# Looking Inside the Black-Box: Capturing Data Provenance using Dynamic Instrumentation

Manolis Stamatogiannakis, Paul Groth, and Herbert Bos

VU University Amsterdam, The Netherlands,
{manolis.stamatogiannakis, p.t.groth, h.j.bos}@vu.nl

**Abstract.** Knowing the provenance of a data item helps in ascertaining its trustworthiness. Various approaches have been proposed to track or infer data provenance. However, these approaches either treat an executing program as a *black-box*, limiting the fidelity of the captured provenance, or require developers to modify the program to make it provenance-aware. In this paper, we introduce DataTracker, a new approach to capturing data provenance based on *taint tracking*, a technique widely used in the security and reverse engineering fields. Our system is able to identify data provenance relations through dynamic instrumentation of unmodified binaries, without requiring access to, or knowledge of, their source code. Hence, we can track provenance for a variety of well-known applications. Because DataTracker looks inside the executing program, it captures high-fidelity and accurate data provenance.

**Keywords:** data provenance, dynamic, taint analysis, taint tracking, PROV

## 1 Introduction

Provenance is a "record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing" [26]. This record can be analyzed to understand if data was produced according to regulations, understand the decision making procedure behind the generation of data, used in debugging complex scientific programs, or used to make trust calculations [25].

Given the need for an explicit record to analyze, the community has studied a variety of ways to record or capture data provenance ranging from modifying applications, to explicitly recording provenance, to reconstructing provenance from the computational environment. In designing a provenance capture system, one must make a trade-off between the *fidelity* of the captured provenance (i.e. how accurate the provenance is) and the *effort* on the part of application developers and/or users to make a system provenance-aware.

In this work, we introduce DataTracker, a new system for capturing provenance that practically eliminates the effort of making an application provenance-aware while still producing *high-fidelity provenance*. Analogous to high-fidelity sound, we use this term to refer to provenance information with minimal amounts

of noise (false-positives) and distortion (misrepresentation of existing relations). DataTracker offers both these qualities as it *a*) eliminates a large number of false-positives by tracking how data are actually used, and *b*) is able to capture provenance at the byte-level. Our system is based on *dynamic taint analysis* (DTA), a method popular with the security research community, allowing our system to leverage already available infrastructure. It can track data provenance for a wide-variety of unmodified binaries ranging from small command line utilities to full-fledged editors like `vim`. Moreover, unlike other systems that can be used to capture high fidelity provenance, DataTracker does not require knowledge of the application semantics. Concretely, the contributions of this paper are:

1. A system, DataTracker[1], to transparently capture data provenance of unmodified binaries based on DTA.
2. An evaluation of the system that shows high-fidelity provenance capture on small inspectable programs.
3. Case studies of provenance capture for well-known applications.

The rest of the paper is organized as follows. Section 2 discusses previous work on capturing provenance and introduces dynamic binary instrumentation and taint analysis, the techniques we use to implement DataTracker. In Section 3 we present the architecture of our system and detail its implementation. Next, in Section 4 we evaluate the provenance produced by it. We use both simple programs that address cases that are not adequately handled by the state of the art, as well as real applications. Finally, in Section 5 we discuss some of the aspects of DataTracker, highlighting possible follow-up work.

## 2 Background and related work

### 2.1 Capturing provenance

Provenance has been widely studied in the database [5], distributed systems [31] and e-science communities [9]. For a comprehensive overview of the field, we refer the reader to Moreau [24]. Furthermore, Cheney et al. [5] and Simmhan et al. [30] provide specialized reviews for databases and e-science respectively. Here, we focus on systems for provenance capture.

We classify provenance capture approaches on a spectrum in terms of how much intervention they require to make an application provenance-aware. By intervention, we mean the modifications of a program or computational environment to capture provenance. Typically, the more intervention required the higher fidelity of provenance and the greater the required effort is.

At the most detailed level are systems modified to be provenance-aware. For example, Trio DBMS [35] extends a relational database system to cope with uncertain data and provenance. Frameworks for modifying programs to record provenance information have also been proposed [23,20].

An alternative take to provenance-awareness is the use of middleware to wrap applications and components. The provenance is generated by the middleware

[1] The source code of DataTracker is available on: http://github.com/m000/dtracker

after inspection of the inputs and outputs of the wrapped components. This approach is popular with the scientific workflow community and includes systems such as Taverna [29], VisTrails [11], Kepler [2] and Wings [17]. Other middleware-based systems like Karma [31] are not tied to a workflow system, but instead tap into the communication stack to capture provenance.

It has also been proposed to capture provenance by exploiting the mechanisms offered by the operating system to trace the activities of programs. Such systems include TREC [34], ES3 [12] and the work of Gessiou et al. [13]. All these systems operate in user-space and don't require special privileges. A slightly different approach is taken by PASS [14], which has been implemented as a Linux kernel extension. From this vantage point, PASS is able to capture provenance from multiple processes at once. The fidelity of the provenance captured by these systems is comparable, as they all retrieve and use similar information (albeit using different mechanisms) and all of them treat traced programs as *black boxes* without tracking how data are actually processed. We consider our system to be an extension of these approaches to support higher fidelity provenance. From them, DataTracker is mostly related to Gessiou's et al. system, in the sense that both use dynamic binary instrumentation.

Finally, newer work [21,28] does not use a-priori instrumentation but attempts to reconstruct provenance directly from data. Without primary access to the actual provenance, this approach will always suffer from lower fidelity.

### 2.2 Dynamic Instrumentation and Taint Analysis

**Dynamic Instrumentation:** DataTracker applies Dynamic Instrumentation on the executing programs using the Intel Pin [19] framework. Pin allows monitoring and interacting with an executing program using a rich API and provides the base platform for the implementation of *Dynamic Taint Analysis* (discussed next). We picked Pin over similar *Dynamic Binary Instrumentation* (DBI) platforms [27,3] because it is considered the easiest to work with while providing high performance without the need for much manual tinkering. Instrumentation techniques which require modification or recompilation of the instrumented programs [33,18] were precluded.

**Dynamic Taint Analysis:** Pioneered by Denning in the 70s [10], the idea of tracking the flow of data though a program is all but new. The technique has remained relevant through the years and has been implemented on different levels, ranging from source code [22], to interpreters[2], to full emulators [8,1]. Its most common applications are in the field of security and intrusion detection [7,1]. However, until now, it hasn't been used for capturing provenance.

When data flow tracking is applied at runtime, it is generally called Dynamic Flow Tracking or, equivalently, *Dynamic Taint Analysis* (DTA). The term *taint* refers to the metadata associated with each tracked piece of data. A short and concise definition of DTA has been given by Kemerlis et al. [16] as: "the process of accurately tracking the flow of selected data throughout the execution of a

[2] E.g. Perl taint mode: `http://perldoc.perl.org/perlsec.html#Taint-mode`

program or system". The four elements that are define a DTA implementation are: *a*) the *taint type*, which encapsulates the semantics tracked for each piece of data; *b*) the *taint sources*, i.e. locations where new taint marks are applied; *c*) the *taint sinks*, i.e. locations where the propagated taint marks are checked or logged; *d*) a set of *propagation policies* that define how that taint marks are handled during program execution.

Given the effectiveness of DTA, recently much research has been done on reusable DTA frameworks. This was largely made possible by the maturing of dynamic binary instrumentation platforms (see above). Dytan [6] uses the Intel Pin [19] DBI framework and provides much flexibility for configuring taint sources and propagation policies. Additionally, it offers some support for implicit data flows (see Section 5). DTA++[15] by Kang et al. focuses on the efficient handling of such implicit flows in benign programs.

A more recent effort (also based on Intel Pin) which emphasizes on performance is libdft [16]. To achieve superior performance, libdft consciously sacrifices some flexibility by supporting only bit or byte sized taint marks and omitting any support for implicit data flows. DataTracker is based on libdft, however we opted to use a modified version which adds support for arbitrary taint marks.

# 3 System

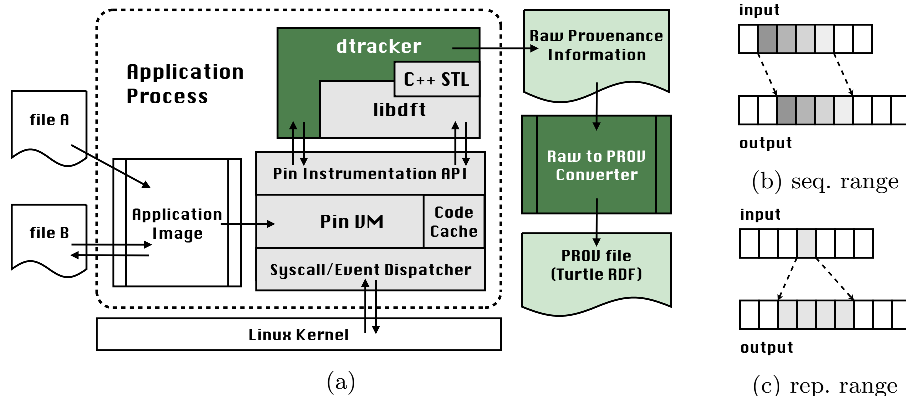
The architecture of DataTracker is illustrated in Fig. 1a. Colored blocks represent the additional components required for capturing provenance information in PROV format from unmodified applications. The darker blocks are those specifically developed for DataTracker. Due to Pin's architecture, application and instrumentation code appear as a single process to the OS and share the same address space. This means that instrumentation code has access to all of the application data and can intercept system-calls made by the application.

## 3.1 Modifications to libdft

A fundamental requirement of DataTracker is the ability to use richer taint marks than those offered by the original libdft. Libdft has been carefully optimized with security applications in mind. For such applications, it has been argued that byte-sized taint marks are large enough for the current crop of security applications based on DTA [6]. So, libdft has limited the size of supported taint marks to either 1b or 1B, which allows for optimizing the taint propagation logic and reducing the memory requirements.

However, the requirements for DTA-based provenance applications are quite different. In this case, the default byte-sized taint marks of libdft just do not provide enough fidelity. In order to accommodate for the higher fidelity we need, we opted to use a modified version of libdft developed at our lab[3]. The modified version shares much code with the original, however the taint mark type and propagation logic can be configured to match the application needs.

[3] Source code available on: `https://git.cs.vu.nl/r.vermeulen/libdft`




(a) (b) seq. range (c) rep. range

Fig. 1: DataTracker architecture (a) and taint ranges (b, c).

For DataTracker, the taint marks associated with each memory location are modeled as set of two-tuples: $\{\langle$`ufd0:offset0`$\rangle$, $\langle$`ufd1:offset1`$\rangle, \ldots\}$. Each of these tuples is 64bit long, and uniquely identifies an offset in a file[4]. The first half of each tuple is a *unique file descriptor* (UFD) which identifies a file during an application session. The second half represents the *offset* of the data within the file mapped to the UFD. Unlike file descriptors provided by the OS, UFDs increase monotonically and are not recycled after closing a file. Thus, they enable us to tell apart data which outlive the file descriptor they were read from. UFDs are only used internally and are resolved back to filenames during the conversion to PROV.

### 3.2 The `dtracker` pin tool and converter

The `dtracker` pin tool is the core component of DataTracker. It implements the following functionality: *a*) identifying when taint should be applied; *b*) properly setting taint marks on data; *c*) logging raw provenance information.

**Identification of data to taint:** When an instrumented program accesses a file for the first time, `dtracker` intercepts the `open()` system call and invokes its *UFD mapper* sub-component. The mapper checks whether the file descriptor returned by `open()` should be watched for input/output operations in order to respectively assign/log taint marks. This check is necessary in order to avoid applying taint on data that are either of no interest or highly unlikely to end-up in the application output. Examples of such files are shared libraries, UI icons, etc. The mapper includes heuristics to identify such files. If the mapper determines that the file descriptor should be watched, it will create a new UFD mapping for it. Additionally, it will check whether the file was created as a result of the system call and if it has been opened for writing. This information is logged and used to avoid generating false `prov:wasGeneratedBy` records.

[4] For simplicity, we prefer the term *"file"* over the more accurate *"file-like resource"*.

**Applying taint marks:** The majority of applications read data from external sources using `read()` and `mmap2()` system calls. The return values and arguments of these calls are intercepted by DataTracker and, if the file descriptor used is watched, taint marks are set on the memory locations where the data were read into. E.g. for a call `read(fd, buf, size)` which returns `n`, DataTracker will assign `tags[buf+i]` $\leftarrow \langle$`ufd[fd]:offset+i`$\rangle$, $\forall$`i` $\in [0,$ `n`$)$. The handling of `mmap2()` is similar. The required `offset` to create the taint mark is acquired by querying the operating system using the `lseek()` system call. For file descriptors where this is not supported (e.g. pseudo-terminal devices), DataTracker keeps separate read/write counters. After the taint marks have been set, their propagation as the program executes is handled by libdft.

**Raw provenance logging and aggregation:** While some pieces of raw provenance are logged by the instrumentation code attached to `open()`, the bulk of logging happens when `write()` and `munmap()` are called. A naive approach for this logging would be to just loop through written buffer and log one entry per tainted memory location. This strategy would easily result in very large log files. Logging large amount of data to disk would also slow-down the execution of the application. To avoid these issues and and produce more compact and meaningful output, `dtracker` includes a simple aggregator for the logged taint marks which condenses logged information into two types of *taint ranges*: *a*) *Sequence ranges* (Fig. 1b), which occur when the same same sequence of consecutive taint marks appears both in the input and the output; *b*) *Repetition ranges* (Fig. 1c), which occur when consecutive output bytes are all marked with the same taint mark. From the supported ranges the most common is the first, which naturally occurs whenever data are moved or copied by the application.

**Raw output to PROV converter:** In order to be able to use existing tools to further process the produced provenance, DataTracker provides a converter from its own raw format to PROV-O, the RDF serialization of PROV. While the bulk of the conversions are simple transformations, the converter script also needs to maintain some internal state, in order to avoid producing false-positives in some specific cases (e.g. false `prov:wasGeneratedBy` triples).

## 4 Evaluation

We carry out a two part evaluation. In the first part, we examine simple baseline programs with transparent and inspectable functionality. The goal is to demonstrate specific cases where our system is able to improve on the quality of produced provenance and produce less false-positives than existing approaches. In the second part, we focus on well-know applications and show how DataTracker can be used to extract useful provenance information from them without requiring modifications. We use the diagrammatic convention of PROV[5]. We have also been able to run bigger applications like AbiWord with DataTracker. However, in this introductory paper we will focus on simpler, more tractable programs.

[5] http://www.w3.org/2011/prov/wiki/Diagrams

```
w ← argv[1].lower();
for i ← 2 to argc − 1 do
    f ← open(argv[i]);
    for ln in f.lines()
    do
        if w in
        ln.lower() then
            print ln;
        end
    end
end
```

(a) **sgrep** pseudocode

```
f ← open(argv[1]);
g ← open(argv[2]);
dummy ← f.readline();
g.write("http://bit.ly/ipaw2014");
```

(b) **tricky** pseudocode

```
for i ← 1 to argc − 1 do
    f ← open(argv[i]);
    f′ ←
    open(argv[i] + ".up");
    while
    (c ← f.getc()) ≠
    EOF do
        if 'a'≤ c ≤'z'
        then
            f′.putc(c +
            'Z' − 'z');
        else
            f′.putc(c);
        end
    end
end
```

(c) **upcase** pseudocode

Fig. 2: Pseudocode for baseline programs.

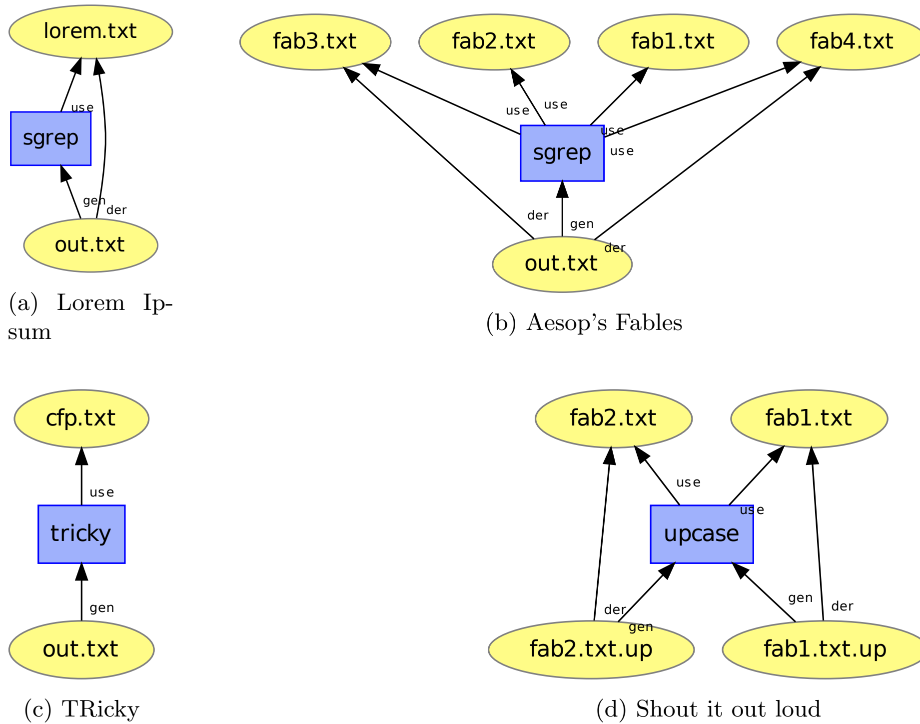



Fig. 3: Output from baseline experiments with DataTracker.

### 4.1 Baseline experiments

**Lorem Ipsum:** In this experiment, we use `sgrep`, a simplified version of the standard `grep` unix utility. It finds lines containing the word $w$ specified as its first argument inside the files specified by the rest of the arguments. The search is case-insensitive and the found lines are printed to the standard output. Its functionality is illustrated as pseudocode in Fig. 2a. We use `sgrep` to find the lines containing word *"dolor"* in a file containing the standard *Lorem Ipsum*[6] passage. The standard output is redirected to file *"out.txt"*.

This test demonstrates that DataTracker is able to produce the same provenance graph as those of techniques like [14,12,13]. In Fig. 3a we can see that DataTracker correctly produces the expected usage and derivation edges. Our system also produces byte level provenance information, which has been omitted from the graph for saving space.

**Aesop's Fables:** Here, the `sgrep` utility is again used. This time, we are looking to find lines containing the word *"lion"* in four files containing Aesop fables. Only two of the four fables actually involve a lion.

We can see in Fig. 3b that DataTracker correctly identifies that the output contains lines (and therefore was derived) from only two out of the four input files. This is an improvement over systems like [14,12,13], which would have also produced false derivation edges for the remaining two files. The reason that DataTracker is able to eliminate these false positives, is that it goes beyond simply tracking how the instrumented program exchanges data with its environment. It actually looks inside the program, a *provenance black box* until now, and determines which of the exchanged data have been used and where.

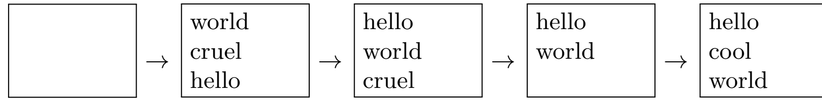
**TRicky:** For this experiment, we use a utility called `tricky` which purportedly scans the input file for urls, and writes them to the specified output file. However, it seems that we've been tricked! In reality, the program always prints the same url regardless of the input it reads, as shown in Fig. 2b.

We ran `tricky` with a call for papers as input that happens to include the exact same string that `tricky` has hardcoded. DataTracker was able to correctly identify that `tricky` generated the output file but its contents actually have nothing to do with the input file (Fig. 3c). Similarly with the previous example, systems that only trace the operations performed by the instrumented program but not the data used would have been tricked into producing a false derivation edge. But in this case, systems that infer provenance by applying content-based heuristics [28] would have also been tricked.

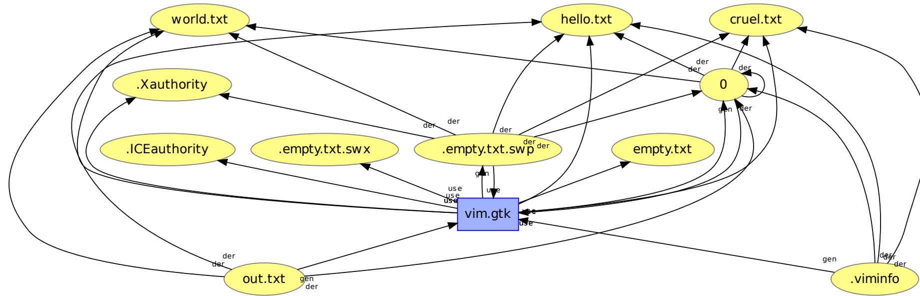
**Shout it out loud:** In this final baseline experiment, we use a utility called `upcase`. This program opens the files specified as arguments and produces one output file for each of them with its contents in uppercase. The pseudocode of `upcase` is shown in Fig. 2c.

This experiment is simply a verification that DataTracker is able to identify the correct derivation edges in the case of multiple inputs and outputs. The generated graph is depicted in Fig. 3d. For `upcase` and $N$ input files, other

[6] A common placeholder text which has been used by typesetters since the 1500s.




(a) `vim` editing scenario steps




(b) provenance graph

Fig. 4: Case study – `vim` editor.

systems would have produced a graph with $N \times N$ edges. Such a result is too vague to be of practical use. With the use of heuristics, the quality of this result could be improved. However, with DataTracker we don't have to resort to heuristics that may fail in other cases.

### 4.2 Case studies

In this section we will present the provenance produced by DataTracker when instrumenting two well-known applications: `vim` editor and Python.

**vim editor:** We used `vim` to run the following editing scenario (illustrated in Fig. 4a):

1. Open *empty.txt* with `vim`.
2. Read *world.txt*, *cruel.txt*, and *hello.txt* into the buffer.
3. Move contents of *hello.txt* to the top of the buffer.
4. Remove contents of *cruel.txt* from the buffer.
5. Type the word *"cool"* in the buffer.
6. Write the buffer to *out.txt* and quit.

The produced provenance graph can is shown in Fig. 4b. We can see that the produced graph is much denser than the ones produced by the baseline programs of Section 4.1. This is because `vim` opens numerous supporting files. We can see that DataTracker correctly didn't produce an $out.txt \xleftarrow{der} cruel.txt$ edge, as the contents of *cruel.txt* were removed in step 3 of the scenario. It also didn't produce an $out.txt \xleftarrow{der} .empty.txt.swp$ edge, even though the contents of *out.txt* were temporarily stored in *.empty.txt.swp* during the session.

Additionally, our system was able to capture provenance attributed to user input. The node labeled "0" in the graph corresponds to the pseudo-terminal

device which is used by the program. We can see that DataTracker correctly produced derivation edges $out.txt \xleftarrow{der} 0$ and $0 \xleftarrow{der} 0$ for it: the former represents the word "cool" we typed, while the latter denotes that whatever we typed was also displayed on the pseudo-terminal. Capturing the user input has remained largely unaddressed by previous work (e.g. [34]). Not only can DataTracker trace provenance back to user's input, but it can also pinpoint which parts of the output were contributed by the user.

**Python scripts:** We used DataTracker to capture the provenance produced by some simple Python scripts in order to test how it performs with interpreted languages. Due to limited space, we will only briefly present our findings. DataTracker was able to produce correct provenance graphs on the file granularity. However, the provenance of some byte ranges was not captured correctly. This can be attributed to *implicit flows*, discussed in Section 5.

# 5 Discussion

The use of DTA allows for tracking of high-fidelity provenance. Following, we discuss some shortcomings of this technique as well as avenues for future work.

**Capturing implicit provenance:** A noteworthy deficiency of DTA is that it cannot easily track *implicit information flows*. An implicit information flow between variables $x$ and $y$ occurs when the value of $y$ is set from a variable/constant $z$ but the execution of the assignment is determined by the value of $x$. This matches cases like conditional assignments (e.g. `if (x) then y=0; else y=1;`) or assignment through lookup tables (e.g. `int v[] = {1, 2, 3}; y = v[x];`). In DTA implementations like libdft [16], where taint marks propagate only through operations directly involving a tainted location, these cases will not result in propagation of taint from `x` to `y`. This problem had already been noted by Denning [10] in her seminal work. The provenance relations that occur as a result of implicit flows are called *implicit provenance*.

Attempting to track implicit flows may result in *over-tainting* and a high number of false-positive, especially when using DTA to analyze malware [32]. For tracking taint through implicit flows in benign programs, Kang et al. propose DTA++ [15]. Their approach uses an offline analysis phase to identify locations where implicit flows occur and cause loss of taint. This is consistent with Cavallaro's observations [4]. However, when using DTA to capture provenance we can safely assume that our programs are benign. So, in principle, techniques like DTA++ could be retrofitted to DataTracker to improve its recall on the retrieved implicit provenance relations.

**Performance:** While performance is acceptable on most command-line programs, issues do exist. E.g. the use of large taint marks may result in increased memory usage. The extent of this effect depends on how much tainted data are used at once. It can be alleviated by attaching to the application after its launch, reducing the amount of un-needed taint applied. We plan to quantitatively study this effect and investigate optimizations to lessen it. Another issue is that DTA is particularly slow when instrumenting interpreted programs (see Section 4.2).

This is because it treats interpreted programs as data and applies taint to them. Investigation of possible solutions to this problem is an area of future work.

## 6 Conclusions

We have presented DataTracker, a novel system for capturing provenance from unmodified binaries based on Dynamic Taint Analysis and implemented using Dynamic Instrumentation. DataTracker advances the state of the art by not treating executing programs as *black-boxes*, inferring provenance by how they interact with their environment, but instead dynamically tracking the flow of data through their internals, capturing high-fidelity provenance along the way. We have shown that DataTracker is able to generate *accurate provenance* in cases where state-of-the-art techniques would have produced false-positives. It is also capable of capturing *user interaction provenance* and generating *high-fidelity provenance* for individual byte ranges within files.

## References


1. Bosman, E., Slowinska, A., Bos, H.: Minemu: The World's Fastest Taint Tracker. In: Proceedings of RAID'11. Menlo Park, CA, USA (2011)
2. Bowers, S., McPhillips, T.M., Ludaescher, B.: Provenance in collection-oriented scientific workflows. Concurr. Comput.: Pract. & Exper. 20(5) (2008)
3. Bruening, D.L.: Efficient,Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. thesis, MIT, Cambridge, MA, USA (2004)
4. Cavallaro, L., Saxena, P., Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment. In: Proceedings of DIMVA'08. Paris, France (2008)
5. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in Databases: Why, How, and Where. Foundations and Trends in Databases 1(4) (2009)
6. Clause, J., Li, W., Orso, A.: Dytan: A Generic Dynamic Taint Analysis Framework. In: Proceedings of ISSTA'07. London, UK (2007)
7. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end Containment of Internet Worm Epidemics. ACM TOCS 26(4) (2008)
8. Crandall, J.R., Chong, F.T.: Minos: Control Data Attack Prevention Orthogonal to Memory Model. In: Proceedings of MICRO 37. Portland, OR, USA (2004)
9. Davidson, S.B., Freire, J.: Provenance and Scientific Workflows: Challenges and Opportunities. In: Proceedings of SIGMOD'08. Vancouver, Canada (2008)
10. Denning, D.E., Denning, P.J.: Certification of Programs for Secure Information Flow. Communications of the ACM 20(7) (1977)
11. Freire, J., Silva, C.T., Callahan, S.P., Santos, E., Scheidegger, C.E., Vo, H.T.: Managing rapidly-evolving scientific workflows. In: Proceedings of IPAW'06. Chicago, IL, USA (2006)
12. Frew, J., Metzger, D., Slaughter, P.: Automatic capture and reconstruction of computational provenance. Concurr. Comput.: Pract. & Exper. 20(5) (2008)
13. Gessiou, E., Pappas, V., Athanasopoulos, E., Keromytis, A., Ioannidis, S.: Towards a Universal Data Provenance Framework Using Dynamic Instrumentation. IFIP Advances in Information and Communication Technology, vol. 376 (2012)

14. Holland, D.A., Seltzer, M.I., Braun, U., Muniswamy-Reddy, K.K.: PASSing the provenance challenge. Concurr. Comput.: Pract. & Exper. 20(5) (2008)
15. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In: Proceedings of NDSS'11. San Diego, CA, USA (2011)
16. 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 (2012)
17. Kim, J., Deelman, E., Gil, Y., Mehta, G., Ratnakar, V.: Provenance Trails in the Wings-Pegasus System. Concurr. Comput.: Pract. & Exper. 20(5) (2008)
18. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of CGO'04. Palo Alto, CA, USA (2004)
19. 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 PLDI'05. Chicago, IL, USA (2005)
20. Macko, P., Seltzer, M.: A General-purpose Provenance Library. In: Proceedings of USENIX TaPP'12. Boston, MA, USA (2012)
21. Magliacane, S.: Reconstructing Provenance. In: Proceedings of ISWC'12. Boston, MA, USA (2012)
22. McCamant, S., Ernst, M.D.: Quantitative information-flow tracking for C and related languages. Tech. Rep. MIT-CSAIL-TR-2006-076, MIT, Cambridge, MA, USA (2006)
23. Miles, S., Groth, P., Munroe, S., Moreau, L.: PrIMe: A Methodology for Developing Provenance-Aware Applications. ACM TOSEM 20(3) (2009)
24. Moreau, L.: The Foundations for Provenance on the Web. Foundations and Trends in Web Science 2(2–3) (2010)
25. Moreau, L., Groth, P.: Provenance: An Introduction to PROV. Synthesis Lectures on the Semantic Web: Theory and Technology 3(4) (2013)
26. Moreau, L., Missier, P.: PROV-DM: The PROV Data Model. Recommendation REC-prov-dm-20130430, W3C (2013)
27. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proceedings of PLDI'07. San Diego, CA, USA (2007)
28. Nies, T.D., Coppens, S., Deursen, D.V., Mannens, E., Walle, R.V.D.: Automatic Discovery of High-Level Provenance using Semantic Similarity. In: Proceedings of IPAW'12. Springer Berlin Heidelberg, Santa Barbara, CA, USA (2012)
29. Oinn, T., Greenwood, M., et al.: Taverna: lessons in creating a workflow environment for the life sciences. Concurr. Comput.: Pract. & Exper. 18(10) (2006)
30. Simmhan, Y.L., Plale, B., Gannon, D.: A survey of data provenance in e-science. SIGMOD Rec. 34(3) (2005)
31. Simmhan, Y.L., Plale, B., Gannon, D.: Karma2: Provenance management for data driven workflows. International Journal of Web Services Research 5(2) (2008)
32. Slowinska, A., Bos, H.: Pointless Tainting?: Evaluating the Practicality of Pointer Tainting. In: Proceedings of EuroSys'09. Nuremberg, Germany (2009)
33. Srivastava, A., Eustace, A.: ATOM: A System for Building Customized Program Analysis Tools. In: Proceedings of PLDI'94. Orlando, FL, USA (1994)
34. Vahdat, A., Anderson, T.: Transparent Result Caching. In: Proceedings of USENIX ATC'98. New Orleans, LA, USA (1998)
35. Widom, J.: Trio A System for Data Uncertainty and Lineage. In: Managing and Mining Uncertain Data, vol. 35 (2009)