Tutorial for using DEC0DE

February 13, 2013

1 Introduction

The following tutorial gives a sense of how DEC0DE handles a phone's memory to infer different kinds of records from it. There is a sequence of steps through which the phone's memory information goes through before the final inferences are made. These are described below with the help of a testcase.

Additionally, the way the major public methods of DEC0DE's classes work together is also briefly described along the way to give the user/developer an idea where could the modifications/additions be made.

2 Stepwise Working of DEC0DE

2.1 Select memory file

Use the Open button on the main form for browsing the memory file of the phone whose records are to be inferred. This needs to be a binary file.

The interface also gives an option of specifying the make of the phone viz. Samsung, Nokia, Motorola etc. through a drop down list. Additionally, one can specify the model of the phone too. Click OK to submit your choices.

The `MainForm` class handles the buttons and the working behind this form. As the main form loads, the `MainForm_Load ( )` method of `MainForm` checks if there is a database of phone block hashes already existing. If it does not find it, it's created through a call to the static method, `CreateAndInitialize ( )` of the `DatabaseCreator` class.

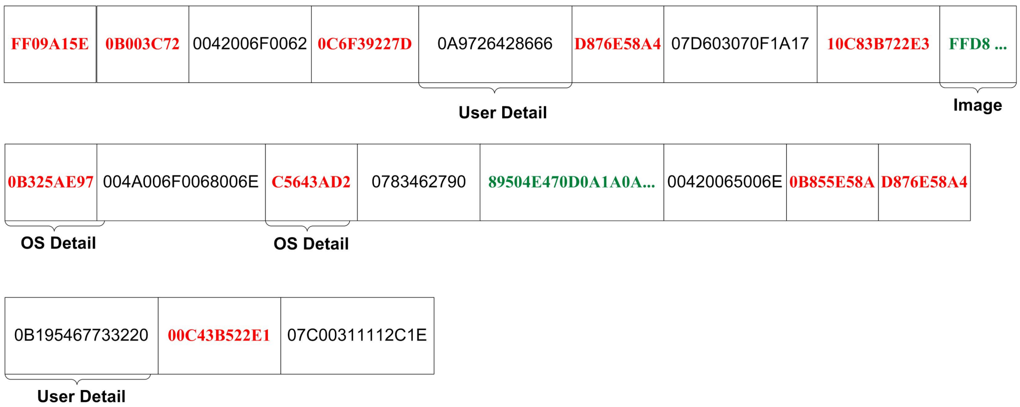
Clicking the open button calls the constructor of `GetMemFileDlg` class, which in turn initializes this form. The `buttonOK_Click ( )` method submits the memory file path, make/model/notes for the phone encapsulated in an object to the `MainForm` class's `toolStripButtonOpen_Click ( )` method, which initializes its own fields with this phone information.

2.2 Select your own filters

The Filters button on the main form opens a dialog box in which one can specify the maximum duration (in days) beyond which he won't like to have any call log, SMS record.

2.3 Begin the process of decoding

Let's assume that the memory file looks like the following sequence of bytes:



The blocks in red represent the phone's operating system or instrument specific details. The ones in green represent the image portions in the memory (PNG, JPG etc.). The blocks in black are the ones that would be relevant for the process of decoding since they correspond to some or the other user specific details.

This blockwise description is just to give a better understanding of this article. `DECODE` would not know of all these demarcations right now, but would eventually figure out.

The Decode button on the main form begins the actual procedure for decoding the memory file. It triggers the `StartWork ( )` method of `MainForm` class, which in turn initializes the fields of `WorkerThread` class with the phone's information and calls the `Start ( )` method of `WorkerThread`. The

`Start ( )` method spawns a new thread for executing the `Run ( )` method of `WorkerThread`. It is the `Run ( )` method which shall call several methods to sequentially go about the major steps in the process of decoding.

2.4 Calculating the file SHA1 hash

This is the first major step that the `Run ( )` method of `WorkerThread` takes. It calls the static method named `CalculateFileSha1 ( )` of the `DcUtils` class, to find the SHA1 hash of the entire file. This is done to uniquely identify the phone's records in the database later.

2.5 Locating image blocks

The image blocks in the memory are identified using the prior knowledge of the starting bytes and formats of PNG, JPG, GIF, BMP images. Identifying images is useful since it's relatively convenient to locate them on the memory using their known byte formats than running inference procedures for it. This effort pays off since it reduces the number of blocks input to the inference procedure and hence improves the system's speed.

The `Run ( )` method next, calls the static method, `LocateImages ( )` of the `ImageFiles` class, which in turn calls the `Process ( )` method of `ImageFiles` to locate blocks corresponding to images in the memory. `Process ( )` further uses methods like `FindPNG ( )`, `FindJPG ( )`, `FindGIF ( )` to get those memory blocks. `LocateImages ( )` returns an object of class `ImageFiles` that encapsulates information about the blocks containing images.

2.6 Loading hashes into the database

Next, the `Run ( )` method calls the static method, `LoadHashesIntoDB ( )` of the `HashLoader` class, which takes the memory file's SHA1 hash, path to the file and other phone related information to find the block hashes of the phone's memory and loads all this information in the database, for future retrieval during the block hash filtering part.

`LoadHashesIntoDB ( )` calls other methods of the `HashLoader` class which:

Calculate the block hashes of the phone's memory by calling the `Filter ( )` method of the `BlockHashFilter ( )` class.

Insert the phone's general information about its make/model/file SHA1 hash as well as all the computed block hashes into the database using static methods from the `DatabaseAccess` class like:

* `HashRunInsert ( )` : Adds a new row for the phone's general details and block hash filtering details such as block size, hash type, slide amount etc. in the database table, `tbl_HashRun`.

* `PhoneInsert ( )` : Inserts into the database table, `tblPhone` only the general details of the phone.

* `InsertHashes ( )` : Inserts into the database table, `tblHash` the newly found block hashes of the phone.

* `HashRunUpdate ( )` : Updates the database table, `tbl_HashRun`, by updating the time to hash and the number of blocks fields of the row to which new block hashes have been added.

`LoadHashesIntoDB ( )` returns to `Run ( )` of `WorkerThread` an object of type `HashLoader` which contans the unique identification number for the phone's records in the database called `PhoneId`.

2.7 Block Hash Filtering

Block hash filtering essentially removes those blocks from consideration which correspond to the phone's operating system and other native code, since these portions would not contain any useful information for triage. It boosts the performance of the system by pruning roughly 69% of the original blocks, with no effect on performance.

`RunBlockHashFilter ( )` of `WorkerThread` is the next method called by `Run ( )`. This method calls the `Filter ( )` method of the `BlockHashFilter` class. The `Filter ( )` method uses the method `HashGetUnfilteredBlocks2 ( )` of the `BlockHashFilter` class to get a list of those block hashes of this phone from the database which did not match the block hashes of other phones in the database.

Using the returned hashes, it determines their corresponding memory blocks. It does so by comparing the returned hashes with the hashes of the original blocks. The blocks whose hash matches with a returned hash are kept for further processing whereas the rest are discarded. `RunBlockHashFilter ( )` returns an object of the `FilterResult` class which contains the list of unfiltered blocks, which could be used as the input to the inference procedures.

This is how the phone's memory blocks would look like, once block hash filtering is completed:

FF09A15E	0042006F0062	0C6F39227D	0A9726428666	D876E58A4	07D603070F1A17	FFD8 ...	0B325AE97	004A006F0068006E

0783462790	89504E470D0A1A0A...	00420065006E	D876E58A4	0B195467733220	00C43B522E1	07C00311112C1E

Note that four memory blocks have been pruned.

2.8 Filtering out image blocks

After block hash filtering is complete, `Run ( )` calls the static method `FilterOutImages ( )` of the class `ImageFiles` for removing the image blocks that were located initially, using the methods `RemoveImageBlock ( )`, `FilterImageBlocks ( )` and `RemoveImages ( )` of `ImageFiles`. They handle different scenarios such as, when an image block is fully within a block hash filtered block or partially within it.

`FilterOutImages ( )` essentially modifies the `filterResult` variable which was passed to it, that stored the block hash filtered blocks to now contain those blocks which have no image block's bytes in them.

This is how the block hash filtered list of blocks would look once the image containing blocks are removed:

FF09A15E	0042006F0062	0C6F39227D	0A9726428666	D876E58A4	07D603070F1A17	0B325AE97	004A006F0068006E

0783462790	00420065006E	D876E58A4	0B195467733220	00C43B522E1	07C00311112C1E

Note that the memory blocks in green have been pruned.

2.9 Field level inference

The blocks that now remain are the ones which are worth decoding, since they would have user centric information viz. call logs, SMS, address book entries with a high probability. Inference in `DECODE` is a two step process, both of which use the Viterbi's algorithm. This is the first step, field level inference used to recover information like phone numbers, timestamps, texts etc. The states in each state machine meant for field level inference emit a single byte before making a transiton to the next state.

To initiate this stage, `Run ( )` calls the `RunViterbi ( )` method of `WorkerThread`. The `RunViterbi ( )` method calls the `Run ( )` method of the `AnchorViterbi` class, with a run type setting of `GeneralParse`, which essentially refers to field level Viterbi inference. The `Run ( )` method uses the `GetAnchorPointBlocks ( )` of `AnchorViterbi` to infer the blocks representing any kind of phone number.

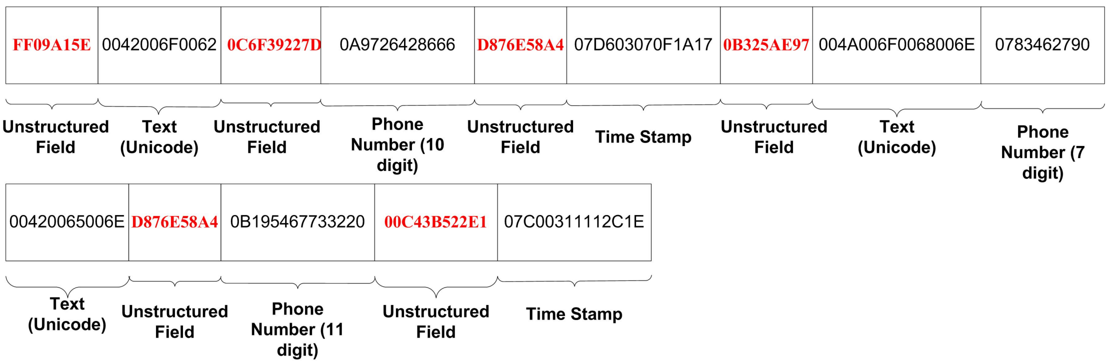
`GetAnchorPointBlocks ( )` calls the constructor of the class `Viterbi` with an `AnchorPoints` runtype setting, which is basically meant to trigger the method `TestAnchorFieldsOnly ( )` in the `StateMachine` class. It initializes an aggregated state machine with any possible format of phone number state machines viz. Nokia, Motorola, Samsung, eleven digit, ten digit, international digit formats. Each state machine in the aggregated HMM, in turn has a set of states, with each state storing a list of possible incoming/outgoing transitions to other states, the transition probability and the emission probabilities for all bytes given that state.

Once the `StateMachine` for all phone numbers is ready, `GetAnchorPointBlocks ( )` calls the `Run ( )` method of `Viterbi` to infer what all blocks correspond to a phone number. A list of these blocks and others lying between two phone number blocks which are sufficiently apart, are returned to the `Run ( )` method of `AnchorViterbi`.

Passing these blocks as arguments, `Run ( )` calls the `Run ( )` method of `Viterbi`, with a run type setting of `GeneralParse`. The constructor of `Viterbi` calls the static method named `GeneralParse ( )` of the class `StateMachine`. This method aggregates the major state machines of each category such as those corresponding to text, any possible phone number format, any possible timestamp format etc. There are several other methods in the `StateMachine` class that aggregate even more specialized state machines like Nokia's seven digit format, international format, Samsung's ten digit format into a generalized state machine, representing all of them. Similarly, there are methods for text and timestamp also.

The `Run ( )` method of the `Viterbi` class uses the Viterbi algorithm to determine the most likely sequence of states in this aggregated HMM and returns the inferred fields to the `Run ( )` method of the `AnchorViterbi` class, followed by returning the results to the `Run ( )` method of the `WorkerThread` class. The inferred fields are returned in the form of an object of the `ViterbiResult` class, that stores a list of objects of the class `ViterbiField`. An object of `ViterbiField` stores the beginning of the field in the memory file, its position in the inferred Viterbi path, its raw bytes, its ASCII and hexadecimal equivalent, apart from its human readable format, which was generated using different methods of the `Printer ( )` class, depending on the field type.

This is how the fields would be demarcated once field level inference is complete:



Following is what the inferred fields actually mean:

* **0042006F0062** : **Bob** (State Machine: **Text/Unicode**)
* **0A9726428666** : **972-642-8666** (State Machine: **Nokia Ten digit phone number**)
* **07D603070F1A17** : **03/07/2006 3:26:23 PM** (State Machine: **Timestamp Nokia**)
* **004A006F0068006E** : **John** (State Machine: **Text/Unicode**)
* **0783462790** : **834-6279** (State Machine: **Nokia Seven digit phone number**)
* **00420065006E** : **Ben** (State Machine: **Text/Unicode**)
* **0B195467733220** : **1-954-677-3322** (State Machine: **Nokia Eleven digit phone number**)

∗ **07C00311112C1E** : **03/17/1980 5:44:30 PM** (State Machine: **Timestamp Nokia**)

∗ Blocks in red are the outputs which could not be explained by any state machine, but the unstructured field state machine. These blocks are of arbitrary length and do not give any useful information about the phone's user.

2.10 Meta Info Creation

The above field level inference results have to be input to the record level Viterbi inference component, but before that, the meta machine for each field needs to be determined, since the state machines for record level inference are based on broader/meta categories. For example, the Nokia, Samsung, Motorola Ten digits phone number's state machine, Seven digit phone numbers state machine or Eleven digit phone number's state machine, all come under the meta state machine: `PhoneNumber`. Similarly, different kinds of time stamps viz. a Nokia Time stamp, a UNIX time stamp, a Motorola timestamp, all fall under the meta state machine: `TimeStamp`.

In context of program execution, `Run ( )` of `WorkerThread` calls the `RunMetaViterbi ( )` method with the field level inference results passes to it as arguments. `RunMetaViterbi ( )` calls the `CreateMetaInfo ( )` method of `WorkerThread` for determining the meta machines corresponding to the state machines of the inferred fields. `CreateMetaInfo ( )` returns a list of of objects belonging to the class `MetaResult`, each of which stores the field's information in the form of a `ViterbiField` object and the meta machine to which its state machine belongs to.

Following is how the results would like, once the meta info for each field has been created:

0042006F0062	0A9726428666	07D603070F1A17	004A006F0068006E	0783462790	00420065006E	0B195467733220	07C00311112C1E
Text	**Phone Number**	**Time Stamp**	**Text**	**Phone Number**	**Text**	**Phone Number**	**Time Stamp**

2.11 Record level inference

Next, the `RunMetaViterbi ( )` method calls the `Run ( )` method of the `Viterbi` class with the run type (as `Meta`, indicating `Run ( )` to start a record level Viterbi inference) and the results from `CreateMetaInfo ( )`

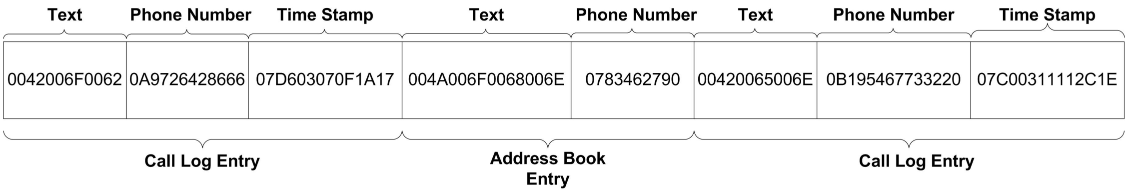
method as arguments to start record level inference.

In record level inference the state machines are designed such that each state in it emits a field (like a phone number, timestamp) before making a transition to the next state. This is different from field level Viterbi inference where each state in the state machine used to emit a single byte before making a transition.

The constructor of the `Viterbi` class calls the `TestMeta ( )` method of the class `StateMachine` which uses the `TestMetaStateMachines ( )` method of `StateMachine` to aggregate several record level state machines (those corresponding to an address book entry, call log record or an SMS) into one HMM. When the `Run ( )` method of `Viterbi` is invoked, it begins executing the Viterbi algorithm and returns the most likely sequence of states in the aggregated HMM as a `ViterbiResult` type to `RunMetaViterbi ( )` of `WorkerThread`.

`RunMetaViterbi ( )` uses `InterpretResults ( )` method of `WorkerThread` for identifying the category of each inferred record and for demarcating the fields stored within it. `InterpretResults ( )` uses the name of the state machine of each record to identify if it is a call log record, SMS or address book record. Once the record has been identified, it calls methods from `WorkerThread` like `GetMetaAddressBookEntry ( )`, `GetMetaCallLog ( )` or `GetMetaSms ( )` to store each record with its fields in a more organized fashion. These methods segregate the fields of a record, for example for an address book record, `GetMetaAddressBookEntry ( )` would extract and store fields like contact's name, phone number, last seven digits, starting position in file etc. into separate fields of an object of the class `MetaAddressBookEntry`. Similarly, for call log records and SMS records, the methods `GetMetaCallLog ( )` and `GetMetaSms ( )` use objects of classes `MetaCallLog` and `MetaSms` respectively for storing their fields. All these records are then stored in lists of objects of class `MetaField` (which is an abstract class extended by `GetMetaSms`, `GetMetaCallLog`, `GetMetaAddressBookEntry`), to be used further on by the `Run ( )` method of `WorkerThread`.

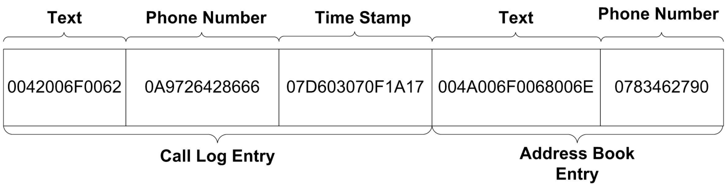
Following is the way blocks would be inferred after record level Viterbi run:



2.12 Post Processing

To get rid of the false positive records that might have been inferred in the above stages, the post-processing component of `DECODE` is used. The lists of records stored using the `MetaField` objects are then passed onto the `PerformPostProcessing ( )` method of `WorkerThread`. It calls the `Process ( )` method of the `PostProcessor` class. The `Process ( )` method uses several other methods of the `Process` class to eliminate duplicate records of any kind, calculating the proximity of a record from the nearest record on the memory file to remove records lying isolated in the memory, verifying the year portion in a timestamp to see whether or not it lies in a plausible range etc.

For instance, the last call log entrys timestamp specifies the year 1980. Its clear that although it follows the correct timestamp format, the record is too old to be considered as a valid call log record. Hence, the record is pruned by the Post Processing component of `DECODE`. This is how the records would look like, after the post processing stage:



Once the post processing is complete, the pruned lists of record entries are returned to the `Run ( )` method of `WorkerThread`. `Run ( )`, now calls the `EndWork ( )` method of the `MainForm` class, passing the record lists and a boolean flag to indicate whether or not the entire procedure ended successfully, as arguments. The `EndWork ( )` method populates labels on the `MainForm` with the number of recovered address book, call log and SMS records.

Hence, the result of running `DECODE` on our test case recovers the following two records:

∗ **Call Log Entry** :
  Name: **Bob**
  Phone Number: **9726428666**
  Time Stamp: **03/07/2006 3:26:23 PM**

∗ **Address Book Entry** :
  Name: **John**
  Phone Number: **834-6279**