Chapter 4

XML CONVERSION OF THE WINDOWS REGISTRY FOR FORENSIC PROCESSING AND DISTRIBUTION

Alex Nelson

Abstract The Windows Registry often contains key data that help determine the activities performed on a computer. While some forensic tools format Registry data for common questions that are required to be answered in digital investigations, their output is geared for standalone use, not for indexable content.

This paper describes RegXML, an XML syntax designed to represent Windows Registry hive files. RegXML captures the logical structure of a hive and notes the locations of found data within hive files. The paper also describes a Python library designed to be used with RegXML and the results obtained upon applying the library to analyze two forensic corpora. Experimental results are presented based on hundreds of disk images, thousands of hive files and tens of millions of Registry cells.

Keywords: Windows Registry, Registry analysis, XML syntax, RegXML

1. Introduction

The Windows Registry is a rich source of information in forensic investigations. Several commercial forensic tools can be used to conduct keyword searches of a Registry, much like they can be used to search through a file system. However, they perform what can be characterized as "first-order analysis" [7] – data such as files and Registry values are examined for the information they contain and the scope of the search is confined to a single computer or a single case. This style of analysis is commonly employed in forensic practice and betrays an underlying problem: forensic analysis is fundamentally a silo activity.

Silos exist because it is difficult to share information about cases, both for legal and technical reasons. The legal reasons arise from the need

to maintain confidentiality. However, the simplest, most-encompassing technical reason is the difficulty of expressing the data in a structured, comparable format.

This paper proposes RegXML, a well-structured, interchangeable format for the Windows Registry. RegXML provides a mechanism for scaling forensic analyses to thousands of data sources. The data sources do not need to be complete, and can be excerpts. Examples include a collection of keys known to represent the Registry footprint of steganographic software [25], or malware summaries from antivirus vendors that comprise exchangeable XML annotations of keys, values and relative modification times.

An analysis of many data sources is categorized as bulk [8] or second-order analysis [7] analysis. Such an analysis is easily realized using a scriptable interface to the analyzer program and a well-structured data format that faithfully represents the contents of the original input items. To perform Registry analysis at this scale, it is also necessary to validate the robustness of the tools that are used. The simplest method for validating robustness is to measure how often a tool reports failure. A better approach is to use the common RegXML format to help validate the capabilities and correctness of forensic tools; in particular, RegXML facilitates the comparison of different approaches. This is especially important when data must be inferred from damaged evidence (e.g., evidence extracted from unallocated content). In the Casey Anthony case, the prosecution introduced as evidence the faulty, poorly-corroborated findings of a single tool [1]. Well-structured, comparable data can help prevent such discrepancies.

This paper makes three contributions. The first is RegXML, an XML representation of hive files, and its implementation under `hivex` [13]. The second is a Python framework for processing RegXML. The third is a time analysis of hives in two research corpora: a real data set collected from several hundred used computers that were purchased on the secondary market [12], and a longitudinal, realistic data set [24].

2. Related Work

RegXML is a component of the Digital Forensics XML (DFXML) syntax and toolset [11]. As with the parent project, our goal is to facilitate the exchange of structured data and metadata. The principal focus is the design and validation of a Registry data representation.

The National Software Reference Library ran the WiReD Project to index software Registry artifacts as an additional component of the Reference Data Set [22]. However, WiReD does not offer a full-fledged XML

schema to describe artifacts. RegXML, with some difference extensions, can serve as a Registry excerpt reporter for this effort.

The CybOX language [15] describes observable artifacts that are relevant to digital forensics, logging and malware characterization, among other activities. A portion of the language describes Registry artifacts. However, the focus is on the live Windows Registry instead of hive file artifacts. For example, the enumeration of hives is for high-level Registry keys such as `HKEY_CURRENT_USER`, not hive file types.

The MAEC Project [16] provides an XML summary of malware behavior, including the file system footprint, Registry footprint and more. The XML Schema Document (Version 1.1) includes descriptors for Registry artifacts in malware under `Registry_Object_Attributes`. A string describes the affected hive, albeit without specifications on how to identify the hive; this can be complicated for hives that do not reside in files in the file system, such as the `HARDWARE` hive, which is only present in memory [20]. Also, if multiple components of a hive are to be described with MAEC, according to the XML Schema Document there would have to be one `Registry_Object_Attributes` element per affected value. The schema also lacks a descriptor for handling encoding types, which may be necessary to handle parser-unfriendly characters (e.g., ®) or steganographic tools that may hide data in the Registry [23].

Much work has been accomplished in the area of Registry analysis since Russinovich's early description of the Registry structure [20]. Morgan [17] and Thomassen [21] discovered the presence of freed-but-not-erased content within the slackspace of a hive file. Thomassen [21] also describes potential parsing errors and provides a useful reference description of the metadata and data structures. As described later in this paper, we were able to verify, by spotting in our real-world usage corpus, a particular structural pointer correction that Thomassen made to the collective knowledge about hive structure. Morgan also provides thorough documentation of data structures and deletion patterns [18], while separately noting data recovery [17].

Carvey and Zhu [2, 26] observed that system restore points contain timestamped copies of the Registry, so systems can provide some history without an investigator needing to take multiple images. Registry Decoder from Digital Forensics Solutions [5] includes these files, and Volume Shadow Copy files for Windows Vista and Windows 7.

Our work employs a research corpus where multiple computers were consistently imaged nightly for several weeks [24]. In our experiments, we did not compare contents within system restore points; instead, we used the differences from the hives on different days.

To permit the verification of the results, our work uses data that is available to other researchers [12]. Our analysis tool is also made available as a modification to the open source `hivex` library [13]. We encourage other Registry-focused projects, such as the Registry Decoder and RegRipper [4], to also implement RegXML in order to validate the completeness of their results. Currently, the outputs of these tools have to be parsed to permit comparisons. Furthermore, any pairwise comparison of Registry tools would require the validation of the comparator. Implementing a common output format such as RegXML would greatly simplify the task of comparing digital forensic tools.

3. Hive Structure

The Registry is a composition of hive files. Each hive is like a file system, essentially making the Registry a multi-volume file system namespace. Most of the hives are stored on disk as files with the following paths [3]:

- Under `%WINDOWS%\system32\config`: `components`, `sam`, `security`, `software` and `system`
- Under `%USER%`: `localsettings\application data\microsoft\windows\usrclass.dat` and `ntuser.dat`

Other hives are volatile and can only be retrieved through memory carving techniques. We focus on in-file-system hives in this work.

Each hive is logically a collection of keys (also called nodes) and values. Unless otherwise noted, the detailed description provided below comes from Thomassen [21], which we found to be accurate through our work with the `hivex` library [13].

Physically, a hive is divided into 4 KiB blocks. Bins reside in an integral number of contiguous blocks, using the first four bytes of the first block for the signature "`hbin`" and leaving the rest for any cells that will fit. The first block is called the base block, which has its own structure. The size of a cell is always a multiple of 8 bytes, and the last active cell in a bin owns the remainder of the bin. Table 1 lists all the cell types and their signatures.

Several of the cell types point to one another. Keys point to their parent keys, while values do not. Keys point to value lists (not to individual values) and value lists only point to values. Subkey lists point to other subkey lists or subkeys. Keys point to security descriptor cells (note that we do not investigate security descriptor cells in this work).

Among all the cell types, only two logically contiguous components can be stored in a physically fragmented manner. One is the list of all the

Table 1. Registry cell types and their signatures [21].

Cell Type	Signature
Key cell	`nk`
Class name cell	
Security descriptor cell	`sk`
Subkey list cell	`lf`, `lh`, `ri`, `li`
Value list cell	
Value cell	`vk`
Value data cell	
Big value data cell	`db`

subkeys of a key, and the other is value data longer than 16,388 bytes, which are supported in hives version 1.4 or greater [18]. Value lists, value data shorter than 16,344 bytes and keys are stored in a contiguous manner.

Most hive data structures report their own size in embedded headers. The base block and all bins report their size after their first four signature bytes. Cells also report their size and allocation status; all the cell types report their size at the beginning. The size of an allocated cell always appears as the negative of the actual size; this becomes a positive number when the cell is freed. Thomassen [21] discovered that if the unallocated space within a hive is parsed, then deleted data or moved data can be found by checking for the sign of the size. Thomassen also found that the space in bins is freed by moving all the allocated data towards the beginning of the bin; data at the end is overwritten only if a newly allocated cell needs the space. Therefore, while the space at the end of a bin may belong to the last used cell, it does not mean that the unused space is devoid of content. If this content is to be reported, its location should be included because there is no guarantee that old data has any consistency with the hive at the time of analysis.

Two hive data structures contain mtimes (modification times). The base block contains an mtime for the hive. The mtime of a key is not guaranteed to be the newest mtime for the entire subtree rooted at the key. The Registry shares this parent-child relative time behavior with file systems, where updating a file in a directory does not mean that the directory itself was updated. The only hives that we encountered with parents consistently newer than children were essentially unused (between one and six keys). We also observed that the mtime in the base block is not guaranteed to be the latest mtime related to the hive; this is discussed in Section 7.

```
<?xml version="1.0" encoding="UTF-8"?>
<msregistry hive_version="1.3"
  name="and Settings\Charlie\ntuser.dat">
  <mtime>2009-11-17T00:33:30.9375Z</mtime>
  <key name="$$$PROTO.HIV" root="1">
    <mtime>2009-11-13T04:58:06.15625Z</mtime>
    <byte_runs>
      <byte_run file_offset="4128" len="92"/> ...
    </byte_runs>
    <key name="AppEvents"> ...
      <key name="EventLabels"> ...
        <key name=".Default"> ...
          <value type="REG_SZ" default="1" value= "Default Beep">
            <byte_runs>
              <byte_run file_offset="5136" len="24"/>
              <byte_run file_offset="5160" len="30"/>
            </byte_runs>
          </value>
        ...
```

Figure 1. RegXML excerpt for a user's `ntuser.dat` hive generated by `hivexml`.

4. Representing Hives with RegXML

One of our goals was to represent the tree structure of hives in an XML syntax (i.e., using RegXML). RegXML primarily represents the logical structure of the hive, although there are ample annotations with the `byte_run` elements that point to the physical storage points.

Figure 1 shows a RegXML sample. Most of the logical structure is represented in a straightforward manner – with `key` elements having `key` and `value` children. However, the following hive data were more difficult to represent:

- **Data Location:** The location of every element is defined to be the metadata and data cells necessary to reconstruct the logical cell and its child references. For example, a key cell with one cell devoted to listing four subkeys would have two byte runs: one for the key cell and one for the list cell. Each child then gets its own byte runs. This is consistent with file systems, where the listing and order are properties of the directory, not the listed file.

- **Time:** All times are represented as ISO 8601 strings in optional `mtime` elements, as in DFXML [11]. The only times that are part of the hive structure are mtimes, which are stored as Windows filetimes [14], where filetime 0 is the first 100-nanosecond interval of the year 1601. Therefore, in the context of hives, a value for

the day 1601-01-01 can only represent null (or a system clock that is skewed beyond hopelessness). We use an element to maintain consistency with the time representation of file systems in DFXML.

- **Encodings:** Aside from labeled raw binary data, we encountered some unexpected unprintable characters (including string types that were abused to hold raw binary). Some path names also include unexpected characters, such as the ® sign included in a product name. Where necessary, the unexpected data are base-64-encoded with some type of encoding attribute.

- **Paths:** For encoding reasons shared with value text, we chose to have each cell only note its name (not its full path) when generating the complete RegXML from a hive. This helped us catch a processing error: we had concatenated path names with forward-slash characters without realizing that the forward-slash is a legal name character (e.g., in MIME-type names such as "`audio/mpeg`") that is used in around 230,000 cells across all of our data. With the general extensibility of XML, a `full_path` attribute could easily be added to any key and value, improving the searchability of excerpted RegXML.

To implement RegXML, we modified the `hivexml` program from the `hivex` Project [13]. `hivex` is a library that reads and writes hive files, providing a shell (`hivexsh`), XML converter (`hivexml`) and bindings for several languages. `hivex` functions primarily by walking the allocated hive from a starting offset (which does not need to be the root) and invoking callback functions on each visited key and value. Unfortunately, due to some difficulties with the walking architecture design of `hivex`, `hivexml` does not currently produce byte runs for key child lists.

5. RegXML Processing with `dfxml.py`

The Digital Forensics XML Python processing library is distributed with the Bulk Extractor and Fiwalk Projects [9, 10]. We have updated this library to parse RegXML as of Bulk Extractor version 1.0.3 and Fiwalk version 0.6.16.

`dfxml.py` processes Digital Forensics XML [11], which presents a file system to the user as file objects. We implemented RegXML processing with key and value objects, which we refer to as cell objects. The Python Expat library [19] is the basic processing engine; it is a stream-based parser with events for starting and ending elements, and an event for encountering character data.

```
#!/usr/bin/env python
"""Usage: python demo_registry_timeline.py <input.regxml>"""
import dfxml, sys
timeline = []
def process(co):
    mtime = co.mtime()
    if mtime != None:
        timeline.append([mtime, co.full_path(), "modified"])
dfxml.read_regxml(xmlfile=open(sys.argv[1], "r"), callback=process)
for record in sorted(timeline):
    print("\t".join( map(str, record)) )
```

Figure 2. Demonstration program that prints a timeline of hive modification times.

Although RegXML and DFXML differ in their structure (where a DFXML file is mostly a stream of `fileobject` elements), the API for interacting with either is equivalent. The only difference is the logical order in which the namespace is parsed. There is also sufficient information in RegXML to represent a file like DFXML `fileobject` elements.

`dfxml.py` invokes a user-supplied callback function when it has completed parsing a key or value, so a user visits the RegXML hierarchy in postorder. Fortunately, if a user's script cannot function in a postorder traversal, then the cell objects and hive namepsace are retained as a property of the object returned by `dfxml.read_regxml()`. Note that metadata elements in Figure 1, including `mtime` and `byte_runs`, are recorded before child keys and values in RegXML. This allows a parent object to be partially populated before all its children are visited, allowing each child to compare its properties against its parent if desired. This ordering improves performance, but is not necessary because it is possible to iterate over cells after completing the XML input.

To demonstrate the API, we consider a scenario where the times in a Registry need to be investigated for consistency with the rest of the system [26]. One necessary step is extracting the times of the keys. Figure 2 shows a program that prints the timeline of all activity within a hive. The figure is adapted from a similarly succinct timeline program for DFXML [11]. The mtimes are retrieved by a function (`co.mtime()`) to ease the processing of cell objects that can be keys or values and, thus, may not have a modified time. Byte runs are also retrieved by a function (`co.byte_runs()`). Other desired data, such as the cell name and full path, are also retrieved using functions, with any base-64 decoding handled transparently. The `registry_object`, to which every cell object maintains a handle, stores a dictionary index of all the cell objects by path (`object_index`). To access the `registry_object`, the

`regxml_reader` is returned by `dfxml.read_regxml()`, and the reader maintains a handle to the `registry_object`. All these pointers to the `registry_object` ease tasks such as determining hive-scoped extrema (e.g., recording the earliest times in an analysis).

6. Analysis

We compiled hive data from two corpora:

- **M57-Patents Scenario (M57):** The M57-Patents Scenario [24] is a four-week series of desktop computer images produced for forensic education and research. Four personas working in a fictitious company have their computers imaged at the end of every workday for a month. The resulting data set contains the day-to-day known ground truth of computer activities.

  Each computer was imaged by shutting it down using the Windows GUI and rebooting into a Linux LiveCD in order to take a forensic image of the hard drive. Therefore, the computers are expected to be in a consistent state at the time of imaging.

  Three of the personas used Windows XP computers, up-to-date through November 2009; the remaining persona used a Windows Vista computer. In all, the corpus contains 83 storage device images, 79 of them desktop images.

- **Real Data Corpus (RDC):** The Real Data Corpus [12] is drawn from a collection of storage media purchased on the second-hand market from around the world. The media are primarily hard drives and flash drives. Note that the hard drives are not guaranteed to be the drives that house the operating system partition or user profiles. Our analysis was performed on a total of 2,027 storage device images.

Table 2 provides the statistics for the from-image-to-analysis data processing. We modified a Python file extraction script from the Fiwalk utility [9] to extract files that ended with the patterns noted in Section 3. If any extraction failed, we excluded all the hives in the associated image from further analysis. We then ran `hivexml` on each extracted hive. If `hivexml` failed, then the XML would be unusable.

To facilitate the analysis, we wrote a script based on the `dfxml.py` interface to convert from XML to SQLite. This was done for multiple reasons. First, running all the `hivexml` output through the `dfxml.py` module serves as a robustness test and an additional data-processing checkpoint. Second, most of the planned queries could have been performed with XML tools like XPath, DOM, in-memory lists, counters and

Table 2. Processing statistics for the hive analysis process.

M57	RDC	Description
83	2,027	Number of media images
0	145	Images whose hive extraction process exited in an error state
25	231	Hives that `hivexml` could not process
1,016	3,053	Hives that `hivexml` could process
0	148	Hives that `dfxml.py` could not process
0	941,701	Cells processed before `dfxml.py` failed
79	198	Total images in SQLite
1,016	2,986	Total hives in SQLite
16,238,068	34,127,394	Total cells in SQLite

hash tables, but it essentially would have been equivalent to writing a SQL engine. Indeed, we did not go straight to SQL from the raw hive files because the advantage of RegXML as a transport and summary syntax could not be ignored.

The reasons for `hivexml` failures primarily involved subkey pointer constraints, which we had not yet relaxed (the `hivex` library does not recognize "li" subkey blocks and assumes "ri" blocks will not point to "ri" blocks). `dfxml.py` processing failures were entirely based on encountering the same path (byte-for-byte) multiple times, violating our assumption of unique paths. We discarded all the cells from hives containing these path errors from further analysis. While this limited the data that were analyzed, we were still able to find interesting inconsistencies in the hives that were not discarded.

We did not include hives extracted from system restore points in our analysis. We decided to limit the hive extraction to only the hives present and active in the file systems at the time of imaging. This was done because we had no guarantees that the restore points would be logically intact, particularly in the RDC drives.

Still, we believe that the RDC hives serve as a fine robustness testbed. Additionally, in the case of the M57 corpus, we have the opportunity to measure changes from the previous day of computer use. As a demonstration analysis, we use this available sequence to measure cell mtime properties.

7. Time Inconsistencies

We noted earlier that the mtime consistency for hives was not exactly what one would expect. An update to an internal key should cause

Table 3. Sources of latest mtimes associated with hives.

	File System	Hive Root	Latest Key
M57	620	499	24
RDC	1,384	1,989	306

an update to the hive that, when recorded to the disk, should update the mtime of the hive in the file system. Table 3 shows how well the assumption plays out. We split the RDC hives into basename types (i.e., `usrclass.dat` and `ntuser.dat` not broken out by profile) and observed roughly the same proportions. Hence, to see the last update time of a hive, at least these three times should be checked.

Table 4. Back-in-time mtime changes in the M57 image `terry-2009-11-19.aff`.

Hive	Backwards Changes	Non-Backwards Changes	% Backwards Changes
SOFTWARE	1,069	188,677	0.57
SYSTEM	316	62,474	0.51
SECURITY	1	149	0.67
LocalService/NTUSER.DAT	2	1,351	0.15

Unfortunately, we also discovered that the assumption that mtimes always increase does not always hold. Within the M57 data, we observed that the Windows Vista computer (Terry's computer) had one day when more than a thousand keys had their mtimes revert to 2006 and 2008, seemingly a reset. The reverted tallies are shown in Table 4. Unfortunately, the only scenario note for Terry that day was "Continuing rebuilding Terry's hard drive." Further, there were no common hive path prefixes. We are still unsure about how these changes occurred.

Meanwhile, the system clocks that had supported disk images in the RDC were not guaranteed to provide accurate time data. Some clocks were skewed to almost a comical degree: one hive reported an earliest modified key in 1980 and a last shutdown time in 2081. In one case, where a system hive file mtime went back to 1981, its earliest key also went back to 1981. We conjecture – but cannot verify without file carving – that the mtime was allowed to be updated to a smaller value.

Lastly, we observed that the transition from null to non-null mtime does occur – on 418,824 (2.6%) cells in M57. Moreover, no mtime was set to null in the scenario.

8. Code Availability

We continue to integrate our `hivexml` modifications into the Red Hat maintained code base. The development revisions that support this work and the RegXML-SQL conversion tool are available at `users.soe.ucsc.edu/~ajnelson/research/nelson_ifip12/index.html`. The revisions to `dfxml.py` are readily available in Bulk Extractor [10] and Fiwalk [9].

9. Future Work

XML differencing, and more generally tree differencing, is a non-trivial problem. RegXML works well to represent a set of keys and values to be searched, but it is geared towards representing Registry data rather than representing search patterns. CybOX [15], MAEC [16] and WiReD [22] also present a particular key or value to find, or find absent or find persistent after deletion. Unfortunately, none of them formally specify what is interesting about the search data, like a backwards time change or use of low-order bits in the timestamps for steganography. Registry Decoder offers basic functionality in hive differencing, but does not yet provide any reporting on differences [6]. Further research is needed on representing file system differences and patterns. This could enable, say, an antivirus approach where changes against prior hives (such as in system restore points) are analyzed in an automated manner.

Our analysis assumed that paths are unique. Upon discovering that the assumption is invalid, we instead required the uniqueness of paths in the analyzed data. It may be important to analyze a hive that contains ambiguous paths. However, we have not yet decided how to disambiguate cells. Reporting parent addresses may be sufficient, but this would best be tested on ambiguous-path hives and on bin carving experiments, which we did not perform.

Some Registry structures are not presently captured in the `hivexml` implementation of RegXML. In particular, security descriptor ("sk") cells and big data cells ("db") documented by Morgan [18] are not yet represented. The focus of this work was on representing keys, contiguous values, times and differences. We note that, while documentation is available, security descriptors were not a focus of our work and the majority of the M57 and RDC hives did not support big value data. Morgan notes that only hives of version 1.4 or higher may use fragmented data [18]. For all the hives that `hivexml` processed at least partially, the hive version was reported as either 1.3 or 1.5. The M57 corpus had 878 hives of version 1.3 and 158 of version 1.5; RDC had 2,901 and 281 hives of versions 1.3 and 1.5, respectively. We suspect that security descriptors will necessitate at least an in-RegXML index. Given that there are still

undocumented structures within the Registry, RegXML version 0.1 is the most appropriate version at this time, and it represents the first step to full specification. Future versions of RegXML will incorporate extensions that account for the undocumented structures.

10. Conclusions

The Windows Registry provides ample evidence of computer activities. RegXML, the XML schema described in this paper, greatly improves the visibility of hive content. XML fits the tree structure of hives well, and RegXML is especially useful for analyzing a hive and indexing activity footprints. The analysis of the timestamps in two research corpora demonstrates the robustness of the implementation in `hivexml` and the DFXML Python library. We hope that this work will stimulate further research related to Registry exploration and open, searchable reporting.

Note that the views and opinions expressed in this paper are those of the author and do not necessarily reflect the views and opinions of the Department of Defense or the U.S. Government.

Acknowledgements

The author wishes to thank Richard Jones for his advice and review of the `hivex` revisions, Simson Garfinkel for providing guidance on this research effort, and Kris Kearton and Kam Woods for providing information related to the M57 corpus.

References

[1] L. Alvarez, Software designer reports error in Anthony trial, *New York Times*, July 18, 2011.

[2] H. Carvey, *Windows Forensic Analysis*, Syngress Publishing, Burlington, Massachusetts, 2009.

[3] H. Carvey, *Windows Registry Forensics: Advanced Digital Forensic Analysis of the Windows Registry*, Syngress Publishing, Burlington, Massachusetts, 2011.

[4] H. Carvey, RegRipper (`regripper.wordpress.com/regripper`).

[5] Digital Forensics Solutions (`dfsforensics.blogspot.com`).

[6] Digital Forensics Solutions, Registry Decoder: Instructions for Offline Analysis Component, Version 1.1 (`code.google.com/p/registrydecoder/downloads/detail?name=RegistryDecoder-Offline-Analysis-Instructions-v1.1.pdf`), 2011.

[7] L. Garcia, Bulk Extractor Windows Prefetch Decoder, Technical Report NPS-CS-11-008, Department of Computer Science, Naval Postgraduate School, Monterey, California, 2011.

[8] S. Garfinkel, Forensic feature extraction and cross-drive analysis, *Digital Investigation*, vol. 3(S), pp. S71–S81, 2006.

[9] S. Garfinkel, Automating disk forensic processing with SleuthKit, XML and Python, *Proceedings of the Fourth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, pp. 73–84, 2009.

[10] S. Garfinkel, Bulk Extractor (`afflib.org/downloads`), 2011.

[11] S. Garfinkel, Digital forensics XML and the DFXML toolset, *Digital Investigation*, vol. 8(3-4), pp. 161–174, 2012.

[12] S. Garfinkel, P. Farrell, V. Roussev and G. Dinolt, Bringing science to digital forensics with standardized forensic corpora, *Digital Investigation*, vol. 6(S), pp. S2–S11, 2009.

[13] R. Jones, `hivex` – Windows Registry "hive" extraction library (`libguestfs.org/hivex.3.html`), 2009.

[14] Microsoft, FILETIME structure, Redmond, Washington (`msdn.microsoft.com/en-us/library/windows/desktop/ms724284(v=vs.85).aspx`), 2011.

[15] MITRE Corporation, CybOX – Cyber Observable Expression, Bedford, Massachusetts (`cybox.mitre.org`), 2011.

[16] MITRE Corporation, Malware Attribute Enumeration and Characterization, Bedford, Massachusetts (`maec.mitre.org`), 2011.

[17] T. Morgan, Recovering deleted data from the Windows Registry, *Digital Investigation*, vol. 5, pp. S33–S41, 2008.

[18] T. Morgan, The Windows NT Registry File Format (version 0.4) (`sentinelchicken.com/data/TheWindowsNTRegistryFileFormat.pdf`), 2009.

[19] Python Software Foundation, `19.5. xml.parsers.expat` – Fast XML parsing using Expat, Wilmington, Delaware (`docs.python.org/library/pyexpat.html`), 2011.

[20] M. Russinovich, Inside the Registry (`technet.microsoft.com/en-us/library/cc750583.aspx`), 1999.

[21] J. Thomassen, Forensic Analysis of Unallocated Space in Windows Registry Hive Files, Master's Thesis, University of Liverpool, Liverpool, United Kingdom, 2008.

[22] D. White, Tracking computer use with the Windows Registry Datasets, National Institute of Standards and Technology, Gaithersburg, Maryland (`www.nsrl.nist.gov/Documents/aafs2008/dw-1-AAFS-2008-wired.pdf`), 2008.

[23] L. Wong, Forensic analysis of the Windows Registry (`www.forensicfocus.com/downloads/forensic-analysis-windows-registry.pdf`), 2007.

[24] K. Woods, C. Lee, S. Garfinkel, D. Dittrich, A. Russel and K. Kearton, Creating realistic corpora for forensic and security education, *Proceedings of the ADFSL Conference on Digital Forensics, Security and Law*, pp. 123–134, 2011.

[25] R. Zax and F. Adelstein, Faust: Forensic artifacts of uninstalled steganography tools, *Digital Investigation*, vol. 6(1-2), pp. 25–38, 2009.

[26] Y. Zhu, J. James and P. Gladyshev, A consistency study of the Windows Registry, in *Advances in Digital Forensics VI*, K. Chow and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 77–90, 2010.