mCarve: Carving Attributed Dump Sets*

Ton van Deursen† — *University of Luxembourg*
Sjouke Mauw — *University of Luxembourg*
Saša Radomirović — *University of Luxembourg*

Abstract

Carving is a common technique in digital forensics to recover data from a memory dump of a device. In contrast to existing approaches, we investigate the carving problem for sets of memory dumps. Such a set can, for instance, be obtained by dumping the memory of a number of smart cards or by regularly dumping the memory of a single smart card during its lifetime. The problem that we define and investigate is to determine at which location in the dumps certain attributes are stored. By studying the commonalities and dissimilarities of these dumps, one can significantly reduce the collection of possible locations for such attributes. We develop algorithms that support in this process, implement them in a prototype, and apply this prototype to reverse engineer the data structure of a public transportation card.

1 Introduction

In digital forensics, the process of recovering data from a memory dump of a device is called *carving*. The main objective of current file carving approaches is to reconstruct (partially) deleted, damaged or fragmented files. A typical example is the analysis of memory dumps from cell phones [1]. Because a file can be permuted in many possible ways, the process of reassembling files is very labor intensive. Therefore, fully and semi-automatic file carving tools have been developed that aid the human inspection process.

Traditional carving approaches aim to analyze a single memory dump. In some cases, however, one may have access to a series of similarly structured dumps. This may result from observing a system that progresses in time, while making memory dumps at regular time intervals, or from dumping the memory of a collection of similar systems. An example is the analysis of the data encoded on a public transportation card. It is possible to collect dumps of several cards after each usage. This will be the running example throughout this paper.

We will investigate the problem of carving sets of dumps under two simplifying assumptions. The first assumption is that we can observe certain relevant properties of the system at the moment of dumping its memory. In this way, we can collect the values of a number of attributes that characterize part of the state of the system, and link that information to the memory dump. An example of such an attribute is the number of rides left on a public transportation card, which can be easily observed from the display of the card reader when validating the card. The carving problem for such attributed dump sets is then described as the problem of finding at which location in the memory dump the attributes are stored.

The second assumption is that the memory layout is either static or semi-dynamic. A memory layout is static if the attributes are stored at the same location in every dump and the dumps have the same length. An attribute is stored semi-dynamically if it is stored alternatingly in a number of different locations. This will allow us to develop algorithms to identify such possible locations in dumps.

Carving dump sets allows one to reverse engineer the memory layout of a system and understand or even manipulate the system's functioning. Several applications can be thought of. A first example is the analysis of the data collected in systems using smartcards, such as the transportation card mentioned above. One can e.g. verify privacy concerns by inspecting which travel information is stored on the card. Another example is the analysis of the data structures of an obfuscated piece of software (e.g. malware) or of a piece of software of which the specifications have been lost (e.g. legacy code).

The problem of carving attributed dump sets is different from the traditional file carving problem. While tra-

*This paper was first published in the Proceedings of the 20th USENIX Security Symposium.

†Ton van Deursen was supported by a grant from the Fonds National de la Recherche (Luxembourg).

ditional file carving tools can be used to obtain information about each dump in a set, the dump set's evolution and known attributes provide additional information not available in traditional file carving.

Our paper is concerned with the problem of extracting this additional information. The main contributions of this paper are: (1) to define the problem of carving dump sets (Section 3); (2) to develop and analyze a methodology for carving dump sets based on two simple operations (Sections 3 and 5); (3) to develop a prototype carving tool, called mCarve (Section 7); and (4) to apply this tool to reverse engineer the data structure of the e-go system (Section 8).

2 Related work

Closest to our work are file carving approaches that try to recover files from raw data. These approaches try to recover the data of a single dump whereas we focus on recovering data (and data structures) of a set of dumps. Garfinkel [7] describes several carving algorithms that recover files by searching for headers of known file formats. These algorithms reconstruct files based on their raw data, rather than using the metadata that points to the content. Cohen [2] formalizes file carving as a construction of a mapping function between raw data bytes and image bytes. Based on this formalization, he derives a carving algorithm and applies it to PDF and ZIP file carving. In recent work, Sencar and Memon [10] describe an approach to identify and recover JPEG files with missing fragments. Common to these file carving approaches is that they are designed for one (or a small set of) known file format(s).

More general, but perhaps less powerful are the approaches that analyze binary data by visual inspection. Conti et al. [3] describe a tool that allows analysts to visually reverse engineer binary data and files. Their tool supports simple techniques such as displaying bytes as pixels, but also more complicated techniques that visualize self-similarity in binary data. Helfman [8] first visualized self-similarity in binary data using dotplot patterns. Using dotplot patterns he revealed redundancy in various encodings of information.

Some information in a memory dump may be constructed using CRCs, cryptographic hashes, or encryption. Since the entropy of these pieces of data is higher than of structured data, they can be detected using entropy analysis. Several methods to efficiently find cryptographic keys are described in [11]. Some of these techniques are based on trial-and-error, while others identify possible keys by measuring entropy. Testing whether a given string is random has been studied extensively. See e.g. [9] for an overview and implementation of the most important algorithms.

3 Carving attributed dump sets

The concept that is central to our research is the concept of a *dump*. A dump consists of raw binary data that is captured from a system, for instance, from a computer's memory, a data carrier or a communication transcript. An example of a dump is the contents of a public transportation card's memory.

We assume that the process of creating a dump can be repeated, allowing us access to a number of dumps of the same system. We call such a collection of dumps a *dump set*. One can, e.g., consider dumps of a number of public transportation cards, both before and after their use. We assume that different dumps of the same system have the same length. If we denote the bit strings of length $n \in \mathbb{N}$ by $\mathbb{B}^n$ and bit strings of arbitrary, finite length by $\mathbb{B}^*$, then a set of dumps of length n is denoted by $S \subseteq \mathbb{B}^n$. The length n of bit string $s \in \mathbb{B}^n$ is denoted by $|s|$ and the number of elements in set S is denoted by $|S|$. In this paper, the closed interval $[i, j]$ will denote the set of integers z such that $i \leq z \leq j$ and the half-open interval $[i, j)$ will denote the set of integers z such that $i \leq z < j$. For $i \in [0, |s|)$ we denote the i-th bit of s by s_i. For $I \subseteq [0, |s|)$, we denote the subsequence of s that consists of all elements with index in I by $s|_I$. The subsequence operator extends to sets of dumps in the obvious way.

A dump contains information about the state of the system, e.g., the number of rides left on a public transportation card or the last time that it was used. We call such state properties *attributes*. For each dump set we consider a set $\mathbb{A}$ of attributes. The function $\mathrm{type}\colon \mathbb{A} \to \mathbb{D}$ assigns to every attribute a finite value domain, where $\mathbb{D}$ denotes the set of all finite value domains. The value of attribute $a \in \mathbb{A}$ expressed in dump s is denoted by $\mathrm{val}_a\colon S \to \mathrm{type}(a)$. For instance, the type of the attribute *rides-left* can be $[0, 15]$ and a particular dump s of a card can have 5 rides left, so $\mathrm{val}_{\textit{rides-left}}(s) = 5$. The type of the attribute *last-used* is the set of all dates between 1/1/2000 and 1/1/2050, extended with the time of day in hh:mm:ss format.

A dump contains the system's attribute values in a binary representation. The mapping from an attribute domain to its binary representation is called an *encoding*. We assume that for a given attribute $a \in \mathbb{A}$ the length of an encoding is fixed, so an encoding of a is a function from $\mathrm{type}(a)$ to $\mathbb{B}^n$ for some $n \in \mathbb{N}$. This function is required to be injective. For the public transportation card, a sample encoding of the *rides-left* attribute is the (5-bit) binary representation and a possible encoding of the *last-used* attribute is the number of seconds since 1/1/2000, 00:00 hrs modulo 2^{32} expressed in binary format. The set of all encodings of $D \in \mathbb{D}$ is denoted by $\mathbb{E}_D$.

We start with the assumption that an attribute is always stored at the same location in all dumps of the system. In

Section 5 we will extend this to semi-dynamic attributes. With this assumption we can identify which bits of the dump are related to a given attribute. This is captured in the notion of an *attribute mapping*. Here we denote the powerset of a set X by $\mathcal{P}(X)$.

Definition 1. *Let $S \subseteq \mathbb{B}^n$ be a dump set with dumps of length n. An* attribute mapping *for S is a function $f\colon \mathbb{A} \to \mathcal{P}([0,n))$, such that*

$$\forall a \in \mathbb{A}\, \exists e \in \mathbb{E}_{\text{type}(a)}\, \forall s \in S\colon\ s|_{f(a)} = e(\text{val}_a(s)).$$

An attribute mapping is non-overlapping *if*

$$\forall a_1, a_2 \in \mathbb{A}\colon\ a_1 \neq a_2 \implies f(a_1) \cap f(a_2) = \emptyset.$$

An attribute mapping is contiguous *if*

$$\forall a \in \mathbb{A}\, \exists i, j \leq n\colon\ f(a) = [i, j).$$

Given a dump set S and all attribute values for each dump in S, the *carving problem for attributed dump sets* is the problem of finding an attribute mapping for S.

The existence of such a mapping does not imply that the attributes are indeed encoded in the dump, but merely that they could have been encoded at the indicated positions in the dumps. Conversely, if an attribute cannot be mapped in S, it means that this attribute is not present through a deterministic, injective encoding. Of course, this does not rule out the possibility that a non-deterministic encoding is used, such as a probabilistic encryption, or that the attribute is stored dynamically, i.e. not always at the same location. We consider the search for high-entropy information and semi-dynamic attributes later in this paper.

The notion of an attribute mapping is illustrated in Figure 1. This example consists of five dumps, $s_1, \ldots s_5$, of length $n = 18$. We look at the attribute *rides-left* (rl) with the values as given in the figure and we consider two possible encodings enc_1 and enc_2. The first encoding is the standard binary encoding of natural numbers. It can be found in the dumps at two different (contiguous) positions: $[5, 8]$ and $[12, 15]$. The second encoding, which is not standard, occurs at positions $[3, 6]$. Each of these three cases defines a contiguous attribute mapping for *rides-left*. There might be more candidate encodings.

	rl	dump	enc_1	enc_2
s_1	4	010100100111010000	0100	1001
s_2	4	001100100001010010	0100	1001
s_3	5	101110101011010100	0101	1101
s_4	6	001010110111011011	0110	0101
s_5	6	111010110011011001	0110	0101

Figure 1: Example of a dump set with three possible attribute mappings.

4 Commonalities and dissimilarities

Given the values of an attribute for the dumps in a dump set S, we can use the commonalities and dissimilarities of these dumps to derive restrictions on the possible attribute mappings for S. Such restrictions are derived in two steps. In the first step we look at dumps that have the same attribute value. In this case, we can derive those positions in the bit strings that cannot occur in the encoding of the attribute. In the second step we look at dumps of which the attribute values differ, allowing us to determine positions in the bit strings that should occur in the encoding of the attribute.

For the first step, we start by observing that an attribute $a \in \mathbb{A}$ induces a partition

$$\text{bundles}(a, S) = \{\{s \in S \mid \text{val}_a(s) = d\} \mid d \in \text{type}(a)\}$$

on a dump set S. An element of this partition is called a *bundle*. Thus, a bundle is a set of dumps with the same attribute value. For instance, Figure 1 shows three bundles for attribute *rides-left* (rl), namely $\{s_1, s_2\}$, $\{s_3\}$, and $\{s_4, s_5\}$.

The *common set* determines which bits in the dumps of a dump set are equal if the attribute values are equal.

Definition 2. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ be a dump set. The* common set *of S with respect to a, denoted by $\text{comm}(a, S) \subseteq [0, n)$, is defined by*

$$\text{comm}(a, S) = \bigcap_{b \in \text{bundles}(a,S)} \{i \in [0, n) \mid \forall s, s' \in b\colon\ s_i = s'_i\}.$$

An example is given in Figure 3. The elements from the common set are marked with an asterisk.

Given that the encoding of an attribute value is deterministic, this gives an upper bound on the bits used for this attribute.

Lemma 1. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then*

1. $\forall a \in \mathbb{A}\colon\ f(a) \subseteq \text{comm}(a, S)$,
2. *if $I_a \subseteq [0, n)$ is a family of sets for $a \in \mathbb{A}$, such that $f(a) \subseteq I_a \subseteq \text{comm}(a, S)$, then the function $f'\colon \mathbb{A} \to \mathcal{P}([0, n))$, defined by $f'(a) \mapsto I_a$, is an attribute mapping.*

The first property states that every possible attribute mapping is enclosed in the common set, so one can restrict the search for attribute mappings to the locations in the common set. The second property expresses that every extension of an attribute mapping is also an attribute mapping, provided that it does not extend beyond the common set.

Next we look at dumps with different attribute values. Injectivity of the encoding function implies that the encoding of two different values must differ at least in one bit. This is captured in the notion of a *dissimilarity set*. This set consists of all intervals that, for each pair of dumps with a different attribute value, contain at least one location where the two dumps differ.

Definition 3. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ be a dump set. The* dissimilarity set *of S with respect to a, denoted by $\mathrm{diss}(a, S) \subseteq \mathcal{P}([0, n))$, is defined by*

$$\mathrm{diss}(a, S) = \{I \subseteq [0, n) \mid \forall s, s' \in S\colon\ (\mathrm{val}_a(s) \neq \mathrm{val}_a(s') \implies \exists i \in I\colon\ s_i \neq s'_i)\}$$

An example of the dissimilarity set is given in Figure 4. The next lemma expresses that every attribute mapping is an element of the dissimilarity set. Consequently, we can restrict the search for possible attribute mappings to the elements of the dissimilarity set.

Lemma 2. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then $\forall a \in \mathbb{A}\colon\ f(a) \in \mathrm{diss}(a, S)$.*

An encoding of an attribute value a must at least contain the indexes from one of the sets in $\mathrm{diss}(a, S)$. This implies that we are mainly interested in the smallest sets in $\mathrm{diss}(a, S)$, i.e. those sets of which no proper subset is in $\mathrm{diss}(a, S)$. In order to make this precise, we introduce some notation.

Let F be a set and let $P \subseteq \mathcal{P}(F)$. We define the *superset closure* of P, notation $\overline{P}$, by $\overline{P} = \{p \subseteq F \mid \exists p' \in P\colon\ p' \subseteq p\}$. A set P is *superset closed* if $P = \overline{P}$. We observe from its definition that $\mathrm{diss}(a, S)$ is superset closed.

Given $P \subseteq \mathcal{P}(F)$, we say that P is *subset minimal* if for every $p, p' \in P$, $p' \subseteq p \implies p' = p$. Thus, a collection of sets is subset-minimal, if no set is a strict subset of any other set in the collection.

Lemma 3. *Let F be a finite set and let $P \subseteq \mathcal{P}(F)$. Then there exists a unique subset-minimal set Q such that $\overline{Q} = \overline{P}$.*

Given P as in Lemma 3, we denote the unique subset-minimal set by $\mathrm{smin}(P)$. Then, in order to determine whether an encoding of an attribute contains at least the indexes from one of the sets in $\mathrm{diss}(a, S)$, it suffices to verify that it at least contains one of the sets from $\mathrm{smin}(\mathrm{diss}(a, S))$.

By combining the results of the previous lemmas, we get the following main result.

Theorem 1. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then*

$$\forall a \in \mathbb{A}\ \exists I \in \mathrm{smin}(\mathrm{diss}(a, S))\colon\ I \subseteq f(a) \subseteq \mathrm{comm}(a, S).$$

This theorem says that if an attribute is expressed in a dump set, then its encoding position should contain at least one of the minimal dissimilarity sets and may not go beyond the common set.

A consequence of the theorem is that by calculating $\mathrm{diss}(a, S)$ and $\mathrm{comm}(a, S)$, we can limit the search space when looking for the attribute mapping $f(a)$ in the dumps. We will now investigate how to further limit the search space.

Let $\mathrm{filter}(A, c) = \{a \in A \mid a \subseteq c\}$ denote the filtration of a collection of sets in A with respect to a set c. It is easy to see that the sets of interest for an attribute mapping in Theorem 1 are characterized by the following set

$$\mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S))) \tag{1}$$

Let R be a set of representatives of $\mathrm{bundles}(a, S)$, i.e. $\forall b \in \mathrm{bundles}(a, S)\ \exists! s \in R\colon\ s \in b$. The following theorem states that the set

$$\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) \tag{2}$$

contains the same index sets as (1). Expression (2) suggests, however, a smaller search space than (1), since the diss function is computed only over a restricted set of indexes and a subset of the dump set.

Theorem 2. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ a dump set. Let R be a set of representatives of $\mathrm{bundles}(a, S)$. Then $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) = \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S)))$.*

To build up our intuition, we first formulate the lemma that by expanding a dump set we might be able to locate an attribute more precisely.

Lemma 4. *Let S, $S' \subseteq \mathbb{B}^n$ be dump sets and $a \in \mathbb{A}$ an attribute. Then $S' \subseteq S \implies \mathrm{diss}(a, S') \supseteq \mathrm{diss}(a, S)$.*

The preceding lemma indicates in particular that a dump set contains more information about an attribute than its subset of representatives. If we filter the $\mathrm{diss}(a, S)$ sets with respect to the $\mathrm{comm}(a, S)$ set, however, then the representatives are sufficient.

Lemma 5. *Let $S \subseteq \mathbb{B}^n$ be a dump set and $a \in \mathbb{A}$ an attribute. Let R be a set of representatives of* $\text{bundles}(a, S)$. *Then* $\text{filter}(\text{diss}(a, S), \text{comm}(a, S)) = \text{filter}(\text{diss}(a, R), \text{comm}(a, S))$.

The filter with respect to the $\text{comm}(a, S)$ set in the preceding Lemma is indeed necessary. In general, the set $\text{diss}(a, R)$ does not coincide with $\text{diss}(a, S)$.

Consider, for instance, the three two-bit dumps $s_1 = 01$, $s_2 = 00$, and $s_3 = 11$. Suppose the dumps encode the attribute a with $\text{val}_a(s_1) = \text{val}_a(s_2) = A$ and $\text{val}_a(s_3) = B$. Then we have the following bundles and dissimilarity sets.

$$
\begin{aligned}
\text{bundles}(a, \{s_1, s_2, s_3\}) &= \{\{s_1, s_2\}, \{s_3\}\} \\
\text{diss}(a, \{s_1, s_2, s_3\}) &= \{\{0\}, \{0, 1\}\} \\
&= \overline{\{\{0\}\}} \\
\text{diss}(a, \{s_2, s_3\}) &= \{\{0\}, \{1\}, \{0, 1\}\} \\
&= \overline{\{\{0\}, \{1\}\}}
\end{aligned}
$$

Thus, in spite of the fact that s_1 and s_2 have a common value for the attribute a, considering both in the dissimilarities set provides more information.

Finally, if we assume that the sizes of the attribute value domains are known, we have an information-theoretic lower bound on the number of bits that must have been used for encoding the attribute. This is expressed in the following lemma, which can be used to further limit the search space. The lemma follows from the pigeonhole principle.

Lemma 6. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then* $\forall a \in \mathbb{A}\colon\ |f(a)| \geq \log_2(|\text{type}(a)|)$.

In Section 6, we will investigate algorithms for determining the sets $\text{smin}(\text{diss}(a, S))$ and $\text{comm}(a, S)$.

5 Cyclic attribute mappings

In this section we extend our results to a class of dynamic mappings, which we call semi-dynamic or cyclic mappings. Cyclic mappings can, for instance, be used to store *trip frames* on a public transportation card. Such a trip frame contains all information related to a single ride. Trip frames are stored in one of a fixed number of slots in the card's memory. When validating the card for a new ride, a new trip frame will be written to the next available slot. If all slots have been filled, the next trip frame will be written to the first slot again, etc. We will show that cyclic mappings can be detected by the same algorithms as static mappings at the cost of introducing a number of derived attributes.

Because cyclic mappings consider the evolution of a given object in time, we will first assume additional structure on the dump set corresponding to the history of an object. We assume that for each dump we can determine to which object it belongs through the attribute id (e.g. the unique identifier of a public transportation card). For each object we further assume that its dumps are ordered as expressed by an attribute $seqnr$.

Definition 4. *Let $S \subseteq \mathbb{B}^n$ be a dump set and let id and seqnr be attributes. We say that the pair $(id, seqnr)$ is a* bundle-ordering *if* $\text{type}(seqnr) = \mathbb{N}$ *and*

$$
\begin{aligned}
&\forall b \in \text{bundles}(id, S)\ \forall s, s' \in b\colon \\
&\qquad s \neq s' \implies \text{val}_{seqnr}(s) \neq \text{val}_{seqnr}(s').
\end{aligned}
$$

Because the combination of a device identifier and a sequence number uniquely determines a dump, we can consider an attribute a as a function on $\text{type}(id) \times \mathbb{N}$. Given $i \in \text{type}(id)$ and $n \in \mathbb{N}$ we will thus write $a(i, n)$ for $\text{val}_a(s)$, where $s \in S$ is the dump uniquely determined by $\text{val}_{id}(s) = i$ and $\text{val}_{seqnr}(s) = n$.

Using this notation, we are now able to derive new attributes from a given attribute a. In particular, we can consider the history of a device. An example is the attribute $a_{\text{-}1}$, which determines the a-value of the direct predecessor of a dump. This attribute is defined by $a_{\text{-}1}(i, n) = a(i, n - 1)$. It is defined on a subset of S, viz.

$$
\begin{aligned}
\{s \in S \mid \exists s' \in S\colon\ &\text{val}_{id}(s') = \text{val}_{id}(s) \wedge \\
&\text{val}_{seqnr}(s') = \text{val}_{seqnr}(s) - 1\}.
\end{aligned}
$$

This generalizes to $a_{\text{-}r}$ for $r \in \mathbb{N}$. By extending the set of attributes with such *derived attributes*, we can automatically verify if a dump contains information on the history (i.e. the previous states) of a device.

This technique is particularly useful when dealing with cyclic attribute mappings. A cyclic mapping of attribute a considers a number of locations to store the value of a, e.g., $[i_1, j_1)$, $[i_2, j_2)$ and $[i_3, j_3)$. In the first dump of an ordered id-bundle the value of a is stored at $[i_1, j_1)$. In the second dump a is stored at $[i_2, j_2)$, etc. The location for the fourth value of a is again $[i_1, j_1)$.

In order to locate a cyclic mapping for attribute a, we will derive new attributes $a_{cycle(x/c)}$, where c is the length of the cycle and x is a sequence number ($0 \leq x < c$). Using notation $\lfloor r \rfloor$ for the *floor* of rational number r, we obtain the following extensional definition of these new attributes:

$$
a_{cycle(x/c)}(i, n) = a(i, c \cdot \left\lfloor \frac{n - x}{c} \right\rfloor + x).
$$

In Figure 2 we show the attributes derived from the rides-left (rl) attribute, assuming a cyclic mapping of length

3. The dumps s_1 to s_5 are consecutive dumps of a single card. In order to find the cycle length of a cyclically mapped attribute, it suffices to search for attributes $a_{cycle(0/c)}$, where c ranges from 2 to the expected maximum cycle length. In the figure we denote $rl_{cycle(x/c)}$ by $rl_{x/c}$.

	rl	$rl_{0/3}$	$rl_{1/3}$	$rl_{2/3}$	$seqnr_{mod(3)}$
s_1	8	8	-	-	1
s_2	7	8	7	-	2
s_3	6	8	7	6	3
s_4	5	5	7	6	1
s_5	4	5	4	6	2
s_5	3	5	4	3	3

Figure 2: Derived attributes with cycle length 3.

We conclude our observations on cyclic mappings by considering pointers to such attributes. An example is the use of a pointer (at a static location), pointing at the block in memory where the information on the most recent trip is stored. Clearly, if the trip information is stored alternatingly at different locations, the pointer will have a similar cyclic behaviour. We can search for such cyclic pointers by introducing attributes $seqnr_{mod(c)}$, which consider the sequence number of the dump modulo cycle length c. Figure 2 contains an example for $c = 3$.

6 Algorithms

In the following we concern ourselves with the two basic carving algorithms, comm and diss.

6.1 Commonalities

The algorithm computing the comm function identifies all positions in which given bitstrings have the same value. We implement it using the function $\mathrm{fc}\colon \mathcal{P}(\mathbb{B}^*) \times \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ which we define recursively as follows, using the symbol $\uplus$ for the disjoint union of sets.

$$\begin{aligned} \mathrm{fc}(\emptyset, I) &= I \\ \mathrm{fc}(\{s\}, I) &= I \\ \mathrm{fc}(S \uplus \{s, s'\}, I) &= \mathrm{fc}(S \cup \{s\}, \{i \in I \mid s_i = s'_i\}) \end{aligned}$$

Obviously, for dumps of length n,

$$\mathrm{comm}(a, S) = \bigcap_{b \in \mathrm{bundles}(a,S)} \mathrm{fc}(b, [0, n)).$$

The bit complexity of this step is $O(n \cdot |S|)$.

The function comm is illustrated in Figure 3. For each of the three bundles we have calculated the fc set as the set of all positions where all dumps from the bundle agree on the bit (indicated by the asterisk symbols). Finally, the comm set cm is the intersection of these fc sets.

	rl	dump	
s_1	4	010100100111010000	
s_2	4	001100100001010010	
		*..******..*****.*	fc
s_3	5	101110101011010100	
		******************	fc
s_4	6	001010110111011011	
s_5	6	111010110011011001	
		..*******.******.*	fc
		...******..*****.*	cm

Figure 3: Calculation of the comm set.

6.2 Dissimilarities

Given a set of bundles, the algorithm for the diss function identifies intervals in which any two bitstrings from different bundles differ in at least one position.

We implement the diss function in the case where the attribute mapping is assumed to be contiguous using the *dissimilarity interval function* $\mathrm{iv}(a, S)(i)$. It denotes the shortest interval that a contiguous encoding of attribute a must have if it is to start at position i. Such an interval does not exist if there are dumps in S which do not differ at any position in $[i, n)$.

Definition 5. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ be a dump set. The* dissimilarity interval function $\mathrm{iv}(a, S) : [0, n) \to \mathcal{P}([0, n)) \cup \{\bot\}$ *of S with respect to attribute a is defined by*

$$\begin{aligned} \mathrm{iv}(a, S)(i) = [i,\ \min\{k \in [i, n) \mid \\ \forall d, d' \in S\colon\ \mathrm{val}_a(d) \neq \mathrm{val}_a(d') \implies \\ \exists j\colon\ (i \leq j \leq k \wedge d_j \neq d'_j)\}\ ] \end{aligned}$$

if the minimum exists and $\bot$ else.

The following lemma expresses that the dissimilarity set for contiguous attribute maps can be obtained from the dissimilarity interval function. To state the lemma, we first need to define subset minimality and superset closure for sets of intervals.

Let $\mathcal{I}_n = \{[i, j] \subseteq \mathbb{N} \mid i, j < n\}$ be the set of intervals in $[0, n)$. We define the *interval-superset closure* of a set $P \subseteq \mathcal{I}_n$ by $\{p \in \mathcal{I}_n \mid \exists p' \in P\colon\ p' \subseteq p\}$. It is easy to see that the interval-superset closure of P is equal to $\overline{P} \cap \mathcal{I}_n$. A set P is said to be *interval-superset closed* if $P \subseteq \mathcal{I}_n$ and $P = \overline{P} \cap \mathcal{I}_n$. We say that P is *interval-subset minimal* if $P \subseteq \mathcal{I}_n$, and for every $p, p' \in P$, $p' \subseteq p \implies p' = p$. It is also easy to see that for every set of intervals $P \subseteq \mathcal{I}_n$, there is a unique interval-subset minimal set $Q \subseteq \mathcal{I}_n$ such that $\overline{Q} \cap \mathcal{I}_n = \overline{P} \cap \mathcal{I}_n$. The proof is analogous to the proof of Lemma 3.

Lemma 7. *Let $S \subseteq \mathbb{B}^n$ be a dump set and $a \in \mathbb{A}$ an attribute. Let the set T be defined by*

$$T = \{\mathrm{iv}(a,S)(i) \in \mathcal{I}_n \mid i \in [0,n) \wedge \mathrm{iv}(a,S)(i+1) \not\subseteq \mathrm{iv}(a,S)(i)\}$$

then T is the interval-subset-minimal set that satisfies $\overline{T} \cap \mathcal{I}_n = \mathrm{diss}(a,S) \cap \mathcal{I}_n$.

To compute $\mathrm{iv}(a,S)(i)$ for $i \in [0,n)$, we assume for simplicity of exposition that no two dumps in S have the same value for attribute a, that is, we are restricting ourselves to a set of representatives R of $\mathrm{bundles}(a,S)$.

A naive algorithm for $\mathrm{iv}(a,R)(i)$ is to first compare two dumps from R then to iterate over all remaining dumps in R comparing each new dump to the first two dumps and all dumps that have already been iterated over. In each comparison of two dumps, the first position after position i in which the two dumps differ is sought for. The maximal such position is returned. More precisely, let $\mathrm{fiv} : \mathcal{P}(\mathbb{B}^*) \times \mathbb{N} \to \mathbb{N} \cup \{-\infty,\infty\}$ be defined recursively as follows. Note that we adopt the conventions $\min(\emptyset) = \infty$, $\max(\infty,k) = \infty$, and $\max(-\infty,k) = k$ for all $k \in \mathbb{N} \cup \{-\infty,\infty\}$.

$$\begin{aligned}
\mathrm{fiv}(\emptyset,i) &= -\infty \\
\mathrm{fiv}(\{s\},i) &= -\infty \\
\mathrm{fiv}(\{s,s'\},i) &= \min\{k \in \mathbb{N} \mid k \geq i, s_k \neq s'_k\} \\
\mathrm{fiv}(R \cup\!\!\!\cdot\, \{s\}) &= \max(\mathrm{fiv}(R,i), \\
&\qquad \max_{s' \in R}\{\mathrm{fiv}(\{s,s'\},i)\})
\end{aligned}$$

Then for any set R of representatives from $\mathrm{bundles}(a,S)$, we have $\mathrm{iv}(a,R)(i) = [i, \mathrm{fiv}(R,i)]$ if $\mathrm{fiv}(R,i) \in \mathbb{N}$ and $\mathrm{iv}(a,R)(i) = \bot$ else. The number of comparisons of two dumps, i.e. the number of calls to $\mathrm{fiv}(\{s,s'\},i)$, is easily seen to be quadratic in $|R|$. We can improve the number of comparisons to $O(|R| \log |R|)$ by sorting the set of dumps first. We will write $s <_i s'$ if and only if $\exists j \in [i,n)\colon\ s_j < s'_j \wedge \forall i \leq k < j\colon\ s_k = s'_k$. We will write $s \leq_i s'$ if $s <_i s'$ or $\forall j \in [i,n)\colon\ s_j = s'_j$.

A more efficient algorithm $\mathcal{A}$ to compute $\mathrm{iv}(a,R)(i)$ runs as follows.

1. Sort the dump set R in ascending order with respect to $\leq_i$. Let $s^{(1)} \leq_i s^{(2)} \leq_i \ldots \leq_i s^{(|R|)}$ be the sorted list of these dumps.

2. For j from 1 to $|R|-1$, compare $s^{(j)}$ with $s^{(j+1)}$. For the comparison, start with the i-th bit and move towards the $n-1$-st bit. Let k_j be the index of the first bit in which $s^{(j)}$ differs from $s^{(j+1)}$. If no such bit exists, output $\bot$ and stop.

3. Output the interval $[i, \max_{j \in [1,|R|]}(k_j)]$.

Theorem 3. *Let $S \subseteq \mathbb{B}^n$ be a dump set and $a \in \mathbb{A}$ an attribute. Let R be a set of representatives of the sets in $\mathrm{bundles}(a,S)$. Then the set T with $\overline{T} \cap \mathcal{I}_n = \mathrm{diss}(a,R) \cap \mathcal{I}_n$ is computed by $\mathcal{A}$ in time $O(n^2\,|R| + n\,|R| \log |R|)$.*

The calculation of the diss set is explained in Figure 4. We start by taking a representative of each of the bundles. Then, starting from the left, we calculate for each position how far to the right we must go in order to find a distinguishing bit for each pair of dumps. For position 0 the first two bits already make a distinction between the three dumps, which gives the interval $[0,1]$ (indicated by the first line with asterisk symbols). For position 1 we need three bits, because s_3 and s_4 coincide at positions 1 and 2. This gives the interval $[2,4]$, etc. Those sets belonging to the subset-minimal diss set are marked with "minimal".

	rl	dump	
s_1	4	010100100111010000	
s_3	5	101110101011010100	
s_4	6	001010110111011011	
		**................	minimal
		.***..............	
		..**..............	minimal
		...**.............	minimal
		....****..........	minimal
		.....****.........	
		......***.........	
		.......**.........	minimal
		etc.	

Figure 4: Calculation of the diss set.

If we combine the comm set from Figure 3 and the diss set from Figure 4, under the assumption that the number of rides is encoded with 4 bits, we obtain the four remaining possibilities from Figure 5. This result includes the three possible attribute mappings from Figure 1.

	rl	dump
s_1	4	010100100111010000
s_2	4	001100100001010010
s_3	5	101110101011010100
s_4	6	001010110111011011
s_5	6	111010110011011001
		...****...........
		....****..........
		.....****.........
		............****..

Figure 5: The resulting attribute mappings.

7 The mCarve tool

We have implemented the algorithms of Section 6 in a prototype called mCarve [12]. The prototype allows the forensic analyst to input a collection of dumps and a collection of attributes. Each of the dumps can be accompanied by its attribute values. The prototype was written in Python and consists of approximately 1200 lines of code (excluding graphical user interface).

After entering the dumps and attributes the user can run the *commonalities* algorithm for an attribute. The output of the algorithm is the set of indexes I for which all dumps with the same attribute value are the same. The set I is used as a coloring mask to display any dump d selected by the user: if $i \in I$, then d_i is colored blue, otherwise red. The *dissimilarities* algorithm computes a subset-minimal set of dissimilarity intervals. Since these intervals may be overlapping, the prototype enumerates them rather than showing them as one coloring mask. This allows the user to step through the intervals. The prototype displays the interval iv by applying a yellow coloring mask to all bits d_i for $i \in \text{iv}$. A *combined* procedure consolidates the results from the commonalities and dissimilarities algorithms.

The prototype further allows users to specify two types of special attributes: a *constant* attribute and a *hash* attribute. The former has a constant value for all dumps and can be used to determine which bits never change. The latter has a different value for all pairwise different dumps and can be used to detect encrypted attributes. The tool allows one to derive new attributes from other attributes. These derived attributes can be used to find cyclic attribute mappings. The tool further allows one to apply an encoding to a selected interval in each dump. A number of standard encodings, such as ASCII and base 10, are implemented. Aside from displaying the output onscreen, the user can choose to export the results to JPEG or to LaTeX (see Figure 7 for an example).

7.1 Performance

We illustrate the performance of our prototype by running our prototype on a generated test suite. The test suite consists of dumps of sizes 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB. For each file size, 5 dump sets were generated. Each dump embeds one attribute at a random position and is encoded in at most 64 bits. The remaining bits are randomly generated.

The running time of the commonalities procedure is linear in the number of dumps and the dissimilarities procedure is quadratic in the number of bundles. Therefore, the execution time of the combined procedure is mainly dependent on the number of bundles in the dump set. Convergence tests show that, in general, fewer than 10 bundles are needed to find an attribute in a dump set. This allows us to restrict our performance tests to dump sets of 10 bundles.

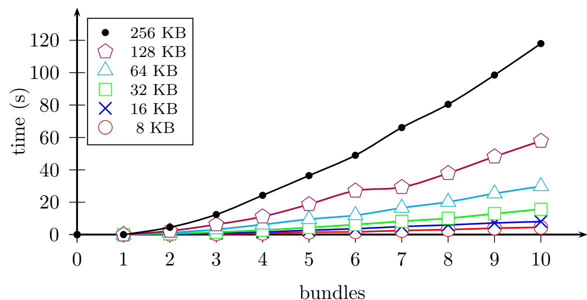

Figure 6: Performance

The tests were run on a Linux machine (kernel 2.6.31-22) with Intel Core 2 6400 @ 2.13 GHz processor running Python 2.6.4. Figure 6 shows on the horizontal axis the number of bundles included in the dump set. On the vertical axis it shows for each of the file sizes the time in seconds (averaged over the 5 dump sets) needed to perform the combined procedure. The test shows that our prototype is best suited for dumps of size smaller than 32KB, but it can deal reasonably well with size up to 256KB. Initial experiments have shown that performance of the tool can be significantly improved by implementing the core procedures in a lower-level language.

7.2 Convergence

Another interesting measure for the mCarve tool is the *rate of convergence* of the carved intervals. We will measure it by computing the number of dumps that are necessary in order to find an attribute in a dump set. For simplicity, we assume that the dumps as well as the attribute values are given by a uniformly random distribution.

Let q denote the bit length of the attribute's encoding in the dump, let N denote the number of dumps and let x be the number of bundles. We first compute the probability of false positives, i.e. the probability of an accidental occurrence of values matching an attribute. The probability that the bit string formed by a particular interval of q bits in all N dumps matches a particular given string of bits is 2^{-qN}. There are $\binom{2^q}{x} \cdot x!$ possible encodings of x different values. The probability that the q bits in all N dumps match one of these representations is therefore $2^{-qN}\binom{2^q}{x} \cdot x!$.

Thus if l denotes the length of the bit strings representing dumps, then the probability p_{nfp} of no false positives

is given by

$$p_{nfp} \geq \left(1 - \frac{2^{-qN} \cdot 2^q!}{(2^q - x)!}\right)^{l-q+1}.$$

The inequality is due to the fact that the product on the right does not concern independent trials. We are interested in those values of x and N for which the probability p_{nfp} is large enough that the discovery of an attribute is not coincidental.

Using the inequality $\binom{n}{k} \leq \frac{n^k}{k!}$ we obtain

$$\left(1 - \frac{2^{-qN} \cdot 2^q!}{(2^q - x)!}\right)^{l-q+1} \geq \left(1 - 2^{q(x-N)}\right)^{l-q+1}.$$

Fixing the number of bundles x and a false positive probability ϵ, we obtain the following inequality for the number of dumps N:

$$\left(1 - 2^{-(N-x)q}\right)^{l-q+1} > 1 - \epsilon.$$

Thus

$$N > \frac{-1}{q} \log_2 \left(1 - (1-\epsilon)^{\frac{1}{l-q+1}}\right) + x.$$

This formula can be used in two ways. If we know the length q of the encoding, we fix a number of bundles x and a false positive probability ϵ and compute the number of dumps N needed for convergence. If we do not know the length of q, we set it to $\log_2(x)$ and perform the same computation. For instance, for dumps of length $l = 1024$, false positive probability of $\epsilon = 0.05$, number of bundles $x = 4$, and length $q = \log_2(x) = 2$ we get $N > 11.14$. This means that to have convergence with probability 0.95 we need to analyze 12 dumps comprising 4 different attribute values.

8 Case study: The E-go system

We illustrate our methodology by reverse engineering part of the memory structure of the Luxembourg public transportation card.

8.1 The E-go system

The fare collection system for public transportation in Luxembourg, called e-go, is based on radio frequency identification (RFID) technology. The RFID system consists of credit-card shaped RFID *tags* that communicate wirelessly with RFID *readers*. Readers communicate with a central back-end system to synchronize their data. Travelers can buy e-go cards with, for instance, a book of 10 tickets loaded on it. Upon entering a bus, the user swipes his e-go card across a reader and a ticket is removed from the card.

Since most RFID readers of the e-go system are deployed in buses the e-go is an *off-line* RFID system [5]. Readers do not maintain a permanent connection with the back-end system, but synchronize their data only infrequently. Since readers may have data that is out-of-date and tags may communicate with multiple readers, the e-go system has to store information on the card.

The RFID tags used for the e-go system are, in fact, MIFARE classic 1k tags. These tags have 16 sectors that each contain 64 bytes of data, totaling 1 kilobyte of memory. Sector keys are needed to access the data of each sector. Garcia et al. [4, 6] recently showed that these keys can be efficiently obtained with off-the-shelf hardware. Therefore, it is easy to create a memory dump of an e-go card.

8.2 Data collection

Over a period of 2 months, we collected 68 dumps for 7 different e-go cards of different types. Four cards are of type *10-rides/2nd-class*, two of type *1-ride/2nd-class* and one of type *1-ride/1st-class*. According to information published by the transportation companies, a card can contain up to 6 products of the same type. We considered two classes of events that change the state of a card: (1) charging the card with a new product (including the purchase of a new, charged card), and (2) validating a ride by swiping the card. After each event we dumped the memory of the card as a binary file. This gave a sequence of consecutive events for each card.

Because the e-go system is an off-line system, we expected to find several attributes encoded on the card. For each event we therefore collected some contextual information, which we attributed to the dump following the event. For charge events we collected the following attributes: card id (the decimal number printed on the card); charged product; date, time and location of charging; card charger id (as printed on the coupon). For validation events we collected: card id; date and time of swiping; expiration time of the ride; card reader id (because the card readers have no visible identification we collected the license plate number of the bus and the location of the reader within the bus); rides left; bus number; bus stop.

These are the attributes that one would expect to find on the card and that are easy to observe. Most of these attributes can be obtained by reading the sales slips or the display of the reader. Since cards are purchased anonymously, no personal identifying information, such as name, address, or date-of-birth can be stored on the card.

In addition to our basic set of dumps, we had access

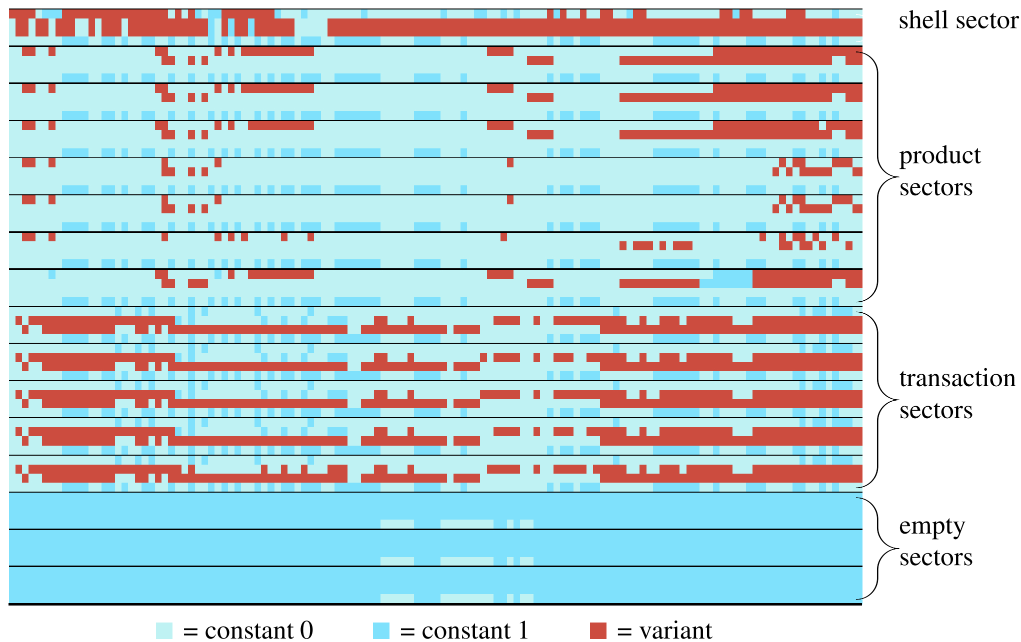

Figure 7: E-go memory layout (applying common to a unique attribute).

to 47 dumps from earlier experiments which were less structured and less documented. We used these dumps to validate the results of the experiments with our main set of dumps.

It is important to note that our analysis is entirely passive: no data on the card needs to be modified and no data needs to be written to the card.

8.3 Data analysis

Using our tools, we verified the presence of three classes of attributes: (1) external attributes (i.e., the observable attributes mentioned above); (2) internal attributes (related to the organization of the data within the card's memory, such as a pointer to the active sector); and (3) attributes with high entropy (such as CRCs and cryptographic checksums). We also searched for cyclic versions of these attributes.

Memory layout. The first step in our analysis is to determine the general memory layout of an e-go card. For this purpose we apply the commonalities algorithm to the constant attribute, i.e., an attribute that has a constant value for every dump. The result of this operation is shown in Figure 7. The card's memory is displayed in 64 lines of 128 bits, giving a total of 8192 bits (1kB). Bits that have a constant value in all dumps are colored differently from bits that vary in value. The recurring structures immediately suggest a partitioning of the memory into 16 sectors of 4 lines each. There seem to be four different types of sectors. The structure of the first sector is unique. We call this sector the *shell sector*. Lines 2 and 3 of the shell sector are identical. Next there are seven sectors with a similar appearance (three of these look a bit less dense than the others because they are used less frequently in our dump set). We call these sectors the *product sectors*. The next five sectors are similar. We call them *transaction sectors*. Finally, there are three *empty sectors*, which we will ignore for the rest of our analysis. They are probably reserved for future extensions of the e-go system.

Further inspection shows that the last line of each sector is constant (over all dumps). This is the 16 byte *sector key*. Because the last lines of each of the sectors (except the empty sectors) are equal, we can conclude that the same key is used for all sectors.[1]

External attributes. The second step in our analysis is to carve the external attributes. This step only revealed the card ID. We can conclude that the other external attributes are either not represented on the card or not at a static location. Figure 8 shows for each sector type which attributes were discovered with our tool. The card ID, which is located in the shell sector in Figure 8, is detected as follows. The output of our tool on the card ID attribute consists of a number of intervals between bits 0 to 37 plus the interval 35 to 108. Clearly, the last interval is too large to contain the card ID, so we can consider that interval a false positive. We conclude that bits 0 to 37 are

[1] In order to not reveal sensitive data, we display keys that are different from those used in the e-go system.

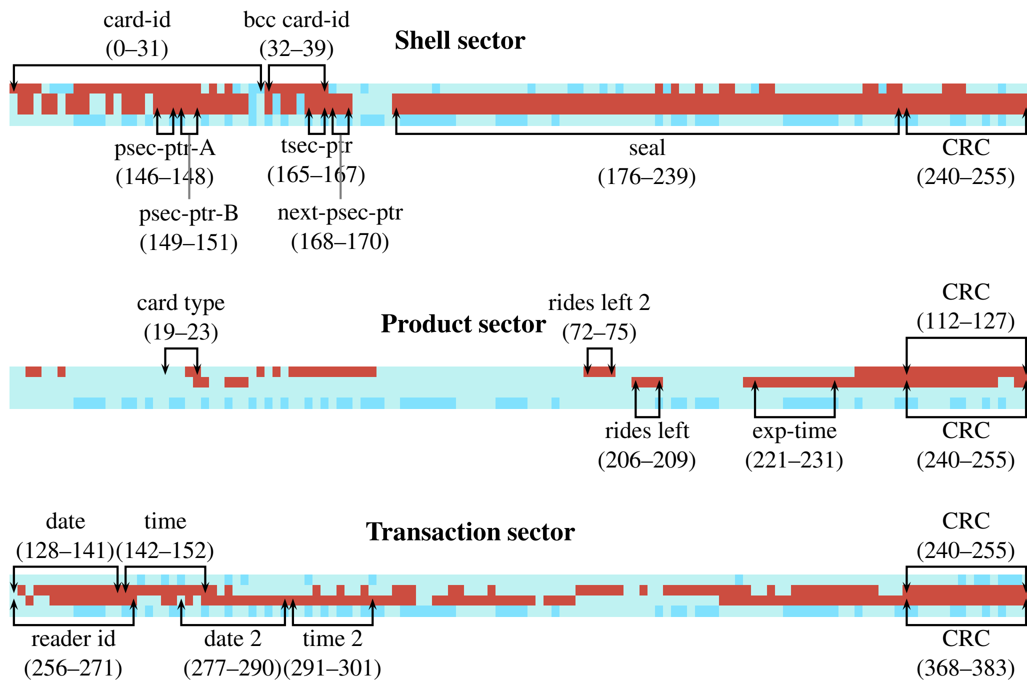

Figure 8: Attributes located in the three sector types.

related to the card ID. Indeed, the MIFARE standard describes that identification numbers are hard-coded in the first 32 bits (4 bytes). If we reverse these 4 bytes and interpret them as a decimal number, we obtain the number printed on the card. The fact that bits 32 to 37 relate to the card ID is also consistent with the MIFARE standard because bits 32 to 39 contain the checksum of the card ID.

Internal attributes. The tool can be used to step through a sequence of dumps and observe the changes between consecutive dumps. In this way, one can step through the "history" of a particular card and observe recurring patterns. This process indicates a periodicity in the updates of the transaction sectors of the e-go card. Successive validation events write to successive transaction sectors, thereby cycling back from the fifth transaction sector to the first. One would expect a similar periodicity in the product sectors, but that is not the case. Writing to the product sectors occurs in an alternating way between two selected sectors. Based on the hypothesis that there is a notion of a "current" sector, we carve for pointers with cycle lengths 2 to 7. By making a selection of those sequences of dumps that showed the cyclic behaviour, we can locate a pointer to the currently active transaction sector (see tsec-ptr in the shell sector of Figure 8). This 3-bit pointer has a cycle of length 5 from 000 to 100. In a similar way one obtains a pointer with cycle 2, located at bit 169. Inspection of dumps reveals that this concerns a 3-bit pointer to the next active product sector (next-psec-ptr, bits 168-170). Two other pointers with cycle 2 are only revealed when carving well chosen subsets of the collection of dumps. In the figure they are labelled with psec-ptr-A and psec-ptr-B. When stepping through the dumps, it becomes clear that after each validation event the values of next-psec-ptr and one of psec-ptr-A or psec-ptr-B are swapped. When charging the card, psec-ptr-A and psec-ptr-B change roles.

Cyclic external attributes. After having been able to locate only a single static external attribute, we continue by searching for dynamically stored external attributes. By using cycle length 5, we can find two locations in each of the transaction sectors related to the date of the most recent validation In Figure 8 these locations are labelled with "date" and "date 2". By stepping through a sequence of dumps swiped on consecutive days, it becomes clear that the date field is a counter. It counts the number of days since 1/1/1997. In our dump set the two dates are always identical. In a similar way we can find two fields related to the time of the most recent validation event. They count the number of minutes since midnight. The first and second time are different, but, surprisingly, their difference is not constant, which would have indicated a relation to the expiration time. The last attribute that can be located in the transaction sector is the reader ID. As explained, we use the license plate of the bus and the location of the reader within the bus to identify each card reader. By combining these two attributes we obtain a new attribute that relates to the reader ID. Surprisingly,

this new attribute does not occur in the dumps, but the license plate attribute does. This means that all readers in a given bus have the same id. When interpreting the reader as a decimal number, one typically obtains numbers in the range from 1 to 150 for readers in a bus and from 10150 to 10200 for readers in a train station. This is consistent with carving for the attribute “bus-or-train”, which points at the higher bits of the reader id.

These attributes were found by reducing cyclic attributes to static attributes as described in Section 5. With this approach an attribute of cycle 5 will change its value only every 5 dumps. As a consequence, this attribute has a rather slow convergence rate. Convergence can be improved, however, by focusing on the *active* transaction sector. In order to do this we created a new set of dumps, each of which only contained the active transaction sector of the old dump. Carving for the static external attributes in this new set of dumps results in the same findings, but the attributes can be located with significantly fewer dumps.

Using this approach we can easily locate three more attributes in the product sectors: the card type, the number of rides left on the card and the expiration time of the current product. A second field related to the number of rides left was also located (rides left 2 in the figure), which equals 12 minus rides left for 10-rides cards and 3 minus rides left for 1-ride cards.

Finding high entropy attributes. While using the tool, one quickly observes that the diss function returns intervals of varying widths sliding through the index set of the dumps. Heuristically, one expects the width of these sliding windows to be shorter over intervals corresponding to high-entropy attributes than over indexes corresponding to low entropy attributes. Furthermore, the step size or distance between two such windows is expected to be smaller for high-entropy intervals.

The observation of short-step narrow sliding windows led to the conjecture that the cards contain cryptographic data.

To confirm the existence of high-entropy attributes, the MD5 hash of the dumps was computed and added as an attribute. The hash serves as a quick indicator for equality or inequality of two dumps and is a more robust approach to labeling distinct dumps with different attribute values than simply enumerating all dumps in a set. Carving for this artificial MD5 attribute amounts to looking for attribute values which change whenever the contents of the dump change. The tool thus revealed an 80-bit string in the shell sector. The same method applied to dumps of the product and transaction sectors revealed 16-bit strings which only change when the data in the corresponding sector changes.

Whereas an 80-bit string was expected to be a cryptographic hash, the 16-bit strings were suspected to be checksums such as CRCs. By trying out a list of commonly used CRCs to the data in the product and transaction sectors, the CRC-16-ANSI with polynomial $x^{16} + x^{15} + x^2 + 1$ was found to produce the observed values.

This step led to the suspicion that a CRC might also be part of the 80 bit string in the shell sector, which was indeed found to be the case. The remaining 64 bits are expected to be a cryptographic hash protecting the integrity of the card’s data.

Evaluation. Our tool performed quite well in this case study. We located the attributes as displayed in Figure 8 and have been able to infer the encoding scheme for most of them. On the other hand, we have not been able to locate all collected attributes. We did not find the date, time and location of charging, the card charger id, the bus number and the bus stop. Our experiments prove that they are not stored in a static or cyclic way on the card. We may assume that if the date and time of charging and the card charger id were represented in the card’s memory, they would have been encoded in the same way as the other dates, times and ids. A search of these encoded values in the binary dumps did not give a hit. Therefore, we conjecture that these attributes are not stored on the card, not even at a dynamically determined location. Given that a validated ride allows for unrestricted travel through the whole country for two hours, there is also no need to store the bus number and bus stop on the card.

As a consequence of carving for internal attributes we have not only located four pointers, but we have also reverse engineered part of the dynamics of updating e-go cards. The transaction sectors are written to cyclically. They contain data related to the history of the card. The current state of each of the products on the card is stored in the product sectors. Every product is assigned to one sector, except the currently active product. This product is updated alternatingly in two sectors. This redundancy is probably built in to keep a consistent product state even if a transaction does not finish successfully.

More safeguards against update errors are found in the frequent checksums that we have been able to locate. A protection against intentional modification of the stored data is the cryptographic seal in the shell sector.

Even though we found the majority of observed attributes, there are still locations in the card’s memory that we have not been able to assign a meaning to. Of course, the current dump set provides no information on the meaning of the constant (blue) bits in Figure 8. The variant (red) bits either have to do with the internal organization of the card or with attributes that we did not or could not observe.

With respect to convergence, we see that the dumps in this case study behave slightly worse than the dumps in

the idealized set from Section 7.2. Finding an attribute requires roughly 12 dumps (or 5 bundles).

Occasionally, we incorrectly entered an attribute value. The algorithms that we developed are not robust against such mistakes, since a single modification in the input can drastically change the output. In practice, however, such mistakes were quickly identified by regularly performing experiments on a subset of the dump set, such as all dumps belonging to a given card.

A very useful feature of our methodology is that in the search for an attribute we do not presuppose a particular encoding of that attribute. This allowed us to search for the combination of license plate number and reader location in order to find the reader ID. Similarly, we found a rides left counter counting down and one that counts up while searching for one attribute.

9 Conclusion and future work

We have defined the carving problem for attributed dump sets as the problem of recovering the attribute mapping and encoding of attributes in a dump. We have proposed algorithms for recovering the attribute mapping and proven their correctness. The first algorithm computes the commonalities to determine the positions in a dump that cannot be contained in the mapping. The second algorithm computes subset-minimal dissimilarities to give a lower-bound on the bits that need to be contained in the attribute mapping. By combining these two algorithms, a set of possible mappings is derived.

In order to validate our approach we have implemented a prototype, called mCarve, with commonality and dissimilarity algorithms. A case study performed on data from the electronic fare collection system in Luxembourg showed that mCarve is valuable in analyzing real-world systems. Using mCarve, we have located more than a dozen attributes on the e-go card as well as their encoding. We have also partly reverse engineered the dynamics of updating e-go cards.

There are several research directions that remain to be explored. To be able to understand the attribute values, the encoding has to be recovered as well. In our case study, we have recovered the encoding of attributes manually, while automatic approaches should in some cases be feasible. Heuristic approaches seem most viable, possibly approaches based on file carving techniques. Secondly, the robustness of our algorithms can be improved. Currently, a small error in the data, due to, for instance, a transmission error or a mistake in inputting the attribute value will make the results unreliable. Although these mistakes can be found by hand, an automatic way would be preferable.

We would like to apply mCarve to other case studies. An interesting application would be the memory of a cell phone. Our performance results show that we have to optimize the implementation of our algorithms to analyze cell phone dumps. Another use of mCarve will be to analyze proprietary communication protocols. By recording the data and applying our algorithms, we could reconstruct their specification.

References

[1] BILLARD, D., AND HAURI, R. Making sense of unstructured memory dumps from cell phones, 2009.

[2] COHEN, M. I. Advanced carving techniques. *Digital Investigation 4*, 3-4 (2007), 119–128.

[3] CONTI, G., DEAN, E., SINDA, M., AND SANGSTER, B. Visual reverse engineering of binary and data files. In *VizSec '08: Proceedings of the 5th international workshop on Visualization for Computer Security* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 1–17.

[4] GARCIA, F. D., DE KONING GANS, G., MUIJRERS, R., VAN ROSSUM, P., VERDULT, R., SCHREUR, R. W., AND JACOBS, B. Dismantling MIFARE classic. In *ESORICS* (2008), pp. 97–114.

[5] GARCIA, F. D., AND VAN ROSSUM, P. Modeling privacy for off-line RFID systems. In *CARDIS* (2010), pp. 194–208.

[6] GARCIA, F. D., VAN ROSSUM, P., VERDULT, R., AND SCHREUR, R. W. Wirelessly pickpocketing a MIFARE classic card. In *IEEE Security and Privacy* (2009), pp. 3–15.

[7] GARFINKEL, S. Carving contiguous and fragmented files with fast object validation. *Digital Investigation 4s* (2007), 2–12.

[8] HELFMAN, J. Dotplot patterns: A literal look at pattern languages. *TAPOS 2*, 1 (1996), 31–41.

[9] RUKHIN, A., SOTO, J., NECHVATAL, J., SMID, M., AND BANKS, D. A statistical test suite for random and pseudorandom number generators for statistical applications. *NIST Special Publication in Computer Security* (2000), 800–22.

[10] SENCAR, H. T., AND MEMON, N. Identification and recovery of JPEG files with msising fragments. *Digital Investigation 6* (2009), 88–98.

[11] SHAMIR, A., AND VAN SOMEREN, N. Playing "hide and seek" with stored keys. *Lecture Notes in Computer Science 1648* (1999), 118–124.

[12] VAN DEURSEN, T., MAUW, S., AND RADOMIROVIĆ, S. mCarve tool, 2011. Available at `http://satoss.uni.lu/mcarve`.

A Proofs

Proof of Lemma 1. We show the first property by contradiction. Assume that there exists an attribute a such that $f(a) \not\subseteq \mathrm{comm}(a, S)$. Then there exists an index $i \in f(a)$ such that $i \notin common(a, S)$. It follows from the definition of comm that there is a bundle that contains bit strings s and s' such that $s_i \neq s'_i$. However, since f is an attribute mapping, index $i \in f(a)$, and $val_a(s) = val_a(s')$, we have that $s_i = s'_i$. Thus, $f(a)$ must be a subset of $\mathrm{comm}(a, S)$.

The second property follows from the fact that if we extend an encoding, it remains an encoding. We know

that $e(val_s(a)) = s|_{f(a)}$ is an encoding for attribute mapping f. By definition of attribute mapping, the map $e'(val_s(a)) = s|_{f'(a)}$ is an encoding as long as for all $j \in f'(a)$ we have $s_j = s'_j$ if $val_a(s) = val_a(s')$ and e' is injective. The former follows from the assumption that $j \in \mathrm{comm}(a, S)$. The latter follows from the fact that extending the range of the encoding maintains the injectivity of it. Hence, $f'(a) \mapsto I_a$ is an attribute mapping. $\square$

Proof of Lemma 2. Let $a \in \mathbb{A}$ and let $s, s' \in S$, such that $\mathrm{val}_a(s) \neq \mathrm{val}_a(s')$. From the definition of an attribute mapping and injectivity of encoding functions, we derive that $s|_{f(a)} \neq s'|_{f(a)}$. Therefore, we can find $i \in f(a)$, such that $s_i \neq s'_i$, and thus $f(a)$ satisfies the definition of $\mathrm{diss}(a, S)$. $\square$

Proof of Lemma 3. We define $Q = \{p \in P \mid \forall p' \in P\colon\ p' \subseteq p \implies p' = p\}$ and prove that this is the required set. From the definition of Q it follows directly that Q is subset minimal.

The inclusion $\overline{Q} \subseteq \overline{P}$ follows directly from $Q \subseteq P$. For the converse, $\overline{P} \subseteq \overline{Q}$, we use the fact that strict set inclusion on $\mathcal{P}(F)$ is well-founded for finite F. Let $p \in \overline{P}$, then there exists $p' \in P$, such that $p' \subseteq p$. We consider two cases: $p' \in Q$ and $p' \notin Q$. If $p' \in Q$, then from $p' \subseteq p$ it follows that $p \in \overline{Q}$, as required. In the second case, $p' \notin Q$, we use the definition of Q to find $p'' \in P$ such that $p'' \subsetneq p'$. Again, we can consider two cases: $p'' \in Q$ and $p'' \notin Q$. In the first case, $p'' \in Q$ we have $p'' \subsetneq p' \subseteq p$, so $p \in \overline{Q}$, as required. In the second case we can repeat this construction to find $p''' \subsetneq p'' \subsetneq p' \subseteq p$. Given well-foundedness, it will be impossible to create an infinite sequence in this way. Therefore, there is a point where the loop will be broken by finding $p^{(k)} \in Q$, such that $p^{(k)} \subsetneq p^{(k-1)} \subsetneq \ldots \subsetneq p' \subseteq p$, which implies that $p \in \overline{Q}$.

Finally, we prove uniqueness. Assume that X and Y are two subset-minimal sets with $X \neq Y$ and $\overline{X} = \overline{P} = \overline{Y}$. Without loss of generality, we may assume that there exists $x \in X$, such that $x \notin Y$. We derive a contradiction and conclude $X = Y$ as follows. If $x \in X$, then $x \in \overline{Y}$. From $x \notin Y$, we find $y \in Y$, such that $y \subsetneq x$. From $y \in \overline{Y}$, it follows that $y \in \overline{X}$, so there exists $x' \in X$ with $x' \subseteq y$. Thus, we have $x' \subseteq y \subsetneq x$ for $x', x \in X$, which contradicts the assumption of subset minimality of X. $\square$

Proof of Lemma 4. By Lemma 3, let T be the unique subset-minimal set for which $\overline{T} = \mathrm{diss}(a, S)$. We show that $T \subseteq \mathrm{diss}(a, S')$.

Let $I \in T$. Then by definition, $\forall s, s' \in S\colon\ (\mathrm{val}_a(s) \neq \mathrm{val}_a(s') \implies \exists i \in I\colon\ s_i \neq s'_i)$. But since $S' \subseteq S$, the statement holds in particular for any two dumps in S'. Thus $I \in \mathrm{diss}(a, S')$. $\square$

Proof of Lemma 5. The inclusion $\mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S)) \subseteq \mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S))$ follows from Lemma 4.

For the reverse inclusion, let $I \in \mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S))$ be an index set in the filtration of $\mathrm{diss}(a, R)$ with respect to the common set of the attribute a of the dumps in S.

Suppose towards a contradiction that $I \notin \mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S))$. Then there must be dumps $s_1, s_2 \in S$ such that $s_1|_I = s_2|_I$, but $\mathrm{val}_a(s_1) \neq \mathrm{val}_a(s_2)$.

Consider representatives $r_1, r_2 \in R$ of s_1 and s_2 such that $\mathrm{val}_a(r_1) = \mathrm{val}_a(s_1) \neq \mathrm{val}_a(s_2) = \mathrm{val}_a(r_2)$. Since $I \subseteq \mathrm{comm}(a, S)$, it follows that $r_1|_I = s_1|_I$, $r_2|_I = s_2|_I$, but $\mathrm{val}_a(r_1) \neq \mathrm{val}_a(r_2)$. This contradicts $I \in \mathrm{diss}(a, R)$. $\square$

Proof of Theorem 2. By Lemma 5, it suffices to prove $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) = \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S)))$.

The inclusion $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) \subseteq \mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S))$ holds, since $\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)}) \subseteq \mathrm{diss}(a, R)$ and $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) \subseteq \mathrm{comm}(a, S)$.

The inclusion $\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)}) \supseteq \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S)))$ holds as follows. Let $I \in \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S)))$. Then $I \in \mathrm{diss}(a, R)$ and $I \subseteq \mathrm{comm}(a, S)$, thus $I \in \mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})$.

The Lemma now follows by uniqueness of subset minimal sets (Lemma 3) and the facts that the dissimilarity sets and filters of dissimilarity sets are superset closed. $\square$

Proof of Lemma 7. T is interval-subset-minimal by definition. It is obvious that $T \subseteq \mathrm{diss}(a, S) \cap \mathcal{I}_n$. Since $\mathrm{diss}(a, S) \cap \mathcal{I}_n$ is interval-superset closed, it follows that $\overline{T} \cap \mathcal{I}_n \subseteq \mathrm{diss}(a, S) \cap \mathcal{I}_n$. Furthermore, for all $i \in [0, n)$, if $\mathrm{iv}(a, S)(i)$ exists, then $\mathrm{iv}(a, S)(i) \in \overline{T} \cap \mathcal{I}_n$.

Suppose towards a contradiction that $\overline{T} \cap \mathcal{I}_n \subsetneq \mathrm{diss}(a, S) \cap \mathcal{I}_n$. Then there exists $I \in \mathrm{diss}(a, S) \cap \mathcal{I}_n$ such that $I \notin \overline{T} \cap \mathcal{I}_n$. Let $I = [i_0, i_1]$ and consider $\mathrm{iv}(a, S)(i_0)$. By definition of $\mathrm{iv}(a, S)$, we have $\mathrm{iv}(a, S)(i_0) \subseteq I$ and we know that $\mathrm{iv}(a, S)(i_0) \in \overline{T} \cap \mathcal{I}_n$. This contradicts $I \notin \overline{T} \cap \mathcal{I}_n$. $\square$

Proof of Theorem 3. We first prove correctness of the algorithm and then compute its time complexity.

Correctness. Let $k = \max_{j \in [1,|R|]}(k_j)]$. By Lemma 7, to prove correctness of the algorithm, we need to show that for any two dumps $s, s' \in R$ there exists an

index $ind \in [i, i+k]$ such that $s_{ind} \neq s'_{ind}$. We show this by iterating over the sorted list of dumps.

Dump $s^{(1)}$ differs from all other dumps within the interval $[i, k_1]$ because it differs from $s^{(2)}$ within this interval and the dump list is sorted. Assuming that for all $j < j_0$ dump $s^{(j)}$ differs from all other dumps within the interval $[i, max(k_1, \ldots, k_j)]$ we show that dump j_0 differs from all other dumps within the interval $[i, max(k_1, \ldots, k_{j_0})]$. First $s^{(j_0)}$ differs from $s^{(j)} < s^{(j_0)}$ on $[i, max(k_1, \ldots, k_{j_0})]$ since $[i, max(k_1, \ldots, k_j)] \subset [i, max(k_1, \ldots, k_{j_0})]$. The dumps $s^{(j)} > s^{(j_0)}$ differ from $s^{(j_0)}$ within the interval $[i, k_{j_0}]$ because $s^{(j_0)}$ differs from $s^{(j_0+1)}$ within this interval and the dump list is sorted. Thus the algorithm correctly computes $\mathrm{iv}(a, R)(i)$.

Complexity. The complexity of the algorithm is given by the complexity to sort the dump set and the complexity to compare adjacent dumps in the sorted list. The bit-complexity for comparing the adjacent dumps $s^{(j)}, s^{(j+1)}$ is k_j. Thus, in the worst case, it is bounded by n, the bit length of the dump. Thus $\mathrm{iv}(a, R)(i)$ can be computed in time $O((n-i)|R| + (n-i)|R| \log |R|) = O((n-i)|R| \log |R|)$.

If $\mathrm{iv}(a, R)(i)$ is computed for all $i \in [0, n)$, the sorting complexity for $i > 0$ can be lowered by taking advantage of the sorted list of dumps with respect to $>_{i-1}$. We merely need to perform a merge-sort for $<_i$ on two sets given by the restrictions $s_{i-1} = 0$ and $s_{i-1} = 1$ and ordered with respect to $<_{i-1}$. This can be performed in time $O((n-i)|R|)$. By summing up the time it takes to compute $\mathrm{iv}(a, R)(i)$ for $i \in [0, n)$ we obtain the theorem. $\square$