raveaux, P. Héroux, and S. Adam. Graph edit distance contest: Results and future challenges. *Pattern Recognition Letters*, 100:96 – 103, 2017.

[4] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. In P. Degano and J. D. Guttman, editors, *Principles of Security and Trust*, pages 410–429. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] M. Ali, S. U. Khan, and A. V. Vasilakos. Security in cloud computing: Opportunities and challenges. *Information Sciences*, 305:357–383, 2015.

[6] P. Alper, K. Belhajjame, and C. A. Goble. Automatic vs manual provenance abstractions: Mind the gap. In *Proceedings of the 8th USENIX Conference on Theory and Practice of Provenance*, TaPP'16, pages 34–39, Berkeley, CA, USA, 2016. USENIX Association.

[7] D. Alrajeh, L. Pasquale, and B. Nuseibeh. On evidence preservation requirements for forensic-ready systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 559–569, New York, NY, USA, 2017. ACM.

[8] P. Alvaro and S. Tymon. Abstracting the geniuses away from failure testing. *Queue*, 15(5):10:29–10:53, Oct. 2017.

[9] M. K. Anand, S. Bowers, T. Mcphillips, and B. Ludäscher. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *Scientific and Statistical Database Management*, pages 237–254. Springer, 2009.

[10] P. Anderson and J. Cheney. Toward provenance-based security for configuration languages. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*, TaPP'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[11] A. Armando, G. Costa, and A. Merlo. Formal modeling and reasoning about the android security framework. In C. Palamidessi and M. D. Ryan, editors, *Trustworthy Global Computing*, pages 64–81, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] M. Armoni and J. Gal-Ezer. Non-determinism: An abstract concept in computer science studies. *Computer Science Education*, 17(4):243–262, 2007.

[13] V. Arvind and J. Torán. Isomorphism testing: Perspectives and open problems. *Bulletin of the European Association for Theoretical Computer Science*, 86:66–84, 2005.

[14] L. Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697, 2016.

[15] M. Backes, S. Bugiel, and S. Gerling. Scippa: system-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 36–45. ACM, 2014.

[16] N. Balakrishnan, T. Bytheway, L. Carata, R. Sohan, and A. Hopper. Towards secure user-space provenance capture. In *8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 16)*, Washington, D.C., 2016. USENIX Association.

[17] N. Balakrishnan, T. Bytheway, R. Sohan, and A. Hopper. OPUS: A lightweight system for observational provenance in user space. In *TaPP 2013*, 2013.

[18] A. Bates, B. Mood, M. Valafar, and K. Butler. Towards secure provenance-based access control in cloud environments. In *Proceedings of the Third ACM*

Conference on Data and Application Security and Privacy, CODASPY '13, pages 277–284, New York, NY, USA, 2013. ACM.

[19] A. Bates, D. J. Tian, G. Hernandez, T. Moyer, K. R. B. Butler, and T. Jaeger. Taming the costs of trustworthy provenance through policy reduction. *ACM Trans. Internet Technol.*, 17(4), Sept. 2017.

[20] A. M. Bates, D. Tian, K. R. B. Butler, and T. Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security 2015*, pages 319–334, 2015.

[21] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 203–208. AAAI Press, 1997.

[22] K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker, S. Miles, J. Myers, S. Sahoo, and C. Tilmes. PROV-DM: The PROV data model. Technical report, World Wide Web Consortium, 2012.

[23] Y. Ben, Y. Han, N. Cai, W. An, and Z. Xu. T-tracker: Compressing system audit log by taint tracking. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–9, 2018.

[24] G. Berrada and J. Cheney. Aggregating unsupervised provenance anomaly detectors. In *11th International Workshop on Theory and Practice of Provenance (TaPP 2019)*, Philadelphia, PA, June 2019. USENIX Association.

[25] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer. Issues in automatic provenance collection. *IPAW*, 6:171–183, 2006.

[26] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[27] P. Buneman, S. Khanna, and W.-C. Tan. Data provenance: Some basic issues. In *FST TCS 2000: Foundations of software technology and theoretical computer science*, pages 87–93. Springer, 2000.

[28] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8):689–694, 1997.

[29] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3):255–259, 1998.

[30] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[31] Y. Cai, G. Cong, X. Jia, H. Liu, J. He, J. Lu, and X. Du. Efficient algorithm for computing link-based similarity in real world networks. In *2009 Ninth IEEE International Conference on Data Mining*, pages 734–739, Dec 2009.

[32] A. Chaudhuri. Language-based security on android. In *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security*, pages 1–7. ACM, 2009.

[33] A. Chavan, S. Huang, A. Deshpande, A. J. Elmore, S. Madden, and A. Parameswaran. Towards a unified query language for provenance and versioning. In *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*, TaPP'15, pages 5–5, Berkeley, CA, USA, 2015. USENIX Association.

[34] J. Cheney. A formal framework for provenance security. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 281–293. IEEE, 2011.

[35] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: a future history. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 957–964. ACM, 2009.

[36] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, 2007.

[37] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[38] S. Cohen, S. Cohen-Boulakia, and S. Davidson. Towards a model of provenance and user views in scientific workflows. In *Data Integration in the Life Sciences*, pages 264–279. Springer, 2006.

[39] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[40] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[41] R. Corin and F. A. Manzano. Taint analysis of security code in the klee symbolic execution engine. In T. W. Chim and T. H. Yuen, editors, *Information and Communications Security*, pages 264–275, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[42] F. Costa, V. Silva, D. de Oliveira, K. Ocaña, E. Ogasawara, J. Dias, and M. Mattoso. Capturing and querying workflow runtime provenance with prov: A practical approach. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 282–289, New York, NY, USA, 2013. ACM.

[43] M. Crosas. The dataverse network: An open-source application for sharing, discovering and preserving data. *D-Lib Magazine*, Volume 17, 2011.

[44] D. Deutch, A. Gilad, and Y. Moskovitch. Selective provenance for datalog programs using top-k queries. *Proc. VLDB Endow.*, 8(12):1394–1405, Aug. 2015.

[45] G. H. Dietmar. On the randomized complexity of monotone graph properties. *Acta Cybernetica*, 10(3):119–127, 1992.

[46] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, page 24, 2011.

[47] J. Edge. A look at ftrace. *LWN-Linux Weekly News-online*, 2009.

[48] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, Mar. 2008.

[49] M. Factor, E. Henis, D. Naor, S. Rabinovici-Cohen, P. Reshef, S. Ronen, G. Michetti, and M. Guercio. Authenticity and provenance in long term digital preservation: Modeling and implementation in preservation aware storage. In *First Workshop on on Theory and Practice of Provenance*, TAPP'09, pages 6:1–6:10, Berkeley, CA, USA, 2009. USENIX Association.

[50] M.-L. Fernández and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 22(6):753 – 758, 2001.

[51] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, pages 201–203, Dec 2010.

[52] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE, 1996.

[53] D. Fourches and A. Tropsha. Using graph indices for the analysis and comparison of chemical datasets. *Molecular Informatics*, 32(9-10):827–842, 2013.

[54] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[55] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53, 2006.

[56] A. Gamkrelidze, L. Varamashvili, and G. Hotz. New invariants for the graph isomorphism problem. *Journal of Mathematical Sciences*, 218(6):754–761, Nov 2016.

[57] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

[58] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, and T. Schaub. Abstract gringo. *Theory and Practice of Logic Programming*, 15(4-5):449–463, 2015.

[59] M. Gebser, R. Kaminski, B. Kaufmann, P. Lühne, P. Obermeier, M. Ostrowski, J. Romero, T. Schaub, S. Schellhorn, and P. Wanko. The potsdam answer set solving collection 5.0. *KI-Künstliche Intelligenz*, 32(2-3):181–182, 2018.

[60] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Challenges in Answer Set Solving*, pages 74–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[61] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. T. Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[62] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In C. Baral, G. Brewka, and J. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 260–265, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[63] A. Gehani, H. Kazmi, and H. Irshad. Scaling spade to "big provenance". In *Proceedings of the 8th USENIX Conference on Theory and Practice of Provenance*, pages 26–33. USENIX Association, 2016.

[64] A. Gehani and D. Tariq. Spade: Support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, pages 101–120. Springer-Verlag New York, Inc., 2012.

[65] A. Gehani, D. Tariq, B. Baig, and T. Malik. Policy-based integration of provenance metadata. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 149–152. IEEE, 2011.

[66] E. Goldberg and Y. Novikov. *BerkMin: A Fast and Robust Sat-Solver*, pages 465–478. Springer Netherlands, Dordrecht, 2008.

[67] E. H. d. Graaf, W. A. Kosters, J. N. Kok, and J. Kazius. Visualization and grouping of graph patterns in molecular databases. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 267–280. Springer, 2007.

[68] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[69] S. Grubb. *auditd.conf(5) Linux User's Manual*. Red Hat, April 2016.

[70] S. Grubb. Linux audit, Dec 2017. `http://people.redhat.com/sgrubb/audit/`.

[71] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer. UNICORN: runtime provenance-based detector for advanced persistent threats. In *27th Annual*

Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society, 2020.

[72] X. Han, T. Pasquier, T. Ranjan, M. Goldstein, and M. Seltzer. Frappuccino: Fault-detection through runtime analysis of provenance. In *Workshop on Hot Topics in Cloud Computing (HotCloud'17)*. USENIX, USENIX, 2017.

[73] X. Han, T. Pasquier, and M. Seltzer. Provenance-based intrusion detection: opportunities and challenges. In *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.

[74] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[75] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[76] R. Hay, O. Tripp, and M. Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128. ACM, 2015.

[77] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6):881–906, Dec. 2017.

[78] R. Hoekstra and P. Groth. PROV-O-Viz-understanding the role of activities in provenance. In *International Provenance and Annotation Workshop*, pages 215–220. Springer, 2014.

[79] R. Hoffmann, C. Mccreesh, and C. Reilly. Between subgraph isomorphism and maximum common subgraph. pages 3907–3914, 2017.

[80] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1723–1740, Baltimore, MD, Aug. 2018. USENIX Association.

[81] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216. ACM, 2011.

[82] N. Husted, S. Qureshi, D. Tariq, and A. Gehani. Android provenance: Diagnosing device disorders. In *Proceedings of the 5th USENIX Conference on Theory and Practice of Provenance*, TaPP'13, pages 4–4, Berkeley, CA, USA, 2013. USENIX Association.

[83] T. D. Huynh, M. O. Jewell, A. S. Keshavarz, S. T. Michaelides, H. Yang, and L. Moreau. The PROV-JSON serialization. Technical report, World Wide Web Consortium, 2013.

[84] T. D. Huynh and L. Moreau. Provstore: A public provenance repository. In B. Ludäscher and B. Plale, editors, *Provenance and Annotation of Data and Processes*, pages 275–277, Cham, 2015. Springer International Publishing.

[85] T. Isohara, K. Takemori, and A. Kubota. Kernel-based behavior analysis for android malware detection. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 1011–1015. IEEE, 2011.

[86] D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(8):1200–1214, Aug 2006.

[87] R. Kaminski, T. Schaub, and P. Wanko. *A Tutorial on Hybrid Answer Set Solving with clingo*, pages 167–203. Springer International Publishing, Cham, 2017.

[88] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[89] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[90] A. P. Kosoresow and S. A. Hofmeyer. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, Sep. 1997.

[91] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[92] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, volume 6, pages 133–144. VLDB Endowment, 2012.

[93] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[94] K. H. Lee, X. Zhang, and D. Xu. Loggc: Garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 1005–1016, New York, NY, USA, 2013. Association for Computing Machinery.

[95] V. Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, page 1594–1597. AAAI Press, 2008.

[96] V. Lifschitz. Datalog programs and their stable models. In *International Datalog 2.0 Workshop*, pages 78–87. Springer, 2010.

[97] V. Lifschitz. *Answer set programming*. Springer International Publishing, 2019.

[98] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In E. Yahav, editor, *Static Analysis*, pages 95–111, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[99] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha. Kernel-supported cost-effective audit logging for causality tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 241–254, Boston, MA, July 2018. USENIX Association.

[100] S. Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[101] P. Macko and M. Seltzer. Provenance map orbiter: Interactive exploration of large provenance graphs. In *TaPP*, pages 1–6, 2011.

[102] P. Macko and M. Seltzer. A general-purpose provenance library. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*, TaPP'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[103] E. Manzoor, S. M. Milajerdi, and L. Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1035–1044, New York, NY, USA, 2016. Association for Computing Machinery.

[104] J. A. P. Marpaung, M. Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 744–749, Feb 2012.

[105] B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[106] J. Mendivelso and Y. Pinzón. A new approach to isomorphism in attributed graphs. In *2014 9th Computing Colombian Conference (9CCC)*, pages 231–239. IEEE, 2014.

[107] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan. Holmes: Real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152, 2019.

[108] A. Mohaisen, O. Alrawi, and M. Mohaisen. Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251–266, 2015.

[109] A. Mohindra, A. Purakayastha, and P. Thati. Exploiting non-determinism for reliability of mobile agent systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 144–153, June 2000.

[110] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241, 2010.

[111] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, et al. The open provenance model core specification (v1. 1). *Future Generation Computer Systems*, 27(6):743–756, 2011.

[112] L. Moreau, J. Freire, J. Futrelle, R. E. Mcgrath, J. Myers, and P. Paulson. The open provenance model: An overview. In *International Provenance and Annotation Workshop*, pages 323–326. Springer, 2008.

[113] L. Moreau, P. Groth, S. Miles, J. Vazquez-Salceda, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, et al. The provenance of electronic data. *Communications of the ACM*, 51(4):52–58, 2008.

[114] B. Morisson. Analysis of the linux audit system. Master's thesis, Information Security Group, Royal Holloway, University of London, 2014.

[115] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[116] K.-K. Muniswamy-Reddy and D. A. Holland. Causality-based versioning. *Trans. Storage*, 5(4):13:1–13:28, Dec. 2009.

[117] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference*, pages 43–56, 2006.

[118] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting execution context for the detection of anomalous system calls. In C. Kruegel, R. Lippmann, and A. Clark, editors, *Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[119] S. Nari and A. A. Ghorbani. Automated malware classification based on network behavior. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 642–647, Jan 2013.

[120] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 252–269, New York, NY, USA, 2017. ACM.

[121] Neo4j. The Neo4J graph platform, Feb 2018. `https://neo4j.com/`.

[122] M. Neuhaus and H. Bunke. A probabilistic approach to learning costs for graph edit distance. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, volume 3, pages 389–393 Vol.3, Aug 2004.

[123] M. Neuhaus and H. Bunke. Self-organizing maps for learning the edit costs in graph matching. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 35(3):503–514, June 2005.

[124] M. Neuhaus and H. Bunke. A convolution edit kernel for error-tolerant graph matching. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 4, pages 220–223, Aug 2006.

[125] P. Okech, N. Mc Guire, C. Fetzer, and W. Okelo-Odongo. Investigating execution path non-determinism in the linux kernel. In *Proc. 14th Real-Time Linux Workshop, Lugano. OSADL*, 2013.

[126] D. D. Oliveira, V. Silva, and M. Mattoso. How much domain data should be in provenance databases? In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*, Edinburgh, Scotland, 2015. USENIX Association.

[127] J. Palat. Introducing vagrant. *Linux J.*, 2012(220), Aug. 2012.

[128] L. Pasquale, D. Alrajeh, C. Peersman, T. T. Tun, B. Nuseibeh, and A. Rashid. Towards forensic-ready software systems. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 9–12, 2018.

[129] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. M. Eyers, M. Seltzer, and J. Bacon. Practical whole-system provenance capture. In *SoCC 2017*, 2017.

[130] T. Pasquier, M. K. Lau, A. Trisovic, E. R. Boose, B. Couturier, M. Crosas, A. M. Ellison, V. Gibson, C. R. Jones, and M. Seltzer. If these data could talk. *Scientific data*, 4, 2017.

[131] T. Pasquier, J. Singh, D. Eyers, and J. Bacon. Camflow: Managed data-sharing for cloud services. *IEEE Transactions on Cloud Computing*, 5(3):472–484, 2015.

[132] T. Pasquier, J. Singh, J. Powles, D. Eyers, M. Seltzer, and J. Bacon. Data provenance to audit compliance with privacy policy in the internet of things. *Personal and Ubiquitous Computing*, 22(2):333–344, 2018.

[133] B. Pérez, J. Rubio, and C. Sáenz-Adán. A systematic review of provenance systems. *Knowledge and Information Systems*, 57(3):495–543, Dec 2018.

[134] D. J. Pohly, S. E. McLaughlin, P. D. McDaniel, and K. R. B. Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *ACSAC 2012*, pages 259–268, 2012.

[135] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, Mar 2006.

[136] S. Ram and J. Liu. Understanding the semantics of data provenance to support active conceptual modeling. In P. P. Chen and L. Y. Wong, editors, *Active Conceptual Modeling of Learning*, pages 17–29, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[137] M. Ramane, B. Vasudevan, and S. Allaphan. A provenance-policy based access control model for data usage validation in cloud. *International Journal of Security Privacy and Trust Management*, 3(5), 2014.

[138] B. Rao and A. Mitra. A new approach for detection of common communities in a social network using graph mining techniques. In *2014 International Conference on High Performance Computing and Applications (ICHPCA)*, pages 1–6, Dec 2014.

[139] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, Jul 2002.

[140] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[141] K. Riesen. Structural pattern recognition with graph edit distance. *Advances in computer vision and pattern recognition, Cham*, 2015.

[142] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959, 2009. 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007).

[143] S. Rostedt. Debugging the kernel using ftrace. *LWN. net*, 2009.

[144] S. Rostedt. Ftrace linux kernel tracing. In *Linux Conference Japan*, 2010.

[145] L. Ryan. *Efficient algorithms for clause-learning SAT solvers*. PhD thesis, Theses (School of Computing Science)/Simon Fraser University, 2004.

[146] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

[147] I. A. Saeed, A. Selamat, and A. M. Abuagoub. A survey on malware and malware detection systems. *International Journal of Computer Applications*, 67(16), 2013.

[148] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[149] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A formal model to analyze the permission authorization and enforcement in the android framework. In *IEEE Second International Conference on Social Computing (SocialCom)*, pages 944–951. IEEE, 2010.

[150] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[151] I. Souilah, A. Francalanza, and V. Sassone. A formal model of provenance in distributed systems. In *Workshop on the Theory and Practice of Provenance*, pages 1–11, 2009.

[152] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

[153] Y. S. Tan, R. K. Ko, and G. Holmes. Security and data accountability in distributed systems: A provenance survey. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013*

IEEE International Conference on Embedded and Ubiquitous Computing, pages 1571–1578. IEEE, 2013.

[154] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1324–1337, New York, NY, USA, 2018. Association for Computing Machinery.

[155] J. Thalheim, P. Bhatotia, and C. Fetzer. Inspector: Data Provenance using Intel Processor Trace (PT). In *proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[156] W3C. An overview of the PROV family of documents, 2013.

[157] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[158] W. Wang and T. E. Daniels. A graph based approach toward network forensics analysis. *ACM Trans. Inf. Syst. Secur.*, 12(1):4:1–4:33, Oct. 2008.

[159] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, July 2003.

[160] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 504–516, New York, NY, USA, 2016. Association for Computing Machinery.

[161] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, 2012. USENIX.

[162] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu. Using provenance patterns to vet sensitive behaviors in android apps. In *Security and Privacy in Communication Networks*, pages 58–77. Springer, 2015.

[163] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, Nov 2010.

[164] T. Zanussi, K. Y. R. Wisniewski, R. Moore, and M. Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *OLS (Ottawa Linux Symposium)*, pages 519–531, 2003.

[165] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 295–310. ACM, 2011.

[166] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 415–425, 2015.