Efficient Differencing of System-level Provenance Graphs

Abstract

Data provenance, when audited at the operating system level, generates a large volume of low-level events. Current provenance systems infer causal flow from these event traces, but do not infer application structure, such as loops and branches. The absence of these inferred structures decreases accuracy when comparing two event traces, leading to low-quality answers from a provenance system. In this paper, we infer nested natural and unnatural loop structures over a collection of provenance event traces. We describe an 'unrolling method' that uses the inferred nested loop structure to systematically mark loop iterations, i.e., start and end, and thus to easily compare two event traces audited for the same application. Our loop-based unrolling improves the accuracy of trace comparison by 20-70% over trace comparisons that do not rely on inferred structures.

CCS Concepts: • **Information systems** → **Data management systems**; • **Data management** → *Data structures*; • **Data structures** → Data access methods; • **Data access methods** → Unidimensional range search.

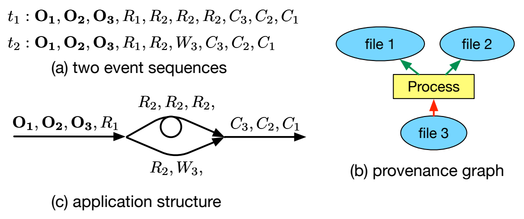
ACM Reference Format:
. 2023. Efficient Differencing of System-level Provenance Graphs. In *Proceedings of 32nd ACM International Conference on Information and Knowledge Management (CIKM).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Data provenance is a record of the origin and evolution of data in a program or application. Audited provenance is vital for establishing reproducibility [4, 6], determining security breaches [11], and for diagnostics [5]. In its raw form, provenance data is simply a series of events, such as system calls or function entry and exits, generated during an application execution. Provenance systems [8, 14, 15] use some of the events to create a directed provenance graph, which enables lineage queries.

CIKM, October 21–25, 2018, University of Birmingham and Eastside Rooms, UK

© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



t_1 : $\mathbf{O_1}, \mathbf{O_2}, \mathbf{O_3}, R_1, R_2, R_2, R_2, C_3, C_2, C_1$
t_2 : $\mathbf{O_1}, \mathbf{O_2}, \mathbf{O_3}, R_1, R_2, W_3, C_3, C_2, C_1$
(a) two event sequences
file 1
file 2
Process
file 3
$R_2, R_2, R_2,$
$\mathbf{O_1}, \mathbf{O_2}, \mathbf{O_3}, R_1$
C_3, C_2, C_1
$R_2, W_3,$
(b) provenance graph
(c) application structure

Figure 1. Low-level event traces, used to generate a provenance graph, omit out some events that represent application structure.

Consider a low-level event sequence, (t_1 in Figure 1(a)) consisting of `open`, `read`, and `close` system calls. The nodes and edges of the provenance graph (in Figure 1(b)) are obtained from some events (in bold), such as the opening of a file by a process for reading or writing, as they determine the causal data flow between the process and the file. Other events, such as reading of the file or close of the file, do not introduce new edges into the provenance graph.

Current systems [8, 14, 15] simply associate these other events with one of two nodes of the causal edge in the graph. These event sequences, however, hold vital information about the program or application structure. For example, if the same application is run twice with different input parameters, and each time it opens the same file for reading but in one trace (t_1 in Figure 1(a)) simply reads the file, and in the other (t_2 in Figure 1(a)) also writes to the file, it exhibits change in application execution behavior due to a different control flow path being exercised on the changed input. Similarly if the same read call is issued multiple times from the same program location, it is indicative of the presence of a loop and branch application structure (Figure 1(c)). Determining this application structure is necessary to improve the fidelity of the provenance graph or to compare two application executions.

In this paper, we infer application structure using event traces generated by multiple executions of an application. We infer two kinds of application structures: loop and branch structures, where a loop corresponds to a set of strongly connected events, and a branch corresponds to a set of optional events, preceded by mandatory events. In general, the repeated occurrence of uniquely identifiable events indicates the occurrence of a loop. However, loops may be nested, may have single or multiple points of entry, or may have branches. Thus, we not only need to infer the existence of a loop but also to determine the type of loop structure to improve the fidelity of the provenance graph.

We use the inferred loop structure to compare any two application event traces. We show that comparing two traces and determining differences such that differences follow loop boundaries results in more accurate reporting of differences than simple edit-distance based comparison of traces. We describe an 'unrolling' based method to obtain loop boundary locations where traces may have diverged or converged.

The rest of the paper is organized as follows: We describe a motivating example in Section 2. We infer loop and branch structures over event traces in Section 3, and 'unroll' graphs based on the inferred structure in Section 3.2. We show the accuracy of our approach in Section 4, describe related work in Section 5, and conclude in Section 6.

2 Motivating Example

We consider an example to show how inferred loops improve accuracy when comparing two traces. Let us assume an application $\mathscr{A}$ was executed two times, the second time with a changed input dataset. Let us further assume that $\mathscr{A}$'s execution results in two event traces t_1 and t_2. Each trace generates a sequence of event identifiers that have been disambiguated and made uniquely identifiable through respective event properties[1].

Figure 2(a) shows two such event traces in which some events repeat. If we compare the linear traces, using a simple comparative measure, such as the edit distance and without accounting for the underlying graph structure, we highlight differences starting at _, i.e., when prefixes do not match, and ending at □, i.e., when a subsequent same event is witnessed again in the other trace. (in general order of comparison matters) As the Figure 2 (a) shows, these event locations of _ and □ are determined without knowledge of graph application structure of traces.

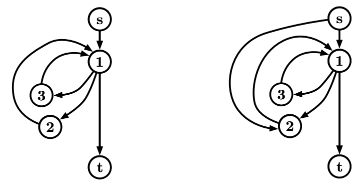
If, however, the traces are represented as graphs and the graph representation of Figure 2(b) is accounted for, event 2 is not the location where differences end. As the graph shows no two different paths merge at event 2. Indeed, the two traces both diverge and converge at event 1. Due to this change in location from 2 to 1, other distinct differences are found that are aligned according to the graph structure. In total, a higher number of differences—3 □s in Figure 2(b) instead of 2— are highlighted than when simply considering the edit distance.

The example only assumes simple nested loop structure. In general, provenance traces may result from unnatural loops (more than one entry point into the loop) as demonstrated in Figure 2(c). Determining the loop hierarchy, i.e., which loop is nested within which loop, and being consistent about where loops start and end, i.e., if the loop starts from event 1 or 2, is important to correctly compare event traces. In the next subsection, we describe how to infer nested natural and unnatural loops and use the loop tree to consistently mark the traces with the chosen start and end points with

[1]We omit the disambiguation details as they are beyond the scope of the current paper and will self-cite the details, if accepted.

$$\mathbf{t_1} : s, \underline{1}, 2, 1, \boxed{3,} 1, 3, 1, 2, \underline{1}, \boxed{t}$$
$$\mathbf{t_2} : s, \underline{1}, \boxed{3,} 1, 3, 1, 2, \underline{1}, 3, 1, \boxed{t}$$

(a)



$$\mathbf{t_1} : s, \underline{1}, 2, \boxed{1,} 3, \underline{1}, 3, \boxed{\underline{\underline{1}},} 2, \boxed{1,} t \qquad \mathbf{t_1} : \underline{s}, 1, \boxed{2,} 1, 3, \underline{1}, 3, \boxed{\underline{\underline{1}},} 2, \boxed{1,} t$$
$$\mathbf{t_2} : s, \underline{1}, 3, \boxed{1,} 3, \underline{1}, 2, \boxed{\underline{\underline{1}},} 3, \boxed{1,} t \qquad \mathbf{t_2} : \underline{s}, \boxed{2,} 1, 3, \underline{1}, 2, \boxed{\underline{\underline{1}},} 3, \boxed{1,} t$$

(b) (c)

Figure 2. (a) traces compared via edit-distance (b) traces compared per natural loop graph (c) traces compared per unnatural loop (entry from both 1 and 2) graph.

respect to the tree using a novel unrolling procedure. Any two marked traces can then be compared for divergence and convergence.

3 Inferring Application Structure

We use notions of loop and loop tree as defined for control-flow analysis [3, 10]. We redefine loop tree to include meta information from provenance traces, and use it to "unroll" the graph to find the points of divergence/convergence. We note that a branch structure is a simplified loop and for simplicity restrict to general loops.

3.1 Inferring Loop Structures

When traces are given, we can create a graph out of the traces. Let $t = (v_1, \ldots, v_s)$ be a trace and $G_t = (V_t, E_t)$ its associated graph, where $v_i \in V_t, \forall i \in \{1, \ldots, s\}$, and $(v_i, v_{i+1}) \in E_t, \forall i \in \{1, \ldots, s-1\}$.

Definition 3.1. Let Q be a subgraph of a graph P. A vertex $v \in V(Q)$ is an *entry point* for Q if v has at least one incoming edge that is not in Q (i.e., is in $E(P) - E(Q)$).

Definition 3.2. We define the *loops* of G inductively as follows.

1. For a graph Q such that $V(Q) = \{v\}$, Q is a loop iff (v, v) is an edge in Q (i.e., (v, v) is a self-loop).
2. Suppose, inductively, that the loops have been defined for any graph of vertex-cardinality less than $|V(G)|$. If G is strongly connected then G is a loop. Moreover, for every entry vertex $v \in V(G)$, the loops of $V(G) - \{v\}$ are loops of G.

Thus, the type of a loop is identified given an SCC and its entry points. If a loop has a single entry point, it is a *natural* loop, and if it has more than one entry point then it is an *unnatural* loop. Strongly connected components are determined as per [18].

Definition 3.3. A *loop tree*, $\mathcal{T}$, of a graph G is defined as follows. The root of $\mathcal{T}$ is a dummy node whose children are

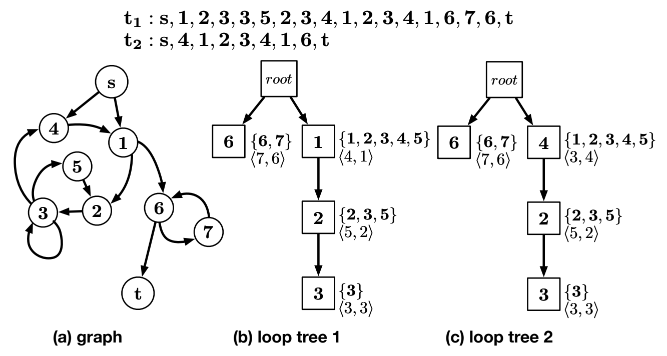


Figure 3. The graph obtained from two traces t_1 and t_2, and its two possible loop trees. In loop trees, the node vertices (in squares) are entry points, numbers in the curly brackets correspond to vertices of the graph, edges in $\langle\rangle$ are back edges.

nodes representing the SCC's of G. The children of a node in $\mathcal{T}$ are defined recursively as follows. For each node α in the tree, arbitrarily fix an entry point v_α of the subgraph G_α (of G) representing node α. (Note that if G itself is a SCC then it has an entry point.) For each SCC C of $G_\alpha - v_\alpha$, create a child of α that represents C.

Figure 3 depicts the loop tree. Let G be the graph created from T_1 and T_2. We perform a depth-first search (DFS) on G; recall that a DFS results in a DFS forest, and that the back edges of G (with respect to the DFS forest) are the edges that go between descendants and ancestors in the forest. The SCCs of G are $\{1, 2, 3, 4, 5\}$ and $\{6, 7\}$. The entry points of the SCC $\{1, 2, 3, 4, 5\}$ can be either 1 or 4, and we pick 1 for the loop tree in Figure 3(b). For $\{6, 7\}$, we have to choose 6. Then, in the subgraph $\{1, 2, 3, 4, 5\} - \{1\}$, one can find the SCC $\{2, 3, 5\}$ and 2 as an entry point. Finally, from $\{2, 3, 5\} - \{2\}$, we will get SCC $\{3\}$ and its entry point 3. If we had chosen 4 as the entry point, then we get an alternate loop tree with a different back edge (Figure 3(c)).

3.2 Creating an Unrolled Graph for Trace Comparison

To compare two traces, we first define the points of divergence/convergence.

Let T_1, T_2 be two traces, and let G_{T_1}, G_{T_2} be the graphs induced by T_1 and T_2, respectively. Also, for a trace $T = (v_1, \dots, v_k)$ in G and for a vertex $v_i \in T$, define $prev(v_i) = v_{i-1}$ if $i > 1$ and $prev(v_i) = \perp$ if $i = 1$ (where $\perp$ denotes nil, i.e., indicating that v_1 does not have a previous vertex in T). Similarly, for $v_i \in T$, define $next(v_i) = v_{i+1}$ if $i < k$, and $\perp$ otherwise (i.e., $i = k$).

Definition 3.4 (*Points of Divergence/Convergence*)**.** Let $T_1 = (v_1, \dots, v_r)$ and $T_2 = (w_1, \dots, w_s)$ be two traces. A vertex $v_i \in T_1$, $i \geq 1$, is said to be a *point of divergence* for T_1 and T_2 if there exists $w_j \in T_2$, $j \geq 1$, such that $v_i = w_j$ and $next(v_i) \neq next(w_j)$. A vertex $v_i \in T_1$, $i \geq 1$, is said to be a *point of convergence* for T_1 and T_2 if there exists $w_j \in T_2$, $j \geq 1$, such that $v_i = w_j$ and $prev(v_i) \neq prev(w_j)$.

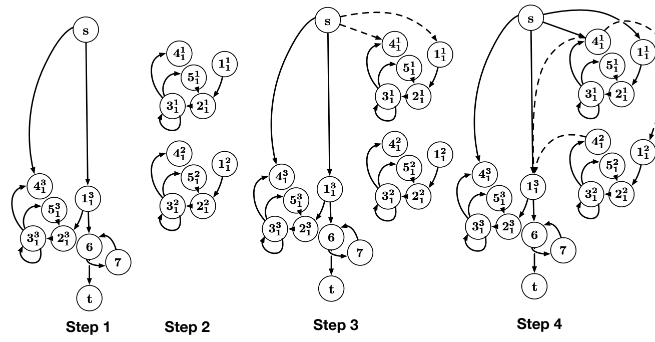


Figure 4. Unrolling of one loop in Figure 3 (a). The back edge is $\langle 4, 1\rangle$, so at first the back edge of G are omitted and k copies of G_1^- are created (Step 1, 2). For the 1st iteration of the loop, all the entry edges from original G to entry nodes in G_1^- are created. (Step 3). For all the other iterations, create an edge from 4_1^{i-1} to 1_1^i and to 1_1^{k+1} for $\forall i \in \{2, \dots, k\}$ (Step 4).

3.2.1 The case where $G = G_{T_1} \cup G_{T_2}$ is a DAG. If $G = G_{T_1} \cup G_{T_2}$ is a directed acyclic graph, then $next(v_i)$, $next(w_j)$ and $prev(v_i)$, $prev(w_j)$, on each respective trace are uniquely defined.

3.2.2 The case where $G = G_{T_1} \cup G_{T_2}$ has loops. In case of loops, the *next* and *prev* are not uniquely defined and we need a loop tree to determine where a loop starts and ends.

Let $\mathcal{T}$ be a loop tree of G and let F_G be a depth-first search (DFS) forest resulting from performing DFS on G. We define an extended graph, Γ_G, from G by repeatedly "unrolling" every loop corresponding to a node in $\mathcal{T}$ as explained next.

(i) Initialize Γ_G to G. Let α be a node in $\mathcal{T}$ of entry point v_α, and let G_α be the subgraph of G corresponding to α. Let $e_1, \dots, e_k$, where $e_i = (u_i, v_\alpha)$, for $i = 1, \dots, k$, be the back edges with multiplicities (i.e., possibly with repetition), with respect to F_G, incoming to v_α in G_α, in the order in which they are traversed by T_1 and then by T_2.

(ii) Let G_α^- be G_α with all these back edges removed. We remove the back edges from Γ_G (step 1), and add k copies of G_α^- to Γ_G. (We will denote them $G_\alpha^{-1}, G_\alpha^{-2}, \dots, G_\alpha^{-k}$) (step 2).) Create edges from the sources of the entry edges in G to all of the entry points in G_α^{-1} (not only v_α^1) (step 3). Let $v_\alpha^{k+1} = v_\alpha$ and let u_α^i, v_α^i denote the copies of vertices u_i and v_α, respectively, for $i = 1, \dots, k$, in the i-th copy of G_α^-.

(iii) For each back edge $e_i = (u_i, v_\alpha)$, for $i = 1, \dots, k$, we add an edge in Γ_G from u_α^i in Γ_G to v_α^{i+1} in Γ_G. We also add edges from u_α^i to v_α^{k+1} for $i = 1, \dots, k$ (step 4).

Consider Γ_G after unrolling all the loops in $\mathcal{T}$. We have:

Theorem 3.1. Γ_G *is a DAG.*

Proof. The proof follows by showing that no cycle in G remains in Γ_G, since our creation of Γ_G does not introduce a cycle that is not a cycle in G.

It is easy to see that the creation of Γ_G cannot introduce a cycle that is not a cycle in G. Therefore, it suffices to show that no cycle in G remains in Γ_G.

Let C be a cycle in G. Since C belongs to one strongly connected component of G, C must be contained in at least one loop in $\mathcal{T}$. Let α be the deepest node in $\mathcal{T}$ such that C is contained in the loop G_α associated with α. By the choice of α, C is not contained in any loop associated with a child of α. It follows from the construction of $\mathcal{T}$ that an edge e of C must be an incoming edge to the entry point of G_α. Since Γ_G is constructed by unrolling all loops in $\mathcal{T}$ including G_α, cycle C is no longer present in any copy of G_α^- in Γ_G (since the copies of edge e now go between different copies of G_α^- in Γ_G). ■

Since Γ_G is a DAG, we can now use the same definition of points of divergence/convergence but on Γ_G.

4 Experimental Evaluation

Provenance Collection. We used the the PIN system [9, 16] to audit provenance traces. PIN uses a combination of system calls and function calls to audit application execution. We consider eight different applications as described in table of Figure 5(a). These applications are taken from Coreutils [2] and SIR [1]. The objective was to experiment with realistic, widely-used programs that have simple source code so, if necessary, we can validate the actual application structure. Raw provenance is collected by executing the application with different input parameters. Table in Figure 5(a) describes properties of collected traces, the obtained graph, and loop tree from the traces.

4.1 Inferred Loops

To measure fidelity, we report, in Figure 5(b), number of inferred natural and unnatural loops for each module. The total number of loops are those loops determined over the subgraph induced by the traces reported in column #3 of table in Figure 5(a). In general, the application may have more loops. As Figure 5 shows, depending on the module, the percentage of unnatural loops can be as high as 30%, showing the necessity to determine both types of loop.

4.2 Precision of Trace Comparison

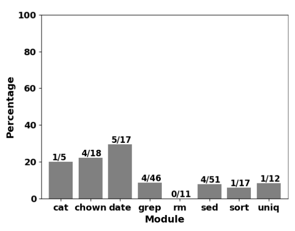
To compute the precision of a difference-based lineage query, we compare two provenance traces, one in which application structure is not inferred, and one in which it is inferred. We use a edit distance measure for comparison. To report the results, we compute if the divergence points were same or different across two methods. We assign a measure of 0 if points of divergence were totally different, and 1 if they are the same. Thus higher the number, the more similar the points of divergence are. As Figure shows, several points of divergence (obtained by comparing two traces with unnatural loops) are ignored if the application structure is not considered (white bar), and a few points are ignored if unnatural loops are not considered (black bar).

Id	Module	Exe.	Cmp.	Avg. Tr.
1	cat	14	91	108.07
2	chown	20	190	1954.25
3	date	16	120	219.81
4	grep	10	45	4072.6
5	rm	20	190	559.65
6	sed	10	45	3933.5
7	sort	20	190	886.80
8	uniq	20	190	752.20

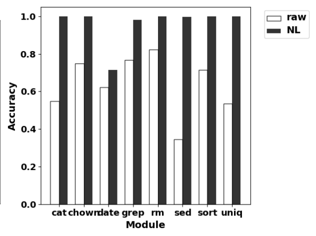
(a)

Id	Div.	Br.	nodes	Depth
1	5	10	93	3
2	22	106	1017	3
3	15	30	400	4
4	40	191	2513	6
5	25	47	474	3
6	30	190	2233	7
7	28	61	608	3
8	11	25	291	3

(b)



(c)



(d)

Figure 5. (a) Trace Details (Exe. = # of executions, Comp.= # of comparisons, Avg. Tr = Average Trace Length)
(b) Trace-induced graph and loop tree details (Div. =total # of divergences across all traces, Br. = total # of branches across all traces, Nodes = total # of nodes across all traces, Depth = total depth of loop tree.)
(c) # of Loop structures (d) Precision of Trace Comparison.

5 Related Work

Collecting provenance at the operating system level provides a broad view of activity across the computer and does not require programs to be modified. Several systems [8, 14, 15] collect provenance by monitoring executions at the operating system. The provided tools and utilities for comparing provenance needed for comparative analysis are minimal and based on edit-distance [7]. Some edit-based methods [4, 12, 19] to compare provenance traces have recently been developed. These methods either do not apply to system call traces [4] or do not account for application structure [12, 19].

We leverage loop identification methods from control flow analysis. The classical algorithm for identifying loops is Tarjan's interval-finding algorithm, but is restricted to natural loops [17]. [20] presents a loop identification algorithm on control-flow graphs for both natural and unnatural loops. We identify loops on the restricted control flow graph that is witnessed by lineage traces and show that we must choose from a set of loop trees to unroll the graph. While loop identification has been used to compress traces [13], ours is the first method to show the use of the inferred loop structures to unroll the graph to compute accurate differences.

6 Conclusion

In this paper, we improved fidelity of system-level provenance graphs by considering application structure. We showed that inferring loops in vital for accurately determining differences across traces and provided a method to unroll a graph based on types of loops so divergence and convergence points are accurately obtained. Our results show comparison accuracy upto 70% over simple edit distance measures. In the future, we plan to identify loop identification in traces from parallel applications to determine how differing inner structure of parallel programs affects performance.

References

[1] 2006. Software-artifact Infrastructure Repository. https://sir.csc.ncsu.edu.

[2] 2016. Coreutils - GNU core utilities. http://www.gnu.org/software/coreutils/.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2 ed.). Pearson Education, Inc.

[4] Zhuowei Bao, Sarah Cohen-Boulakia, Susan B Davidson, Anat Eyal, and Sanjeev Khanna. 2009. Differencing provenance in scientific workflows. In *ICDE*.

[5] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *SIGCOMM*.

[6] Juliana Freire, Cláudio T. Silva, Steven P. Callahan, Emanuele Santos, Carlos E. Scheidegger, and Huy T. Vo. 2006. Managing Rapidly-Evolving Scientific Workflows. In *IPAW*.

[7] Ashish Gehani, Raza Ahmad, Hassaan Irshad, Jianqiao Zhu, and Jignesh Patel. 2021. Digging into big provenance (with SPADE). *Commun. ACM* 64, 12 (2021), 48–56.

[8] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Middleware*.

[9] Intel. 2012. Pin: A Dynamic Binary Instrumentation Tool. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

[10] Ken Kennedy and John R. Allen. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc.

[11] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *NDSS*. 16.

[12] Yuta Nakamura, Tanu Malik, and Ashish Gehani. 2020. Efficient Provenance Alignment in Reproduced Executions. In *12th International Workshop on Theory and Practice of Provenance (TaPP 2020)*. USENIX Association. https://www.usenix.org/conference/tapp2020/presentation/nakamura

[13] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R De Supinski. 2009. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel and Distrib. Comput.* 69, 8 (2009), 696–710.

[14] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical whole-system provenance capture. In *SoCC*.

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

[16] Vijay Janapa Reddi, Alex Settle, and et. al. 2004. Pin: a binary instrumentation tool for computer architecture research and education. In *Workshop on Computer architecture education*.

[17] Robert Tarjan. 1973. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing*. 96–107.

[18] Robert Endre Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[19] P. Thavasimani, J. Cała, and P. Missier. 2019. Why-Diff: Exploiting Provenance to Understand Outcome Differences From Non-Identical Reproduced Workflows. *IEEE Access* (2019).

[20] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. 2007. A new algorithm for identifying loops in decompilation. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings 14*. Springer, 170–183.