De-Anonymizing Live CDs through Physical Memory Analysis

Andrew Case

andrew@digdeeply.com

Digital Forensics Solutions

Abstract

Traditional digital forensics encompasses the examination of data from an offline or "dead" source such as a disk image. Since the filesystem is intact on these images, a number of forensics techniques are available for analysis such as file and metadata examination, timelining, deleted file recovery, indexing, and searching. Live CDs present a serious problem for this investigative model, however, since the OS and applications execute in a RAM-only environment and do not save data on non-volatile storage devices such as the local disk. In order to solve this problem, we present a number of techniques that support complete recovery of a live CD's in-memory filesystem and partial recovery of its deleted contents. We also present memory analysis of the popular Tor application, since it is used by a number of live CDs in an attempt to keep network communications encrypted and anonymous.

1 Introduction

Traditional digital forensics encompasses the examination of data from an offline or "dead" source such as a disk image. Under normal circumstances, evidence is obtained by first creating an exact, bit-for-bit copy of the target disk, followed by hashing of both the target disk and the new copy. If these hashes match then it is known that an exact copy has been made, and the hash is recorded to later prove that evidence was not modified during the investigation. Besides satisfying legal requirements, obtaining a bit-for-bit copy of data provides investigators with a wealth of information to examine and makes available a number of forensics techniques. Since the copy will contain the entire filesystem, including its metadata, as well as previously deleted data, investigators can then perform a number of operations such as timelining, hashing of files, recovery of deleted information via file carving, metadata examination for both the filesystem itself and the files it contains, orderly indexing and searching of data, and more. After obtaining a disk image, application specific examination also becomes possible, with popular targets being web browser activity, local email storage, recently accessed files, and any backup facilities such as Windows System Restore.

Unfortunately for digital forensics investigators, live CDs disrupt the normal process in which evidence is obtained, analyzed, and presented. Since Live CDs live entirely in RAM, there is no physical disk to image nor is there currently a method to get a bit-for-bit copy of the file-system. Instead investigators, after obtaining a memory capture, are left with no useful set of tools to perform file-system level analysis. Although primitive techniques such as the use of *grep* and *strings* can locate relevant information, these tools do not place results into any context, making it difficult to perform deep analysis with them. Similarly, analysis of userland applications is just as difficult since there are no logs or history to examine. The lack of useful tools currently makes examination of live CDs very difficult, and for investigators without programming ability, it is almost impossible to perform any meaningful analysis.

The anonymity provided by live CDs has not gone unnoticed in the offensive computing and privacy communities and a number of projects have been released leveraging this feature. For instance, "The Amnesic Incognito Live System (TAILS)" [15] live CD proudly boasts on its front page that "all outgoing connections to the Internet are forced to go through the Tor network" and that "no trace is left on local storage devices unless explicitly asked". The popular Backtrack live CD provides a complete penetration testing environment in which to perform a number of network and host based attacks [4]. While this distribution is created for legitimate purposes, it also provides a powerful weapon for those wishing to perform malicious attacks. Consideration of only these two live CD distributions is enough to warrant interest by forensics investigators and incident response personnel without even considering the large number of other distributions.

The work presented in this paper is aimed at de-anonymizing the TAILS live CD by enabling traditional forensics techniques against it as well as performing memory analysis of Tor as deployed by the distribution. Forensics techniques are enabled by reconstruction of the entire in-memory file system, as well its metadata, and also recovery of previously deleted file system structures. As explained in the technical sections of this paper, this requires analysis of a number of Linux kernel subsystems in order to piece together all needed information. Memory analysis of Tor revealed that it makes minimal effort to securely erase memory after it has been used and this allows recovery of historical data such as HTTP headers and requests, downloaded files, visited URLs, and Tor-specific data such as the identity of other Tor network nodes. We also show that the research performed and tools developed during this project are applicable against a number of other live CD distributions and not just TAILS.

2 Related Work

Forensics memory analysis has received considerable attention in the last few years, with most of the work focusing on kernel data structures and functionality. Inspired by the 2005 DFRWS memory challenge [8], a number of publications were released that attempted to pull information from Windows systems such as processes, threads, files, network connections, and other relevant data [1, 3, 12, 13, 17]. Similar work was performed against Linux systems [5-7, 17] after the release of the DFRWS 2008 memory challenge [9]. There has also been substantial work in Mac OS X physical memory analysis [14]. The results of these works made possible recovery of a wealth of allocated and deallocated information inside the kernel.

Scanning memory for data structures using signatures has been a large component of these earlier systems, but recent work by a team from Georgia Tech has highlighted the weaknesses in this approach [10]. This work showed that all tested scanners could be bypassed by modifying certain structure's members to avoid signatures. In order to solve this problem, the team developed a novel technique in which members of the EPROCESS Windows process structure were "fuzzed" in order to determine if the member was vital to the operation of the system, and, if so, what value ranges can it hold. After this fuzzing was complete, they were able to build a strong signature that could not be bypassed by malware without it introducing serious operating system stability issues. Since scanning for structures is generally required for recovering data to which the kernel or an application no longer has a reference, the ability to generate proper signatures is vital.

3 Reconstruction of the In-Memory Filesystem

To properly reconstruct the filesystem we need precise knowledge of how the filesystem is stored in memory. To illustrate how this knowledge was obtained during our research process, the following sections detail each step.

3.1 Required Linux Internals

Since the recovery of information revolves around parsing of kernel data structures, we present a quick introduction to relevant structures. The kernel structures involved in filesystem handling and that are necessary for comprehension of the presented research include: *struct dentry*, which represents a directory entry (directory or file); *struct inode*, which corresponds to a physical inode of the filesystem; *struct address_space*, which links all the physical pages of an inode together; and *struct page*, which represents a physical page. Related to these structures is the page cache, which is a store of all in-memory *page* structures and is required for reconstructing file contents. The final item requiring explanation is the kernel's file-grained memory allocator, the *kmem_cache*. Briefly, this cache is used to quickly allocate and deallocate structures of the same type, and is used by a number of kernel subsystems to hold structures that are frequently used and have relatively short lifespans. Previously, our team has presented a deep examination of *kmem_cache* internals for memory analysis of general Linux systems and this work showed that it is possible to recover previously deallocated processes, memory mappings, and network information from the caches [6].

3.2 Brief Overview of Stackable Filesystems

In order to coherently present the details of our file system recovery algorithm, we must first discuss stackable filesystems. This special filesystem overlays multiple "versions" of a filesystem, called branches or layers and often with different read/write permissions, onto a single mountpoint. In the case of live CDs, this is a perfect solution as the initial file system contained on the CD needs to be readable, but the boot process and system users also need to be able to create, modify, and delete files. To meet these needs, stackable filesystem implement complex logic that allows for multiple filesystems to be presented as a normal, single filesystem, hiding the multiple branches from end users. In the case of TAILS, Backtrack, and Ubuntu, the stackable filesystem used is *aufs* [2] and this is the filesystem chosen for our analysis.

3.3 Userland View of TAILS and *aufs*

To demonstrate the use of the *aufs* stackable filesystem in TAILS, we first present the view of it from userland. The following figure shows the mounts and mount points relevant to our work.

```
# cat /proc/mounts
aufs / aufs rw,relatime,si=4ef94245,noxino
/dev/loop0 /filesystem.squashfs squashfs
tmpfs /live/cow tmpfs
tmpfs /live tmpfs rw,relatime
```

Figure 1 – TAILS Mount Points.

As shown in Figure 1, *aufs* is mounted as the root filesystem, denoted by "/". It also shows that */dev/loop0* contains the filesystem on the distribution's CD and that both */live* and */live/cow* are mounted using the *tmpfs* filesystem, discussed shortly.

```
# cat /sys/fs/aufs/si_4ef94245/br0
/live/cow=rw
# cat /sys/fs/aufs/si_4ef94245/br1
/filesystem.squashfs=rr
```

Figure 2 – Reading the Branch Mount Points.

Figure 2 further expands on this information by showing that */live/cow* is the writable branch in the stackable filesystem and that */filesystem.squashfs* is the read only version. Note that the bold number after *si_* is the same as the parameter passed to the aufs root mount. The *sysfs* output is consistent with the idea presented previously that the CD is kept in a read-only state within the branch and that changes by the user will be written inside of the writable branch and merged on top of the CD contents.

3.4 *aufs* Internals

aufs implements filesystem branches by creating two sets of data, one that the user sees and a second "hidden" set of data that it uses to track the information that is currently at the top of the filesystem stack. This hidden information includes directory entries, inodes, superblocks, and everything else needed to implement a filesystem. Since our goal is to enumerate all files and directories within this stacked filesystem, our recovery code must be aware of the complex relationships between hidden and non-hidden files and metadata.

To hide directory entries, each *aufs*-controlled kernel *dentry* has its *d_fsdata* member point to an *au_dinfo* structure, which contains an array of *au_hdentry* structures. By accessing this array at the branch index of the file of interest, the hidden kernel dentry of the exposed dentry can be found. Figure 3 illustrates this memory layout.

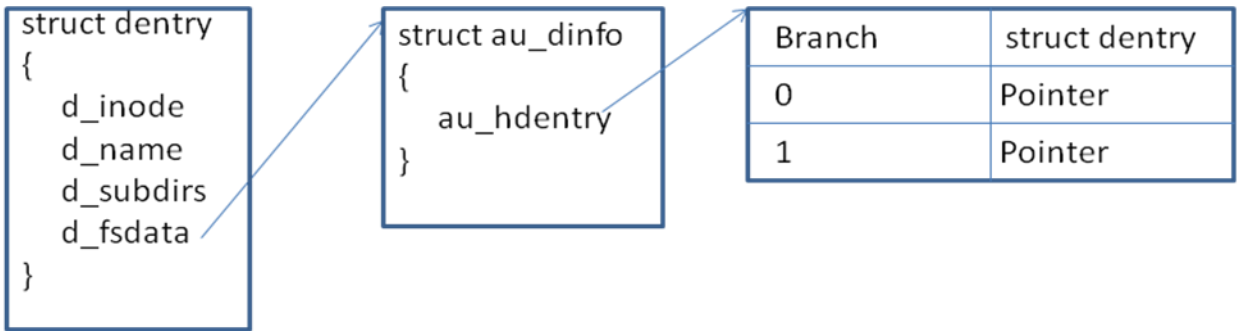

Figure 3. Relationship between in-kernel and aufs directory entries.

Similarly, inodes are hidden by embedding each kernel inode structure within an *aufs_icntnr* structure. This structure contains an array of *au_hinode* structures which can be accessed at the branch index to obtain the hidden inode. Figure 4 illustrates this relationship.

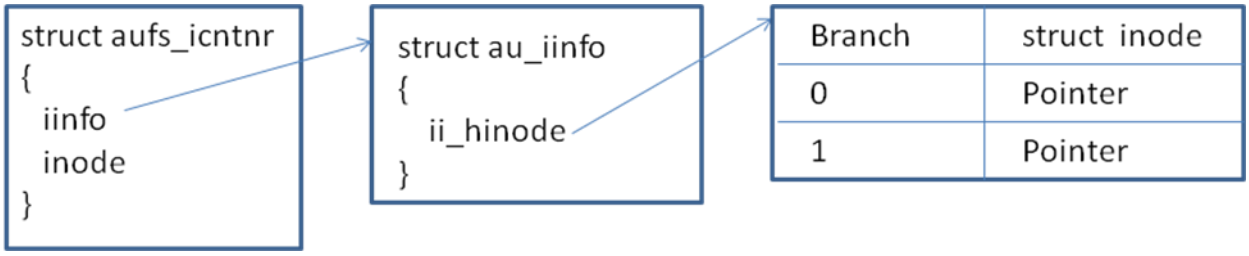

Figure 4. Relationship between in-kernel and aufs *inodes.*

Note that is important to recover files from the proper branch, since files created and modified since boot are only contained in the writable branch, and that recovering the read-only branch would be a waste of time since imaging of the booted CD or DVD would recover the same information. We note that our project supports only *aufs* configurations with two branches, since this is the only configuration used by live CDs.

3.5 Reconstruction Algorithm

Our filesystem reconstruction algorithm works by parsing the kernel data structures necessary to recurse directories, enumerate files, and gather metadata (such as information normally available via the Unix *stat* command). The first step in this process is enumeration of each superblock until the one representing the *aufs* filesystem is found. This enumeration occurs over the *superblocks* list that contains a *struct super_block* for each active filesystem. Once the target superblock is found, its root directory, contained in the *s_root* structure member, can then be recursed.

This recursive procedure starts by gathering each file in the current directory and then walking the *d_subdirs* member of the current dentry. The dentry member is a list of all subdirectories of the directory structure, and for each one found, the processing function is recursively called. To gather relevant metadata information, the name of the current file being processed is copied from the *d_name.name* of the dentry structure and information, such as size, mode, inode number, and MAC times, are pulled from the dentry's inode. A pointer to this inode is contained in *d_inode* member of the structure. In order to obtain the correct inode information, the hidden inode corresponding to the pointer in *d_inode* must be analyzed. Otherwise, inaccurate data from the exposed inode will be gathered, leading to inconsistent timelining and the inability to gather file contents as discussed next.

To gather file contents, first the size of the file in question must be obtained from the hidden inode's *i_size* member, and then each page of the file must be obtained from the page cache. The page cache contains a *struct page* for each active physical page in the system and an inode's linkage into the page cache is contained within its *address_space*. *page_tree* member. In order to gather all contents of a file, this tree must be queried per page, indexed by position in the file. The kernel uses this demand-paged method for handling page requests since, for example, if someone requests a page 10MB into a file, only one particular page needs to be accessed and not all the ones before it. This also means that our gathering algorithm iterate over entire files, querying each page and then gluing them together into a complete file.

To properly duplicate the in-memory filesystem, the processing scripts, described later, make an effort to not only copy directories and files in the same tree structure as in memory, but they also work to preserve timestamps. Since timelining is a vital piece of most investigations and since timestamps must be verifiable in legal proceedings, it is imperative that this detail is not overlooked. To preserve timestamps, once a file is written, its timestamps are updated to match that of the in-memory filesystem. This occurs by copying the MAC times contained in the appropriate hidden inode into the on disk inode created by the scripts. Assuming that output is directed at a mountpoint which can be set to read-only after processing is complete, these extra steps will ensure that all timestamps stay accurate.

3.6 Recovering Deleted Filesystem Information

Recovery of deleted files is often paramount to an investigation and during this research project an effort was made to recover as much deleted information as possible. The good news is that through careful *kmem_cache* analysis it is possible to recover deleted directories and filenames, including all the metadata contained in inodes about these files. Since both *dentry* and *inode* structures are contained within the *kmem_cache*, recovery of this information is straightforward as either can be used to link to the other. Furthermore, *aufs* creates its own *kmem_cache* caches, including ones for its structures that contain the hidden directory entries and inodes. Though the first method, dealing directly with native kernel structures, was chosen, either choice would have led to the recovery of previously deleted file system structures and metadata. Complete details of the *kmem_cache* and recovery of its contents are explained in a previous publication [6].

Unfortunately, no orderly method for recovery of the data associated with deleted files was discovered, since pages are freed and removed from the page cache upon deallocation. This effectively removes any linkage between files and physical pages, making orderly recovery seemingly impossible. Of course this does not prevent examination of individual deallocated pages using traditional live forensics techniques.

4 Memory Analysis of Tor

In order to fully deconstruct the defensive systems of the TAILS distribution, we also chose to perform memory analysis of Tor. The combination of filesystem and Tor memory analysis attacks the two key components that TAILS and other live CD offer for anti-forensics. While we currently have some interesting results, this is a work in progress.

4.1 Initial Recovery of Data

Due to the size and complexity of Tor, complete memory analysis of its data structures and functions would consume weeks of work. In order to motivate future work, this section will list the results of our initial analysis of Tor and discuss potential future targets of research.

Before deep analysis of Tor was performed, the classic forensics technique of using *strings* and *grep* to find interesting information was performed on memory dumps of the Tor process to ensure that useful information is indeed not overwritten on deallocation. To test this we installed the *privoxy* proxy server, configured it to send requests through Tor, and then set the *http_proxy* environment variable to the address of the *prixovy* server. Once this was completed, we then used *wget* to recursively download information from a number of websites, all of which contained tens of web pages, downloads (doc, pdf, etc), and other information. To verify that this information was still in Tor's memory after the requests were completed, we used Michal Zalewski's *memfetch* [11] utility to

download memory regions of the Tor process such as the heap, .data segment, and .bss segment. *strings* and *grep* were then run across the extracted memory regions and it was confirmed that information such as the HTTP headers, file and web pages contents, virtual hosts of requested pages, and more were contained in clear text in memory.

4.2 Recovering the Chunks Freelist

To support orderly collection of this data, Tor's source code was analyzed to locate the data structures used to store sensitive information. After this analysis, two Python scripts were written to gather data from the targeted data structures. The first script targeted the *freelists* array that links all deallocated *chunks* of memory previously used by Tor. Each element of the *freelists* array is a *chunk_freelist_t* structure, which contains the size of the items allocated (*alloc_size*) and a pointer to the beginning of the chunks freelist (*head*). Each chunk is represented by a *chunk_t* structure and can be enumerated by walking the *next* list embedded in the structure until a NULL pointer is found. The size of each data member is contained in the chunk's *datalen* member and the *data* member provides the address of where the chunk's data starts. Using these two members, all information from every free chunk can be gathered. For clarification, Figure 5 lists the important members of the associated structures.

```
typedef struct chunk_freelist_t {
   size_t alloc_size; // size of chunk
   int cur_length;   // number on list
   chunk_t *head; // first free chunk on list
}
typedef struct chunk_t {
   struct chunk_t *next;  // pointer in list
   size_t datalen; // # of used bytes in chunk
   char *data;
} chunk_t;
```

Figure 5. Tor Chunk Freelists Structure.

4.3 Recovery of the Cell Pool

The second Python script developed targeted the *cell_pool* memory pool of *packed_cell_t* structures. In Tor, every incoming and outgoing message is wrapped in a *cell* and, unless cleaned, the cell pool contains every cell used by Tor to perform communications. Enumeration of the cell pool gathers all of these cells, including their full content. In order to gather these cells, our script first had to locate the *cell_pool* address and then walk each of its *empty_chunks*, *used_chunks*, and *full_chunks* members. Note that these "chunks" are not the same as the previously discussed *freelist* chunks. Instead, the pool chunks are represented by a *mp_chunk_t* structure that embeds a doubly linked list of chunks (*next* and *prev*), contains a backpointer to the owning pool, the size of chunk's data (*mem_size*), and the address of the dynamically allocated data (*mem*). Since the memory pool structures are agnostic to the type of data stored, the *mem* buffer stores whatever data is being tracked. In the instance of the cell pool, these are *packed_cell_t* structures which contain a *next* pointer to the next cell in the list and *payload* buffer that contains an incoming or outgoing message's content.

The pool walking script gathers cells by finding the address of *cell_pool*, and for each of chunk lists (empty, used, and full), it walks all contained *mp_chunk_t* structures and extracts data contained in the *mem* buffer, which is *packed_cell_t* structures. This effectively recovers every cell that hasn't been trimmed by the system. Again, for reader clarity the structures with relevant members documented is illustrated in the following figure:

```
struct mp_pool_t {
   struct mp_chunk_t *empty_chunks;
   struct mp_chunk_t *used_chunks;
   struct mp_chunk_t *full_chunks;
   size_t item_alloc_size;
}
struct mp_chunk_t {
   unsigned long magic;
   mp_chunk_t *next;
   mp_chunk_t *prev;
   mp_pool_t *pool;
   size_t mem_size;
   char mem[1]; /*< Storage for this chunk. */
}

typedef struct packed_cell_t {
   struct packed_cell_t *next;
   char body[CELL_NETWORK_SIZE];
} packed_cell_t;
```

Figure 6. Structures related to packed cell recovery.

5 Implementation

In this section the implementation of each of the developed analysis components is briefly discussed.

5.1 Test System

Our test system consisted of a VMware workstation installation that was set to boot from a disk image, either TAILS or Backtrack, depending on the testing being performed. Special consideration needed to be taken during this project due to the volatile state of live CDs and the potential to lose analysis scripts and other post-boot data in the case of system hang or reboot. To alleviate this issue, after all necessary development tools were installed during the initial live CD boot, a VMware snapshot of the system was taken. This allowed for immediate return to the state of the machine within the configured live CD environment. The proper use of VMware and its snapshots facility saved countless hours of research time compared to rebooting the live CD and reinstalling packages each time the VM needed to restart.

5.2 File System Recovery

The initial implementation of the filesystem recovery modules, including both analysis of allocated and deallocated information, used loadable kernel modules. This approach was chosen as it allowed for rapid development, debugging, and experimentation with in-memory data. Once the algorithms for recovery of filesystem information were developed and tested, they were reimplemented as Volatility[17] plug-ins. Volatility is the most popular tool for forensic memory analysis and provides the plug-in developer with a feature-rich API that handles many of the low-level details necessary for physical memory processing. Due to the popularity of Volatility within the forensics community and the fact that traditional forensics memory examinations focus on captured memory images, once the developed scripts are released publicly, they will be immediately useful to hundreds of investigators.

5.3 Tor Analysis

Currently, the Tor analysis scripts are written as standalone Python scripts that parse Tor specific structures from memory in order to recover data in an orderly manner. Targeted memory sections of the Tor process were acquired using the *memfetch* [11] utility, which has the ability to copy arbitrary regions of a running process's memory to disk. We are now in the process of converting these scripts to Volatility plugins.

6 Conclusions

Live CDs present a difficult problem for forensics investigators equipped with current tools. Since live CDs completely avoid the local disk, they leave no evidence that traditional techniques and tools can discover. In order to address this issue, this whitepaper has presented novel forensics techniques to recover the entire in-memory filesystem from a number of popular live CDs as well as the ubiquitous *tmpfs* Linux filesystem. This paper also presented novel methods that allow for recovery of deleted filesystem information in an orderly manner, and discussed some of the downfalls in attempting to recovery previously deleted file contents. An initial Tor memory analysis effort was also described and we note that due to the amount and sensitivity of the information discovered, future work in this area looks very promising. Once the presented research is published and the developed plugins distributed with the Volatility project, investigators will have a powerful tool for performing live CD investigations.

7 Future Work

While we have presented a wealth of forensically interesting information, there are still avenues of related research left to be explored. The first goal is testing of the developed filesystem plugins against a number of other *aufs* configurations, since the TAILs method is only one of many. As an example, an astute reader may have wondered why files changed since boot were not simply recovered by copying the TAILS */live/cow* directory, which would have indeed worked, but would have been useless against Ubuntu and Backtrack as their *aufs* configurations do not expose the writable branch. The second is the development of plugins that can handle other in-memory filesystems, such as *unionfs* [16], which is also used by popular live CD distributions.

The previously described Tor research has led to a number of other avenues that will be explored in future research. The first is recovery of cells that have been dropped from the cell pool. This will require scanning of heap memory for the structures as Tor no longer has a reference to them. We believe this will be possible, however, since the *magic* member of *mp_chunk_t* is set to a hardcoded value, making it an ideal scanning signature. The second area of research will be recovery of cell encryption keys, which would allow for decryption of each cell's *payload* data. Beyond the clear-text information already contained in Tor memory, this would reveal a wealth of previously sent and received network data.

8 Acknowledgements

The author would like to thank Dr. Golden Richard of the University of New Orleans, Michael Auty of the Volatility Memory Analysis Project, and Dionysus Blazakis of Independent Security Evaluators for guiding the research and reviewing the paper.

References

[1] A. Arasteh, "Forensic memory analysis: from stack and code execution history," *DFRWS*, 2007.
[2] aufs, *http://aufs.sourceforge.net/*, 2010.
[3] B. Dolan-Gavitt, "Forensic analysis of the windows registry in memory.," *Digital Investigation*, vol. 5, 2008.
[4] BackTrack, *http://www.backtrack-linux.org/*, 2010.
[5] A. Case, *et al.*, "FACE: Automated digital evidence discovery and correlation," *Digital Investigation*, vol. 5, pp. S65-S75, 2008.
[6] A. Case, *et al.*, "Treasure and tragedy in kmem_cache mining for live forensics investigation," *Digital Investigation*, vol. 7, pp. S41-S47, 2010.
[7] A. Case, *et al.*, "Dynamic recreation of kernel data structures for live forensics," *Digital Investigation*, vol. 7, pp. S32-S40, 2010.
[8] DFRWS, "Foresnics Challenge," *http://www.dfrws.org/2005/challenge/*, 2005.
[9] DFRWS, "Forensics Challenge," *www.dfrws.org/2008/challenge/index.shtml*, 2008.
[10] B. Dolan-Gavitt, *et al.*, "Robust Signatures for Kernel Data Structures," *ACM Conference on Computer and Communications Security*, 2009.
[11] lcamtuf, "memfetch," *http://lcamtuf.coredump.cx/soft/memfetch.tgz*.

[12] A. Schuster, "Searching for processes and threads in Microsoft Windows memory dumps," *Digital Investigation*, vol. 3, pp. 10-16, 2006.
[13] A. Schuster, "The impact of Microsoft Windows pool allocation strategies on memory forensics," *Digital Investigation*, vol. 5, pp. S58-S64, 2008.
[14] M. Suiche, "Mac OS X Physical Memory Analysis," *Blackhat DC*, 2010.
[15] TAILS, "TAILS Live CD," *https://amnesia.boum.org/*, 2010.
[16] unionfs, "Unionfs: A Stackable Unification File System," *http://www.fsl.cs.sunysb.edu/project-unionfs.html*, 2010.
[17] Volatility, *https://www.volatilesystems.com/default/volatility*.