



DFRWS 2016 Europe — Proceedings of the Third Annual DFRWS Europe

Forensic analysis of cloud-native artifacts



Vassil Roussev*, Shane McCulley

Greater New Orleans Center for Information Assurance (GNOCIA), University of New Orleans, New Orleans, LA, 70148, USA

A B S T R A C T

Keywords:

Cloud forensics
Google docs format
Reverse engineering
Cloud-native artifacts
kumodocs
kumodd

Forensic analysis of cloud artifacts is still in its infancy; current approaches overwhelming follow the traditional method of collecting artifacts on a client device. In this work, we introduce the concept of analyzing *cloud-native digital artifacts*—data objects that maintain the persistent state of web/SaaS applications. Unlike traditional applications, in which the persistent state takes the form of files in the local file system, web apps download the necessary state on the fly and leave no trace in local storage.

Using *Google Docs* as a case study, we demonstrate that such artifacts can have a completely different structure—their state is often maintained in the form of a complete (or partial) log of user editing actions. Thus, the traditional approach of obtaining a snapshot in time of the state of the artifacts is *inherently* forensically deficient in that it ignores potentially critical information on the evolution of a document over time. Further, cloud-native artifacts have no standardized external representation, which raises questions with respect to their long-term preservation and interpretation.

© 2016 The Authors. Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

Introduction

The traditional business model of the software industry has been *software as a product* (SaaS); that is, software is acquired like any physical product and, once the sale is complete, the owner can use it as they see for an unlimited period of time. The alternative—*software as a service* (SaaS)—is a subscription-based model, which did not start becoming practical until the emergence of widespread Internet access some two decades ago. Conceptually, the move from SaaS to SaaS shifts the responsibility for operating the software and its environment from the customer to the provider. Technologically, such a shift was enabled by the growth of the Internet as a universal means of communications, and was facilitated by the emergence of the web browser as a standardized client user interface (UI) platform.

The traditional analytical model of digital forensics has been client-centric—the investigator works with physical evidence carriers, such as storage media or integrated compute devices (e.g., smartphones). On the client (or standalone) device it is easy to identify where the computations are performed and where the results/traces are stored. Therefore, research has focused on discovering and acquiring every little piece of log and timestamp information, and extracting every last bit of discarded data that applications and the OS may have left behind.

The introduction of *Gmail* in 2004—the first web 2.0 application in widespread use—demonstrated that all the essential technological prerequisites for mass, web-based SaaS deployments have been met. The introduction of the first public cloud services by Amazon in 2006 enabled *any* vendor to rent scalable, server-side infrastructure and become a SaaS provider. A decade later, the transition to SaaS is moving at full speed, and the need to understand it forensically is becoming ever more critical.

This massive technological shift presents a *qualitatively* new challenge for forensics; one that cannot be addressed by minor adjustments to tools and practices. Specifically,

* Corresponding author.

E-mail addresses: vassil@roussev.net (V. Roussev), smcculle@my.uno.edu (S. McCulley).

the SaaS model disrupts the familiar client-centric world—both code and data are delivered over the network on demand, and thus become moving forensic targets. For example, a *Google Docs* document shows up as nothing more than a hyperlink on the local disk; the actual content is downloaded and made available for editing only in the browser.

In this work, we approach the problem by going directly to the data source—the service provider—using both public and private APIs and data structures. This leads to a new approach that, we believe, is a preview of how cloud forensic tools will be built.

Related work

The primary focus of previous work on cloud storage forensics has been on adapting the traditional application forensics approach to finding client-side artifacts. This involves blackbox differential analysis, where before and after images are created and compared to deduce the essential functions of the application. Section (Client-based data acquisition & analysis) summarizes representative work in this area.

Section (API-based data acquisition & analysis) presents a more recent alternative, which seeks to avoid the limitations of client acquisition by working with the provider's API.

Client-based data acquisition & analysis

Chung et al. (2012) analyzed four cloud storage services (*Amazon S3*, *Google Docs*, *Dropbox*, and *Evernote*) in search of traces left by them on the client system that can be used in criminal cases. They reported that the analyzed services may create different artifacts depending on specific features of the services, and proposed a process model for forensic investigation of cloud storage services based on the collection and analysis of artifacts of the target cloud storage services from client systems. The procedure includes gathering volatile data from a Mac/Windows system (if available), and then retrieving data from the Internet history, log files, and directories. On mobile devices they rooted an Android phone to gather data and for iPhone they used iTunes information like backup iTunes files. The objective was to check for traces of a cloud storage service exist in the collected data.

In Hale (2013), Hale analyzes the *Amazon Cloud Drive* and discusses the digital artifacts left behind after an Amazon Cloud Drive account has been accessed or manipulated from a computer. There are two possibilities to manipulate an Amazon Cloud Drive Account: one is via the web application accessible using a web browser and the other is a client application provided by Amazon and can be installed on the system. After analyzing the two methods, he found artifacts of the interface in the web browser history, and among cached files. He also found application artifacts in the Windows registry, application installation files on default location, and an SQLite database used to keep track of pending upload/download tasks.

Quick and Choo (2013) analyzed *Dropbox* and discuss the artifacts left behind after a *Dropbox* account has been accessed, or manipulated. Using hash analysis and keyword

searches they try to determine if the client software provided by *Dropbox* has been used. This involves extracting the account username from browser history (Mozilla Firefox, Google Chrome, and Microsoft Internet Explorer), and the use of the *Dropbox* through several avenues such as directory listings, prefetch files, link files, thumbnails, registry, browser history, and memory captures. In follow-up work, Quick and Choo (2014) use a similar conceptual approach to analyze the client-side operation and artifacts of *Google Drive*, and provide a starting point for investigators.

Martini and Choo (2013) have researched the operation of *ownCloud*, which is a self-hosted file synchronization and sharing solution. As such, it occupies a slightly different niche as it is much more likely for the client and server sides to be under the control of the same person/organization. They were able to recover artifacts including sync and file management metadata (logging, database and configuration data), cached files describing the files the user has stored on the client device and uploaded to the cloud environment or vice versa, and browser artifacts.

API-based data acquisition & analysis

The client-side acquisition approaches discussed so far have one big assumption in common; namely, that *all* the data artifacts of interest *can* be acquired from the client. The problem is that this is *not* true in the general case, and is likely to be not true in the common case. As illustrated on Fig. 1, the client can no longer be considered the original source of the data. Rather, it maintains a *cached* version that is *likely* incomplete (in more ways than one) and potentially out of date.

Considering the above functional architecture, there are three major lapses in client-based acquisitions of cloud-hosted data:

Partial replication. The most obvious problem is that none of the clients working with a cloud storage account may have a complete copy of the data. Currently, cloud storage providers offer selective replication so that devices with less local storage (smartphones) are not overwhelmed. Going forward, as data accumulates online, it would become increasingly impractical (and unnecessary) to maintain a complete local copy. Amazon already offers unlimited storage at \$60/year, and that is a lot of data to clone locally. From a forensic standpoint, a client-based acquisition is blind to the overall picture, and has no means to guarantee completeness.

Artifact revisions. Most storage services provide automatic revision tracking that keeps copies of previous

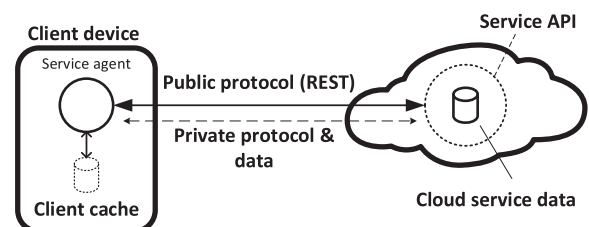


Fig. 1. SaaS application architecture.

versions of user files. However, these are not present in the client cache, and are only recalled on demand (via a web interface). Forensically, this is another dimension along which a client-based acquisition is blind and incomplete.

Cloud-native artifacts. Web applications rarely store persistent state on the client devices. There are some notable exceptions, such as the caching of user credentials and offline operation while disconnected, but the norm is that all data is hosted on the server side. This gives rise to the concept of *cloud-native* data, which we use to describe internal data structures used by SaaS applications that are not stored on the client persistently. This clearly creates a problem for client-side methods as the data of forensic interest is not present locally.

In Roussev et al. (2016), we argued that the only way to fully address the first two aspects of the problem is to utilize the service provider's official API. We developed a cloud drive acquisition tool, *kumodd*, which can perform full API-based acquisition of four major cloud providers: Google Drive, Dropbox, Box, and Microsoft OneDrive. Our tool can enumerate and download *all* files associated with one of the above accounts and *all* of their revisions. Further, it can acquire *snapshots* of cloud-native artifacts in standard formats, such as PDF, via the API.

However, *kumodd* cannot possibly acquire cloud-native artifacts in their original form because they are not part of the official API supported. For example, a Google Docs document is represented as a hyperlink and there are no means in the API to acquire the content. From a developer's point of view, such artifacts are internal data structures and there is no necessity to provide access to them via the API. In effect, there is a private communication protocol between the client and server components of the web app that is used alongside the public one (Fig. 1).

The remainder of this discussion focuses on the acquisition and analysis of cloud-native artifacts, using Google Docs as a case study.

Understanding Google Docs

For the purposes of this discussion, we use Google Docs to refer to the entire suite of online office, productivity and

collaboration tools offered by Google. We use *Documents*, *Sheets*, *Slides*, etc., to refer to the individual applications in that suite.

In all likelihood, the very first cloud-native tool with forensics applications is *DraftBack* (draftback.com): a browser extension created by the writer and programmer James Somers, which can replay the complete history of a *Documents* document. The primary intent of the code is to give writers the ability to look over their own shoulder and analyze how they write. Coincidentally, this is precisely what a forensic investigator would like to be able to do—rewind to any point in the life of a document, right to the very beginning.

In addition to providing the in-browser playback (using the *Quill* open source editor (Chen and Mulligan) of all the plaintext editing actions—either in fast-forward, or real-time mode—*DraftBack* provides an analytical interface which maps the time of editing sessions to locations in the document (Fig. 2).

This can be used to narrow down the scope of inquiry for long-lived documents. Somers' work, although not motivated by forensics, is an example of SaaS analysis that does not rely on trace data resident on the client—all results are produced solely by (partially) reverse engineering the web application's private protocol. Assuming that an investigator is in possession of valid user credentials, or such are provided by Google under legal order, the examination can be performed on the spot; any spot with a browser and an Internet connection.

These observation served as a starting point of our own work in an effort to build a true forensic tool that understands the needs of the investigative process.

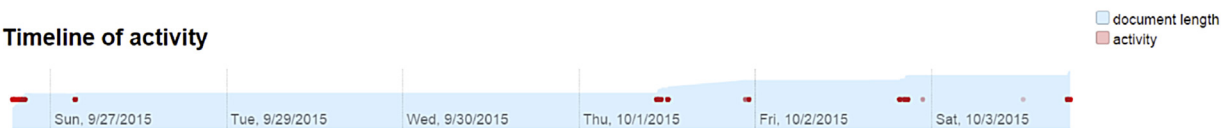
Documents

In 2010, Google unveiled a new version of Google Docs (Google, 2010a), allowing for greater real-time online collaboration. The new *Documents* editor, named *kix*, handles rendering elements like a traditional word processor—a clear break from prior practices where an editable HTML element was used. *Kix* was “designed specifically for character-by-character real time collaboration using

Document data/stats summary*:

Total time: 03:00:42s

Timeline of activity



Where in the document were the changes?

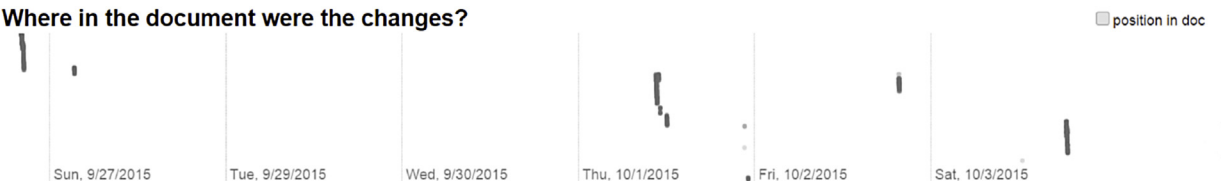


Fig. 2. *DraftBack* analytical interface (edited for size).

operational transformation" (Google, 2010b). (*Operational transformation* is a concurrency management mechanism that eschews preventive locking in favor of reactive, on-the-fly resolution of conflicting user actions by transforming the editing operation to achieve consistency.) Another important technical decision was to keep a detailed history of document revisions that allows users to go back to *any* previous version; this feature is available to any collaborator with editing privileges.

Google's approach to storing the revisions is also different than most prior solutions—rather than keep a series of snapshots, the complete history of editing actions—since the creation of the document—is retained. When a specific version is needed, the log is replayed from the beginning until the desired time; replaying the entire log yields the current version. This design means that, in effect, there is no delete operation that irrevocably destroys data, and that has important forensic (and privacy) implications.

To support fine-grain revisions, as well as collaborative editing, user actions are pushed to the server as often as every 150 ms, depending on the speed of input. In collaboration mode, these fine-grained actions, primarily insertions and deletions of text and images, are merged on the server end, and a unified history of the document is recorded. The actions, potentially transformed, are then pushed to the other clients to ensure consistent, up-to-date views of the document.

The number of *major revisions* available via the public API corresponds to the major revisions shown to user. Major style changes seem to prompt more of those types of revisions; for example, our working document where we keep track of our experiments has over 5100 incremental revisions but only six major one. However, the test document we used for reverse engineering purposes, has 27 major revisions with less than 1000 incremental ones. It appears the passage of time since last edit also plays a role. Starting a new session does not seem to be enough to trigger a new major revision.

The internal representation of the document, as delivered to the client, is in the form of a JSON object called *changelog*. The structure is deeply nested but contains one array per revision, with most elements of the array containing objects (key–value pairs). Each array ends with identifying information for that revision as follows: an epoch timestamp in Unix format, the Google ID of the author, revision number, session ID, session revision number, and the revision itself.

Each time the document is opened, a new session is generated, and the number of revisions that occur within that session are tracked. Some revisions, such as inserting an object, appear as a single entry with multiple actions in the form of a multiset, which contains a series of nested dictionaries. The keys in the dictionary are abbreviated (2–4 characters)—almost certainly for performance reasons—but not outright obfuscated.

The changelog contains a special *chunked snapshot* object, which contains all the information needed to create the document as of the starting revision. The length of the snapshot varies greatly depending on the number of embedded *kix* objects and paragraphs; it has only two entries (containing default text styles) for revisions starting at 1.

For any revision with text in the document, the first element of the snapshot consists of a plaintext string of all text in the document, followed by default styles for title, subtitle, and headings *h1* through *h6*, language of the document, and first paragraph index and paragraph styles. The next several elements are all *kix* anchors for embedded objects like comments or suggestions, followed by a listing of each contiguous format area with the styles for those sections that should be applied, as well as paragraphs and associated IDs used to jump to those sections from a table of contents.

Fig. 3 shows the representation of a minimal example document; one in which the text "Test document" has been typed. In this case, the snapshot (starting on line 3) contains the state of the document before the very last update.

```
{ "changelog":
  [ [ { "ty": "is", "s": "ent", "ibi": 11 }, 1453673743519, "04167822183031715453", 7, "3581dbec97c438a0", 5, null ] ],
  "chunkedSnapshot":
    [ [ { "ty": "is", "s": "Test docum", "ibi": 1 },
      { "ty": "as", "sm": { "hs_h1": { "sdef_ts": { "ts_fs": 18.0, "ts_fs_i": false },
        "hs_h2": { "sdef_ts": { "ts_fs": 14.0, "ts_fs_i": false },
        ...
        "hs_h6": { "sdef_ts": { "ts_bd": false, "ts_bd_i": true,
          "ts_fg": "#666666", "ts_fg_i": false, "ts_it": true, "ts_it_i": false },
          "ei": 0, "st": "headings", "si": 0, "fm": false },
      { "ty": "as", "sm": { "lgs_l": "en", "ei": 0, "st": "language", "si": 0, "fm": false },
      { "ty": "as", "sm": { "ps_al_i": true, "ps_awa_i": true, "ps_ifl_i": true, "ps_il_i": true, "ps_ir_i": true,
        "ps_klt_i": true, "ps_kwn_i": true, "ps_ls_i": true, "ps_sa_i": true, "ps_sb_i": true, "ps_sm_i": true },
        "ei": 11, "st": "paragraph", "si": 11, "fm": false },
      { "ty": "as", "sm": { "ts_bd": false, "ts_bd_i": true, "ts_bgc": null, "ts_bgc_i": true,
        "ts_ff": "Arial", "ts_ff_i": true, "ts_fg": "#000000", "ts_fg_i": true, "ts_fs": 11.0,
        "ts_fs_i": true, "ts_it": false, "ts_it_i": true, "ts_sc": false, "ts_sc_i": true,
        "ts_st": false, "ts_st_i": true, "ts_un": false, "ts_un_i": true, "ts_va": "nor", "ts_va_i": true },
        "ei": 11, "st": "text", "si": 0, "fm": false } ] ] }
```

Fig. 3. Chunked snapshot for a document containing the text "Test document" (shortened).

te—the typing of the last three letters: “ent”. Thus, the snapshot contains a text insertion for the string “Test docum” (line 4, highlighted), as well as a number of default style definitions and other basic document properties. The log part of the document contains a single insertion of the string “ent” (line 2, highlighted) with the appropriate time stamp and identifying information.

More generally, document description from revision x to revision y , there would be a snapshot of the state at revision x , followed by $y - x$ entries in the changelog describing each individual change between revisions x and y . The ability to choose the range of changes to load, allows *kix* to balance the flexibility of allowing users to go back in time, and the need to be efficient and not replay needlessly ancient document history.

The changelog for a specific range of versions can be obtained manually by using the development tools built into modern browsers. After logging in and opening the document, the list of network requests contains a load URL of the form: https://docs.google.com/documents/d/<doc_id>/load?<doc_id>&start=<start_rev>&end=<end_rev>&token=<auth_token>, where *doc_id* is the unique document identifier, *start_rev* is the initial revision (snapshot), *end_rev* the end of the revision range, and *auth_token* is an authentication token (Fig. 4). The revisions start at one and must not exceed the actual number of revisions, and the start cannot be greater than the end.

To retrieve the document from the command line, we can compose request using the URL in the address bar and the necessary authentication headers (Google Chrome provides a convenient “copy as cURL” option that constructs the full command line automatically). Alternatively, the URL could be opened in a browser window.

To facilitate automated collection, we built a *Python* tool, called *kumodocs*, which uses the Google Drive API to acquire the changelog for a given range of versions. It also parses the JSON result and converts it into a flat CSV format to simplify its automated processing with existing tools. Each line contains a timestamp, user id, revision number, session id, session revision, action type, followed by a

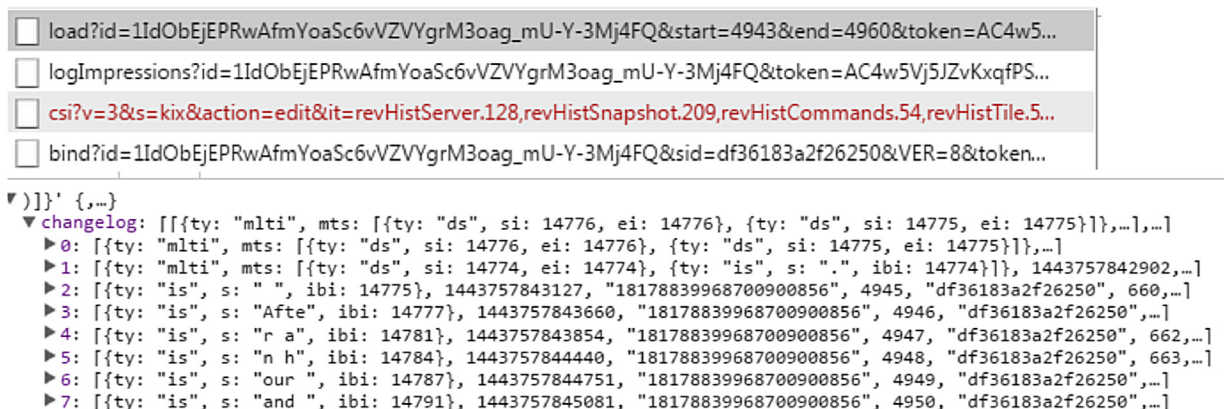
dictionary of key-value pairs involved in any modifications. This format is closer to that of traditional logs and is easier to both examine the editing events manually, and to use command-line text processing tools. The style modification are encoded in dictionaries so that they can be readily used (in *Python*, or *JavaScript*) to replay the events in a different editor.

The first stage in this process is to obtain the plaintext content of the documents, followed by the application of the decoded formatting styles, and the addition of embedded objects (like images). Once the changelog is acquired, obtaining the plaintext is relatively easy by applying all string insert and delete operations, and ignoring everything else.

Actions manipulating page elements, such as a table, equation, picture, etc., have a type of *ae* (add element), *de* (delete element), or *te* (tether element); the latter is associated with a *kix* anchor and *kix* id. Element insertions are accompanied by a multiset of style adjustments, containing large dictionaries of initialization values. Objects like comments and suggestions only contain *anchor* and *id* information in the changelog, and no actual text content.

Picture element insertions contain source location (URL), with uploaded files containing a local URL accessible through HTML5's FileSystem API (filesystem: https://docs.google.com/persistent/docs/documents/<doc_id>/image/<image_id>). Inserting an image from Google Drive produces a source URL in the changelog from the googleusercontent.com domain (Google's CDN). Upon further examination of the HTML elements in the revision document, we established that they were referencing a different CDN link, even immediately after insertion. As expected, images inserted from URLs also had a copy in the CDN given that the source might not be available after insertion.

By analyzing the network requests, we found that the (internal) *Documents* API has a *renderdata* method. It is used with a POST request with the same headers and query strings as the load method used to fetch the changelog:



The screenshot shows a browser's network log with several requests. The third request, highlighted in red, is a 'load' request to Google Docs. Below it, the JSON response is expanded, showing a 'changelog' array with multiple entries. Each entry contains metadata like 'mts' (timestamp) and 'si' (session id), followed by a dictionary of changes. The changes include text insertions ('is') and deletions ('de') with specific 'kix' identifiers and 'ibi' values.

```

load?id=1IdObEjEPRwAfmYoaSc6vZVYgrM3oag_mU-Y-3Mj4FQ&start=4943&end=4960&token=AC4w5...
logImpressions?id=1IdObEjEPRwAfmYoaSc6vZVYgrM3oag_mU-Y-3Mj4FQ&token=AC4w5Vj5JzVXqfP...
csi?v=3&s=kix&action=edit&it=revHistServer.128,revHistSnapshot.209,revHistCommands.54,revHistTile.5...
bind?id=1IdObEjEPRwAfmYoaSc6vZVYgrM3oag_mU-Y-3Mj4FQ&sid=df36183a2f26250&VER=8&token...

]]}' {,...}
▼ changelog: [{ty: "mlti", mts: [{ty: "ds", si: 14776, ei: 14776}, {ty: "ds", si: 14775, ei: 14775}],...}]
▶ 0: [{ty: "mlti", mts: [{ty: "ds", si: 14776, ei: 14776}, {ty: "ds", si: 14775, ei: 14775}],...}]
▶ 1: [{ty: "mlti", mts: [{ty: "ds", si: 14774, ei: 14774}, {ty: "is", s: ".", ibi: 14774}], 1443757842902,...}]
▶ 2: [{ty: "is", s: " ", ibi: 14775}, 1443757843127, "18178839968700900856", 4945, "df36183a2f26250", 660,...}]
▶ 3: [{ty: "is", s: "Afte", ibi: 14777}, 1443757843660, "18178839968700900856", 4946, "df36183a2f26250",...}]
▶ 4: [{ty: "is", s: "r a", ibi: 14781}, 1443757843854, "18178839968700900856", 4947, "df36183a2f26250", 662,...}]
▶ 5: [{ty: "is", s: "n h", ibi: 14784}, 1443757844440, "18178839968700900856", 4948, "df36183a2f26250", 663,...}]
▶ 6: [{ty: "is", s: "our ", ibi: 14787}, 1443757844751, "18178839968700900856", 4949, "df36183a2f26250",...}]
▶ 7: [{ty: "is", s: "and ", ibi: 14791}, 1443757845081, "18178839968700900856", 4950, "df36183a2f26250",...}]

```

Fig. 4. Example load request and changelog response.

https://docs.google.com/document/d/<doc_id>/renderdata?id=<doc_id>

The renderdata request body contains, in effect, a bulk data request in the form:

```
renderOps:{"r0":["image",{"cosmoId":"1dv ... cRQ",
                        "container":"1Ss ... xps"}],
          "r1":["image",{"cosmoId":"1xv ... 3df",
                        "container":"1Ss ... xps"}],
          ...
        }
```

The *cosmoId* values observed correspond to the *i_cid* attribute of embedded pictures in the changelog, and the container is the document id. The renderdata response contains a list of the CDN-hosted URLs that are world readable.

To understand the behavior of CDN-stored images, we embedded two freshly taken photos (never published on the Internet) into a new document; one of the images was embedded directly from the local file system, the other one via Google Drive. After deleting both images in the document, the CDN-hosted links continued to be available (without authentication); this was tested via a script which downloads the images every hour and those remained available for the duration of the test (72 h).

In a related experiment, we embedded two different pictures in a similar way to a new sheet. Then, we deleted the entire document from Google Drive; the picture links remained live for approximately another hour before disappearing. Taken together, the experiments suggests that an embedded image remains available from the CDN, as long as *at least one revision* of a document references it; once all references are deleted, the object is garbage collected. Forensically, this is an interesting behavior that can potentially uncover very old data, long considered destroyed by its owners.

From a security perspective, the CDN design is not unreasonable—it has the security model of a dead drop—anyone who knows the location can access it. Given the length of the identifier and its apparent randomness, it would be infeasible to guess it. From the point of view of application design, it is effectively necessary to maintain CDN objects that could *potentially* be needed to restore a previous version of a document. However, the behavior is not necessarily intuitive to users, and can bring back to life artifacts long considered erased.

Reverting to a previous version is another operation that does *not* destroy the editing history of the document; instead, a *revert* operations containing a snapshot of the desired new state is added to the history. In other words,

the *reversion* operation itself can be later walked back and the state before it can be examined, consistent with the append-only design chosen by Google.

Access to embedded Google Drawings objects is a different from embedded images—the changelog references them by a unique drawing id. The drawing could then be accessed by https://docs.google.com/drawings/d/<drawing_id>/image?w=<width>&h=<height>. This URL *does* require Google authentication and the authenticated user must have appropriate access permissions.

Slides and drawings

We found that the *Slides* app uses a similar changelog approach to transfer the state of the artifacts. That is, the data is communicated as an abstract data structure, which is interpreted and rendered by the JavaScript client code. The overall formatting of the log was similar, with the most important difference being that it was sent as an array of arrays and values instead of a dictionary and values as the *Documents* were (Fig. 5). It appears that all of the keys had been removed from the dictionaries and only the values sent instead, in effect, as a tuple. This makes the reverse engineering a bit more cumbersome, but it still allows us to track the mapping between chosen actions and their encoding, as before.

The first element in each update contains an integer encoding of the type field, with 15 corresponding to string insertion, 16 to text deletion, 3 to text box creation, and so on (the Appendix provides a summary of our findings). The multiset operation in the description of more complex events, such the creation of a new slide, with operations detailing the type of slide inserted, followed by several insert actions for each text box for that slide.

```
{"changelog":[[[4,[1,[365760,205740],[274320,365760]],45,[],[0,"en"]],[13,0,1,"m","1"],[13,0,2,null,"m"],[3,"n:slide",158,[2.032025098800659,0.0,0.0,1.1430000066757202,15252.0,27432.0],[55,0,54,16],"n"],[3,"n:text",108,[1.8287999629974365,0.0,0.0,1.3716000318527222,27432.0,173736.0],[55,1,44,0,54,1],"n"],[41,"n:text",null,"bodyPlaceholderListEntity"],[42,"n:text",null,"bodyPlaceholderListEntity",0,[13,17,19,21,22,23,24,25,26,28,29,31,32,33,36,37],[42,"n:text",null,"bodyPlaceholderListEntity",1,[13,17,19,21,22,23,24,25,26,28,29,31,32,33,36,37],[42,"n:text",null,"bodyPlaceholderListEntity",2,[13,17,19,21,22,23,24,25,26,28,29,31,32,33,36,37],[42,"n:text",null,"bodyPlaceholderListEntity",3,[13,17,19,21,22,23,24,25,26,28,29,31,32,33,36,37],[42,"n:text",null,"bodyPlaceholderListEntity",4,[13,17,19,21,22,23,24,25,26,28,29,31,32,33,36,37],[42,"n:text",null,"bodyPlaceholderListEntity",5,[13,17,19,21,22,23,24,25,26,28,29,31,32,33,36,37]]]]]]]
```

Fig. 5. Slides changelog example (truncated).

```

{"changelog": [
  [[3,"g27de7cf84_0_0",108,[2.292,0.0,0.0,0.2674,63984.0,37722.0],[44,0,45,1],"p",
    1444063509783,"08413168629437028300",2,"f13f7456cc71754",0,null],
  [[15,"g27de7cf84_0_0",null,0,"T"],1444063511799,"08413168629437028300",3,"f13f7456cc71754",1,null],
  [[15,"g27de7cf84_0_0",null,1,"e"],1444063512119,"08413168629437028300",4,"f13f7456cc71754",2,null],
  [[15,"g27de7cf84_0_0",null,2,"s"],1444063512448,"08413168629437028300",5,"f13f7456cc71754",3,null],
  [[15,"g27de7cf84_0_0",null,3,"t"],1444063512448,"08413168629437028300",6,"f13f7456cc71754",3,null],
  [[3,"g27de7cf84_0_1",99,[0.1432,0.0,0.0,0.3263,174285.0,78309.0],[22,381,15,"#CFE2F3",19,"#000000"],"p",
    1444063520352,"08413168629437028300",12,"f13f7456cc71754",7,null]]],
  "chunkedSnapshot": [
    [[1,[365760,274320],[302400,427680]],[45,[],[0,"en"]],[13,0,0,null,"p"],[13,0,1,"m","l"],[13,0,2,null,"m"]],
    [12,"m",0,2,[],[12,"l",0,1,[],[12,"p",0,0,[]]]]
  ]
}]

```

Fig. 6. Drawings changelog/snapshot example.

The first slide consists of a title and subtitle, where the title id is *i0* and subtitle id is *i1*. Every other text box has a unique id such as *g675a3a03f_0_2*, with the final element being incremented whenever a new text box is created with that id. The first number appears to be a version/session identifier—closing the slide and reopening the web page causes it to increase by 1; the 10-digit prefix also changes occasionally but we have not yet established the exact circumstances. Text box descriptions have a six-element list describing the starting *x,y* position in the page, orientation, and scalar size associated with it. The granularity of changes for *Slides* is higher than in *Documents*, as each character has its own revision, rather than occasionally being grouped together with others. Each insertion details the *id* of the text box and the index to be inserted in that text box, and deletion is similar in that a text box id and range are given.

Adding a slide consists of a group of operations (a transaction) containing the creation of a slide, setting of slide attributes, insertion of text boxes (based on the template). Duplicating a slide is a large transaction, consisting of the creation of the same slide type, and—for each text box on old slide—the addition of a box, as well as the respective text and style. Deletion is another transaction, where each box is deleted from the slide first, followed by the slide itself. Changing the theme of a slide creates a massive number of actions inside a transaction with an entirely new slide being created, and each text box is created and has 30–40 modification actions associated with it, followed by the old slide having all of its text boxes deleted and, finally, the old slide itself deleted.

As shown on Fig. 6, the structure of drawing objects' changelog is a simplified version of the *Slides* changelog. Unlike Fig. 5, this one shows the complete artifact for a drawing consisting of a single text box with the word "Test".

Sheets

We applied the same differential analysis approach to the *Sheets* app as before, and monitored the network interactions with the server to understand the protocol. It appears that *Sheets*, which also supports incremental versioning after every update, works differently. When a request for a specific version is performed, the response is a browser-ready HTML document. In other words, both the computations and the HTML encoding are performed on the server, and the final result is spoonfed to the browser for rendering; no dynamic adjustments necessary. It is still feasible to extract the state of the spreadsheet after every update; however, critical information—such as formulas—is not available.

The solution to this problem is to use the Google Sheets API, which provides the means to extract the content of individual cells, including formulas. Such an effort is beyond the scope of this discussion, in part, due to the very different communication protocol adopted in *Sheets*. The results of the API calls to retrieve cell ranges are encoded in XML using the Atom Syndication Format (RFC 4287, 5988) and will require more complex parsing than the lightweight JSON used in *Documents* and *Slides*.

Suggestions and comments

Suggestions are marked up edits to the document that can be accepted, or rejected, by the collaborators; this is similar to the "track changes" mode in *Microsoft Word*. They are present in the changelog and are treated similarly to other changes; however, they have dedicated operation types that allow the editor to treat them differently in terms of formatting and UI (Fig. 7).

Comments are not explicitly represented in the changelog; instead, a *kix* anchor id is present (Fig. 8). Fortunately, the Google Drive API has a *list* method, which

```

▼ changelog: [,...]
▶0: [{ty: "dss", si: 85, ei: 98}, 1444094632128, "18178839968700900856", 174, "49d79c9f90192ecb", 135,...]
▶1: [{ty: "mkti", mts: [{ty: "iss", sugid: "suggest.uzuelc2xg0i", s: "n", ibi: 85},...], 1444094636436,...]
▶2: [{ty: "iss", sugid: "suggest.uzuelc2xg0i", s: "e", ibi: 86}, 1444094637189, "18178839968700900856",...]
▶3: [{ty: "iss", sugid: "suggest.uzuelc2xg0i", s: "w", ibi: 87}, 1444094637807, "18178839968700900856",...]
▶4: [{ty: "iss", sugid: "suggest.uzuelc2xg0i", s: " ", ibi: 88}, 1444094640910, "18178839968700900856",...]

```

Fig. 7. Suggestion changelog example (truncated).

```

▼ changelog: [,...]
  ▼ 0: [{ty: "as", sm: {das a: {cv: {op: "insert", opIndex: 0, opValue: "kix.p5y4wk55ms36"}}}, ei: 128,...},...]
    ▼ 0: {ty: "as", sm: {das a: {cv: {op: "insert", opIndex: 0, opValue: "kix.p5y4wk55ms36"}}}, ei: 128,...}
      ei: 128
      fm: false
      si: 115
      ▶ sm: {das a: {cv: {op: "insert", opIndex: 0, opValue: "kix.p5y4wk55ms36"}}}
      st: "doco anchor"
      ty: "as"
    1: 1444094979511

```

Fig. 8. Comment changelog example (truncated).

allows the retrieval of all comments associated with a document, including deleted ones. However, the actual content of deleted comments is stripped away; only current and resolved ones are available.

Thus, both comments and suggestions are part of the long-term history of a document and are readily recoverable either via the public, or the private service interface.

PoC tool: kumodocs

At present, our proof of concept tool, *kumodocs*, works on *Documents* and *Slides* artifacts. Given a range of revisions (corresponding to a time interval), the tool will acquire and interpret the changelog and will produce a plaintext version of the document as of the last specified revision. *Kumodocs* will also acquire all embedded images that were active for at least part of the period.

For *Slides*, we map all text edits from the changelog to the individual text boxes, and output the result in a series of files: *slide0_box0.txt*, *slide0_box1.txt*, *slide1_box0.txt*, etc. (empty ones are skipped). We also acquire all suggestions and comments associated with the document as appropriately named text file.

Summary

Our experience with analyzing the three *Google Docs* applications validates our motivating concerns. Namely, we saw that—even within the same suite of tools—the approaches to maintaining and rendering the internal state of the artifacts vary in a non-trivial fashion. Based on the observed differences, it is almost certain that these three products have been developed by different teams, and each product bears the stamp of its designers.

Documents is closest to how most web application are developed—data is communicated in abstract form in structured JSON and rendered on the client. *Slides*'s protocol is clearly not designed to be human-readable—it appears that its design is dominated by efficiency concerns. The numeric encoding of operations and flat data structure (the use of JSON is nominal) makes it faster to interpret on the client. Like *Documents*, the data is communicated in pure form and all rendering is performed by the client. *Sheets* takes an entirely server-centric approach—all calculations and all rendering work is done on the server.

In the absence of any standardized external representations—such as the ones used by standalone client applications—there is a definite need to develop tools and

(most likely) new formats that allow the acquisition, and long-term preservation and interpretation of cloud-native artifacts. Similar to *draftback*, our prototype has the ability to perform a basic playback of text editing using the *Quill* open source editor.

Discussion

Based on our analysis, there are several interesting implications for the forensic examination of *Google Docs* artifacts.

Online preview. One unexpected results is that it is *not* unsound for an investigator to review a document in editing mode (in order to have access to all revisions since creation). Since the history of the document is an append-only log, it is practically impossible for any user to spoil the document, as any modifications can easily be undone. However, whole document deletion is still a problem so we need a “write blocker” app/browser extension that monitors HTTP requests and filters out requests that can spoil the evidence.

The golden hour. It appears that Google's CDN, which hosts embedded objects (images), keeps them around for about an hour after deletion. This opens up the opportunity to potentially recover from last-minute deletions, by combining methods from browser and memory forensics and the retrieval of remnant CDN objects.

Reverse engineering is still critical. Our experience shows that reverse engineering is still needed in the cloud forensics environment, although the emphasis will likely shift to network protocols. While prior work (Roussev et al., 2016) showed that the public API is a valuable source of evidence, this experience has brought back into focus the need to understand how SaaS applications work by means of reverse engineering their private protocol and data structures. Fortunately, this is graybox (and not blackbox) analysis as we can monitor all communications and can instrument the client (JavaScript) code at critical junctures.

Long-term preservation is a challenge. One conceptual challenge is the problem of both storing the acquired evidence, and retaining the ability to correctly interpret it. Internal data, like the changelog, is an irreplaceable source of evidence; however, it needs to be interpreted/rendered in order to have meaning to the analyst. Unlike traditional standalone applications, we do not have the ability to retain the application's code (which is split between the client and the server); thus, any solution would involve some

level of equivalency translation. So far, we have addressed the lowest-hanging fruit—replaying plaintext editing commands, and acquiring embedded images. A more complete solution would be to translate the log into commands for a comparable application, where all formatting can be faithfully reproduced so that the visual appearance is retained.

How representative is Google Docs? It is important for follow-up work to understand to what degree the design of *Google Docs* is representative of a broader class of online collaborative application suites in general. We have done some preliminary studies on several similar tools, such as *Zoho Writer*, *Microsoft Word Online*, and *Dropbox Paper*. The initial impression is that, as per the real-time requirements of such apps, incremental updates are continuously sent to the server, and that fine-grain versions of the document are made available to the user (and the investigator). Unlike *Google Docs*, we did not readily identify an internal API mechanism by which the log of editing action could be retrieved. However, there are indications that the log itself likely exists on the server and that the versions shown to the user are generated from it on the fly. There are also signs that the append-only log is an idea that appeals to developers; e.g., reverting to an older version in *Zoho Writer* causes is to be added to the list of user-selectable versions (tagged with “reverted”); *Word* and *Paper* have similar concepts.

Conclusion

In this work, we performed an initial examination of *Google Docs* artifacts in an effort to understand the challenges and opportunities presented by *cloud-native* artifacts. We define such artifacts as data objects used by web/SaaS applications that are hosted exclusively in the cloud infrastructure, and are not stored persistently by client devices. The specific contributions to the field are as follows:

Problem formulation. We argued that the traditional approach of client-side evidence acquisition is completely blind to cloud-native artifacts. Further, the artifacts of greatest importance are internal data structures that contain important historical information are internal data structures. Therefore, we need to develop forensic tools that can acquire and interpret them. Further, we need to develop the means to *independently* render the history of an artifact for archival purposes.

Artifacts & behavior analysis. We performed an analysis of *Google Docs* artifacts, with a primary emphasis on the *Documents* and *Slides* applications and their *changelog* internal data structure. We greatly expanded upon Sommers initial analysis (Sommers) and systematically documented our finding (Appendix: Changelog keys). Further, we investigated the mechanisms used for embedding objects into the artifacts and showed that Google's CDN is a potentially vast source of recoverable data, with apparently unlimited timeframe.

PoC tool development. The main practical result of this work is the development of a set of proof-of-concept tool that extracts and processes the history of *Documents* and *Slides*. Currently, we can extract the text

content for any fine-grain revision, the embedded images and drawings, as well as the history of comments associated with the document. The tool is called *kumodocs*, and is available on *GitHub* at <https://github.com/kumofx/kumodocs>.

In addition to forensics, the tool can also be used to perform a quick privacy audit as it will identify all images, suggestions, comments that have been ostensibly deleted, but are still recoverable.

In the immediate future, we expect to complete the analysis of the remaining apps in the *Google Docs* suite, and to release a more complete specification document similar to (Metz, 2012). Following that, we expect to build a complete solution that allows for the screening, acquisition, and long-term preservation of *Google Docs* evidence.

Appendix. Changelog keys

This appendix contains an (incomplete) set of operation and attribute encodings used in the two versions of the changelog. Its purpose is illustrative; a complete description will be the subject of a separate specification document.

Table 1
Documents changelog keys.

Key/key: value	Interpretation
<i>Operations</i>	
mlti	multi-operation (transaction)
is, ds	insert/delete string
ae, de, ue, te	embedded elements: add, delete, update, tether (to anchor)
rvrt	revert to earlier revision
op	operational transformation
sdef_ps, sdef_ts	set default paragraph/text style
as, sm	adjust/modify
msfd, usfd, sas	suggestion added/rejected/accepted
sugid	suggestion id
<i>Operation attributes</i>	
mts	multi-operation description
s	string argument
si, ei	starting/ending index
ibi	insert before index
tbs_al, tbs_of	table alignment/offset
das_a	datasheet anchor
<i>Document style & attributes</i>	
ds_pw, ds_ph	page width/height
ds_mt, ds_mb	top/bottom margin
ds_ml, ds_mr	left/right margin
lgs_l	language
<i>Header styles</i>	
hs_t, hs_st, hs_nt	title, subtitle, normal text
hs_h1 ... hs_h6	h1/dots/h6
<i>Paragraph style</i>	
ps_hdid, ps_hd	heading id/style
ps_al, ps_ls	horizontal alignment line space
ps_il, ps_ifl	indent line/first line (amount)
ps_sb, ps_sa	space before/after paragraph (amount)
<i>Text style</i>	
ts_ff, ts_fs	font family/size
ts_fg, ts_bg	foreground/background color
ts_bd, ts_it	bold/italic
ts_un, ts_st	underline/strikethrough
ts_sc, ts_va	small caps, vertical alignment

Table 2

Slides changelog codes.

Code	Interpretation
<i>Operations</i>	
0	delete box
3	add box
4	multiset
5	modify box
6	adjust page element
9	adjust page style
12	add slide
13	delete slide
14	move slide
15	add text
16	delete text
17	adjust text style
18	set slide attributes
22	insert table
43	transition
44	insert image
<i>Style modifications</i>	
[0, 1]	bold flag
[1, 1]	italics flag
[2, 1]	underline
[4, <i>hexvalue</i>]	color
[5, <i>fontfamily</i>]	font family
[6, <i>fontsize</i>]	font size (6..400)
[7, <i>fontmod</i>]	super/subscript font (1 = super, 2 = sub)
[11, <i>spacing</i>]	line spacing (100/115/150/200)
[12, <i>halign</i>]	horizontal alignment (1 = left (default), 2 = center, 3 = right, 4 = justified)
[20, 1]	strikethrough flag
[44, <i>valign</i>]	vertical alignment (0 = top, 1 = middle, 2 = bottom)

References

- Chen J, Mulligan B. Quill rich text editor. URL, <https://github.com/quilljs/quill/>.
- Chung H, Park J, Lee S, Kang C. Digital forensic investigation of cloud storage services. Digit Investig 2012;9(2):81–95. <http://dx.doi.org/10.1016/j.diin.2012.05.015>.
- Google. The next generation of Google Docs. 2010. URL, <http://googleblog.blogspot.com/2010/04/next-generation-of-google-docs.html>.
- Google. Google drive blog archive: May 2010. 2010. URL, http://googledrive.blogspot.com/2010_05_01_archive.html.
- Hale J. Amazon cloud drive forensic analysis. Digit Investig 2013;10(3):259–65. URL, <http://dx.doi.org/10.1016/j.diin.2013.04.006>.
- Martini B, Choo K-KR. Cloud storage forensics: ownCloud as a case study. Digit Investig 2013;10(4):287–99. URL, <http://dx.doi.org/10.1016/j.diin.2013.08.005>.
- Metz J. Expert witness compression format version 2 specification, working draft. 2012. <https://goo.gl/iXmkBf>.
- Quick D, Choo KR. Dropbox analysis: data remnants on user machines. Digit Investig 2013;10(1):3–18. <http://dx.doi.org/10.1016/j.diin.2013.02.003>.
- Quick D, Choo K-KR. Google drive: forensic analysis of data remnants. J Netw Comput Appl 2014;40:179–93. URL, <http://dx.doi.org/10.1016/j.jnca.2013.09.016>.
- Roussev V, Barreto A, Ahmed I. Forensic acquisition of cloud drives. In: Peterson G, Shenoi S, editors. Advances in Digital Forensics, vol. XII. Springer; 2016.
- Somers J. How I reverse engineered Google docs to play back any documents keystrokes. URL, <http://features.jsomers.net/how-i-reverse-engineered-google-docs/>.