

Forensic Triage for Mobile Phones with DEC0DE

Robert J. Walls Erik Learned-Miller Brian Neil Levine
Dept. of Computer Science, Univ. of Massachusetts, Amherst
{rjwalls, elm, brian}@cs.umass.edu

Abstract

We present DEC0DE, a system for recovering information from phones with unknown storage formats, a critical problem for forensic triage. Because phones have myriad custom hardware and software, we examine only the stored data. Via flexible descriptions of typical data structures, and using a classic dynamic programming algorithm, we are able to identify call logs and address book entries in phones across varied models and manufacturers. We designed DEC0DE by examining the formats of one set of phone models, and we evaluate its performance on other models. Overall, we are able to obtain high performance for these unexamined models: an average recall of 97% and precision of 80% for call logs; and average recall of 93% and precision of 52% for address books. Moreover, at the expense of recall dropping to 14%, we can increase precision of address book recovery to 94% by culling results that don't match between call logs and address book entries on the same phone.

1 Introduction

When criminal investigators search a location and seize computers and other artifacts, a race begins to locate off-site evidence. Not long after a search warrant is executed, accomplices will erase evidence; logs at cellular providers, ISPs, and web servers will be rotated out of existence; and leads will be lost. Moreover, investigators make the most progress during on-scene interviews of suspects if they are able to ask about on-scene evidence. Mobile phones are of particular interest to investigators. Address book entries and call logs contain valuable information that can be used to construct a timeline, compile a list of accomplices, or demonstrate intent. Further, phone numbers can provide a link to a geographical location via billing records. For crimes involving drug trafficking, child exploitation, and homicide, these leads are critical [17].

The process of quickly acquiring important evidence on-scene in a limited but accurate fashion is called *foren-*

sic triage [16]. Unfortunately, digital forensics is a time-consuming task, and once computers are seized and sent off site, examination results are returned after a months-long work queue. Getting partial results on-scene ensures certain leads and evidence are recovered sooner.

Forensic triage is harder for phones than desktop computers. While the Windows/Intel platform vastly dominates desktops, the mobile phone market is based on more than ten operating systems and more than ten platform manufacturers making use of an unending introduction of custom hardware. In 2010, 1.6 billion new phones were sold [15], with billions of used phones still in use. *Smart phones*, representing only 20% of new phones [15], store information from thousands of applications each with potentially custom data formats. The more popular *feature phones*, while simpler devices, are quick to be released and replaced by new models with different storage formats. Both types of phones are problematic as phone application, OS, and file system specifications are closely guarded as commercial secrets. Companies do not typically release information required for correct parsing.

Assuming the phone is not locked by the user, the easiest method of phone triage is to simply flip through the phone's interface for interesting information. This time-consuming process can destroy the integrity of evidence, as there is no guarantee data will not be modified during the browse. Similarly, backups of the phone may be examined, but neither backups nor manual browsing will recover deleted data and data otherwise hidden by the phone's interface. Hidden data can include metadata, such as timestamps and flags, that can demonstrate a timeline and user intent, both of which can be critical for the legal process.

Forensic investigation begins with data acquisition and the parsing of raw data into *information*. The challenge of phones and embedded systems is that too often the exact data format used on the device has never been seen before. Hence, a manual process of reverse engineering begins — a dead-end for practitioners. Recent research on

automated reverse engineering is largely focused on the instrumentation of the system and executables [1,6]. While accurate and reasonable for the common Windows/Intel desktop platform, construction of a new instrumentation system for every phone architecture-OS combination in use would require significant time for each and an expertise not present in the practitioner community.

In this paper, we focus on a *data-driven approach* to phone triage. We seek to quickly parse data from the phone without analyzing or instrumenting software. We aim to obtain high quality results, even for phones that have not been previously encountered by our system. Our solution, called *DECODE*, leverages success from already examined phones in the form of a flexible library of probabilistic finite state machines. Our main insight is that the variety of phone models and data formats can be leveraged for recovering information from new phones. We make three primary contributions:

- We propose a method of *block hash filtering* for revealing the most interesting blocks within a large store on a phone. We compare small blocks of unparsed data from a target phone to a library of known hashes. Collisions represent blocks that contain content common to the other phones, and therefore not artifacts specific to the user, e.g., phone numbers or call log entries. Our methods work in seconds, reducing acquired data by 69% on average, without removing usable information.
- To recover information from the remaining data, we adapt techniques from natural language processing. We propose an efficient and flexible use of probabilistic finite state machines (PFSMs) to encode typical data structures. We use the created PFSMs along with a classic dynamic programming algorithm to find the maximum likelihood parse of the phone's memory.
- We provide an extensive empirical evaluation of our system and its ability to perform well on a large variety of previously unexamined phone models. We apply our PFSM set — unmodified — to six other phone models from Nokia, Motorola, and Samsung and show that our methods are able to recover call logs with 97% recall and 80% precision and address books with 93% recall and 52% precision for this set of unseen models.

There are a series of commercial products that parse data from phones (e.g., .XRY, cellebrite, and Paraben). However, these products rely on slow, manual reverse engineering for each phone model. Moreover, none of these products will attempt to parse data for previously unseen phone models. Even the collection of all such products does not cover all phone models currently on the market, and certainly not the set of all models still in use.

In contrast, we design and evaluate a general approach for automatically recovering information on previously unseen devices, one that leverages information from past success.

2 Methodology and Assumptions

Our goal is to enable triage-based data recovery for mobile phones during criminal investigations. Below, we provide a definition of triage, our problem, and our assumptions. Unlike much related work, our focus is not on incident response, malware analysis, privilege escalation, protocol analysis, or other topics related to security primitives. We aim to have an impact on any crime where a phone may be carried by the perpetrator before the crime, held during the crime, used as part of the crime or to record the crime (e.g., a trophy photo), or used after the crime.

The triage process. The process of quickly acquiring important evidence on-scene in a limited but accurate fashion is called *forensic triage* [16]. Our goals are focused on the law enforcement triage process, which begins with a search warrant issued upon probable cause, or one of the many lawful exceptions [12] to the Fourth Amendment (e.g., incidence to arrest). Law enforcement has several objectives when executing a search and performing triage. The first is locating all devices related to the crime so that no evidence is missed. The second is identifying devices that are not relevant to the crime so that they can be ignored, as every crime lab has a months-long backlog for completing forensic analysis. That delay is only exacerbated by adding unneeded work. The third is interviewing suspects at the crime scene. These interviews are most effective when evidence found on-scene is presented to the interviewed person. Similarly, quickly determining leads for further investigation is critical so that evidence or persons do not disappear. Central to all of these objectives is the ability to rapidly examine and extract vital information from a variety of devices, including mobile phones.

Phone triage is not a replacement for gathering information directly from carriers; however, it can take several weeks to obtain information from a carrier. Moreover, carriers store only limited information about each phone. While most keep call logs for a year, other information is ephemeral. Text message content is kept for only about a week by Verizon and Sprint, and the IP address of a phone is kept for just a few days by AT&T [3]. In contrast, the same information is often kept by the phone indefinitely and, if deleted, it is still possibly recoverable using a forensic examination.

The less time it takes to complete a triage of each device, the more impact our techniques will have. While some crime scenes involve only a few devices, increas-

ingly crime scenes involve tens and potentially hundreds of devices. For example, an office can be the center of operations for a gang, organized crime unit, or para-military cell. Typically little time is available and, in the case of search warrants, restrictions are often in place on the duration of time that a location can be occupied by law enforcement. In military scenarios, operations may involve deciding which, if any, of several persons and devices in a location should be brought back using the limited space in a vehicle; forensic triage is a common method of deciding.

Problem definition. Our goal is to enable investigators to extract information quickly (e.g., in 20 minutes or less) from a phone, regardless of whether that exact phone model has been encountered before. We limit our results to information that is common to phones — address books and call logs — but is stored differently by each phone. Triage is not a replacement for a secondary, in-depth examination; but it does achieve shortened delay with a minimal reduction in *recall* and *precision*. Recall is the fraction of all records of interest that are recovered from a device; precision is the fraction of recovered records that are correctly parsed.

Data acquisition. We make the following assumptions in the context of on-site extraction of information from embedded devices. The technical process of extracting a memory dump from a phone starts off very differently compared to laptops and desktops. Data on a phone is typically stored in custom solid state memory. These chips are typically soldered onto a custom motherboard, and data extraction without burning out the chip requires knowledge of pinouts. For that reason, several other methods are in common use for extracting data. Broadly, data can be extracted representing either the *logical* or *physical* layout of memory. Often these representations are referred to as the logical or physical *image* of a device, respectively.

A logical image is typically easier to obtain and parse; however, it suffers from some serious limitations. First, it only contains information that is presented by the file system or other application interfaces. It omits deleted data, metadata about content, and the physical layout of data in memory (which we use in our parsing). Second, logical-extraction interfaces typically enforce access rules (e.g., preventing access to a locked phone) and may modify data or metadata upon access. Examples of logical extraction include using phone backup software or directly browsing through a phone using its graphical user interface. Due to the above deficiencies, our techniques operate directly on the physical image.

A physical image contains the full layout of data stored in a phone’s memory, including deleted data that has not yet been overwritten; however, parsing raw data presents

a significant challenge to investigators — one our techniques attempt to address. We discuss the parsing challenges further in Section 3.2.

Physical extraction requires an interface that is below the phone’s OS or applications. There are a few different ways of acquiring a physical image. For example, some phones are compatible with *flasher* boxes [11], while others allow for extraction via a JTAG interface, or physical removal of the chip. Physical extraction typically takes between a few minutes and an hour depending on the extraction method, size of storage, and bus bandwidth. When we evaluate our techniques, we assume the prior ability to acquire the physical image of the phone.

Numerous companies sell commercial products that acquire data from phones, both logically and physically. This acquisition process is easier than the recovery of information from raw data, though still a challenge and not one we address. Of course, we do not expect our methods to be used on phones for which the format of data is already known. But no company offers a product that addresses even a large portion of the phone market and no combination of products covers all possible phones, even among the market of phones still being sold. Used phones in place in the US and around the world number at least an order of magnitude larger than phones still being manufactured.

Limitations of our threat model. We assume the owner of the phone has left data in a plaintext, custom format that is typical of how computers store information. We allow for encryption and even simple obfuscation, but we do not propose techniques that would defeat either. While this threat model is weak, it is representative of phone users involved in traditional crimes. Some smart-phones encrypt data, most do not; and almost all feature phones do not, and they represent 80% of the market [15]. Further, it is not possible for one attacker to encrypt the data of every other phone in existence, and our techniques work on all phones for which plaintext can be recovered. In other words, while we allow for any one person to encrypt their data, it does not significantly limit the impact of our results.

3 Design of DECODE

In this section, we provide a high-level overview of DECODE including its input, primary components, and output.

DECODE takes the physical image of a mobile phone as input. We can think of the physical image as a stream of bytes with an unknown structure and no explicit delimiters. DECODE filters and analyzes this byte stream to extract important information, presenting the output to the investigator. The internal process it uses is composed

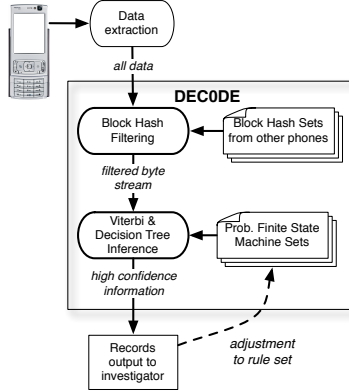


Figure 1: An illustration of the DECODE’s process. Data acquired from a phone is passed first through a filtering mechanism based on hash sets of other phones. The remaining data is input to a multistep inference component, largely based on a set of PFSMs. The output is a set of records representing information found on the phone. The PFSMs can optionally be updated to improve the process.

of two components, illustrated in Fig. 1: (i) block hash filtering and (ii) inference.

DECODE uses the block hash filter to exclude subsequences of bytes that do not contain information of interest to investigators. The primary purpose of this filtering is to reduce the amount of data that needs to be examined and therefore increase the speed of the system.

DECODE parses the filtered byte stream to extract information first in the form of *fields* and then as *records*. Fields are the basic unit of information and they include data types such as phone numbers and timestamps. Records are groups of semantically related fields that contain evidence of interest to investigators, e.g., address book entries. The inference component is designed to be both extensible and flexible, allowing an investigator to iteratively refine rules and improve results when time allows.

3.1 Block Hash Filtering

DECODE’s block hash filtering component (BHF) is based on the notion that long identical byte sequences found on different phones are unnecessary for triage. That is, such sequences are unlikely to contain useful information for investigators. Mobile phones use a portion of their physical memory to store operating system software and other data that have limited utility for triage. BHF is designed to remove this cruft and reduce the number of bytes that needs to be analyzed, thereby increasing the speed of the system.

Description. DECODE’s block hash filter logically divides the input byte stream into small subsequences of

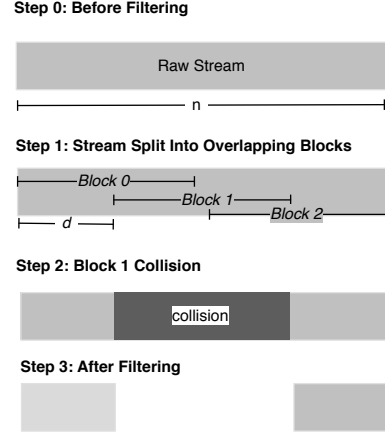


Figure 2: Block hash filtering takes a stream of n bytes and creates a series of overlapping blocks of length b . The start of each block differs by $d \leq b$ bytes. Any collision of the hash of a block with a block on another phone (or the same phone) is filtered out.

bytes. We refer to each of these subsequences as a *block*. DECODE filters out a block if its hash value matches a value in a library of hashes computed from other phones. Blocks may repeat within the same phone, but only the first occurrence of each block remains after filtering. DECODE uses block hashes, rather than a direct byte comparison, to improve system performance; However, BHF may lead to erroneous filtering due to block collisions. One type of collision arises when blocks with different byte sequences share the same hash value. Another type of collision occurs when blocks share the same subsequence even though they actually contain user information. Currently, DECODE mitigates the risk of collisions by using a cryptographic hash function and a sufficiently large block size.

To make the filter more resilient to small perturbations in byte/block alignment, DECODE uses a sliding window technique with overlap between the bytes of consecutive blocks [22]. In other words, the last bytes of a block are the same as the first bytes of the next block.

More formally, DECODE logically divides an input stream of n bytes, into blocks of b bytes with a shift of $d \leq b$ bytes between the start of successive blocks. The SHA-1 hash value for each block is computed and compared to the hash library. DECODE filters out all matched blocks. Fig 2 illustrates a simple example.

As we show empirically in Section 5, nearly all of the benefit of block hash filtering can be realized by just using another phone of the same make and model. This result ensures BHF is scalable as the test phone need not be compared to all phones in an investigator’s library.

The general idea of our block hash filter is similar to work by a variety of researchers in a number of do-

0042006F0062000B0B01000300000B1972642866600008130207D603070F1A17
 └──────────┬──────────┬──────────┘
 Unicode 11-digit phone number Timestamp

Figure 3: A simplified example of raw data as stored by a Nokia model phone, labeled with the correct interpretation. DECODE outputs a call log: the Unicode string “Bob”; the phone number (0xB digits long and null terminated) 1-972-642-8666; and the timestamp 3/7/2006 3:26:23 PM.

mains [9,13,22]. Our primary contribution is the empirical analysis of the technique in the phone domain. Further discussion of related work is given in Section 6.

3.2 Inference

After block hash filtering has been performed, what remains is a reduced ad hoc data source about which we have only minimal information. Our goal is to identify certain types of structured information, such as phone numbers, names, and other data types embedded in streams of this data.

Parsing phones is particularly challenging due to the inherent ambiguity of the input byte stream. Along with the lack of explicit delimiters, there is significant overlap between the encodings for different data structures. For example, certain sequences of bytes could be interpreted as both a valid phone number and a valid timestamp. For these reasons, simple techniques like the unix command `strings` and regular expressions will be mostly ineffective.

DECODE solves this ambiguity by using standard probabilistic parsing tools and a probabilistic model of encodings that might be seen in the data. DECODE obtains the maximum likelihood parse of the input stream creating a hierarchical description of information on the phone in the form of fields and records. More concretely, the output of DECODE is a set of call log and address book *records*. Each record is comprised of *fields* representing phone numbers, timestamps, strings, and other structures extracted from the raw stream.

3.2.1 Fields and Records

Within the block filtered data source, we have no information about where records or fields begin or end, and we have no explicit delimiters. Fig 3 shows simplified example data that could encode an address book entry in a Nokia phone; DECODE would receive this snippet embedded and undelineated in megabytes of other data. Unlike large objects, such as jpegs or Word docs, such small artifacts are difficult to isolate and can easily appear randomly.

To infer information found on phones, DECODE uses standard methods for *probabilistic finite state machines* (PFSMs), which we describe here. As implied above,

we have a lower level of *field* state machines that encode raw bytes as phone numbers, timestamps, and other types. We also have a higher level of *record* state machines that encode fields as call log entries and address book entries. For example, a call log record can be flexibly encoded as a phone number field and timestamp field very near to one another; the encoding might also include an optional text field.

Each field’s PFSM consists of one or more *states*, including a set of *start* states and a set of *end* states. Each state has a given probability of transitioning to another state in the machine. Each state *emits* a single byte during each state transition of the PFSM. The emitted byte is governed by a probability distribution over the bytes from 0x00 to 0xFF. Restricting the set of bytes that can be output by a state is achieved by setting the probability of those outputs to zero. For example, an *ASCII alphabetic* state would only assign non-zero probabilities to the ASCII codes for “a” through “z” and “A” through “Z”. Every PFSM in DECODE’s set is targeted towards a specific data type. If correctly defined, a field’s PFSM will only accept a sequence of bytes if that sequence is a valid encoding of the field type. We constructed the field PFSMs based on past observations (see Section 4.1).

Examples of DECODE’s specific field types include 10-digit phone numbers, 7-digit phone numbers, Unicode strings, and ASCII strings. Each specific field is associated with a *generic field* type such as *text* or *phone number*. Some fields have fixed lengths and others have arbitrary lengths.

We define *records* in a similar manner. Records are represented as PFSMs, except that each state emits a generic field rather than a raw byte.

Given the set of PFSMs representing each field type that we have encoded, we then *aggregate* them all into a single *Field PFSM*. We separately aggregate all record PFSMs into a single *Record PFSM*. The aggregation naively creates transitions from every field’s end state to every other field’s start states with some probability, and we do the same for compiling records. (We discuss setting these probabilities below.) In the end, we have two distinct PFSMs that are used as input to our system, along with data from a phone.

3.2.2 Finding the maximum likelihood sequence of states

Our basic challenge is that, for a given phone byte stream that is passed to the inference component of DECODE, there will be many possible way to parse the data. That is, there are many ways the PSFMs could have created the observed data, but some of these are more likely than others given the state transitions and the output probabilities. To formalize the problem, let $B = b_0, b_1, \dots, b_n$ be the stream

of n bytes from the data source. Let $S = s_0, s_1, \dots, s_n$ be a sequence of states which could have generated the output bytes. Our goal then, is to find

$$\arg \max_{s_0, s_1, \dots, s_n} P(s_0, s_1, \dots, s_n | b_0, b_1, \dots, b_n), \quad (1)$$

i.e., the maximum probability sequence of states given the observed bytes. These states are chosen from the set encoded in the PFSM given to DECODE. The probabilities assigned to PFSM's states, transitions, and emissions affect the specific value that satisfies the above equation.

In a typical *hidden Markov model*, one assumes that an output byte is a function only of the current unknown state, and that given this state, the current output is independent of all other states and outputs. Using this assumption, and noting that multiplying the above expression by $P(b_0, \dots, b_n)$ does not change the state sequence which maximizes the expression, we can write

$$\begin{aligned} & \arg \max_{s_0, \dots, s_n} P(s_0, \dots, s_n | b_0, \dots, b_n) \\ = & \arg \max_{s_0, \dots, s_n} P(s_0, \dots, s_n | b_0, \dots, b_n) P(b_0, \dots, b_n) \\ = & \arg \max_{s_0, \dots, s_n} P(s_0, \dots, s_n, b_0, \dots, b_n) \\ = & \arg \max_{s_0, \dots, s_n} P(s_0, \dots, s_n) P(b_0, \dots, b_n | s_0, \dots, s_n) \\ = & \arg \max_{s_0, \dots, s_n} P(s_0, \dots, s_n) \prod_{i=0}^n P(b_i | s_i). \quad (2) \end{aligned}$$

Naively enumerating all possible state sequences and selecting the best parse is at best inefficient and at worst intractable. One way around this is to assume that the current state depends only on the state that came immediately before it, and is independent of other states further in the past. This is known as a first order Markov model, and allows us to write

$$\begin{aligned} & \arg \max_{s_0, \dots, s_n} P(s_0, \dots, s_n) \prod_{i=0}^n P(b_i | s_i) \\ = & \arg \max_{s_0, \dots, s_n} P(s_0) P(s_1 | s_0) P(s_2 | s_0, s_1) \\ & \dots P(s_n | s_0, s_1, \dots, s_{n-1}) \prod_{i=0}^n P(b_i | s_i) \\ = & \arg \max_{s_0, \dots, s_n} P(s_0) \prod_{i=1}^n P(s_i | s_{i-1}) \prod_{i=0}^n P(b_i | s_i). \quad (3) \end{aligned}$$

The Viterbi algorithm is an efficient algorithm for finding the state sequence that maximizes the above expression. The complexity of the Viterbi algorithm is $\mathcal{O}(nk^2)$ where n and k are the number of bytes and states. For a full explanation of the algorithm, see for example the texts by Viterbi [24] or Russell and Norvig [21].

3.2.3 Fixed length fields and records

Markov models are well-suited to data streams with arbitrary length fields. For example, an arbitrary length text string can be modeled well by a single state that might transition to itself with probability α , or with probability $1 - \alpha$ to some other state, and hence terminating the string. Unfortunately, first order Markov models are not well-suited to modeling fields with fixed lengths (like 7-digit phone numbers), since it is impossible to enforce the transition to a new state after 7 bytes when one is only conditioning state transition on a single past state. In other words, a first order Markov model cannot “remember” how long it has been in a particular state.

Since it is critical for us to model certain fixed length fields like dates and phone numbers, we had two options:

- Add a *separate new state* for every position in a fixed length field. For example, a 7-digit phone number would have seven different separate states, rather than a single state.
- Implement an m th order Markov model, where m is equal to the length of the longest fixed length field we wish to model.

The first option, under a naive implementation, leads to a very large number of states, and since Viterbi is $\mathcal{O}(nk^2)$, it leads to impractical run times.

The second option, using an m th order Markov model, keeps the number of states low, but can also lead to very large run times of $\mathcal{O}(nk^{m+1})$. However, by taking advantage of the fact that *most* state transitions in our model only depend upon a single previous state, and other structure in our problem, we are able to implement Viterbi, even for our fixed length fields, in time that is close to the implementation for a first order Markov model with a small number of states. Similar techniques have been used in the language modeling literature to develop efficient higher-order Markov models [14].

3.2.4 Hierarchical Viterbi

DECODE uses Viterbi twice. First, it passes the filtered byte stream to Viterbi with the Field PFSM as input. The output of the first pass is the most likely sequence of generic fields associated with the byte stream. That field sequence is then input to Viterbi along with the Record PFSM for a second pass. We refer to these two phases as *field-level* and *record-level* inference, respectively.

The hierarchical composition of records from fields (which are in turn composed of bytes) can be captured by a variety of statistical models, including context free grammars. The main reason we chose to run Viterbi in this hierarchical fashion, rather than integrating the information about a phone type in something like a context free grammar, was to limit the explosion of states. In particu-

lar, because we have a variety of fixed-length field types, such as phone numbers, the number of states required to implement a seamless hierarchical model would grow impractically large. Our resulting inference algorithms would not have practical run times.

The decomposition of our inference into a field-level stage and a record-level stage makes the computations practical at a minimal loss in modeling power. The reason that DECODE can operate on phones that are unseen is that record state machines are very general. For example, we don't require timestamps to phone numbers to appear in any specific order for a call log entry. We require only that they are both present.

3.2.5 Post-processing

The last stage of our inference process takes the set of records recovered by Viterbi and passes them through a decision tree classifier to remove potential false positives. We refer to this step as *post-processing*. We use a decision tree classifier because it able to take into account features that can be inefficient to encode in Viterbi. For example, our classifier considers whether a record was found in isolation in the byte stream, or in close proximity to other records. In the former case, the record is more likely to be a false positive. Our evaluation results (Section 5) show that this process results in significant improvements to precision with a negligible effect on recall.

We use the Weka J48 Decision Tree, an open source implementation of a well-known classifier (<http://www.cs.waikato.ac.nz/ml/weka>). In general, a decision tree can be used to decide whether or not an input is an example of the target class for which it is trained. The classifier is trained using a set of feature tuples representing both positive and negative examples. In our case, the decision tree decides whether a given record, output from our Viterbi stage, is valid or not. We selected a set of features common to both call log and address book records: number of days from the average date; frequency of phone numbers with same area code; number of different forms seen for the same number (e.g., 7-digit and 10-digit); number of characters in string; number of times the record appears in memory; distance to closest neighbor record. We do not claim that our choice of features and classifier is optimal; it merely represents a lower bound for what is possible.

Post-processing does not inhibit the investigator, it is a filter intended to make the investigator's work easier. To this end, DECODE can make both the pre- and post-processing results available ensuring that the investigator has as much useful information as possible.

For our evaluation, the positive training examples consisted of true records from a small set of phones called our *development set* (described in detail in Section 5).

Generic type	Specific type	Num. States
Records		
Call logs	Nokia call log composed of text, phone num., timestamps	8
	General call log composed of text, phone num., timestamps	9
Address books	General address book composed of phone numbers, text	5
Fields		
Phone number	ASCII	11
	Unicode	22
	Nokia 10 digit	6
Timestamp	UNIX	4
	Samsung	4
	Nokia	7
Text	ASCII bigram	6
	Unicode	7
Number index	Nokia number index	1
unstructured	unstructured	1

Table 1: Examples of types that we have defined in DECODE.

To create the negative training examples, we used a 10 megabyte stream of random data with byte values selected uniformly at random from 0x00 to 0xFF. We input the random data to DECODE's Viterbi implementation and used the resulting output records as negative examples. We found that this provided better results than using negative examples found on real phones.

4 Implementation of State Machines

In the previous section, we presented DECODE's design broadly; in this section, we focus on the core of the inference process: the probabilistic finite state machines (PFSM).

DECODE's PFSMs support a number of generic field types such as phone number, call log type, timestamp, and text as well as the target record types: address book and call log. Table 1 shows some example field types that we have defined and the number of states for each. In all, DECODE uses approximately 40 field-level and 10 record-level PFSMs.

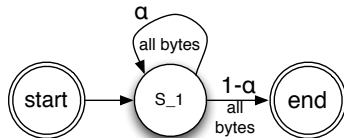
Most fields emit fixed-length byte sequences. For example, the 10-digit phone number field is defined as 10 states in which state k (for $k \neq 1$) can only be reached by state $k - 1$. The state machine for a 10-digit phone number as found on many Nokia phones is:



As mentioned in the previous section, each state emits a single byte; since Nokia often stores digits as nibbles, each state in the machine encodes two digits. The emission probability is governed by both the semantics of the Nokia encoding and real-world constraints. For example

a 10-digit phone number (in the USA) cannot start with a 0 or a 1 and therefore the first state in the machine cannot emit bytes 0x00-0x1F, i.e., the emission probability for each of these bytes is zero.

Some fields, such as an *unstructured byte stream* have arbitrary length. Such a field is simply defined by a single state with probability α of transitioning to itself, and probability $1 - \alpha$ of terminating. In fact, this specific field is special: DECODE uses the unstructured field as a “catch-all” for unknown or unstructured portions of the byte stream. Byte sequences that do not match a more meaningful field type will always match the *unstructured* field, which is:



We emphasize that our goal is *not* to produce a full specification of the format of a device. While we would certainly be delighted if this were an easy problem to solve, we note that we can extract significant amounts of useful information from a data source even when large parts of the format specification are not understood. Hence, rather than solving the problem of complete format specification, we seek to extract as many records as possible according to our specification of records. It is also important to note that our field and record definitions may ignore large amounts of structure in a phone format. Only a minimal amount of information about a phone’s data organization is needed to define useful fields and records. We return this point in Section 5.3.

4.1 Coding State Machines

We created most of the PFSMs used in DECODE using a hex editor and manual reverse engineering on a small subset of phones that we denote as our *development set*. We limited the development set to one phone model each from four manufacturers with multiple instances of each model: the Nokia 3200B, Motorola v551, LG G4015 and Samsung SGH-T309. We intentionally did not examine any other phone models from these manufacturers prior to the evaluation of DECODE (Section 5) so that we could evaluate the effectiveness of our state machines on previously unobserved phone models.

We also used DECODE itself to help refine and create new state machines, both field and record level, for the development phones. This process was very similar to how we imagine an investigator would use DECODE during the post-triage examination.

Once we reached high recall for the development set, we fixed the PFSMs and other components using

DECODE without modification for the extent of our evaluation regardless of what model was parsed.

Selecting Transition Probabilities. A sequence of bytes may match multiple different field types. Similarly, a sequence of fields may match multiple record types. Viterbi accounts for this by choosing the most likely type. It may appear that a large disadvantage of this approach is that we must manually set the type probabilities for both fields and records. However, Viterbi is robust to the choice of probabilities: the numerical values of the field probabilities are not as important as the probability of one field relative to another.

5 Evaluation

We evaluated DECODE by focusing on several key questions.

1. How much data does the block hash filtering technique remove from processing?
2. How effectively does our Viterbi-based inference process extract fields and records from the filtered data?
3. How much does our post-processing stage improve the Viterbi-based results?
4. How well does the inference process work on phones that were unobserved when the state machines were developed?

Experimental Setup. We made use of a number of phones from a variety of manufacturers. The phones contained some GUI-accessible address book and call log entries, and we entered additional entries using each phone’s UI. A combination of deleted and GUI-accessible data was used in our tests; however, most phones contained only data that was deleted and therefore unavailable from the phone’s interface but recoverable using DECODE. The phones we obtained were limited to those that we could acquire the physical image from memory (i.e., all data stored on the phone in its native form). The list of phones is given in Table 2. Our evaluation focuses on feature phones, i.e., phones with less capability than smart phones.

As stated in Section 4.1, we performed all development of DECODE and its PFSMs using only the Nokia 3200B, Motorola v551, LG G4015, and Samsung SGH-T309 phones. We kept the *evaluation set* of phones separate until ready to evaluate performance. We acquired the physical image for all phones using Micro Systemation’s commercial tool, .XRY.

We focus on two types of records: address book entries and call log entries. We chose these record types

Make	Model	Count	MB
PFSM Development Set			
Nokia	3200b	4	1.4
Motorola	V551	2	32.0
Samsung	SGH-T309	2	32.0
LG	G4015	2	48.0
Evaluation Set			
Motorola	V400	2	32.0
Motorola	V300	2	32.0
Motorola	V600	2	32.0
Motorola	V555	2	32.0
Nokia	6170	2	4.9
Samsung	SGH-X427M	2	16.0

Table 2: The phone models used in this study. The table shows the number we had of each and the size of local storage.

because of their ubiquity across different phone models and their relative importance to investigators during triage. We evaluate the performance of DECODE’s inference engine based on two metrics, *recall* and *precision*. Recall is the fraction of all phone records that DECODE correctly identified: the number of true positives over the sum of false negatives and true positives. If recall is high, then all useful information on a phone has been found. Precision is the fraction of extracted records that are correctly parsed: the number of true positives over the sum of false positives and true positives. If precision is high then the information output by DECODE is generally correct.

Often these two metrics represent a trade-off, but our goal is to keep both high. In law enforcement, the relative importance of the two metrics depends on the context. For generating leads, recall is more important. For satisfying the *probable cause* standard required by a search warrant application, moderate precision is needed. Probable cause has been defined as “fair probability”¹ that the search warrant is justified, and courts do not use a set quantitative value. For evidence meeting the *beyond a reasonable doubt* standard needed for a criminal conviction, very high precision is required, though again no quantitative value can be cited.

For each of our tested phones, we used .XRY not only to acquire the physical image, but also to obtain *ground truth results* that we used to compare against DECODE’s results. It was often the case that DECODE obtained results that .XRY did not. And in those cases, we manually inspected the result and decided whether they were true or false positives (painstakingly using a hex editor). We made conservative decisions in this regard, but were able to employ a wealth of common sense rules. For example, if a call entry seemed to be valid and recent, but was several years from all other entries, we labeled it as a false positive. Similarly, an address book entry for “A.M.”

¹United States v. Sokolow, 490 U.S. 1 (1989)

is most reasonably assumed to be a true positive while “!Mb” is most reasonably a false positive; even though both have two letters and two symbols, the latter does not follow English conventions for punctuation. It would be impractical to program all such common sense rules and our manual checking is stronger in that regard. Occasionally, DECODE extracts partially correct or noisy records. We mark each of these records as wrong, unless the only error is a missing area code on the phone number.

5.1 Block Hash Filtering Performance

The goal of BHF is to reduce the amount of data that DECODE must parse, reducing run time, without sacrificing recall. On average, we find that BHF is able to filter out about 69% of the phone’s stored data without any measurable effect on inference recall. The BHF algorithm has only two parameters: the shift size d and the block size b . Our results show that the shift size does not greatly affect the algorithm’s performance, but it has a profound effect on storage requirements. Also, we found that performance varies with block size, but not as widely as expected.

For each value of b and d that we tested, we kept the corresponding BHF sets in an SQL table. The database was able to match sets in tens of seconds, so we do not report run time performance results here. As an example, on a moderately resourceful desktop, DECODE is able to filter a 64 megabyte phone, with $b = 1024$ and $d = 128$, in under a minute.

Ideally, we (and investigators) would want our hash library to be comprised entirely of new phones. If our library contains used phones, there is a negligible chance that the same common user data (e.g., an address book entry with the same name and number) will appear on different phones, align perfectly on block boundaries, and be erroneously filtered out. Regardless, it was impractical for us to find an untouched, new phone model for every phone we tested. If data was filtered out in this fashion because of our use of pre-owned phones, it would likely have shown up in the recall values in the next section; since the recall values are near perfect, we can infer this problem did not occur.

Filtering Performance. First, we examined the effect of the block size b on filtering. Fig. 4 shows the overall filter percentage of our approach for varying block sizes. In these experiments, we set $d = b$ so that there was never overlap. The line plots the average for all phones. As expected, the smaller block sizes make more effective filters. However, a small block size results in more blocks and consequently, greater storage requirements. On average in our tests, 73% of data is filtered out when $b = 256$, while only slightly less, 69%, is filtered out when $b = 1024$.

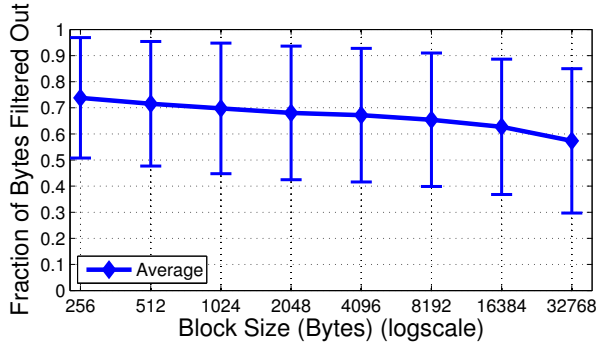


Figure 4: The average performance of BHF as block size varies for all phones listed in Table 2 (logarithmic x-axis). Error bars represent one standard deviation. In all cases we set $d = b$ (i.e., shift size is equal to block size), but performance does not vary with d in general.

Second, we examined the affect of the shift amount d on filtering. In our tests, we fixed $b = 1024$ and varied $d = \{32, 64, 128, 256, 512, 1024\}$. However, there is less than a 1% difference in filtering between $d = 32$ and $d = 1024$ for all phones. (No plot is shown.) Again, the affect of d is on storage requirements, which we discuss below.

Third, we isolated what type of data is filtered out for each phone using fixed block and shift sizes of $b = 1024$ and $d = 128$; we use these values for all other experiments in this paper. Fig. 5 shows the results as stacked bars; the top graph shows filtering as a percentage of the data acquired from the phone, and the bottom graph shows the same results in megabytes. For each of the 25 phones, the bottom (blue) bar shows the percentage of data filtered out because the block was a repeated, constant value (such as a run of zeros). The middle (black) bar shows the percentage of data that was in common with a different instance of the same make and model phone. The top red bar shows the percentage of data that can be filtered out because it is only found on some phone in the library that is a different make or model. The data that remains after filtering is shown in the top, white box.

On average, 69% of data is removed by block hash filtering. Generally, the technique works well. On average, half of the filtered out data was found on another phone of the same model. These percentage values are in terms of the complete memory, including blocks that were filled with constants (effectively empty). Therefore, as a percentage of non-empty data, the percentage of filtered out data is higher. These results suggest that it is often sufficient to only compare BHF sets of the same model phone. However, in some models less than 3% of data was found on another instance of the same model. This poor result was the case for the Samsung SGH-X427M and Motorola V300. Finally, the results shown in the

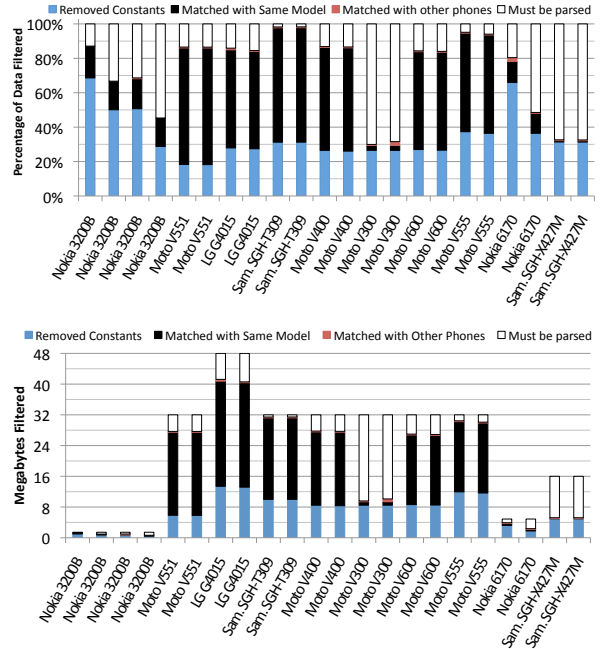


Figure 5: The amount of data remaining after filtering is shown as solid white bars, as a percentage (top) and in MB (bottom). On average, 69% of data is successfully filtered out. Black bars show data filtered out because they match data on another instance of the same model. Blue bars show data filtered out because it is a single value repeated (e.g., all zeros). Red bars show data filtered out because it appears on a different model. ($b = 1024$ bytes, $d = 128$ bytes)

Fig. 5 (bottom), suggest that the performance of BHF was not correlated with the total storage space of the phone.

Our results in the next section on inference, in which DECODE examines only data remaining after filtering, demonstrate that filtering does not significantly remove important information: recall is 93% or higher in all cases.

Storage. An important advantage of our approach is that investigators can share the hash sets of phones, without sharing the data found within each phone. This sharing is very efficient as the hash sets are small compared to the phones. The number of blocks from each phone that must be hashed and stored in a library is $O((n - b)/d)$, though only unique copies of each block need be stored. Given that $n \gg b$, the number of blocks is dependent on n and d and the affect of b on storage is insignificant. However, since it is required that $d \leq b$, the algorithm’s storage requirements does depend on b ’s value in that sense. As an example, for a 64 megabyte phone, when $b = 1024$ bytes and $d = 128$ bytes, the resulting BHF set is 524,281 hash values. At 20-bytes each, the set is 10 megabytes (15% of the phone’s storage). Since we need perhaps only one or two examples of any phone model, the cumulative space needed to store BHF sets for an enormous number

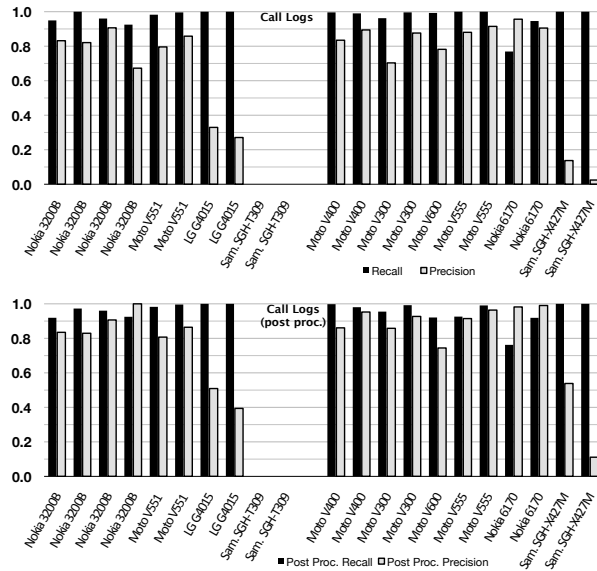


Figure 6: Precision and recall for call logs. (Top) Results after only Viterbi parsing. (Bottom) Results after post-processing. Left bars are development set; right bars are evaluation set. In all graphs, black is recall and gray is precision. On average, development phones have recall of 98%, and precision of 69% that increases to 77% after post processing. On average, evaluation phones have recall of 97%, and precision of 72% that increases to 80% after post processing. The T309 had no call log entries, which explains in part DECODE’s poor performance for the X427M.

of phone models is practical. Since BHF gains nearly all benefit from comparing phones of the same model, comparison will always be fast.

In order to be effective, the library needs to be constructed using the same hash function and block size for all phones; however, the shift amount need not be the same. This is important because the storage requirement of the library is inversely proportional to the shift size and thus is minimized when $d = b$. Conversely, BHF removes the most data when $d = 1$. We can effectively achieve maximal filtering with minimal storage using $d = b$ for the library and $d = 1$ for the test phone. The cost of this approach is more computation and consequently higher run times. A full analysis is beyond the scope of this paper.

5.2 Inference Performance

To evaluate our inference process, we used DECODE to recover call log and address book entries from a variety of phones. In our results, we distinguish between the performance of the Viterbi and decision tree portions of inference. Additionally, we make clear the performance of DECODE on phones in our development set versus

phones in our evaluation set. All results in this section assume that input is first processed using BHF.

Fig. 6 shows the performance of our inference process for call logs; the top results are before the post-processing step and the bottom after post-processing. The white-space break in the chart separates the development set of phones (on the left), and the evaluation set (on the right). We put the most effort toward encoding high quality PFSMs for the Nokia and Motorola phones. Not surprisingly, the results are best in general for these makes, indicating that the performance of DECODE is dependent on the quality of the PFSMs. However, the results also show that DECODE can perform well even for the previously unseen phones in the evaluation set. Overall, recall of DECODE is near complete at 98% for development phones and 99% for evaluation phones. Precision is more challenging, and after Viterbi is at 69% for development phones and 72% for evaluation phones. It is important to note that no extra work on DECODE was performed to obtain results from the phones in the evaluation set, which is significant compared to methods that instrument executables or perform other machine and platform dependent analysis. After post-processing, the precision for the development and evaluation phones increased to 77% and 80% respectively.

Fig. 7 shows the performance of our inference process for address book records. As before, the top results are after filtering but not post-processed while the bottom are post-processed. Overall, recall of the DECODE is again high at 99% for development phones and 93% for evaluation phones. Precision after Viterbi is 56% for development phones and 36% for evaluation phones. After post processing by the decision tree, the precision for all phones increased, by an average of 61% over the Viterbi-only results, a significant improvement. For development phones, precision increases to 65% on average. (Note that the development phones are used to train the classifier.) For evaluation phones, precision increases significantly to 52%.

While performance is not perfect, we could likely improve performance by using a different set of PFSMs for each different phone manufacturer. In our evaluation, all PFSMs for all manufactures are evaluated at once. Because our goal is to allow for phone triage, we don’t reduce the set of state machines for each manufacturer; however, a set of manufacturer-specific state machines could improve performance at the expense of being a less general solution.

We also note that when recall is high, it is easier to discover the intersection of information found on two independent phones from the same criminal context; that intersection is likely to be a better lead than most.

When necessary, we can prioritize precision over recall. Fig. 8 shows the results of culling records for where the

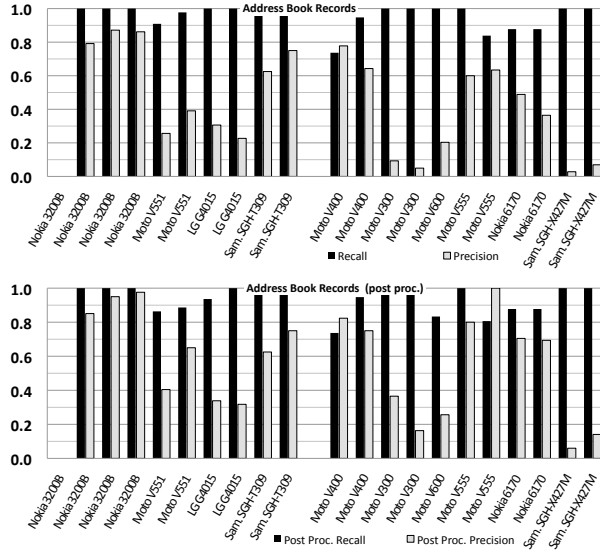


Figure 7: Precision and recall for Address Book entries. (Top) Results after only Viterbi parsing. (Bottom) Results after post-processing. On average, development phones have recall of 99%, and precision of 56% that increases to 65% after post processing. On average, evaluation phones have recall of 93%, and precision of 36% that increases to 52% after post processing. N.b, The first Nokia has no address book entries at all.

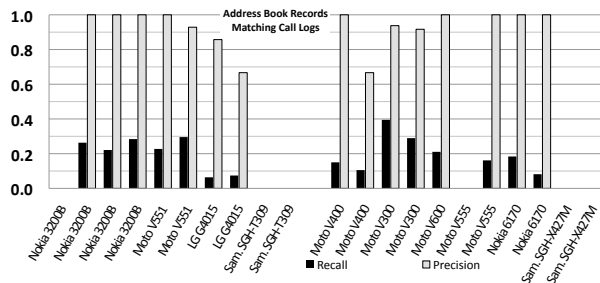


Figure 8: Precision and recall for Address Book entries after results are culled that do not match phone numbers in DECODE's call logs for the same phone. For some phones, all results are culled. On average, development phones have recall of 16%, and precision of 92% (when results are present). On average, evaluation phones have recall of 14%, and precision of 94% (when results are present).

phone number in the address book does not also appear in the call log: precision is increased to 92%, although recall drops to 14%. (We don't show the same process for call logs.) This simple step shows how easy it is to isolate results for investigators that deem precision of results more important than recall. Moreover, the results that are culled are still available for inspection.

Execution time. Inference is the lowest component of

DECODE. The post processing step takes a few seconds, but the Viterbi component takes significantly longer. On average, DECODE's Viterbi processes 12,781 bytes/sec. The smaller phones in our set (Nokias) finish in a few minutes, while the larger Motorola can completed in about 15 minutes. Since the Viterbi processing already works with distinct blocks of input produced by the BHF component, it would be straightforward to produce a parallel version of Viterbi for our scenario, thereby greatly increasing speed.

5.3 Limitations

Our evaluation is limited in a number of ways in addition to what was previously discussed. First, as with any empirical study, our results are dependent on our test cases. While our set of phones is limited, it contains phones from a variety of makes and models. In future work, we aim to test against additional phones. Second, our tests are performed only on call logs and address book entries. Presently, we are extending DECODE to examine other artifacts, including stored text messages. Since many phone artifacts are similar in nature — text messages are stored as strings, phone numbers, and dates — extending DECODE to parse additional types is easier than creating the initial PFSMs.

Our approach also has a number of limitations. First, we don't address the challenge of acquiring the physical memory image from phones, which is an input needed for DECODE. Here, we have leveraged existing tools to do so. However, acquisition is an independent endeavor and varies considerably with the platform of the phone. Part of our goal is to show that despite hardware (and software) differences, one approach is feasible across a spectrum of devices. Second, DECODE's performance is tied strongly to the quality of the PFSMs. Poorly designed state machines, especially those with few states, can match any input. We do not offer an evaluation of whether it is hard or time consuming to design high quality PFSMs or other software engineering aspects of our problem; we report only our success. Third, a single PFSM has an inherent endianness embedded in it. DECODE does not automatically reorganize state machines to account for data that is the opposite endianness. Fourth, we have not explicitly demonstrated that phones do indeed change significantly from model to model or among manufactures. This assertion is suggested by DECODE's varied performance across models but we offer no overall statistics.

It is also important to note that DECODE is an *investigative* tool and not necessarily an *evidence-gathering* tool. Tools for gathering evidence must follow a specific set of legal guidelines to ensure the admissibility of the collected evidence in court. For example, the tool or technique must have a known error rate (for example, see

Daubert v. Merrell Dow Pharmaceuticals, 509 U.S. 579 (1993)).

Finally, our approach is to gather artifacts that match a description that may be too vague in some contexts. For example, DECODE ignores important metadata that is encoded in bit flags that may indicate if an entry is deleted. Such metadata can be critical in investigations. It is our aim to have DECODE parse more metadata in the future.

6 Related Work

Our work is related to a number of works in both reverse engineering and forensics. We did not compare DECODE against these works as each has a significant limitation or assumption that does not apply well to the criminal investigation of phones.

Polyglot [2], Tupni [6], and Dispatcher [1] are instrumentation-based approaches to reverse engineering. Since binary instrumentation is a complex, time-consuming process, it is poorly suited to mobile phone triage. Moreover, our goal is different from that of Polyglot, Tupni, and Dispatcher. We seek to extract information from the data rather than reverse engineer the full specification of the device’s format.

Other previous works have attempted to parse machine data without examining executables. Discoverer [5] attempts to derive the format of network messages given samples of data. However, Discoverer is limited to identifying exactly two types of data — “text” and “binary” — and extending it to additional types is a challenge. Overall, it does not capture the rich variety of types that DECODE can distinguish.

LearnPADS [7,8,25] is another sample-based system. It is designed to automatically infer the format of ad hoc data, creating a specification of that format in a custom data description language (called PADS). Since LearnPADS relies on explicit delimiters, it is not applicable to mobile phones.

Cozzie et al. [4] use Bayesian unsupervised learning to locate data structures in memory, forming the basis of a virus checker and botnet detector. Unlike DECODE, their approach is not designed to parse the data but rather to determine if there is a match between two instances of a complex data structure in memory.

In our preliminary work [23], we used the Cocke-Younger-Kasami (CYK) algorithm [10] to parse the records of Nokia phones. While this effort influenced the development of DECODE, it was much more limited in scope and function.

The idea of extracting records from a physical memory image is similar to *file carving*. File carving is focused on identifying large chunks of data that follow a known format, e.g., jpegs or mp3s. Some file carving techniques match known file headers to file footers [18,20] when they

appear contiguously in the file system. More advanced techniques can match pieces of images fragmented in the file system relying on domain specific knowledge about the file format [19]. In contrast, our goal is to identify and parse small sequences of bytes into records — all without any knowledge of the file system. Moreover, we seek to identify information within unknown formats that only loosely resemble the formats we’ve previously seen.

DECODE’s filtering component is similar to number of previous works. Block hashes have been used by Garfinkel [9] to find content that is of interest on a large drive by statistically sampling the drive and comparing it to a bloom filter of known documents. This recent work has much in common with both the `rsync` algorithm [22], which detects differences between two data stores using block signatures, as well as the Karp-Rabin signature-based string search algorithm [13], among others.

7 Conclusions

We have addressed the problem of recovering information from phones with unknown storage formats using a combination of techniques. At the core of our system DECODE, we leverage a set of probabilistic finite state machines that encode a flexible description of typical data structures. Using a classic dynamic programming algorithm, we are able to infer call logs and address book entries. We make use of a number of techniques to make this approach efficient, processing data in about 15 minutes for a 64-megabyte image that has been acquired from a phone. First, we filter data that is unlikely to contain useful information by comparing block hash sets among phones of the same model. Second, our implementation of Viterbi and the state machines we encoded are efficiently sparse, collapsing a great deal of information in a few states and transitions. Third, we are able to improve upon Viterbi’s result with a simple decision tree.

Our evaluation was performed across a variety of phone models from a variety of manufactures. Overall, we are able to obtain high performance for previously unseen phones: an average recall of 97% and precision of 80% for call logs; and average recall of 93% and precision of 52% for address books. Moreover, at the expense of recall dropping to 14%, we can increase precision to 94% by culling results that don’t match between call logs and address book entries on the same phone.

Acknowledgments. This work was supported in part by NSF award DUE-0830876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. We are grateful for the comments and assistance of Jacqueline

Feild, Marc Liberatore, Ben Ransford, Shane Clark, Jason Beers, and Tyler Bonci.

References

- [1] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *ACM Proc. CCS*, pages 621–634, 2009.
- [2] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *ACM Proc. CCS*, pages 317–329, 2007.
- [3] Computer Crime and Intellectual Property Section, U.S. Department of Justice. Retention Periods of Major Cellular Service Providers. <http://dgsearch.no-ip.biz/rnrfiles/retention.pdf>, August 2010.
- [4] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging For Data Structures. In *Proc. USENIX OSDI Symposium*, pages 255–266, Dec 2008.
- [5] W. Cui, J. Kannan, and H. J. Wang. Discoverer: automatic protocol reverse engineering from network traces. In *USENIX Security Symp.*, pages 1–14, 2007.
- [6] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Proc. ACM CCS*, pages 391–402, 2008.
- [7] K. Fisher, D. Walker, and K. Q. Zhu. LearnPADS: automatic tool generation from ad hoc data. In *Proc. ACM SIGMOD*, pages 1299–1302, 2008.
- [8] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *Proc. ACM POPL*, pages 421–434, 2008.
- [9] S. Garfinkel, A. Nelson, D. White, and V. Roussev. Using purpose-built functions and block hashes to enable small block and sub-file forensics. In *Proc. DFRWS Annual Forensics Research Conference*, pages 13–23, Aug 2010.
- [10] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000.
- [11] K. Jonkers. The forensic use of mobile phone flasher boxes. *Digital Investigation*, 6(3–4):168–178, May 2010.
- [12] N. Judish et al. Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations. US Dept. of Justice, 2009.
- [13] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987.
- [14] R. Luk and R. Dampier. Inference of letter-phoneme correspondences by delimiting and dynamic time warping techniques. In *IEEE Intl Conf on Acoustics, Speech, and Signal Processing*, volume 2, pages 61–64, Mar. 1992.
- [15] R. C. C. Milanesi, T. H. Nguyen, C. Lu, A. Zimmermann, A. Sato, H. D. L. Vergne, and A. Gupta. Market Share Analysis: Mobile Devices, Worldwide, 4Q10 and 2010. <http://www.gartner.com/DisplayDocument?id=1542114>; see also <http://tinyurl.com/4zandx4>, Feb 2011.
- [16] R. P. Mislan, E. Casey, and G. C. Kessler. The growing need for on-scene triage of mobile devices. *Digital Investigation*, 6(3–4):112–124, 2010.
- [17] National Gang Intelligence Center. National Gang Threat Assessment 2009. Technical Report Document ID: 2009-M0335-001, US Dept. of Justice, <http://www.usdoj.gov/ndic/pubs32/32146>, Jan 2009.
- [18] A. Pal and N. Memon. The evolution of file carving. *Signal Processing Magazine, IEEE*, 26(2):59–71, March 2009.
- [19] A. Pal, H. T. Sencar, and N. Memon. Detecting file fragmentation point using sequential hypothesis testing. *Digital Investigation*, 5(S1):2–13, 2008.
- [20] G. Richard and V. Roussev. Scalpel: A frugal, high performance file carver. In *Proc. DFRWS Annual Forensics Research Conference*, August 2005.
- [21] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.
- [22] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Feb 1999.
- [23] 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.
- [24] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967.
- [25] Q. Xi, K. Fisher, D. Walker, and K. Zhu. Ad hoc data and the token ambiguity problem. In *Proc. Intl Symp Practical Aspects of Declarative Languages*, pages 91–106, 2009.