Towards Embedding Data Provenance in Files

Thye Way Phua[1], Panos Patros[2] , Vimal Kumar[1]
University of Waikato
[1]Department of Computer Science
[2]Department of Software Engineering
Hamilton, New Zealand
Email: thyewayphua@gmail.com, {panos.patros, vimal.kumar}@waikato.ac.nz

***Abstract*—Data provenance (keeping track of *who* did *what*, *where*, *when* and *how*) boasts of various attractive use cases for distributed systems, such as intrusion detection, forensic analysis and secure information dependability. This potential, however, can only be realized if provenance is accessible by its primary stakeholders: the end-users. Existing provenance systems are designed in a 'all-or-nothing' fashion, making provenance inaccessible, difficult to extract and crucially, not controlled by its key stakeholders. To mitigate this, we propose that provenance be separated into system, data-specific and file-metadata provenance. Furthermore, we expand data-specific provenance as changes at a fine-grain level, or *provenance-per-change*, that is recorded alongside its source. We show that with the use of delta-encoding, provenance-per-change is viable, asserting our proposed architecture to be effectively realizable.**

***Index Terms*—Data Provenance, Embedded Data Provenance, File System, End-to-end Data Provenance, Self-contained Provenance, Delta-encoding**

I. Introduction

Data provenance is commonly defined as the derivation history of data [1], [2], [3]. It is metadata that describes the creation and evolution of data objects. For example, the provenance for a compiled C code may constitute of (1) *Alice* – the user who ran the compilation, (2) *gcc 7.5.0*– the compiler used to compile the code, (3) *03*– compile arguments such as optimisation level and (4) *libacl.so.1* – libraries used. The potential of data provenance is quickly realized and is proposed to be used in various applications such as cloud forensics [4], [5], tracking of data movement in cloud [6], [7] and across the microservices cloud-native applications [8], anomaly detection [9], intrusion detection [10], malware detection [11] and more.

Current provenance systems excel in use cases that can be fulfilled within a single system or application but struggle in use cases that involves multiple systems, such as in cloud computing. We attribute this limitation to the design choice of storing provenance separate from the data it describes, resulting in a weak link between data and its provenance [12]. When data objects move from one system to another, the provenance also needs to be transferred, which is more difficult than it appears to be. Provenance systems such as Progger [7] capture and store provenance of all data objects into a single log file. This means that provenance of all data objects is entangled together, making extraction of provenance difficult. This is a significant issue as provenance captured cannot be accessed by its primary stakeholders: the end-users or data owners. This situation is akin to owning a piece of artwork but having its provenance fully controlled by a third-party, and therefore, relying on their discretion to make the provenance available. This shortcoming significantly restricts the potential of data provenance.

In this paper, we make the following contributions:

- Highlight design issues in existing provenance systems that prevent them from meeting contemporary use cases.
- Propose a block-based technique to embed data provenance into files.
- Propose a high-level software architecture to meet these requirements improving on the state-of-the art.
- Evaluate prototype to ensure that both functional and non-functional requirements are met.

The following scenario is used as a motivating example throughout this paper, with the goal of understanding the provenance for outputs `b` and `c` respectively:

1) `dd if=/dev/urandom of=a bs=4k count=1`
2) `sort a -o b`
3) `dd if=/dev/urandom of=a bs=4k count=1`
4) `sort a -o c`

File A is created using `dd`, populated with some random bytes from the kernel random bytes generator. The contents of File A are sorted and written to output File B. Another stream of random bytes are written to File A as before, and once again the contents of File A are sorted and written to output File C. While the provenance for File A, B and C may seem obvious to us, how existing systems are implemented makes provenance either inaccessible or even if accessible, incorrect. The problem is further exacerbated on contemporary cloud architectures that rely on ephemeral containers that do not maintain the state of their filesystems. The following sections further elaborate the issues in existing systems.

II. Inaccessible Provenance

We consider early provenance systems such as Lineage File System [13] and PASS [12] to be single-system provenance. This is because such systems store data provenance in either a centralized database or log file on the host. In the event of a data transfer between hosts, data provenance is not preserved

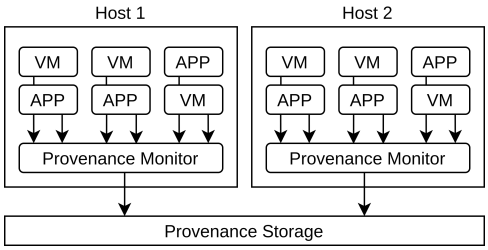

Fig. 1. Simplified representation of current cloud (end-to-end) provenance architecture

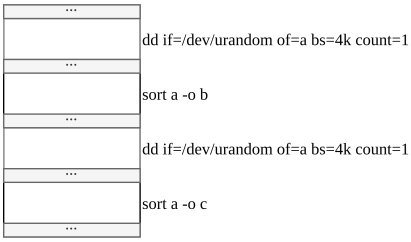

Fig. 2. Extracting provenance from single log file

and the provenance chain is broken [14]. Even if the data is returned to the source, the provenance system would not be able to identify it and continue tracking its provenance treating it as new. For example, say File `A` was created on Host `X` and sent to Host `Y`, Host `Y` performed some modifications (`A'`) and returned it to Host `X`. From the provenance system's perspective, `A` and `A'` are two different entities.

Provenance can only be complete if tracked end-to-end throughout the file's lifetime. In distributed systems, such as clouds, the need for end-to-end provenance tracking becomes apparent [15], [16], [6]. Figure 1 illustrates a simplified representation of existing cloud data-provenance work. In most systems, a centralized data provenance store is employed to gather provenance from multiple hosts and even application within each hosts. Problems arise in this approach if data is transmitted between a host outside of the cluster of systems that is recording data provenance to the centralized data store. Therefore, we argue that the definition of end-to-end provenance needs to be strengthened.

A. Extracting Provenance

In current cloud systems, it is unlikely that provenance will be shared with the end-users, as these systems are designed for and function in an *all-or-nothing* manner, whereby there is no mechanism to extract provenance logs for a particular object of interest.

Now, assume that the broken provenance chain is not an issue and we are able to extract provenance for some object of interest. Using the `sort` example from §I, Figure 2 illustrates the provenance captured using that example. The provenance log shows that File `A` was created by `dd`, followed by File `B` being created as a result of the `sort` command using input File `A`. File `A` is then overwritten by `dd`, followed by the same `sort` procedure, this time producing File `C`. The issue here is that provenance for File `B` and File `C` captured using existing systems will be identical even though their outputs are different. This shows that with previous designs, extracting provenance for a simple example such as this is difficult to achieve. Consider a real-world scenario with large amount of data and large number of files, along with the need to fulfill the security or privacy requirements of data provenance [17].

III. Proposed Design

Premise 1: Every file has provenance
Premise 2: Everything is a file
Conclusion: Everything has provenance

Our proposed concept is built upon UNIX's philosophy of *Everything is a File (or File Descriptor)*. The advantage of this approach is that a common interface can be defined and used on a wide range of resources to achieve the same level of consistency.

A. The Provenance Stack

We first separate provenance into independent components. Using the provenance stack [18], we identify the following as a minimum:

(1) System-provenance. This is what most existing provenance systems capture. What falls within the purview of system provenance are process management operations, such as creating a child process (`fork`) and executing a new process type (`execve`).

(2) File provenance. These events concern only the file but not the system or the data. Examples include changing file ownership (`chown`) and changing file permissions (`chmod`).

(3) Data provenance. This describes exactly how our data was created and modified throughout its lifetime. In a distributed environment, this enables to understand "what has changed" without having access to the artifacts that were involved in the creation of our data.

B. Self-contained Provenance

With provenance split up, the same is needed for storage. Previous work indicated the need of tighter relationships between data and its provenance [19]. We propose system provenance to remain as a centralized log file or database within the system. File and data provenance, however, should be part of the file, as a self-contained, single unit. Thus, there would be no need for querying or filtering provenance for any specific file—and thus, make provenance available to the end users.

C. Level-of-Detail (LOD)

We borrow the concept of level-of-detail (LOD) [20] used in 3D model representation for our proposed design. Figure 3 shows our proposed design for the `sort` example from §I. Each file (`A`, `B` and `C`) is composed of at least one version (Δ) and its current state of data. A version is created on every write, and each version is associated to a process that requested the write. A process has relationships with multiple objects, such as libraries used, executable, process arguments, execution environment and input files.

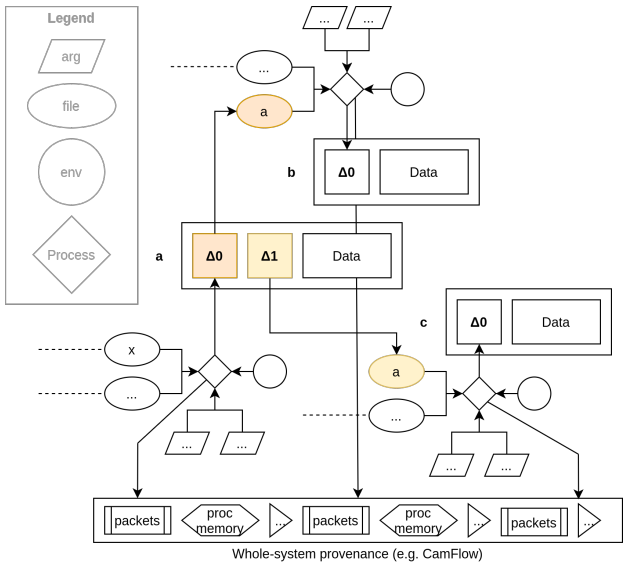

Fig. 3. Overview of our proposed (1) separation of concern (2) self-containing and (3) level-of-detail provenance design

Level	Description
L3	Access to whole-system provenance
L2	Have access to all dependency
L1	Have access to one or more dependency
L0	Only have access to object of interest and nothing else

Data Provenance Clarity ↑ ↓ Data Confidentiality Level

We can see that File `A` has had two versions, and File `B` was derived using the first version of File `A`. File `C` was derived using the second version of File `A`, thus solving the issue in §2.1. We propose a minimum of 4 levels of detail. The higher the level, the better the provenance clarity; similarly, the lower the level, the less sensitive the provenance data is. For example, say, the object of interest is File `B`. If we have access to this and only this file, we can still understand on a high level how this data was created without any ambiguity, which is through `sort` and File `A`. If we also have access to File `A`, we can then see that File `A` has been modified since `B` was produced, hence re-running the process will yield different outcome. Furthermore, if we have access to the whole-system provenance, with a high level of certainty, will be able to re-enact the sequence of events and yield the same outcome. Another way of thinking about this is its similarity to Mandatory Access Control.

D. Versioning through delta-encoding

Storing derivation history of data is similar to what a versioning file system does. A naive approach is to store all versions verbatim, as implemented by the Elephant File System [21]. This, however, generates an enormous amount of data, prioritizing quantity (granularity) over quality (how *meaningful* versions over). Provenance, however, has stricter requirements; it requires granularity [18], as missing any detail renders provenance incomplete and thus of limited use. Consequently, we need a *lossless* versioning system.

A step-up to that would be to compress all versions: rolling back to a particular version becomes a matter of decompressing a version. Currently available compression algorithms suffice for this. Compression, however, in the worst-case scenario, will produce a file of equal or even larger size for data with high entropy, which is also incidentally, a vector for an availability attack. As a simple proof of concept, we generated a 1GB file of random bytes using `/dev/urandom`, and made a modification on a single byte to produce a new version. Compressing both versions takes up exactly 2GB, which is equivalent to storing both versions verbatim. Instead of storing the complete version, we need a method to effectively encode this 1-byte change. This is where delta-encoding can be utilized. Delta encoding is the process of taking two inputs, a source and a target, and computing their differences to produce a smaller patch file to transform source into target. We ran the same experiment as above, using `bsdiff` [22] and observed that the 1-byte change is encoded into 180 bytes of information, a significant improvement over compression.

IV. Prototype

Our prototype was implemented as a Filesystem in Userspace (FUSE) [23]. Our decision to use FUSE is backed by the extensive support on most UNIX operating system, this allows us to test our implementation against multiple systems with little to no modification to our code. A kernel solution such as implementing a new filesystem may provide us with better implementation flexibility and yield better performance. However, a FUSE approach is easier to implement and is therefore more suitable for our proof-of-concept needs. We developed our prototype on Ubuntu 18.04 running FUSE version 3.9.

A. Approach considered

There are many ways in which we can embed provenance into files. We first considered ways that can make use of existing system design to embed provenance metadata so that little to no modification needs to be done to the system. We first looked at extended attributes that many Linux file system supports. Extended attributes is a pre-allocated space in the inode that allows users or developers to assign name:value pairs to files and therefore there is a potential of using it as provenance storage. However, there are two issues with this approach. First is that its name:value pair is quite constrained and not suitable for provenance metadata. Second is that from our experiemnts, none of the native tools or protocols in the UNIX system preserves extended attributes during file transfer. There are however some tools that support transferring of extended attributes when a certain flag is used. `rsync` for instance allows one to specify the `-X` flag (or `-E` flag for `rsync` version 2.6 and before) to enable the transfer of extended attributes. Similarly with the linux copy `cp` command, the `-a` flag or `-preserve=allflag` is required for extended attributes to be preserved. Secure Copy (SCP), Hypertext

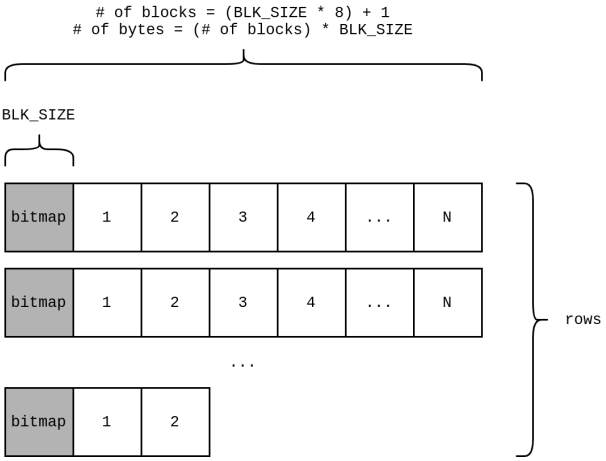

Fig. 4. Architecture of files to efficiently embed provenance and data blocks

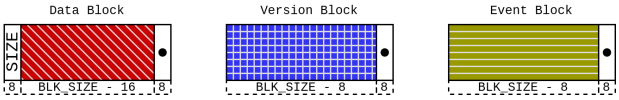

Fig. 5. Types of blocks as defined in our reference design

Transfer Protocol (HTTP), File Transfer Protocol (FTP) completely lacks the support for extended attributes. Furthermore, this approach depends on the filesystem. Extended attributes is not support by Windows; instead, its equivalent in Windows is called Alternate Data Streams.

We, therefore, concluded that storing provenance directly with the data is the most all-inclusive approach. The simplest way of achieving this is to simply concatenate, or append data provenance to the end of the data, like so: `<data><data provenance>`. Our conclusion is validated by the existence of Flexible Image Tansport System (FITS) [24]. FITS is a file format that embeds metadata by prepending metadata to the original data. It is used by astronomers to aid transport, archival and processing of data files. This approach is simple and can fulfil our needs to some extent. However, a file format is not what we need, as this approach may jeopardise the integrity of our data. For example, checksum verification will no longer work as the data is constantly changing. Instead, what we need is a system wide approach where provenance can be tracked on all file objects does not change the definition of *data*. Besides, the approach of prepending or appending provenance to data requires shifting and rewriting all data beyond N+1 bytes, where an operation is performed at byte N, making it extremely inefficient and slow.

B. A Block Approach

Instead, we propose a block-level approach to achieve this goal without the need for shifting. Our approach operates at units of blocks. Each block is of a fixed size (`BLK_SIZE`) and can be specified accordingly. A file is made up of multiple blocks, spanning multiple rows, as illustrated in Figure 4. A row contains a total of `(BLK_SIZE * 8) + 1` number of blocks, which means the number of bytes in a row is (`number of blocks * BLK_SIZE`). Each row begins with a bitmap block, to keep track of used and unused blocks. When a row is fully occupied, a new row is initialized. Each block in the new row is lazily allocated, i.e. a block is only allocated when it is required. Blocks are never deallocated, but will be freed instead, by marking them as free in the bitmap. If all blocks in a row are freed, only then will these blocks be deallocated.

Each block is of a type, meaning a block is restricted to contain only one type of data. An arbitrary number of block types can be defined accordingly. In our reference design, we utilize four types of blocks, which we have identified as bitmap block, data block, versions block and events block, as illustrated in Figure 5. The bitmap block keeps track of used and unused blocks, and the data block holds the current state of the data. Versions and Events block are data provenance, as we have split them into two classes:

- Versions – a chain of data modifications that has been performed on the data in the form of deltas. Captures how a piece of data was modified (evolved) since the creation up to its present state.
- Events (or file-metadata provenance) – operations that have been performed on the file containing the data. For example, change of file ownership (chown), change of file permissions (chmod), etc

Each type of block is maintained as a linked-list where each block contains a pointer to the next block, with the last block pointing to null, as illustrated in Figure 6. Each type of block may have its own structure. In this reference design, the version and events block are regular blocks with 8 bytes reserved at the end of the block to store its next pointer. Data blocks are similar, but with another 8 bytes reserved at the beginning of the block to store the number of available bytes left in the block, this allow insertion or deletion of bytes in a data block without the need of rewriting all subsequent blocks (further elaborated in the next section).

When a new file is initialized, one block of each block type is initialized. These blocks will never be freed and they may have a header to describe subsequent blocks. The headers for the 3 types of blocks are defined as in Table I.

V. Evaluation

We evaluate our proposed system to understand the system performance impact on having provenance embedded into files (non-functional requirement), as well as demonstrating the use cases that our proposed concept enables (functional requirements).

A. System Performance Evaluation

All performance benchmarks were performed on a machine running Intel i7-8850H @ 2.60 GHz, 32.0 GiB of DDR4 RAM and a 500 GB NVMe drive. The machine ran a fresh install of Ubuntu 18.04.5 LTS [25] and running ext4 [26] filesystem. We ran the benchmarks on 3 different setups. First, *baremetal*– this is the base performance of our freshly install system. Next, *vanilla FUSE*–we ran the same benchmark on a FUSE

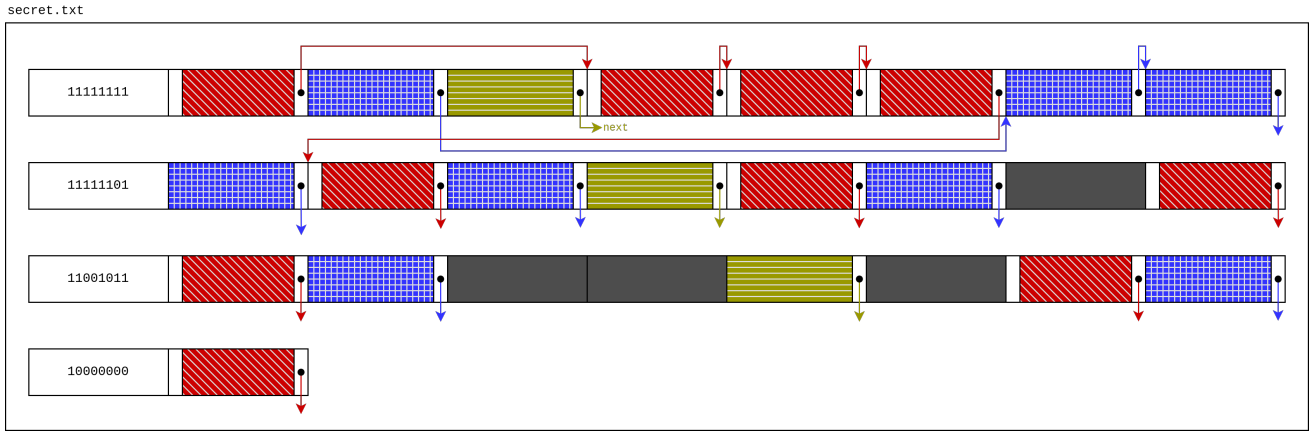

Fig. 6. Example of an augmented file-structure that efficiently includes data and provenance information

filesystem that does nothing to measure the overheads imposed by FUSE, as FUSE by itself imposes quite a significant overhead [27]. Finally, we run benchmark on our prototype, we call this setup *FUSE + Provenance*.

1) Systemcall overhead: We first measure the overheads placed on each file related systemcalls–`open`, `read`, `write`, `close` and `create`. The experiment uses 4KiB size files, and the time measured is from when the systemcall is invoked to the time the systemcall returns. The objective of this benchmark is to evaluate how exactly our approach affects each step in any file related operation. We observe that the overheads are quite significant compared to its baremetal performance. However, most of the overhead observed are incured by FUSE. For example, the overhead added by FUSE to read the a is 2500%, while our prototype only created an additional 10.94% of overhead. We observe that in our prototype, `write` and `create` incur the most overhead. This makes sense because we are writing blocks and the overhead comes from (1) computing deltas, (2) allocating blocks and (3) writing to blocks not in a linear fashion. Nevertheless, the absolute overheads are not significant enough to impact system responsiveness.

2) File I/O performance: We are also interested in file I/O performance, specifically linear read/write and random read/write performance. This benchmark would allow us to observe a more realistic overhead in software applications and thus user experience. We ran benchmarks using filesystem benchmarking tool iozone using a file size of 100 MiB. Similar to the previous benchmark, we observe significant overheads just from FUSE itself. Our proposed system performs significantly well in both linear and random reads. In fact, it performs better than the vanilla FUSE filesystem. We speculate that this is due to our filesystem making request in blocks that utilizes the kernel cache more efficiently. Linear write performance in our system only incurred a 4.43% overhead, which is impressive. However, random writes sees a 3.3x overhead.

Offset	Size	Description
		Data block header
0x00	__ie32	Lower 32-bits of data size in bytes
0x04	__ie32	Upper 32-bits of data size in bytes
0x08	__ie32	Lower 32-bits of pointer to last data block
0x0c	__ie32	Upper 32-bits of pointer to last data block
		Version block header
0x00	__ie32	Number of versions
0x04	__ie32	Cummulative checksum
0x08	__ie32	Lower 32-bits of remaining bytes available in last block
0x0C	__ie32	Upper 32-bits of remaining bytes available in last block
0x10	__ie32	Lower 32-bits of pointer to last version (to allow quick insertion into the last allocated version block)
0x14	__ie32	Upper 32-bits of pointer to last version (to allow quick insertion into the last allocated version block)
		Event block header
0x00	__ie32	Number of events
0x04	__ie32	Cummulative checksum
0x08	__ie32	Lower 32-bits of remaining bytes available in last block
0x0C	__ie32	Upper 32-bits of remaining bytes available in last block
0x10	__ie32	Lower 32-bits of pointer to last version (to allow quick insertion into the last allocated event block)
0x14	__ie32	Upper 32-bits of pointer to last version (to allow quick insertion into the last allocated event block)

TABLE I
BLOCK HEADER FORMAT

systemcall	baremetal	FUSE	overhead	FUSE + Provenance	overhead
systemcall latency in miliseconds, smaller is better					
create	8.45	37.73	346.51%	94.81	151.29%
open	2.72	18.36	575.00%	23.11	25.87%
read	0.45	11.70	2500.00%	12.98	10.94%
write	17.12	22.32	30.37%	95.60	328.32%
close	0.32	0.47	46.88%	0.47	0.00%
Total	29.06	90.58	211.70%	226.97	150.57%

TABLE II
SYSTEMCALL LATENCY MEASURED IN MILLISECONDS

	baremetal	FUSE	overhead	FUSE + Provenance	overhead
I/O time taken for 100MiB file, lower is better					
linear read	131.25	155.68	18.61%	156.21	0.34%
linear write	86.05	112.56	30.81%	117.55	4.43%
random read	499.63	512.18	2.51%	477.57	-6.76%
random write	198.86	612.11	207.81%	2612.21	326.75%

TABLE III
FILE I/O PERFORMANCE OVER A 100 MIB FILE

B. Example Use Case: SLA Violation Detection

We set up a test environment comprising of multiple Docker containers and two volumes. We prepared two scenarios of SLA violation that we hope our system could detect. First, we would like to detect when data is being moved from one volume to another. In this case, we assume that the volumes are located in a different country. We performed the transfer from one volume to another, and our system has successfully showed on each piece of data file that the file has been transferred from one volume to another. This is an important piece of information in detecting SLA violation concerning data sovereignty [28]. The end-user, without the need of an auditor, is able to see and prove that their data has indeed been transferred from one location to another. Using the same scenario, the end-user can also ensue that SLA claims has been met. For example, a cloud provider may guarantee that data is being backup every day. The end-user can verify the provenance on their data file to ensure that backups were actually made.

In additon to the above, the following use cases are not possible to be satisfied using existing provenance system but we expect that they can be accomplished with our proposed architecture:

1) Alice knows what has changed without the need to know what *exactly* has contributed to the change.
2) If a change were malicious or by mistake, Alice would be able to revert the change, without relying on any inputs.
3) Alice can enforce separate access controls on the provenance, such that none or only a subset of provenance can be viewed.
4) The cloud provider can make provenance available immediately to the end-user, without the need for querying or filtering, or risks of breaching of other tenant's privacy.
5) In the event of a dispute, or an audit, Alice and/or Bob can present their provenance to validate or invalidate a cloud providers provenance.
6) In the event of a security breach, having access to the system provenance log files will not compromise any past or current tenants data privacy.
7) In the event of a malicious cyber attack, the whole system can be safely and provably reverted to an earlier state.

C. Verdict, Limitations and Future Work

On average, while the overhead seems significant, the absolute number is rather small. The next question that we ask is that *is this an acceptable level of overhead*. We ran benchmarks to compare with other well received FUSE filesystems such as Veracrypt [29] (a filesystem that encrypts files on-the-fly). We found that our prototype has near identical performance. Taking into consideration that this prototype was built on top of FUSE, a lot of performance bottleneck comes from the extensive amount of context switching due to the fact that the filesystem is running in userspace. If we implement this prototype in the kernel, the elimination of context switching itself will see a significant performance boost. Apart from that we will gain implementation flexibility and thus enabling us to better optimise the process. For example by implementing a cache [30] or scheduling requests to be sent out efficiently [31].

We recognise a few shortcomings in our current prototype. We recognise that by embedding provenance into the file itself raises the question of *'What would happen when a file is deleted?'*. Ideally, provenance data should not be deleted when the file is deleted simply because one can hide their wrongdoings by deleting the file. For example, a malicious insider at a cloud provider can make a copy of a customer's confidential data and delete it to destroy any evidence. In our prototype, when a file is deleted, we simply copy the file's provenance and the deletion action to syslog. The purpose of this is to simply record the deletion action. However, in future work, we envision the concept of 'file-system level' provenance, and its responsibility is for capturing and recording history of what has happened on the file-system, and thus maintains and identifies file deletion operations.

Another potential limitation is that some programs may not behave as we expected. The text editor vi (or Vim) is one example. We observed that provenance data is deleted each time a file is saved. We verified this behaviour with `strace` and we learnt that a new file is created each time a file is written (saved). While not visible to the user, vi creates a backup file, copies the original contents and save the modified file as a new file, hence the preceding provenance data is deleted. This limitation, however, is not unique to our proposed approach and is also present in existing provenance systems [7]. However, this issue can be addressed if our aforementioned recommendation for file-system level provenance is realised.

VI. Conclusion

We identified limitations in existing provenance system that limit the use cases for data provenance that require tracking across multiple systems. We advocate two things: (1) the need for versioning to ensure true preservation of derivation history of data objects, and (2) embedding data provenance into files to provide true end-to-end provenance tracking. The evaluation of our prototype shows that while it incurs some overheads, it does not affect the usability of the system, while adds guarantees to achieving end-to-end provenance tracking.

Acknowledgment

This research is supported by STRATUS (Security Technologies Returning Accountability, Trust and User-Centric Services in the Cloud), a project funded by New Zealand's Ministry of Business, Innovation and Employment (MBIE).

References

[1] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *ACM Sigmod Record*, vol. 34, no. 3, pp. 31–36, 2005.

[2] O. Q. Zhang, R. K. Ko, M. Kirchberg, C. H. Suen, P. Jagadpramana, and B. S. Lee, "How to track your data: Rule-based data provenance tracing algorithms," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2012, pp. 1429–1437.

[3] M. Imran, H. Hlavacs, F. A. Khan, S. Jabeen, F. G. Khan, S. Shah, and M. Alharbi, "Aggregated provenance and its implications in clouds," *Future Generation Computer Systems*, vol. 81, pp. 348–358, 2018.

[4] S. Haque and T. Atkison, "A forensic enabled data provenance model for public cloud," *Journal of Digital Forensics, Security and Law*, vol. 13, no. 3, p. 7, 2018.

[5] P. M. Trenwith and H. S. Venter, "A digital forensic model for providing better data provenance in the cloud," in *2014 Information Security for South Africa*. IEEE, 2014, pp. 1–6.

[6] C. H. Suen, R. K. Ko, Y. S. Tan, P. Jagadpramana, and B. S. Lee, "S2logger: End-to-end data tracking mechanism for cloud data provenance," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, pp. 594–602.

[7] R. K. Ko and M. A. Will, "Progger: An efficient, tamper-evident kernel-space logger for cloud data provenance tracking," in *2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 2014, pp. 881–889.

[8] V. Podolskiy, M. Patrou, P. Patros, M. Gerndt, and K. B. Kent, "The weakest link: revealing and modeling the architectural patterns of microservice applications," 2020.

[9] C. Liao and A. Squicciarini, "Towards provenance-based anomaly detection in mapreduce," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 647–656.

[10] X. Han, T. Pasquier, and M. Seltzer, "Provenance-based intrusion detection: opportunities and challenges," in *10th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.

[11] C. J.-W. Chew, V. Kumar, P. Patros, and R. Malik, "ESCAPADE: Encryption-type-ransomware: System call based pattern detection," in *International Conference on Network and System Security*. Springer, 2020, pp. 388–407.

[12] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *Usenix annual technical conference, general track*, 2006, pp. 43–56.

[13] C. Sar and P. Cao, "Lineage file system," *Online at http://crypto.stanford.edu/cao/lineage.html*, pp. 411–414, 2005.

[14] R. Hasan, R. Sion, and M. Winslett, "Introducing secure provenance: problems and challenges," in *Proceedings of the 2007 ACM workshop on Storage security and survivability*, 2007, pp. 13–18.

[15] P. D. McDaniel, K. R. Butler, S. E. McLaughlin, R. Sion, E. Zadok, and M. Winslett, "Towards a secure and efficient system for end-to-end provenance." in *TaPP*, 2010.

[16] K.-K. Muniswamy-Reddy, U. J. Braun, D. A. Holland, P. Macko, D. Maclean, D. W. Margo, M. I. Seltzer, and R. Smogor, "Layering in provenance systems," in *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, 2009.

[17] J. Cheney and R. Perera, "An analytical survey of provenance sanitization," in *International Provenance and Annotation Workshop*. Springer, 2014, pp. 113–126.

[18] R. K. Ko and T. W. Phua, "The full provenance stack: Five layers for complete and meaningful provenance," in *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*. Springer, 2017, pp. 180–193.

[19] S. Sultana and E. Bertino, "A file provenance system," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 153–156.

[20] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.

[21] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch, "Elephant: The file system that never forgets," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. IEEE, 1999, pp. 2–7.

[22] "Binary diff/patch utility." [Online]. Available: http://www.daemonology.net/bsdiff

[23] C. Henk and M. Szeredi, "Fuse: Filesystem in userspace," *Online at http://sourceforge. net/projects/fuse*, vol. 92, 2012.

[24] D. C. Wells and E. W. Greisen, "Fits-a flexible image transport system," in *Image Processing in Astronomy*, 1979, p. 445.

[25] Canonical. [Online]. Available: https://releases.ubuntu.com/18.04/

[26] "Ext4 (and ext2/ext3) wiki." [Online]. Available: https://ext4.wiki.kernel.org/index.php/Main_Page

[27] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {FUSE} or not to {FUSE}: Performance of user-space file systems," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, 2017, pp. 59–72.

[28] Z. N. Peterson, M. Gondree, and R. Beverly, "A position paper on data sovereignty: The importance of geolocating data in the cloud," 2011.

[29] "Free open source disk encryption with strong security for the paranoid." [Online]. Available: https://www.veracrypt.fr/code/VeraCrypt/

[30] P. Patros, D. Dilli, K. B. Kent, and M. Dawson, "Dynamically compiled artifact sharing for clouds," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 290–300.

[31] P. Patros, K. B. Kent, and M. Dawson, "SLO request modeling, reordering and scaling," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, 2017, pp. 180–191.