Efficient Smart Phone Forensics Based on Relevance Feedback

Saksham Varma Robert J. Walls Brian Lynn Brian Neil Levine
School of Computer Science
University of Massachusetts, Amherst, MA, USA
{svarma, rjwalls, blynn, brian}@cs.umass.edu

ABSTRACT

When forensic triage techniques designed for feature phones are applied to smart phones, these recovery techniques return hundreds of thousands of results, only a few of which are relevant to the investigation. We propose the use of relevance feedback to address this problem: a small amount of investigator input can efficiently and accurately rank in order of relevance, the results of a forensic triage tool. We present LIFTR, a novel system for prioritizing information recovered from Android phones. We evaluate LIFTR*'s ranking algorithm on* 13 *previously owned Android smart phones and three recovery engines — DEC0DE, Bulk Extractor, and* `Strings`*— using a standard information retrieval metric, Normalized Discounted Cumulative Gain (NDCG).* LIFTR*'s initial ranking improves the NDCG scores of the three engines from 0.0 to an average of* 0.73*; and with as little as* 5 *rounds of feedback, the ranking score increases to* 0.88*. Our results demonstrate the efficacy of relevance feedback for quickly locating useful information among the large amount of irrelevant data returned by current recovery techniques. Further, our empirical findings show that a significant amount of important user information persists for weeks or even months in the expired space of a phone's memory. This phenomenon underscores the importance of using file system agnostic recovery techniques, which are the type of techniques that benefit most from* LIFTR.

Category and Subject Descriptors
D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

General Terms Forensics, Phones

1. INTRODUCTION

Mobile phones are both a staple of modern life and an abundant source of information for law enforcement. A suspect's phone may contain information about their activities, movements, accomplices, and potentially even direct evidence of a crime. Understanding the limits of what information is recoverable from phones is important for both lawful investigations and for individuals seeking confidentiality when phones are lost or stolen.

Often, the information contained on a phone is time-sensitive, and most useful as a means to help propel an ongoing investigation. Before sending the phone off to a forensic lab for an examination that could take months, investigators first employ a process known as *forensic triage* to quickly recover important information from the phone. Triage is intended to be a first step before a more exhaustive and resource-intensive examination. As the amount and diversity of data on mobile phones increases, it becomes increasingly challenging to identify the relevant information within the limited time available. Our goal is to advance techniques to make this possible.

Previous work on triage has focused on using probabilistic inference on feature phones [21], and deterministic feature search for desktop systems [2,6]. In this paper, we show that these existing approaches encounter significant problems when applied to smart phones. First, smart phones contain a great deal more information than feature phones. While feature phones are often limited to less than 100 MB of storage space, current smart phones store gigabytes of data. Consequently, locating important information often means an investigator must sift through an impractical amount of data.

Second, while the relatively limited number of smart phone operating systems would seem to make the investigator's job easier, mobile phones are supported by many different file systems and growing number of proprietary NAND storage devices with poorly documented flash translation layers (FTLs). Important data frequently lies in the deleted and expired sections of memory (as we show), and it is often impractical and time consuming to create a specialized parser for each phone.

Finally, smart phones contain a wider variety of information, much of which is not relevant to the current investigation. Recovering information from smart phones requires a method of converting raw data into information. The data may be parsed correctly as a phone number, text, URL, or date, but it is not always content that is relevant to the context of the investigation.

We present the design and evaluation of a novel system, LIFTR, for prioritizing information recovered from Android phones. Our system is designed to complement existing triage approaches. LIFTR ranks information according to a combination of scoring and relevance feedback rounds from the investigator. LIFTR is a general approach that can be used with any data recovery technique that provides a string representation and offset of the recovered information. As we show, current techniques can return thousands, and in some cases, millions of *unranked* results per phone. Our approach prioritizes information relevant to an investigation, enabling the investigator to complete the triage process quickly and efficiently.

The primary insight behind LIFTR is that once we can locate a small piece of relevant information, we can quickly locate more by leveraging the spatial locality of data and semantic relationships between NAND pages.

LIFTR is especially powerful for recovering information from expired storage or in cases where the file system cannot be recon-

SPSM'14, November 7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2955-5/14/11 ...$15.00.
http://dx.doi.org/10.1145/2666620.2666628.

structed — we find that for our set of pre-owned evaluation phones, upward of half of the NAND pages are expired. Other data on phones will be easily recoverable because the files are in allocated storage. In these cases, we expect parsing information is relatively easy (though labor intensive, it does not present a research challenge). On the other hand, reconstruction of files from expired space presents a fundamental problem in forensics [5,16]. Fragmentation or loss of critical segments can prevent recovery of a full file, requiring a specialized parser per application file format, without guarantees of recovering any information. Systems like DEC0DE [21] and Bulk Extractor [6] overcome this limitation due to their file system agnostic inference techniques. However, being generalized *recovery engines*, they do not consider the semantics behind the content recovered and hence return large amounts of irrelevant results bearing no connection to the phone user. LIFTR tries to address these problems using an algorithm for ranking results from recovery engines in the order of their relevance to the investigation.

We make several contributions.

- We show empirically that previous approaches to forensic triage (DEC0DE and Bulk Extractor) do not scale to resource-rich smart phones. For one phone, DEC0DE returned over 6.2 million results, of which only about 12 thousand were relevant.
- We introduce LIFTR, a system that quickly identifies the important information by ranking inference results using relevance feedback. LIFTR works in concert with existing triage techniques to provide a reliable ranking of results. It is especially powerful when analyzing expired or deleted storage areas.
- We examine LIFTR on 13 pre-owned Android phones from 6 different manufacturers, using three recovery engines: DEC0DE, Bulk Extractor, and `Strings`. All of the phones contained residual data from the previous owners. The recovery engines return unranked results, resulting in a ranking score near zero out of 1.0. LIFTR is able to rank results initially with an average ranking score of 0.73 when provided with 5 items of information from the investigator (e.g., first or last names, phone numbers or email addresses). Without such hints, LIFTR can leverage information about the file system and provide an initial ranking score of 0.43. With just 5 rounds of investigator feedback, the ranking score increases to 0.88 and 0.73, respectively. Increasing to 20 questions to the investigator, the ranking further improves to 0.91 and 0.79 for the two approaches.
- An open-source, prototype implementation of LIFTR[1].
- We develop techniques to parse and analyze the Yaffs file system. We use these techniques to characterize the lifetime and expiration of data on pre-owned phones; we find that data can live on the device for weeks or months after it has been logically deleted. Further, over half of the NAND pages (56%) contained deleted or expired data. Our results are consistent with previous work in secure data deletion on flash memory [14,15,22].

2. PROBLEM DEFINITION AND METHODOLOGY

2.1 Problem Definition

Our goal is to extract user-centric information from mobile phones that is relevant to an investigation. Previous work on phone forensics [18,21] focused on retrieving all information from a phone's data store, regardless of its provenance. While this approach is appropriate for phones with small amounts of storage and few applications, smart phones store data from innumerable applications, some of which is significantly more important than others.

Our approach takes the output of these systems as a starting point (we call them *recovery engines*), and identifies the content that is most important to an investigation, according to investigator feedback. Specifically, all information is ranked based on a combination of the investigator's relevance feedback, the actual content, and storage system locality information. We output a ranked list so that the investigator need only examine the most-highly ranked information among the tens of thousands of results returned by the recovery engines. We evaluate LIFTR based on relevance of the results returned and the amount of feedback required from the investigator.

NAND Flash. Our focus is on phones; as such, LIFTR's design and evaluation is in the context of NAND-based flash storage. Unlike magnetic disk drives, flash does not overwrite memory in place. Instead, whenever a file is changed, the modified portion is written to unallocated storage, leaving the *expired* data in the flash memory. These expired pages persist for an indefinite period of time — potentially days or weeks under normal usage; see Section 5. Recovery engines, such as DEC0DE, take advantage of this *deletion latency* to recover information that has been logically removed.

NAND file systems fall into two general categories: log-structured file systems designed to work with the raw storage (e.g., Yaffs) and block-device file systems that require an intermediate flash translation layer (FTL). FAT, Ext4, and Samsung's RFS are three common block-device file systems found on Android phones.

For block-device file systems, reconstructing files (or even identifying expired data) from the raw bytes requires knowledge of the FTL — often intractable given the proprietary and diverse nature of these algorithms. Instead, LIFTR assumes as little as possible about the system that stores the information, so that it is compatible with previously unexamined operating systems, file systems, and flash translation layers. Even so, LIFTR can take advantage of file system information when it is available.

Information Types. We are particularly interested in recovering information from the deleted (i.e., expired) pages in memory. These are the pages that are not accessible through a logical examination (i.e., via API calls to the phone's operating system). Because NAND does not overwrite memory in place, multiple copies of a page with slightly different content may be spread across the phone. As a result, even if a phone owner deletes an address book entry that record may still reside on the phone.

Our system works with whatever information is output by recovery engines, as long as the recovered information includes its location on the storage device. Such information includes address book entries, call logs, SMS messages, Facebook chats, and data from smart-phone applications.

2.2 Methodology

Our system recovers data from the *physical image* of a phone. Unlike the *logical image*, the physical image contains the complete layout of bytes in the phone's memory, including deleted data, and is free from any access restrictions that could prevent our system from using data from certain applications.

We do not address the problem of image extraction, assuming that the phone's physical image is already accessible. Also, we assume that the data in the extracted image is not encrypted; otherwise, the tested recovery engines would not be able to extract any information.

Pre-processing Data. LIFTR requires that a phone's data has been pre-processed in three stages.

[1] LIFTR's source code is available at `http://forensics.umass.edu/projects.php`.

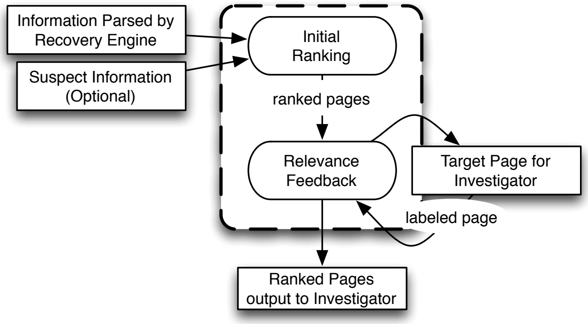

Information Parsed by Recovery Engine
Suspect Information (Optional)
Initial Ranking
ranked pages
Relevance Feedback
Target Page for Investigator
labeled page
Ranked Pages output to Investigator

Figure 1: A high-level overview of LIFTR and its two stages: initial ranking, and relevance feedback. LIFTR takes an unranked list of information parsed by the recovery engine and returns a ranked list of the most important NAND pages.

- First, the phone's physical image is acquired. Depending upon the model and file system of the phone at hand, hardware acquisition techniques (e.g., Joint Test Action Group (JTAG) connections), or software acquisition techniques (e.g., `nandump` or `dd`) can be used [7,19,20].
- Second, portions of the acquired image are filtered out when they are not expected to have user-centric information and are therefore of no interest to the investigator. These portions include binaries and resource files from the operating systems and applications. For example, DEC0DE's *block hash filtering* efficiently removes such data by removing blocks of data that have been observed on other phones, compared by hash value.
- Third, a recovery engine transforms raw data into information, including names, phone numbers, email addresses, and text messages. Examples include DEC0DE [21], DIMSUM [9], Bulk Extractor [6], and more simply, `Strings`. We require that the tool return the corresponding offset in the original phone image and string-based output. E.g., phone numbers should be normalized to text.

The results are input to LIFTR, which we detail in the next section.

It's important to note that **DEC0DE and Bulk Extractor return an unranked, unscored list of results**. The results are grouped together by type (e.g., all credit cards and phone numbers), but grouping is also not practical for thousands or millions of results. Moreover, being unaware of the underlying file system layout, they are not designed to treat files like `contacts2.db`, that are rich in user data, any differently from others. Consequently, these engines have no way of prioritizing user-centric results over the rest.

3. DESIGN OF LIFTR

In this section, we provide a general overview of LIFTR and detail its components. We evaluate the effectiveness of LIFTR in Section 4.

Phone triage tools are often designed to return all data that can be parsed as legitimate information. This design works well for feature phones, but not for smart phones. As we show in Section 4, DEC0DE can return millions of results for smart phones containing data from real users, but only hundreds or thousands of results are relevant user data during triage. Sifting through false positives quickly becomes counterproductive for the investigator.

LIFTR expects that in a triage process, investigators can only review the top *n* results. Accordingly, it ranks information returned by a recovery engine (e.g., DEC0DE or Bulk Extractor) and asks the investigator to confirm the top-ranked page of results. Based on the feedback, it re-ranks the list. As more feedback is provided, our results show that accuracy improves, but of course the amount of feedback must be kept to a minimum. (We explore the effectiveness of up to 20 responses from a simulated investigator in our evaluations.) LIFTR combines the feedback with information from the file and NAND storage-system, and statistics about the content itself to re-rank the results.

An overview of LIFTR is illustrated in Figure 1. Parsed *information* is input from a recovery engine, and combined with information, if any, known to the investigator. Such *a priori* information can include a few names or phone numbers. LIFTR's operation is comprised of two primary stages.

1. **Initial sorting.** Prior to asking for any investigator relevance feedback, LIFTR ranks the input information according to a relevance metric. At this stage, LIFTR also makes use of any *a priori* information provided. In general, better initial sorting implies that less feedback is needed from the investigator.

2. **Relevance feedback.** After the initial sorting, the investigator is asked to label a subset of the pages as true or false positives. The results are re-ranked after each instance of feedback.

We discuss both of these stages in greater detail below.

Page-level granularity. NAND file systems store files in pages. Files are stored across one or more pages, that are not necessarily contiguous in memory. When a file is deleted, the pages remain but are expired. Before they are reused by another file, NAND pages are wiped completely. Therefore, all information on a page belongs to a single file — there is no slack space in a NAND page.

Tools such as DEC0DE and Bulk Extractor search for information in units that we refer to as *fields*; e.g., a date, a phone number, or a credit card number. Each field is returned independent of and without its page information. However, we find that it is often more useful to consider information at a page-level granularity. As such, LIFTR both asks for feedback and presents results at the page level.

First, a page is the unit of write for the file system and, as such, the page typically belongs to just one file.

Second, pages are small and contain few fields, reducing the chances that an investigator will overlook a field and result in a false negative. In our experiments with real phones, DEC0DE returned on average 24 fields per page, and on average 7 relevant fields per true positive page; Bulk Extractor returned on average 6 fields per page, and on average 5 relevant fields per true positive page.

Third, in our experiments, we found that when fields from the same page are presented together to an investigator, they provide context that allows the investigator to determine more accurately their meaning. For example, a date field in isolation may be hard to evaluate as a true positive or not; but a surrounding set of phone numbers and strings, or similar dates, provide a context for deciding the correct feedback.

3.1 Initial Ranking

LIFTR's algorithm for the *initial ranking* takes as input a set of information fields discovered by a recovery engine, each tagged with its byte offset in the original image. The byte offset is used to group the fields by NAND page. Our goal is to rank the phone's pages based on how likely they are to contain information relevant to the user and investigation. To do so, LIFTR assigns an initial *quality score* to a page *p* using a *normalized weighted sum* of a set of features calculated for each field:

$$\text{quality}_0(p) = \frac{\sum_{i=1}^{|F_p|} \sum_{j=1}^{m} w_j a_{ij}}{|F_p|} \quad (1)$$

where F_p is the set of fields contained on page p; a_i is a vector of m features calculated for the i^{th} field using one of the features we discuss below, e.g., file system knowledge or *a priori* knowledge, and text value; and w is a vector of weights for each of these features, that sum to 1. LIFTR's initial ranking of pages is sorted from the highest to lowest quality scores.

For our evaluation, we consider different combinations of three field features, based on an investigator's *a priori* knowledge, limited file system information, and the quality of the field text, respectively. LIFTR can be easily extended to support additional features.

1. **Investigator's *a priori* knowledge.** In some cases, before forensic analysis begins, an investigator will have basic external knowledge about the case, the phone owner, or an accomplice; e.g., a first or last name, email address, or phone number. When we allow its use in our evaluations, LIFTR sets the value of fields that contain strings supplied by the investigator with a 1, and 0 otherwise. In our evaluations we allow only 5 strings as *a priori* input; see Section 4.

2. **File system knowledge.** In some cases, an investigator will have knowledge about the OS and how it is designed to store information. For example, in Android phones, the `contacts2.db` and `mmssms.db` files are used for storing call logs, address book records, and SMS messages. If used, this feature sets the value of fields found in pages belonging to these two files with a 1, and a 0 otherwise.

3. **Text false positives.** Smart phones contain a large amount of text, some of which is user created, but most is detritus such as cached Web pages, application resources, and high level code or configuration. We use a set of fixed rules to filter out these often-seen false positives. First, we assign a feature value of 0 to text fields that are three characters or fewer in length. Second, we look for fields that contain HTML documents and other bits of code. We give a value of 0 to fields using CamelCase or words commonly found in code (e.g., `SELECT` or `div`). We generated this list ahead of time using information found on unused, freshly installed Android phones. For efficiency, we only calculate this feature value for pages that have a non-zero score for one of the above two features.

At first blush, it may seem that with the file system feature we are providing the answer key to LIFTR and no work remains. However, these two files contain a lot of information that is not easily recoverable by DEC0DE, Bulk Extractor, or `Strings`, and much of what is recovered is not useful (e.g., the metadata and portions of the schema). It's important to note that we evaluate LIFTR primarily using unallocated pages, under the assumption that the phone's information has been deleted and reinstalled (for many of our phones, this was actually the case). The original files are not easily reconstructed from these pages because not all pages are present, and it is not always clear if a page belongs to a particular file. We could have written a specialized SQLite fragment parser for these files, but our goal is to provide a general technique that makes use of only the fields output by recovery engines — in turn, those engines also seek to recover information in a way that is independent of the file type being recovered, without solving the difficult problem of generalizable file reconstruction. In short, these engines also seek to be generalizable to the largest number of scenarios. Section 5 contains more details on our recovery of unallocated pages.

3.2 Relevance Feedback Stage

LIFTR's initial ranking orders only those pages with a non-zero *a priori* or file system score; all other pages will have a zero quality score after initial ranking. During the relevance feedback stage, LIFTR refines the initial ranking by using investigator feedback.

At a high-level, our algorithm works as follows. LIFTR presents the investigator with the top-ranked page and asks him to label all of the relevant fields on that page. Using the positive labels, LIFTR then increases the quality score for all semantically related pages based on how strongly each relates to the current page. Then, LIFTR updates the page ranking, and it asks for feedback on the top-ranked page that hasn't been sent to the investigator. This cycle is repeated as many times as the investigator wishes. We discuss the details of each step below.

In general, the more feedback, the better the final ranking; but we find that the investigator need only label a few pages to bring about significant improvements in the page order. Further, the number of fields per page is typically very small with around 24 fields per page on average for DEC0DE, and only 7 useful fields per relevant page.

The details of the process are as follows.

1. Marking fields. As part of our implementation of LIFTR, we have written a small interface that makes it easy to quickly label the fields on a page. An investigator marks entire fields as relevant or not. LIFTR then breaks fields into *tokens* by splitting on whitespace and punctuation. (Phone numbers are not split by hyphens or other punctuation.) When the investigator marks all fields on a page, we say that a *round* has completed.

In sum, for a given page p, F_p is the set of all fields. We let K_p be the set of *all* tokens from all fields on page p. We say a token is relevant if the field it came from was marked as relevant. We let $T_p \subseteq K_p$ be the subset of all tokens that are *relevant* on a page p. To be clear, once a field (and thus its derivative tokens) is marked relevant on any page, it is relevant for all pages on the phone.

2. Finding semantically related pages. We consider two pages to be *semantically related* if the pages share a token that has been previously marked as relevant and is not in a token blacklist (described below). The intuition here is that once the investigator marks a few tokens as relevant (whether names, email addresses, or phone numbers), we should evaluate other pages that share this content.

In sum, we let $\mathcal{R} = \{r_1, \ldots, r_i\}$ be the ordered set of pages that have been marked by the investigator, where r_i is the page marked in the ith round. We let $\mathcal{P}$ be the unordered set of pages as-yet-unmarked that share at least one token with a field from a page in $\mathcal{R}$, where the field has been marked as relevant.

3. Blacklisting tokens. LIFTR maintains a blacklist of tokens to exclude when calculating the quality score for the pages. Tokens are added to the blacklist for one of three reasons: *(i)* they are found in a natural language dictionary; *(ii)* they match a token found on a set of newly installed and unused Android phones; *(iii)* they are present in a field not marked as relevant for a page in $\mathcal{R}$. We let $\mathcal{B}$ be the set of blacklisted tokens.

The negative feedback helps to identify tokens that have no connection to the user and hence are of no interest to the investigator. This becomes important when a positively marked field string contains irrelevant tokens. For example in the string "I called John Doe today", the tokens "John" and "Doe" are highly relevant whereas "I", "called", and "today" provide little information. This approach prevents LIFTR from giving undue credit to pages bearing semantic relations to such tokens.

4. Calculating the new quality score. When a field is marked by the investigator as relevant on the current page, LIFTR increases the quality score for the pages in $\mathcal{P}$ as follows. For a given $p \in \mathcal{P}$, each $\tau \in T_p$ contributes a quality score proportional to its *Inverse Document Frequency (IDF)*. The IDF of a token is a measure of how often a token appears in the set of all pages, and it is calculated as:

$$\mathrm{idf}(\tau) = \log\left(\frac{|\mathcal{P} \cup \mathcal{R}|}{|P_\tau|}\right), \text{where } P_\tau = \{q | q \in \mathcal{P} \cup \mathcal{R}, \tau \in K_q\} \quad (2)$$

Note that P_τ is the set of pages in $\mathcal{R}$ and $\mathcal{P}$ that contain τ. We use IDF as a means of reducing the impact of tokens that appear too frequently throughout the phone. It allows us to have investigators mark whole fields without distinguishing separate tokens, and yet the impact of irrelevant tokens present in relevant fields is diminished. For instance, we find that many frequently occurring irrelevant tokens, that are not in the blacklist $\mathcal{B}$ are specific to the phone at hand and hence no generalized rules can be hard-coded to avoid them. However, owing to high frequency, they have a low IDF score and would therefore have minimal impact on the quality scores.

Once a new page p has been marked by the investigator in round i (and moved from $\mathcal{P}$ to $\mathcal{R}$), all the pages remaining in $\mathcal{P}$ are re-scored. The quality score for a page $q \in \mathcal{P}$ after i rounds of relevance feedback is the sum of quality score from the previous round and the IDF scores for each relevant token that q shares with p (ignoring those tokens found in the blacklist $\mathcal{B}$).

$$\mathrm{quality}_i(q) = \mathrm{quality}_{i-1}(q) + \sum_{\tau \in T_q - \mathcal{B}} \mathrm{idf}(\tau) \quad (3)$$

The value of $\mathrm{quality}_0(q)$ is the initial ranking score for q, as shown in Equation 1. Note that $\mathrm{quality}_0(p) = 0$ for pages that did not appear in the bootstrap set. Also, tokens are only evaluated in the rounds during which they first appear; e.g., if a token "foo" appears in page r_1, then it is ignored if it later appears in r_2.

Presentation Order. We refer to the order in which LIFTR presents pages to the investigator as the *presentation order*. Currently, we present the unlabeled pages with the highest current quality score. While this approach is not necessarily the most efficient for minimizing the number of questions that we need to ask the investigator, we find it is an effective heuristic.

4. LIFTR EVALUATION

In this section, we evaluate LIFTR's initial ranking and relevance feedback stages. Broadly, we focus on three important questions:

1. How effective is LIFTR at ranking relevant pages using different recovery engines?
2. How much feedback is required for LIFTR to be effective, and how does that translate to time required of investigators?
3. What page properties have an effect on relevance feedback?

We discuss each of these questions below.

4.1 Evaluation Methodology

LIFTR's goal is to help focus an investigator's limited resources by identifying pages that are most likely to contain information relevant to the investigation. It works in concert with existing inference approaches to seamlessly increase precision and scale. Once we have identified the important pages, the investigator can then perform a more thorough examination and perhaps employ more specialized techniques to extract additional information.

For our experiments we simulated feedback from an investigator, such that a field on a page is marked relevant if it contains a token also present in the `contacts2.db` or the `mmssms.db` file, e.g., a name, phone number, or email address.

Inference Engines. We tested the effectiveness of LIFTR to improve the results of three different triage techniques:

1. **DEC0DE** [21], a probabilistic parsing approach that supports many types of data commonly found on mobile phones.
2. **Bulk Extractor** [2,6], a regular-expression based approach originally designed for desktop systems.
3. **`Strings`**, a common UNIX utility for identifying strings of printable characters in a file.

In many ways, DEC0DE and Bulk Extractor represent opposite design philosophies for recovering data. While DEC0DE is designed to be very flexible and prioritize the recovery of all information at the expensive of false positives, Bulk Extractor uses strictly defined regular expressions to limit the amount of irrelevant information, paying the cost of decreased true positives.

`Strings` represents a simple baseline approach that makes minimal assumptions about the structure of the underlying data. LIFTR's good performance with `Strings` demonstrates that the effectiveness of our system does not depend on the underlying engine.

We did not modify Bulk Extractor or `Strings` before applying them to smart phones, and made only minor changes to the DEC0DE workflow. DEC0DE typically groups the inferred fields into records — e.g., nearby text and phone number fields might be grouped together as an address book entry — and returns a list of these records to the investigator. We found it necessary to forgo the record step and only consider the field-level results. Records, as they are defined by DEC0DE, are centered around phone numbers existing in close proximity to names and other identifying information. However, since Android uses relational databases for storing data, the name for an address book entry will not necessarily appear in close proximity to the phone number in memory.

With careful modification, it is likely that DEC0DE and Bulk Extractor will perform significantly better than what we observed. For example, we could add additional regular expressions to Bulk Extractor that are specifically targeted toward Android data. However, our intention was not to measure the performance of the inference techniques, but instead to evaluate the effectiveness of LIFTR and demonstrate how it can benefit each of the recovery engines.

Phones. A partnering research group provided us with both the logical and physical images of 13 previously owned phones[2]. These 13 Android phones, listed in Figure 2, were from 6 different manufacturers with users that lived in Canada, Hungary, India, Israel, Singapore, Serbia and the US. Each of these phones contained varying amounts of residual data from their previous users.

Before applying the recovery engines, we pre-filtered each physical image, to *(i)* reduce the amount of data passed to the inference tools, and *(ii)* when it could be identified, limit our analysis to the user data partition in the image. For 8 of the 13 phone images, we applied a filtering technique based on the Yaffs file system; we describe this approach in more detail in Section 5. For the remaining 5 phones phones, we employed the block hashing filtering approach proposed by Walls et al. [21].

[2] We received IRB approval for this process. Phone images provided by Simson Garfinkel at the Naval Postgraduate School.

	DEC0DE				Bulk Extractor				`Strings`			
	Pages	True Pages	Fields	True Fields	Pages	True Pages	Fields	True Fields	Pages	True Pages	Fields	True Fields
HTC Desire HD	228,860	1,834	6,247,115	12,275	518	55	5,233	216	189,182	1,629	6,509,666	13,875
Samsung Galaxy Y	175,873	600	4,594,060	2,954	263	0	694	0	139,109	1,526	3,747,699	26,492
Motorola XT701	144,618	207	3,049,543	869	118	0	650	0	120,923	207	2,762,831	984
HTC Evo 4g	78,680	3,178	1,853,638	22,850	2,689	323	21,806	1,250	68,216	2,761	1,774,627	18,243
Samsung Galaxy Y Duos	60,422	3,612	1,530,630	21,399	2,150	1,715	18,550	14,235	51,324	4,093	1,281,450	35,334
HTC Wildfire	55,759	137	1,019,033	837	118	36	1,031	321	46,360	115	1,176,778	872
Samsung Galaxy Mini	53,456	1,390	1,329,715	4,031	1,329	275	5,599	1,139	40,288	434	1,020,422	2,134
Sony Xperia x10	30,063	16	855,464	154	20	4	57	22	22,447	22	958,791	159
Dell XCD35	20,494	221	594,469	1,621	90	24	528	49	16,191	222	539,018	2,268
Dell XCD28	17,932	118	495,099	841	27	6	190	20	13,189	78	433,917	734
HTC Legend	10,238	19	198,873	121	26	0	28	0	7,721	14	194,448	108
Huawei Ideos	8,829	667	170,143	6,799	81	9	1,106	15	7,777	683	202,654	7,022
Huawei 8500	6,305	3	116,524	41	12	0	28	0	5,764	3	135,157	50

Figure 2: Page and field statistics for the three recovery engines. On average, less than 2% of the results returned by the recovery engines are relevant; for more than half of the phones this percentage is below 0.5%.

Figure 2 shows the number of pages returned by each of the tested inference techniques. For DEC0DE and `Strings`, the number of returned results is typically orders of magnitude greater than the actual number of relevant results. The percentage is less drastic for Bulk Extractor, but the number of fields returned by the tool is still often in the hundreds or thousands.

Because we did not have direct access to the phones, our evaluations were limited to the provided images, i.e., we could not manually manipulate the phones. However, this limitation mimics the restrictions placed on real world investigations, wherein the investigator must take great care not to modify the phone (beyond the extent necessary to extract the image).

Ground Truth. We collected ground truth from the contacts and SMS databases taken from the logical image of each of the phones. That is, we used tokens collected from the *allocated* versions of the SQLite databases `contacts2.db` and `mmssms.db`, respectively.

We considered a field provided by a recovery engine to be *relevant* if it contained a token — name, phone number, email address — found in one of the two ground truth databases. For example, if the allocated version of the `contacts2.db` file included the number 212-555-0123, we considered any field from any page containing this number as relevant. Consequently, relevant fields are not limited to the pages in contacts or SMS databases. We described the field tokenization process previously in Section 3.2.

LIFTR presents results to the investigator at a page-level granularity. We define a *relevant page* as any page that contains at least one relevant field — a field with a token pertinent to the investigation. For our evaluation, field types were either phone numbers, names or email addresses.

Limiting our ground truth to pages that have tokens belonging to the SMS and contacts databases gives us a lower bound on the amount of data present on the target phone. While there are other potential data sources we could draw our ground truth from (e.g., the geolocation database), grabbing this information would require modification to DEC0DE and Bulk Extractor. Further, our goal is to test the effectiveness of our feedback approach, and not that of the recovery engines. In other words, DEC0DE may not correctly infer all of the relevant fields on the phone. When evaluating our ranking, we only consider the relevant fields that each recovery engine was able to identify.

Initial Ranking Features. In our experiments, we consider two combinations of the initial-ranking features from Section 3.1: *(i)* file system knowledge paired with text quality and *(ii) a priori* information paired with text quality. For the *a priori* method we use all of the phones from our set, 13 in total. For the file system feature, we limit the evaluation to the 8 phones with a parsable Yaffs user data partition. We used a feature weight of 0.6 for file system and *a priori*, and 0.4 for text quality.

Our set includes 9 phones with the Yaffs file system. We wrote a special Yaffs parser to identify expired pages and label each page with its likely filename; we describe this process in Section 5. One of nine Yaffs phones could not be parsed properly by our techniques. This is because we were unable to separate the user data partition from the rest of the physical image. The other four phones used either Ext4 or RFS (a variant of FAT) and their physical images were acquired below a flash translation layer, making it impossible for us to label the pages with their suspected file names without knowledge of the FTL's mapping algorithms. In the non-Yaffs phones, LIFTR was not aware of the file to which a page belonged.

For the *a priori* setting, we select five relevant tokens randomly from the pool of all relevant/user related tokens like a name, phone number or an email address. The initial sorting score of pages to which these tokens belong would be increased, causing them to rise above the rest. The performance of LIFTR using the *a priori* approach is averaged over 30 trials for each of the 13 phones. In the setting where we use file system knowledge, the initial sorting scores of pages associated with user-content rich files like contacts and SMS are stepped-up instead, causing such pages to improve their ranks. Leveraging file system information, being a deterministic approach, needs to be run only once for each phone.

We tried a combination of all three ranking features, but found that it does not provide substantial improvement over the combination of *a priori* and text quality. We suspect this lack of improvement is due to our evaluation methodology wherein all simulated *a priori* knowledge was on the phone. In a real-world scenario, it is possible that only a subset of the investigator's *a priori* knowledge appears in the recovered fields.

Filtering. We filtered the parsable Yaffs images to only include the expired pages; see Section 5 for details. For the other phones, we used block hash filtering, which cannot distinguish between current and expired pages.

Normalized Discounted Cumulative Gain. We use *normalized discounted cumulative gain* (NDCG) [8] to measure the effectiveness of LIFTR's sorting. An NDCG score of 1 means an ideal ranking, that is, all of the top *k* pages are relevant. An NDCG score of 0 means the worst possible ranking, where none of the top *k* pages are relevant. Here, *k* is a *cutoff rank*, which depends on the expected number of relevant results and on how many top-ranked results would the user be willing to sift through. More formally, the discounted cumulative gain at rank *i* is More formally, the discounted cumulative gain at rank i is calculated as follows.

$$\mathrm{dcg}(i) = v_1 + \sum_{j=2}^{i} \left(\frac{v_j}{\log_2 j}\right) \qquad (4)$$

where $v_j = 1$ if the page at rank j is relevant, and 0 otherwise. We obtain the normalized DCG score by dividing the DCG score by the maximum possible score for some cutoff k.

$$\mathrm{ndcg}(k) = \frac{\mathrm{dcg}(k)}{1 + \sum_{j=2}^{k} (1/\log_2 j)} \qquad (5)$$

Order matters for NDCG. For example, given a ranked list A, where only the top half is relevant, and another ranked list B, where only the bottom half is relevant, then $\mathrm{ndcg}(A) > \mathrm{ndcg}(B)$, assuming A and B are the same length.

4.2 Impact of Initial Sorting and Relevance Feedback

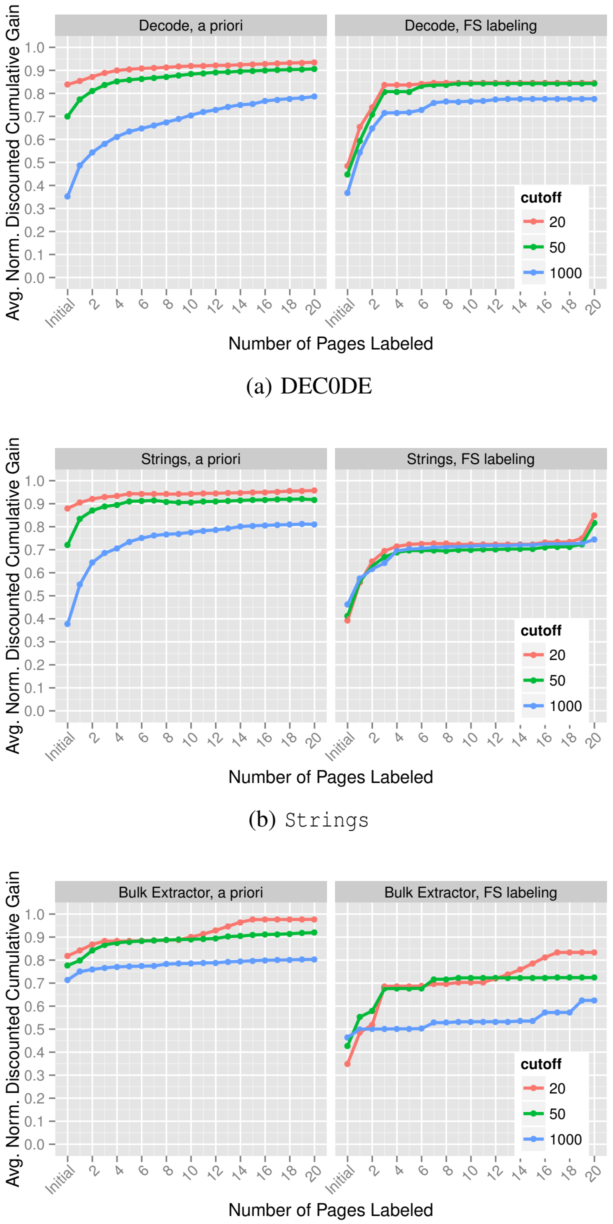
Figures 3(a), 3(b), and 3(c) show the NDCG results for LIFTR using DEC0DE, `Strings`, and Bulk Extractor, respectively, averaged over all phones.

Initial Scoring. DEC0DE, Bulk Extractor, and `Strings` all produce unsorted results: their NDCG scores are near to zero. LIFTR's initial sorting is significantly better. Using 5 tokens of *a priori* information provides a significant benefit for LIFTR's initial sorting stage, resulting in an average NDCG of 0.73 across all three recovery engines for $k = 50$. The NDCG for initial sorting is denoted by the "Initial" tick mark on the far left side of x-axes in Figures 3(a), 3(c), and 3(b). Initial sorting with file system information gives an average NDCG of 0.43, for $k = 50$, across the three recovery engines. This discrepancy is due in part to the number of pages that belong to the contacts and SMS databases that are not actually relevant. For example, many of these pages contain schema related information and other database metadata. Giving a higher initial score to contacts and SMS databases has the effect of also wrongly benefiting such irrelevant pages belonging to these files.

Relevance Feedback. The experiments demonstrate that the relevance feedback stage results in a significant improvement in the NDCG scores with a small amount of feedback. For example, after labeling just 5 pages, the score for the *a priori* scenario rises from an initial NDCG value of 0.73 to 0.88, for $k = 50$; and from 0.48 to 0.71, for $k = 1000$.

The graphs for DEC0DE and `Strings` also depict the differences in increases of the NDCG score when given more feedback, for different cutoff ranks. There is a steeper increase for $k = 1000$ as compared to $k = 20$ or $k = 50$. This shows that the initial ranking algorithm is able to fill up the top few ranks with relevant pages (high NDCG at initial for $k = 20$), however the initial sorting incorrectly ranks a significant number of irrelevant pages over relevant pages at ranks above 50. We see that the relevance feedback quickly corrects these mistakes (steep increase in NDCG for $k = 1000$). This demonstrates the importance of relevance feedback in LIFTR. At the same time, it also establishes the need for having initial ranking for bootstrapping the feedback stage. A few relevant pages in the early ranks greatly increases the pace at which LIFTR discovers other good pages, and helps LIFTR ignore the large pool of bad ones.

Variation. Each line in Figures 3(a), 3(c), and 3(b) is an average of all phones for the given experiment. The average of all NDCG scores allows us to aggregate many experiments into these three figures. There is variation among phones that is hidden by the average NDCG, but it is not representative of the variation within each phone. Figure 4 shows the per-phone NDCG values for DEC0DE using the *a priori* approach, for a cutoff of $k = 1000$ (i.e., the blue line in the left plot of Figures 3(a)). Error bars show 95% confidence intervals over 30 trials, each using a randomly selected set of 5 tokens. For ten of the phones, relevance feedback always improves the NDCG. For three phones, the relevance feedback approach provides no advantage.



(a) DEC0DE

(b) `Strings`

(c) Bulk Extractor

Figure 3: The average NDCG score for DEC0DE, `Strings` and Bulk Extractor. The high initial score for $k = 20$ shows that LIFTR's *a priori* and *file system* ranking effectively places relevant pages early in the list. These top pages bootstrap the feedback process, enabling LIFTR to discover the large number of remaining pages — evidenced by the steady NDCG increase for $k = 1000$. LIFTR's results are consistent for all three recovery engines.

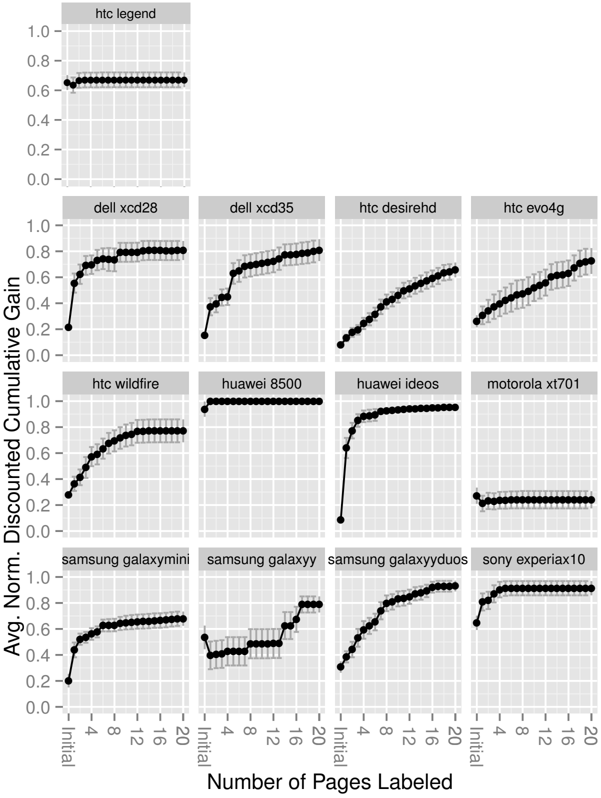


Figure 4: Per-phone average NDCG results for DEC0DE using the *a priori* approach, for a cutoff of $k = 1000$. Error bars show 95% c.i. across 30 trials per point. For most of the phones, relevance feedback consistently improves the NDCG score.

Discussion. The average NDCG score for all approaches and settings increases with feedback but does not reach its maximum value of 1. There are two possible reasons for this limitation. First, there are certain relevant pages on the phone that are not semantically related to other relevant pages and so no amount of feedback can guide LIFTR in identifying such pages that are sitting by themselves. Second, the NDCG score for a cutoff of k is not only affected by the number of relevant results among the top k, but also the positions at which they appear. Even if LIFTR is able to rank the relevant pages among the top k positions, unless all of those pages appear before all irrelevant pages, the score will be less than unity.

It is also important to note that the pages recovered during relevance feedback are not limited to those belonging to contacts and SMS databases. They could include pages associated with other files that have relevant tokens. For instance, some phones have user data in the SQLite Write Ahead Log file, which is like a rollback journal used by SQLite for atomic commits and transaction rollbacks. This shows that simply extracting contacts and SMS database files would not yield all the relevant data on the phone.

It is also interesting to note that the NDCG plot for Bulk Extractor is not as smooth as DEC0DE or `Strings`, due to the fact that only a few phones have have more than 50 pages and fewer have more than a 1000 pages among the inference results after using Bulk Extractor. Hence, the NDCG values are averaged across fewer phones.

4.3 Measuring Investigator Work

The intuition behind LIFTR's feedback approach is that an investigator can perform a small amount of manual analysis to greatly improve the overall quality of the returned results.

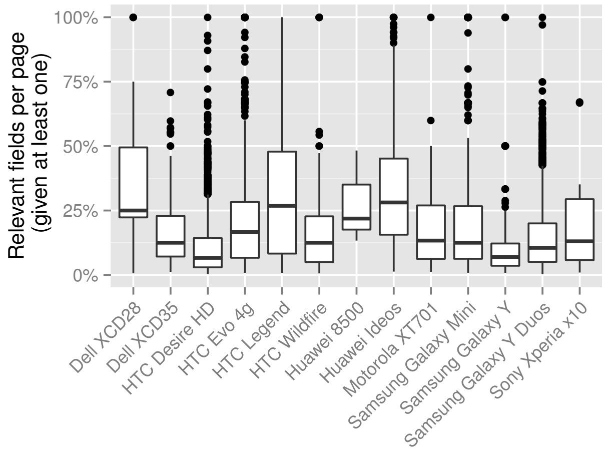


Figure 5: The percentage of relevant fields averaged across all relevant pages. The majority of fields returned by the recovery engines — even for relevant pages — are false positives.

For our evaluation, we measure investigator work in terms of the number of pages that they must manually label. Because a single page may require multiple labels, we differentiate between *(i)* the total number of pages that the investigator labels and *(ii)* the number of fields that the investigator must label. Recall from Section 3.2 that an investigator need only label the positive fields, as LIFTR will default to marking everything it has shown the investigator as a false positive. On average, this approach leads to 7 labels per relevant page. If we assume that it takes an investigator 5 seconds marking a label, then it would take him around 35 seconds per page, and around 11 minutes to complete 20 page. As our results for the three recovery engines suggest, 20 pages is sufficient for most phones to achieve an NDCG of at least 0.8.

We examined another approach where the investigator marked whole pages as relevant or not, rather than the individual fields on each page. This alternate approach did not perform as well. Figure 5 shows why: among pages with at least one relevant field, typically less than half of the fields on the page are relevant.

4.4 Strongly Related Pages

Our technique is most effective when the investigator provides positive labels to pages that are semantically well connected to other pages, owing to the co-occurrence of relevant tokens among them.

In order for relevance feedback to be effective within i pages of feedback, the initial sorting must include a relevant page in the top i result pages. We find that it is more effective to bring up true positive pages than it is to penalize false positive ones. This is largely because irrelevant pages greatly outnumber relevant ones; 20 pages of feedback are not sufficient for identifying most of the irrelevant pages or tokens.

5. RESIDUAL DATA IN ANDROID

In this section, we detail our techniques for parsing and analyzing the Yaffs file system. We use these techniques with LIFTR to more effectively filter the physical images, and to provide the file labels for initial sorting.

We also show that significant amounts of data, including user information, persists in the expired segments of the phone's memory for periods as long as weeks or months. Further, this expired data makes up more than half of the NAND pages for some phones.

Since this data cannot typically be recovered by parsers that aim at reconstructing the file system, file system agnostic inference techniques used by recovery engines like DEC0DE and Bulk Extractor are a potential solution to the problem. The sheer amount of irrelevant results returned by these engines, however, highlights the importance of using LIFTR for quickly locating information with evidentiary utility, from these results.

NAND flash does not overwrite data in place. Instead, when an object is modified, the data is written to a new page, leaving the expired chunk in storage. These expired pages persist for an indefinite period of time. This residual data offers interesting opportunities for forensic triage, as the old pages contain deleted data and can potentially be used to track the changes over time. Our results are consistent with previous work in secure data deletion on flash memory [14,15,22].

5.1 Yaffs Overview

Yaffs, common on Android phones prior to Gingerbread, is a log-structured file system designed to work with NAND storage. The file system treats everything as an object or piece of an object. Each Yaffs object is stored as a sequence of *chunks*. Typically, a chunk is equivalent to a NAND page. Yaffs uses *header chunks*, to store object metadata such as the object type, timestamps, and permissions, and *data chunks* to store the actual bytes of a file. Non-file objects such as directories and hard-links, are made up of just a single (header) chunk. In Yaffs, chunks are the unit of write and *blocks* are the unit of erasure. Blocks are made up of contiguous chunks in memory, often 64 chunks to a block.

Chunks may be in one of three states: erased, expired, or current. A chunk is considered *erased* if it resides on an erased block, that is, the block is devoid of any data and contains all `0xFF` values. In contrast, *expired* chunks contain once-valid data that has been replaced by a more recent chunk. These chunks either belong to deleted objects or are old pieces of current objects. Both erased and expired chunks are ignored by the file system.

Finally, *current* chunks contain the most recent version of each object chunk. Yaffs keeps track of all of the current chunks and uses them to reconstruct all objects in the file system.

When a file is modified, one or more chunks transition from the current state to expired. Eventually, through a garbage collection process, Yaffs will reclaim storage space by erasing an entire block of expired chunks. Recall that the unit of erasure in NAND is a block. However, it is common for expired and current chunks to reside on the same block.

Upon phone startup, Yaffs scans through all of the chunks to reconstruct the file system state. In order to determine which chunks are current and which are expired, Yaffs uses the *out-of-bounds area* (OOB) — a small region of memory adjacent to each page — to store file system metadata. This metadata includes the *block sequence number* which is assigned when a block is picked for writing. Because Yaffs always fills the current block before moving to the next block (in absence of power loss or shutdown), the block sequence number, and the order of chunks within the block, represents a temporal ordering: the higher the sequence number, the more recently the block was written. Note that blocks with consecutive sequence numbers may not be physically adjacent in memory.

5.2 Breakdown of Chunk Composition

A significant portion of the phones consists of residual data, with 56% on average for the set of pre-owned phones. Figure 6 shows the fraction of different chunk types for a set of Android phones. As we discussed above, Yaffs chunks are either headers or data and are always in one of three states: erased, expired, or current. This gives us the 5 chunk categories shown in Figure 6.

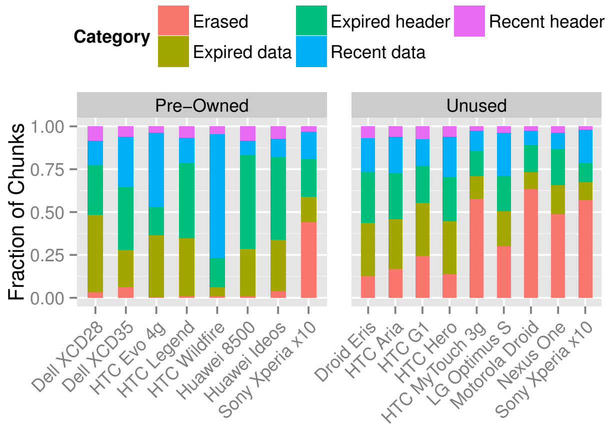


Figure 6: Fraction of each chunk type for the user data partition. On the pre-owned phones, over half (56%) of all storage belongs to expired memory chunks.

The phone set includes 8 of the pre-owned phones from Section 4 (those with a parseable Yaffs user data partition), in addition to 9 unused phones loaded with synthetic data[3]. We limit our analysis to the user data partition of each phone.

On average across both the unused and pre-owned phones, roughly 26% of the chunks are current, with 27% erased and the remaining 47% expired data. Pre-owned phones have a greater fraction of current chunks than the unused phones: an average of 31% versus 20%, respectively. And pre-owned phones have fewer erased chunks than unused phones: an average of 12% versus 43%, respectively. The difference in erased chunks is due to the garbage collection process. Garbage collection is expensive and typically only erases blocks as needed.

By definition, each object may only have one current header chunk and, if it is a file object, one or more current data chunks. On average across the used phones, current headers make up only 5% of the chunks whereas current data chunks make up roughly 26%. Interestingly, the ratio of header to data chunks is significantly higher in the expired data. For the pre-owned phones, expired headers make up 30% and expired data chunks make up 27%. Anytime an object is modified, whether it be the object's metadata or actual contents, Yaffs will write a new header chunk. In this way, expired headers track changes to an object over time.

Much of the storage space is dedicated to SQLite databases and associated files. The majority of the expired chunks belong to deleted objects, typically SQLite temporary files such as the journal files used for rollbacks.

5.3 Filtering Images Using Yaffs

Recovery engines such as Bulk Extractor or DEC0DE are designed to be able to recover information from deleted (i.e., expired) portions of an image. With information from Yaffs metadata, we can more effectively filter the phone image to focus on the expired chunks. We implemented a parser to identify the unique set of expired data chunks, and for the 8 pre-owned phones, we were able to reduce the size of the images by an average of 76% for our experiments in Section 4. Note that expired header chunks contain only

[3] We obtained the 9 phones with synthetic data from Via Forensics: `https://viaforensics.com/`.

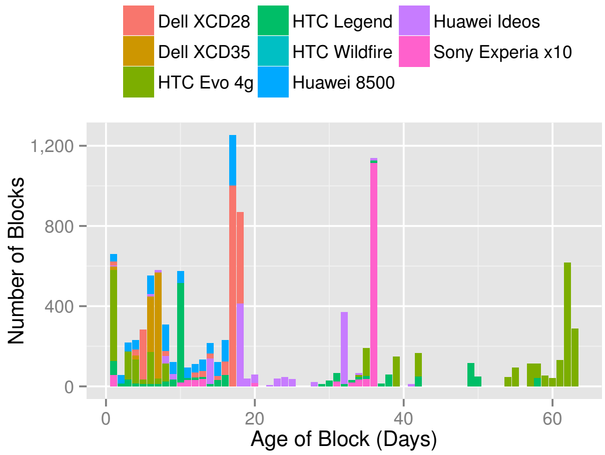


Figure 7: The age of blocks, in days, for the user data partition of pre-owned phones. Blocks remain in memory for weeks or months after originally written, with half older than 14 days and a quarter older than 34 days.

outdated metadata, rather than the type of information the inference engines and investigators target. Because the header data and allocated chunks are interleaved with expired data, a technique that is blind to the file system, such as block hash filtering [21] would not be able single out the expired chunks.

5.4 Block Churn

Our analysis of the 8 pre-owned phones showed that blocks remain in memory for weeks or months after originally written. Any expired chunks in those blocks are accessible via DEC0DE or other recovery engines. We describe our findings and the process for estimating block age below.

The block sequence numbers provide a temporal ordering of blocks in the file system; the block with the highest sequence number was written most recently while lowest number block is the oldest. However, the sequence numbers do not directly tell us the exact write time of a block.

More precisely, a block's *write period* is a time range between when the first and last chunks were written to the block. The block write period is useful for *(i)* providing a bound on when expired chunks were valid and *(ii)* as a means for estimating the rate at which blocks are written.

Header chunks contain object timestamps. If a block contains a header, we can use that header to help estimate the block's write period. If a block does not contain a header, we can use the time estimates for adjacent blocks (by sequence number).

At a high level, we employ the following algorithm to estimate the write period for each block. First, for each block we record the object timestamps stored in any header chunks that are present on the block. Header chunks are written anytime an object is created or modified, and contain three timestamps: `atime`, `mtime`, and `ctime`. Unlike the Unix equivalents, `atime` does store last access time by default. Typically, `atime` stores the creation time of the object. The write time for the header chunk is then the greater of either the `ctime` or `mtime`, ignoring the effects of caching which appear to be negligible in practice. The block's write time is bounded by the oldest and most recent header write times.

When a block does not contain any header chunks, we have to estimate the write period using adjacent blocks. For example, consider blocks b_1, b_2, b_3 with sequence numbers 1, 2, and 3 respectively. Assuming, we already have write period estimates for b_1 and b_3 (calculated using the header chunks). The sequence numbers give us the write order for the three blocks. In other words, b_2 must have been written after b_1 but before b_3. Therefore, the write period of b_2 must be between the last write of b_1 and the first write of b_3.

In practice, we observe that some header chunks used seemingly inconsistent timestamps, e.g., some headers stored a modification time that was before the object's creation time. This is due in part to garbage collection, as we discuss further below. To avoid this issue, we only use timestamps from newly created objects. Focusing solely on new objects does not represent a limitation as phones frequently create new and temporary files. We can find newly created objects by looking for expired headers such that the header is the oldest chunk found for an object and the number of bytes listed in the OOB field is zero.

Figure 7 shows the block ages for the pre-owned phones. We calculated the block age relative to the age of the most recent block of each phone. For the HTC Evo 4g 37% of the blocks were over 60 days old. Across all phones, half of all blocks were older than 14 days, and a quarter were older than 34 days.

Garbage Collection. Yaffs periodically copies current chunks to new blocks, freeing the original block for deletion. These chunks are copied exactly, so any moved header chunks will retain their old timestamps. Consequently, chunks moved due to garbage collection may appear to be newer than they actually are. For example, imagine the most extreme case in which a data chunk is written early in the phone's life, but always remains current because the file it belongs to never changes. Over time that chunk will be moved to new blocks as the result of garbage collection with each subsequent block having a higher sequence number. In other words, there may be a significant difference between when a chunk was written to a given block and when the chunk was originally written.

Because blocks sequence numbers are always incremented, we can estimate the rate of block deletion and garbage collection by looking at the missing sequence numbers in the image.

5.5 Inferring Yaffs Parameters

Before we can analyze the Yaff image, we have to infer the important Yaffs parameters that may be different for each phone. For each phone image we need to infer the block size, chunk size, OOB size, and OOB tag offset. Knowing the OOB tag offset enables us to parse the Yaffs metadata store in the OOB, e.g., the block sequence number and object identifier.

This OOB offset is not directly controlled by Yaffs. We can quickly determine this offset by comparing the OOBs for different headers of the same file object. For the same object, the object identifier should remain consistent across different header chunks. This approach relies on file objects that are unlikely to have been deleted, e.g., the contacts database.

To estimate the Yaffs parameters, we scan the image for header chunks belonging to a file known to be on the image. For example, the `contacts2.db` file is present on most Android phones. We repeat the scan assuming a variety of different parameter values. The most likely parameters are those that produce the most valid file headers. In practice, all of the phones we examined used a chunk size of 2048 bytes, an OOB of 64 bytes, and a block size of 64 pages; however, we saw at least three different OOB offset values.

6. RELATED WORK

In addition to DEC0DE and Bulk Extractor, LIFTR is related to prior work in acquiring physical images from mobile phones [13,19, 20] and Android file system analysis [3,17].

Park et al. [12] propose a technique for clustering and recovering fragmented SQLite records of the same file residing in expired pages, without parsing or recreating the underlying file system. LIFTR could be used to augment their technique.

Beebe et al. [1] implement an unsupervised learning algorithm for clustering results from simple text search queries on raw data, using self-organizing neural networks. In contrast, LIFTR is a supervised approach, and would complement their mechanisms.

Foster [4] propose a file system independent technique, sector hashing, for identifying if a target file was ever present on a storage device. The work does not target user-generated information.

Marturana et al. [10,11] use machine learning models to determine if copyright-infringing data or child abuse materials are present on a mobile device. Such models are more difficult to train in a general setting, where the investigator is interested in user-specific details which typically lack distinctive features.

7. CONCLUSION

We proposed the use of relevance feedback to quickly pinpoint information relevant to an investigation. Our system, LIFTR, addresses a major issue limiting current forensic triage techniques: only a small fraction of the information returned by recovery engines is relevant to an investigation.

When applied to smart phones, the recovery engines returned hundreds of thousands of results, most of which were irrelevant. LIFTR overcomes this limitation by finding those pages that are most likely to contain useful information, getting feedback from the investigator, and using that information to rank the results. Further, we show how a small amount of background information about a suspect can greatly improve LIFTR's performance.

Our evaluation was performed using Android phones from a variety of makes and models. We tested LIFTR with three different recovery engines, each differing significantly from the others in its inference approach. Our results demonstrate that feedback on as few 20 NAND pages is more than sufficient to identify the top 100 most relevant pages out of the hundreds of thousands of false positives returned by the recovery engine.

Acknowledgements. We thank Simson Garfinkel for making the corpus of phones available to researchers. This work was supported in part by the Office of Naval Research (N00244-12-1-0057).

8. REFERENCES

[1] N. L. Beebe, J. G. Clark, G. B. Dietrich, M. S. Ko, and D. Ko. Post-retrieval search hit clustering to improve information retrieval effectiveness: Two digital forensics case studies. *Decision Support Systems*, 51(4):732–744, 2011.

[2] R. Beverly, S. Garfinkel, and G. Cardwell. Forensic Carving of Network Packets and Associated Data Structures. In *Proc. DFRWS Digital Forensics Research Conference*, pages 78–89, Aug. 2011.

[3] K. D. Fairbanks. An analysis of Ext4 for digital forensics. In *Proc. DFRWS Digital Forensics Research Conference*, pages 118–130, August 2012.

[4] K. Foster. Using Distinct Sectors in Media Sampling and Full Media Analysis to Detect Presence of Documents from a Corpus. Master's thesis, Naval Postgraduate School, Monterey, California, September 2012.

[5] S. L. Garfinkel. Carving contiguous and fragmented files with fast object validation. In *Proc. Digital Forensic Research Workshop (DFRWS)*, pages 2–12, August 2007.

[6] S. L. Garfinkel. Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, 32:56–72, 2013.

[7] A. Hoog. *Android forensics: investigation, analysis and mobile security for Google Android*. Elsevier Science, 2011.

[8] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[9] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. DIMSUM: Discovering Semantic Data of Interest from Un-mappable Memory with Confidence. In *Proc. ISOC Network and Distributed System Security Symposium*, February 2012.

[10] F. Marturana, G. Me, R. Berte, and S. Tacconi. A Quantitative Approach to Triaging in Mobile Forensics. In *Proc. IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 582–588, July 2011.

[11] F. Marturana and S. Tacconi. A machine learning-based triage methodology for automated categorization of digital media. *Digital Investigation*, 10(2):193–204, 2013.

[12] J. Park, H. Chung, and S. Lee. Forensic analysis techniques for fragmented flash memory pages in smartphones. *Digital Investigation*, 9(2):109–118, 2012.

[13] D. Quick and M. Alzaabi. Forensic analysis of the android file system yaffs2. In *Proc. Australian Digital Forensics Conference*, 2011.

[14] J. Reardon, S. Capkun, and D. Basin. Data node encrypted file system: Efficient secure deletion of flash memory. In *Proc. USENIX Security Symposium*, August 2012.

[15] J. Reardon, C. Marforio, S. Capkun, and D. Basin. User-level Secure Deletion on Log-structured File Systems. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 63–64, May 2012.

[16] G. G. Richard III and V. Roussev. Scalpel: A Frugal, High Performance File Carver. In *Proc. DFRWS Digital Forensics Research Conference*, August 2005.

[17] M. Spreitzenbarth, S. Schmitt, and C. Zimmermann. Reverse engineering of the android file system (yaffs2). Technical Report CS-2011-06, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Department Informatik, June 2011.

[18] J. Tuttle, R. J. Walls, E. Learned-Miller, and B. N. Levine. Reverse Engineering for Mobile Systems Forensics with Ares. In *Proc. ACM Workshop on Insider Threats*, October 2010.

[19] T. Vidas, C. Zhang, and N. Christin. Toward a general collection methodology for android devices. In *Proc. DFRWS Digital Forensics Research Conference*, pages 14–24, August 2011.

[20] D. Votipka, T. Vidas, and N. Christin. Passe-Partout: A General Collection Methodology for Android Devices. *IEEE Transactions on Information Forensics and Security*, 8(12):1937–1946, Dec 2013.

[21] R. J. Walls, E. Learned-Miller, and B. N. Levine. Forensic Triage for Mobile Phones with DEC0DE. In *Proc. USENIX Security Symposium*, August 2011.

[22] M. Wei, L. M. Grupp, F. M. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *Proc. USENIX Conference on File and Storage Technologies*, February 2011.