Digital Investigation 11 (2014) S46–S56

Contents lists available at ScienceDirect

Digital Investigation

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

Cooperative mode: Comparative storage metadata verification applied to the Xbox 360

Alex J. Nelson [a, b, *], Erik Q. Steggall [a], Darrell D. E. Long [a]

[a] University of California, 1156 High Street, Santa Cruz, CA 95064, USA
[b] Prometheus Computing, LLC., 6514 Twin Lake Drive, New Market, MD 21774, USA

Keywords:
Differencing
Forensic strategies
Tool measurements
Embedded systems
Xbox 360
Digital forensics XML

A B S T R A C T

This work addresses the question of determining the correctness of forensic file system analysis software. Current storage systems are built on theory that is robust but not invincible to faults, from software, hardware, or adversaries. Given a parsing of a storage system of unknown provenance, the lack of a sound and complete analytic theory means the parsing's correctness cannot be proven. However, with recent advances in digital forensic theory, a measure of its incorrectness can be taken.

We present *FSNView*, an *N-Version programming* utility. *FSNView* reports exhaustively the metadata of a single disk image, using multiple storage system parsers. Each parser reports its perspective of the metadata in Digital Forensics XML, a storage language used recently in a study on differential analysis. We repurpose the tools used for studying the changes in file systems from time to the changes in file systems from perspective. The differences in metadata summaries immediately note bugs in at least one of the tools employed. Diversity in tools and their analysis algorithms strengthens the analysis of a storage subject.

We apply file system differencing to study the external storage of the Microsoft Xbox 360 game console. The console's storage serves as an exemplar analysis subject; the described strategy is general to storage system analysis. The custom volume management and new-though-familiar file system are features typical to an embedded system analysis. Two open-source utilities developed solely for analyzing this game console, and a third developed for general file system forensics, are extended to compare storage system metadata perspectives. We present a new file system and revisions to the DFXML language, library, and differencing process, which were necessary to enable a principled, automatic evaluation of storage analysis tools.

Introduction

Storage forensics is one of the oldest branches of the digital forensics field. Yet, even today, the question of complete storage analysis correctness from case to case goes unanswered. There are two underlying reasons for this uncertainty:

Unknown data veracity The storage data an analyst receives may be the results of arbitrary corruption, from deliberate actions or operating system faults.

Unknown tool veracity The behaviors of a storage analysis tool on every possible case of storage corruption, even no corruption at all, may be unknown.

Due to the sheer scale of data involved in any case, analysts must employ storage-interpreting tools. Unfortunately, the correctness of any of these tools is not yet possible to formally validate. If a validated storage analyzer

* Corresponding author.
E-mail address: ajnelson@cs.ucsc.edu (A.J. Nelson).

http://dx.doi.org/10.1016/j.diin.2014.05.004
1742-2876/

existed, it would implement a formal specification for parsing file systems and volume systems (Chen and Avizienis, 1978). However, no formal specification exists for the at-rest recorded state of file systems commonly in use, especially as they are employed by arbitrary operating system versions. Formally specifying and verifying a file system's run-time behavior would lessen fault chances to the chances of bit rot occurring, but formalizing file system behavior is an active research area (Keller et al., 2013). If the trends of Linux kernel development generalize to other operating systems, file system code is a significant, volatile portion of kernel code (Lu et al., 2013), leaving little chance for formalizing. In short, *validating* storage system analysis is far from a solved problem.

Validation is not so pressing a concern in most investigations. Some file systems require enough collective attention that much documentation, reverse engineering, and well-supported analysis software exists, such as with FAT and NTFS (Carrier, 2005; Hibshi et al., 2011). However, it is possible the subject data may be in a format not supported by an analyst's regular array of tools. For instance, undocumented, embedded systems sometimes use common file systems (Fang et al., 2012), and sometimes develop their own with no developer reference available. In the custom-storage cases, specialized analysis tools may exist, even sometimes from hobbyist developers; but the correctness of that software is completely unknown.

This work addresses the problem of *verifying* storage analytic tool results, versus *validating* them. Instead of proving that the results one has attained are correct, instead a mechanism is developed to show that the results are consistent.

We employ an automated peer review technique among multiple tools, by having them *vote* on file system metadata. When compared, tools' metadata interpretations denote the portions of storage systems under dispute by any set of parsers. Metadata is the analytic target because higher-level results of a tool inevitably depend on the tool's interpretation of the subject metadata. Inconsistencies between tools highlight tool bugs, but do not hinder analysis if the disputed namespace portions—generally far smaller than the whole subject namespace—are irrelevant to a case.

Metadata voting, using a time-tested software engineering method, only became possible with recent developments in the digital forensics literature.

- The language Digital Forensics XML (DFXML) expresses the metadata of a storage system (Garfinkel, 2012). It includes some succinct data summaries, like file hashes.
- Garfinkel et al. (2012) described a general strategy for data differencing in several digital forensic applications. They describe applying the differences to data, in order to learn about the data.
- The actual voting method comes from *N-Version programming* (Chen and Avizienis, 1978). Solutions to a problem are measured by consensus of multiple tools. Those tools implement the same formal specification, and their answers, expressed in a mutually agreed-upon format, determine the correct or approximate answer to the problem.

DFXML, with its differentiability, is the first data interchange format that makes practical storage tool comparison possible. This is an essential advancement to the analytic process, where review of multiple tool results is cumbersome at best presently. For instance, it is simple to verify that a key evidentiary file in an investigation—*e.g.* a contraband image—is found with multiple tools, if they each say the same file is at a particular location. However, with mechanically enumerated differences in results, it is also possible and practical to identify that no exonerating evidence was missed. In the same example, some malware may hide in obscured files, designed to download that exact evidentiary data to frame a suspect.

Practical tool comparison also enables development of new capabilities, and refinement of existing capabilities. This paper demonstrates such a knowledge sharing. This work analyzes the storage system of the Xbox 360 gaming device. Its commercial forensic support is not broadly announced—for instance, one anonymous vendor we spoke with reserved the necessary interface components for people who called to ask. We found reverse-engineering documentation and some open source analytic tools, one a hobbyist project, that performed analysis of the console's specialized file system, XTAF. We extended the tools to generate DFXML, turning them into valuable contributors on understanding console research data we generated. They also became "peer reviewers" of an implementation of Xbox storage analysis within The SleuthKit (TSK) (Carrier, 2003). TSK's FAT implementation, in turn, could inform odd corner cases of XTAF behavior, as the two file systems are quite similar.

While the XBox 360 storage system is the exemplar subject of this work, the theory and techniques described herein apply to any storage system. For less-understood, less-supported embedded systems, the techniques can give some robustness to metadata results, even if many of the results must be derived from little-known or custom-written utilities. For better-understood storage systems, comparative metadata verification is an important step to answering an over-decade-old call to openly review low-level extraction engines of forensic tools (Carrier, 2002).

This work makes these contributions of theory:

- A technique for measuring and reporting discrepancies in single-subject, multiple-investigating-tool analysis of storage systems.
- The use of DFXML and differential analysis as automated peer review of different tools on the same data.
- Methods to turn open-source storage analysis programs of unknown correctness into practical and valuable extra "eyes" on a problem.

This work makes these contributions of practice:

- A new userspace file system, *UPartsFS*, which expands the applicability of storage tools that only understand a single partition as input.

- Extensive refinements to the DFXML language, its Python interface, and its differencing process.
- An XML schema that precisely describes the structure of DFXML documents.
- A storage analysis comparator utility, *FSNView*.
- An Xbox 360 storage analysis extension to The SleuthKit.
- An analysis of the structures of the Xbox 360 storage system, verified by three tools.

Outline

Section 2 provides the background on previous work that helped begin multi-tool analysis automation. Sections 2.1 and 2.2 cover the comparison algorithm and summary language. Section 3 moves from the analytic strategy to the analytic subject of this paper, by describing technical characteristics and challenges of the Xbox 360 external storage system. From the similarities to the FAT file system, it becomes evident why this storage system serves well as an exemplar file system for a tool comparison study.

For experimental materials, Section 4 describes data produced, to both evaluate file system parsers, and to investigate game console artifact generation. Sections 5, 6 and 7 describe the various implementations, including updates to the background materials that made the file system comparator tool *FSNView* possible.

For evaluation, Section 8 measures the consensus of the adapted tools on the generated disk images. Section 8.1 describes some of the types of artifacts that could be found amongst the data, focusing on artifacts that were improved by knowledge of the tested file system.

For present and future context, Section 9 notes related work, and Section 10 describes directions for future research. Finally, Section 11 concludes.

Background: theory and frameworks

Until recently, comparing storage parsers was a specialized task, difficult to perform between n (>2) tools. Since many tools to which users are accustomed are based on graphical user interfaces (Hibshi et al., 2011), comparing results would mean performing a significant amount of extra work:

1. Analysis of the data would need to be performed with each of the n tools, each producing some sort of mechanically readable output.
2. Each of the n outputs would need to be compared with each of the other $n-1$. This step can be represented as a complete directed graph (complete digraph), where nodes are tool results and edges are comparisons. The graph is directed because tool comparisons are not guaranteed to be symmetric. Because the digraph is complete, there are $n(n-1)$ comparisons performed.
3. Each comparison technique employed, *e.g.* mapping CSV columns from one format to another, should also undergo evaluation for soundness and completeness of differential results. For instance, the set of files added comparing in one direction should equal the set of files removed if comparing in the reverse direction. This can mean up to $n(n-1)$ evaluations performed – one evaluation per distinct comparison technique.

These steps mean it takes an impractical amount of work to compare multiple storage parsers, if each output and comparison requires extra attention. However, implementing a single common, differentiable format reduces the amount of work necessary: the $n-1$ comparisons in step 2 can happen with one method. The rest of this section describes a comparison-ready output format for storage parsers, and the prior work in analyzing differences with that format.

Digital forensics XML

DFXML is an XML syntax that summarizes storage systems. It is capable of expressing partitions of a storage device, file metadata, hash sets, and provenance of its documents (Garfinkel, 2012). Its principal value is in sufficiently describing storage systems for most processing and analytic needs, without requiring access to the storage system itself. For files, DFXML encodes metadata one expects of an inode, and some data summary attributes, like content checksums. The XML is straightforward to generate by programmatically writing strings. A Python library has been available and used for years to read DFXML documents (Garfinkel, 2009, 2012).

Most decisions a forensic analysis tool will make hinge on its understanding of the storage metadata. DFXML makes it possible for tools to both enumerate and thoroughly compare their fundamental views of a storage subject (Garfinkel et al., 2012).

Differential analysis

Differential analysis is a fundamental act in scientific work: comparing two samples of data, and enumerating differences as one can and needs. Garfinkel et al. previously described a general differencing strategy in forensics (Garfinkel et al., 2012). In their work, the focus was on exploring differences in various types of subject data, and though the generating tool was not of specific importance, DFXML arose frequently in their examples.

The general strategy considered two images, a *preimage* and a *postimage*. Each image would contain a set of *features*; *e.g.* for a disk, features could be distinct email addresses found with text search, or files found with a file system walk. The strategy formalized a set of change primitives that would transform the feature set of the preimage to the post's. More granularly, an algorithm following the strategy would operate on, or simply return, sets of features that:

- Were new or removed;
- Had a name or some names added or removed (which together would capture renames);
- Changed location within the image; or
- Had some other non-identity metadata changed.

The strategy summarized differential analytic tasks from several domains. However, file systems require extra attention that we will describe in Section 5.3.

Analytic subject: Xbox 360 and the XTAF file system

Though the Xbox 360 is generally regarded as a gaming console, it has the capabilities of a modern personal computer, and thus is a potential target for forensic analysis. The Xbox 360 is networked, multi-user, and uses standard 2.5″ SATA hard disks. The console's file system, XTAF, is similar to the well understood FAT-16 and FAT-32. XTAF was only meant to be configured for a small family of storage devices, centered around a single console device. As a result, it requires and records less configuration information than found in FAT.

The Xbox 360 manages its disk partitions with a custom volume manager. Partition locations are hard-coded, and the last partition's size is a function of the hard disk size. Fortunately, the partitions are easy to detect with some simple rules. Much of the file system data structure information has been previously documented by others, such as on the Free60 Wiki (Free60, 2012).

XTAF is not the only file system the console understands. The console is capable of reading FAT from USB thumb drives, including some sold to distribute system updates (Hryb, 2010). Microsoft appears to have abandoned the use of XTAF after the 360 generation, opting to use NTFS in their Xbox One console (iFixit, 2013). While the value of analyzing XTAF systems will dwindle with decreasing 360 popularity, analyzing it has provided valuable lessons.

The XTAF file system

XTAF is a file system similar to FAT (Free60, 2012; Xynos et al., 2010). File allocations are still handled with block chains in a File Allocation Table. Directory entries and inodes are a single, conflated concept. The superblock is also a simple structure. This section summarizes the customizations XTAF has taken on since FAT, by others' reports and manual inspection.

XTAF has simplified some of the fields that FAT used in its directory entries. The most significant changes in interpreting the directory entry data are how timestamps and names are handled. Timestamps use one format for modification, access, and creation times, with a granularity of two seconds. Names are a single 42-byte field, that are only designed to store ASCII characters. Hence, if suspected directory entries with non-ASCII characters are reported by a tool, they are more likely a bug in the parser than an alternate encoding.

Partition management

The Xbox 360's external drive does not maintain a table of partitions. Instead, the partition locations and sizes are simply hard-coded, and one must scan a drive to determine precisely how many are present. As many as six partitions appear on an XTAF drive, at the offsets given in Table 1. These partitions are found by scanning the drive for the string `XTAF`, checking the byte offset for sector alignment, and checking if its surrounding bytes appeared to be a superblock.

Table 1
XTAF partitions. Sectors are 512 bytes. Descriptions were supplied by the Free60 wiki (Free60, 2012). Similarities to the FAT variant are determined by the size of the file system being over approximately a gigabyte.

Offset (sectors)	Length (sectors)	Like FAT-*n*	Partition description
1024	4194304	FAT32	System cache (encrypted)
4195328	4587520	FAT32	Game cache (encrypted)
8782848	422272	FAT16	System Extended
9205120	262144	FAT16	System Extended 2
9467264	524288	FAT16	Backwards-compatibility for prior generation of Xbox software
9991552	Rest	FAT32	User data

Cryptanalysis of encrypted drive partitions is out of the scope of this paper.

Designing XTAF data

From related studies on the Xbox 360's storage, we found hints of information available in a console one would not typically expect.

- In a string inspection of the drive, without access to file system details, Bolt noted that he had managed to recover a URL [16, Figure 7.22]. What he did not note in the text was that the URL was embedded within the text pattern of an HTTP header, indicating the possibility that network traffic may be recorded on the drive. Bolt's console actions were clearly described, encouraging experimental repeatability.
- There has been some controversy over the possibility of transaction details, particularly credit card numbers, being recoverable from used Xbox 360 consoles (Podhradsky et al., 2011; Protalinski, 2012).

We designed standard console behavior data as a series of disk images of two consoles. Several personas perform actions using only the standard GUI: sending messages; watching a movie; playing single-player games; and playing multi-player games between the two consoles, which required paying for Xbox Live "Gold" level accounts. Account payments were done two ways through the game console: with redeemable Xbox Live cards; and low-limit, prepaid, non-refillable debit cards. Disk images were taken at every step believed to create a distinct type of disk artifact, such as downloading a player profile. At completion of the disk image sequences, the debit card balances were exhausted to prevent future charges. Actions were photographed with smart phones from the same manufacturer and cellular network, in an attempt to keep consistent clocks external to the consoles. In all, a longitudinal disk image series of two disks resulted, containing artifacts of standard game console behavior, and possibly payment artifacts.

An extensive analysis of these data, including artifact discovery, file attribution, and user ascription, is out of scope of this work on comparative metadata verification. We discuss some of the disk artifacts that are enhanced with knowledge of the file system.

Improving DFXML and differencing for tool evaluation

The prior work on using DFXML to compare different storage system states can translate into comparing different storage system parsers. However, the adaptation was not straightforward, and required improving or in some cases re-implementing the background material (Section 2).

Formalizing the DFXML language

DFXML was an ad-hoc standard, with documents generated by two C/C++-based tools and a handful of Python scripts. The *dfxml.py* Python library was designed to process DFXML files, allowing users to work with streams of `fileobject` objects in a SAX framework (Garfinkel, 2009). Informally, what this library could read would be considered "valid" DFXML. However, this validity was only scoped in practice to Python programs that included *dfxml.py*.

In order for other tools to implement DFXML with a clearer definition of "valid," we developed an XML schema that formalizes the DFXML language (DFXML Working Group, 2013). Now, a tool that purports to generate DFXML can validate its output against the DFXML schema with the *xmllint* utility (Veillard, 1999).

Implementing new DFXML Python bindings

DFXML lacked a mutative object model. Analyzing storage systems with DFXML required generating DFXML files and then reading them with *dfxml.py*. To improve flexibility, we implemented a type-safe, object-oriented model in Python. Now, objects have mutable properties; are serializable and de-serializable, conformant to the DFXML schema when reading and writing; and are capable of doing extensive equality comparisons.

To illustrate the API change on accessing file attributes, consider a `fileobject` with an inode number. Before, the API would allow accessing with "`fi.inode()`"; but modification, or initially setting, required serializing an XML string and re-reading it to access the property later. Alternatively, a non-public object interface would allow setting values, but the fragility of such a programming approach should be immediately evident. Now, a `FileObject` (capitalized) can have its inode number set, with "`fi.inode = 5555`". Python property getters and setters (Python Software Foundation, 2014) make type-checking transparent.

This new API is of particular benefit to Python-based file system parsers. They can create entire DFXML documents, `DFXMLObject`s, using these bindings. A `DFXMLObject` contains a list of `VolumeObject`s and/or `FileObject`s, and `VolumeObject`s contain a list of `FileObject`s. Also, each object type can now perform a verbose difference with its type, which enabled us to modify the previous differential analysis model of Garfinkel et al. (Garfinkel et al., 2012).

Modularizing idifference.py

Formerly, comparing DFXML was specialized to a single utility, *idifference.py* (Garfinkel et al., 2012). *idifference* was designed to report the differences of two disk images, or their DFXML summaries. To inspect file system parsers, we initially re-purposed *idifference* to compare DFXML of different tools interpreting the same image. However, the tool was performing too many of the differential analysis steps, to the detriment of its analytic capability. We split the functionality of *idifference* into distinct tools, enabling more general analysis. This modularizing induced a new intermediary analytic state, which represents file system differences as DFXML. This opens new applications of the DFXML language.

Comparing tools using DFXML requires a more complete enumeration of file attribute differences. The *idifference* utility produced a practical enumeration of differences, primarily file names, timestamps, and checksums. However, a tool's report of a file can vary in any of the file's known attributes. These new features in particular enabled re-implementing comparison:

- The `FileObject` equality operator is recursively defined as evaluating whether each of the properties of two `FileObject`s are equal. Some properties, *e.g.* the `id` property defined as unique to any run of a tool, are ignored. Comparing tools, the `id` would declare almost all files as unequal if the tools don't extract the exact same features, in the same order.
- A more general comparison operator returns the set of properties that differ.
- To deal with unimportant and likely too-numerous differences, the difference set can be determined as a relevant change or not as a run-time configuration of the called application.
- `FileObject`s also support holding a baseline object as the property `original_fileobject`. Essentially, each `FileObject` tracks the differences of itself, once matched.

To analyze changes to an entire storage system, no special logic is necessary to analyze files' property differences — aside from choosing some property changes to ignore. The task of storage differencing is now mapping files from one metadata summary to the other.

The *idifference.py* utility performed two tasks, which are now separated using the new `Object` bindings:

1. Identify the relevantly-changed files.
2. Report on the changes. This report is a tabular format derived from the in-memory file objects.

Using the new `Object`s, these tasks are now two utilities that communicate with differentially-annotated DFXML as an intermediary state. The first step is replaced with a program that maps old files to new, *make_differential_dfxml.py*. This program's *matching algorithm* simplifies in the case of FAT or XTAF, thanks to these file systems not having a concept of "inode" beyond directory entries. The algorithm is quite similar to the general differential analysis strategy described by Garfinkel et al. (2012):

1. For two disk images or their DFXML summaries, assign one to be the *preimage*, and the other the *postimage*. Define four dictionaries, $files_{pre,alloc}$, $files_{pre,unalloc}$,

files$_{post,alloc}$, and *files*$_{post,unalloc}$. Each of these dictionaries will store file references, keyed by the pair (*partition_number*, *path*). The *unalloc* dictionaries store sets of references for each key, as deleted content is free to be encountered multiple times.
2. For each allocated file encountered in the *preimage*, store a reference to the file in *files*$_{pre,alloc}$.
3. Store unallocated files of the preimage in *files*$_{pre,unalloc}$.
4. For each allocated file encountered in the *postimage*, see if its (*partition_number*, *path*) pair is in *files*$_{pre,alloc}$. If it is, the file has been matched to one in the preimage. Remove the file from the preimage's dictionary. If it isn't, store the file in the *files*$_{post,alloc}$ dictionary, using the partition and path pair as a key.
5. Store unallocated files of the postimage in *files*$_{post,unalloc}$.

The matching algorithm diverges from the general differential analysis strategy here, due to file systems storing potentially multiple deleted versions of files. These steps match the remnants:

6. For all the sets in the *unalloc* dictionaries that contain just one reference, match any of the files that have matching partition numbers and paths. If there are multiple references, the directory entry byte address may serve to match; but that is on the assumption that deleted content remains in place until it is garbage-collected.
7. At this point, *files*$_{pre,alloc}$ lists files that were present, but have since been deleted. If there is a *single* matching entry remaining in *files*$_{post,unalloc}$, the files are matched, allowing us to infer some intermediary state between the two images if analyzing a single disk at two times. That is, before the file was deleted, other metadata of the file may have changed.
8. All remaining files in the *pre* and *post alloc* dictionaries are considered deleted and added, respectively.
9. All remaining files in the *pre* and *post unalloc* dictionaries are considered deleted and added, respectively; and possibly too ambiguous to match.

Matching in XTAF or FAT is simpler than in other file systems that separate inodes and directory entries. The original *idifference* algorithm (Garfinkel et al., 2012) used inode numbers as well as paths, for matching and rename detection. While the reimplementation of *idifference.py* still uses inode numbers, it also includes matching steps that obviate their need in case a file at a path is given a new inode number. Inode number reassignment happens when a program updates files by using a temporary "working" file, performing saves by renaming the working file to the original file path. For XTAF, an "inode number" is computed the same as The SleuthKit computes a FAT "inode number," as a function of the directory entry's byte address within the file system. Hence, the inode number is effectively ignored in the matching algorithm when analyzing FAT or XTAF. A discussion of matching in other file systems is beyond the scope of this paper.

The matching program outputs all of the differences as an annotated DFXML stream. File properties that change receive a `delta:changed_property="1"` attribute; new files, a `delta:created_file="1"`; and similarly for deleted, modified, and metadata-changed files. Partitions receive similar annotations. Partition annotations are particularly useful when entire partitions can be missed due to a tool crashing.

The second *idifference.py* step is replaced with another program that summarizes that DFXML stream in the same format as the *idifference* report, *summarize_differential_dfxml.py*. Files are listed in path or modification time order, grouped as new, deleted, modified, or merely having changed metadata. However, *summarize_differential_dfxml.py* is no longer the logical end of differential analysis with the *idifference* program flow. Several benefits arise from this separation of duties and use of intermediary state:

- *The differences are now completely enumerable and distributable.* Previously, limitations of the reporting format, coupled with difficulties caused from missing file identification information, lead the *idifference* developers to ignore files marked unallocated. Producing a DFXML stream means these unallocated files can once again be included for analysis, though matching them presents new challenges. Most of these challenges are out of the scope of this paper, for reasons discussed in Section 10.
- *The differences can be analyzed separately from producing them.* The *idifference* report format is one way to review changes in a storage system. However, using differentially-annotated DFXML opens storage changes up to wide-scale, metadata-level analysis, in line with some of the original goals of DFXML (Garfinkel, 2009).
- Including the `original_fileobject` is a useful mechanism for distributing file sets as well. A manifest of files extracted from a disk image can accompany those files, including the original `fileobject` data locations used for extraction. This is important for any forensic analysis process that relies on file extraction with follow-up analysis performed by other tools.

Byte runs to note more than content locations

Discrepancies in DFXML between tools often meant needing to look at metadata structures. We propose additional `byte_runs` elements to denote metadata addresses as well as data addresses, using these *facets*:

`data` Regular file content, which is the `byte_runs` element as presently used.
`inode` The byte addresses of the inode and inode-indirect blocks in POSIX file systems; or the MFT entry, including non-resident extensions, in NTFS.
`name` The byte addresses of the directory entry that references the inode.

These additional byte runs would greatly benefit analyzing deleted files and finding mis-matches in differencing. They would be particularly valuable for comparing tool results. Ideally, multiple tools run on the same data will

identify the same file references, which may include many deleted files that lack much identifying information. Byte addresses of their metadata would serve to match them between the tools' results. Fortunately, in XTAF and FAT, metadata address information is proxied by the formulaic "inode number" definition used in TSK, a function of the byte address of the directory entry.

Programs extended for DFXML comparison

The tools in this section were selected according to these criteria:

- Open-source, with permissive modification licenses.
- Language, preferring C and Python.
- Execution in userspace.

Uxtaf

uxtaf (Ladan, 2007) is a userspace implementation of an XTAF file system parser in C. The developer used this as a reference implementation for a BSD kernel module for interacting with the file system. The utility provides a simple shell for navigating a file system, and reporting information similar to the `stat` command. DFXML generation was implemented by copying and extending logic from the shell's `cd`, `ls`, and `cat` functions, outputting XML with string printing.

Py360

py360 (Arkem, 2011a) is a file analysis suite for the Xbox 360, implemented in Python. The suite provides a file system walker, and modules for analyzing file formats particular to the gaming console. Instead of print statements, we added functions to populate `Objects` of the new DFXML bindings.

The SleuthKit

The SleuthKit (TSK) (Carrier, 2003) is a file system forensic suite, capable of analyzing many partition and file systems. The TSK libraries powered the initial implementation of DFXML, via *fiwalk* (Garfinkel, 2009).

TSK did not have an XTAF implementation at the time this research commenced. However, its FAT implementation could be adapted to read XTAF data structures, carrying forward over a decade of parser stabilizations.

Tools developed for DFXML comparison

UPartsFS: extending single-partition file system parsers

Specialized tools designed for single-file-system analysis frequently lack two important features: analyzing file systems embedded in a partition system; and bindings to compressed formats. Some of the tools selected for comparative analysis of the Xbox 360's storage could only analyze a partition image. One could operate in a "disk mode," by jumping to the disk's primary data partition; but it skipped over up to five other partitions doing so.

Initially, each XTAF parser was instrumented to also detect whether it was analyzing a partition or a disk, conditionally translating within-file-system offsets to within-disk. Implementing partitions turned out be a significant engineering effort that would be impractical to repeat for every single-file-system tool one would want to compare. We realized through multiple implementations of the same logic that any amount of volume detection, even mostly hard-coding XTAF internal regions, is an issue that is simply out of scope of analyzing an individual partition image. However, the logic would need to be written at least once, with an interface fit for any file system analysis tool.

We implemented *UPartsFS*, a userspace file system that reads a disk image's partition table, and presents the partitions as large, virtual files. *UPartsFS* is based on FUSE (Szeredi, 2003) and The SleuthKit's volume parsing system. Using this file system, file system parsers neither need to deal with partition logic themselves, nor rely on a kernel-level partition table reader. Tools also do not need to implement bindings to disk image formats. This last point may seem menial at first glance; but without image format bindings and partition delineations, a tool one needs to use could require a spare hard drive and machine-work day per case. In storage analysis, that is a significant distraction.

DFXML files produced using the files of *UPartsFS* are scoped to single partitions. A concatenating script, *cat_volumes.py*, takes the resulting files and creates DFXML of a whole-disk walk, including translating file system offsets to disk offsets.

FSNView: a single-data, multi-interpreter DFXML reporter

This research culminates in the *FSNView* utility. This tool produces two tables for a set of tools and a single disk image:

1. A summary of the number of files and partitions encountered by each tool. Differences in file counts foreshadow the differences found between the tools.
2. A summary of the file attribute difference counts, when each tool is compared to each other tool. Each comparison is performed twice, switching which is considered the "Baseline" image.

Evaluating multi-tool analysis of Xbox 360 storage

Using the DFXML-based reporting system *FSNView*, we were able to identify development bugs as we melded code bases from three independent open source projects and scattered, incomplete documentation into an Xbox 360 storage parser in The SleuthKit.

Tables 2 and 3 were generated with the *FSNView* utility. Table 2 provides an overview of the relative recovery statistics of each parser used in the test. Table 3 shows a pairwise comparison of all the tools, each comparison performed twice in case test results would turn out asymmetric. The statistics of Table 3 illustrate differences in metadata.

Table 2
Summary processing statistics for three DFXML-producing XTAF analyzers.

	fiwalk	py360	uxtaf
Partitions processed	5	6	4
Allocated directories	65	58	56
Allocated files	293	231	231
Allocated other	0	0	0
Unallocated directories	1	14	8
Unallocated files	2	15	11
Unallocated other	15	0	0

The counts of these tables show many different issues, each a valuable lesson in tool behaviors. Among the bugs made evident from non-zero counts, these were particularly instructive:

- An early iteration of Table 3 showed disagreement between two tools on the timestamps of a single file. On closer inspection, we found this was actually a misidentification: a file in a directory had a prior, marked-unallocated version in the same directory as its current, allocated version with different timestamps. One tool had misidentified the old version as the current, allocated version.
- We did not expect any renames, given the definition of inode numbers in XTAF. However, early iterations of Table 3 reported over a hundred renames when comparing tools to *uxtaf*. *uxtaf* was truncating the last file extension, *e.g.* cutting "`.xex.tmp`" to "`.xex`." Inspection of the code showed *uxtaf*'s general name extraction strategy was to rely on the file name length recorded as a field of the directory entry. However, the recorded directory name length does not include the last extension, which could be extracted by reading up to the first `0xFF` byte. *uxtaf* was the only tool relying on recorded name lengths, causing massive file mismatches that have since been corrected.
- Another early issue we resolved was interpreting the correct end of FAT cluster chains. *py360* initially defined a single end-of-file cluster value, `0x0FFFFFFF` with masking appropriate to the XTAF variant. However, *fiwalk* and *uxtaf* used special FAT entry values inherited from the original FAT file systems, including the EOF marker range `0x0FFFFFF7`–`0x0FFFFFFF`. *py360* had the less-correct definition, as some of those high cluster values appeared in our data, and were clearly incorrect as data cluster pointers.
- Some of the differences come from subtle interpretation issues. Directory size is always recorded as 0 in XTAF. Following the FAT chain of clusters allows the size to be interpreted differently, and a decision must be made on whether to include the entire cluster after the allocated directory entries. This precision is necessary to determine a rule for hashing the directory's contents. These tools do not agree on a consistent interpretation of directory contents.
- *fiwalk* reported more files than the other tools, even accounting for a partition *uxtaf* failed to parse. However, some of these additional files were clear false positives, generated by reading the overwritten contents of an unallocated directory. This bug, presently unresolved, occurred multiple times in *fiwalk*, each time pointing to the same cluster address as content. We initially found this bug by observing a path, consisting of the empty string, being reported multiple times. Non-unique paths had previously indicated bugs in reporting file allocation status.

Table 3
Counts of file system parsing discrepancies between three Xbox 360 analysis utilities, *fiwalk* (fi), *uxtaf* (ux) and *py360* (p3). Counts are in differences from the first program's DFXML output to the second program; *e.g.*, "missed files" indicates the number of files the first program found that the second didn't. "Files" includes directories, unless otherwise noted. These statistics are from 1 disk image.

Differences in …	fi-p3	fi-ux	p3-fi	p3-ux	ux-fi	ux-p3
Additional files	13	3	53	0	55	12
Allocated	5	3	37	0	37	2
Unallocated	8	0	16	0	18	10
Missed files	53	55	32	12	22	0
Allocated	37	37	5	2	3	0
Unallocated	16	18	27	10	19	0
Renamed files	0	0	0	0	0	0
Allocated	0	0	0	0	0	0
Unallocated	0	0	0	0	0	0
Metadata						
SHA-1 (dirs)	6	60	1	55	52	55
SHA-1 (files)	8	2	2	8	1	8
SHA-1 (other)	0	0	0	0	0	0
Filesize (dirs)	62	61	53	0	52	0
Filesize (files)	0	0	1	0	1	0
Filesize (other)	0	0	0	0	0	0
Modified time	0	0	0	0	0	0
Access time	0	0	0	0	0	0
Metadata change time	0	0	0	0	0	0
Creation time	0	0	1	0	1	0
Data byte runs (dirs)	1	61	0	64	52	64
Data byte runs (files)	1	0	2	0	1	0

This experience affirms that there is value in seeing a thorough enumeration of metadata differences across the entire file system. When run individually, each of these tools appeared to be working properly. However, upon inspecting the contrast of their outputs it becomes obvious that each tool has unique flaws due to the differing strategies, algorithms, and conventions each has for parsing and reporting a file system. Programmatically comparing their metadata reports has helped to uncover significant and subtle parser bugs that would have gone missed using any one in isolation. The tools can be fixed and adjusted to make their results converge; but this would merely remove tool discrepancies on known data, and should not be mistaken for the impossible ideal of completely correcting tool behavior.

Artifact recovery

An exhaustive analysis of the artifacts discoverable on the Xbox 360 is beyond the scope of this paper. However, some of the questions that motivated the data design could be answered with DFXML-based techniques, raising the

importance of verifying the file system metadata perspective.

We used Bulk Extractor (Garfinkel, 2013) to investigate artifact creation. Bulk Extractor is a pattern-matching engine, capable of matching regular expressions; identifying data in several encodings, such as GZip-compression; and recursively scanning for patterns within encoded regions. Data that matches a scanner is reported with an address on the disk, possibly noting decoders used to translate content. An accompanying script, *identify_filenames.py*, matches the address to files found with a DFXML mapping. Among Bulk Extractor's built-in scanners are URL recognizers, which could supply context for HTTP headers; and a credit card data scanner.

The view of the file system analysis affected whether some artifacts could be attributed to files. Bulk Extractor could not find on the drive any of the used credit card numbers. One false positive arose that did not match an initially-reported number (Podhradsky et al., 2011). HTTP headers were found on the drive. The *identify_filenames.py* script identified cookie content stored in files named `BrowserData`. *fiwalk* and *uxtaf* found the same file paths; *py360* had missed the files. Those same files include XBox Live account names, making it possible to attribute browser activity to an account on the console. Some timeline analysis was made difficult by the system clock resetting to November, 2005 if the console was left unplugged.

Overall, this experience with DFXML and Bulk Extractor demonstrates there are multiple benefits to creating new DFXML generators. Tool integration was one of DFXML's early goals. Tool testing has now come about as another form of integration.

Related work

Practices

Hibshi et al. surveyed forensic practitioners at a conference, providing some insight into tool use practices and motivations (Hibshi et al., 2011). Their survey results include discussion on multi-tool use for verifying results, but no statistics on frequency. One key point that has influenced tool comparability is that the majority of users prefer interacting with tools through GUIs. If there were instead a preference for script-driven interactions in the field, it is probable that an analytic framework for tool comparison, or even tool differencing, would have emerged sooner. However, the product development needs follow the demand, for the GUI, and for performance. Evaluation of the extraction engines, called for over a decade ago (Carrier, 2002), has not been in such a strong demand.

The National Institute of Standards and Technology (NIST) released a draft standard on deleted file content reporting (NIST, 2009). This standard is complementary to our approach. NIST publishes standards, against which other tools are measured. In contrast, testing tools against each other is a continuous test of both the tools and behaviors in data. Some of the discovered behaviors in either may later be codified in standard tests such as NIST's.

Nelson used real-system data to evaluate a forensic tool in development. He analyzed realistic-use (Woods et al., 2011) and real-use (Garfinkel et al., 2009) disk image data as he developed a syntax for representing the Windows Registry as an XML format, RegXML (Nelson et al., 2012). The differential analysis algorithm examples include RegXML differencing (Garfinkel et al., 2012).

Other tool comparison in storage forensics

In storage forensics research, tools have a common set of denominators when compared. Usability is an important factor in product development and presentation (Hibshi et al., 2011), though somewhat independent of baseline functionality such as disk imaging (Carrier, 2002). Single-datum, multi-tool storage analysis is a topic of infrequent publishing outside of NIST's testing team (National Institute of Standards and Technology, 2003).

Casey and Stanley evaluated two tools in a "live" incident response scenario. The design of the scenario precluded consistent disk imaging. Their evaluation tested capabilities in common between the tools on the same compromised system. Manson et al. (2007) evaluated multiple tools on a set of disk images, with an emphasis on verifying results using multiple tools. They performed some *qualitative* file system analysis with each tested tool, *e.g.* verifying that deleted and encrypted files were found and clearly identified, respectively. They did not perform *quantitative* file system analysis, *e.g.* matching all deleted files among tools. However, their report did compare many other factors of their employed software. Grispos et al. (2011) compared two acquisition methods and several analysis tools on a single Windows Mobile device, extensively comparing recovery results for over eighty generated smartphone artifacts. Some of their tools demonstrated difficulty with the phone's file system, TFAT, another variant of FAT.

Single-datum, multi-tool testing has occurred also for *anti*-forensics. Geiger evaluated six commercial tools designed to eliminate evidence (Geiger, 2005). His work contributed heuristics for detecting if any of the tools were used, and evaluating their coverage of artifact removal. The emphasis was on differences in tool behaviors, somewhat like in this work. However, aside from discrepancies in identifying relevant features, Geiger reached the additional goal of developing some fingerprints of tool actions.

Xbox analysis

As an undocumented system, ripe for reverse engineering, the Xbox 360 has attracted analytic attention. In forensic analysis, some have covered acquisition and high-level disk artifacts (Bolt, 2011; Xynos et al., 2010). The *py360* project produced documention on some of the file structures (Arkem, 2011b). The console has seen the hardware re-purposed, for alternate gaming or operating systems (LibXenon.org, 2011).

The 360 generation of Xbox did not require the same feats needed to analyze the storage of the original Xbox (Huang, 2002). RAM extraction is as dependent on software exploits for the original generation (Rabaiotti and Hargreaves, 2010) as it is for the 360, which requires techniques such as the "Reset glitch hack" (Free60, 2012).

Future research

Today, the amount of forensic research disk images, freely available to immediately download, is small. The total count of distinct machines behind disk images that have any records of ground truth is presently under forty, per the listings on the Forensics Wiki (2014). It would benefit the forensic community at large if more data were available for tool testing. However, generating realistic data includes risks, such as identifying aspects of the researchers; vulnerabilities in their environment; or even the users in the data set, despite anonymization attempts (Narayanan and Shmatikov, 2008). Generating data more restricted in form (Adelstein et al., 2005) carries less risk, but provides less analysis opportunity. Acquiring data without provenance—such as the Real Data Corpus (Garfinkel et al., 2009)—provides the greatest system entropy and most varied analysis, but at near-total loss of most ground truth. We made a realistic data set of game console usage, including real financial transactions. It was a logistical challenge to divide the actions of the fake console personas from the real people required to transact real money. It would benefit the forensics community to have further experiences, guidelines, and adversarial research on creating forensic data sets. This may be aided by a survey of experiences in creating and releasing forensic data. The more research data are available, the more tools and techniques can be tested and refined.

These experiences with measuring tool differences showed areas where the language of storage summarization, even abstracted past DFXML, can be improved. Analyzing XTAF was quite simplified by not needing to deal with inodes separate from directory entries. Analyzing other file systems will require clarifying some programmatic definitions of storage artifacts, such as the identity of deleted files.

The definition of "correct" results in general storage analysis is a controversial issue, due to the possibilities of encountering genuinely ambiguous artifact results. For instance, a corrupted FAT directory can store two files with the same name, byte-for-byte. Research in identifying classes of ambiguities in storage would help to reduce the confusion possible from interpreting inconsistent file system states, and possibly bring us closer to the ideal of complete, agreeable knowledge of a storage system's active state and fragments. Again, the more realistic and real data are available, the more enriched the vocabulary of storage analysis will be.

Conclusion

Digital storage forensics is reliant on storage-parsing tools. If one of these tools has errors, users and developers deserve to hear about them. Unfortunately, because tool result comparison has to date been so manually intensive, these errors are less likely to be discovered, let alone reported. If the tool produces results in a well-structured format, such as DFXML, this work shows it is easier to automate discovery of tool discrepancies. Further, small projects, well-versed in analyzing a single, exotic file system, can contribute to any analysis by implementing the same well-structured format.

Storage metadata is fundamental to any investigation. Hence, total reporting on the metadata improves confidence in all storage analysis. We may not be able to fully, formally validate arbitrary results soon; but we can now verify with ease.

Resource availability

Xbox 360 disk images supporting this work are available at Digital Corpora. *FSNView*, *UPartsFS*, and our modifications to the projects mentioned are available on Github. The page at this URL lists the locations of data, code repositories, and versions used to support this work: https://users.soe.ucsc.edu/~ajnelson/research/nelson_dfrws14/ .

Acknowledgments

We thank D J Capelis, Jim Whitehead, Simson Garfinkel, Kam Woods, Cora Frantz, Thomas Matthew, Sébastien Bourdon-Richard, Sarah Martin, Barbara Guttman, Mary Laamanen, and the anonymous reviewers for providing valuable discussion and input for this paper.

This publication results from research supported by the Naval Postgraduate School Assistance Grant/Agreement No. N00244-12-1-0066 awarded by the NAVSUP Fleet Logistics Center San Diego (NAVSUP FLC San Diego). The views expressed in written materials or publications, and/or made by speakers, moderators, and presenters, do not necessarily reflect the official policies of the Naval Postgraduate School nor does mention of trade names, commercial practices, or organizations imply endorsement by the U.S. Government.

References

Adelstein F, Gao Y, Richard III GG. Automatically creating realistic targets for digital forensics investigation. In: DFRWS '05; 2005.

Arkem. Xbox 360 file specifications reference http://www.arkem.org/xbox360-file-reference.pdf; 2011b.

Arkem. py360 user guide http://www.arkem.org/py360-user-guide.pdf; 2011a.

Bolt S. XBOX 360 forensics: a digital forensics guide to examining artifacts. 1st ed. Syngress; 2011.

Carrier B. Open source digital forensics tools: the legal argument. Tech. Rep., @Stake; 2002.

Carrier B. File system forensic analysis. Addison-Wesley; 2005.

Carrier B. The Sleuth Kit (TSK) & autopsy: open source digital forensics tools http://sleuthkit.org/; 2003 [last accessed 27.01.14].

Chen L, Avizienis A. N-version programming: a fault-tolerance approach to reliability of software operation. In: FTCS-8; 1978.

Fang J, Jiang Z, Chow K-P, Yiu S-M, Hui L, Zhou G, et al. Forensic analysis of pirated Chinese Shanzhai mobile phones. In: Peterson G, Shenoi S, editors. Advances in digital forensics VIII, IFIP advances in information and communication technology. Berlin Heidelberg: Springer; 2012. pp. 129–42.

Free60. http://free60.org/; 2012 [last accessed 04.02.14].

Garfinkel SL. Automating disk forensic processing with SleuthKit, XML and Python. In: SADFE '09; 2009. pp. 73–84.

Garfinkel S. Digital forensics XML and the DFXML toolset. Digital Investigation 2012;8(3–4):161–74.

Garfinkel S. Digital media triage with bulk data analysis and bulk_extractor. Computers & Security 2013;32:57–72.

Garfinkel SL, Farrell P, Roussev V, Dinolt G. Bringing science to digital forensics with standardized forensic corpora. In: DFRWS '09, Quebec, Canada; 2009.

Garfinkel S, Nelson AJ, Young J. A general strategy for differential forensic analysis. In: DFRWS '12; 2012.
Geiger M. Evaluating commercial counter-forensic tools. In: DFRWS '05; 2005.
Grispos G, Storer T, Glisson WB. A comparison of forensic evidence recovery techniques for a windows mobile smart phone. Digital Investigation 2011;8(1):23–36.
Hibshi H, Vidas T, Cranor L. Usability of forensics tools: a user study. In: IMF '11; 2011.
Hryb L. USB memory support for the Xbox 360 coming April 6th [last accessed 27.01.14], http://majornelson.com/2010/03/26/usb-memory-support-for-the-xbox-360-coming-april-6th/; 2010.
Huang A "bunnie". Keeping secrets in hardware: the Microsoft XBox™ case study. Tech. Rep. AIM-2002-008. Massachusetts Institute of Technology; 2002.
iFixit. Xbox one teardown http://www.ifixit.com/Teardown/Xbox+One+Teardown/19718; 2013 [last accessed 27.01.14].
Keller G, Murray T, Amani S, O'Connor L, Chen Z, Ryzhyk L, et al. File systems deserve verification too!. In: PLOS '13; 2013.
Ladan R. uxtaf https://github.com/rene0/xbox360; 2007 [last accessed 11.02.14].
LibXenon.org. http://libxenon.org/; 2011 [last accessed 05.02.14].
Lu L, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Lu S. A study of Linux file system evolution. In: FAST '13; 2013.
Manson D, Carlin A, Ramos S, Gyger A, Kaufman M, Treichelt J. Is the open way a better way? Digital forensics using open source tools. In: HICCS '07; 2007.
Narayanan A, Shmatikov V. Robust de-anonymization of large sparse datasets. In: IEEE S&P '08; 2008.
National Institute of Standards and Technology. NIST computer forensic tool testing program http://www.cftt.nist.gov/; 2003 [last accessed 06.02.14].
Nelson A. RegXML: XML conversion of the Windows registry for forensic processing and distribution. In: Peterson G, Shenoi S, editors. Advances in digital forensics VIII, IFIP advances in information and communication technology. Berlin Heidelberg: Springer; 2012. pp. 51–65.
NIST. Active file identification & deleted file recovery tool specification (draft for comments) http://www.cftt.nist.gov/DFR-req-1.1-pd-01.pdf; 2009 [last accessed 09.02.13].
DFXML Working Group. dfxml_schema https://github.com/dfxml-working-group/dfxml_schema/tree/v1.1.0; 2013 [last accessed 27.01.14].
Podhradsky AL, D'Ovidio R, Casey C. Identity Theft and Used Gaming Consoles: recovering personal information from Xbox hard drives. In: AMCIS '11; 2011.
Protalinski E. Microsoft investigating used Xbox 360 credit card hack http://www.zdnet.com/blog/security/microsoft-investigating-used-xbox-360-credit-card-hack/11260; 2012 [last accessed 27.01.14].
Python Software Foundation. Built-in functions http://docs.python.org/3.3/library/functions.html/#property; 2014 [last accessed 25.01.14].
Rabaiotti JR, Hargreaves CJ. Using a software exploit to image RAM on an embedded system. Digital Investigation 2010;6(3–4):95–103.
Szeredi M. File system in user space README http://www.stillhq.com/extracted/fuse/README; 2003.
Veillard D. The XML C parser and toolkit of Gnome http://xmlsoft.org/; 1999 [last accessed 27.01.14].
Forensics Wiki. Forensic corpora http://www.forensicswiki.org/wiki/Forensic_corpora; 2014 [last accessed 12.02.14].
Woods K, Lee C, Garfinkel S, Dittrich D, Russel A, Kearton K. Creating realistic corpora for forensic and security education. In: ADFSL '11; 2011.
Xynos K, Harries S, Sutherland I, Davies G, Blyth A. Xbox 360: a digital forensic investigation of the hard disk drive. Digital Investigation 2010;6(3–4):104–11.