Provenance expressiveness benchmarking on non-deterministic executions

Sheung Chi Chan
Heriot Watt University

James Cheney
University of Edinburgh
The Alan Turing Institute

Pramod Bhatotia
Technische Universität München

Abstract

Data provenance is a form of meta-data recording inputs and processes. It provides historical records and origin information of the data. Because of the rich information provided, provenance is increasingly being used as a foundation for security analysis and forensic auditing. These applications require provenance with high quality. Earlier works have proposed a provenance expressiveness benchmarking approach to automatically identify and compare the results of different provenance systems and their generated provenance. However, previous work was limited to benchmarking deterministic activities, whereas all real-world systems involve non-determinism, for example through concurrency and multiprocessing. Benchmarking non-deterministic events is challenging because the process owner has no control over the interleaving between processes or the execution order of system calls coming from different processes, leading to a rapid growth in the number of possible schedules that need to be observed. To cover these cases and provide all-around automated expressiveness benchmarking for real-world examples, we proposed an extension to the automated provenance benchmarking tool, `ProvMark`, to handle non-determinism.

1 Introduction

There are many different provenance systems in the provenance research field. Some of them aim to record the data exchange flow and help answer queries such as how the data attained the current state and which entities are responsible for the modifications. Also there are some provenance systems that record system activities and provide ways for later reproduction of the execution flows or evidence for security auditing or digital forensics. Some examples of these systems include PASS [9], Hi-Fi [11], SPADE [8], OPUS [1], LPM [2], Inspector [14], and CamFlow [10]. These systems aim to collect provenance information from different components in different operating systems and generate high-level provenance graphs after certain post-processing. As there are no mandatory standards for provenance descriptions of system activity, different provenance systems may generate or filter out different information during the post-processing period and give different output for monitoring the same set of executions.

In earlier work, Chan et al. [4, 5] proposed an expressiveness benchmarking approach to elucidate the relationship between operating system activities and the resulting provenance graphs. They built an automated system to generate provenance benchmarks to describe system call behaviour and allow users to analyse and compare quality of provenance data, allowing a simple way for correctness and correctness checking with additional criteria. Later, Chan et al. [6] demonstrated an application of the ProvMark tool proposed in earlier work [4] for integrity checking and abnormality detection of provenance records which allows users to discover problematic provenance results. One of the big limitations mentioned by the above literature is that they only handled deterministic executions. This is far from realistic because concurrency and non-determinism are present in all realistic applications.

```
#include<unistd.h>
void main() {
  while(1) {
    if(fork()) {
      //Escalate to root privilege
    } else {
      //Add fake user with root privilege
    }
  }
}
```

Listing 1: Malicious execution with non-determinism

Many provenance systems for security and forensic auditing require a high level of accuracy and integrity of the collected data and mechanisms. This is because those data may act as important evidence for identifying and proving the existence of possible security incidents like intrusions or illegal system access. Those data may also be used for auditing regulatory compliance. Although in Chan et al. [6] the authors show that ProvMark is capable of discovering potential integrity problems for those provenance systems, the larger scale application remains an open problem because many of the realistic applications and attacks are working under non-deterministic settings, like the one shown in Listing 1 in which the real intrusion may exist in different non-deterministic branches. For a larger coverage of possible applications and to increase the usefulness of the ProvMark approach, it is necessary to extend the ProvMark mechanism to handle non-deterministic input. This setting allows ProvMark to provide better expressiveness benchmarking functionality for provenance systems on collecting accurate and complete provenance information for both deterministic and non-deterministic program execution. This could help to analyse the capabilities of those provenance systems for security and forensic applications. In addition, adding support for non-deterministic input can also provide a broader understanding to normal users of the behaviour and reliability of different provenance systems for non-deterministic program execution arising in real-life applications.

This paper proposes an extension to the ProvMark tool that can also handle non-deterministic program execution and covers a large range of provenance expressiveness benchmarking towards more realistic application executions. This approach shows that ProvMark is more capable of handling realistic executions to provide a beneficial comparison to users of provenance data quality checking for security applications. The extensions aim to answer the following questions.

- What are the differences between provenance graphs generated for deterministic or non-deterministic executions?
- How can we cover all of the non-deterministic paths to retrieve reliable results?
- How to identify if a certain non-deterministic branch has been executed in a trial run?
- How to analyse or compare provenance benchmarks for non-deterministic executions?

The structure of the rest of this paper is as follows. Section 2 provides some background of the ProvMark tool and its application to integrity checking and abnormality detection with deterministic input. Section 3 describes our extension of ProvMark in detail, including the methodology and implementation. Also, discussion of how to ensure non-deterministic path coverage is included. Section 4 shows some sample results generated by the extended ProvMark for certain non-deterministic examples. Additional analysis and comparison of the extension and the results themselves are included in this section. Section 5 discusses the usefulness of the ProvMark extension and discuss further limitations. Finally, Section 6 concludes and discusses ongoing and future steps.

2 Background

In Chan et el. [4, 5], the authors proposed *provenance expressiveness benchmarking* and then developed an automated system ProvMark to handle the benchmarking process. Their major contributions included the identification of how different system call behaviours contribute to the final provenance graph results and generate provenance benchmarks for each of the system calls automatically. These provenance benchmarks show how three different provenance systems (SPADE [8], OPUS [1] and CamFlow [10]) each handle the same set of operations in a different manner. They also act as a basis to identify what information is processed by each provenance system in the final provenance graph if certain activities happened in the execution period. Developers of those provenance systems can then use these provenance benchmarks to compare provenance data qualitatively for different applications. Besides, it also helps to discover possible problems, bugs and malfunctions in those provenance systems when some unexpected provenance benchmark has been received.

ProvMark works by comparing the provenance resulting from monitoring a *background program* that performs background activity such as process initialization and termination with a *foreground program* which also performs the additional target activities. Another key contribution of the original ProvMark tool is the adoption of answer set programming to help solve the hard graph/sub-graph isomorphism problems of property graphs [3]. This allows a more flexible way to compare provenance graphs with large numbers of elements including property labels.

For the provenance benchmarking process, we need to make use of certain provenance recording tools to generate provenance graphs at the beginning. We need to start the chosen provenance collecting tools with certain settings, then we need to execute the background and foreground binaries. ProvMark uses a unified format to analyse the provenance graph, so those provenance graphs need to be converted into this format. Datalog is the data format chosen for this purpose. After that, generalization process is performed to filter out volatile properties. These volatile properties are mainly timestamps and identifiers that are always different on different trial runs and do not provide much information for the real behaviour of a specific set of executions. Besides, they also act as noise because they are not pertained for each run and keep changing for every trial run which affect the accuracy of the graph filtering and benchmark generation process. Lastly, the generalized foreground and background graphs are compared to filter out the duplicate part and the remaining part in the foreground graph is the resulting provenance benchmark.

In a later paper, Chan et al. [6] performed small manual experiments to simulate unreliable sources of provenance collection to demonstrate the feasibility of the integrity checking and abnormality detection by ProvMark on deterministic executions. Their preliminary results show that ProvMark was able to assist the discovery of integrity problems in provenance records and underlying problems or attacks.

The above mentioned literature just concentrates on deterministic input. They are not sufficient to handle realistic cases which contain non-deterministic execution. This paper aims to extend ProvMark's benchmarking process to support both deterministic and non-deterministic activity sequences and make the tool applicable to more realistic situations.

In a general context, 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, which can be 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. Both *Socket system calls* and *Streaming system calls* are system calls handling communication and data transfer across different artefacts, processes and components. *Streaming system calls* perform internal communication and *Socket system calls* perform internet communications and streaming through sockets. *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 that make use of some *randomization and redundancy* 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.

3 Extending ProvMark for non-determinism

As mentioned above, our main objective is to extend ProvMark and its automated provenance benchmarking approach to non-deterministic activities. The original design of *ProvMark* requires comparisons of two provenance graphs (ignoring some volatile properties that keep changing in each trial executions), one named as background graph (bg) which contains the provenance describing the background activity and the other named as foreground graph (fg) which contains the provenance describing the target activity combined with the background activity. One of the major features of the *ProvMark* tool is to compare and identify the difference between the two graphs which represents the provenance structure of the target events. ProvMark retrieves the two provenance graphs by executing a benchmark program with CPP directive statement. The target activities in this special benchmark program are enclosed by the `#ifdef TARGET` CPP directive statement. The program is then compiled into two different binaries with or without the `TARGET` keyword defined. This results in two slightly different binaries which perform similar background activities, and one of them performs additional target activities. As a result, ProvMark retrieves two slightly different provenance graphs bg and fg, from the execution of the two similar programs and compares them to get the difference. The difference forms the provenance benchmark describing the target activities. In this section, we define our extension on ProvMark to handle provenance benchmarking on non-deterministic events.

3.1 Overview of the extension

The provenance benchmark generated by ProvMark can identify the kernel execution patterns for certain action sequences. The same action sequences may behave in a non-deterministic manner in the kernel throughout multiple executions. 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 [10] 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* [7, 12, 13]. 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 system calls) happening in the kernel and return debugging information describing all executions and action sequences. Similar to the LSM hook of CamFlow, Ftrace is a kernel utility which requires some module to pass the result back to the user level for processing. We choose *trace-cmd* to act as the front-end of Ftrace which is a tool shipped with many Linux distributions. 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 2, we put two system calls in separate threads. Both of the system calls will be executed eventually, but the execution order is non-deterministic. Another example in Code Snippet 3, we apply randomization to conditional branches, so different system calls are executed determined by the result of the random source. Different system calls may be executed for each trial run. 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 group the similar graphs together according to the combination and order of system calls in each of the graphs. 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 action 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<unistd.h>
void main() {
        if(fork()) {
                //Action 1
        } else {
                //Action 2
        }
}
```

Listing 2: Example for non-determinism with different orders

```
#include<stdlib.h>
#include<time.h>
void main() {
        srand(time(0));
        if (rand()%2 == 0) {
                //Path 1
        } else {
                //Path 2
        }
}
```

Listing 3: Example for non-determinism with different paths

In some of the cases, the executions of all possible combinations of an non-determinisitic input may not be possible if the number of system calls increases or more complex 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 a better coverage, we run the trial execution for multiple times more than the total number of combinations. For example, we will execute 12–16 trial runs for non-deterministic events that have 4 combinations. Although this may not be enough to cover all possible schedule, repeating for 3 to 4 times more than the total number of schedules is enough to cover most of the common schedules and those leftovers are only some less frequent schedules in those test cases we are using with small number of schedules. Scaling up to much larger numbers of non-deterministic schedules may require significant increase of the trial. In general, if most of the schedules are following a normal distribution, that amount of trial execution should cover most of the schedules for smaller cases. We then process the graph generalization and benchmarking processing on top of this assumption. We also provide an evaluation of the coverage in Section 4.

3.2 Tracing kernel actions

The original ProvMark tool uses a specialized module to control the underlying provenance recording module like SPADE [8] or CamFlow [10]. Those tools will monitor the execution of the binaries and return provenance graphs as a result. The process will execute multiple times to collect a set of background graphs and a set of foreground graphs for the generalization process. After that, we generalize each of the groups separately to avoid the incomparable problems between graphs representing different non-deterministic paths.

The first thing we need to do is to classify and group the graphs with the same schedules (non-deterministic path). Although it is possible to obtain the same provenance result on the execution of different non-deterministic path, we consider two trial runs on the same schedule if they have exactly the same set of system calls and execution order. The main reason for this assumption is because if they have the exact same set of executions, then it is unnecessary to distinguish them as they result in the same provenance benchmark. As a result, we can classify those provenance graphs by matching their system call order list. We use an existing kernel framework Ftrace and its user-level control client trace-cmd to retrieve all of the system call executions that are passing through the permission checking in the kernel performed by the Linux Security Enhancement (Linux SE).

In each trial run of the foreground program, we 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 undesired 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`, the background graph is always deterministic and so all of them should have the same sequence of system calls received from the *trace-cmd* components. This statement should also be true for foreground graphs of deterministic input.

3.3 Fingerprinting and grouping

After the execution of the binaries, the provenance recording tools will return a set of provenance graphs. In addition, the Ftrace framework will also return a set of system call schedules which relays through the trace-cmd utility to the user level. Although in our experiment, we are considering small sets of system calls initially, we eventually intend to adapt *ProvMark* for use on larger target non-deterministic 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. For easy classification, analysing efficiency and readability consideration, it is essential to control the size of these identifiers. To do this, 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 always a fixed size. This fingerprint is used directly as the identifier for the graph.

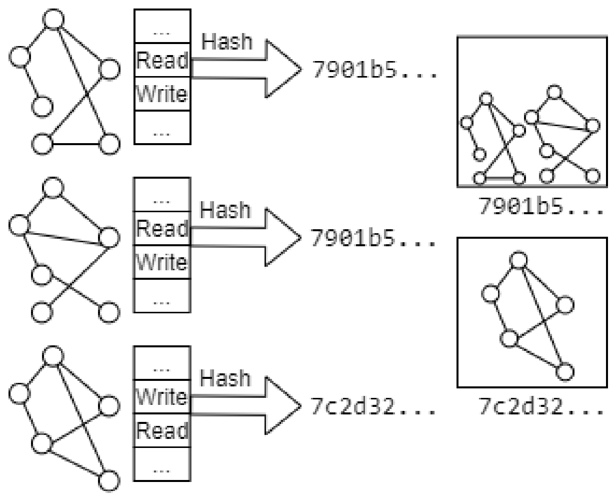


Figure 1: Fingerprinting and Grouping

The next step is to group the foreground graphs into similar groups and generalize the graph sets separately. 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 groups should also be preserved. The result of this stage should be a generalized background graph and a set of generalized foreground graphs, each with its own fingerprint identifier. Figure 1 shows an illustration of the fingerprinting and grouping process. Those graphs will be classified into groups and handled separately. The generalization process will be done for each group of graphs separately.

3.4 Generate multiple benchmarks

The clear difference from the handling of deterministic events is that there may exist multiple foreground graphs that need to go through ProvMark process. Each generalized foreground graph is associated with its unique 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. This implementation treats each of the non-deterministic schedules as a separate deterministic schedule and generates a provenance benchmark one by one. This approach transforms the non-deterministic execution to multiple deterministic executions for the provenance benchmarking process. As a result, each of the separate comparisons generates a different provenance benchmark, all of these benchmarks contribute to the group of provenance results for this specific non-deterministic execution.

Recalling the motivation 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 a one to one relationship. The resulting provenance benchmark represents the key elements to describe a certain action sequence for that specific provenance system, thus it acts as a benchmark for the tool. It can also be used as a pattern to identify the existence of the action sequence in a runtime environment for the same provenance system. 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 the new trials, we can confirm the existence of the matching of non-deterministic action sequences and we can even label them 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 behaviours 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.

4 Result analysis

This section provides a basic evaluation of our non-deterministic handling extension. We create one small non-deterministic program input for the evaluation.

4.1 Testing program

In the presence of non-determinism, there will not be much difference in the recording subsystem of ProvMark. Ftrace works in parallel with the provenance recording tools to capture the system call lists which are then concatenated and hashed to form the fingerprint identifiers. The additional step required from the original ProvMark design is the classification and grouping steps and treating each of the groups as a separate deterministic case for further processing.

Code Snippet 4 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. The total number of possible schedules is 6. We labelled the two write events as W_1 and W_2 and the two read events as R_1 and R_2. Because the write events and the 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 1.

```
#include <time.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main() {
int id=open("test.txt", O_RDWR);
char buf[1];
#ifdef PROGRAM
if (fork()) {
    read(id, buf, 1); // R1
    read(id, buf, 1); // R2
} else {
    write(id, "TEST", 1); // W1
    write(id, "TEST", 1); // W2
}
#endif
close(id);
}
```

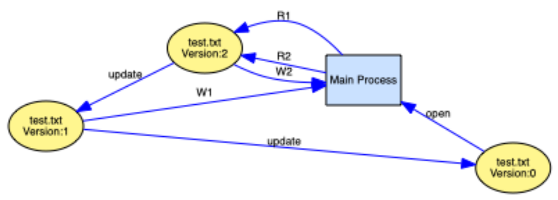
Listing 4: Sample benchmark program with non-determinism

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

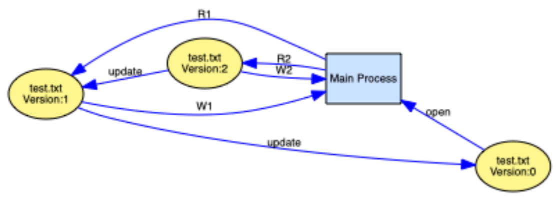
Table 1: All possible execution paths of Code Snippet 4

4.2 Sample provenance result

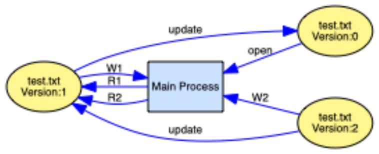
From the description of the testing program above, we understand that there are at most six different execution schedule possible for the execution of the program shown in Code Snippet 4. Theoretically, the choice of executing each of the schedule depends on many factors and should be considered as random. Thus we do not have any certainty guarantee that all schedules are covered because there may exist some less frequent combinations or we may simply be unlucky. We use the provenance tool SPADE [8] for this experiment to try generating provenance benchmarks for each of the schedules.



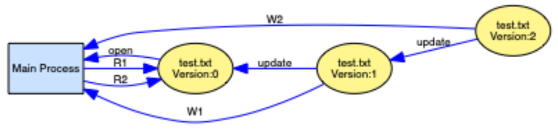
(a) Path #1



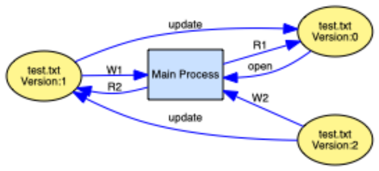
(b) Path #2



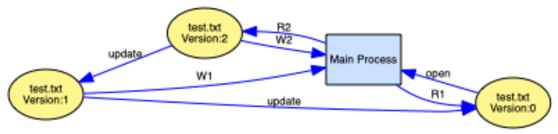
(c) Path #3



(d) Path #4



(e) Path #5



(f) Path #6

Figure 2: Provenance benchmarks for Code Snippet 4

Figure 2a-2f shows the six different benchmarks as an example. The `write` system call will trigger a version update of the files. As there are exactly two `write` events in all of the possible schedules, thus there are three versions of the file `test.txt` in the graph results. The different order of the `write` and `read` system calls does affect some edges as they are pointing out from a different version of the same file. Although they look similar, it does reflect that the read event is initiated from different versions of the file. For example, the result of Path #3 shown in Figure 2c has two `read` events between two `write` events. In this path schedule, the file has been updated once before the two `read` events, thus the two edges representing the `read` event points to version 1 of the file. On the contrary, the two `read` events happen before any `write` event in Path #4 (Figure 2d), which makes the resulting graphs contains two `read` edges pointing to version 0 of the file. The yellow oval represents the files in different versions while the blue rectangle represents the main process of the program execution. Due to limited space, some other property labels have been omitted.

4.3 Non-deterministic schedules coverage

One of the important considerations for the provenance benchmark generation of non-deterministic input is the coverage of possible schedules. In general, if there are *N* different schedules for a program, 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 4. As mentioned in Table 1, there are six possible schedules for this program. We defined twenty test cases. Each of the test cases has a different number of trial executions ranging from one trial to twenty trials. Each of the test cases is executed 10 times and results 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 3.

From the chart shown in Figure 3, we can see the average number of different schedules observed after executing varying numbers 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 cases 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 successfully cover all 6 schedules in 9 runs of the 10 trial executions of the test case, but there is 1 run that only covers 5 schedules for the 20 trial executions test case which make it not 100% coverage. Even if the experiment does succeed in covering all of the schedules, it is still not guaranteed 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 process or increasing the number of trials does help to increase the full coverage rate but it has no guarantee for full coverage.

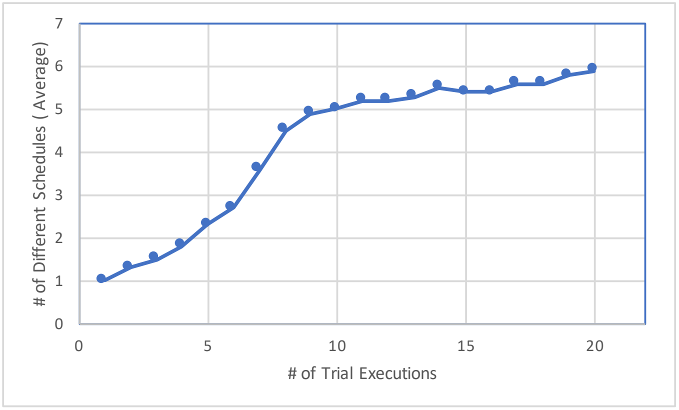


Figure 3: Average schedule covered for Code Snippet 4

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. It is always important to consider the balance between performance and accuracy. In addition to the above simple trial and error approach to determine the best threshold balance for the performance and full coverage, we also plan to make use of a symbolic execution approach to act as an evaluation mechanism to help us determine the best trial run needed for each of the experiment. This approach is one of the planned enhancements to our extension to ProvMark and are further discussed in Section 5.

5 Discussion

There are two limitations from our extension requiring further enhancement. In the integrity checking case study in Chan et al. [6], it provides two features. One of them is to identify if integrity problems and abnormality do exist in an execution. The other one is to showcase the provenance for the problematic part which helps the developers to locate the problem. Our extension to allow ProvMark to handle non-deterministic input still contains certain limitations. There is a possibility of false negatives with non-determinism. In our simple program, each possible path schedule contains exactly two read and two write events. Thus the random loss of system call records is identifiable from the incorrect number of the resulting graph as all schedule has four system calls for the correct execution.

But in reality, this may not be true for every execution. An example is shown in Listing 5. In this program, the execution path is either a single read event or a read event followed by a write event. If the current execution goes through the second path and the audit log of the write event is lost and causes an integrity problem, our approach will not be able to detect it as the resulting audit log and provenance graph will falsely match the model provenance graph for the first path and get an empty result. This creates a false negative event for the integrity checking. Currently, we can only repeat the experiment for more runs to reduce (but not eliminate) the false negative rate, it remains a limitation for the approach and requires further enhancement.

```
#include<stdlib.h>
#include<time.h>
void main() {
        srand(time(0));
        if (rand()%2 == 0) {
                read(...);
        } else {
                read(...);  write(...);
        }
}
```

Listing 5: Example for special case

Aside from the possible false negative detection, our approach contains another limitation. If there is a loss in the provenance collection, we are able to identify it but we may fail to detect the path of the target execution or locate the problematic path schedule. The major reason is that it is possible that two or more paths behave very similarly to each other when a certain system call is missing from the input. This remains one of the limitations of our approach extending to non-determinism and integrity checking reasoning. It requires future enhancement. Our approach does allow ProvMark to handle non-deterministic inputs, and additional consideration of every case and applications of this approach should be the next target for future research and enhancement.

In reality, the handling of non-determinism is one of the obstacles to using ProvMark for security analysis and intrusion detection besides the scalability problem. 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. It also helps to understand the behaviour of certain activity sequences and how provenance systems describe them. On top of that, the behaviour and reliability of the provenance system could also be evaluated by comparing benchmarks for (non-)deterministic and executions and across different provenance systems.

6 Conclusion

The original design objective of the automated ProvMark tool is aiming to provide all-around automated expressiveness benchmarking for real-world applications and executions. Thus ProvMark's initial lack of support for nondetermimism was one of the biggest obstacles to applying ProvMark to security analysis, intrusion detection and integrity checking. We proposed an extension to ProvMark to handle non-determinism. We then provide simple evaluation with small program examples. We also discuss limitations of our approach and possible future enhancements of enhanced ProvMark. We demonstrate that the enhancement of ProvMark helps to analyse the behaviour and reliability of the provenance systems used for security and related applications. This can help promote dependable provenance systems for specific security applications like intrusion detection, auditing and forensic evidence trace generation.

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 discover possible paths of execution and analyse them to understand what activity sequence has been executed in a runtime session. This work provides an alternative for identifying malicious behaviour in certain execution. Our extension to ProvMark is evaluated and shown to work well for handling different kinds of non-determinism. It is important to appreciate that we are not modifying the basic functionality of the ProvMark tool, thus the original assumptions and limitations of ProvMark still apply, with the exception that non-deterministic execution has been properly handled by our extension. Our work successfully extends ProvMark to handle non-deterministic coverage, both in experiment and a realistic case study. This approach allows ProvMark to provide more complete and realistic analysis and benchmarking of provenance tools, which result in more reliable and consistent provenance systems.

Acknowledgements

Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. Cheney was also supported by ERC Consolidator Grant Skye (grant number 682315) and an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS). This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under contract FA8650-15-C-7557.

References

[1] Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. OPUS: A lightweight system for observational provenance in user space. In *Proceedings of the 5th USENIX Workshop on Theory and Practice of Provenance (TaPP 2013)*. USENIX Association, 2013.

[2] Adam M. Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 2015)*, pages 319–334, 2015.

[3] Sheung Chi Chan and James Cheney. Flexible graph matching and graph edit distance using answer set programming. In *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings*, pages 20–36, 2020.

[4] Sheung Chi Chan, James Cheney, Pramod Bhatotia, Thomas Pasquier, Ashish Gehani, Hassaan Irshad, Lucian Carata, and Margo Seltzer. ProvMark: A provenance expressiveness benchmarking system. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 268–279, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Sheung Chi Chan, Ashish Gehani, James Cheney, Ripduman Sohan, and Hassaan Irshad. Expressiveness benchmarking for system-level provenance. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, Seattle, WA, June 2017. USENIX Association.

[6] Sheung Chi Chan, Ashish Gehani, Hassaan Irshad, and James Cheney. Integrity checking and abnormality detection of provenance records. In *12th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2020)*, Charlotte, NC, June 2020. USENIX Association.

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

[8] Ashish Gehani and Dawood Tariq. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International ACM/IFIP/USENIX Middleware Conference (Middleware 2012)*, pages 101–120, 2012.

[9] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 43–56, 2006.

[10] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David M. Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 2017)*, pages 405–418, 2017.

[11] Devin J. Pohly, Stephen E. McLaughlin, Patrick D. McDaniel, and Kevin R. B. Butler. Hi-Fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012)*, pages 259–268, 2012.

[12] Steven Rostedt. Debugging the kernel using ftrace. *LWN. net*, 2009.

[13] Steven Rostedt. Ftrace linux kernel tracing. In *Linux Conference Japan*, 2010.

[14] Joerg Thalheim, Pramod Bhatotia, and Christof Fetzer. Inspector: Data Provenance using Intel Processor Trace (PT). In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 25–34. IEEE, 2016.