Tradeoffs in Automatic Provenance Capture

Manolis Stamatogiannakis[1], Hasanat Kazmi[2], Hashim Sharif[2], Remco Vermeulen[1], Ashish Gehani[2], Herbert Bos[1], and Paul Groth[3]

[1] VU University Amsterdam,
{manolis.stamatogiannakis, r.vermeulen, h.j.bos}@vu.nl

[2] SRI International, {hasanat.kazmi, hashim.sharif, ashish.gehani}@sri.com

[3] Elsevier Labs, p.groth@elsevier.com

Abstract. Automatic provenance capture from arbitrary applications is a challenging problem. Different approaches to tackle this problem have evolved, most notably *a. system-event trace analysis*, *b. compile–time static instrumentation*, and *c.* taint flow analysis using *dynamic binary instrumentation*. Each of these approaches offers different trade-offs in terms of the granularity of captured provenance, integration requirements, and runtime overhead. While these aspects have been discussed separately, a systematic and detailed study, quantifying and elucidating them, is still lacking. To fill this gap, we begin to explore these trade-offs for representative examples of these approaches for automatic provenance capture by means of evaluation and measurement. We base our evaluation on UnixBench—a widely used benchmark suite within systems research. We believe this approach will make our results easier to compare with future studies.

Keywords: provenance, SPADE, taint tracking, LLVM, strace

1 Introduction

Automated provenance capture systems[4] which collect provenance information with minimal or no modification to a given application are important solutions for tracking and exposing provenance [4]. Mainly, they reduce the need for software to be re-engineered specifically for provenance. Additionally, they can capture more complete provenance as instrumentation can be done both broadly (e.g., across every application) and deeply (e.g., within the application itself). Automated provenance capture is complementary to disclosed provenance systems such as workflow management systems, version control systems, or databases, which require active engineering of the software to enable them to capture provenance [4].

There are number of different methods for automated provenance collection with varying trade-offs in requirements (e.g., the availability of source code),

[4] These are sometimes termed OS level provenance systems.

impact on application performance, granularity of provenance collected, and level of instrumentation required. The aim of this paper is to investigate these trade-offs. In particular, we compare three representative methods—system-event trace analysis, compile-time static instrumentation, and dynamic binary instrumentation—using their implementations for SRI's open source SPADEv2 [9] provenance middleware.

Our analysis is based on UnixBench [20], a widely used benchmark suite. We are aware that UnixBench emphasizes on performance of system calls and is not meant as a comprehensive performance benchmark. However, we believe that the results produced by it are still relevant for the evaluation of automatic provenance collection: Most such systems [12,8,18,3] tap (one way or another) into information derived from system calls. This is also true for the three systems we study (see §3). For this, supplemented with knowledge of specific features and requirements of a workload, the results produced by UnixBench can be used as input to decide on the suitability of a particular provenance collection method or system.

To the best of our knowledge, this is the first paper to comparatively benchmark provenance systems using a common systems benchmark. The need for exactly such benchmarks in provenance systems has been highlighted by the ProvBench series of workshops[5]. We discuss further steps towards the standardization of provenance benchmarks in §6. Standardized benchmarks are essential to provide a baseline for comparing iterations of the The contributions of this paper are as follows:

- A systematic comparison of three automated provenance capture systems using the UnixBench benchmark suite.
- An examination of the trade-offs when using these three methods.

The rest of this paper is organized as follows, we begin with a description of the evaluation platform and the three systems used. Experimental results are then presented. This is followed with a discussion of those results and their implications. Finally, we present future work and conclude.

2 Evaluation Platform

In this section, we discuss the framework we use for the evaluation, as well as some implementation details for the three fundamentally different methods of automated provenance capture we study.

2.1 SPADE

The SPADEv2 [9] provenance middleware aims to track the provenance of data that arises from multiple sources, possibly distributed over the wide area, and at varied levels of abstraction. Our choice of the SPADEv2 middleware was motivated by a number of factors. First, SPADEv2 has a modular design, allowing

[5] https://sites.google.com/site/provbench/

Table 1: Overview of provenance collection methods properties.

	system call analysis	**static, compile-time instrumentation**	**dynamic, instruction-level instrumentation**
integration effort	easy	medium	easy
prov. granularity[6]	file-level	function-level	byte-level
analysis scope	process and children	process, no dyn. lib.	process and children
false positives	many	depends on configured scope	negligible, tracks use of individual bytes
execution overhead	depends on the size of program I/O	depends on the number of function calls	high, depends on the taint tag type used
Reporter	`strace` reporter	LLVMTrace	DataTracker

most of its provenance filtering, storage, and query infrastructure to be used regardless of the instrumentation approach. Second, the distribution includes a number of *reporter* modules, each of which can be used to collect provenance using a different methodology. As a result, we can easily plug the different methods of instrumentation for our comparison while benefiting from SPADEv2's infrastructures. Third, the system supports storage of provenance in a number of data formats, including queryable ones such as the Neo4j graph database and the H2 (or any JDBC-compliant) SQL database. Fourth, the SPADEv2 platform can be configured and managed with a control utility. This allows an analysis to be repeatably executed (in order to measure behavior over multiple runs).

It is worth noting that the results of collecting provenance from the same program on different operating systems may differ substantially in runtime and storage overhead. Our comparisons have all been performed on Linux (see also Section 2.3).

2.2 Provenance Collection Methods and Reporters

In our experiments, we used implementations of three representative methods for automatic (i.e., non-disclosed) provenance capture: *a.* system-event trace analysis, *b.* compile-time static instrumentation, and *c.* instruction-level dynamic instrumentation. An overview of the properties of these methods is presented in Table 1. We now present the details of the specific SPADEv2 reporters we used that implement these methods. It is important to emphasize that the implementations of the three methods used in this evaluation are not necessarily the best or the fastest, but they serve as representative examples. For instance, it may be that a highly optimized taint analysis solution improves the performance of instruction-level dynamic instrumentation significantly, but the performance gap with compile-time solutions would most likely remain.

System-Event Trace: Strace Reporter. This first method for collecting data provenance treats the monitored program as a black box. By watching its interaction with the operating system, the method infers the set of artifacts that the program uses and generates.

The implementation we use monitors such interaction with the `strace` tool, which is available for Linux and Android. `strace` uses the *ptrace* facility available in Unix-like operating systems to learn which system calls (along with their arguments) are made by the program being monitored for provenance collection.

While tapping on `strace` simplifies the implementation of the reporter, it comes at a high cost because `strace` pauses the process twice for each system call. In order to avoid unnecessary overhead, `strace` reporter configures `strace` so that only the subset of system calls related to data flow are traced. Even after that performance may still be degraded for system-call heavy workloads.

The output of `strace` is parsed to generate the appropriate Open Provenance Model (OPM) [17] provenance elements.[7] Doing so imposes an additional overhead, compared to an implementation building directly upon the *ptrace* facility. The particular OPM elements generated are: *a. Process* elements for the operating system analog, *b. Artifact* elements for the files read or written, *c. Used* or *WasGeneratedBy* edges (depending on the use of the files), and *d. WasTriggeredBy* edges when one process creates another.

Compile-time Solutions: LLVMTrace The second approach for provenance capture is to instrument programs at compile time. Since a compilation of the application is required to enable provenance collection, compile-time solutions come closest to disclosed provenance capture techniques. However, no manual adaptation of the software is required.

Here, we use LLVMTrace as our representative implementation. It tracks intra-program data flows, providing a more precise dependency analysis. LLVMTrace utilizes the LLVM framework [14] to automatically add provenance instrumentation to applications at compile-time, using a custom compiler optimization pass [22]. The instrumentation is added at the entry and exit of each function call and logs its name, arguments, return value, and the thread that invoked it. Thus, LLVMTrace enables us to record the trace of function calls that occur during program execution. While this analysis obviously does not extend to dynamic libraries (see analysis scope in Table 1), compile-time library interposition is used to intercept and log calls to libc functions.

The produced logs are parsed in order to produce OPM provenance elements: *a. Process* elements are generated for each function call, *b. Artifact* elements are used to represent the function call arguments and return value, *c. Used* edges are used to associate a function with its argument, and *d. WasGeneratedBy* edges are used for return values.

---

[6] We use concrete rather than relative terms to describe the granularity of provenance. This is because in different application domains, a relative term (e.g. "fine-grained") may refer to different granularities.

[7] OPM can then be easily converted to the W3C PROV recommendation [11].

Thread-specific attributes are added to each provenance element, in order to to separate recorded activity from different threads into individual paths in the resulting provenance graph. The transformation from the function call trace to the provenance representation only captures *direct* data flows. Other types of information flow (e.g. use of shared buffers) are not captured.

Dynamic Instruction-Level Solution: DataTracker DataTracker [21] is a tool that captures provenance using *Dynamic Taint Analysis* (DTA). The analysis is applied as *Dynamic Binary Instrumentation* (DBI) using the Intel Pin [15] and libdft [13] frameworks. DataTracker adds instrumentation which determines how the application uses the data as it executes. This allows the tool to strongly reduce the number of false positives in the captured provenance compared to methods based on heuristics—albeit at a high cost. Like system-events based solutions, DBI has the benefit that provenance can be collected directly from unmodified binaries, without requiring development effort to make applications provenance-aware.

The type of taint metadata used by DataTracker is configurable. In [21], sets of <*file descriptor, offset*> pairs are used for tracking the provenance of each memory location. In this work, we instead opted to use *bitsets*—where each bit represents a file descriptor. We made this change because the implementation of `std::set` in `libstdc++` proved very inefficient in practice. The research of data structures that will enable DTA to track each input byte individually, while offering reasonable performance, is an open problem.

We used SPADEv2's *Domain-Specific Language Reporter* [9] (DSL reporter) to integrate DataTracker with SPADEv2. DSL reporter is middleware to allow the quick integration of new provenance sources with the SPADEv2 kernel. A converter transforms DataTracker's intermediate provenance representation to the OPM-based [17] language of DSL-reporter. The following OPM provenance elements are produced: *a. Process* elements are generated for each tracked OS process, *b. Artifact* are used to represent files and byte ranges[8], *c. Used* edges are used to associate input artifacts with processes, and *d. WasGeneratedBy*, *WasDerivedFrom* edges are used to associate output byte ranges with processes and input artifacts.

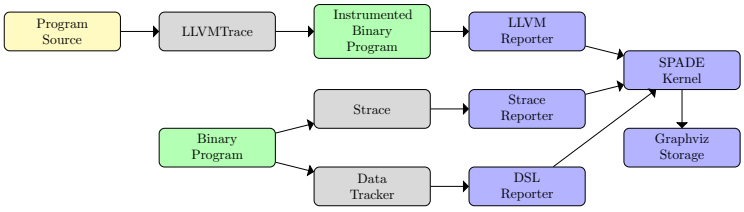

Fig. 1: Provenance collection workflow for the three SPADEv2 reporters.

[8] Byte ranges have a `memberof:` field pointing back to the file they come from.

The integration of the three reporters results in the provenance collection workflow illustrated in Fig. 1. As we are not interested in querying the capture provenance, we used the Graphviz storage backend of SPADEv2. An advantage of this choice is that it makes it easy to extend this work by adding results for the volume of collected provenance. Such information is readily available directly from Graphviz.

2.3 Hardware and OS

We ran our experiments on a machine featuring Intel Xeon E5-2630 CPU, with 6 cores clocked at 2.30GHz. The machine was configured with 1GB DDR3 memory module and a SSD storage module with 40GB capacity. We used 32 bit Ubuntu Linux 14.04.3 LTS to run our experiments. We used GCC 4.8.4 and LLVM 3.6.0 to compile UnixBench. GCC was used for `strace` reporter and DataTracker. LLVM was used for LLVMTrace.

3 Experimental Results

For our experiments, we use the UnixBench [20] benchmark suite. UnixBench was originally developed in 1983 at Monash University. It was adopted and popularized by Byte magazine in the 1990's and updated and revised by many people over the years. It still remains a popular general-purpose benchmark suite for the evaluation of the overall performance of Unix-like systems.

UnixBench is comprised of multiple parts that measure different aspects of a system's performance. Its main focus is to test how a system performs in basic operations such as file I/O, IPC, process creation, and system call invocation. Such operations are often tapped to extract provenance information [12,8,18,3], and thus are relevant to capturing provenance. This is also the case for the three SPADEv2 provenance reporters we study: *a.* `strace` reporter produces provenance solely by analyzing system calls, *b.* LLVMTrace traces the wrapper functions of the system calls, and finally *c.* DataTracker introduces taint when data are read, and logs provenance on writes.

We ran UnixBench first without any provenance reporter running (*baseline*) and then once for each of the three provenance collectors we study. The performance results can be seen in Table 2. Moreover, Fig. 2 shows the slowdown imposed by each reporter, compared to the baseline performance. In our study, we had to skip the Dhrystone (string handling performance) and Whetstone (floating point performance) tests of UnixBench. The former was skipped because of problems running it with LLVMTrace. The latter test would be of little interest, as all three of the studied reporters do not focus on floating point computation. The list of the performed UnixBench tests and a description of what they measure are as follows:

1. **execl-xput:** How fast the current process image can be replaced with a new one, as a result of an `execve` system call.

Table 2: Performance and Index scores for UnixBench tests.

Units for ops are as following: *a.* KBps for the fcopy-* tests *b.* loops per minute for the shell-* tests *c.* loops per second for the rest of the tests.

Test	**baseline**		**strace**		**LLVMTrace**		**DataTracker**	
	ops	index	ops	index	ops	index	ops	index
execl-xput	2285.5	531.5	668	155.4	1816.8	422.5	0.8	0.2
fcopy-256	120115.1	725.8	3303.5	20	91354.1	552	3624.7	21.9
fcopy-1024	352158.3	889.3	13133.3	33.2	397054.4	1002.7	7737.1	19.5
fcopy-4096	885101	1526	50492	87.1	954774	1646.2	11025.7	19
pipe-xput	813880.7	654.2	13745.5	11	711530.6	572	27658.9	22.2
pipe-cs	132217.1	330.5	6537.8	16.3	105752.7	264.4	11083.3	27.7
spawn-xput	7525.9	597.3	3229.9	256.3	1.4	0.1	12.2	1
shell-1	3816.4	900.1	1219.8	287.7	2291.3	540.4	2.6	0.6
shell-8	491.1	818.5	166.2	277.1	480.6	801	0.3	0.6
syscall	1140408.8	760.3	8388.9	5.6	695653	463.8	17921.7	11.6
Index Score		**720.8**		**53.3**		**257.8**		**4.6**

2. **fcopy-256, fcopy-1024, fcopy-4096:** Speed of a file-to-file copy using different buffer sizes.
3. **pipe-xput, pipe-cs:** Speed of communication over pipes. In the first test, the read and writes on the pipe happen from a single process. In the second test a second process is spawned, so the communication also includes a context switch between the two.
4. **spawn-xput:** A simple `fork-wait` loop to measure how much time is needed to create and then destroy a process.
5. **shell-1, shell-8:** Execution speed for the processing of a data file. The processing is implemented using common unix utilities, wrapped in a shell script. The two tests differ in the number of concurrently executing scripts.
6. **syscall:** System call overhead. The test uses `getpid` to measure this. The specific system call is chosen because it requires minimal in-kernel processing, so its main overhead comes from the switch between kernel and user mode.

4 Discussion of Experimental Results

In Table 1, we presented the overall features of three representative provenance collection methods. After evaluating their performance with UnixBench, we can draw conclusions with regard to the performance trade-offs involved when choosing which provenance method to use.

Integration Effort. The integration effort for using each reporter is associated with the changes required: *a.* for the tracked programs themselves *b.* the platform where the programs run on. There seems to be a correlation between the integration effort required and the runtime overhead of provenance collection.

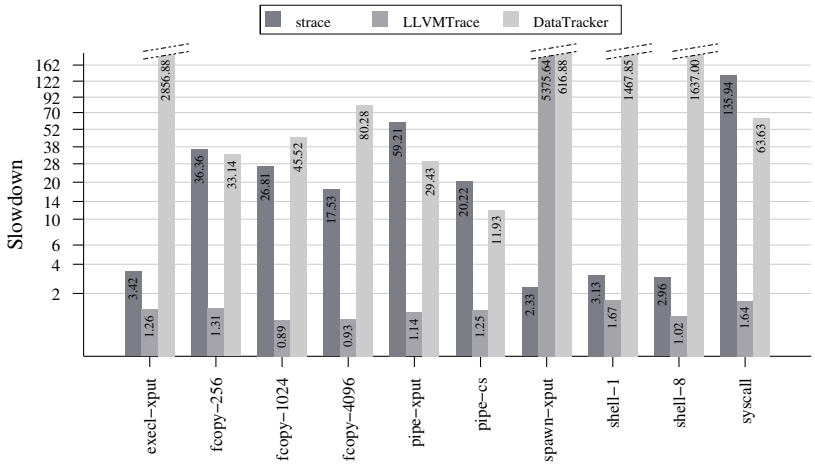

Fig. 2: Slowdown for the individual UnixBench tests.

From the studied reports, LLVMTrace requires the most integration effort because each tracked program has to be recompiled from its source. However, it also presents the lowest runtime overhead during provenance collection. On the other hand, `strace` reporter and DataTracker are the easiest to deploy, requiring no modification to the underlying platform (Linux) and working on unmodified binaries. However, their runtime overhead would be ranked from high to prohibitive. Specifically for system-event tracing, the overhead can easily be reduced if some integration effort is invested to modify the underlying platform. This is the approach taken in [18,3] which impose a very low runtime overhead (<4%). It should be noted however that these works either exclude the store runtime overhead from performance measurements [18], or use in-memory databases [3] to reduce it.

Provenance Granularity. The UnixBench results appear to be counter-intuitive when correlated with the granularity at which each method works. One would expect that system call tracing, which only tracks file-level provenance, would be the method with the lowest overhead. However, this doesn't appear to be the case for the implementations we study. The reason for this are two-fold: *a.* UnixBench focuses on system-call stress-testing, so a method relying on system-call analysis will suffer. *b.* The `strace` tool was not designed with efficiency in mind. It has to stop the program execution two times for each system call, in order to inspect its arguments and return value. In x86 this translates to trapping the SYSENTER, SYSEXIT instructions, a particularly expensive operation.

Another important observation related to provenance granularity, is that tracking such fine-grained provenance may nullify benefits from batching I/O. This is attested by the fcopy-* tests for DataTracker, where we can see that there is no benefit from using a larger buffer size. This can be explained by the fact

that irrespective of the buffer size, DataTracker has to individually update/log the metadata from all the memory locations.

False Positives. False positives are highly undesirable when collecting any type of data. Their presence degrades the value of a dataset. Provenance is no exception to this. However, as our measurements showed, in the case of provenance reducing false positives to very low levels comes at a significant runtime cost. Thus, in cases where false positives can be tolerated or easily filtered-out later, faster methods should be preferred.

In principle, provenance-based false positives originate from the fact that we treat software components as black-boxes and try to "guess" the provenance relations they produce. When a bad guess is made, a false positive is generated. DTA (as implemented by DataTracker), on the other hand, is a "track everything" attempt to see how those software black-boxes use the data, thus eliminating the need for guessing. In cases where the functionality of a software module is well known, this approach is clearly overkill. Recent efforts [16] attempt to eliminate this trade-off by switching between DTA and lightweight logging.

Analysis Scope. Not extending the scope of analysis to dynamic libraries, as LLVMTrace does, may appear as a limitation at first blush. However, it turns out that usually, this is a reasonable trade-off. This is because dynamic libraries usually have well-known behavior, which makes it possible to keep runtime overhead low by not tracking the internals of the library while still producing accurate provenance. This could be problematic however in programs that have pushed functionality into dynamic libraries. This is a quite common design strategy for many web-browsers. In that case, specific dynamic libraries have to also be recompiled to include provenance instrumentation. This choice represents a potential trade-off between the analysis scope and the ease of integration.

Overall, we see that when selecting a provenance system, there are key trade-offs between the extent of instrumentation both from a breadth and depth perspective and the resulting performance. We suggest that, when quick deployment is of importance, system event tracing is a good choice. Depending on whether the deployment will be permanent or not, one may choose to invest on deploying tools like Hi-Fi [18] or LPM [3] instead of using the tool used in our study. On the other hand, compile-time instrumentation can combine good performance with potentially less false positives for applications where the source code is available. The reduced false positives is a result of tracking provenance at a finer granularity than system events analysis. However, the effort required to properly apply compile-time instrumentation may become substantial for programs that use multiple dynamic libraries. Finally, the overhead added by DataTracker is prohibitive for time-critical applications. For this, it is best reserved for special cases. E.g. if one is interested in understanding the provenance produced by legacy applications (no source code, little/no documentation), tools like DataTracker may help identify properties that are masked by tools operating on a higher level.

5 Related Work

Performance and overhead of provenance capture systems has been identified as an important topic for the adoption of provenance systems. In their provenance primer [5], Carata et al. identify two dimensions of overhead: *a.* temporal overhead, which is the focus of this work, and *b.* spatial overhead, which is associated with the cost of storing the captured provenance. An important observation they make, is that the available data about performance of provenance capture systems are *not directly comparable*. This calls for the standardization of some benchmarks which can be used to have comparable results for future systems.

In general, it seems there is more interest in the spatial aspect of provenance overhead. Simmhan et al. [19] include only spatial overhead as a dimension in their provenance taxonomy. The ProvBench [1] effort focuses on collecting reference traces to help assess provenance storage and query processing time. Firth and Missier [7] have proposed to synthetically create provenance graphs. Similar efforts would help to also get better understanding of provenance collection runtime overheads.

This focus on spatial overhead could be partially explained by the fact that for many disclosed provenance systems, the runtime overhead is already low [5]. Glavic [10] observes that provenance can be intensive both in terms of computation and required storage. Moreover, he notes that by using DTA to capture fine-grained provenance (similar to DataTracker), one can generate a very large volume of provenance data from a small set of input files. So, provenance collection can be used as a benchmark workload for Big Data.

Finally, we note that we use only three implementations of automatic provenance capture methods. There are many other implementations such as PASS [12], ES3 [8], OPUS [2], Hi-Fi [18], Linux Provenance Modules [3] and PLUS [6]. Each of these systems has there own optimizations and capabilities for provenance. However, we believe the methods described here are broadly representative of these approaches.

6 Future Work and Conclusion

6.1 Future Work

In this work, we focus on exploring trade-offs related to the performance of provenance capture. There are many opportunities for future work. Here, we focus on those opportunities to do with further benchmarking.

Non-performance trade-offs. The community has shown interest in issues related to the storage and querying of captured provenance (see §5). So we believe that it would be interesting to include measurements about the storage required by each provenance capture method. Another systems-related aspect that would be of interest is the memory requirements of each approach. Having low-memory requirements becomes important in shared environments (e.g. virtualized servers or multi-purpose server boxes).

Comprehensive benchmarking. As we have already mentioned, UnixBench puts emphasis on the system call performance. However, in many real-life workloads system calls account only for a fraction of the execution time. To achieve a more comprehensive evaluation, more types of benchmarks should be used. We initially had planned to use selected Coreutils[9] as micro-benchmarks to complement the results of UnixBench in this work. Coreutils include small data manipulation programs with well-understood behavior, which may also include substantial computation in the user-space. However, due to to time constraints, we had to defer their publication. Another option would be to use a set of more complex programs as the basis for comparing provenance tools and methods. E.g. [18] uses the compilation of the Linux kernel as a benchmark and the Postmark mail server benchmark. In addition to that, [3] uses the BLAST benchmark which is based on biological sequencing. Using larger benchmark suites should also be investigated. However, it is not necessary that all tests in a benchmark suite will be suitable for benchmarking provenance capture. E.g. the SPEC benchmark makes heavy use of interpreted programs.

Qualitative Benchmarks. Another aspect of provenance capture is the quality of the produced provenance. In order to assess a method or tool with regard to its quality, we need an established ground truth against which we compare. If we know the ground truth for a given set of tasks, then we can calculate the precision/recall of each compared method and rank them accordingly. Disclosed provenance systems could be used to establish a ground truth for qualitative benchmarks of non-disclosed tools and methods. Another option would be to use tools like DataTracker which are not prone to false positives and can produce fine-grained provenance.

6.2 Conclusion

In this paper, we studied the performance of three methods for automated provenance capture, as implemented for the SPADEv2 provenance middleware. We used UnixBench, a widely used benchmark, focusing mostly on performance of system calls. As UnixBench does not include adequate variety of workloads, our presented results are clearly not enough to fully evaluate the performance of the studied methods. However, we believe that the trade-offs we present can still provide some insights on the suitability of the methods for capturing provenance of specific workloads. More importantly, we consider this work as a first step for the systematic and multi-faceted performance evaluation of provenance capture systems. Having such information will provide a baseline for the concrete assessment of improvements in future provenance capture methods and systems.

References

1. ProvBench: A Provenance Repository for Benchmarking (2013), https://github.com/provbench, [Online; Feb. 2016]

[9] GNU Coreutils: http://www.gnu.org/software/coreutils/

2. Balakrishnan, N., Bytheway, T., Sohan, R., Hopper, A.: OPUS: A Lightweight System for Observational Provenance in User Space. In: Proceedings of USENIX TaPP'13. Lombard, IL, USA (Apr 2013)
3. Bates, A., Tian, D., Butler, K.R.B., Moyer, T.: Trustworthy Whole-system Provenance for the Linux Kernel. In: Proceedings USENIX SEC'15. Washington, DC, USA (Aug 2015)
4. Braun, U., Garfinkel, S., Holland, D.A., Muniswamy-Reddy, K.K., Seltzer, M.I.: Issues in Automatic Provenance Collection. In: Proceedings of IPAW'06. Chicago, IL, USA (May 2006)
5. Carata, L., Akoush, S., Balakrishnan, N., Bytheway, T., Sohan, R., Se ltzer, M., Hopper, A.: A Primer on Provenance. ACM Queue 12(3), 10:10–10:23 (Mar 2014)
6. Chapman, A., Blaustein, B.T., Seligman, L., Allen, M.D.: PLUS: A provenance manager for integrated information. In: Proceedings IEEE IRI'11. Las Vegas, NV, USA (Aug 2011)
7. Firth, H., Missier, P.: ProvGen: Generating Synthetic PROV Graphs with Predictable Structure. In: Proceedings of IPAW'14. Cologne, Germany (Jun 2014)
8. Frew, J., Metzger, D., Slaughter, P.: Automatic capture and reconstruction of computational provenance. Concurr. Comput.: Pract. & Exper. 20(5) (2008)
9. Gehani, A., Tariq, D.: SPADE: Support for Provenance Auditing in Distributed Environments. In: Proceedings of Middleware'12. Montreal, Canada (Dec 2012)
10. Glavic, B.: Big Data Provenance: Challenges and Implications for Benchmarking. In: Specifying Big Data Benchmarks. Pune, India (Dec 2012)
11. Groth, P., Moreau, L.: PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note NOTE-prov-overview-20130430, W3C (Apr 2013), http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/
12. Holland, D.A., Seltzer, M.I., Braun, U., Muniswamy-Reddy, K.K.: PASSing the provenance challenge. Concurr. Comput.: Pract. & Exper. 20(5) (2008)
13. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In: Proceedings of VEE'12. London, UK (Mar 2012)
14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of CGO'04. Palo Alto, CA, USA (2004)
15. Luk, C.K., et al.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of PLDI'05. Chicago, IL, USA (Jun 2005)
16. Ma, S., Zhang, X., Xu, D.: ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In: Proc. of NDSS'16. San Diego, CA, USA (Feb 2016)
17. Moreau, L., et al.: The Open Provenance Model core specification (v1.1). Future Generation Computer Systems 27(6), 743 – 756 (2011)
18. Pohly, D.J., McLaughlin, S., McDaniel, P., Butler, K.: Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In: Proceedings of ACSAC'12. Orlando, FL, USA (Dec 2012)
19. Simmhan, Y.L., Plale, B., Gannon, D.: A Survey of Data Provenance in e-Science. SIGMOD Rec. 34(3) (2005)
20. Smith, B., Lucas, K., et al.: UnixBench: The original BYTE UNIX benchmark suite (2011), https://github.com/kdlucas/byte-unixbench, [Online; Feb. 2016]
21. Stamatogiannakis, M., Groth, P., Bos, H.: Looking Inside the Black-Box: Capturing Data Provenance using Dynamic Instrumentation. In: Proceedings of IPAW'14. Cologne, Germany (Jun 2014)
22. Tariq, D., Ali, M., Gehani, A.: Towards Automated Collection of Application-Level Data Provenance. In: Proceedings of USENIX TaPP'12. Boston, MA, USA (2012)