Assisted deletion of related content

Author(s):
Ritzdorf, Hubert; Karapanos, Nikolaos; Capkun, Srdjan

Publication Date:
2014

Permanent Link:

https://doi.org/10.3929/ethz-a-010290605 →

Assisted Deletion of Related Content

Hubert Ritzdorf Nikolaos Karapanos Srdjan Čapkun

Institute of Information Security
Department of Computer Science
ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

ABSTRACT

On primary storage systems content is often replicated, converted or modified, and the users quickly lose control over its dispersal on the system. Deleting content related to a particular project from the system therefore becomes a labor-intensive task for the user. In this paper we present *IRCUS*, a system that assists the user in securely removing project-related content, but does not require changes to the user's behavior or to any of the system components, such as the file system, kernel or applications. IRCUS transparently integrates within the user's system, operates in user-space and stores the resulting metadata alongside the files. We implemented and evaluated our system and show that its overhead and accuracy are acceptable for practical use and deployment.

1. INTRODUCTION

Recent cases of information leakage as well as laws and court orders that compel citizens and companies to surrender their devices and data [6, 9, 26, 29, 34], have motivated the need for *secure data deletion*. When faced with a coercive adversary, who is able not only to get access to the devices or data, but can also force the user to disclose all access credentials (e.g., passwords), removing data from a system is the safest way to truly preserve its confidentiality.

However, when deleting content from a personal storage device, a typical user faces the following challenges. Content is replicated across different files, possibly in different formats, e.g., file copies or file containers, and the user has to manually identify all files which share pieces of sensitive content or are related in some other way (e.g., belonging to the same project). This procedure should allow the user to perform meaningful deletion without leaving any sensitive content inadvertently exposed (i.e., not deleted). This is, nevertheless, a tedious and complicated task on modern systems with large storage and complex file dependencies. Moreover, if the system has been running for a longer time, the user will likely not be aware or not even remember where certain files and content reside. In most cases, the user is left with the task of manually inspecting files and explicitly choosing which files to delete.

Previous work has studied secure deletion for complete devices, data blocks or files [2, 5, 7, 14, 33]. However, all of these approaches assume that the user is able to identify the files or data blocks that need to be deleted, as well as those that must be preserved. None of those approaches consider easy secure deletion for content that is replicated across different files. To identify files with shared content, previous work has studied information flow [8, 21], file provenance [20, 36] and causality between files [24]. However, these works were application [20, 43] or operating-system specific [36] and required modification of existing systems [13, 24, 25] or virtualization [16, 21].

In this paper, we introduce *IRCUS* (Identification of Related Content from User Space), a system designed to assist the user with file or project deletion, by proposing related files upon deletion. More precisely, the goal is to assist the user in identifying related files that belong to the same project, by leveraging common content, file co-location and similar access patterns. Once the user wants to delete a file, IRCUS presents all related files and the user decides which files to delete in order to delete the content. IRCUS requires no user actions prior to the file deletion, such as labeling. In accordance with the security principle of least privilege and to allow easy adoption in all types of local and remote storage, IRCUS is implemented in user space.

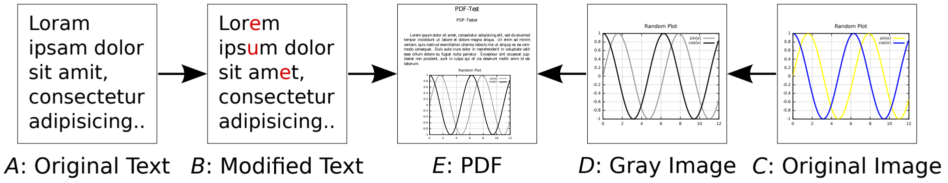
IRCUS is based on a generic approach for discovering project-related files, independent of operating system, storage type or specific applications. As IRCUS executes in user space, it can be installed as a user application and adapted to specific user requirements. Contrary to previous work, our work requires no modification to other applications (e.g., text processing applications) or the operating system. The system overhead is smaller than in most previous proposals. Low overhead is important, because secure deletion is an infrequent operation that should not interfere with normal system operation.

In summary, we make the following contributions.

- We propose a novel approach for the continuous identification of project-related files with the goal of assisting the user in securely deleting sensitive content.
- We design a generic architecture that does not depend on specific applications or systems. Moreover, it requires no modification to the existing applications or the underlying operating system and it can instead be executed as a user-space application.

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA

ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664287.



A: Original Text *B*: Modified Text *E*: PDF *D*: Gray Image *C*: Original Image

Figure 1: Example PDF Generation: As they share content, files A, B, C and D are related to file E. Securely Deleting just E does not securely delete its content.

- We create a prototype implementation of our system, explain our reasoning behind certain implementation decisions and conduct an experimental evaluation of our system in order to demonstrate its effectiveness as well as its performance.

The rest of the paper is structured as follows. In Section 2 we present the problem statement and the matching background in Section 3. In Section 4 we discuss the design principles of our approach and its features. We present our solution and its results in Sections 5 and 6. We discuss related work in Section 7 before concluding in Section 8.

2. PROBLEM STATEMENT

We consider the problem of secure deletion of project-related content. More specifically, our goal is to design a system which, based on observed file system events, learns which user files belong to the same "project" or more generally to the same context. Furthermore, we want our system to be easily integrated into existing platforms and work well across local and remote (e.g., cloud) storage. As a result, the system should assist a user during the file and project deletion processes by displaying related files and the types of their relationships.

Although this problem could be solved by relying on users to label files and their relationships, that approach would require a change in the user behavior and in the mode of system use, which is not realistic in most applications. Instead, our goal is to build a system that will track user behavior and file content and suggest file relationships to the user upon deletion, therefore reducing the user's workload.

On a modern system content is copied, slightly modified, converted or bundled up with other content, e.g., in archives. Common examples for shared content include text documents, presentations and attachments to e-mails. An example more specific to (computer science) research is the compilation of PDF files from text and image files, where the information ends up being contained both in the compiled output and in the source files. In this example, the deletion of the PDF file therefore does not remove any content from the system. In addition, users re-use texts, presentations and implement manual versioning, inadvertently leaving traces of various information within their local and remote systems [23]. After a longer operation of the system the users are therefore unaware of all the copies and relations between different files on the system and require assistance in identifying the files that contain project information.

Identifying files that belong to the same project and/or are a part of the broader context can only be fully done by the user. The notion of a common context is user and possibly organization-specific. No system can fully capture this notion unless it takes input from the user. Since we want to minimize if not fully remove user's involvement we focus on file relations that can be inferred by the system without explicit user input; these relationships can then be presented to the user, who can decide if they are truly meaningful. The relationships that we consider are primarily based on overlapping content, correlated access and file co-location within the file system.

Relying solely on content overlap misses high-level project connections such as in the case of projects that contain a project description and the implementation source code files that have little or no content overlap. We also miss connections with files that are encrypted or encoded in an unrecognized format, e.g., proprietary file formats. To discover these semantic connections we assume that files with related content will frequently be accessed together. This assumption is validated by our small user study in Section 3.3. We therefore classify files with strongly correlated access patterns as related to discover project-based relations.

In summary, our goal is to enable secure deletion of content by finding copies or different versions (through shared content) or files that are part of the same project (through shared access patterns or co-location). These files are presented to the user for deletion. Finally, only the user can decide which files should be securely deleted together to delete specific content.

3. BACKGROUND

Before presenting our proposed architecture, we provide an intuitive description on which files we consider to be "related", such that they have to be deleted together, in order for the secure deletion operation to have the desired result. Furthermore, we motivate the need for designing an efficient system, by showing that the naive approach to solving the given problem would lead to an impractical system due to poor performance. To understand file system operations generated by user actions, we also performed a small user study and present the derived conclusions.

3.1 Related Files

We consider two types of related files. First, files with overlapping content which we call *content-related.* Second, files that exhibit patterns of being accessed together which we call *access-related.*

The content which is shared between two content-related files, say, A and B might be considered sensitive by the user. Thus, when the user chooses to delete file A, our system should also propose file B to the user for deletion.

Content-related files can be explained with the example of PDF generation. Imagine that the user creates a confidential PDF file which contains text and an image. The components are shown in Figure 1. The original text A was modified and saved under a name B, while the original image C was converted to grayscale and saved as D. Finally, the PDF E is then generated out of B and D.

If the user decides to securely delete the PDF, without deleting its components, the PDF can be easily restored. Therefore, files A, B, C and D are content-related to the PDF and should be deleted together with the derived PDF. We can also observe that files A and B are *similar* and are symmetrically content-related, while file B is *contained* in the PDF E.

Access-related files may not share any content at all, however, they can be semantically linked. As an example, consider a project description and a source code file. They are part of the same project and would be frequently accessed together. Linking them can reveal related files to the user. Additionally access relation can link files with overlapping content that is hard or impossible to identify from file system operations, such as encrypted content or proprietary file formats. To increase the accuracy of access relations, we take file co-location into account.

3.2 Naive Approach

One way to address the problem of identifying related files for deletion is the following naive approach. We can implement a system that simply searches for related files on demand, i.e., right before deleting a chosen file. In this reactive approach the identification of access-related files would become much harder, as only the file timestamps are available at that time. Additionally, all files would have to be checked for their timestamps, which would not be scalable.

The reactive approach would also require at least one search through the complete file system for content-related files. After the initial search, further searches might be necessary in order to build a relation graph as in Figure 1.

Using the same system as presented in Section 5 we implemented such a reactive search for content-related text documents. We found that most users had at least 128MB of text data, with an average file size of roughly 16KB. For such file sizes our system spent an average of 1.37 seconds per MB for comparison. Therefore, the reactive approach would take at least 175 seconds for most users, not including the time for file lookup, comparisons to other file types or additional searches to establish a relation graph.

Overall, the reactive approach would have an impractical runtime. Even after applying parallelization the waiting time would significantly impact the user experience. As this approach is not scalable, we conclude that it is not suited to solve the problem.

Alternatively, the naive approach could also be used in a proactive manner in order to eliminate the user waiting time before each deletion. However, this would require the aforementioned scan of all relevant file types to be performed after any change in the file system. This constant load would render the naive approach unusable.

3.3 User Study

To gain a better understanding of realistic file system behavior, we performed a small user study. As we record file system operations and their content, the study was very sensitive, so that we could only get four participants. Participants had the chance to remove sensitive entries from the recordings before sending the recordings back to us.

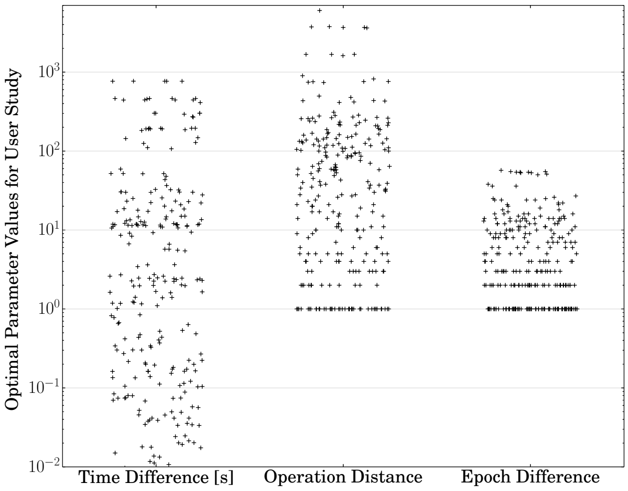


Figure 2: Parameter evaluation based on the user study: Epoch Difference has the smallest variance. Time Difference is measured in seconds and Operation Distance is measured in number of operations.

After manually labeling the user study we observed that related files were mostly accessed shortly after each other both in terms of access time and operation sequence. Often we observed no intermediate unrelated operation. As in previous work [37] we therefore find a temporal proximity between related operations. To determine the parameter for temporal proximity, we performed an analysis as seen in Figure 2. Besides the time difference between related operations and their distance in the operation sequence, we also measured the *epoch difference*.

A new epoch begins, when a new file is written to. Therefore, subsequent write operations to the same file, possibly interleaved with other read operations, do not start a new epoch. Intuitively, the epochs should capture project tasks, e.g. generating a PDF. As the time span of a task, e.g. time taken to edit a file, or the number of operations, e.g. due to the number and size of input files, can differ greatly, using the epoch distance should be more stable. Figure 2 validates this theory as the y-axis is logarithmic and the variance in the epoch difference is therefore magnitudes smaller.

However, epochs will also contain noise in the form of unrelated operations close to related operations. We tried to find features to identify related operations. The number of temporally close operations and their interleaving allowed us to more precisely identify valid access relations. This was partly due to the fact that configuration files and unrelated files were usually just read once, e.g., to parse the configuration or check the file's magic number.

Additionally we compared the paths of related operations and found a strong correlation between the longest common subsequence of paths and whether the corresponding operations were related. This was due to users mostly keeping related files in the same part of the file system and is consistent with previous results [3]. We therefore use co-location to identify related files. However, using the process ID for classification proved to be an unreliable identifier. While it pointed to true positives, it also included false positives, such as configuration files or previously accessed documents.

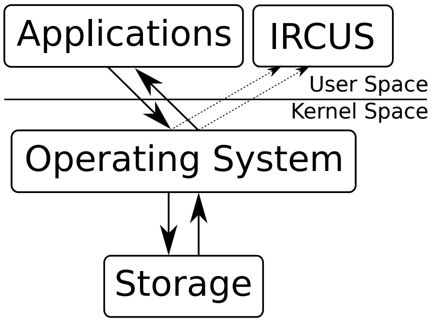


Figure 3: General Architecture: The operating system notifies IRCUS about file system events and their contents. IRCUS can operate in user space.

4. ARCHITECTURE

In this section, we describe the architecture of our proposed solution, called *IRCUS* (Identification of Related Content from User Space). Figure 3 depicts the overall architecture. IRCUS relies on the observation of file system operations in order to identify relations between files. To achieve this, IRCUS, which runs as a user-space application owned by the user, receives file system events from the underlying system which are triggered by the user's running applications. These events include typical file operations such as open, read, write, flush, rename and delete. In the case of read and write operations, the associated content that was read or written is expected to be included in the received event notification, as this is necessary in order to identify content-related files. More specifically, each read/write event notification contains the following information:

$$\langle \textit{operation, path, offset, num\_bytes, content, pid} \rangle$$

where *operation* is the type of the file operation (read or write) on the file specified by *path*. The *content* of size *num_bytes* is read from or written to the *offset*. Optionally, *pid* is the process ID of the process that triggered this operation.

Since IRCUS only looks at file system level operations, its operation is agnostic to and independent of the kind of applications that generate these operations. It also does not have to monitor any kind of interprocess communication (e.g., copying and pasting a text excerpt from one application to another). This is due to the fact that if any kind of interprocess communication leads to a file relation, the relation will be eventually captured by IRCUS through the inspection of the file operations initiated by the communicating processes.

We assume that the user of the system will not behave maliciously, i.e., will not try to actively hide file relations or insert dummy operations. In this work, we also ignore the possibility of malware on the system.

As mentioned above, IRCUS runs in user space and is owned by a specific user of the system. IRCUS is only allowed to monitor file system operations that were initiated by processes running with the privileges of that particular user. We assume that there is a pre-defined set of directories that should be monitored for file system operations relevant for that user. The most typical scenario, which we follow in our implementation, is monitoring the user's home directory.

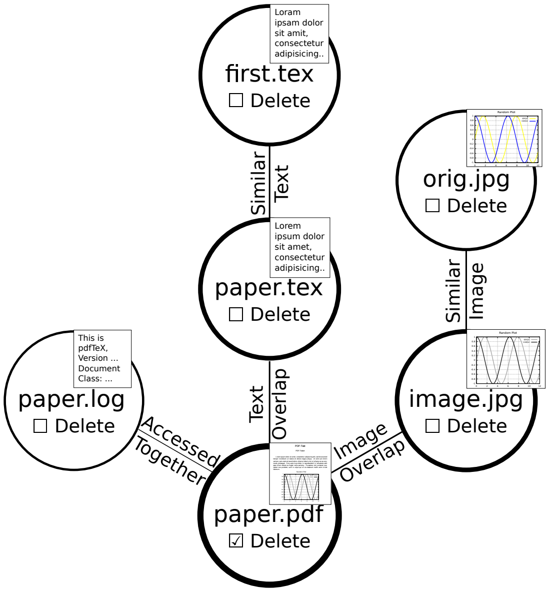


Figure 4: Generated Relation Graph: When planning to delete `paper.pdf`, the user is shown such a relation graph. Clicking on vertices or edges reveals additional information.

In a multi-user system, each user may have his own version of the IRCUS application running and monitoring each user's own directory. The underlying system should dispatch file operation events triggered by each user's applications to the corresponding instance of IRCUS.

To avoid extra file system operations and to efficiently perform the required operations for discovering relations between files, IRCUS maintains a cache that stores recent file operations. The operations are grouped by the accessed file and the files are stored in subdivision based on their file type. Every file type has a separate cache, because we want to establish content relations between certain file types. IRCUS does compare text documents to the text extracted out of a PDF document, while it does not compare text documents to image documents. A subdivided cache provides flexibility and efficiency for this chosen approach.

We picked the caching strategy based on the results of the user study, presented in Section 3.3, that related operations have a small epoch difference. We therefore keep operations from recent epochs in the cache along with the corresponding files. As previously discussed, the operations enable us to track the access relation of files more precisely. Furthermore, the robustness of epoch difference allows us to reliably evaluate content relations independent of specific operation patterns.

Once IRCUS has identified an access or content relation it stores the relation information alongside the two files as previously suggested [25]. The stored information contains the type of relation and the strength of the relation, e.g., the similarity of two images. The stored information is accessible from user space and can be used to educate the user about file relations.

Whenever the user wants to securely delete a chosen file he is able to consult IRCUS which other related files he might

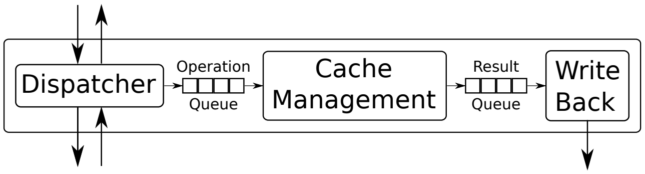


Figure 5: Implementation Design: Based on FUSE, IRCUS intercepts file system operations and multiplexes them for analysis.

also want to delete. IRCUS takes the user-provided file as input and creates a relation graph by transitively evaluating the file's relations. An example of the visualization of such a graph that would be displayed to the user is sketched in Figure 4. The graph only contains the files with the strongest relation to the provided file and provides information on the type of relation between files as well as the strength of each relation (conveyed by the thickness of each vertex). The score-based filtering [25, 36] and visualization [3] of such graphs have been previously studied. Finally, the user is able to select the subset of the graph, which should be securely deleted.

5. IMPLEMENTATION

In this section we outline how we designed our implementation, how we applied the different parts of the architecture and how we improved the performance.

The implementation of IRCUS is subdivided into multiple parts as seen in Figure 5. To observe file system operations the dispatcher thread is based on FUSE [15]. FUSE is more powerful than required, as it intercepts file system operations and enables request or result modifications. The dispatcher thread uses FUSE to multiplex the file system operations. Before returning the operation result to the OS, the dispatcher copies the operation and its content into the operation queue.

The cache management thread receives the operations, organizes the caches and evaluates access and content relation. For IRCUS we have implemented four types of caches: Text, Image, PDF and Binary. Binary is the default that contains all non-matched entries. These caches allow us to demonstrate the capabilities of IRCUS for detecting content relation between simple and complex file types.

The cache entry for a file consists of a sparse representation of the file parts that were read or written. Additionally the cache contains a file-type specific addendum that is used for more efficient content comparison. To perform content comparison IRCUS uses different algorithms for the different file types. These algorithms are not optimal, yet still they demonstrate the feasibility of our scheme.

Previous work [23] suggested the Longest Common Subsequence as a text comparison metric. However, in IRCUS this was too computationally expensive. We therefore adopted Broder's resemblance [4]. It allows us to compute a fixed size "sketch" for every text document and compute the resemblance using two sketches. We store the sketch as the addendum, in order to avoid multiple computations.

For image comparison we extract the color histograms using OpenCV and perform a histogram comparison. Here again we can save the histogram as addendum. When comparing binary data we try to detect identical blocks of data. However, these blocks might be unaligned, therefore we use the rsync algorithm [39]. The rsync algorithm uses a rolling hash to quickly identify potential candidates and then uses a strong signature, saved in the addendum, to compare the data blocks. We filter the results so that blocks of mostly or only zeros are not identified as content related. We also limit the maximum cache size of a binary file, to avoid memory exhaustion when large files are read or written.

PDF is a complex file type, as it is a file container. For PDF files we perform three types of comparison: PDF-to-Text, PDF-to-Image and PDF-to-PDF. To enable comparison we extract text and images from the PDF using the poppler library and save them as addendum. For PDF-to-Text we compare the extracted text to the given text with the previously presented text comparison. Analogously for PDF-to-Image we compare all images from the PDF to the given image. Finally to perform PDF-to-PDF we extract both PDFs and perform text comparison and pairwise image comparison.

IRCUS has to avoid unnecessary content comparisons. For some file types content comparisons are only useful after caching the complete file. To make the most use of content comparisons and to avoid comparisons that have to be repeated after a file update, IRCUS triggers the content comparisons on two events: when a file gets flushed or when a file gets evicted from the cache. The flush operation is triggered by FUSE when the file is closed. This usually means that this file has been completely updated. Furthermore IRCUS performs a content comparison before a file is evicted to detect similarity with recently inserted files.

To further reduce the number of content comparisons, IRCUS performs a comparison between a pair of files only if at least one of the files has been written. This is based on the assumption that IRCUS was in place from the beginning, so that a potential content relation would have previously been detected. This heuristic avoids a lot of content comparisons, as lots of files are only read, e.g., configuration and input files. In case IRCUS was not present from the beginning an initial search could be performed as described in Section 3.2.

However, even with the presented approach we still observe duplicate content comparisons. If file A is compared to B shortly before B is evicted from the cache, the comparison between A and B might be duplicate. Whether it is duplicate depends on a possible modification of A or B in the meantime. To keep track of the order of events, we introduce a logical clock [22]. The logical clock is incremented whenever the cache state changes. For every cached file we save the logical timestamp of the last modification and the last comparison. By checking these timestamps we can avoid unnecessary comparisons.

Once a content relation is detected, it is saved in the cache entries of the corresponding files. The cache entries also contain scores for the access relation between files. We found that the following way of computing the access relation was both simple and powerful:

$$AR(A,B) = \sum_{op_A \in Ops_A} |\{op_B \mid ed(op_A, op_B) \leq T_{ED}\}|$$

To compute the access relation score of two files A and B, IRCUS counts the number of temporally close operations on B around every operation on A. Temporally close operations have an epoch difference $ed(\cdot)$ smaller or equal to the epoch difference threshold T_{ED}. Then IRCUS adds up all the counts for the final score. Therefore, the access relation is symmetric, simple to compute and allows effective filtering as explained in Section 3.3. From the results of the user study we chose the threshold for "closeness" $T_{ED} = 12$. This is a conservative choice to avoid an overfitting to our small dataset.

The cache entry for a file is treated as a logical unit and so it is evicted atomically once the file has not been accessed in the past T_{ED} epochs. The value of T_{ED} is tradeoff between higher detection and higher overhead. After the eviction-triggered content comparisons have been performed, the cache entry is removed from the cache and inserted into the result queue.

The write back thread takes elements out of the result queue. It is responsible for permanently storing the relation information. We decided to store the information in the extended attributes of the file system. Before storing the relation information of file A, the write back thread filters the relations to reduce false positives and thereby also reduce storage overhead. Based on our labelled user data we tested eight different techniques with different parameter values for filtering false positives while trying to preserve true positives.

Three techniques worked well:

Top X Percent Keep the highest access relation scores of A. For this technique the top 5% gave the best results.

X-times Bigger than Mean Compute the mean of A's access scores and keep only scores at least 4-times bigger than the mean.

X-times Bigger than Local Graph Mean Compute the mean score of the local graph consisting of A and all its related files and only keep scores at least 4-times bigger than the mean.

As it preserved most of the true positives while filtering false negatives we picked the "4-times bigger than Mean" filter. Besides such access relations we store all content relations and access relations of co-located files. We define files as co-located, if their paths have a Longest Common Subsequence of at least 75%.

In the extended attributes alongside each file we store its related files with the corresponding access relation and content relation score. If the files were accessed by the same process we also store the process name for later display to the user. The file relations are stored as bi-directional links. This allows efficient updates, e.g., in the case of rename operations. In this case we can quickly identify the related files and update their extended attributes.

Once the user wants to perform a secure deletion, the related files are available in the extended attributes. Using multiple lookups the user can construct a relation graph as in Figure 4 that can be enriched with the stored information, such as the relation strength and the process names.

6. EVALUATION

In this section we present our experimental results. We tested the performance in terms of throughput and latency and conducted a case study for detection results.

As IRCUS is designed for a desktop system, we tested our implementation from Section 5 on a Lenovo Thinkpad X230 running GNU/Linux equipped with a regular hard disk. As underlying file system we picked reiserfs, a standard Linux file system with support for extended attributes.

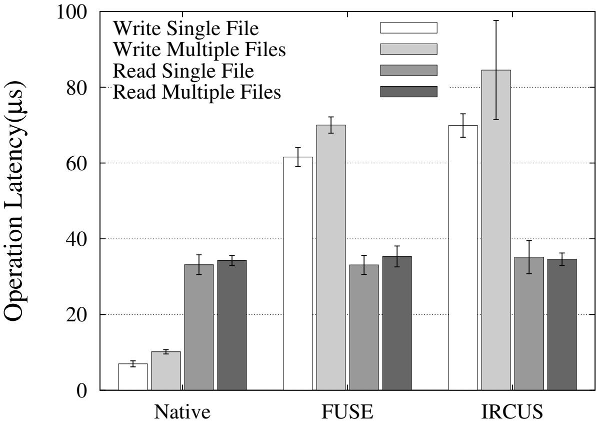


Figure 6: Latency Measurements for different scenarios averaged over 20 repetitions with standard deviation.

6.1 Performance

To evaluate the runtime performance of our system we test its throughput and latency in different scenarios. As points of reference we chose the native file system performance and the performance of a "simple overlay" FUSE file system. This way we can investigate the overhead introduced by FUSE. As "simple overlay" we chose the FUSE example file system "fusexmp_fh" that simply forwards requests.

In order to test latency and throughput we generate 32 pseudo-random, binary 16 MB files. We then use all or some of the files as follows. All the described operations are sequential.

Write Single File Measure the performance of writing a single file from memory to disk 32 times, repeatedly overwriting it.

Write Multiple Files Write all 32 files from memory to disk to fill the cache. Then, measure the performance for writing all 32 files from memory to disk but using different file names.

Read Single File Measure the performance of reading a single file from disk repeatedly.

Read Multiple Files Measure the performance of reading all 32 files sequentially from disk.

We chose these measurements to highlight potential performance differences. When running measurements with multiple files our system has to evaluate and manage file relations, e.g., perform content comparisons. None of this is necessary if only a single file is used. Therefore, "Write Multiple Files" tests the worst-case performance of our system as all files are written and the cache is filled from the beginning.

In order to eliminate the influence of caching, we clear the Linux page cache before each of the 32 measurement

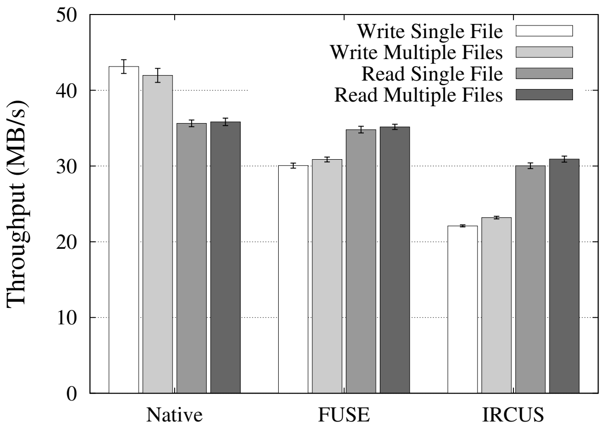


Figure 7: Throughput Measurements for different scenarios averaged over 20 repetitions with standard deviation.

steps. We did not use the FUSE `directio` option to reduce caching, as this option led to reduced functionality of the file system.

The latency measurements are shown in Figure 6. Each of the tests has been repeated 20 times and results have been averaged. We see that FUSE introduces a significant increase in write latency. This is due to the additional copy operations and transitions between kernel and user space. Additionally, IRCUS requires time for the dispatcher to copy the file system operation into the operation queue. As explained "Write Multiple Files" tests the worst-case performance of our system. Evaluating the different file relations occupies the cache management thread so that the operation queue fills up and blocks the dispatcher thread. We picked a fixed-size operation queue of 32 elements to demonstrate such performance bottlenecks.

Overall, the additional latency of IRCUS over FUSE is caused partially by multiplexing the operation inside the dispatcher thread that we cannot avoid. This latency overhead is not general for the architecture, but is specific for our implementation based on FUSE. A different implementation could avoid such overhead.

In Figure 7 we present our throughput measurements. As the file operations in the tests are sequential, the results for the native system are directly related to the latency results. The throughput of FUSE is lower due to its extra latency. FUSE also introduces an overhead when all the write operations are performed on the same file. Finally the throughput of IRCUS is the lowest due to the additional latency on top of FUSE. However, the IRCUS throughput is sufficient to watch high definition videos without interferences.

6.2 Detection

As it makes use of heuristics, IRCUS is not perfect. The exact detection accuracy, however, depends on the specific workload and the used applications. To provide a deeper understanding about the features and shortcomings of IRCUS we present multiple test cases. These test cases are separated into different work profiles as previous work has shown that work profiles differ with roles [23]. The test cases were performed in the setup described above.

All of these tests are only performed once. Additional repetitions would increase the accuracy as random correlations and other noise could be filtered out.

Office Workload

These test cases simulate the usage of IRCUS in an office environment. Based on previous work [17], these test cases contain web content, emails, documents, pictures and presentations. We use the following applications: Firefox, Thunderbird, LibreOffice, Gimp and Gedit. To create a realistic scenario, Firefox and Thunderbird are constantly running, while other applications are opened and closed. We now describe the performed test cases and the resulting detection by IRCUS.

Saved Attachment A binary file is received as a mail attachment and automatically stored inside the mail database. The user then saves the attachment as a file.

As the attachment is encoded when it is stored inside the mail database and IRCUS is unaware of the database's format, IRCUS cannot establish a content relation. However, due to the correlated access patterns, IRCUS establishes a link between the saved file and the mail database. Additionally, an incorrect access relation to an unrelated file is generated.

Shared Image A text document containing an image is opened, the image is copied to the clipboard and inserted into a newly created presentation.

Even though these files are opened directly after another, the text document has already left the cache before the presentation is cached. This is due to the many additional file system operations by LibreOffice. However, as the used clipboard manager saves clipboard images to the disk, the document and the presentation are indirectly linked through the clipboard file. These links are access relations as the image is represented in different formats. Besides this access relation, both files have a few false positives, such as a LibreOffice configuration file used for both files.

Multiple Downloads A file is downloaded to the Downloads folder, after additional browsing and three intermediate downloads, the file is downloaded again and saved in a different folder.

The additional browsing creates so much noise in the form of write operations that the two identical downloads are never compared to each other. Therefore this relation is undetected.

Copy Operation After a binary file has been downloaded, it is copied into a new file.

The copy operation is successfully detected through a content overlap. Additionally both files are linked to the Firefox history through access relations. This is due to the original file download.

Save-As DOC A document is opened in LibreOffice and small changes are made throughout the document. The resulting document is saved under a different name.

This relation is undetected since IRCUS can't interpret DOC files and they are therefore treated as binary. Additionally, there was too much noise for the creation of an access relation.

Save-As Text A text file is opened inside Gedit, where small changes are made and a paragraph is repositioned. The resulting text file is saved under a different name.

As IRCUS analyzes the text differences it finds a content relation between the two files. This demonstrates the importance of file types when comparing to the **Save-As DOC** test case.

Copy-Paste DOC An unchanged text sequence is copied from one document into another document.

Similarly as before this is undetected, as DOC files are treated as binary files and the binary representations differ.

Copy-Paste Text An unchanged text sequence is copied out of one text file into another text file.

As the copied percentage was sufficiently big, a content relation between the two files is observed by IRCUS. The other access related files, however, are false positives.

Copy-Paste JPG A JPG is opened in Gimp, and a part of it is selected and copied into a new image.

As IRCUS performs a histogram-based comparison the content overlap is detected. Additionally, the two files have an indirect access relation through a Gimp temporary file.

Engineering Workload

These test cases simulate the usage of IRCUS in an engineering environment. They are based on compilations for which IRCUS tries to identify the input files.

PDF Generation A PDF is compiled out of a text and two image sources using `pdflatex`.

The three input files are detected as inputs through content relations, access relations and the common process ID. Additionally, the generated bibliography file is linked to the PDF through an access relation and the common process ID. This demonstrates the feasibility of detecting related files for complex file types.

Source Code Compilation An executable is compiled out of multiple source code files using `gcc` and `g++`.

Access relations link the resulting executable to all source and object files. Based on the executable all related files can be identified.

6.3 Discussion

We observe that IRCUS is able to detect relations for the "known" file types (Text, Image and PDF) reliably and precisely. For other file types, it relies on access relations, which contain false positives, especially if only a single operation is observed. We could filter some of these false positives using blacklists, but we aim to provide a general approach. Instead we want to use aggregated information to detect frequent false positives, e.g., a configuration file that is related to many documents.

One could argue that IRCUS needs to understand all file types to be useful. However, previous work [17] has shown that relatively few file types cover most of the related files. Additionally, IRCUS successfully detected access relations, such as between the source code files and the executable or the saved attachment and the mail database.

When relations including databases, e.g., mail database or browsing history, are detected the user should be warned upon deletion to delete such content through the appropriate application interface.

We can also observe the trade-off between increased runtime overhead and improved detection results due to the number of cached operations. Caching less operations (smaller T_{ED}) requires fewer content comparisons and thereby reduces resource usage. When encountering noise due to unrelated operations as in the **Multiple Downloads** test case, however, content relations might be missed due to the smaller threshold T_{ED}.

Overall, implementing additional file types and tuning the parameters could significantly improve the detection results. Fortunately, as IRCUS runs in user space, users or organizations can add such file types based on their specific needs and adapt the system performance to their user profile.

7. RELATED WORK

In this section we survey related work from the different areas this work touches on.

Information Flow Tracking

Systems such as TaintDroid [8] or Panorama [41] employ kernel/OS modifications to perform taint tracking. They face issues as over- or undertainting. CleanOS [38] secures sensitive data on Android by encrypting it in memory and modifying the Dalvik interpreter to decrypt it with keys from the cloud. The Aquifer system [28] restricts information flow based on a policy framework to prevent accidental data disclosure on Android. It limits interactions between applications based flow data provided by systems like TaintDroid or CleanOS. PRIVEXEC [30] creates virtual, encrypted file systems for different applications. This isolates applications from each other. TightLip [42] uses doppelganger processes with bogus data to observe the flow of labelled data. TightLip triggers hen the two processes use different system call arguments.

In contrast our system does not require kernel or OS modifications, is not dependent on taint tracking and does not impose any restrictions on the system.

Forensics

Related work in forensics includes Backtracker [21], a tool to identify the events leading to a malicious file operation. Backtracker tracks all system calls through virtualization or kernel introspection and displays a graph of the relevant events. To reduce the significant number of false positives, later work [37] has enhanced it with data flow analysis. Forensix [12] performs full system call logging and identifies information sources for specific files based on the logs. Keypad [10] and Flight Data Recorder [40] log file system events in a robust and efficient manner to aid analysis. Other systems [1, 43] have been proposed to undo file system changes by malicious processes using external servers. Privacy Oracle [19] analyzes network traces generated by black-box testing and identifies reordered, fragmented or modified sensitive data.

Our system does not require virtualization or additional servers and is targeted at a non-malicious scenario.

Provenance and Causality

Related work has used provenance to improve the quality of desktop search results [36] or to helpfully annotate files during normal user operations [20]. However, these systems require tracking the data flow in all running Windows applications and hooking into specific Windows applications respectively. Thereby, they are OS- and application specific and cannot be generally applied. Previous work also defined Provenance-Aware Storage Systems [25], their functionality and concluded that provenance information should be stored alongside the files as metadata. The stored provenance information was complex including also used executables, their arguments and libraries. This information was stored as a directed acyclic graph, as the approach avoided the generation of cycles. Work building upon that [18] tried to identify the sources of information leaks through "transient provenance". Other work [13, 24] provided undo operations based on causal file relationships using kernel instrumentation.

Our system has a different goal and is independent of the OS and the applications.

Studies on Provenance/Reuse

A study on desktop provenance [17] used application instrumentation to identify that "53.7% of all Excel, PowerPoint, Word, PDF and text files . . . were related to at least one other file". The relationships were usually established through "Copy-and-Paste" or "Save as" events. Most relationship clusters were found to be small. Another study on reuse [23] identified content reuse in presentations within an organization. It found Longest Common Substring Distance to work well for identifying related texts and reported some partial and many exact copies. Overall reuse was around 10-20% and the majority of employees preferred starting from existing work.

These studies show significant reuse and frequent occurrence of related documents and therefore motivate our work.

Identification of Content Overlap or Similarity

Different approaches have been proposed to evaluate content overlap or similarity based on the different usage.

To probabilistically identify similarity between multiple pieces of high-dimensional data, techniques such as locality-sensitive hashing [11] can be used. These functions will probabilistically assign similar data to the same bucket.

Different techniques such as Rabin Fingerprints [27] have been proposed to identify overlaps in binary data. The synchronization tool rsync [39] identifies unmodified file parts to avoid unnecessary copy operations. To do this efficiently, rsync computes a fast rolling checksum and a more secure checksum in case of a match.

To identify text similarity more complex metrics than longest common subsequence and longest common substring have been explored. Broder [4] studied how the resemblance and containment of two documents can be evaluated efficiently. After extracting a fixed-size "sketch" through random sampling, sketches can be used to compare documents. This saves storage and computation time when comparing many documents. Similarly, other work tried to compute a document fingerprint [35] to find matches of a certain length.

As we do not claim to provide an optimal implementation, we have only experimentally tested some of these techniques while integrating others might increase the performance of the system.

Secure Deletion

Secure Deletion has been studied in different scenarios and for different devices [31]. Previous work studied how to securely delete data through overwriting [14] and on flash devices with wear-leveling [32]. Based on the older work on how to extend such properties [2, 7] recent work has expanded into policy- and cloud-based solutions [5, 33].

These approaches assume that the user knows exactly what to delete, whereas our approach assists in finding the data that has to be securely deleted.

8. CONCLUSION

We introduced the concept of assisted deletion of related content, where the user is presented with files that should be securely deleted together to protect data confidentiality. We have demonstrated the feasibility through a user space prototype and shown that it can analyze complex file types and use heuristics to identify related data independent of its binary representation.

In the future we plan to study more user data in order to differentiate different user types and generate more accurate results. As the adoption of such systems depends on their usability, we plan to investigate how to make related content deletion more user-friendly.

9. ACKNOWLEDGEMENTS

This work was supported by the Swiss National Science Foundation (SNSF) through a Sinergia Project under grant No. CRSII2_136318/1.

10. REFERENCES

[1] A. Bacs, R. Vermeulen, A. Slowinska, and H. Bos. System-Level Support for Intrusion Recovery. DIMVA'12. Springer-Verlag, 2013.

[2] D. Boneh and R. J. Lipton. A Revocable Backup System. In *USENIX Security Symposium*, 1996.

[3] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister. Evaluation of Filesystem Provenance Visualization Tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2476–2485, Dec. 2013.

[4] A. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97. IEEE Computer Society, 1997.

[5] C. Cachin, K. Haralambiev, H.-C. Hsiao, and A. Sorniotti. Policy-based Secure Deletion. CCS '13. ACM, 2013.

[6] R. Chirgwin. Surrender your crypto keys or you're off to chokey, says Australia. The Register, 2014.

[7] G. D. Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson. How to Forget a Secret. In *STACS*, Lecture Notes in Computer Science. Springer, 1999.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. OSDI'10. USENIX Association, 2010.

[9] J. Garside. Vodafone reveals existence of secret wires that allow state surveillance. The Guardian, 2014.

[10] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An Auditing File System for Theft-prone Devices. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11. ACM, 2011.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] A. Goel, W.-c. Feng, D. Maier, W.-c. Feng, and J. Walpole. Forensix: A Robust, High-Performance Reconstruction System. ICDCSW '05. IEEE Computer Society, 2005.

[13] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser Intrusion Recovery System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05. ACM, 2005.

[14] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security Symposium*, 1996.

[15] C. Henk and M. Szeredi. FUSE, Filesystem in Userspace. http://fuse.sourceforge.net/.

[16] Y. Huang, A. Stavrou, A. K. Ghosh, and S. Jajodia. Efficiently Tracking Application Interactions Using Lightweight Virtualization. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '08. ACM, 2008.

[17] C. Jensen, H. Lonsdale, E. Wynn, J. Cao, M. Slater, and T. G. Dietterich. The Life and Times of Files and Information: A Study of Desktop Provenance. CHI '10. ACM, 2010.

[18] S. Jones, C. Strong, D. D. E. Long, and E. L. Miller. Tracking Emigrant Data via Transient Provenance. TaPP '11, 2011.

[19] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. CCS '08. ACM, 2008.

[20] A. K. Karlson, G. Smith, and B. Lee. Which Version is This?: Improving the Desktop Experience Within a Copy-aware Computing Ecosystem. CHI '11. ACM, 2011.

[21] S. T. King and P. M. Chen. Backtracking Intrusions. *ACM Trans. Comput. Syst.*, 23(1), Feb. 2005.

[22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), July 1978.

[23] Y. Mejova, K. De Schepper, L. Bergman, and J. Lu. Reuse in the Wild: An Empirical and Ethnographic Study of Organizational Content Reuse. CHI '11. ACM, 2011.

[24] K.-K. Muniswamy-Reddy and D. A. Holland. Causality-based Versioning. *Trans. Storage*, 5(4), Dec. 2009.

[25] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware Storage Systems. ATEC '06. USENIX Association, 2006.

[26] L. Munson. Lavabit appeals contempt of court ruling surrounding handover of SSL keys. Naked Security, 2014.

[27] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01. ACM, 2001.

[28] A. Nadkarni and W. Enck. Preventing Accidental Data Disclosure in Modern Operating Systems. CCS '13. ACM, 2013.

[29] J. Oates. Youth jailed for not handing over encryption password. The Register, 2010.

[30] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. PrivExec: Private Execution as an Operating System Service. In *IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[31] J. Reardon, D. Basin, and S. Capkun. SoK: Secure Data Deletion. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13. IEEE Computer Society, 2013.

[32] J. Reardon, C. Marforio, S. Capkun, and D. Basin. Secure Deletion on Log-structured File Systems. *ASIACCS*, 2012.

[33] J. Reardon, H. Ritzdorf, D. Basin, and S. Capkun. Secure Data Deletion from Persistent Media. CCS '13. ACM, 2013.

[34] A. Rusbridger. David Miranda, schedule 7 and the danger that all reporters now face. The Guardian, 2013.

[35] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03. ACM, 2003.

[36] S. Shah, C. A. N. Soules, G. R. Ganger, and B. D. Noble. Using Provenance to Aid in Personal File Search. ATC'07. USENIX Association, 2007.

[37] S. Sitaraman and S. Venkatesan. Forensic Analysis of File System Intrusions Using Improved Backtracking. IWIA '05. IEEE Computer Society, 2005.

[38] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. OSDI'12. USENIX Association, 2012.

[39] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, February 1999.

[40] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight Data Recorder: Monitoring Persistent-state Interactions to Improve Systems Management. OSDI '06. USENIX Association, 2006.

[41] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. CCS '07. ACM, 2007.

[42] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping Applications from Spilling the Beans. NSDI '07. USENIX Association, 2007.

[43] N. Zhu and T. Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *Dependable Systems and Networks*, 2003.