SYSTEM*– Ultimate Forensic Querying

Author names deleted for review

Abstract This paper describes a novel, XML-based approach towards managing and querying forensic traces extracted from digital evidence. This approach has been implemented in SYSTEM, a prototype system for forensic analysis. SYSTEM systematically applies forensic analysis tools to evidence files (e.g., hard disk images). Each tool produces structured XML annotations that can refer to regions (byte ranges) in an evidence file. SYSTEM stores such annotations in an XML database, which allows us to *query* the annotations using a single, powerful query language (XQuery). SYSTEM provides the forensic investigator with a rich query environment in which browsing, searching, and pre-defined query templates are all expressed in terms of XML database queries.

1 Introduction

A typical digital forensic investigation involves these four phases:

1. media capture (e.g., forensic disk duplication);
2. feature extraction (e.g., parsing file systems, mailboxes, chatlogs, etc.);
3. analysis (browsing, querying, correlating);
4. reporting (writing down findings for court).

This paper addresses two key problems that occur in the feature extraction and analysis phases of a computer system investigation. First, the amount of data to process in a typical investigation is huge. Modern computer systems are routinely equipped with hundreds of gigabytes of storage and a large investigation will often involve multiple systems, so the amount of data to process can run into terabytes. The amount of time available for processing this data is often limited (e.g., because of legal limitations). Also, the probability that a forensic investigator will miss important traces increases every day, because there are simply too many objects to keep track of.

Second, the diversity of the data present on a typical hard disk is overwhelming. A disk image contains a plethora of programs and file formats. This complicates processing and analysis and has led to a large number of special-purpose forensic analysis tools (browser history analyzers, file carvers, file-system analyzers, etc.) While it is clear that the output of different tools can and should be combined in meaningful ways, it is difficult today to obtain an integrated view on the output of different tools. And again, it is quite unlikely that a forensic investigator has both the time and the knowledge to apply all appropriate tools to the evidence at hand.

Our approach to solving these problems involves these key elements:

- a clean separation between feature extraction and analysis;
- a single, XML-based output format for forensic analysis tools;
- the use of XML database technology for storing and querying the XML output of analysis tools.

Feature extraction and analysis are often interleaved and are sometimes seen as a single step. By separating feature extraction from analysis, we can, to a large extent, automate the feature extraction phase. This is essential for dealing with the ever-increasing amounts of input data. The use of XML as an intermediate format allows us to manage the heterogeneity of both the input data and of forensic feature extraction tools. Different tools with a similar function can be wrapped so that they produce similarly structured (XML) output. That output can then be processed by a single analysis tool that no longer has to deal with the idiosyncracies of various input formats. Finally, by storing the XML annotations in a database system, we obtain all the benefits of declarative, general-purpose query languages.

*True system name replaced for review.

To test this approach, we have implemented a prototype system called SYSTEM. SYSTEM automatically extracts features from disk images and stores those features in a high-performance XML database system. The XML database *and* the disk-image data that is referenced by the XML annotations can be accessed through XQuery [1], an XML query language. Since we do not expect all forensic analysts to be XQuery experts, we provide, through a web interface, a number of predefined query templates and standard analyses (e.g., a timeline).

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 gives an architectural overview of SYSTEM. Section 4 describes application areas in which SYSTEM can be useful. Section 5 gives an overview of our initial experiences with the prototype. Finally, Section 6 presents our conclusions and our plans for future work on SYSTEM.

2 Related Work

Our work on SYSTEM is related to several other fields and efforts. First, and perhaps foremost, we are aware of several ongoing projects in the law enforcement community that aim to automate feature extraction for large evidence sets. The need for such automation has been expressed by various authors [3, 4, 5, 6]. Unfortunately, very little is published about these projects. One such project is the Computer Forensic Investigative Toolkit (CFIT) [5], a system developed by Australia's Defence and Science Technology Organization. To the best of our knowledge, CFIT focusses on automatic feature extraction and data visualization rather than the querying of extracted features.

SYSTEM builds on recent advances in information retrieval and on XML-based information retrieval in particular. XML database systems are relatively new and large forensic data sets pose significant challenges to them.

Mainstream commercial toolkits such as Encase and FTK provide a user-friendly interface to a built-in set of forensic analysis tools. EnCase also provides its own scripting language, but no API that allows one to plug in existing, external tools written in a common programming language. SYSTEM differs principally from these tools by its use of a queryable, intermediate data store that isolates feature extraction from analysis. As we will argue in this paper, this offers important benefits.

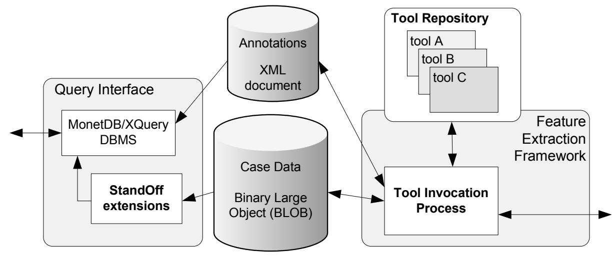

Figure 1: SYSTEM Framework Architecture

3 SYSTEM

The SYSTEM framework consists of three components (see Figure 1). The *tool repository* houses a collection of feature-extraction tools. The *feature extraction manager* orchestrates the invocation of these tools, merges their XML outputs, and stores the result in the *storage subsystem*. The storage subsystem consists of binary large objects that hold raw evidence data and an XML database that holds all extracted features.

3.1 The Feature Extraction Manager

From SYSTEM's perspective, an investigation starts when one or more raw digital evidence items, usually disk images, are fed to the system. Initially nothing is known about the content of these evidence items. The content is simply a single piece of binary data that we will refer to as a Binary Large OBject (BLOB).

The feature extraction manager is responsible for extracting from the input BLOBs as many useful features as possible. It does this by running tools from the tool repository in the correct order and by applying them to the correct inputs. It also tracks which objects have already been annotated by other tools and prevents duplicate annotations.

It is the tasks of individual tools to extract specific features from the BLOBs. A tool will normally operate on one or more byte ranges in the current BLOB set. Such a byte range is called a *region*. A tool extracts features from regions and outputs the extracted data in the form of an XML fragment. An XML fragment produced by a tool may contain references to contiguous byte ranges in the BLOB. Such a byte range is called a *region*. Since a tool's XML output refers back to the BLOB, a tool is also said to *annotate* (parts of) a BLOB. The combination of XML and BLOB is in database literature often referred to as stand-off annotation [7]; the XML describes/annotates the BLOB.

The feature extraction manager collects the XML fragments produced by tools and integrates those fragments into a single, large XML document, which is effectively a tree. It will attach any newly derived annotations to their parents in the current tree.

Annotations produced by one tool can be used as input for other tools; this allows the feature extraction manager to create an increasingly larger set of annotations. This is illustrated in Figures 2. These figures show a case in which three evidence files (A, B, and C) are processed. The initial annotation tree could look like Figure 2. After the feature extraction manager has run a volume detection tool and collected that tool's output, the new annotation tree could look like the second step. Next, SYSTEM will run file-system parsers that take the volumes as input. After file systems have been recognized SYSTEM will run more specific tools such as individual document analyzers, registry analysis, unallocated cluster carving, etc.

For robustness, the feature extraction manager runs each tool in separate processes so that a tool crash will not result in a framework crash. The output of malperforming tools is discarded to avoid corrupted data.

3.2 The Tool Repository

The *Tool Repository* is a set of feature extraction tools. A tool consist of some extraction program and a *wrapper*. A program is wrapped by creating a *tool-executable wrapper* and a *tool input descriptor*. The tool-executable wrapper describes how to invoke the tool and converts the tool's output to XML (see Figure 4). We assume that many existing forensic programs can be made to produce XML by wrapping them. While this is generally true for command-line programs, it is obviously much more difficult to wrap GUI-based programs.

The tool input descriptor is an XQuery expression that selects input for the tool. Specifically, the query selects existing XML fragments from the global, case-wide annotation tree. Input descriptors are restricted to selecting XML nodes that refer to a region in one of the BLOBs. When invoking a tool, the feature extraction manager executes the input descriptor query and passes both the resulting XML fragment and the associated BLOB data to the tool. Table 1 lists several example tools. For each tool, we give its input descriptor query.

Tool name	Rifiuti
Description	Lists recently deleted files by looking at the Recycle Bin log files (usually named ”INFO2”).
Input selection	Selects all files named `INFO2`
Input query	`//file[@name[ends-with(.,"/INFO2")]]`
Tool name	Registry Parser
Description	Analyzes Windows configuration information, e.g. browser settings, installed services, and user details.
Input selection	Selects all files in directory `/Windows/System32/config/` and all files named `NTUSER.DAT`
Input query	`//file[@name[starts-with(., "/Windows/System32/config/") || ends-with(., "NTUSER.DAT")]]`
Tool name	EXIF Extractor
Description	Extracts metadata from images, e.g. a picture's recording date and time and the type of camera used.
Input selection	Selects all files with mime-type ‘image’
Input query	`//file[mime[contains(.,"image")]]`
Tool name	Carving Tool
Description	Uses header/footer signatures to locate images, URLs, zip-files, etc. in unallocated space.
Input selection	Selects any region node that has not been annotated by another tool
Input query	`//*[not(container)]`

Table 1: Input Descriptor examples

SYSTEM distinguishes two types of tools: extraction tools and BLOB-extending tools. Extraction tools read data from a region, interpret it, and produce XML that says something about (parts of) that region. This type of tool is suitable for extracting modest amounts of information from regions. A good example of such a tool is a log parser. All tools listed in Table 1 are extraction tools.

BLOB-extending tools produce not only XML but also raw binary data. This new data is logically appended to the BLOB from which the tool reads its input data. An

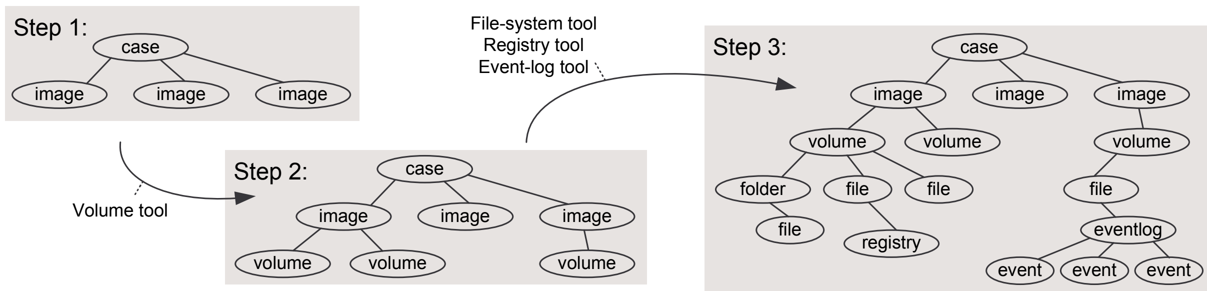

Figure 2: Feature Extraction Example

example of a BLOB-extending tool is a tool that decompresses compressed files. Such a tool would logically append the uncompressed data to a BLOB. In reality, SYSTEM does not physically extend BLOBs; the details of SYSTEM's *virtual* BLOB mechanism are described in the next subsection.

The XML output generated by tools is almost completely free-format: there is no predefined output schema. To obtain an integrated view across the output of different tools, however, it is important that tools adhere to some conventions. All SYSTEM tools that extract timestamped information, for example, produce the same XML tag (`timestamp`) to mark the timestamp. This allows us to obtain a timeline that includes information from multiple tools. Similarly, all file-system parsers use a common set of tags in their XML output.

In their XML output, tools can incorporate *region nodes*. A region node is an XML fragment that refers to a segment of the input BLOB: a region. A region can denote various entities: a file, a sentence, an e-mail message, or even an entire disk; a tool is free to specify any region it can identify. The following region nodes match the example given in Figure 2. Notice how they identify BLOB regions using the XML attributes `start` and `end`.

```
<case id="test-case" date="01-02-2006">
  <image id="1" name="A" start="0" end="15000000"/>
  <image id="2" name="B" start="15000000" end="35000000"/>
  <image id="3" name="C" start="35000000" end="40000000"/>
</case>

<case id="test-case" date="01-02-2006">
  <image id="1" name="A" start="0" end="15000000">
    <volume type="FAT32" start="0" end="10000000"/>
    <volume type="NTFS" start="10000000" end="15000000"/>
  </image>
  <image id="2" name="B" start="15000000" end="35000000"/>
  <image id="3" name="C" start="35000000" end="40000000">
    <volume type="EXT2" start="35000000" end="40000000"/>
  </image>
</case>
```

Region nodes produced by one tool can be selected by the input descriptors of other tools. This way SYSTEM maintains the relationship between annotations and their origin.

3.3 The Storage Subsystem

SYSTEM's storage subsystem stores and gives access to BLOBs and to the XML tree that annotates those BLOBs.

BLOBs are managed by SYSTEM's BLOB manager, which gives access to both the original BLOB input data (usually disk images) and to the logical BLOB extensions produced by tools. A BLOB extension involves a data transformation (e.g., decompression). Any necessary transformation information is provided by the tool that extends the BLOB. Instead of physically extending a BLOB with new data, the virtual BLOB manager stores this transformation and the input and output address ranges involved in the transformation.

Both tools and queries require BLOB access. To provide a transparent interface to these clients, a *virtual BLOB server* has been created which can be asked to retrieve any region from a logical BLOB. Such a request essentially consists of a BLOB identifier, a start offset, and an end offset (see Figure 3). The virtual BLOB server forwards such requests to the BLOB manager which will dynamically apply any transformations necessary to (re)produce the data that has been requested.

This BLOB storage strategy —storing transformations rather than data— allows us to keep storage requirements under control. If necessary, the virtual BLOB server can be extended with a cache, but at present no data is cached.

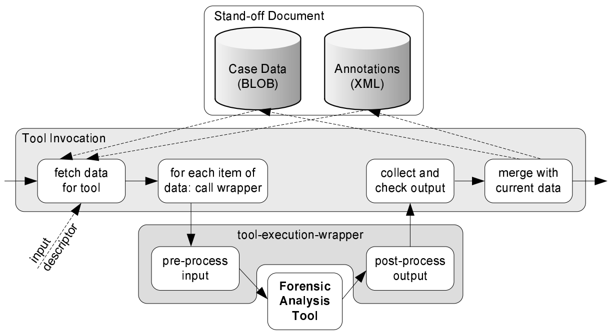

Figure 4: SYSTEM Tool Wrapping

```
for $i in doc("case.xml")//url
where contains($i,"google")
return $i
```

Figure 5: XQuery: returning all ‘Google’ URL’s found by SYSTEM

The XML annotations are stored in MonetDB [2], a high-performance database system that provides several frontends, including an XQuery frontend. All queries in our system are issued to this database system and are expressed in an extended version of the XQuery [1] query language. XQuery is an expressive, general-purpose query language in which XML data can be selected, sorted, grouped, and joined. Figures 5 and 6 show two example queries.

To integrate the XML with the BLOB-data, we defined a additional XQuery functions that link XML elements (region nodes) to the corresponding data in a BLOB. These functions allow us to include BLOB data in query results. Since most of the BLOB access complexity resides in the BLOB manager, the implementation of these functions was relatively straightforward.

A more involved XQuery extension is used to relate regions based on their BLOB positions. There are cases in which multiple tools extract data from the same objects. A URL scanner and an e-mail analyzer, for example, could both annotate the same files, but would extract different features from it. In previous work [citation omitted], we defined *stand-off* extensions through which relationships between overlapping BLOB regions can be expressed. With these extensions, one can, for example, find an e-mail that contains a particular URL, even though

```
let $d := doc("case.xml")
let $f := $d//folder[@name="My Documents"]
let $r := for $i in $f//file
          where $i/mime="application/x-zip"
          order by
            $i/accessed/date descending
          return element "zipfile" {
            $i/@name
          }
return subsequence($r, 1, 20)
```

Figure 6: XQuery: return the names of the 20 last accessed ZIP files located in any ”My Documents” folder or subdirs thereof

```
for $i in doc("case.xml")//file
where ends-with($i/@name,"INDEX.DAT")
return element "file" {
  attribute { "name" } { $i/@name },
  $i/select-narrow::url
}
```

Figure 7: XQuery: return all URL’s in IE history files (INDEX.DAT)

these entities were discovered by unrelated tools. Figure 7 illustrates one of these extensions, the `select-narrow` operator. It selects only those regions that are contained (by BLOB position) in the context region; in this case it selects URL’s inside the INDEX.DAT-files. While we consider the stand-off extensions useful, most of our current queries do not involve these extensions.

3.4 Implementation

Much of our feature extraction framework code consists of Python and Bash shell scripts. The tool collection consists of existing forensic tools (both publicly available tools and tools that we developed in-house. The current collection includes a volume analysis tool, parsers for various file systems (FAT, NTFS, etc.), parsers for various log files (e.g., Windows event log), a file-hashing tool, a link file analyzer, a file carver, and more. Where necessary, these tools were wrapped using scripts. As mentioned, we use MonetDB and an extended version of XQuery for XML storage and access. The BLOB manager and the virtual BLOB server are Python programs.

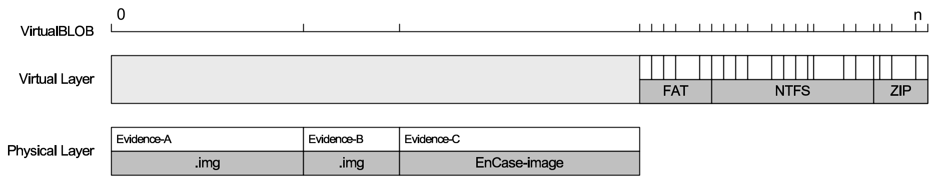

Figure 3: Virtual BLOB Example

Users access SYSTEM applications through a simple web interface. The result of an XQuery can be XML data, which, in turn can be displayed and formatted in a browser using XSL stylesheets. This is an easy way to quickly create a front-end.

4 Forensic Applications

Using SYSTEM, we have implemented a number of small but useful forensic applications. These applications have been tested on several cases; the size of the disk images in these cases ranged from 40 to 240 gigabytes.

The applications cover a range of functions — browsing, searching, knowledge bases— and illustrate the versatility of our query-based approach. Forensic investigators, however, need not be familiar with the XQuery language; they access the SYSTEM applications through simple web interfaces.

4.1 Timeline Browser

Browsing remains one of the principal ways in which forensic examiners discover information. The main type of browsing that is supported by mainstream forensic tools such as EnCase and FTK is file-system browsing. While this is one useful perspective, other perspectives are often equally important and can help reduce the amount of data under investigation. Examples of such perspectives include time and users.

Using SYSTEM, we have implemented a simple timeline browser. Through a web interface, a forensic examiner can select a date/time range of interest. The start and end times are then plugged into the following parameterized XQuery template:

```
let $d := doc("case.xml")
let $all :=
  for $i in $d//%item%/date[
    @unixtime <= %dateupper% and
    @unixtime >= %datelower%]
  order by $i/@unixtime cast as xs:integer
  return $i
let $current :=
  subsequence($all , %start%, %size%)
for $i in $current
  return element "event" {
    $i,
    element "name" { name($i/parent::*) },
    element "subject" {
      name($i/parent::*/parent::*)
    },
    element "description" {
      $i/parent::*/parent::*/(text|message)
    },
    element "file" {
      for $j in $i/ancestor::file
      return element "file" { $j/@* }
    }
}
```

The resulting query selects *all* XML fragments that contain a timestamp. Where a tool such as EnCase can display a time-ordered view of file-system metadata, SYSTEM shows all timestamped information extracted from the input BLOBs by *different* tools. This includes not only file-system metadata, but also entries from chat logs, EXIF information from digital pictures, etc. This way, an investigator obtains an integrated view of the information produced by various extraction and analysis tools. She could see, for example, that movie files are created in the file system at approximately the same time that suspects are discussing such a transfer using a chat program. The results displayed by the timeline browser also include links to the *derivation history* of result objects. By clicking on such a link, the investigator would learn that a chat log entry was extracted from a file (by a chat log parser) that was extracted from a zip archive (by a zip parser), which was discovered in an NTFS file system (by an NTFS parser) that was found in a disk image (by a volume analyzer).

4.2 Photo Search

The photo search application finds digital images that satisfy certain conditions. Figure 8 shows the query form that is presented to users. An investigator can select the camera model that was used to record the image, the date/time on which the recording took place, the resolution of the image, etc.

For brevity, we omit the underlying *query template*. The query constructed from that template combines file-system metadata and EXIF information extracted from digital images. The query produces XML region nodes and some additional metadata. The query result includes image previews which are generated by requesting the relevant regions from the virtual BLOB server.

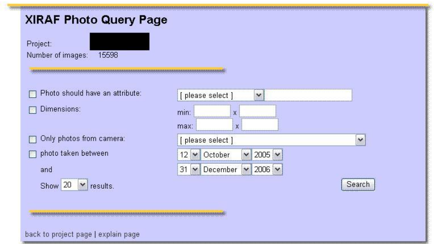

Figure 8: SYSTEM Photo Query

4.3 Child Pornography Detection

SYSTEM can be used to match case information against existing *knowledge bases*. We define a *knowledge base* as structured, relatively static information about a certain subject. A typical forensic example of a knowledge base is a database of hash values of files that have been determined to contain child pornography (digital images or movies). Our child pornography detection program uses SYSTEM to match files present in a case against a hash database that was compiled by the [nationality omitted] police. (Other countries have similar databases.) The hash database has been converted to XML and is preloaded into SYSTEM's XML database. During the feature extraction phase, SYSTEM computes MD5 hash values of all files discovered by file-system tools and other tools. Like all features discovered by feature-extraction tools, these hash values are also stored in the XML database. By pressing a single button, an investigator can execute a query that matches these hash values against values present in the hash database. This results in an overview of known child-pornographic material that is present in the case data.

This application matches all objects that have been marked as as a 'file' against the database. This also includes 'files' discovered by our carving tool, which searches for known headers and footers in the unallocated space of a file system. As a result, and in contrast with similar functions in mainstream tools, the application therefore also discovers child-pornography in unallocated clusters. Moreover, this requires no changes to the query that is used to execute the database match.

5 Discussion

Although it is too early for a formal evaluation of the suitability of SYSTEM in forensic analysis, our early experiences so far have confirmed the intuition that motivated this research, but already highlight a number of issues to be addressed in the next iteration of system design and experimentation.

5.1 Flexible and Powerful Querying

Examples of questions that pop-up in forensic investigation include straight-forward ones like '*When was file X last modified?*', but also high-level information needs of prosecutor or attorney, such as '*Does this computer contain (traces of) child pornography (CP)?*'.

A strong point in our approach is that the SYSTEM architecture offers the opportunity to express the questions popping up in the forensic investigation process as queries in the general-purpose XQuery language. So, the forensic analysis is not limited to a set of predefined investigation patterns. To illustrate the flexibility of this approach, consider a collection of tools for the analysis of file-system information, the computation of MD5 digests, the analysis of log files, carving, and the extraction of EXIF metadata from images.

Assuming that we have represented the CP hash-sets in an XML document called `CP-hashset.xml`, the high-level example question for the existence of CP could then be formulated as a query that checks for existence of files with an MD5 check-sum that exists in the CP database:

```
for $i in doc("case.xml")//file
where some $j in doc("CP-hashset.xml")//md5
      satisfies data($i/md5) = data($j)
return $i
```

Additional queries may extend this collection based on the occurrence of words or URLs that are frequently encountered in CP cases. The resulting set of matching files will be used in follow-up queries, for example to provide a list of `.exe` files not identified by the hash database for additional investigation.

In other words, the declarative nature of the query language enables new ways of processing the data, on-the-fly, as needed for the specific case at hand. By parameterizing the previous CP query by the case's filename (`case.xml` in the example), the same pattern can be reused for different investigations — *independent of how the files in the case have been extracted!* Keeping these queries for later reuse provides a way to capture knowledge of the investigation process in SYSTEM. Essentially, this process extends the set of tools defined in the feature extraction manager with new means to analyse case data. At the moment of writing, the SYSTEM prototype provides already the query patterns to produce timelines, to identify traces of CP (an extension of the query given above), to search photos, and the templates for reoccurring browsing strategies and the collection of summary statistics.

Structural queries are surprisingly useful, even when combining only two tools. For example, after a filesystem-tool and an exif-tool have been applied to the data, the investigator can already create a timeline to display file-activity together with events such as a photo being taken. Or, select files created (or deleted) within two days from a photo being shot. Without SYSTEM, answering such questions always resulted in the need to write a custom script, a time-consuming and error-prone process.

Currently, the query facilities are limited to structural constraints, and very limited keyword matching. Given the activity in defining the XQuery-Fulltext standard however, we expect that this shortcoming can be overcome in the near future. Specifically, the XQuery engine used in our current implementation has announced extensions that provide basic information retrieval functionality.

5.2 Wrapping Tools

A few aspects contribute to the 'wrap-ability' of tools:

- possibility to capture/represent the tool's output;
- possibility to provide tool with correct input;
- amount of overhead introduced;
- the types of tools that can be wrapped (programming interface);
- the amount of time it takes to wrap a tool;
- the behaviour of the system in case the tool produces bad results.

The data model using XML to represent stand-off annotations of a (virtual) BLOB seems to provide a good way to overcome many of the problems with treating binary data inside XML files. Due to the uniform output format of tools, there is no real need to conform to a certain programming interface. As long as a tool accepts BLOB data and/or XML as input, and returns XML (and, if needed, additional binary data) as output, the tool can be easily wrapped, given that the tool does have an interface beyond its GUI. SYSTEM itself currently provides `C`, `Python`, and commandline (`bash`/`cygwin`) interfaces.

Our approach to feeding the tool the correct input is the use of *Input Descriptors*. Although the results will be based on previous tools (which possibly produced wrong results), the missed objects and the false-positive rate seems to be rather low. We do acknowledge that more research should be performed in this area. In particular, we think that knowledge-bases could contribute significantly to improving the quality of the tool input.

The amount of time it takes to wrap a tool depends on its output format. If the tool is already able to produce output in XML format, wrapping could be a matter of minutes, but when the output format of the tool needs to be completely rewritten to become XML, or maybe even the output has to be split into BLOB and XML, wrapping might become more difficult. On the positive side, more and more tools already produce XML as output.

SYSTEM runs its tools in separate processes, thereby avoiding to crash itself whenever a tool crashes. The overhead that this entails is worthwhile. We don't need to focus on the quality of individual tools, and can concentrate on the stability of the feature extraction framework itself.

5.3 Performance Aspects

An important aspect of forensic tools is the need to query cases in interactive time. SYSTEM's runtime performance depends on the size of the resulting XML-document, and the efficiency of the database backend.

5.3.1 Size of XML

In a case for demonstration purposes (2x120GB hd), the extraction framework created a 130MB XML document containing 2.2 million XML elements of which 86,000 file-objects (filesytem objects and carved files). Other annotations included over 460,000 identified date-objects. We expect that the feature extraction framework will be able to extract many more features as new tools are added. Many of the new tools, however, will be file-specific, so the XML document should only grow slowly from this point. Notice that the observed ratio of 240GB to 130MB gives a compression factor of roughly 1000.

Additional experiments have to point out if the current performance figures will scale up when handling up to 10TB of binary data, which is our current target to better represent the real-life forensic investigation.

5.3.2 Extraction Time

The amount of overhead introduced by the extraction phase of SYSTEM is currently rather high: about 3 minutes per tool invocation (depending on the size of the XML in the database). The overhead can be attributed to two major cost factors.

A significant cost results from using an XQuery database system that does not (yet) support updates. Consequently, each tool updates the XML document by merging in its modifications, incurring the cost of copying and parsing all this data (at every tool invocation). Also, this processing strategy allows only to run tools sequentially. As the database backend has recently been extended with update functionality, we expect to reduce this cost factor significantly: by avoiding the repeated parsing and materialisation costs, as well as allowing multiple (independent) tools to run in parallel.

Another contributing factor is that we execute tools in separate processes. This design decision has been taken to shield the extraction manager from failing tools, in practice outweighting the performance penalty incurred.

The following timings are indicative for typical extraction operations.

- Parsing the file systems of a reasonably used and modern computer containing several volumes and a total of about 80,000 files takes approximately five minutes.
- Hashing the content of all files discovered on such a system takes several hours.
- Extracting EXIF information from several thousand JPEG images takes several hours, mainly because of high BLOB server overhead.
- Parsing Windows event log files takes a few seconds.
- Marking files in unallocated space based on header and footer information takes several hours, mainly because this work is carried out by a relatively slow Python script.

5.3.3 Query Processing

To give an indication of SYSTEM's query performance, we provide indicative timings of the applications discussed in Section 4. The timeline browser selects and sorts 500,000+ date objects on the fly in less than five seconds. Likewise, the CP detection programs requires less than five seconds to matching over a 100,000 case file hashes against more than 100,000 database hashes. The chosen database system performs very well at 'join'-queries like the join of two hash-sets. The photo search application requires approximately three seconds to find 1000 images with EXIF information; further selections on a subset of these images are instantaneous.

Except for the database schema itself, there have not been any optimizations in terms of additional indices, and query-caching. Nor have we made an attempt to define a number of views on the data, like a timeline view. Another possibility is to add a middle-tier that could lower the pressure on the database server.

The main bottleneck in the architecture of SYSTEM is the lack of caching, both during feature extraction and querying: neither 'simple' queries for looking up a single node (as used in browsing scenarios), nor requests to the Virtual BLOB server are being cached in the current implementation. Each time (a part of) a file in a file system is requested, its path is looked up in a database, which in turn is converted to a file-object, which is then read.

When rendering large query results, XSLT processing can become the bottleneck, but this can be avoided by disallowing certain queries, and showing only the top-K results for each query.

6 Conclusion and Future Work

This paper has given an overview of the SYSTEM framework. While it is too early to draw definitive conclusions, we feel that the following key benefits of our approach have already surfaced:

- The separation of feature extraction and analysis brings benefits to both phases. SYSTEM extracts features automatically, which is essential when processing large input sets.
- The use of XML as a common, intermediate output format for tools allows us to integrate the output of diverse, independent tools that produce similar information. This allows us to deal both with the heterogeneity present in the input data (e.g., different browser types) and with the diversity of forensic analysis tools. These benefits are demonstrated quite clearly both by our timeline browser and by our child pornography detection program.
- By storing extracted features in an XML database system, we can analyze those features using a single, general-purpose, powerful query language. In addition, we benefit automatically from advances that are made in the area of XML database systems (new query features, improved indexing strategies, etc.)

Our early results with SYSTEM are encouraging, but it is important to realize that the SYSTEM prototype is just that: a prototype. Our experiences indicate clearly that significant additional work is needed to turn SYSTEM into a production system for forensic analysis.

First, we are continuously expanding our tool set. An increasing number of forensic tools produce XML output and this is of obvious benefit to SYSTEM. We are presently looking into integrating the output of TULP2G [8], an open-source mobile phone analysis tool, into SYSTEM. In a large-scale investigations, many mobile phones can be seized. If information extracted from those phones is converted to a uniform XML format, then SYSTEM can be used to issue queries that span all phones. In addition, information extracted from mobile phones can be matched against information from other sources, including disk images.

Second, we are looking into augmenting SYSTEM with knowledge bases that contain expert knowledge about specific types of digital traces. Good examples are the locations of important Windows registry keys, the locations of useful log files and characteristic file-header information. Such knowledge can be captured either in the form of static XML subdatabases or in the form of a query database.

Third, we are working on adding full-text indexing to SYSTEM and the corresponding query functions. The current prototype implementations supports only structural XML queries. Adding full-text indexing will enable content-and-structure (CAS) queries which will obviously increase SYSTEM's query capabilities.

References

[1] S. Boag, D. Chamberlin, M.F. Fernndez, D. Florescu, J. Robie, and J. Siméon. XQuery Specification 1.0.

[2] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. ACM SIGMOD Conference*, June 2006.

[3] F. Buchholz and E.H. Spafford. On the role of file system metadata in digital forensics. *Digital Investigation*, 1:298–309, 2004.

[4] B. Carrier and E.H. Spafford. Getting physical with the digital investigation process. *International Journal of Digital Evidence*, 2(2), 2003.

[5] G. Mohay, A. Anderson, B. Collie, O. De Vel, and R. McKemmish. *Computer and Intrusion Forensics*. Artech House, 2003.

[6] A. Sheldon. The future of forensic computing. *Digital Investigation*, 2:31–35, 2005.

[7] H.S. Thompson and D. McKelvie. Hyperlink semantics for standoff markup of read-only documents. In *Proceedings of SGML Europe '97*, Barcelona, Spain, May 1997.

[8] J. van den Bos and R. van der Knijff. TULP2G - An Open Source Forensic Software Framework for Acquiring and Decoding Data Stored in Electronic Devices. *International Journal of Digital Evidence*, 4(2), 2005.