PROV$_{2R}$: Practical Provenance Analysis of Unstructured Processes

Manolis Stamatogiannakis*, Elias Athanasopoulos†, Herbert Bos* and Paul Groth‡

*Vrije Universiteit Amsterdam
{manolis.stamatogiannakis, h.j.bos}@vu.nl

†University of Cyprus
eliasathan@cs.ucy.ac.cy

‡Elsevier Labs
p.groth@elsevier.com

Abstract

Information produced by Internet applications is inherently a result of processes that are executed locally. Think of a web server that makes use of a CGI script, or a content management system where a post was first edited using a word processor. Given the impact of these processes to the content published online, a consumer of that information may want to understand what those impacts were. For example, understanding from where text was copied and pasted to make a post, or if the CGI script was updated with the latest security patches, may all influence the confidence on the published content. Capturing and exposing this information provenance is thus important in order to ascertaining trust to online content. Furthermore, providers of internet applications may wish to have access to the same information for debugging or audit purposes. For processes following a rigid structure (such as databases or workflows), disclosed provenance systems have been developed that efficiently and accurately capture the provenance of the produced data. However, *accurately* capturing provenance from *unstructured* processes, e.g. user-interactive computing used to produce web content, remains a problem to be tackled.

In this paper, we address the problem of capturing and exposing provenance from unstructured processes. Our approach, called *PROV*$_{2R}$ (*PROV*enance *R*ecord and *R*eplay) is composed of two parts: 1) the decoupling of provenance analysis from its capture; and 2) the capture of high fidelity provenance from unmodified programs. We use techniques originating in the security and reverse engineering communities, namely, *record and replay* and *taint tracking*. Taint tracking fundamentally addresses the data provenance problem, but is impractical to apply at runtime due to extremely high overhead. With a number of case studies we demonstrate that *PROV*$_{2R}$ enables the use of taint analysis for high fidelity provenance capture, while keeping the runtime overhead at manageable levels. In addition, we show how captured information can be represented using the W3C PROV provenance model for exposure on the Web.

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" [Moreau and Missier 2013]. One can analyze this record to understand if a web page uses correctly licensed content, to understand the decision making procedure behind a figure or to help debug complex programs, [Moreau and Groth 2013]. As [Groth et al. 2009] argued, understanding the provenance of data created by internet applications is an important factor in establishing trust.

1.1 Unstructured processing

While provenance can provide an important signal for trust, modifying applications to *disclose* provenance is a major undertaking. For applications (e.g. workflow systems [Oinn et al. 2006]) that interact with the world and behave in a well-structured way or for use cases that require it (e.g. climate change modeling [Ma et al. 2014]), this is often a worthy investment.

However, information is predominantly processed by off-the-shelf applications. The developers/vendors of such applications have very little incentive to make them provenance-enabled. Moreover, applications may be extended or modified by third parties, making matters worse. E.g. a web server may support extensions that allow using different programming languages to dynamically produce

content. More importantly, when the processing of information includes user interaction, an additional factor of unpredictability is introduced: For the same given task, different users may use different information sources, a different set of tools (often not known a-priori), or even similar tools but in a different combination. We call this type of computing where (a) arbitrary off-the-shelf applications are used and (b) applications may be freely combined in order to perform a specific task, *unstructured processing*.

When it comes to internet content, unstructured processes play an important role both in the production as well as the delivery of that content. E.g. a document may be created using MS-Word using local and online sources, perhaps processed by other programs beforehand. Or content may be served and updated by nginx server running PHP scripts through CGI.

1.2 Capturing provenance from unstructured processes

System events based provenance analysis

While provenance of online content can provide an important signal for trust, the nature of unstructured processing makes it difficult to capture it. An idea that has been extensively explored in to address this problem is deriving provenance by analyzing events collected by existing mechanisms of the operating system [Frew et al. 2008; Holland et al. 2008; Gehani and Tariq 2012; Gessiou et al. 2012; Pohly et al. 2012; Bates et al. 2015; Chirigati et al. 2016]. The main benefits of this approach are that *a*) these mechanisms are transparent to the tracked programs, so there is no need to spend effort to make programs provenance-aware and *b*) the mechanisms tend to be lightweight, so they incur only a moderate execution overhead. While some of these systems include support to redo a given process (e.g. ReproZip [Chirigati et al. 2016]), they are lacking information to perform detailed provenance analytics for the entire system execution.

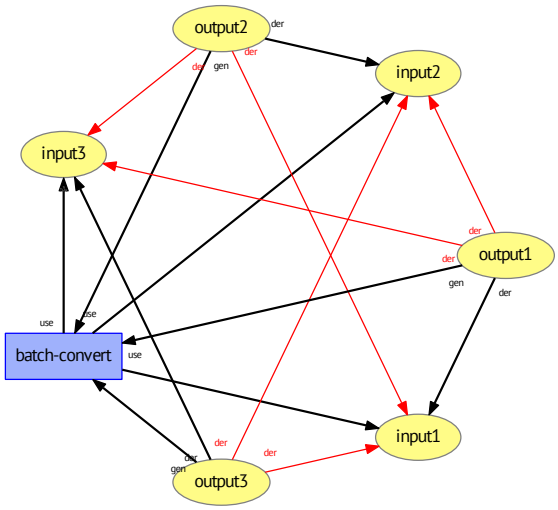

Fig. 1: The n-by-m problem for $n = 3, m = 3$. Each output file is produced using one input source. However, $n \cdot m$ derivation edges are produced. The red edges are *false positives*.

This is the primary downside of system-event based approaches: only limited fidelity provenance is gathered. For the case of provenance, high fidelity means a) low false positives ratio, and b) accurate representation of provenance relations. The low fidelity of the captured provenance can be attributed to the fact that system-event based methods treat traced applications as *black boxes* – they trace their interaction with the operating system but not how data are actually processed.

A particular manifestation of this, is what Carata et al. termed as *the n-by-m problem* [Carata et al. 2014]: When a process reads n input files, outputs m files, without knowledge of what is going on inside of the process, *every* output will file will be attributed to *all* input files, thus producing $n \cdot m$ derivation edges in the provenance graph. Fig. 1 illustrates this problem. There are two broad solutions to this problem [Carata et al. 2014]: one is specifically instrumenting the program to output semantically correct relationships; and the second is to employ binary instrument techniques such as taint tracking.

Taint-analysis for provenance

Pioneered by Peter and Dorothy Denning in the 70s [Denning and Denning 1977], tracking the flow of data though a program is and old but recurring technique. The technique is also commonly referred to as *Taint Tracking* or *Taint Analysis* because of its popularity in security applications, where the tracked data are considered tainted. Taint analysis boils down to tracking how data propagate and affect the operation of a system during a program's execution. Hence, it can *fundamentally address the*

problem of capturing provenance from unstructured processes. If we track the flow of data through the execution of a program, we can tell the origin of each piece of data produced by it.

The idea of using taint analysis to capture provenance was first explored with DataTracker [Stamatogiannakis et al. 2014]. It was demonstrated that dynamic taint analysis can effectively address the n-by-m problem, producing high fidelity provenance with no developer instrumentation overhead. Even so, the runtime overhead of DataTracker is very steep for the tool to be considered for practical use [Stamatogiannakis et al. 2016]. Moreover, optimizations that have been previously explored for security applications of taint analysis [Bosman et al. 2011; Kemerlis et al. 2012] are not apropos to provenance applications. This is because while security applications can deliver results by tracking only a few bits of metadata [Clause et al. 2007], provenance requires more detailed tracking.

Execution record and replay

Deterministic execution record and replay is a much more recent concept compared to taint analysis [Dunlap et al. 2002]. The technique captures a trace of the entire execution trace of a program or system, and allows replaying it elsewhere for analysis. Deferring analysis to the replay allows heavyweight techniques such as taint analysis to be applied without obstructing the original execution much. Moreover, it allows deciding the type of the analysis a-posteriori. These properties make deterministic record and replay an excellent match for analyzing provenance of unstructured processes. E.g. one can use system-events based provenance analysis, or taint analysis, or both, depending on the case.

Decoupling of provenance analysis from execution was (to our best knowledge) first explored in [Stamatogiannakis et al. 2015] and later in [Ji et al. 2016]. The difference between these two approaches is that the first uses full-system record and replay, while the latter uses process-based execution record and replay. While applying taint analysis was possible for both systems, both narrowed their scope to system-events analysis only. In this work, we aim to fill this gap and complete the picture of decoupled provenance analysis.

1.3 Contribution of this work

Overall, we summarize the key issues for capturing provenance from unstructured processes as follows:

1) There is 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 to make a system provenance-aware.
2) Provenance is secondary to the function of the application. Performance of the primary function of the application has to be maintained at adequate levels.
3) It is difficult to decide *a priori* what provenance to capture. Developers need to ensure that enough information is captured to allow for all relevant future analysis. If a developer fails to foresee the need for a particular kind of provenance data (say, the loading of an application's libraries) and does not capture it, subsequent analysis that requires it may be difficult or impossible. On the other hand, opting for a more detailed analysis, "just to be on the safe side", may impose an overhead that is not acceptable to the user.

In this work, we begin to tackle these difficulties by applying, in combination, two techniques from the security and systems communities: taint-tracking and record and replay. We build upon our prior work on using taint analysis for provenance [Stamatogiannakis et al. 2014] and record and replay for provenance [Stamatogiannakis et al. 2015] by tying them together into the *PROV*$_{2R}$ system. This combination is an important addition as it addresses deficits in both prior approaches.

Overall, we aim to use taint tracking to attack the n-by-m problem and low fidelity provenance, while employing record and replay to ensure high performance capture. Record and replay has, as we will show, the additional benefit that it enables provenance capture to be performed after the fact when analysis needs are better known. Our system is based on the PANDA framework (see §3.2) and uses components developed by us as well as the wider community.

Concretely, the contributions of this paper are as follows:

- *PROV*$_{2R}$ – a system that combines taint tracking and record and replay for provenance capture;
- evidence of the effectiveness of record and replay as a substrate for provenance capture from unstructured processes;

- an evaluation of our approach for two internet application scenarios;
- an evaluation of the performance and space requirements of record and replay in the context of provenance using the UnixBench benchmark.

The rest of the paper is organized as follows. We begin by providing some background on taint analysis and execution record and replay in §2. This is followed by a system description in §3. We then present two case studies in §4, where we evaluate the functionality of $PROV_{2R}$. Finally, we provide a benchmark based performance evaluation in §5 and conclude.

2 Background

$PROV_{2R}$ builds on two fundamental technologies for providing system-level provenance, namely *taint analysis* and *record and replay*. Taint analysis is a core methodology for tracking how information disseminates in a system, while record and replay allows for an on-demand execution of the analysis. This significantly drops the cost of using taint analysis in comparison to systems that apply it "inline", as the system executes. In this section, we provide a short background of how these two technologies work. We stress here that $PROV_{2R}$ is the first system to build on taint analysis and record and replay for provenance. Therefore, we do not attempt any direct comparison of our system with related systems that rely on taint analysis or record and replay for delivering other applications. Our aim is primarily at introducing readers coming from different backgrounds to these two technologies, highlighting their challenges and practical benefits, and emphasizing on the key characteristics that make $PROV_{2R}$ a practical solution for collecting provenance from unstructured processes.

For a comprehensive overview of data provenance research, we refer the reader to Moreau [Moreau 2010], as well as Cheney et al. [Cheney et al. 2009] for databases, and Simmhan et al. [Simmhan et al. 2005] for provenance in e-science.

2.1 Taint analysis

A short and concise definition of taint analysis has been given in [Kemerlis et al. 2012] as: "the process of accurately tracking the flow of selected data throughout the execution of a program or system". The four elements that define a taint analysis 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. In addition to its applications in security and intrusion detection [Portokalidis et al. 2006; Costa et al. 2008; Bosman et al. 2011], the technique has also been used for information leak detection [Masri et al. 2004], and more recently, for capturing provenance [Stamatogiannakis et al. 2014].

The main limitation of taint analysis is that it imposes extremely high execution overheads. As a result, a recurring topic of the research around taint analysis, is how to optimize it. Minemu [Bosman et al. 2011] and libdft [Kemerlis et al. 2012] take a systems-based approach, investing in taking maximum advantage of the underlying hardware registers and caching mechanisms. Other approaches include static analysis of programs to extract information that would help accelerating taint analysis [Saxena et al. 2008] or using spare cores to parallelize program execution with taint analysis [Jee et al. 2013]. An alternative path in making taint analysis affordable is to only enable it on demand [Fetzer and Süßkraut 2008; Cavallaro and Sekar 2011; Ma et al. 2016]. This however unavoidably leads to unexpected performance degradation when taint analysis is turned on. Moreover, the effectiveness of such systems is limited by the detectors they use to turn taint analysis on. An important part of the overhead depends on the type of taint being tracked. It has been argued that for many security applications, byte-sized taint marks are large enough [Clause et al. 2007].This observation leaves room for optimizations like those in [Bosman et al. 2011; Kemerlis et al. 2012], while sacrificing the flexibility to use the system for other applications.

Through the years, taint analysis has been implemented on different abstraction levels, ranging from source code [McCamant and Ernst 2006], to interpreters[1], to libraries [Clause et al. 2007; Kang

1. E.g. Perl taint mode: http://perldoc.perl.org/perlsec.html#Taint-mode

et al. 2011; Kemerlis et al. 2012], to process emulators [Bosman et al. 2011], and even full system emulators [Crandall and Chong 2004; Portokalidis et al. 2006]. One take-away from this trend is that, depending on the platform you are using, re-implementing taint analysis is often preferred over trying to reuse existing implementations.

2.2 Record and Replay

Recording and replaying of full system executions became popular in the early 2000s with applications almost exclusively related to debugging and security/intrusion analysis [Dunlap et al. 2002; Xu et al. 2003]. After all, the ability to replay an execution with limited overhead is ideally suited to finding rare and non-deterministic error conditions and for deep analysis of attacks. The functionality was popularized when VMware started shipping it in some of its products [Xu et al. 2007]. This led to further research and improvements [Chow et al. 2008, 2010]. Aftersight [Chow et al. 2008] was the first framework to demonstrate the benefits of decoupled analysis using the new feature. Unfortunately, VMware discontinued the feature a few years later due to lack of resources for further development[2].

Taint analysis has always been a popular target for the decoupling capabilities offered by record and replay. Paranoid Android [Portokalidis et al. 2010] used record and replay in order to use taint analysis for intrusion detection on mobile phones. DiskDuster [Bacs et al. 2012] explored record and replay based intrusion recovery. Both systems used custom QEMU-based [Bellard 2005] record and replay implementations. PANDA [Dolan-Gavitt et al. 2015] streamlined record and replay for QEMU, adding a rich API and providing a stable research platform to the community. This led to a string of publications that use PANDA for diverse purposes [Dolan-Gavitt et al. 2013; Whelan et al. 2013; Dolan-Gavitt et al. 2016; Stamatogiannakis et al. 2015]. To our best knowledge [Stamatogiannakis et al. 2015] and *PROV*$_{2R}$ (its follow-up) are the first provenance systems utilizing full system execution record and replay.

A different approach to record and replay is applying the technique to specific processes, rather than the whole system. Platforms that implement this include PinPlay [Patil et al. 2010], Mozilla-rr[3] and Arnold [Devecsery et al. 2014]. We believe that process-level record and replay is not as a good match for capturing provenance from unstructured processes as its full-system counterpart. The reason is that one needs to know upfront which processes need to be recorded, which preculdes finding unknown connections between processes. However, systems based on process-level record and replay, such as RecProv [Ji et al. 2016], may be easier to employ for occasional provenance analysis.

Finally, another line of research on record and replay is the application of the technique on multiprocessors. This has been explored by systems such as ReEmu [Chen and Chen 2013], ReSeer [Wang et al. 2016] and Samsara [Ren et al. 2016]. However, the focus of these works is addressing the intricacies of multiprocessor record and replay rather than providing a platform for further use.

3 System description

We implement a provenance capture and analysis system based on record and replay where provenance capture is decoupled from application runtime. Instead, provenance capture is performed post-hoc on an execution trace based on the needs of the analysis user. We now describe our analysis methodology and its realization.

3.1 Analysis methodology

PROV$_{2R}$ is based upon the 4-stage methodology developed in [Stamatogiannakis et al. 2015]:

1) **Execution Capture**: At runtime, we capture a self-contained, replayable execution trace.
2) **Application of instrumentation**: In this stage, depending on goal of the analysis, we select an instrumentation plugin to process the execution trace and generate a provenance graph.

2. See: http://www.replaydebugging.com/2011/09/goodbye-replay-debugging.html
3. See: http://rr-project.org/

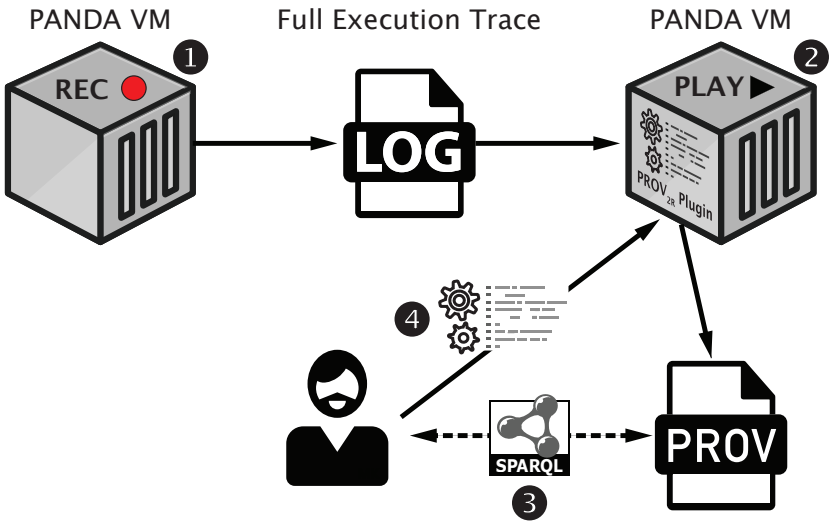

Fig. 2: Workflow overview of $PROV_{2R}$: (1) an execution trace is recorded, (2) an analysis plugin is run over the trace – generating provenance, (3) an analyst queries the provenance for interesting information, (4) the analyst starts another round of analysis using a different plugin.

3) **Provenance analysis**: In this stage, the user interrogates the provenance graph using a query language to focus on, and/or select portions of, the graph.
4) **Selection and iteration**: Based on the provenance analysis, the user can select a portion of the execution trace on which to apply additional, more intensive, instrumentation. To do this, the user starts again from stage 2.

We implement the methodology as a set of *loosely coupled analysis modules* built on top of the PANDA framework, which we describe in more detail below. This approach matches the iterative nature of the methodology better than implementing it as a monolithic provenance analysis framework. The combination of these modules, PANDA, and output to standards formats makes up $PROV_{2R}$. In Fig. 2 we show how our methodology is implemented in $PROV_{2R}$.

We should stress that at the time of writing, PANDA was the only full-system record and replay platform we knew that offered *a*) a maintained codebase; *b*) an API to work with for the analysis part; *c*) an active community around it. Other systems we took into consideration were Aftersight [Chow et al. 2008] and Samsara [Ren et al. 2016]. However, the first is based on the discontinued VMware record and replay implementation, while the latter had just been released and doesn't offer a documented API to work with. We now describe our implementation in more detail.

3.2 The PANDA analysis framework

The *Platform for Architecture-Neutral Dynamic Analysis* (PANDA) [Dolan-Gavitt et al. 2014, 2015] is an open source framework for full-system analysis based on execution record and replay. It is based on the QEMU emulator [Bellard 2005]. PANDA recording works by taking a snapshot of the guest memory and subsequently recording all the non-deterministic inputs to the guest CPU. This *non-determinism trace* contains: (i) data entering through the virtual CPU port inputs, (ii) data written to memory through DMA, (iii) hardware interrupt data. The recorded information enables the *deterministic replay* of all the execution at any later time. We refer the reader to §2.1 in [Dolan-Gavitt et al. 2015] for further details on the internals of PANDA. There are two important implications stemming from this design:

1) *PANDA execution traces are self-contained*. This means that unlike other record and replay systems (e.g. [Ren et al. 2016]), PANDA traces can be replayed and analyzed without requiring access to the VM image used to record them. This greatly simplifies the management and storage of the traces, as each trace is essentially a standalone execution artifact.
2) *PANDA cannot "go live" during the replay phase.* I.e., it is not possible to alter the state of the VM during replay and then resume the VM in order to explore a different execution path. The

reason behind this is that in addition to the emulated CPU, QEMU (on which PANDA is based) also provides emulated hardware devices such as disk controllers or network interfaces. PANDA does not record the state of the emulated devices, as this was not deemed a priority for a system designed as a replay-based analysis framework.

One key feature of PANDA is its plugin architecture which allows writing analysis modules in C and C++. The core of PANDA only deals with the implementation of the record and replay features and the plugin hooking mechanism. Analysis functionality is always encapsulated within plugins. PANDA allows plugins to insert instrumentation at different granularities: per instruction, per memory access, per context switch etc. More importantly, PANDA offers a framework for the plugins to interact with each other. This allows implementing complex analysis functionality by composing it from several smaller plugins. This approach follows the Unix philosophy of relying on combining a plethora of small tools, rather than building monolithic be-all, end-all tools. PANDA allows plugins to either invoke functionality of other plugins through *API calls*, or register plugin-specific *callbacks*. The specifics of the PANDA plugin mechanism are detailed in §2.2 of [Dolan-Gavitt et al. 2015].

3.3 The *PROV$_{2R}$* plugins

As we mentioned earlier in §3.1, *PROV$_{2R}$* is a collection of *loosely coupled* analysis modules that can be used for provenance analysis. The modules have been implemented as PANDA plugins and take advantage of the facilities provided by the PANDA core for plugin-plugin interaction. From the plugins detailed in the following sections, `taint2` and `file_taint` have been developed by the PANDA community, while the rest have been developed by us. Source code for all plugins we used is available online[4].

3.3.1 The Linux OSI plugin

A key step in capturing provenance from a VM is the extraction of OS-level semantics like processes, files, libraries etc. from it. PANDA only offers a low-level Virtual Machine Introspection (VMI) interface, allowing us to peek into the registers and the memory of the VM. This leaves a semantic gap that we need to fill in order to achieve full Operating System Introspection (OSI). A simple approach to this problem would be to use a guest-OS driver to export the desired information to the hypervisor. This approach is common in commercial hypervisors (e.g. VMware) and for closed-source operating systems (e.g. TEMU Windows OSI [Yin and Song 2010]). However, since PANDA can't "go live" during replay (see §3.2), the technique is not applicable for *PROV$_{2R}$*.

For this, we implemented the Linux OSI module for *PROV$_{2R}$* by reconstructing the guest-OS semantics purely from the VM hardware state, using the PANDA VMI. This approach has been employed by several systems in the past, like TEMU [Yin and Song 2010], DroidScope [Yan and Yin 2012], VMWatcher [Jiang et al. 2007], etc. Because the exact layout of the Linux kernel data structures is determined at compile time, we had to implement a `kernelinfo` module to extract the kernel offsets we need from the guest-OS. This step only needs to be run once whenever a new kernel is installed. Knowing these offsets, one can subsequently traverse the Linux kernel data structures and extract the desired runtime information from the guest OS.

Additionally, our plugin implements two new PANDA callbacks that aim to simplify further analysis. The callbacks allow other plugins to register analysis functions that will be called when a new process is created or when a process is terminated. To achieve this, Linux OSI plugin registers a context switch callback with PANDA[5]. Then, the new and terminated processes can be calculated respectively as $P_{new} = P_{cur} \setminus P_{prev}$ and $P_{term} = P_{prev} \setminus P_{cur}$, where P_{cur} is the current process set and P_{prev} the process set at the previous context switch.

4. https://github.com/m000/panda/tree/prov2r/

5. Process creation and termination always coincides with a context switch. For IA32/IA32E architectures, the context switch concludes with a write to the page directory base register (CR3). PANDA context switch callbacks run whenever CR3 is written to.

3.3.2 The PROV-Tracer plugin

PROV-Tracer essentially implements system events based provenance analysis for PANDA. The analysis is similar to other state-of-the-art systems (e.g. [Holland et al. 2008; Gessiou et al. 2012; Pohly et al. 2012]), with the difference that it is implemented at the hypervisor level. Despite its limitations system events based analysis is an important first step when applying our methodology (see §3.1), as it provides quick results that help us to guide further analysis. PROV-Tracer taps on the Linux OSI plugin we described above, using both its introspection functionality and callbacks. Its operation can be summarized as:

1) Upon starting, registers to the Linux OSI plugin for receiving notifications on the creation/destruction of processes. It also registers to PANDA for notifications about context switches as well as notifications for the execution of `SYSENTER` and `SYSEXIT` instructions.
2) Keeps track of the active process on each context switch. This allows associating system calls with processes.
3) When a system call occurs, it associates it with the current process and decodes its arguments. Arguments such as system call flags are decoded by the plugin itself. On the other hand, file descriptors need to be translated to file-names by the Linux OSI plugin.
4) Does book-keeping of the file usage (creation, reads, writes etc.) and emits provenance information when needed. False positives are avoided as much as possible. E.g. if a file is opened but never read, it will not be mentioned as used by the process in the emitted provenance.

The system calls decoding is implemented by extracting from the Linux source code the name, number of arguments, and type of arguments for each system call. These are translated into a dynamic library and loaded when the plugin is bootstrapped. Using this information, PROV-Tracer can use the PANDA VMI interface to provide the proper semantics for each `SYSENTER` and `SYSEXIT` instruction that is executed.

PROV-Tracer emits provenance in a compact intermediate format. This choice was made to reduce the complexity of the required book-keeping. E.g. duplicates may arise because a process opens and reads from a file multiple times. It is easier to eliminate these duplicates after PROV-Tracer completes its analysis, rather than doing on-line deduplication of the output. A python script (`raw2ttl.py`) is used to convert the provenance to W3C PROV [Groth and Moreau (eds.) 2013].

W3C PROV is a standard for provenance interchange published the World Wide Web Consortium. Production of W3C PROV compatible provenance is useful as it allows one to take advantage of existing tooling for manipulating provenance ranging from provenance specific visualizations [Kohwalter et al. 2016] to generation of natural language explanations [Richardson and Moreau 2016]. At the time of standardization, there were over 60 implementations that supported PROV [Huynh et al. 2013]. Moreover, this enables the provenance generated by $PROV_{2R}$ to interoperable with provenance generated by other more domain specific tooling (e.g. neuroscience [Keator et al. 2013]). W3C PROV specifies a number of serialization formats including XML and RDF. We chose RDF and its Turtle syntax as an output format as it is easily usable with command line tools. We refer the reader to W3C PROV-O specification for more details [Lebo et al. 2013].

3.3.3 The `taint2` plugin

The `taint2` plugin [Whelan et al. 2013] serves as the core for taint analysis in the PANDA framework. From the four components of a taint analysis system (see §2.1), `taint2` plugin defines and implements the taint type and the taint propagation policies. Additionally, it provides an API for other plugins to implement the remaining two components—taint sources and sinks. The API is used to apply taint on memory locations and registers, as well as to query for it. Taint is applied on byte-level granularity and is stored in a *shadow memory* map, similar with other taint analysis implementations [Kemerlis et al. 2012]. To illustrate how `taint2` and its supporting plugins work, we will use the notation in TABLE 1.

The taint type used by `taint2` is a *set of integers*. As `taint2` is meant to be a reusable taint analysis framework, no particular meaning is ascribed to the taint marks. This is left for the plugins that use `taint2` through its API. This contrasts with other taint analysis implementations, where

Symbol	Meaning
M / T / F	Memory / Shadow Memory / File
$M[n]$ / $T[n]$ / $F[n]$	The n-th value held in Memory / Shadow Memory / File.
$\{a, b, \ldots, n\}$	Set of values.
$\oplus$	Arithmetic operator.
$\alpha \leftarrow \beta$	Assignment of value from β to α.
$A \implies B$	Implication of B, given A.
$\alpha \underset{der}{\rightarrow} \beta$	Provenance derivation: α derives from β.

TABLE 1: Notation used for `taint2`, `file_taint`, `file_taint_sink` plugins.

the taint type is often chosen to reflect the semantics of a specific application. E.g. Argos [Portokalidis et al. 2006] associates a single bit with each byte in memory, which is adequate to indicate whether the byte originated from the network. For provenance analysis, DataTracker [Stamatogiannakis et al. 2014] uses a set of ⟨src:offset⟩ tuples, in order to attribute provenance to multiple locations and sources. The integer sets used by `taint2` allow for powerful analyses to be performed, without making taint operations unnecessarily complicated. For cases where a more powerful abstraction is required, it is straightforward to map the integer sets handled by `taint2` to a richer data type. This mapping should happen at the the plugins that implement the taint sources and sinks, which are the ones that need to interpret the taint marks.

With regard to the taint propagation policy, `taint2` always propagates taint on direct assignments and arithmetic operations. This is the standard baseline for taint propagation across all taint analysis systems. In addition to the above, `taint2` can also be configured to propagate taint on pointer dereferencing. Whether it should be enabled is an open discussion [Slowinska and Bos 2009; Dalton et al. 2010] and largely depends on the goals of the analysis being performed. Using the notation in TABLE 1, we could summarize taint propagation in `taint2` as:

$$M[x] \leftarrow M[y] \implies T[x] \leftarrow T[y]$$
$$M[x] \leftarrow M[y_1] \oplus M[y_2] \implies T[x] \leftarrow T[y_1] \cup T[y_2]$$
$$M[x] \leftarrow M[M[y]] \implies T[x] \leftarrow T[y] \cup T[T[y]] \quad \textit{(optional)}$$

The plugin is implemented by translating the low-level TCG code [Bellard 2005] executed by QEMU to LLVM IR code [Lattner and Adve 2004] and inserting the taint propagation operations there. The resulting IR code is finally executed by the LLVM JIT compiler. This implementation decision makes the `taint2` plugin target-agnostic, as instrumentation is applied in higher-level LLVM instructions rather than the instructions of the ISA emulated by QEMU. For more implementation details of the `taint2` plugin, we refer the reader to §4 in [Whelan et al. 2013].

3.3.4 Taint source plugin

As we mentioned previously, the PANDA `taint2` plugin is only responsible for maintaining and propagating the taint metadata during execution. Injection and inspection of taint is left to be implemented by other plugins that use the `taint2` API. The `file_taint` plugin is used in *PROV*$_{2R}$ as the taint source for the taint analysis. The plugin injects taint to data read from the VM disk. Currently, for each run the plugin applies taint to bytes coming from one file, configurable at runtime. Because data from a single file are tracked, there is no need to map the integers contained in each taint mark to a more complex data type. The integers correspond directly to offsets within the file. Using the notation in TABLE 1, when reading from input file F, `file_taint` applies taint as following:

$$M[x] \leftarrow F[y] \implies T[x] \leftarrow \{y\}$$

The current capabilities of the `file_taint` plugin have proved adequate for our case studies (see §4). In our second case study (see §4.2), where we need to track taint from multiple sources, we repeat

the analysis once per source. In more complex scenarios where multiple sources need to be processed concurrently, a layer mapping the integers tracked by `taint2` to a more complex data type can be added.

`file_taint` depends on the Linux OSI plugin to reverse-map file descriptors to file paths and determine whether taint has to be applied on the data read. As we have mentioned in §3.3.1, the reverse mapping is achieved by traversing the kernel data structures in memory. This way, both regular files and pseudo-files (e.g. `/dev/ttyX`) are supported[6]. Additionally, the `syscalls2` plugin is used in order to hook on specific file-related system calls (`open`, `read`, etc.). The decoding of system calls by `syscalls2` works similar with PROV-Tracer. The reason of this duplication of functionality is that `syscalls2` and PROV-Tracer were developed simultaneously but completely independently.

3.3.5 Taint sink plugin

As a taint sink for $PROV_{2R}$ we implemented and used the `file_taint_sink` plugin. Until now taint analysis in PANDA had only been used by plugins that were analyzing the influence of tainted data to program control flow, so there was no plugin for inspecting the taint of bytes written to a file. Similar with `file_taint`, it uses the APIs provided by the `taint2`, Linux OSI and `syscalls2` plugins. However, `file_taint` can log taint for multiple files, specified at runtime.

If F is the tracked input file and F' is an output file, then the following derivations can be made using the logged taint:

$$F'[x] \leftarrow M[y] \implies \begin{cases} F' \underset{der}{\rightarrow} F \\ F'[x] \underset{der}{\rightarrow} F[z] \end{cases} \quad \forall z \in T[y], T[y] \neq \varnothing$$

In addition to logging, the plugin keeps track of the *taint count number*, which tells us how many tainted bytes were written to the file. This is a simple metric which can be used to tell us how much the output file was affected by the input, without looking on the written actual data. As such, it can help filter-out false-positives in a provenance graph created because of the n-by-m problem.

3.4 User-driven Provenance Analysis

As discussed previously, provenance produced by the system is represented using the W3C PROV recommendations [Groth and Moreau (eds.) 2013] and serialized using Turtle RDF. This enables us to leverage existing Semantic Web infrastructure for provenance analysis. The integration with this generic infrastructure is done by specifying query result formats for obtaining parameter values that can then be fed back into the replay and instrumentation system (i.e. PANDA). The use of SPARQL in combination with W3C PROV provides a full featured language for users to interrogate and analyze the provenance produced by PROV-Tracer.

In the case study that follows we employed a combination of the Redland RDFLib Rasqal command line tool[7] for performing SPARQL queries in conjunction with other Unix command line tools.

4 Functional evaluation

In this section, we demonstrate how offline taint analysis can be used to provide high-fidelity provenance.

Broadly, the scenarios described below use the following pattern. *Alice* and *Bob* are responsible for updating the website of *example.org*. The typical web update workflow is used: *a*) the document and its associated resources are downloaded locally (HTTP GET), *b*) the document is altered using various editing tools, *c*) the updated document and resources are re-uploaded (HTTP POST). We should note that this workflow is still at the core of web publishing, although it usually remains hidden under

6. Reverse mapping of file descriptors corresponding to network connections is not currently supported. In future, it can be implemented by extending the Linux OSI plugin.

7. See: http://librdf.org/rasqal/

several abstraction layers. E.g. when a page is updated using a web-based editor there is the illusion that the document is edited "online". But under the hood, it is still a series of GET and POST requests that are responsible for the update. Only the editing tools are different.

Because we are focused on the use of *PROV*$_{2R}$ for provenance analysis, we assume for simplicity that both Alice and Bob are working on a shared machine.[8] We discuss the applicability to a distributed setting below in §4.3.

To implement the scenarios, we used a single core PANDA virtual machine running Debian 7. The machine was allocated 512MB of memory and was up to date with the latest versions of the packages of the specific release. The version of the Linux kernel is 3.2. Moreover, the nginx 1.2.1 was installed for serving web pages. PHP was enabled through the php5-fpm backend.

4.1 Web document update

In our first case study, we consider the following scenario:

1) Alice downloads the front page of example.org.
2) Alice edits the document and *fixes* a link that points to the wrong page.
3) Alice re-uploads the HTML document and the image.
4) Bob downloads the front page of example.org.
5) Bob *removes* a paragraph of text.
6) Bob re-uploads the the HTML document.

Chief editor Ed reviews the page at the end of the day. He wants to know what changes have been made and who made them. For this, he requests an analysis of the recorded execution trace of the past day.

4.1.1 Initial Analysis

To answer this query, one first runs a system-events based provenance analysis on the execution trace (`scenario1.et`) using the PROV-Tracer plugin (see §3.3.2). Thus, a raw provenance trace is obtained which is then converted to W3C PROV.

```
$ qemu -replay scenario1.et -panda "osi;osi_linux;prov_tracer"
$ ./raw2ttl.py < prov_out.raw > scenario1.ttl
```

The produced W3C PROV trace in this case contains 2047 triples. One can then query that trace for information about the file of interest `www/index.html`. A small snippet of this trace is as follows:

```
<file:/home/alice/index.html> a prov:Entity .
<file:/home/alice/index.html> rdfs:label "/home/alice/index.html" .
<file:/home/alice/index.html> rdf:type dt:texthtml .
<file:/home/alice/index.html> prov:wasGeneratedBy <exe://curl~2977> .
<file:/home/alice/index.html> prov:wasDerivedFrom <file:/etc/nsswitch.conf> .
<file:/home/alice/index.html> prov:wasDerivedFrom <file:/etc/host.conf> .
<file:/home/alice/index.html> prov:wasDerivedFrom <file:/etc/resolv.conf> .
<file:/home/alice/index.html> prov:wasDerivedFrom <file:/etc/hosts> .
<file:/dev/tty1> prov:wasGeneratedBy <exe://curl~2977> .
<file:/dev/tty1> prov:wasDerivedFrom <file:/etc/nsswitch.conf> .
<file:/dev/tty1> prov:wasDerivedFrom <file:/etc/host.conf> .
<file:/dev/tty1> prov:wasDerivedFrom <file:/etc/resolv.conf> .
<file:/dev/tty1> prov:wasDerivedFrom <file:/etc/hosts> .
<exe://curl~2977> prov:startedAtTime 44501782 .
<exe://curl~2977> prov:endedAtTime 59329210 .
```

In this case, the query would look for which tools generated the last versions of the file found and which inputs were used in the process. Because of the n-by-m problem, the query will return many superfluous `prov:wasDerivedFrom` triples, as even the simplest programs need to open and read several files (configuration files, dynamic libraries, etc). Some ways of filtering the superfluous triples are:

- by heuristics, e.g. excluding files not in user's home directory,
- by introspection, e.g. extracting the command line used to invoke each tool.

8. For example, both users are logged into the same account.

4.1.2 Taint Analysis

For this scenario, let's only consider only the files in user's directories as sources of the output file. One can then apply taint analysis in order to determine how these files affected the produced output.

Before starting with taint analysis, one queries for the start/stop time of the processes and then trims down the trace to reduce processing time. Even if taint processing is not performed online, it remains a heavyweight job. Note that the trimmed-down trace will still contain full-system information, but only for the lifetime of the specific process. The following query selects all the activities, `?act`, their associated start and end times that generated `index.html` files:

```
SELECT ?file ?act ?startTime ?endTime WHERE
{
  ?file prov:wasGeneratedBy ?act.
  ?act a prov:Activity .
  ?act prov:startedAtTime ?startTime .
  ?act prov:endedAtTime ?endTime .
FILTER (regex(str(?file), "index.html"))
}
```

In this case, two activities will be returned, along with their trim points: a `vim` editor used by Alice and a `nano` editor used by Bob. One can now use the `scissors` PANDA plugin to trim the trace:

```
$ qemu -replay scenario1.et -panda scissors:start=77908461,end=512637242,name=vim.et
$ qemu -replay scenario1.et -panda scissors:start=648095995,end=1369823251,name=nano.et
```

Now taint analysis can be run on each sub-trace, using `index.html` as both the taint-source used by the `file_taint` plugin and the taint-sink used by the `file_taint_sink` plugin.

```
$ qemu -replay vim.et -panda file_taint:filename=index.html \
   -panda file_taint_sink:sink=index.html
$ qemu -replay nano.et -panda file_taint:filename=index.html \
   -panda file_taint_sink:sink=index.html
```

This will result in the `file_taint_sink` plugin to generate two taint logs that can be used to answer's Ed's original query. The format of the taint log is similar to this (shown for Alice's execution trace):

```
% /home/alice/index.html  000015:0x0a:              036690: 'i':0:
000000: '<':1:000000      000016: '<':1:000016     036691: 'n':0:
000001: '!':1:000001      ...                      036692: 'e':0:
000002: 'D':1:000002      036677:0x20:1:036677     036693: 'a':0:
000003: 'O':1:000003      036678: 'h':1:036678     036694: 'g':0:
000004: 'C':1:000004      036679: 'r':1:036679     036695: 'e':0:
000005: 'T':1:000005      036680: 'e':1:036680     036696: '.':0:
000006: 'Y':1:000006      036681: 'f':1:036681     036697: 'h':0:
000007: 'P':1:000007      036682: '=':1:036682     036698: 't':0:
000008: 'E':1:000008      036683: '"':1:036683     036699: 'm':0:
000009:0x20:1:000009      036684: 'd':0:           036700: 'l':0:
000010: 'h':1:000010      036685: 'a':0:           036701: '"':0:036720
000011: 't':1:000011      036686: 't':0:           036702:0x20:1:036721
000012: 'm':1:000012      036687: 'a':0:           036703: 't':1:036722
000013: 'l':1:000013      036688: '_':0:           ...
000014: '>':1:000014      036689: 'l':0:
```

For each byte of the output its offset and value are printed, followed by a taint flag and the offset of the bytes that affected it in the original file. This simple format allows for quick processing of the log with external tools. E.g. using the following command, one can quickly filter the untainted output bytes:

```
$ awk -F: '$3 ~ /0/{ print; }' vim_taint.log
```

For Alice, this shows the text introduced which starts on offset `036684`. The newly inserted text reads: `data_lineage.html`. Also exposes an interesting quirk of `vim` is exposed: new line characters (`0x0a`) like the one on offset `000015` are always re-inserted by the editor each time you write a file, rather than copied from the input file.

Similarly, for Bob, one has to search for discontinuities in the output taint in order to identify the location where text was removed. It is also possibly to retrieve the original text that was removed from

the execution trace. It is important to note that the whole of the analysis we describe can be performed using solely the captured execution traces, without relying on any other sources of information (such as versioning filesystems or log files). This shows that execution traces can be used as *self-contained provenance artifacts*.

4.2 License Tracking

1) Alice downloads several images from a website.
2) Alice creates a new HTML document using some of the images.
3) The HTML document is converted to PDF format and uploaded to example.org.

Chief editor Ed wants to know if Alice used any images that need to be licensed before using them in a public document.

4.2.1 Initial Analysis

As previously, to answer Ed's question one replays the execution and captures provenance with the PROV-Tracer plugin. It's important to note that this analysis can be applied to any execution trace.

One can run query to see which images the pdf file was derived from.

```
SELECT ?file WHERE
{
   <file:/home/alice/trip.pdf> prov:wasDerivedFrom ?file .
   FILTER (regex(str(?file), ".gif$|.jpg$"))
}
```

The query determines images by their extension. The query returns:

```
row: [file=uri<file:/home/alice/tree.jpg>]
row: [file=uri<file:/home/alice/trip.gif>]
row: [file=uri<file:/home/alice/pentadog.gif>]
```

To determine the license, we find the activity that retrieved the files from the web. In this case, it is a recursive wget. We look at the others file that were retrieved by the same wget and see if any are license files. Using the following query:

```
SELECT ?file ?act ?licence
WHERE {
  <file:/home/alice/trip.pdf> prov:wasDerivedFrom ?file .
  ?file prov:wasGeneratedBy ?act .
  ?licence prov:wasGeneratedBy ?act .
  FILTER (regex(str(?file), ".gif$|.jpg$") && regex(str(?licence), "licence", "i"))
}
```

The query results in:

```
row: [file=uri<file:/home/alice/tree.jpg>, act=uri<exe://wget~2859>,
    licence=uri<file:/home/alice/LICENCE.txt>]
row: [file=uri<file:/home/alice/trip.gif>, act=uri<exe://wget~2859>,
    licence=uri<file:/home/alice/LICENCE.txt>]
row: [file=uri<file:/home/alice/pentadog.gif>, act=uri<exe://wget~2859>,
    licence=uri<file:/home/alice/LICENCE.txt>]
```

While this answers Ed's query, one may want to confirm that the images were indeed included in the PDF. This is where taint analysis comes in.

4.2.2 Taint Analysis

To do this, again we run a query to find the time period when the activity that generated the PDF was run in order to trim down the trace.

```
SELECT ?act ?startTime ?endTime WHERE
{
  <file:/home/alice/trip.pdf> prov:wasGeneratedBy ?act.
  ?act a prov:Activity .
  ?act prov:startedAtTime ?startTime .
  ?act prov:endedAtTime ?endTime .
}
```

Executing the query results in the following:

```
[act=uri<exe://wkhtmltopdf~2862>, startTime=374374451, endTime=814746951]
```

The trace is then trimmed down and taint analysis is run on it in order to calculate the *taint count number* (see §3.3.5) for `trip.pdf` for all the three images. This will allow us to deduce whether the images were used in the creation of the pdf file, or e.g. read but discarded internally. The computed taint numbers are: `tree.jpg`: 285551, `trip.gif`: 149307 and `pentadog.gif`: 285551. All three numbers for `trip.pdf` exceed by two orders of magnitude the taint numbers of other files written to by the converter. Which makes us conclude that the main use of these inputs was to generate the pdf file.

One interesting observation is that `tree.jpg` and `pentadog.gif` have the same taint count number. This can be explained by the fact that `taint2`, in contrast to libdft [Kemerlis et al. 2012] and DataTracker [Stamatogiannakis et al. 2014], also propagates taint on pointer dereference operations by default. This results in spreading of taint to more locations and oftentimes the whole of the output. This phenomenon is termed *over-tainting* and has been studied in detail in [Slowinska and Bos 2009]. In our case, overtainting doesn't seem to affect our analysis. E.g. `trip.html` also had a taint count of 285551. However, configuration files used by the converter had a taint count of 0. In cases where overtainting turns out to be a problem, there has been research on how to intelligently identify when pointer dereference shouldn't propagate taint [Kang et al. 2011].

4.3 Discussion of case studies

4.3.1 Extension to a distributed setting

In the scenarios above, all execution was done on a single machine. We briefly describe here one possible way to apply our approach to the distributed setting.

In a distributed setting, each machine would capture its execution trace, which would be periodically sent to a central server or location that would maintain the necessary information to replay the execution traces. The notion of a central location that collects, aggregates, and manages provenance is known as a provenance store in the literature [Simmhan et al. 2008; Groth and Moreau 2009].

Execution traces of interest can be replayed at the provenance store and provenance generated for each of the machines. The provenance generated by distributed machines can be connected through a number of possible mechanisms. For example, a heuristic can be used that narrows down the window of time a file was received by a machine in its provenance trace and looking in the corresponding time frame within the sender's provenance, one can use port numbers to establish a connection between the sender and receiver file representation. Another mechanism is to add specific provenance metadata in the HTTP headers as advocated by the W3C's Provenance Access and Query Note [Klyne et al. 2013].

4.3.2 Trace indexing

In our case studies, we started with the knowledge that the trace contains the processes we are interested in analyzing. However, this may not always be the case, especially in the case of a distributed setup like the one we described above. This problem can easily be addressed with the general approach of our methodology. Before traces are archived, a lightweight indexing analysis may run. The analysis will annotate the traces with information that could help the user to quickly narrow down the analysis to a small portion of the collected traces. The annotations can be stored in the same RDF store with the W3C PROV information produced by $PROV_{2R}$ and queried through the same interface. For example, if we are interested in a specific directory rather than a file, the indexing pass may just process the `open` and `close` system calls. Then, with the use of regular expressions in our SPARQL query, we can quickly isolate the parts of the trace that are of interest.

4.3.3 Supported Provenance Analyses

[Glavic 2014] categorizes three common types of provenance used for analysis, paraphrasing, given a data item d, these are:

- **Data**: Which data was used in the production of d
- **Transformation**: Which transformations or processes were involved in deriving d

- **Agents, Auxiliary, and Environment**: What users or other factors were involved in deriving d.

Broadly, most provenance analyses are concerned with retrieving information about the above categories. In the scenarios above, we cover all three types of provenance used for analysis but at different levels of granularity. In Table 2, we map each scenario to Glavic's categorization.

	Data	**Transformation**	**Agents**
Web document update	Fine	Coarse	-
License Tracking	Coarse	Fine	Coarse

TABLE 2: Mapping scenarios to provenance analysis types at fine and coarse grained levels.

Generally, *PROV*$_{2R}$ supports fine grained analysis of Data and Transformation provenance as demonstrated in our scenarios. For Agents, Auxiliary, and Environment, taint tracking is less applicable and instead we rely on content analysis or heuristics to determine these impacts (e.g. the user involved). However, additional plugins that focus on understanding auxiliary factors could be built.

An important point is that each of these analysis is *progressive*. Where we needed fine grained or more detailed information, we processed the execution trace again. In particular, we applied taint analysis. This is the fundamental affordance of using execution traces as the substrate for provenance analysis, we can obtain new information by replaying the execution trace with new instrumentation (e.g., if we were interested in identifying all the influences of a data item, we could run a plugin that implements dynamic slicing.)

Thus, a key constraint is how to effectively apply the sort of iterative analysis described in our methodology. Note, in both scenarios we cut down the trace to focus on a temporal region of the trace, we believe that with better tooling other views could be possible (e.g. see §4.3.2 above). One can imagine a more full featured provenance analytics environment based on these notions. We leave this for future work.

4.3.4 Provenance-based forensics

Provenance-based forensics have been an active research topic in the past years [Lu et al. 2010; Pohly et al. 2012; Bates et al. 2015, 2016]. A prerequisite for provenance-based forensics is ascertaining that provenance is collected securely. Towards this end, it has been proposed to push provenance collection to the kernel [Pohly et al. 2012] and use existing kernel mechanisms to establish a chain-of-trust for the collected provenance [Bates et al. 2015]. Although these systems address the problem of secure provenance collection, they do not offer any flexibility in terms of the performed analysis. Since initiating a forensics analysis is usually a response to an unexpected situation (akin to an unstructured process), relying on a fixed type of provenance analysis may be problematic.

We believe that *PROV*$_{2R}$, and future systems based on the same principles, may provide an alternative to these approaches, that combines security with flexibility. The security in this case is a result that hypervisors operate on an even lower semantic level than the kernel, thus they are much harder to subvert. Hypervisor vulnerabilities do exist, but are a rarity: [Wikipedia 2016] lists only a handful of such attacks in the period 2007-2016, many of which are viable only in the presence of some misconfiguration. For example, Cloudburst [Kortchinsky 2009] exploited the unneeded 3D graphics acceleration which was enabled by default in VMware's server-oriented ESX solution. At the same period, 29 known exploits for the Linux kernel have been reported [CVE Details 2016]. Therefore, we could argue that collecting provenance at the hypervisor level using record and replay offers, at least, comparable security with kernel-based solutions.

5 Recording performance evaluation

Given the flexibility of record and replay in combination with taint analysis for provenance capture, we now investigate whether such an approach is viable in practice by exploring the costs associated with using record and replay for provenance analysis. Our goals with this performance evaluation are

Test	PANDA	qemu-tcg		qemu-kvm		VMware	
	index	index	speedup	index	speedup	index	speedup
dhry	111.0	175.2	1.58	2079.5	18.73	3366.3	30.33
whet	87.8	89.5	1.02	784.9	8.94	828.2	9.43
execl-xput	27.6	32.1	1.16	2115.4	76.64	2228.0	80.72
fcopy-256	82.1	112.1	1.37	2894.8	35.26	4806.2	58.54
fcopy-1024	127.0	167.6	1.32	4558.2	35.89	3011.8	23.71
fcopy-4096	265.8	335.0	1.26	8195.7	30.83	7685.1	28.91
pipe-cs	88.7	120.4	1.36	2463.7	27.78	2592.4	29.23
pipe-xput	39.5	49.9	1.26	621.3	15.73	1925.4	48.74
spawn-xput	106.1	132.4	1.25	1958.2	18.46	2363.7	22.28
shell-1	55.8	72.4	1.30	3627.0	65.00	3386.0	60.68
shell-8	62.2	72.0	1.16	3363.3	54.07	3115.0	50.08
syscall	252.2	302.9	1.20	3282.0	13.01	3430.9	13.6
Index Score	88.9	112.2	1.26	2441.3	27.46	2856.4	32.13

TABLE 3: Index scores and speedup for UnixBench tests.

to: *a*) provide evidence on the feasibility of our approach; *b*) establish a baseline for the performance one can expect.

Towards this end, we first focus on the cost of recording an execution trace, as this is the primary limiting factor in whether the technique is applicable. Previous work [Dolan-Gavitt et al. 2014, 2013] already provides some ad-hoc figures on the expected recording performance of PANDA. However, these figures are not easy to reuse because they are not associated with a specific workload. Second, we focus on the size of the execution trace generated by PANDA. The central challenge of our research is to be able to generate provenance for any part of a computing session that would be of interest. In this scenario, recording may be an always-on feature. Therefore, it is important to explore its storage requirements in more depth.

We ran our experiments on a computer with an Intel i7-6700K CPU (4 cores, 4.00GHz), 16GB DDR4 RAM and a 250GB SATA SSD disk. The VM machine configuration is the same we described in §4.

5.1 Recording execution overhead

In [Stamatogiannakis et al. 2016] we used UnixBench for measuring the cost of online taint analysis to generate provenance with DataTracker. The results were very discouraging with regards to potential online use of the tool. The index score achieved was 0.6% of the base index score, a more than $1000\times$ slowdown.

To see if PANDA and record and replay can improve on this situation, we ran the same benchmark on our PANDA VM image. We used four different hypervisors in order to get a complete picture:

- **PANDA**: PANDA (based on QEMU 1.0.1 with recording turned on.
- **qemu-tcg**: PANDA with recording turned off.
- **qemu-kvm**: PANDA with recording turned off and the KVM acceleration enabled.
- **VMware**: Results of UnixBench running in VMware Player 12 on the host machine.

The results of the benchmark can be seen in TABLE 3. Next to the index score, we also show the speedup achieved with each configuration, compared to PANDA with recording enabled. The difference in performance between PANDA and qemu-tcg corresponds to the recording performance overhead. We can see that for the UnixBench workload, PANDA recording results in a 20% slowdown[9] compared to qemu-tcg. This figure is consistent with what has been reported in previous work [Dolan-Gavitt et al. 2013].

9. Or inversely a speedup of $1.26\times$ when recording is turned off.

TABLE 4: Generated trace size for UnixBench tests.

Test	Time (sec)	Trace Size (MiB)	Rate (MiB/sec)
dhry	135	268.46	1.99
whet	158	327.32	2.07
execl-xput	102	246.83	2.42
fcopy-256	148	275.01	1.86
fcopy-1024	150	276.17	1.84
fcopy-4096	149	276.29	1.85
pipe-xput	135	271.81	2.01
pipe-cs	136	496.32	3.65
spawn-xput	106	285.45	2.69
shell-1	136	494.12	3.63
shell-8	196	503.51	2.57
syscall	195	272.03	1.40
Cumulative	1746	3993.33	2.29

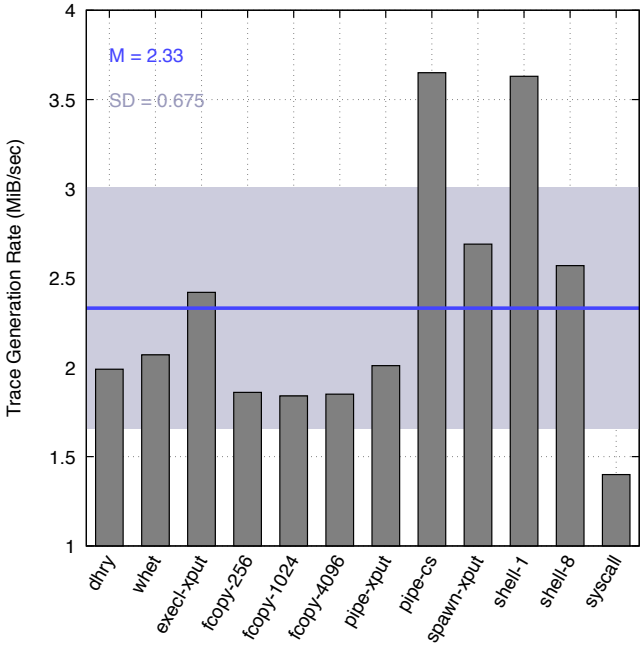

Fig. 3: Trace generation rate for UnixBench tests.

However, it is evident that compared to hardware-assisted virtualization (qemu-kvm, VMWare), the performance of qemu-tcg and PANDA recording is lagging. This level of performance may be acceptable for debugging/reverse engineering purposes—the primary application of PANDA—, however it makes the platform unsuitable for capturing provenance on production systems. On the upside, record and replay systems that support hardware acceleration have already been presented and scheduled for release [Ren et al. 2016]. In summary, while the slowdown of PANDA recording compared to qemu-kvm ($27\times$) or VMWare ($32\times$) may not be adequate for deploying production systems on the platform, PANDA does bring taint-tracking based provenance analysis within reach.

5.2 Space requirements

[Stamatogiannakis et al. 2015] highlighted the importance of managing the disk space requirements for execution trace recording and highlighted strategies to tackle the problem. The most straightforward of these strategies was to simply use compression to reduce the size of the execution traces. An important aspect of the compression for our use case is that it should be possible to perform it *online*.

Towards this end, we studied the execution trace generated by UnixBench with regards to its compressibility. Initially, we captured individual execution traces for each of the benchmarks in the suite. The time, size and rate of recording are presented in TABLE 4 and Fig. 3. We can generally observe that the rate at which PANDA generates execution traces is fairly modest. Moreover, it appears that the benchmarks that involve inter-process communication seem to generate traces at a faster rate.

Next we tested how the trace generated by UnixBench compresses with three different compressor families and different compression levels (default levels underlined): (i) *gzip* (DEFLATE) – levels 3, 6, 9; (ii) *bzip2* (Burrows-Wheeler) – level 9; (iii) *xz* (LZMA) – levels 3, 6, 9. The times and compression ratios achieved are presented in Fig. 4. We can see that the gzip and bzip2 compressors achieve only modest compression for PANDA traces. On the other hand, xz achieves excellent compression rates – over $14\times$. More interestingly, these rates are achieved even when the compressor runs with a fairly low compression level (xz -3). Furthermore, in Fig. 5 we can see where each compressor falls with regards to the trace generation rate of UnixBench. Given that PANDA VMs run on a single CPU core, it seems that it's feasible to, for example, run three VMs on a quad-core machine and dedicate the fourth core to online compression of the generated traces.

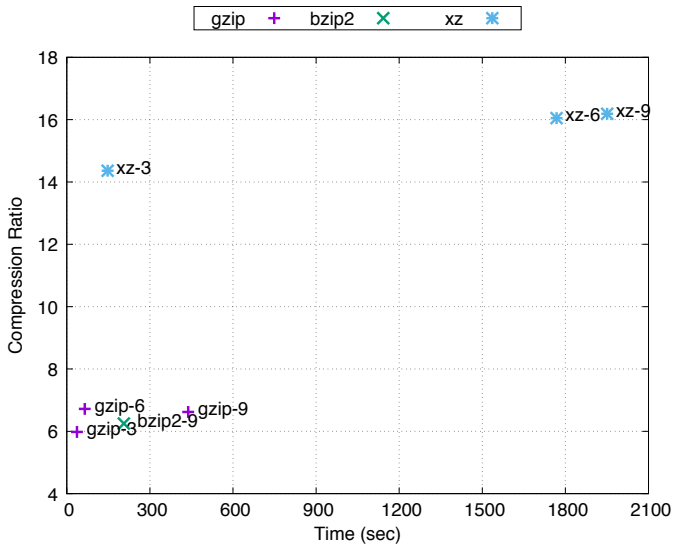

Fig. 4: UnixBench trace compression time/ratio plot for different compressors.

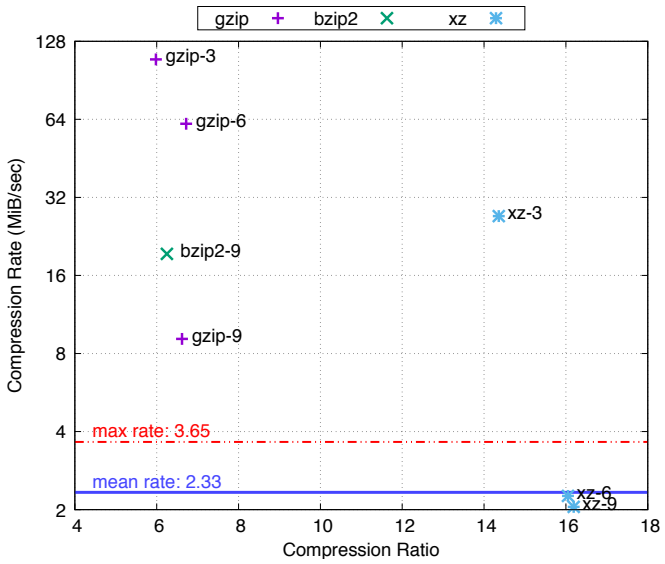

Fig. 5: UnixBench trace compression ratio/rate plot for different compressors. The horizontal lines show the mean and max trace generation rates.

This is a very positive result, however there are still aspects that should be investigated: How will be affected the margin between the compression and trace generation rates in Fig. 5 with the advent of faster record and replay systems [Ren et al. 2016] that are capable of using KVM acceleration? If encrypted data (high entropy) are present in the execution trace, how much will the compression rate fall? Or, if we restrict the memory usage of xz, what will be the effect on the achieved compression ratio? Such questions are left open for future research.

6 Conclusion

The information consumed on the Internet is largely produced by a variety of locally executed processes. What we term unstructured processes. In many cases, these processes impact the trustworthiness and interpretability of that information. Thus, capturing the information's provenance is vital. This is a daunting task, as one has to carefully balance the overhead imposed to capturing provenance with the accuracy of the captured information. In this paper, we present how we can work around choosing one over the other.

Using full execution deterministic record and replay offered by the PANDA platform, we decouple the analysis of provenance from the execution of programs. This offers unparalleled flexibility in the types of provenance analyses we can use. Importantly, this means that we can capture highly accurate provenance when necessary using a technique called taint tracking. This is a powerful technique which has been very popular for security applications, but until now was impractical to use for provenance analysis on live systems because of its overhead. Moreover, we explore the execution and storage overheads associated with record and replay. We conclude that record and replay is coming into maturity and is a becoming a viable alternative to current provenance analysis solutions.

Acknowledgments

We would like to thank the PANDA community and especially Brendan Dolan-Gavitt, Patrick Hulin, Tim Leek and Ryan Whelan for providing the base platform for this research.

References

Andrei Bacs, Remco Vermeulen, Asia Slowinska, and Herbert Bos. 2012. System-Level Support for Intrusion Recovery. In *Proceedings of DIMVA'12*. DOI: http://dx.doi.org/10.1007/978-3-642-37300-8_9

Adam Bates, Devin J. Pohly, and Kevin R. B. Butler. 2016. *Secure and Trustworthy Provenance Collection for Digital Forensics*. Springer, New York, NY, 141–176. DOI: http://dx.doi.org/10.1007/978-1-4939-6601-1_8

Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-system Provenance for the Linux Kernel. In *Proceedings of USENIX SEC'15*.

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of USENIX ATC'05*.

Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker. In *Proceedings of RAID'11*. DOI: http://dx.doi.org/10.1007/978-3-642-23644-0_1

Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Selter, and Andy Hopper. 2014. A Primer on Provenance. *Commun. ACM* 57, 5 (2014), 52–60. DOI: http://dx.doi.org/10.1145/2596628

Lorenzo Cavallaro and R. Sekar. 2011. Taint-enhanced Anomaly Detection. In *Proceedings of ICISS'11*. DOI: http://dx.doi.org/10.1007/978-3-642-25560-1_11

Yufei Chen and Haibo Chen. 2013. Scalable Deterministic Replay in a Parallel Full-system Emulator. In *Proceedings of ACM SIGPLAN PPoPP'13*. DOI: http://dx.doi.org/10.1145/2442516.2442537

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (April 2009). DOI: http://dx.doi.org/10.1561/1900000006

Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of SIGMOD'16*. DOI: http://dx.doi.org/10.1145/2882903.2899401

Jim Chow, Tal Garfinkel, and Peter M. Chen. 2008. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of USENIX ATC'08*.

Jim Chow, Dominic Lucchetti, Tal Garfinkel, Geoffrey Lefebvre, Ryan Gardner, Joshua Mason, Sam Small, and Peter M Chen. 2010. Multi-stage Replay with Crosscut. In *ACM SIGPLAN Notices*, Vol. 45. ACM, 13–24.

James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of ISSTA'07*. DOI: http://dx.doi.org/10.1145/1273463.1273490

Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. 2008. Vigilante: End-to-end Containment of Internet Worm Epidemics. *ACM TOCS* 26, 4 (December 2008). DOI: http://dx.doi.org/10.1145/1455258.1455259

Jedidiah R. Crandall and Frederic T. Chong. 2004. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of MICRO-37*. DOI: http://dx.doi.org/10.1109/MICRO.2004.26

CVE Details. 2016. Linux Kernel Vulnerability Statistics. (November 2016). Retrieved November 17, 2016 from http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html

Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2010. Tainting is Not Pointless. *SIGOPS Operating Systems Review* 44, 2 (April 2010), 88–92. DOI: http://dx.doi.org/10.1145/1773912.1773933

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977). DOI: http://dx.doi.org/10.1145/359636.359712

David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. 2014. Eidetic Systems. In *Proceedings of USENIX OSDI'14*.

Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2014. *Repeatable Reverse Engineering for the Greater Good with PANDA*. Technical Report CUCS-023-14. Columbia University. DOI: http://dx.doi.org/10.7916/D8WM1C1P

Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of PPREW'15*. DOI: http://dx.doi.org/10.1145/2843859.2843867

Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. (May 2016). DOI: http://dx.doi.org/10.1109/SP.2016.15

Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. In *Proceedings of ACM SIGSAC CCS'13*. DOI: http://dx.doi.org/10.1145/2508859.2516697

George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. In *Proceedings of USENIX OSDI'02*. DOI: http://dx.doi.org/10.1145/1060289.1060309

Christof Fetzer and Martin Süßkraut. 2008. Switchblade: Enforcing Dynamic Personalized System Call Models. In *Proceedings of ACM SIGOPS EuroSys'08*. DOI: http://dx.doi.org/10.1145/1357010.1352621

James Frew, Dominic Metzger, and Peter Slaughter. 2008. Automatic Capture and Reconstruction of Computational Provenance. *Concurr. Comput.: Pract. & Exper.* 20, 5 (April 2008), 485–596. DOI: http://dx.doi.org/10.1002/cpe.1247

Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of Middleware'12*. DOI: http://dx.doi.org/10.1007/978-3-642-35170-9_6

Eleni Gessiou, Vasilis Pappas, Elias Athanasopoulos, AngelosD. Keromytis, and Sotiris Ioannidis. 2012. Towards a Universal Data Provenance Framework Using Dynamic Instrumentation. IFIP Advances in Information and Communication Technology, Vol. 376. 103–114. DOI: http://dx.doi.org/10.1007/978-3-642-30436-1_9

Boris Glavic. 2014. *A Primer on Database Provenance*. Technical Report RIIT/CS-DB-2014-01. Illinois Institute of Technology.

Paul Groth, Simon Miles, and Luc Moreau. 2009. A Model of Process Documentation to Determine Provenance in Mash-ups. *ACM Trans. Internet Technol.* 9, 1 (February 2009), 3:1–3:31. DOI: http://dx.doi.org/10.1145/1462159.1462162

Paul Groth and Luc Moreau. 2009. Recording Process Documentation for Provenance. *IEEE Transactions on Parallel and Distributed Systems* (September 2009). DOI: http://dx.doi.org/10.1109/TPDS.2008.215

Paul Groth and Luc Moreau (eds.). 2013. *PROV-Overview: An Overview of the PROV Family of Documents*. W3C Working Group Note NOTE-prov-overview-20130430. World Wide Web Consortium. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/

David A. Holland, Margo I. Seltzer, Uri Braun, and Kiran-Kumar Muniswamy-Reddy. 2008. PASSing the provenance challenge. *Concurr. Comput.: Pract. & Exper.* 20, 5 (April 2008), 531–540. DOI: http://dx.doi.org/10.1002/cpe.1227

Trung Dong Huynh, Paul Groth, and Stephan Zednik (eds.). 2013. *PROV Implementation Report*. W3C Working Group Note NOTE-prov-implementations-20130430. World Wide Web Consortium.

Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *Proceedings of ACM SIGSAC CCS'13*. DOI: http://dx.doi.org/10.1145/2508859.2516704

Yang Ji, Sangho Lee, and Wenke Lee. 2016. RecProv: Towards Provenance-Aware User Space Record and Replay. In *Proceedings of IPAW'16*, Marta Mattoso and Boris Glavic (Eds.). DOI: http://dx.doi.org/10.1007/978-3-319-40593-3_1

Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy Malware Detection Through VMM-based Out-of-the-box Semantic View Reconstruction. In *Proceedings of ACM SIGSAC CCS'07*. DOI: http://dx.doi.org/10.1145/1315245.1315262

Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of NDSS'11*.

D. B. Keator, K. Helmer, J. Steffener, J. A. Turner, T. G. M. Van Erp, S. Gadde, N. Ashish, G. A. Burns, and B. N. Nichols. 2013. Towards structured sharing of raw and derived neuroimaging data across existing resources. *NeuroImage* 82 (2013), 647–661. DOI: http://dx.doi.org/10.1016/j.neuroimage.2013.05.094

Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of VEE'12*. DOI: http://dx.doi.org/10.1145/2151024.2151042

Graham Klyne, Paul Groth (eds.), Luc Moreau, Olaf Hartig, Yogesh Simmhan, James Myers, Timothy Lebo, Khalid Belhajjame, and Simon Miles. 2013. *PROV-AQ: Provenance Access and Query*. W3C Working Group Note NOTE-prov-aq-20130430. World Wide Web Consortium. http://www.w3.org/TR/2013/NOTE-prov-aq-20130430/

Troy Kohwalter, Thiago Oliveira, Juliana Freire, Esteban Clua, and Leonardo Murta. 2016. Prov Viewer: A Graph-Based Visualization Tool for Interactive Exploration of Provenance Data. In *Proceedings of IPAW'16*. DOI: http://dx.doi.org/10.1007/978-3-319-40593-3_6

Kostya Kortchinsky. 2009. Cloudburst: A VMware Guest to Host Escape. In *Black Hat Conference*.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO'04*. DOI: http://dx.doi.org/10.1109/CGO.2004.1281665

Timothy Lebo, Satya Sahoo, Deborah McGuinness (eds.), Khalid Behajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. 2013. *PROV-O: The PROV Ontology*. W3C Recommendation REC-prov-o-20130430. World Wide Web Consortium. http://www.w3.org/TR/2013/REC-prov-o-20130430/

Rongxing Lu, Xiaodong Lin, Xiaohui Liang, and Xuemin (Sherman) Shen. 2010. Secure Provenance: The Essential of Bread and Butter of Data Forensics in Cloud Computing. In *Proceedings of ACM SIGSAC ASIACCS'10*. DOI: http://dx.doi.org/10.1145/1755688.1755723

Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of NDSS'16*.

Xiaogang Ma, Peter Fox, Curt Tilmes, Katharine Jacobs, and Anne Waple. 2014. Capturing provenance of global change information. *Nature Clim. Change* 4, 6 (06 2014), 409–413. DOI: http://dx.doi.org/10.1038/nclimate2141

Wes Masri, Andy Podgurski, and David Leon. 2004. Detecting and Debugging Insecure Information Flows. In *Proceedings of ISSRE'04*. DOI: http://dx.doi.org/10.1109/ISSRE.2004.17

Stephen McCamant and Michael D Ernst. 2006. *Quantitative Information-Flow Tracking for C and Related Languages*. Technical Report MIT-CSAIL-TR-2006-076. MIT, Cambridge, MA, USA. DOI: http://dx.doi.org/1721.1/34892

Luc Moreau. 2010. The Foundations for Provenance on the Web. *Foundations and Trends in Web Science* 2, 2–3 (November 2010). DOI: http://dx.doi.org/10.1561/1800000010

Luc Moreau and Paul Groth. 2013. Provenance: An Introduction to PROV. *Synthesis Lectures on the Semantic Web: Theory and Technology* 3, 4 (2013). DOI: http://dx.doi.org/10.2200/S00528ED1V01Y201308WBE007

Luc Moreau and Paolo Missier. 2013. *PROV-DM: The PROV Data Model*. Recommendation REC-prov-dm-20130430. W3C. http://www.w3.org/TR/2013/REC-prov-dm-20130430/

Tom Oinn, Mark Greenwood, and et al. 2006. Taverna: lessons in creating a workflow environment for the life sciences. *Concurr. Comput.: Pract. & Exper.* 18, 10 (2006). DOI: http://dx.doi.org/10.1002/cpe.993

Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of CGO'10*. DOI: http://dx.doi.org/10.1145/1772954.1772958

Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proceedings of ACSAC'12*. DOI: http://dx.doi.org/10.1145/2420950.2420989

Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. 2010. Paranoid Android: Versatile Protection for Smartphones. In *Proceedings of ACSAC'10*. DOI: http://dx.doi.org/10.1145/1920261.1920313

Georgios Portokalidis, Asia Slowinska, and Herbert Bos. 2006. Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation. In *Proceedings of EuroSys'06*. DOI: http://dx.doi.org/10.1145/1217935.1217938

Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. 2016. Samsara: Efficient Deterministic

Replay in Multiprocessor Environments with Hardware Virtualization Extensions. In *Proceedings of USENIX ATC'16*.

Darren P. Richardson and Luc Moreau. 2016. Towards the Domain Agnostic Generation of Natural Language Explanations from Provenance Graphs for Casual Users. In *Proceedings of IPAW'16*. DOI: http://dx.doi.org/10.1007/978-3-319-40593-3_8

Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of CGO'08*. DOI: http://dx.doi.org/10.1145/1356058.1356069

Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A Survey of Data Provenance in e-Science. *SIGMOD Rec.* 34, 3 (2005). DOI: http://dx.doi.org/10.1145/1084805.1084812

Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2008. Karma2: Provenance management for data driven workflows. *International Journal of Web Services Research* 5, 2 (2008). DOI: http://dx.doi.org/10.4018/jwsr.2008040101

Asia Slowinska and Herbert Bos. 2009. Pointless Tainting?: Evaluating the Practicality of Pointer Tainting. In *Proceedings of EuroSys'09*. DOI: http://dx.doi.org/10.1145/1519065.1519073

Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking Inside the Black-Box: Capturing Data Provenance using Dynamic Instrumentation. In *Proceedings of IPAW'14*. DOI: http://dx.doi.org/10.1007/978-3-319-16462-5_12

Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2015. Decoupling Provenance Capture and Analysis from Execution. In *Proceedings of USENIX TaPP'15*. http://dare.ubvu.vu.nl/handle/1871/53077

Manolis Stamatogiannakis, Hasanat Kazmi, Hashim Sharif, Remco Vermeulen, Ashish Gehani, Herbert Bos, and Paul Groth. 2016. Trade-offs in Automatic Provenance Capture. In *Proceedings of IPAW'16*. DOI: http://dx.doi.org/10.1007/978-3-319-40593-3_3

Tao Wang, Jiwei Xu, Wenbo Zhang, Jianhua Zhang, Jun Wei, and Hua Zhong. 2016. ReSeer: Efficient search-based replay for multiprocessor virtual machines. *Journal of Systems and Software* (2016), –. DOI: http://dx.doi.org/10.1016/j.jss.2016.07.032

Ryan Whelan, Tim Leek, and David Kaeli. 2013. Architecture-Independent Dynamic Information Flow Tracking. In *Proceedings of CC'13*. DOI: http://dx.doi.org/10.1007/978-3-642-37051-9_8

Wikipedia. 2016. Virtual machine escape. (November 2016). Retrieved November 17, 2016 from https://en.wikipedia.org/wiki/Virtual_machine_escape

Min Xu, Rastislav Bodik, and Mark D. Hill. 2003. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of ACM ISCA'03*. DOI: http://dx.doi.org/10.1145/859618.859633

Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. 2007. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of MoBS'07*.

Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings USENIX SEC'12*.

Heng Yin and Dawn Song. 2010. *TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution*. Technical Report UCB/EECS-2010-3. EECS Department, University of California, Berkeley. Retrieved November 17, 2016 from http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html