available at www.sciencedirect.com

journal homepage: www.elsevier.com/locate/diin

Forensic analysis of the Firefox 3 Internet history and recovery of deleted SQLite records

Murilo Tito Pereira[*]

Brazilian Federal Police, Computer Forensics, R. Laudelino Coelho, 55, 60415-430 Fortaleza, CE, Brazil

ARTICLE INFO

Article history:
Received 10 October 2008
Received in revised form
19 January 2009
Accepted 20 January 2009

Keywords:
Firefox 3
Internet history
SQLite
Browser forensics
Recovery of deleted records

ABSTRACT

Firefox 3 uses a new system, based on various SQLite databases, to store Internet history, bookmarks, form field data and cookies. This paper presents the main properties of these databases, what fields are of forensic interest and what information the available tools can extract. It shows that even if records in these databases are deleted, remnants may be found in unallocated disk space, due to the fact that SQLite utilizes temporary transaction files. The paper proposes an algorithm to recover deleted SQLite entries based on known internal record structures. The tool developed can recover deleted history records and the methodology applied in this work can be used with other applications that also employ SQLite databases.

1. Introduction

Firefox 3 is the new version of Mozilla browser, released in the middle of June, 2008. According to Mozilla Foundation, it set a new Guinness World Record with 8.002.530 downloads within the first 24 hours (Mozilla Foundation, 2008). Given its popularity, forensic examiners can expect to encounter this browser and the associated artifacts in future cases. From a forensic perspective, the most important feature is the use of a completely new system to store Internet history, bookmarks, form field data and cookies. This information has always been a good source of evidence for forensic investigators and this new system can provide more elements than those provided by version 2. Jones and Belani (2008) provided a study on Web Browser Forensics, including Firefox 2 history and cache system.

This paper describes this new history system and proposes an algorithm to recover deleted history records. When this paper was written, there were no released studies known to the author that addressed the forensic aspects of this implementation of SQLite utilized for the new history system. Utilities are already available to interpret some of the artifacts generated by Firefox 3, like Firefox Forensics (http://www.machor-software.com), FoxAnalysis (http://www.forensic-software.co.uk) and Firefox 3 Extrator (http://www.firefoxforensics.com). Their main features and scope will be addressed in Section 2.5. However, it is important for forensic examiners to understand how the underlying databases are structured and what additional information can be obtained. The cache system is not a focus of this study, since Firefox 3 kept the same cache model from version 2.

The paper is organized as follows: Section 2 shows how the new Firefox 3 history system works, the main databases used and the relevant information to an investigation. Section 3 details the internal structure of database records used by

[*] Tel.: +55 85 33924963; fax: +55 85 33924873.
E-mail address: murilo.mtp@dpf.gov.br

1742-2876/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2009.01.003

Firefox 3 and how this information can be recovered from unallocated space. An algorithm to recover these records and the testing results are described in Section 4, and Section 5 presents the conclusion of this study.

2. Analysis of Firefox 3 history system

This section presents the new Firefox 3 history system, along with discussion of the files that compose the databases and their main characteristics.

This new system, called *Places*, is completely different from version 2, and is built using SQLite, a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine (SQLite, 2008a). To the user, a significant change implemented in this new system is that all the URL content is indexed. As such, the user can type any part of the URL and the autocomplete function will show all URLs that match, unlike Firefox 2 that only show the matches comparing the beginning of the URL.

Firefox 3 uses several SQLite databases, located in the user profile folder, which the default path is *C:\Documents and Settings\<user>\Application Data\Mozilla\Firefox\Profiles\<profile folder>* for Windows XP and *C:\Users\<user>\AppData\Roaming\Mozilla\Firefox\Profiles\<profile folder>* for Windows Vista. Each database is a single file, with the ".sqlite" extension and those of greatest forensic value are explained below. SQLite Manager program (http://code.google.com/p/sqlite-manager) or SQLite command line tool (http://www.sqlite.org/download.html) can be used to query SQLite databases.

2.1. **Places.sqlite** *database*

The *places.sqlite* database is responsible for storing URLs accessed and bookmarks. Many tables compose it and the most relevant ones are described below.

2.1.1. Moz_places *table*

The *moz_places* is the central table in *places.sqlite* database, since it stores all the URLs. The *moz_places* table contains the following fields:

- **id**: The table's primary key.
- **url**: Stores the URL. It can also store queries, as, for example, "place:queryType = 0&sort = 8&maxResults = 10", that returns the 10 most visited URLs.
- **title**: Stores the page's title.
- **rev_host**: Stores the reverse host name.
- **visit_count**: Stores the total visit count.
- **hidden**: Indicates if the URL will be displayed by the autocomplete function. A value of 1 keeps it hidden. Examples of hidden URLs are queries and RSS feeds.
- **typed**: Indicates if the URL has been typed (value 1) or not (value 0).
- **favicon_id**: Relates to the favicon's ID on *moz_favicons* table.
- **frecency**: The word "frecency" is a combination of the words "frequency" and "recency". **Frecency** is a score given to each unique URL, encompassing bookmarks, history and tags. This score is determined by the amount of revisitation, the type of those visits, how recent they were, and whether the URL was bookmarked or tagged (Mozilla Developer Center, 2008c). This value is used by the autocomplete function to show the highest scores first on the list.

Some elements of forensic interest in the *moz_places* table include:

1. It stores the history for a default period of 90 days. In Firefox 2 this period is 9 days and in Internet Explorer 20 days.
2. If the **typed** field has a value 1, it indicates the user manually entered the URL.
3. The **frecency** field can be important, together with **visit_count** and **typed**, to determine the relation between the user and a URL.
4. The values of **typed** and **frecency** are not available to the user without direct access to the database.

2.1.2. Moz_bookmarks *table*

To store bookmarks, *Places* use the *moz_bookmarks* table. The table fields are:

- **id**: The table's primary key.
- **type**: Bookmark type: 1 for URL, 2 for folder and 3 for a separator.
- **fk**: A foreign key, related to *moz_places* table. The *moz_bookmarks* does not store the URL, which is stored in *moz_places* table.
- **parent:** Used together with **position** to determine the organization of bookmarks. **parent** indicates the parent folder **id** of this record.
- **position:** Indicates the position inside the folder.
- **title**: Stores the page's title.
- **keyword_id**: Firefox allows the user to associate a keyword to a bookmark. This field relates to the **id** in *moz_keywords* table.
- **folder_type**: Identifies if the record is a folder or not.
- **dateAdded**: Stores the date and time when the record has been added in PRTime format (Mozilla Developer Center, 2008b).
- **lastModified**: Stores the date and time when the record has been last modified in PRTime format.

The important elements in this table are **dateAdded** and **lastModified**. By default, these fields are not shown in the bookmarks interface, but this can be changed by the user.

2.1.3. Moz_historyvisits *table*

Another important table is the *moz_historyvisits*, which stores the visit date, the visit type and a pointer to the URL of the previous web page from which a link was followed. The fields are:

- **id**: The table's primary key.
- **place_id**: Relates to the **id** in *moz_places* table.
- **from_visit**: Stores a pointer to the URL of the previous web page from which a link was followed to the current page, similarly to the "Referer" field in HTTP request headers. An example on how to decode this field is showed below.

Table 1 – Visit type values.

Value	Name	Description
1	TRANSITION_LINK	This transition type means the user followed a link and got a new toplevel window.
2	TRANSITION_TYPED	This transition type means the user typed the page's URL or selected it from URL bar autocomplete results.
3	TRANSITION_BOOKMARK	This transition is set when the user followed a bookmark to get to the page.
4	TRANSITION_EMBED	This transition type is set when some inner content is loaded. This is true of all images on a page, and the contents of the iframe. It is also true of any content in a frame, regardless if whether or not the user clicked something to get there.
5	TRANSITION_REDIRECT_ PERMANENT	This transition is set when the transition was a permanent redirect.
6	TRANSITION_REDIRECT_ TEMPORARY	This transition is set when the transition was a temporary redirect.
7	TRANSITION_DOWNLOAD	This transition is set when the transition is a download.

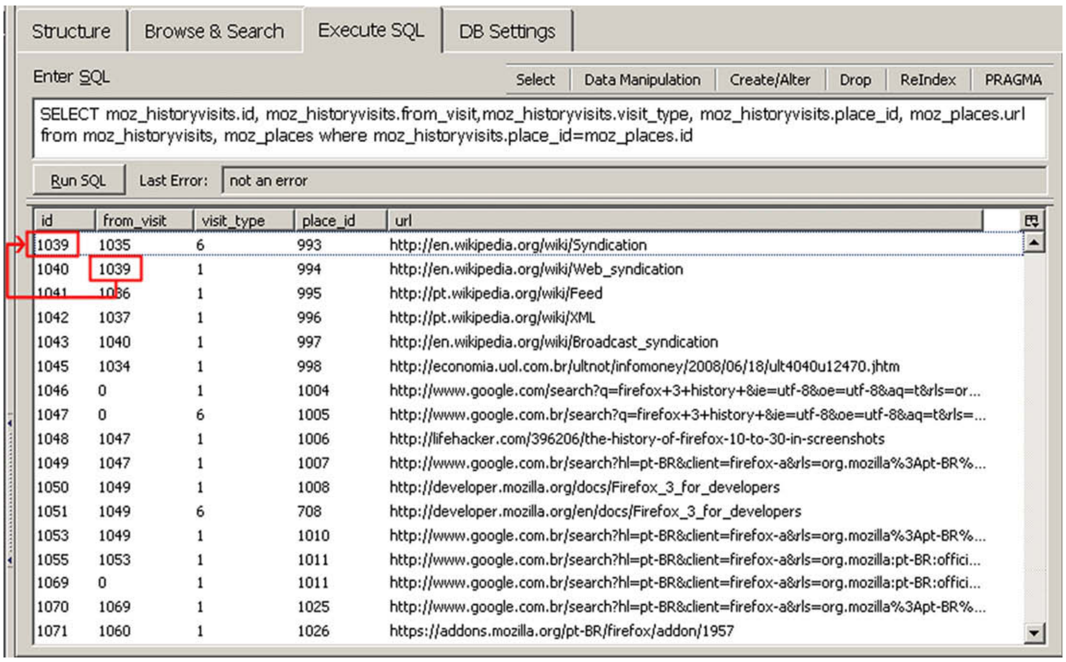
- **visit_date**: Presents the date and time when the URL has been accessed in PRTime format.
- **visit_type**: Identifies how the URL has been accessed. It has 7 possible values, listed in Table 1 (Mozilla Developer Center, 2008a).
- **session**: According to Mozilla documentation (Mozilla Developer Center, 2008a), it stores "the session ID that this page belongs to". The way the session ID is assigned was not determined.

The field **from_visit** relates to the **id** number for a record in table *moz_historyvisits*, or 0 when there is not a referring page, for example, when the page is accessed from bookmarks. Fig. 1 shows an excerpt of a *moz_historyvisits* table, opened with SQLite Manager, where the following query was executed:

"*SELECT moz_historyvisits.id, moz_historyvisits.from_visit, moz_historyvisits.visit_type, moz_historyvisits.place_id, moz_places.url from moz_historyvisits, moz_places where moz_historyvisits.place_id = moz_places.id*"

This query selects the main fields from *moz_historyvisits* plus the **url** field from *moz_places*. As an example, when decoding the **from_visit** field for the second line, it is possible to conclude that the user accessed the URL http://en.wikipedia.org/wiki/Syndication and followed a link in this page to the current page (http://en.wikipedia.org/wiki/Web_syndication). This decoding process is explained below:

- The **place_id** field in second line points to the record 994 in the *moz_places* table, which contains the **url** value "http://en.wikipedia.org/wiki/Web_syndication", indicating the current page;
- The field **from_visit** refers to the *moz_hystoryvisits* itself, pointing to the entry containing the URL from which the user followed a link to the current page. On this example, the **from_visit** points to record 1039;



```
SELECT moz_historyvisits.id, moz_historyvisits.from_visit,moz_historyvisits.visit_type, moz_historyvisits.place_id, moz_places.url
from moz_historyvisits, moz_places where moz_historyvisits.place_id=moz_places.id
```

id	from_visit	visit_type	place_id	url
1039	1035	6	993	http://en.wikipedia.org/wiki/Syndication
1040	1039	1	994	http://en.wikipedia.org/wiki/Web_syndication
1041	1036	1	995	http://pt.wikipedia.org/wiki/Feed
1042	1037	1	996	http://pt.wikipedia.org/wiki/XML
1043	1040	1	997	http://en.wikipedia.org/wiki/Broadcast_syndication
1045	1034	1	998	http://economia.uol.com.br/ultnot/infomoney/2008/06/18/ult4040u12470.jhtm
1046	0	1	1004	http://www.google.com/search?q=firefox+3+history+&ie=utf-8&oe=utf-8&aq=t&rls=or...
1047	0	6	1005	http://www.google.com.br/search?q=firefox+3+history+&ie=utf-8&oe=utf-8&aq=t&rls=...
1048	1047	1	1006	http://lifehacker.com/396206/the-history-of-firefox-10-to-30-in-screenshots
1049	1047	1	1007	http://www.google.com.br/search?hl=pt-BR&client=firefox-a&rls=org.mozilla%3Apt-BR%...
1050	1049	1	1008	http://developer.mozilla.org/docs/Firefox_3_for_developers
1051	1049	6	708	http://developer.mozilla.org/en/docs/Firefox_3_for_developers
1053	1049	1	1010	http://www.google.com.br/search?hl=pt-BR&client=firefox-a&rls=org.mozilla%3Apt-BR%...
1055	1053	1	1011	http://www.google.com.br/search?hl=pt-BR&client=firefox-a&rls=org.mozilla:pt-BR:offici...
1069	0	1	1011	http://www.google.com.br/search?hl=pt-BR&client=firefox-a&rls=org.mozilla:pt-BR:offici...
1070	1069	1	1025	http://www.google.com.br/search?hl=pt-BR&client=firefox-a&rls=org.mozilla%3Apt-BR%...
1071	1060	1	1026	https://addons.mozilla.org/pt-BR/firefox/addon/1957

Fig. 1 – Example of moz_historyvisits table, opened with SQLite Manager.

- On record 1039 (first line), the **place_id** points to the URL “http: //en.wikipedia.org/wiki/Syndication”, which is the previous page;
- Hence, the URL referred by **place_id** 994 (“http: //en.wikipedia.org/wiki/Web_syndication”) has been followed from the web page indicated in **place_id** 993 (“http: //en.wikipedia.org/wiki/Syndication”).

The information from **visit_type** field is much more complete than from **typed** field (*moz_places* table). It indicates exactly how the URL has been accessed, and can be important to determine the user’s behavior.

Other potentially useful information in the *moz_historyvisits* table is that when the same URL is accessed more than once, Firefox registers all of them, creating one new record for each access. These records have the same **place_id** value, pointing to the same URL, as, for example, the records with **id** 1055 and 1069 in Fig. 1. Since *moz_historyvisits* table also records the date and time for each visit (**visit_date** field), it is possible to determine the complete history access of each URL.

The information presented by **from_visit** and **visit_type** fields as well as the complete history access of each URL are not directly available through the standard browser interface.

2.2. Formhistory.sqlite *database*

Firefox also stores other data of investigative interest in addition to history and bookmarks. The *formhistory.sqlite* database stores the values entered by the user in form fields on web pages.

This database has just one table, called *moz_formhistory*, with three fields: **id**, **fieldname** and **value**. The field **id** is the primary key, **fieldname** is the name of the field in the form and **value** is the value entered by the user.

While each record does not have a date and time field, the **id** sequence can help determine the order in which the values were entered. Additionally, there does not appear to be any limit for the number of values that can be stored.

One value of **fieldname** that may appear frequently is “searchbar-history”. This value represents the Firefox search bar, but it does not indicate which search engine was used. Other interesting commonly found data are user names from web servers, e-mail recipient addresses, credit card numbers, and so on, as shown in Fig. 2.

2.3. Downloads.sqlite *database*

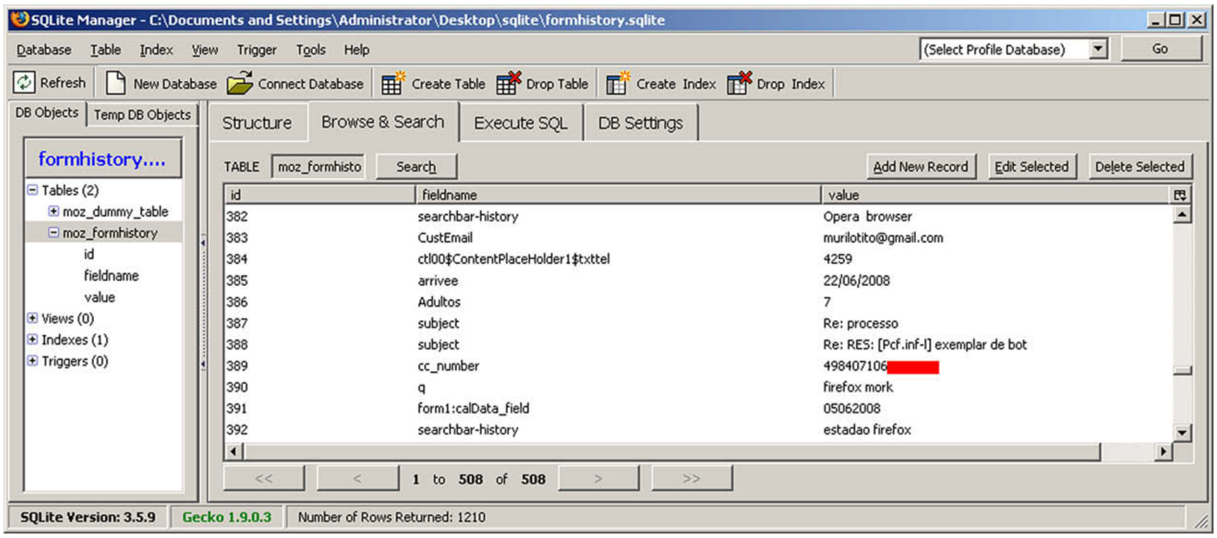
The information documenting downloads is detailed in a separate database, called *downloads.sqlite*. Among other information, it also stores data necessary to implement the download pause and restart functionality.

The *downloads.sqlite* database has only one table, called *moz_downloads*. Significant fields include:

- **id**: The table’s primary key.
- **name**: Name of the file being downloaded.
- **source**: Download source URL.
- **target**: The download destination (full path).
- **startTime** and **endTime**: Start and end date/time of the download, respectively, in PRTime format.
- **state**: Indicates the download state: completed, paused or cancelled.
- **referrer**: The web page’s URL containing the link that the user followed to the current download.
- **currBytes**: Currently downloaded bytes.
- **maxBytes**: The total bytes to be downloaded.

Much of this information may be relevant in an investigation, mainly the **source**, **target**, **referrer**, **startTime** and **endTime**. The **referrer** field is potentially a relevant and important piece of information that is not available to the user through the standard browser interface.

Despite this database being independent of *places.sqlite*, it is possible, in some cases, to associate the downloaded file name to the **url** field in *places.sqlite*, establishing a relation between the two tables.



id	fieldname	value
382	searchbar-history	Opera browser
383	CustEmail	murilotito@gmail.com
384	ctl00$ContentPlaceHolder1$txttel	4259
385	arrivee	22/06/2008
386	Adultos	7
387	subject	Re: processo
388	subject	Re: RES: [Pcf.inf-l] exemplar de bot
389	cc_number	498407106[redacted]
390	q	firefox mork
391	form1:calData_field	05062008
392	searchbar-history	estadao firefox

Number of Rows Returned: 1210

Fig. 2 – Example of moz_formhistory table, opened with SQLite Manager.

2.4. *Cookies.sqlite database*

In Firefox 3, cookies are stored in a database called *cookies.sqlite*. This database, with one table named *moz_cookies*, stores the traditional cookie data, as shown in Fig. 3. The *moz_cookies* fields have self-explanatory names, including **id**, **name**, **value**, **host**, **path**, **expiry**, **lastAccessed**, **isSecure** and **isHttpOnly**. Differently from other tables, in *moz_cookies* the **id** represents the date and time the cookie was created in PRTime format.

2.5. *Available utilities*

There are currently few utilities to extract data from Firefox 3 SQLite databases, such as: FoxAnalysis (http://www.forensic-software.co.uk), Firefox Forensics (http://www.machor-software.com) and Firefox 3 Extractor (http://www.firefoxforensics.com).

FoxAnalysis processes the four databases presented previously and shows the content in five tabs: Website History, Bookmarks, Cookies, Downloads and Form History. Fig. 4 shows FoxAnalysis interface.

In the ''Website History'' tab, it presents the fields from *moz_places* plus **visit_type** and **visit_date** from *moz_historyvisits* table. As discussed in Section 2.1.3, when the same URL is accessed more than once, Firefox registers all accesses. An example of this behavior is shown in Fig. 4, where the same URL (lines 3–6) is listed four times with different visit dates. FoxAnalysis does not process the information contained in **from_visit** field.

In the other tabs, FoxAnalysis presents the field contents extracted from the corresponding tables (*moz_bookmarks, moz_cookies, moz_downloads* and *moz_formhistory*), decoding date and time and download status. It does not extract all fields and hides the ones with no explicit forensic value, which, in some cases, can be a drawback. For instance, it omits the **id** from *moz_formhistory*, that, as said, can help determine the order in which the values were entered.

Firefox Forensics processes most of Firefox 3 SQLite databases, extracts the field contents and decodes date and time, in a similar way of FoxAnalysis (see Fig. 5). In the ''History'' tab, it presents the fields from *moz_places*, including **hidden** and **typed**, plus **visit_date** from *moz_historyvisits* table, but it does not include the **visit_type** field. As FoxAnalysis, Firefox Forensics does not extract the information contained in **from_visit** field.

Firefox 3 Extractor is a command line utility that extracts field contents from all tables of all SQLite databases used in Firefox 3, decoding date and time. Fig. 6 shows the main menu of Firefox 3 Extractor.

The three utilities can export the data to HTML or CSV format. The main shortcoming in these utilities is the inability to process the **from_visit** field.

3. Recovering deleted SQLite records

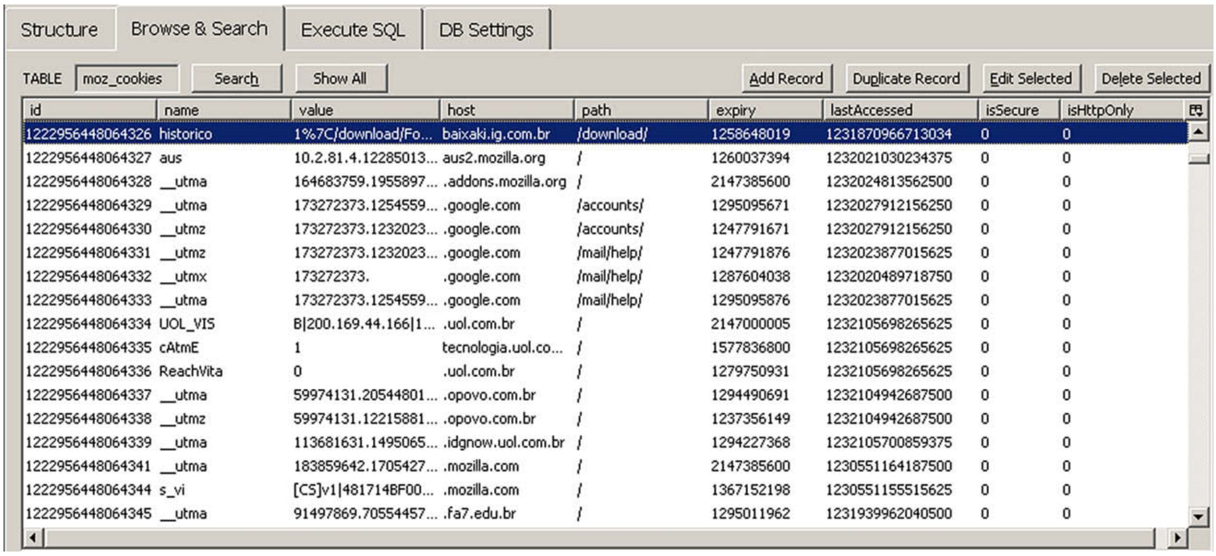
Users commonly want to delete usage logs to protect their privacy. Firefox has an option to clear user's private data, as shown in Fig. 7 in its default configuration. The user can clear his private data whenever he wants or he can configure Firefox to clear the data every time the browser closes. In this last configuration, the analysis of database files would be fruitless, unless there is a way to carve the records from free space.

Additionally, Firefox stores the history for a default period of 90 days and then starts to delete old history records. From a forensic point of view, recovery of the database records is very important whether they were deleted through normal use of the browser or intentionally deleted by the user.

To understand how to recover SQLite records from unallocated space, this section presents the manner in which Firefox/SQLite deletes records, shows why database fragments can be found on disk and describes the internal structure of an SQLite database.

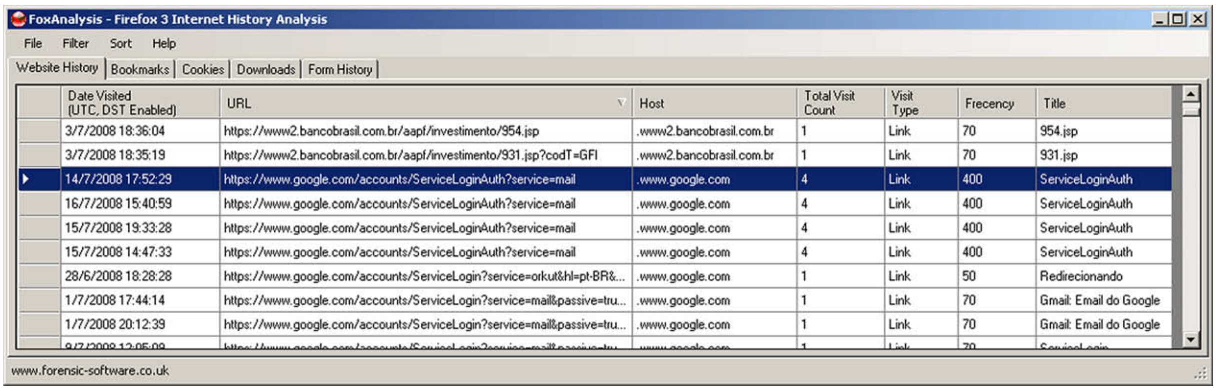
3.1. *How Firefox 3 deletes history records*

This section details results of research and testing to determine whether deleted Firefox 3 records can be recovered from a hard disk. The tests performed consisted of clearing the



id	name	value	host	path	expiry	lastAccessed	isSecure	isHttpOnly
1222956448064326	historico	1%7C/download/Fo...	baixaki.ig.com.br	/download/	1258648019	1231870966713034	0	0
1222956448064327	aus	10.2.81.4.12285013...	aus2.mozilla.org	/	1260037394	1232021030234375	0	0
1222956448064328	__utma	164683759.1955897...	.addons.mozilla.org	/	2147385600	1232024813562500	0	0
1222956448064329	__utma	173272373.1254559...	.google.com	/accounts/	1295095671	1232027912156250	0	0
1222956448064330	__utmz	173272373.1232023...	.google.com	/accounts/	1247791671	1232027912156250	0	0
1222956448064331	__utmz	173272373.1232023...	.google.com	/mail/help/	1247791876	1232023877015625	0	0
1222956448064332	__utmx	173272373.	.google.com	/mail/help/	1287604038	1232020489718750	0	0
1222956448064333	__utma	173272373.1254559...	.google.com	/mail/help/	1295095876	1232023877015625	0	0
1222956448064334	UOL_VIS	B|200.169.44.166|1...	.uol.com.br	/	2147000005	1232105698265625	0	0
1222956448064335	cAtmE	1	tecnologia.uol.co...	/	1577836800	1232105698265625	0	0
1222956448064336	ReachVita	0	.uol.com.br	/	1279750931	1232105698265625	0	0
1222956448064337	__utma	59974131.20544801...	.opovo.com.br	/	1294490691	1232104942687500	0	0
1222956448064338	__utmz	59974131.12215881...	.opovo.com.br	/	1237356149	1232104942687500	0	0
1222956448064339	__utma	113681631.1495065...	.idgnow.uol.com.br	/	1294227368	1232105700859375	0	0
1222956448064341	__utma	183859642.1705427...	.mozilla.com	/	2147385600	1230551164187500	0	0
1222956448064344	s_vi	[CS]v1|481714BF00...	.mozilla.com	/	1367152198	1230551155515625	0	0
1222956448064345	__utma	91497869.70554457...	.fa7.edu.br	/	1295011962	1231939962040500	0	0

Fig. 3 – Example of moz_cookies table, opened with SQLite Manager.



FoxAnalysis - Firefox 3 Internet History Analysis

File Filter Sort Help

Website History | Bookmarks | Cookies | Downloads | Form History

Date Visited (UTC, DST Enabled)	URL	Host	Total Visit Count	Visit Type	Frecency	Title
3/7/2008 18:36:04	https://www2.bancobrasil.com.br/aapf/investimento/954.jsp	.www2.bancobrasil.com.br	1	Link	70	954.jsp
3/7/2008 18:35:19	https://www2.bancobrasil.com.br/aapf/investimento/931.jsp?codT=GFI	.www2.bancobrasil.com.br	1	Link	70	931.jsp
14/7/2008 17:52:29	https://www.google.com/accounts/ServiceLoginAuth?service=mail	.www.google.com	4	Link	400	ServiceLoginAuth
16/7/2008 15:40:59	https://www.google.com/accounts/ServiceLoginAuth?service=mail	.www.google.com	4	Link	400	ServiceLoginAuth
15/7/2008 19:33:28	https://www.google.com/accounts/ServiceLoginAuth?service=mail	.www.google.com	4	Link	400	ServiceLoginAuth
15/7/2008 14:47:33	https://www.google.com/accounts/ServiceLoginAuth?service=mail	.www.google.com	4	Link	400	ServiceLoginAuth
28/6/2008 18:28:28	https://www.google.com/accounts/ServiceLogin?service=orkut&hl=pt-BR&...	.www.google.com	1	Link	50	Redirecionando
1/7/2008 17:44:14	https://www.google.com/accounts/ServiceLogin?service=mail&passive=tru...	.www.google.com	1	Link	70	Gmail: Email do Google
1/7/2008 20:12:39	https://www.google.com/accounts/ServiceLogin?service=mail&passive=tru...	.www.google.com	1	Link	70	Gmail: Email do Google

www.forensic-software.co.uk

Fig. 4 – FoxAnalysis interface.

private data and then searching for known deleted strings. It was determined that:

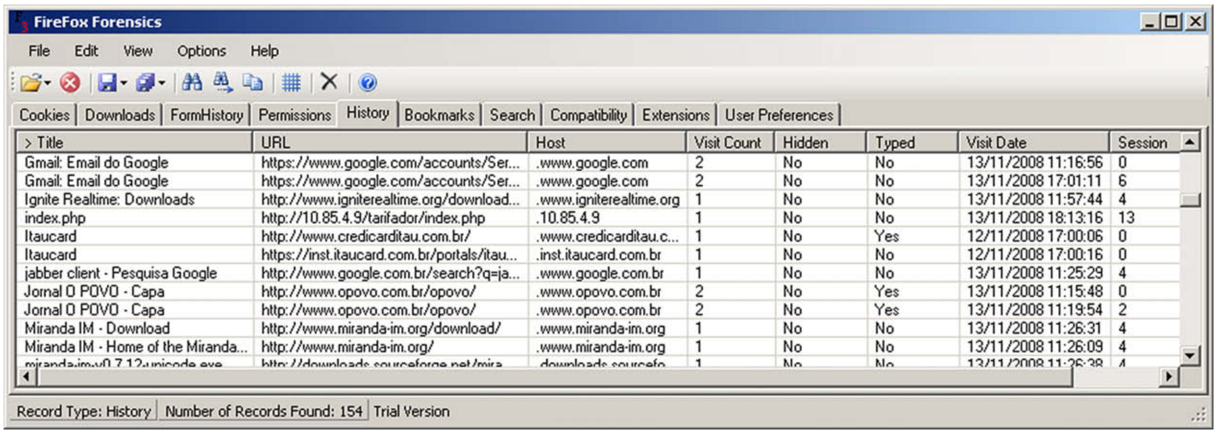
- Bookmarks are not deleted when the clear private data option is invoked by the user; the corresponding bookmark records remain in *moz_places* table. As previously discussed, bookmark records in *moz_bookmarks* table do not store the URL, only a foreign key related to *moz_places* table.
- Records in *moz_places* with queries are not deleted after a clear private data action.
- Considering x the greatest **id** value of the *moz_places* table before a clear private data and y the greatest **id** value after the clear private data; a new entry to be created in *moz_places* will be assigned the **id** value $y+1$, and not the value $x+1$. This causes these new entries to get **id** values already assigned to older entries.
- If a user deletes a bookmark, the corresponding history record is not deleted and remains in *moz_places* table.
- When deleting expired history records or when the clear private data option is invoked by the user, Firefox 3/SQLite effectively erases the content of each record, filling the space with zeros.
- Although the record's content is wiped, when searching all disk, record vestiges was found in unallocated space. The situations where this can happen will be explored in the next section.

3.2. *SQLite records in unallocated space*

As testing demonstrated, fragments of SQLite records may be found in unallocated space. Two situations can cause this to occur. The first is more uncommon and consequently it has less importance for a forensic analysis; as *places.sqlite* is a normal file in the file system, it can be reallocated by the operating system and the sectors it used to occupy will be marked as free on disk, but they still have data on it until the operating system decides to re-use them.

The other situation that results in records in unallocated space is the use of temporary files by SQLite. There are seven types of temporary files used by SQLite and the most important for a forensic analysis is the *rollback journal*. According to SQLite documentation (SQLite, 2008c): "*A rollback journal is a temporary file used to implement atomic commit and rollback capabilities in SQLite. The rollback journal is always located in the same directory as the database file and has the same name as the database file except with the 8 characters "**-journal**" appended. The rollback journal is usually created when a transaction is first started and is usually deleted when a transaction commits or rolls back. But there are exceptions to this rule.*"

When the database is used in "exclusive locking" mode the rollback journal is not deleted at the end of each transaction, but only when the exclusive access mode is exited, that happens when the browser is closed. When the transaction is



FireFox Forensics

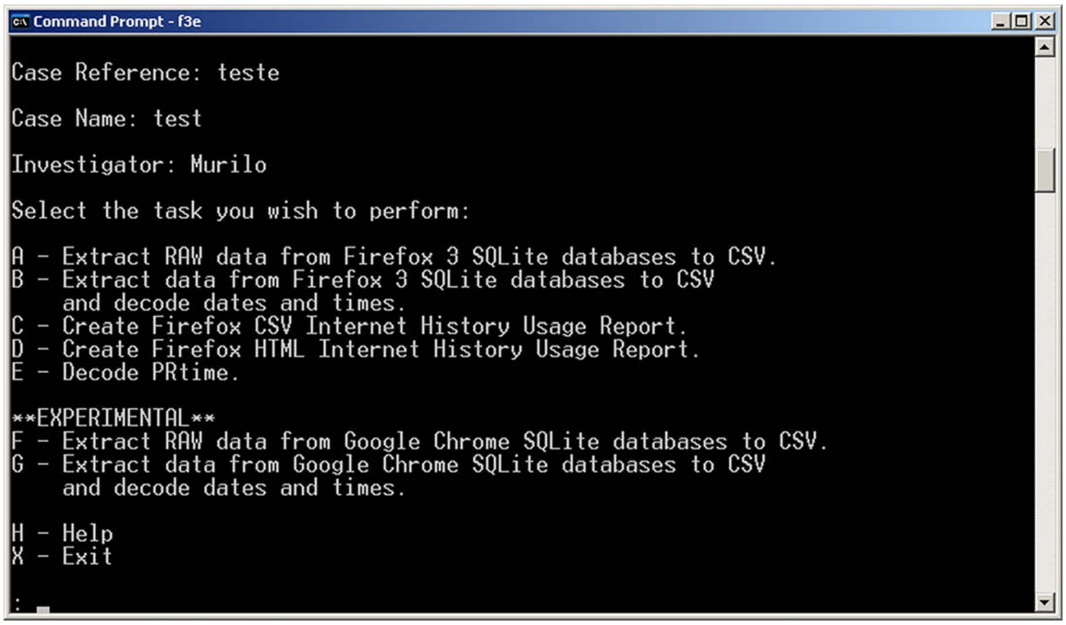
File Edit View Options Help

Cookies | Downloads | FormHistory | Permissions | History | Bookmarks | Search | Compatibility | Extensions | User Preferences

> Title	URL	Host	Visit Count	Hidden	Typed	Visit Date	Session
Gmail: Email do Google	https://www.google.com/accounts/Ser...	.www.google.com	2	No	No	13/11/2008 11:16:56	0
Gmail: Email do Google	https://www.google.com/accounts/Ser...	.www.google.com	2	No	No	13/11/2008 17:01:11	6
Ignite Realtime: Downloads	http://www.igniterealtime.org/download...	.www.igniterealtime.org	1	No	No	13/11/2008 11:57:44	4
index.php	http://10.85.4.9/tarifador/index.php	.10.85.4.9	1	No	No	13/11/2008 18:13:16	13
Itaucard	http://www.credicarditau.com.br/	.www.credicarditau.c...	1	No	Yes	12/11/2008 17:00:06	0
Itaucard	https://inst.itaucard.com.br/portals/itau...	.inst.itaucard.com.br	1	No	No	12/11/2008 17:00:16	0
jabber client - Pesquisa Google	http://www.google.com.br/search?q=ja...	.www.google.com.br	1	No	No	13/11/2008 11:25:29	4
Jornal O POVO - Capa	http://www.opovo.com.br/opovo/	.www.opovo.com.br	2	No	Yes	13/11/2008 11:15:48	0
Jornal O POVO - Capa	http://www.opovo.com.br/opovo/	.www.opovo.com.br	2	No	Yes	13/11/2008 11:19:54	2
Miranda IM - Download	http://www.miranda-im.org/download/	.www.miranda-im.org	1	No	No	13/11/2008 11:26:31	4
Miranda IM - Home of the Miranda...	http://www.miranda-im.org/	.www.miranda-im.org	1	No	No	13/11/2008 11:26:09	4

Record Type: History | Number of Records Found: 154 | Trial Version

Fig. 5 – Firefox Forensics interface.



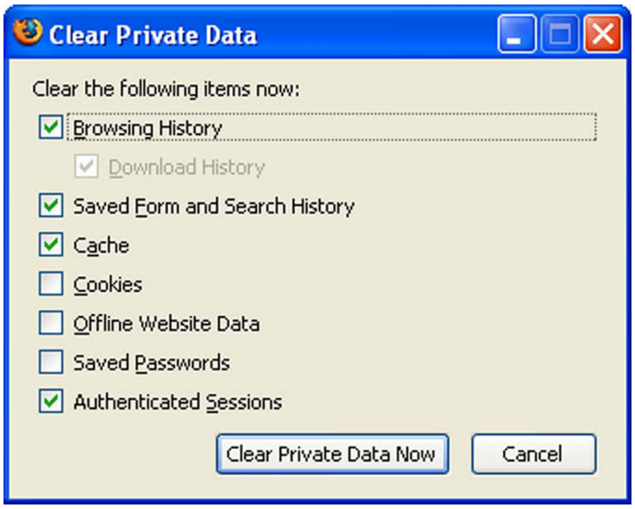
```
Case Reference: teste

Case Name: test

Investigator: Murilo

Select the task you wish to perform:

A - Extract RAW data from Firefox 3 SQLite databases to CSV.
B - Extract data from Firefox 3 SQLite databases to CSV
    and decode dates and times.
C - Create Firefox CSV Internet History Usage Report.
D - Create Firefox HTML Internet History Usage Report.
E - Decode PRtime.

**EXPERIMENTAL**
F - Extract RAW data from Google Chrome SQLite databases to CSV.
G - Extract data from Google Chrome SQLite databases to CSV
    and decode dates and times.

H - Help
X - Exit

: _
```

Fig. 6 – Firefox 3 Extractor main menu.

concluded, the header is zeroed. The exclusive locking mode is recommended to save operations in the file system, mainly at embedded environments.

It was verified that Firefox 3 uses exclusive locking mode, by observing the journal creation and its exclusion only when the browser is closed, as well as through analyzing the Firefox source code (Mozilla Cross-References, 2008a). This behavior results in the creation of deleted files or file fragments containing history records in unallocated disk space. This means that if the user configures Firefox to clear private data at each end of session, and even knowing that SQLite effectively deletes records by zeroing its space, even so it is possible to find records in disk, because of the rollback journal.

Another important characteristic of rollback journal is that it works with pages, not records. This means that if the user



Clear Private Data

Clear the following items now:

- [✓] Browsing History
  - [✓] Download History
- [✓] Saved Form and Search History
- [✓] Cache
- [ ] Cookies
- [ ] Offline Website Data
- [ ] Saved Passwords
- [✓] Authenticated Sessions

Fig. 7 – Menu to clear user's private data in its default configuration.

changes one record, all the records composing this page will go to the rollback journal, what increases the chances of finding records in unallocated space. It is important to note that records within the rollback journal have the same structure as the ones in the database. This structure will be discussed in the next section.

Besides the rollback journal, there are six more types of temporary files generated by SQLite, depending on the situation. It was detected the creation of the *statement journal* (*places.sqlite-stmtjrnl* file), which, likewise the rollback journal, can be used to recover history records.

3.3. *Internal structure of the SQLite database*

The SQLite version 3 files have a known header, which is "SQLite format 3". However, it is unlikely to recover the entire sqlite file, because the file tends to become fragmented or overwritten. As said, the temporary files created by SQLite, especially the rollback journal, are the main source of records in unallocated space. In addition, these journals have the header zeroed to indicate a successful transaction, making a signature search impracticable.

Thus, the most efficient method to recover the records is to search by the records themselves and ignore all the database structure. To do this, knowledge of their internal structure is required, what is presented in SQLite documentation (SQLite, 2008d). Essentially, an SQLite database is stored in segments, called pages, structured as B−trees, for index, or B+trees, for tables. The root and internal pages of B+trees contain only navigation information and the data fields are stored in leaf pages, as shown in Fig. 8 (Owens, 2006, page 350). To recover the records, all the B+tree structure will be ignored.

The SQLite records have the following sequential structure: the record size, the key value, the header and the data segment (D1 to DN), as shown in Fig. 9. The record size includes only the

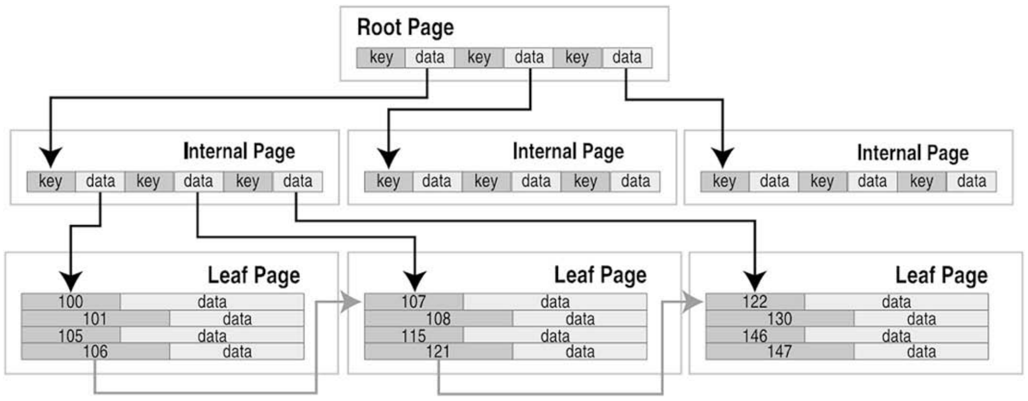

Root Page
key data key data key data
Internal Page
Leaf Page
100 data 101 data 105 data 106 data
107 data 108 data 115 data 121 data
122 data 130 data 146 data 147 data

Fig. 8 – B+tree structure of SQLite databases (Owens, 2006, page 350).

header and data sizes. The header comprises the header size (*hsize*) and an array of types and sizes (T1 to TN), which describes each field stored in the data segment. Both the *hsize* and the array, as well the **id,** are represented as a variable-sized 64-bit integer value. For this kind of value, SQLite uses a compression method based on Huffman coding (Wikipedia, 2008). This means that despite being a 64-bit integer, it will use 1 byte for values up to 127, 2 bytes up to 16,383, 3 bytes up to 2,097,151, and so on, up to 9 bytes (SQLite, 2008b). While this helps to reduce the database size, it complicates the identification and recovery of SQLite database records.

It is important to note that the record size, the key, and the integers in the header are stored using Huffman coding. Integers in the data segment do not use a compression method; however, they are stored in big-endian format, independently of the operating system used (SQLite, 2008d).

The array of types and sizes uses one value to identify the field type and the size. It uses the values in Table 2 as reference.

3.4. *SQLite record decoding example*

To clarify how the records are stored, an example of how to decode an SQLite record from *moz_places* table is presented. As previously discussed, SQLite uses a compression method based on Huffman coding to store integers in the header. If the value is greater than 127 two or more bytes will be used, depending on the case, to represent this value. It is expected that the value of **id** and the size of **url** and **title** will be, eventually, greater than 127. This makes it difficult to define the field boundaries and to recover the record.

The *moz_places* table from *places.sqlite* database has the following fields: **id** (integer), **url** (text), **title** (text), **rev_host** (text), **visit_count** (integer), **hidden** (integer), **typed** (integer), **favicon_id** (integer), **frecency** (integer). As previously mentioned, the header, which stores the header size and the array of types and sizes, is located between the record key (**id)** and the content of the first field (**url**). This example does not include the record size that is located before the **id**.

Fig. 10 presents a portion of SQLite file, that represents a *moz_places* record, with the hexadecimal representation on the left and ASCII on the right. In this example, the **id** is 396, but it is not stored as 0×018C. It is greater than 127, and is stored using Huffman coding, so the **id** is represented by the values 0×83 and 0×0C (decimals 131 and 12). Naming the first number *x* and the second *y*, the value can be decoded in the following way: $(x-128)\times 128+y$, what results in 396. The use of Huffman coding is identified by the fact that the first byte is greater than 127. For a 3 byte sequence *xyz*, the value can be decoded in the following way: $(x\times 128)\times 16384+(y-128)\times 128+z$.

The header size is the next byte, 0×0B, decimal 11. After the header size, there is always one byte, with the value 0×00, and then the array of types and sizes. The **url** field size is in Huffman coding, since the next value is 0×82. Therefore, the **url** size is represented by 0×82 and 0×37 (decimals 130 and 55), which decodes as $(130-128)\times 128+55=311$. According to Table 2, the value 311 represents a text field with size $(311-13)/2=149$ bytes.

The next field is **title**, with the value 0×21, decimal 33. According to Table 2, the value 33 represents a text field with size 10 bytes. Next field, **rev_host**, is the same thing, with value 0×2B that represents a text field with size 15 bytes.

Next, it follows five integers, representing the size of the fields: **visit_count** = 1, **hidden** = 1, **typed** = 1, **favicon_id** = 0 and **frecency** = 1, with no use of Huffman coding. According to Table 2, the value 1 represents a one byte signed integer and the value 0 means NULL, which has no content. With knowledge of the header, it is not difficult to extract the data fields, remembering that integers in data fields are stored with no compression and in big-endian format. The values in the example are:

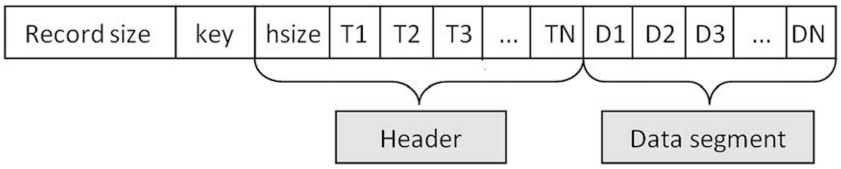


Fig. 9 – Record structure.

Table 2 – Field type values

Value	Meaning	Length of data in bytes
0	NULL	0
N from 1 to 4	Signed integer	N
5	Signed integer	6
6	Signed integer	8
7	IEEE float	8
8–11	Reserved for future use	N/A
N > 12 and even	BLOB	(N-12)/2
N > 13 and odd	Text	(N-13)/2

- **url**: http://www.cvc.com.br/site/_monteViagemCompleta/monteViagemCompleta1.jsf;jsessionid=c0a82a6f30daa39545e41b9043529a9aafd27b6738a7.e38Pb34Mch8Lc40Naxb0
- **title**: Portal CVC
- **rev_host**: rb.moc.cvc.www.
- **visit_count**: 1
- **hidden**: 0
- **typed**: 0
- **favicon_id**: NULL (no content)
- **frecency**: 50 (0–32)

4. Algorithm to recover deleted records

In this section, an algorithm to recover Firefox 3 history records from unallocated space is proposed. This algorithm is specifically design to recover entries from *moz_places* table.

4.1. *Algorithm*

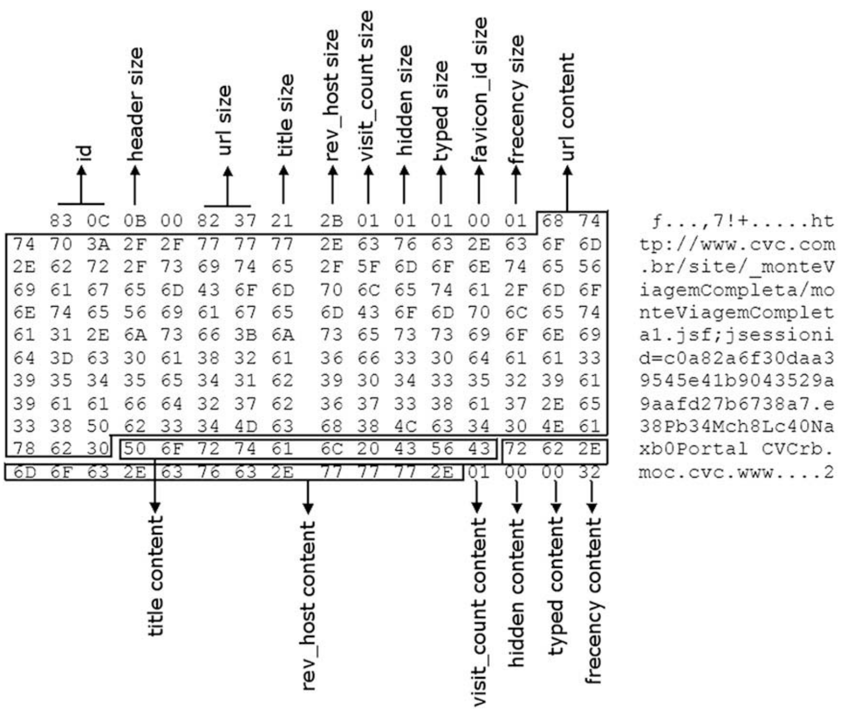
By knowing the record structure, it is possible to develop an algorithm to recover its content. The records by themselves do not present any signature or magic numbers. In case of history records from *moz_places* table, the best way to search is through the protocol indication on **url** field, as, for example, "http://". After finding a URL, it is possible to make some consistency checks to confirm whether it is a history record or not.

The first test to be made is verifying the five bytes before the protocol signature. These bytes should represent the array of types and sizes of five integers, which are **visit_count**, **hidden**, **typed**, **favicon_id** and **frecency**. It is possible to establish limits for these field values. As integers, according to Table 2, they should have values between 1 and 6, or 0 to represent NULL, but it is possible to narrow down this range.

Integers in the data segment are not stored using compression; if the array of types and sizes indicates a integer field use one byte, this integer value can be up to 256; if it uses two bytes the value can be up to 65,536; three bytes up to 16,777,216, and so on. The value of **visit_count** can grow as the user uses the browser, but it is expected to be less than 16,777,216, and, consequently, be stored using up to 3 bytes. Although an integer value in the array of types and sizes should be between 1 and 6, in case of **visit_count**, it should be only 1, 2 or 3.

The **hidden** and **typed** are like Boolean fields and their values in the data segment are always 0 or 1, needing 1 byte, each one, to be stored, what means that their values in the array of types and sizes are always 1.

In the same way of **visit_count**, the **favicon_id** and **frencency** fields can grow as the browser is used, but they are expected to be less than 16,777,216, and, consequently, be stored using up



```
   83 0C 0B 00 82 37 21 2B 01 01 01 00 01 68 74    ƒ...,7!+.....ht
74 70 3A 2F 2F 77 77 77 2E 63 76 63 2E 63 6F 6D    tp://www.cvc.com
2E 62 72 2F 73 69 74 65 2F 5F 6D 6F 6E 74 65 56    .br/site/_monteV
69 61 67 65 6D 43 6F 6D 70 6C 65 74 61 2F 6D 6F    iagemCompleta/mo
6E 74 65 56 69 61 67 65 6D 43 6F 6D 70 6C 65 74    nteViagemComplet
61 31 2E 6A 73 66 3B 6A 73 65 73 73 69 6F 6E 69    a1.jsf;jsessioni
64 3D 63 30 61 38 32 61 36 66 33 30 64 61 61 33    d=c0a82a6f30daa3
39 35 34 35 65 34 31 62 39 30 34 33 35 32 39 61    9545e41b9043529a
39 61 61 66 64 32 37 62 36 37 33 38 61 37 2E 65    9aafd27b6738a7.e
33 38 50 62 33 34 4D 63 68 38 4C 63 34 30 4E 61    38Pb34Mch8Lc40Na
78 62 30 50 6F 72 74 61 6C 20 43 56 43 72 62 2E    xb0Portal CVCrb.
6D 6F 63 2E 63 76 63 2E 77 77 77 2E 01 00 00 32    moc.cvc.www....2
```

Fig. 10 – Example of SQLite record.

to 3 bytes. In case of **favicon_id**, it can also be NULL, what means its value in the array of types and sizes may be 0.

Using a *grep* style notation, the values in the array of types and sizes of these five integers should be: [1–3]11[0–3][1–3], that is, the first byte should be 1, 2 or 3, the second and third must be 1, the fourth can vary from 0 to 3 and the fifth from 1 to 3.

If the five values are consistent, there is a great chance that it is a history record, but it is not confirmed yet. The algorithm should continue reading the text field sizes (**url**, **title** and **rev_host**), which may have a broad range of values and it is not feasible to carry out consistency checks.

Differently from integer fields, which always occupy one byte in the array of types and sizes, the text fields may take up more than one byte, whenever the value is greater than 127 (stored using Huffman coding). Table 2 indicates that an odd value N greater than 13 identifies a text field in the array of types and sizes, and the size of the text is $(N - 13)/2$. If, for example, N is 129, this field will occupy two bytes in the array and 58 bytes, which is the size of the text, in the data segment. For one of these text fields to occupy 3 bytes in the array, N must be 16,385 or greater, what means a text of length 8186 or greater, which is not likely to happen. Therefore, the algorithm considers that the **url**, the **title** and the **rev_host** fields occupy 1 or 2 bytes in the array. As the array is being parsed backwards, to read one of these text fields, the algorithm should read two bytes at offsets x and $x - 1$; if the value at $x - 1$ is greater than 127 the field is in Huffman coding and is using the 2 bytes, otherwise, the value at $x - 1$ should be discarded and only one byte, at offset x, should be used.

The header size, followed by a byte 0×00, is located before the **url** field size and indicates the length of the array of types and sizes, including the header size itself and the byte 0×00. In case of *moz_places* table records, the header size occupies always 1 byte, with a minimum value of 10 (1 byte for each integer field + 1 byte for each text field + 1 byte for the header size + 1 byte for 0×00) and a maximum value of 13 (1 byte for each integer field + 2 byte for each text field + 1 byte for the header size + 1 byte for 0×00).

The algorithm considers that a Firefox 3 history record was found if the header size is followed by a byte 0×00 and matches the field sizes read. After the header size and all field sizes are known, it is straightforward to read the data segment, however there is still the **id** and the record size to be read. The algorithm assumes that the maximum value for **id** and record size are 2,097,151 and 16,383, and, consequently, they will occupy up to 3 and 2 bytes, respectively. The same approach used to read the text fields can be employed to read the **id** and the record size. Additionally, the record size can be tested against the sum of header and data sizes. A simplified version of the algorithm in pseudocode is shown below.

```
1. While not EOF do (Find the next protocol indication (http://
   or https:// or ftp:// or file://))
   1.1 Read 5 bytes before the protocol indication to the vari-
       ables visit_count, hidden, typed, favicon_id and
       frecency
   1.2 HeaderSize = 5 // this variable keeps track of the header
       size
   1.3 If visit_count > 0 and visit_count < 4 and
          hidden = = 1 and typed = = 1 and
          favicon_id > -1 and favicon_id < 4 and
          frecency > 0 and frecency < 4

       Then go to 1.4 Else go to 1

   1.4 Read 7 bytes before visit_count to the variables record_
       size, id, hsize, zero, url, title and rev_host (To simplify this
       pseudocode, this step assumes record_size, id, url, title
       and rev_host occupy only one byte. If the value of some of
       these fields is greater than 127, it will use more than one
       byte and this step would need to read more bytes.)
   1.5 HeaderSize + = 1 + 1 + sizeof(url) + sizeof(title) + sizeof
       (rev_host)
   1.6 If hsize = = HeaderSize and zero == 0×00 then go to 1.7
       else go to 1
   1.7 Seek to the beginning of the protocol indication
   1.8 For each field, read its content
```

4.2. *Testing methodology and results*

Based on the previously discussed algorithm, an application in C, called *ff3hr*, was developed to identify and extract records from unallocated space. It is specifically designed to identify and process *moz_places* table entries, and currently will not recover other Firefox records.

The command line interface program takes a file name as input and prints the records to a file or standard output. The output is in the following order: *ID, URL, Title, Rev_host, Visit Count, Typed, Frecency*. The fields **hidden** and **favicon_id** are not displayed because they do not present relevant information.

The file to be processed can be a whole disk image or just the unallocated space. If it is a disk image the records from the regular *places.sqlite* file will be printed too. The program searches and recovers records based in four signatures: http://, https://, ftp:// and file://.

To test the program efficiency, Firefox 3.0.4 was installed on a computer running Windows XP in a NTFS partition. Firefox was used for browse the Internet during a five day period. In the end of the period, the *moz_places* table had 254 records, with highest **id** value equal to 507. Though no records were voluntarily deleted, Firefox deletes automatically some entries, as those containing embedded links (type 4 in Table 1), which have a lifetime of just one day (Mozilla Cross-References, 2008b). Three of these 254 records were Firefox default queries, so only 251 counts. The *moz_historyvisits* table had 253 entries. After invoking the clear private data option, the *moz_places* and the *moz_historyvisits* tables had, respectively, 28 and 0 records.

A raw image of the hard drive was created and the *ff3hr* program was executed on the entire image. The program recovered a total of 670 *moz_places* records, some of which were duplicates. Further analysis revealed that 396 unique records were recovered: the 251 records that were present in the *moz_places* table prior to invoking the clear private data option, and 145 additional records that had expired and were deleted automatically by Firefox as a result of normal use of the application. Furthermore, the highest **id** value within the recovered records was 518, indicating that the program did not recover 122 records from the total of 518. Some reasons underlying non-recovered records are discussed in the next paragraph. In this particular test, these non-recovered records consist of short lifetime entries, like embedded links.

Some records were found more than once: 32 records were found 4 times, 38 were found 3 times and 102 were found 2 times. The user behavior and the file allocation policy enforced by the operating system influence the results when recovering data from unallocated sectors. Moreover, the way SQLite organizes the pages with records and how these pages are copied to temporary files can determine the amount of records recovered and explain the fact of records being found more than once. If a record belongs to a page that is frequently accessed, it may be found many times; on the other hand, if a record belongs to a page that is not frequently accessed it may not be found.

No false positives were identified during this test. Two records were recovered partially overwritten; they were located in a sector boundary, and the second sector was re-used, erasing part of the record content. To inform the user when this happens, the program tests if the **typed** content is equal to 0 or 1, and if it is not, it prints "Alert: overwritten record". In the same way, if the record size read is not equal to the sum of sizes of the header and the data segment, but the other consistency checks are correct, the tool considers that record size and/or **id** were overwritten and the record content is intact.

A second test used Firefox 3.0.1 installed on a computer running Windows XP in a NTFS partition. In this case, the option to clear private data whenever the browser closes was enabled, and Firefox was used during 15 days. On the last day, the history was not cleared and the *moz_places* and the *moz_historyvisits* tables had, respectively, 102 (excluding the default queries) and 30 records. Most of the records in *moz_places* table were from RSS feeds which do not generate entries in the *moz_historyvisits* table. The highest **id** in *moz_places* table was 972 and in *moz_historyvisits* was 30. The routine of clearing the private data on every close makes the total number of created records in *moz_places* greater than 972, however this number was not calculated.

A raw image of the hard drive was created and the *ff3hr* program was executed on the entire image. The program found 1276 *moz_places* records, with 581 unique entries and the highest **id** value equal to 972. As expected, the 102 non-deleted records present in the *moz_places* were recovered. As in the first test, no false positive was found, and two records were recovered partially overwritten.

The application and its source code are available at http://sourceforge.net/projects/ff3hr/.

5. Conclusion

This work presented the new approach Firefox 3 uses to store information about Internet history, bookmarks, downloads, form field data and cookies. The utilities available to process Firefox 3 SQLite databases were tested and showed that they can extract the relevant data and export them to CSV or HTML. However, the information contained in the **from_visit** field, which is very important for a forensic investigation, is not parsed by any of the tools tested.

The paper demonstrated that file fragments containing SQLite records are commonly found in unallocated disk space. This behavior is attributed, mainly, to the use of temporary files by SQLite. These temporary files do not use the database tree structure, and the attempt to carve them should depend only on the record itself.

The paper also discussed the SQLite record internal structure and proposed an algorithm to recover Firefox 3 history records based on this structure. In the tests made, our tool recovered many of the deleted records, and, in some cases, the same entry was recovered more than once. The quantity of records recovered depends a lot on the reallocation policy of unused sectors. The tests showed that even if the user clears the private data on every browser close, it is possible to find and recover history records.

The high false positive rate generated by carving data in unallocated space can be a drawback of such algorithms. However, the consistency checks imposed in our algorithm are very efficient, and in the tests made, the tool found no false positive record.

The same methodology applied to recover *moz_places* entries can be used to carve records from other tables. In case of *moz_places*, the protocol indication facilitates to identify a record; with other tables, like *moz_historyvisits*, which all fields are numbers, to identify a record is more complex, and many false positives can occur.

In fact, the same approach presented in this study may be used to recover records from other applications that also use SQLite databases, as, for example, Chrome web browser, Safari web browser, and IPhone operating system.

REFERENCES

Jones Keith, Belani Rohyt. Web browser forensics. Security focus, <http://www.securityfocus.com/infocus/1827>; 2008.

Mozilla Cross-References. Mozilla source code, <http://mxr.mozilla.org/mozilla-central/source/toolkit/components/places/src/nsNavHistory.cpp>; 2008a.

Mozilla Cross-References. Mozilla source code, <http://mxr.mozilla.org/mozillasvn-all/source/projects/fennec/toolkit/components/places/src/nsNavHistoryExpire.cpp>; 2008b.

Mozilla Developer Center. nsINavHistoryService – MDC, <http://developer.mozilla.org/en/docs/nsINavHistoryService>; 2008a.

Mozilla Developer Center. PRTime, <http://developer.mozilla.org/en/docs/PRTime>; 2008b.

Mozilla Developer Center. The Places frecency algorithm, <http://developer.mozilla.org/en/docs/The_Places_frecency_algorithm>; 2008c.

Mozilla Foundation. Mozilla sets new guinness world record with Firefox 3 Downloads, <http://www.mozilla.com/en-US/press/mozilla-2008-07-02.html>; 2008.

Owens Michael. The Definitive Guide to SQLite. s.l. Apress; 2006.

SQLite. SQLite Home Page, <http://www.sqlite.org>; 2008a.

SQLite. SQLite Version 3 Overview, <http://www.sqlite.org/version3.html>; 2008b.

SQLite. Temporary Files Used By SQLite, <http://www.sqlite.org/tempfiles.html>; 2008c.

SQLite. SQLite Database File Format, <http://www.sqlite.org/fileformat.html>; 2008d.

Wikipedia. Huffman coding, <http://en.wikipedia.org/wiki/Huffman_code>; 2008.