# Towards a Universal Data Provenance Framework Using Dynamic Instrumentation

Eleni Gessiou[1], Vasilis Pappas[2], Elias Athanasopoulos[2], Angelos D. Keromytis[2], and Sotiris Ioannidis[3]

[1] Computer Science & Engineering Department
Polytechnic Institute of NYU
gessiou@cis.poly.edu
[2] Department of Computer Science
Columbia University
{vpappas,elathan,angelos}@cs.columbia.edu
[3] Institute of Computer Science
Foundation for Research and Technology - Hellas
sotiris@ics.forth.gr

**Abstract.** The advantage of collecting data provenance information has driven research on how to extend or modify applications and systems in order to provide it, or the creation of architectures that are built from the ground up with provenance capabilities. In this paper we propose a universal data provenance framework, using dynamic instrumentation, which gathers data provenance information for real-world applications without any code modifications. Our framework simplifies the task of finding the right points to instrument, which can be cumbersome in large and complex systems. We have built a proof-of-concept implementation of the framework on top of DTrace. Moreover, we evaluated its functionality by using it for three different scenarios: file-system operations, database transactions and web browser HTTP requests. Based on our experiences we believe that it is possible to provide data provenance, transparently, to any layer of the software stack.

## 1 Introduction

All systems occasionally store their activity in log files, a process we usually refer to as logging. A log file can contain error messages, warnings, and important information for a user debugging a problem or investigating the condition of a running process. However, traditional logging is very limited. It lacks of semantic and critical information related to the internal state of an object. For example, a log file can hardly represent the state change for a database executing a particular query. Thus, we seek of tools and technologies that can monitor an object and collect intrinsic information associated with its internal state. More precisely, instead of having a collection of log files containing warning and error messages we are interested in the data provenance information.

Data provenance – describing how an object came to be in its present state – is an area that has drawn much research attention lately. There is a number of

D. Gritzalis, S. Furnell, and M. Theoharidou (Eds.): SEC 2012, IFIP AICT 376, pp. 103–114, 2012.

schemes that have been proposed on how to apply it in practice and in widely used applications. Some representative provenance schemes have been proposed for file systems [18], databases [6], web applications [13,17,22], cloud computing [19], smart phone operating systems [8], and web browsers [16]. One could argue that applying data provenance at a low level, e.g. at the system call-level, would be sufficient. However, that is not always the case. Applying data provenance to different layers of the software stack can provide different levels of information. In the database case, for example, it would be much more efficient to apply it at the query-level, where all the data, along with any meta-information (like ownership) is available, than at the system call-level, where one would get very limited information like I/O operations on file blocks. Also, in the web browser case, data provenance provided solely at the system-call level will be unable to capture and record higher-level information. This can be the URLs the browser renders, in the case the communication channel is over HTTPS. On the other hand, the function-call level, as provided by OpenSSL [23], has access to the encrypted data stream used by an HTTPS connection.

By looking at these examples it becomes clear that depending on the case, data provenance must be provided at the layer closest to the information we would like to capture and document. This may prove to be a very challenging task, especially when dealing with large and complex software applications such as web browsers and databases. For example, Firefox 4 consists of more than 5M lines of code, spread in almost 40,000 files. In the case of MySQL 5.5, there are 1.2M lines of code, in almost 3,000 files.

In this paper, instead of exploring how we can extend an application to provide data provenance, we take an alternate route. We propose, and build, a universal data provenance framework using dynamic instrumentation. The main goal of our framework is to provide an easy way to prototype any data provenance scheme, no matter the size and complexity of the system it is applied on. We argue that lightening the implementation burden of such data provenance applications would lead research on this field forward, allowing researchers to test and evaluate their ideas more easily. Our framework is built on dynamic instrumentation, and especially on DTrace. Its key feature is that it can greatly assist the user in discovering paths in the system that interesting data pass through. For example, all URLs and search keywords that flow inside a browser. After the discovery phase, the user can dynamically instrument these points to record provenance about this transit data. Moreover, in cases where the source code of a system is available, our framework could be used for simply discovering points of interest in the system and evaluating a data provenance application. Then, prototyping a low-overhead source-code level implementation of the data provenance application at production level is trivial, as the user knows exactly which points to instrument in the system.

The choice of dynamic instrumentation for such cases seems ideal as it provides several advantages. First of all, it requires no changes to the source code of the system that data provenance capabilities are going to be added in. Second, it can be easily enabled or disabled, even at runtime in some cases, as we shall see

later on. Finally, it removes the requirement of having the source code, although having it could be helpful during the discovery phase. On the other hand, the main disadvantage of dynamic instrumentation is its runtime overhead. However, this is not an issue in our case as we perform system-call and function-level, but not instruction-level instrumentation, which can be extremely expensive.

**Contribution.** The contributions of this paper are the following.

- We deliver a generic instrumentation framework for acquiring information at the system-call and function-call level. The collected information can reveal data provenance information.
- We design techniques for easily discovering the important points, which are associated with particular state changes of a process. For example, our framework can automatically discover the functions that process SQL queries in SQLite.
- We provide three case studies with real world systems: (a) a file system, (b) a database (SQLite), and (c) a web browser (Safari).

## 2 Background

Tracing is the process of observing the execution of a program for collecting useful information of diagnostic and systemic nature. Various techniques have been developed for supporting this facility throughout the development cycle as well as after the deployment of a system. Instrumentation is one such technique that allows someone to augment the execution of a program with new, user-provided, code that aids in collecting data for analyzing the behavior of a system.

DTrace is a dynamic instrumentation utility that focuses on production systems. It allows the instrumentation of both user-level as well as kernel-level code in a unified and safe manner and has absolutely zero performance cost when disabled. Initially, DTrace was developed for Sun's Solaris 10, but it has been integrated also into Apple's Mac OS X/Darwin (since 10.5/9) [3], FreeBSD (since 7.1) [9], as well as into other microkernel designs such as QNX (i.e., the Unix-like, real-time OS for embedded systems) [20]. Moreover, Oracle Linux latest version included a port of DTrace for Linux [14].

Since the primary focus of DTrace is production systems, it was designed around two key properties: (a) zero performance cost when disabled and (b) absolute system safety when enabled. Its dynamic nature allows to be injected on demand into virtually every place of a running system without suffering from the performance burden of static "disabled probes". Systems that support static instrumentation typically induce some disabled probes overhead. Dynamic instrumentation allows truly zero cost, since the probes are dynamically attached and detached on demand, and hence, they are "absent" when the instrumentation is disabled. User-provided instrumentation code, also known as analysis code is written in a high-level language, named D, that is subjected to a set of run-time checks for guaranteed safety.

D is C-like but it also resembles AWK [2] a lot in terms of structure. It has support for all ANSI C operators, it allows access to user- and kernel-level

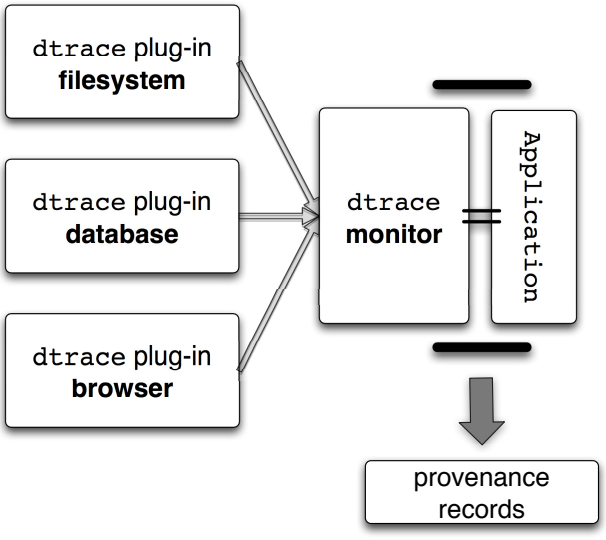


**Fig. 1.** A complex scenario where provenance information from different sources is combined

variables and data structures, and offers dynamic user-defined variables, structs, unions, and associative arrays. The scoping rules of the language, its intrinsic data types, as well the program structure are explained in great detail in [5].

The core part of DTrace lies inside the OS kernel and includes all the necessary facilities for providing an infrastructure for dynamic, arbitrary, tracing. User-level processes become DTrace consumers by communicating with that in-kernel component and enabling instrumentation. However, the DTrace framework does not perform any instrumentation of the system. This functionality is provided by the providers; kernel-level parts, typically loadable modules, that communicate with the core engine using a well-defined API. Providers declare to DTrace the points that can potentially instrument by providing a callback function. All in all, DTrace provides merely a skeleton for supporting future instrumentation methodologies. Nonetheless, it comes with a set of ready-to-use providers that have no observable overhead when disabled.

## 3 Design

The goal of the framework is to extract and gather provenance information related to the activity of complex systems, such as a web browser or a database. The collection of information is carried out by monitoring a process at the system-call and function-call level. This task requires the knowledge of all valid entry points, where data flows inside the running process. We refer to all system and function calls, which process critical information as valid entry points.

For example, a modern web browser, such as Mozilla Firefox, has a function that processes each input URL. This function is critical, since each processed URL denotes a semantic transition in the state of the web browser during its life cycle. The framework is generic enough for allowing information gathering from multiple valid entry points. The core engine of the framework is based on DTrace, which gives all low-level primitives for dynamic instrumentation of running processes. Furthermore, we provide two basic features for reducing the complexity associated with identifying *interesting* valid data entries. First, we provide a highly configurable logging component and, second, we provide *assisted discovery*. We now analyze both of these features in detail.

### 3.1 Configurable Logging

The logging component delivers the core functionality of our framework as it is responsible for generating provenance information. By default, the component monitors all system calls of all running process. For each monitored system call the following information is collected: (a) system-call arguments, (b) return value, (c) user id, (d) process name, (e) process id and (f) timestamp. The component provides an interface where a user can configure which processes and which system calls are monitored. This component can also be configured to log library function calls, by specifying the library's name along with the name of the function. For example, a security researcher who wants to monitor all symmetric cryptographic operations of a running program, can specify a monitoring set of valid entry points composed by: `EVP_EncryptInit_ex`, `EVP_EncryptUpdate_ex`, and `EVP_EncryptFinal_ex`. All these three functions can be found in the OpenSSL [23] library which is used as the *de facto* standard for cryptographic operations.

Another example is the following. Suppose that we want to log all the `write()` system calls issued by the text editor `vi`. In that case, the following script is generated by our framework (note that the predefined variable `arg0` holds the return value in the second probe):

```
syscall::write:entry
/execname == "vim"/
{
    self->arg1 = arg1;
}

syscall::write:return
/execname == "vim" && self->arg1/
{
    printf("%Y %s (%d) uid %d write: %s -> %d",
           walltimestamp, execname, pid, uid,
           copyinstr(self->arg1), arg0);
    self->arg1 = 0;
}
```

### 3.2 Assisted Discovery

Complex systems, such as web browsers and databases, are composed by a huge code base. While these systems run, there are many different transitions that can significantly alter their state. For example, each query processed by a database can trigger a group of transitions that will drive the database in a completely new state. It is challenging to identify all valid entry points, which initiate all these state transitions. A key feature of our framework is that it can greatly reduce the search space of such points. This is achieved by first instrumenting all the functions and checking for a specific value that the user has given as input in the argument list of the monitored system. If found, the name of the function, the call stack, or both, are logged. Then, the user can decide which of the collected functions qualify as valid entry points. The subset of functions narrowed down by our technique is significantly reduced compared to the overall size of the system in the average case.

*Example:* We now present a real-world example for a better understanding of how assisted discovery works. Suppose that we have a new application for data provenance in a database system. To evaluate its efficiency, we first need to log all the queries. Normally, we have two options. We can either download the source of the database and find the appropriate places in the code for logging the queries, or, we can proxy the inputs (queries) of the database system using an external software component. The first approach can be very difficult, depending on the size and complexity of the system, or completely infeasible, in case the source code is not available. The second one would be difficult to enable/disable as it would require restarting the database system. In addition, a proxy can hardly provide any information associated with internal states of the system. For example, consider a SQL query which is triggered by a user-generated query and it is initialized and executed internally in the database for optimization purposes. This query cannot be captured by the external proxy, since it is generated internally inside the database. On the other hand, our framework can easily discover the appropriate points for monitoring and logging any information we need transparently and on demand. This is achieved by instrumenting all the functions within the database system and search for a special argument value, at runtime. This argument value should be a query string, like `'SELECT assisted FROM discovery;'`, and the output of our framework will be a set of function names that had this character sequence as an argument.

## 4 Case Studies

To demonstrate the effectiveness of our framework, we evaluated it in three scenarios, namely file-system, database and web browser data provenance. The reason behind choosing these applications is because they have been already studied in the literature [18,6,16]. The case studies are presented in order of increasing complexity for our framework and, interestingly, the most complex one revealed some limitations that we consider in Section 5.

### 4.1 File-System

Data provenance in the context of file-systems was introduced in PASS [18]. To gather provenance information, PASS was implemented in the Linux kernel. Here, we show how we can gather similar information using our framework, without having to dig and alter the code of the kernel. We define the set of system calls that create or change the current state of data, as provenance in a file-system. This set includes the following systems calls: `creat`, `write`, `chmod`, `chown` and `unlink`. We also include the `open` system call for completeness. The output includes a timestamp (indicating the exact moment that the system entered a system call), the executable's name, the process id and the user id that makes the system call. Also, in case that the system call's arguments contain useful information, like the location of file, or the file's permissions, we log this information too.

An example of the information that is recorded by our framework when a user edits a file using the `TextEdit` application on Mac OS X is shown bellow:

```
2012 Jan 7 23:44:52 TextEdit (23624) uid 501 open: .../paper.txt -> 15
2012 Jan 7 23:44:52 TextEdit (23624) uid 501 write: 15, My paper -> 9
2012 Jan 7 23:44:52 TextEdit (23624) uid 501 close: 15 -> 0
```

This scenario shows that using the powerful logging capabilities of our framework we can gather system call level information about the file-system by simply specifying the set of the file-system related system calls, without performing any changes in the operating system itself.

### 4.2 Database (SQLite)

The second scenario we consider is in the context of databases and demonstrates the assisted discovery feature of our framework. Our goal is to locate the appropriate point (i.e., function) to log queries in SQLite [1]. Initially, we created a simple table with a few columns and added some test data. Then, our framework instrumented all the functions in `libsqlite3.dylib`[1] at runtime.

Then, while assisted discovery was enabled, we performed a few different types of queries (insertions, deletions, updates, etc.). Each time the given query string appears in the argument list of an instrumented function, assisted discovery logs it. An example of these logs is the following:

```
sqlite3 (27426) libsqlite3.dylib:sqlite3_prepare_v2 'create table ..'
```

Our framework logged several functions, but, just by looking at their names, it was obvious that `sqlite3_prepare_v2` was the most appropriate function to instrument in order to log all the queries. Then, given the syntax of each SQL command, we parsed them in order to keep track of any changes made to the database. Generally, the provenance information about the database entries include the creation (`create`) and deletion (`drop`) of tables and their contents

[1] `.dylib` is the filename extension for dynamically loaded libraries in Mas OS X.

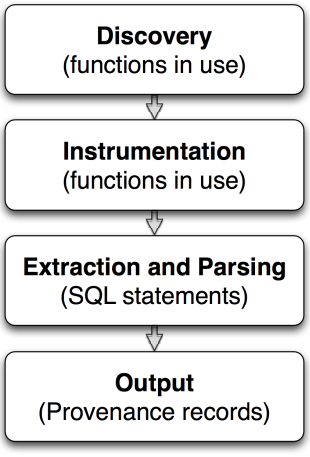


**Fig. 2.** The steps extracting provenance information from the SQLite database system using assisted discovery

(`inserts`) with corresponding timestamps. Also, the process name and id combined with the user id are logged for every SQL command that is executed. The procedure described above is also depicted in Figure 2.

We also calculated the performance overhead of logging provenance information using dynamic instrumentation versus logging the same information by altering the source code. We modified the `sqlite3_prepare_v2` function to record every SQL query made. Then, we measured the total time needed to perform one million `insert` operations in both versions of SQLite, the non-modified but dynamically instrumented and the modified one. On average, the performance overhead of dynamic instrumentation in this case was around 0.8%. Thus, for simple operations, like logging, there is no much difference performance-wise.

### 4.3 Web Browser (Safari)

Another example for applying the idea of provenance is that of the browser. We treat the notion of data provenance in a browser as it has been proposed in [16]. The main idea is the correlation between a search term and the URL that the user finally follows from the search engine. In other words, a search term is the provenance of the URL that the user asks for.

We implement this scenario by tracking the system calls that are called, such as `read`, `write`, `send` and `recv` using our framework. Although sufficient as a proof-of-concept implementation, we have to note that system call level tracking produces redundant information. It becomes too hard to separate the useful information for provenance from "noise". Moreover, the system call level instrumentation cannot be used when data are encoded, encrypted, etc. Finally,

assisted discovery failed to locate functions, in a higher level than system calls, where the search keywords and the URLs could be logged. We discuss more about this in Section 5.

## 5 Discussion and Future Directions

The last usage scenario, extracting provenance information from a web browser, revealed some interesting limitations of our framework against very large and complex systems. These limitations are mostly due to the restricted instrumentation actions of DTrace. Combining a more powerful tool in our framework, like the dynamic binary instrumentation of Pin [15], seems sufficient to overcome these limitations, but in some extra cost. Also, tools implemented on top of Pin, like libdft [12] which provides byte-level data flow tracking, could potentially add more features to our framework.

Although Pin does not come built-in in any operating system, it can be installed in many of them, like Microsoft Windows, Linux and Mac OS. As DTrace, Pin can also be used to instrument arbitrary functions and system calls, but not on demand. The program to be instrumented has to be executed through Pin. The main advantage of Pin over DTrace though is that there is no limit in what an instrumentation actions can be. Although that may not seem very important, there are at least two cases that this feature extends the functionality of our framework. First off, we can better track the data in the assisted discovery phase when it is encoded, or encapsulated inside a complex type. As an example, suppose that we want to track a string value in a large software system that happens to use a custom string class and we know nothing about its internal representation. By using Pin during the assisted discovery phase though we can either convert them to simple character sequences or use any specific compare function they provide. The second advantage of unconstrained instrumentation actions is the ability to alter the state and the data of the program.

To better understand and demonstrate the extra functionality that such a tool adds to our framework, let us consider again the example of implementing data provenance in a browser. More precisely, suppose that in order to implement a hypothetical data provenance scheme we want to log all the URLs that a user navigated to in the Safari web browser. Safari uses the WebKit open source HTML rendering engine, where URLs are represented by `class KURL`. To successfully track a user-input URL during the assisted discovery phase, the instrumentation action has to compare `KURL` objects with the input string. Then, after we locate the appropriate function to instrument in order to log all the URLs, our framework will convert the `KURL` object to a simple string and then log it along with a timestamp, etc.

## 6 Related Work

Data provenance has been a very active field of research lately, focusing both on theoretical and practical issues. We discuss below the works most related to our dynamic data provenance framework.

Most of the recent work has been on data provenance in database systems [4]. Also, more theoretical aspects of the provenance notions (where, why and how) have been studied [6]. In our work we focus on the practical aspects of data provenance, on any software application and not simply databases.

The authors of [16] describe how a browser enabled with provenance capabilities can improve user experience. They provide several scenarios for history search, web search and download management, that a browser with provenance can offer. In Section 4.3 we show an example of how this kind of data provenance application could be achieved using our framework, without any modification to the actual web browser.

Story Book is a system that adds provenance in different systems [21]. It runs in user space and treats provenance events as a generic event log. It collects application-specific provenance and supports a file system and a database. PASS [18] supports provenance at the system level and is a layer grafted in a file system. It gathers provenance for all file activities by inspecting system calls. Again, an example of how this could be implemented using our framework is given later on. As an advantage, our approach requires no modifications to the file system, or any other layer for that matter.

Another system that combines both static and dynamic analysis to trace provenance is Garm [7]. While our framework is transparent, Garm has to rewrite the binary when the binary executed. The main advantage of our approach over Garm's is that our framework can be applied to any level of the system, capturing even very high-level information. On the other hand, Garm only operates on low level data operations.

Recently, the notion of provenance has also been introduced in cloud infrastructures [19]. Our framework could be successfully applied in such environments as well, as it is execution environment agnostic.

Transient provenance [10] keeps information about emigrant data, like files that were moved, who moved them, and when they were moved, aiming at facilitating the administrators in case of a leak. The authors propose the creation of ghost objects so as to be able to record when a file is leaked from the central system, and moreover by whom, by logging the user ID. This way, after an information leakage happens, the suspects, the timeline and the objects of leakage can be reduced up to the people, timestamps and files, recorded by the ghost objects, giving administrators a valuable hint.

iLeak [11] is a system proposed for detecting inadvertent information leaks. Although not related to data provenance, iLeak is related to our work as it is also implemented on top of DTrace. Any sensitive data within a system have to be initially marked with the appropriate tags. Then, iLeak constantly monitors for their leakage by dynamically instrumenting the system's communication channels (sockets, SSL connections, etc.) using DTrace.

## 7 Conclusion

We designed and implemented a data provenance framework that can be used for rapid prototyping of any data provenance scheme. Our framework is built on dynamic instrumentation and provides an easy way to locate the right points to instrument, as well as a configurable logging component.

Our experience from implementing three real-world examples of data provenance applications, already proposed in the literature, demonstrated the easiness and practicality of our framework. Moreover, some limitations that were revealed under very large and complex systems open room for improving our framework by integrating more powerful dynamic instrumentation tools.

**Acknowledgements.** The project was being co-financed by the European Regional Development Fund (ERDF) and national funds, was a part of the Operational Programme "Competitiveness & Entrepreneurship" (OPCE II), Measure "COOPERATION" (Action I). This work was also supported in part by DARPA Contract FA8650-11-C-7190, NSF Grant CNS-09-14312, FP7-PEOPLE-2010-IOF project XHUNTER and Marie Curie Actions - Reintegration Grants project PASS. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, or the NSF.

## References

1. Sqlite, `http://www.sqlite.org/`
2. Aho, A.V., Kernighan, B.W., Weinberger, P.J.: The AWK Programming Language. Addison-Wesley (1988)
3. Apple. dtrace(1) Mac OS X Manual Page, `http://developer.apple.com/mac/library/documentation/Darwin/Reference/ManPages/man1/dtrace.1.html`
4. Buneman, P., Tan, W.-C.: Provenance in databases. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 2007, pp. 1171–1173. ACM, New York (2007)
5. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proceedings of the USENIX Annual Technical Conference (ATC), pp. 15–28 (2004)
6. Cheney, J., Chiticariu, L., Tan, W.-C.: Provenance in Databases: Why, How, and Where. Foundations and Trends in Databases 1(4), 379–474 (2007)
7. Demsky, B.: Garm: cross application data provenance and policy enforcement. In: Proceedings of the 4th USENIX Conference on Hot Topics in Security, HotSec 2009, p. 10. USENIX Association, Berkeley (2009)
8. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight provenance for smart phone operating systems. In: Proceedings of the 20th USENIX Security Symposium, San Francisco, CA (August 2011)
9. FreeBSD. DTrace – FreeBSD Wiki, `http://wiki.freebsd.org/DTrace`
10. Jones, S., Strong, C., Long, D.D.E., Miller, E.L.: Tracking emigrant data via transient provenance. In: Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP 2011), Heraklion, Greece (June 2011)

11. Kemerlis, V.P., Pappas, V., Portokalidis, G., Keromytis, A.D.: iLeak: A lightweight system for detecting inadvertent information leaks. In: Proceedings of the 6th European Conference on Computer Network Defense (EC2ND), Berlin, Germany, pp. 21–28 (October 2010)
12. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: Practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), London, UK (March 2012)
13. Lakshmanan, G.T., Curbera, F., Freire, J., Sheth, A.: Guest editors' introduction: Provenance in web applications. IEEE Internet Computing 15(1), 17–21 (2011)
14. Linux, O.: Trying out dtrace, `http://blogs.oracle.com/wim/entry/trying_out_dtrace`
15. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005)
16. Margo, D.W., Seltzer, M.: The case for browser provenance. In: Proceedings of the First Workshop on on Theory and Practice of Provenance, pp. 9:1–9:5. USENIX Association, Berkeley (2009)
17. Michaelis, J.R., McGuinness, D.L.: Towards Provenance Aware Comment Tracking for Web Applications. In: McGuinness, D.L., Michaelis, J.R., Moreau, L. (eds.) IPAW 2010. LNCS, vol. 6378, pp. 265–273. Springer, Heidelberg (2010)
18. Muniswamy-Reddy, K.-K., Holland, D.A., Braun, U., Seltzer, M.: Provenance-aware storage systems. In: Proceedings of the Annual Conference on USENIX 2006 Annual Technical Conference, p. 4. USENIX Association, Berkeley (2006)
19. Muniswamy-Reddy, K.-K., Macko, P., Seltzer, M.: Provenance for the cloud. In: Proceedings of the 8th USENIX Conference on File and Storage Technologies, FAST 2010, pp. 14–15. USENIX Association, Berkeley (2010)
20. QNX. The community portal for qnx software developers, `http://community.qnx.com/sf/projects/dtrace/`
21. Spillane, R., Sears, R., Yalamanchili, C., Gaikwad, S., Chinni, M., Zadok, E.: Story book: an efficient extensible provenance framework. In: Proceedings of the First Workshop on Theory and Practice of Provenance, pp. 11:1–11:10. USENIX Association, Berkeley (2009)
22. Theoharis, Y., Fundulaki, I., Karvounarakis, G., Christophides, V.: On provenance of queries on semantic web data. IEEE Internet Computing 15, 31–39 (2011)
23. Viega, J., Messier, M., Chandra, P.: Network security with OpenSSL. O'Reilly Media (2002)