FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory *

Nick L. Petroni, Jr.[†] *npetroni@cs.umd.edu*
AAron Walters[‡] *awalters@4tphi.net*
Timothy Fraser[†] *tfraser@umiacs.umd.edu*
William A. Arbaugh[†] *waa@cs.umd.edu*

† *University of Maryland, College Park, MD 20742, USA*
‡ *4tphi Research, Vienna, VA 22182, USA*

Abstract

We present the Forensic Analysis ToolKit (FATKit)–a modular, extensible framework that increases the practical applicability of volatile memory forensic analysis by freeing human analysts from the prohibitively-tedious aspects of low-level data extraction. FATKit allows analysts to focus on higher-level tasks by providing novel methods for automatically deriving digital object definitions from C source code, extracting those objects from memory images, and visualizing the underlying data in various ways. FATKit presently includes modules for general virtual address space reconstruction and visualization, as well as Linux- and Windows-specific kernel analysis.

Keywords: computer forensics, digital evidence, digital investigation, incident response, volatile memory analysis

1 Overview

For the purposes of incident response and analysis, volatile system memory represents a valuable yet challenging medium for the collection of digital evidence. While traditional digital forensic techniques have focused on disk drives and other more lasting data sources [16, 10, 9, 17], system memory can provide a great deal of information about the system's runtime state at the time of, or just after, an incident [39, 12, 17]. Details such as running processes, loaded libraries, logged-in users, listening network sockets, and open files are all available in system memory. This information can provide a great deal of context, particularly when used in conjunction with traditional forensic data sources. Additionally, recent advances in real-world threats have shown a trend towards memory-only modification whenever possible [27, 29, 5], thereby rendering traditional post-mortem analysis techniques blind to the existence of intruders. While the need for access to forensic data extracted from volatile memory has been demonstrated [17, 12], a number of barriers make this access difficult.

System memory is different from its less-volatile counterparts in a number of significant ways. First, the transient nature of the data makes it more difficult to collect without modifying the data itself, a quality known as *preservation* within the forensics community [17]. The longer the system runs, the more memory changes, possibly replacing clues left at the time of the incident [17]. However, stopping the system from

[0]NOTICE: This is the author's version of a work that was accepted for publication in Digital Investigation. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Digital Investigation, Volume 3, Issue 4, December 2006, DOI information: 10.1016/j.diin.2006.10.001

running (at least in any conventional way) will likely destroy much or all of the evidence. To address these issues, a number of systems have been proposed to aid with the collection of data (referred to here as "images") from volatile memory [21, 23, 25, 12]. These techniques, while not perfect, have highlighted an even more challenging problem, which Carrier refers to as the *Complexity Problem* [7].

The second difficulty facing digital forensic analysis of system memory is the complexity of collected data. On a typical multi-programmed system, a single physical address space frequently contains machine instructions, initialized and uninitialized data, and architecture-specific data structures for a number of different programs. Additionally, many processors support memory virtualization, whereby each program or operating system may view the same physical address space in different ways [28]. Sections of this virtual address space may be in memory, on disk, or may not exist at all. Finally, depending on implementation specifics, each program can organize itself within its virtual address space in almost arbitrary ways. Differences in programming languages, compilers, operating system application programming interfaces (APIs), and system libraries make each system slightly different. Unlike disks, where data has a predefined structure for portability and backwards compatibility, most instances of data in main memory are not meant to be used by different versions of different programs in a single consistent manner.

Because of these challenges, there is currently no tool or procedure for performing comprehensive analysis of low-level data collected from a running system. Furthermore, the few techniques that do exist provide either minimal information, as in the case of string searches and checksum comparisons [17], or implement the time consuming, hand-coded reproduction of data structures and algorithms utilized by system software such as the operating system [2, 22, 4]. In today's world of frequent software updates, such hand-coded approaches constantly require the attention of experts in order to keep up with new releases. Even small changes, such as build-time configuration or compiler flags, can change the low-level footprint of a program's digital objects. More importantly, extending the set of objects available to the expert for analysis requires more low-level coding for each extension. This lack of suitable tools forces analysts to expend much time and resources on low-level data gathering rather than more profitable high-level forensic investigation.

In an attempt to provide a more comprehensive solution, we have developed the Forensic Analysis ToolKit (FATKit). FATKit is a modular framework that allows incident responders to extract, analyze, aggregate, and visualize forensic data at the various levels of data complexity inherent to system memory. As a note to the reader, for the purposes of this paper, extraction refers to the derivation of meaning and structure from a given image and *not* the mechanism used to obtain raw data from the system itself. The design of FATKit is centered around the concept of defining system abstractions and creating modules to implement the necessary functionality at each level of abstraction such as physical memory, virtual memory, program data structures, and application-specific data interpretation. This layered approach allows experts to reason from the point of view of a particular abstraction (e.g., a particular process' virtual address space), and make that data available for higher-level analyses (e.g., the semantics of a particular application). Where possible, an emphasis has been placed on reusability and automation, leaving more time for analysis and presentation rather than extraction.

This paper presents an introduction to our forensic analysis approach based on system abstractions, details the design of our toolkit, describes a set of modules written to automate and analyze software running on Intel IA-32 hardware, and relates some specific results produced by our work thus far. Some of the most promising results include:

- *Automated interpretation of C data structures.* One of the most useful inputs to a forensics analysis system is information about the structure of low-level data and its relation to high-level constructs. We have developed an automated tool for extracting all data types from a C program and mapping those types to their low-level representations. The resulting data mapping can be used directly by our analysis tools in a manner that gives the analyst a program-level view of in-memory objects. For our

Linux kernel example, this includes over 1100 data structure definitions available to the investigator for examination.

- *Address space reconstruction.* Our implementation of virtual to physical address translation and address space reconstruction allows the analyst to work within a process' virtual address space rather than the combined physical address space. This allows the investigator to interpret addresses and pointers directly. While we have only implemented the IA-32 virtual address translation module, the framework provides for simple extension to other architectures.

- *Support for programmable analysis extensions.* Built on top of our memory object framework, an investigator can easily write extension modules to manipulate and analyze low-level data structures as if they were high-level objects. Our example modules can perform tasks such as listing all processes and modules from kernel memory and searching for hidden processes in just a few lines of code. Since it is impossible to predict which objects will be of interest in a given incident, the ability to quickly craft such functions greatly speeds up analysis. To demonstrate the generality of our approach, we have implemented process-listing functionality for both Linux and Windows systems.

- *Low- and high-level data visualization modules.* One of the most useful features of our framework is the ability to add graphical representations of data at various levels of abstraction. One such visualization module that we have implemented is an "object browser" that allows the analyst to expand and collapse in-memory objects and their nested fields, follow pointers, and cast objects to other object interpretations. Our address space viewer, which interacts with the object browser, can be used to display data instances in both their virtual address space, if available, and the original physical memory image. The viewer also provides for color-coded objects, hexadecimal and ASCII depictions of each byte, and support for overlaying symbol names or analyst notes at a particular offset.

The remainder of this paper proceeds as follows. In Section 2, we describe in more detail the difficulties surrounding the extraction of meaningful information from low-level memory dumps and provide some intuition behind our approach. Section 3 describes FATKit's architecture in detail, while in Section 4 we present a number of specific modules and analyses that demonstrate the utility of our framework. Finally, Sections 5 and 6 describe related and future work respectively.

2 From Bits to Evidence: Data Extraction in Volatile Memory

In this section, we provide an introduction to the problem and process of obtaining evidence from an image of system memory. The typical scenario involves an investigator or system analyst who has reason to believe a crime or break-in has occurred. Having dumped system memory in an attempt to preserve evidence, the investigator's goal is to answer both high-level and technical questions about the incident based on the collected data. Examples of the types of information an investigator might consider include: which users were logged on to the system at the time in question, what other computers were communicating with the system, what files did a particular user have open, and through what program did the attacker gain entry. To answer these questions, the analyst will first need to extract the necessary data at the proper level of abstraction, and then process that data in search of a conclusion. The general approach we have taken is to automate as much of the former as possible and to provide tools that allow the expert to focus on the latter. We therefore start by considering the problem of extracting structure from low-level data.

In its most basic statement, the problem is one of deriving structure and meaning from a stream of bits found in memory. While we may be able to glean some information by examining this low-level data independently, we are much more likely to succeed with the added knowledge of how the system operates. Main memory is a complex data store, utilized by both hardware and software to maximize efficiency while supporting multi-programming and protection. The contents of memory at any given moment are the result of a number of transformations performed on behalf of the operating system, applications, and even devices using direct memory access (DMA). Each of these entities has its own view of system memory, which is frequently different from the views of other entities. For example, processes are typically given their own virtual address space that maps uniquely to the underlying hardware. Parts of this address space may be spread out in various regions of main memory, may be currently on disk, or may not be accessible at all. Additionally, programs (and their runtime instances as processes) may utilize their own memory views in drastically different ways. Differences such as language, compiler, data structure choice, and algorithm frequently result in diversity at the lowest level. The key to extracting meaningful structure from memory is therefore a twofold process: (1) reconstructing the way a particular piece of software (e.g., an operating system or process) interacts with the underlying physical memory and (2) identifying how that software organizes and interprets data within its own view of memory.

In general, having more information about a particular system will result in more success on the part of investigators and their tools. For example, reconstructing a particular process' view of memory will be easier with detailed knowledge of the underlying processor architecture and operating system. Similarly, information such as source code, programming language, compiler version, and compiler flags can provide a great deal of information about how a program utilizes the memory it has available to it. For example, knowledge of a particular C data type and its compiled format will allow the analyst to decode such an object after locating it in memory. Decomposition of these objects can provide valuable information about program state and even lead to the location of other objects in program memory through the interpretation of pointers. One major challenge in this area is the specificity of information that is required. First, simple changes made in the course of the software life cycle, such as data structure modification, result in changes to the low-level format of that data. Additionally, knowledge of (or possession of) the compiler or assembler might be necessary to obtain the mapping between the high-level language and the machine representation, at least without performing a fair amount of reverse engineering.

In practice, an investigator will not have complete access to all of this information, particularly if they are analyzing a system that has been compromised or modified by an attacker in unknown ways. Still, even when there are unknowns, experts may be able to extract clues from the data and proceed based on certain assumptions until that hypothesis is proved or disproved sufficiently. Examples of analyses that might lead to such clues are searching for human-readable strings or well-known "magic numbers," comparison with known files or hashes, and attempting to disassemble sections that are believed to be machine code.

Based on the above discussion, we now categorize a set of tasks an investigator might want to perform.

- *Data identification.* The analyst's first task is typically to identify as many specifics about the image as possible. Which architecture is being used? Which operating system? Which parts are data and which are machine code? Is there a specific file a piece of data or program came from? Information about the origins of the data will frequently help to apply known structure during analysis. While some of this information will hopefully come from a system administrator or other site-specific experts, much of it will need to be gathered directly by investigators.

- *Data view transformation.* The analyst may wish to view certain data from the point of view of a particular entity in the system. For example, reconstructing a particular process' address space will allow the investigator to reason about that process in a more natural way. Data may need to go through multiple transformations to be represented correctly from the point of view of a particular entity.

- *Overlaying data structure.* Once a given program has been identified, the analyst can obtain a large amount of information by "decoding" data based on known program structures. In some cases the analyst may know exactly where certain data structures should be located in a given program. In other cases, the analyst may wish to test certain regions for a known structure.

- *Extracting program structure.* In order to apply structure, the investigator first needs to have obtained that structure from the target software. To date this has largely been performed manually, particularly in the absence of source code. In Section 3.3 we present a tool for automating this step when C source code is available, as is the case for popular Open Source operating systems like Linux.

- *Data abstraction and analysis.* As data is identified and decoded, the analyst will frequently want to make that data available in a more abstract form for higher-level analyses. For example, an investigator may seek to aggregate the properties of all instances of a particular C data-structure, such as an operating system's process structure, in order to analyze which users were associated with which processes or determine if any processes were hidden from various system utilities.

- *Data visualization.* Presentation is an important aspect of analysis, particularly when a large amount of information is available [8]. Investigators benefit from the ability to represent low-level data in a number of pictorial forms. Examples include depicting virtual to physical address space mappings, highlighting certain data structures in a linear address space, and representing histograms of potentially encrypted data.

Unfortunately, these are all difficult and time-consuming tasks to accomplish manually. However, they must be accomplished before the analysts can begin their true work – deriving high-level forensic conclusions. The difficultly of performing these initial tasks manually limits the practicality of forensic analysis by forcing investigators to consume time and resources that might otherwise be more productively spent on high-level thinking. In practical situations, where time and resources are a finite quantity, this expenditure can put some kinds of forensic analysis beyond reach. FATKit addresses this difficulty by providing automated tools for these initial tasks, thereby increasing the kinds of forensic analysis that are feasible in practical environments.

3 The Forensics Analysis ToolKit

In this section, we describe the Forensics Analysis ToolKit in more detail. The stated goal of our toolkit is to provide analysts with a framework in which they can reconstruct views of data at different levels of abstraction and perform analyses on that data.

We begin by introducing our notion of abstraction and transformation for forensic analysis of volatile data. We then describe the high-level architecture implemented by FATKit and explain the various components. All FATKit analysis tools are currently implemented in the Python programming language [37]. Finally, we describe our profile extraction tool for C programs, implemented in OCaml [18] as a module for the C Intermediate Language (CIL) program analysis tool [33].

3.1 Abstraction and Data Views

In [7], Carrier presents a general introduction to the nature of "Forensic Analysis and Examination" tools and the utility of abstraction within those tools. Abstraction, which typically refers to the masking of underlying implementation or detail, is a useful tool for computer scientists or anyone dealing with complexity. As it applies to digital forensic tools, Carrier explains that one of the fundamental challenges to an examiner is the low-level complexity of the data he or she is examining. He goes on to define abstraction layers based on their inputs, outputs, and "error," a term given to potential failures of the tool or inadequacies of the abstraction itself. Figure 1 shows the set of abstractions at the core of the FATKit framework.

In our architecture, analysts deal with data from a particular "view," which is an abstraction on top of a machine-level address spaces. Views may represent structured, semi-structured, or unstructured data depending on the amount of information available and the nature of the data. Views can be applied to any

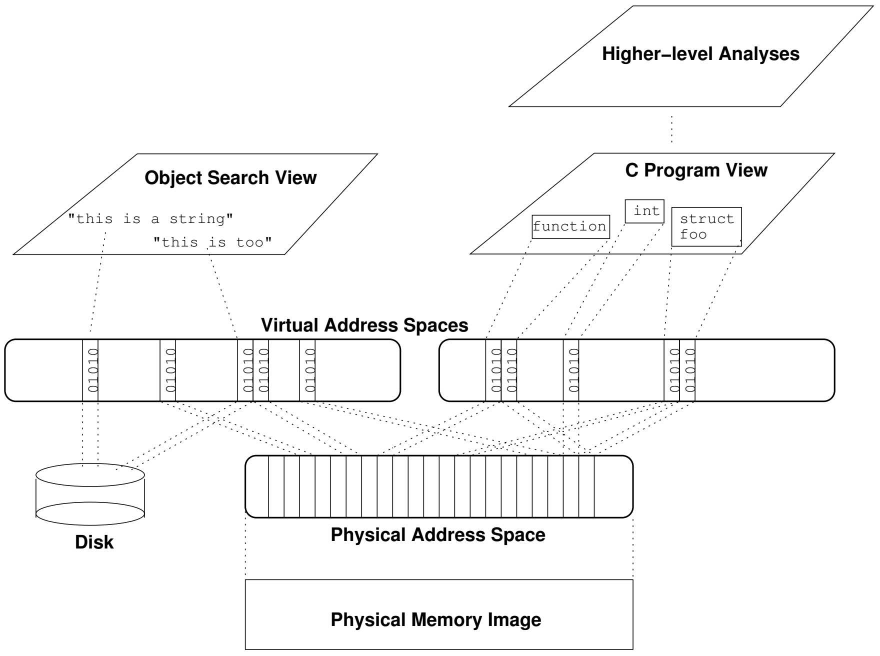


Figure 1: Abstractions in physical memory data.

address space that supports the underlying data model. Within each view exists a set of "objects." In the diagram, an example of a C program executing in a virtual address space is shown. The objects within this view correspond to instances of the types found in that particular C program, including user-defined types and any "native" types, such as integers or floating point numbers, that are provided by the processor architecture. Unstructured views may be utilized to perform data identification tasks such as searching for string or known file headers. Finally, views may utilize other views to perform higher-level analyses and aggregate data.

As shown in Figure 1, the lowest level of data is the binary input that was collected from the machine in question. While depicted as a physical memory dump, there are other possibilities. For example, an entry in the Linux `/proc` file system may been dumped, providing access directly to a processes virtual address space rather than the physical address space. For the remaining examples in this paper we assume access to a physical memory image, but the concepts remain consistent when starting from a higher-level of abstraction.

3.2 FATKit Architecture

In this sub-section, we describe how the above abstractions are represented in FATKit software architecture. FATKit is a modular framework that allows the analyst to add analysis modules where appropriate. With an emphasis on automation and reuse, modules should try to be as general as possible to support future features. Figure 2 shows the major components of the toolkit, which we now describe in more detail.

Address Spaces. The `AddressSpace` class is used as the base class for implementations of processor-supported address space modules. Our current implementation provides a physical address space, for direct access to the file input, and an IA-32 paged virtual address space, discussed in Section 4. Address spaces simulate random access to a linear set of binary data through a `read` function and must support a test to identify whether a given region is accessible in the represented address space. This is necessary for cases

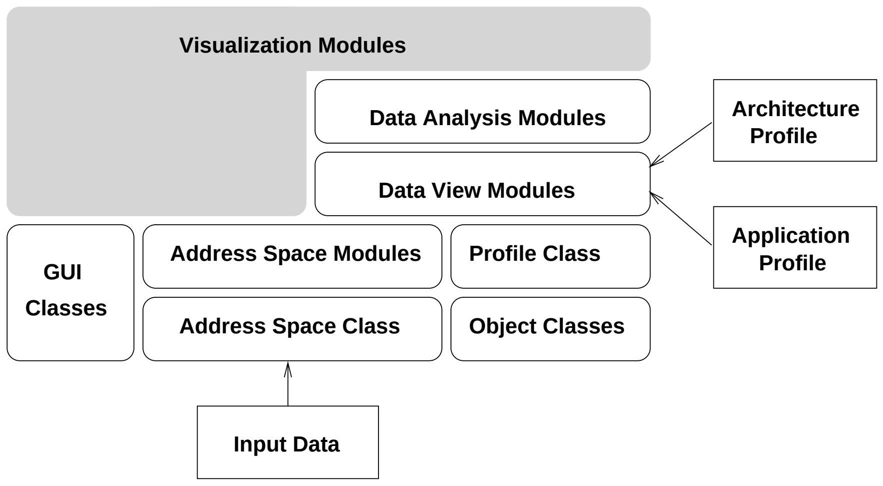


Figure 2: FATKit software architecture.

where the input data has been truncated or corrupted, or where the requested section of a virtual address space was not in physical memory when data was collected. Investigators can have multiple instances of address spaces open at the same time, giving them access to data from a variety of viewpoints simultaneously.

Objects and Profiles. The `Object` class is an abstraction on any data that might be found in an address space. All objects exist within an address space and occur at a particular offset. Objects also have a type, typically associated with a higher-level data type such as a C structure or an ASCII string. Our current object model supports many semantics of the C programming language, including casts to other types of objects, dereferencing of pointer values, and arrays of finite length. In addition to these simple constructs, we have also implemented a number of higher-level features in the object framework such as “deep member accesses” and generic list functions. Deep member accesses provide a syntactic mechanism for analysts to easily reference data structures accessible through a series of references within other data structures (similar to the C `->` syntactic sugar). Polymorphic list iteration functions allow the investigator to perform operations on a series of objects in memory by identifying start and end objects (or addresses) and the member name or function to use for identifying the next element of the list.

Since objects take different forms at the lowest level depending on the program, language, and architecture, we have implemented profiles that define object formats at runtime. Profiles are composed of descriptions of two forms – “native types” and “compound types.” The former is based on the processor architecture and indicates such properties as byte order and the size and format of integers and floating-point numbers. Compound types are user-defined types that may be collections of native types, other compound types, pointers, and arrays (similar to C structures). Profiles can be manually created or, preferably, generated automatically. However, the latter typically requires access to source code and the correct compiler. We describe such an approach in Section 3.3.

Data View Modules. Views represent abstractions on address spaces that define structure within them. Views typically utilize profiles of a particular application or set of applications (such as a program and its dynamically-linked libraries) that are believed to exist within the address space in question. If profiles provide information about “what data looks like,” views provide algorithms to find “where data is located.” We have implemented Linux and Windows kernel views, as well as a “utility” view that performs a number of low-level functions such as searching for human-readable strings, applying hash functions, and pattern matching.

Generally speaking, views implement a set of methods for extracting data specific to a particular application domain. Views may also be composed such that the output of one view is utilized by another, higher-level analysis. While this decomposition is somewhat arbitrary when a single application is considered (since the higher-level analysis could have been part of the application view), the benefit is support for analyses that might be performed on multiple applications.

Data Analysis Modules. Once data has been collected at the view level, analysts are finally able to carry out their primary duty – to make judgments and assessments about the data. Data analysis modules provide analysts with the ability to record and automate certain tasks that they may wish to perform routinely on data collected as part of an investigation. These analyses may be as simple as filtering out relevant objects from large amounts of data or may include complex data aggregation or comparison operations among one or more views. Unlike views, which apply program-level structure to the underlying address spaces, analysis modules help analysts make decisions about the data as it should be interpreted at a semantic level.

GUI and Visualization Modules. Presentation is an important part of analysis [8]. Investigators who can easily navigate and organize their data are more likely to have success during an investigation. Because of the large amounts of information present in physical memory, it is important to provide some basic tools to help analysts navigate address spaces and views while also enabling specific extension where necessary. FATKit provides a graphical user interface (GUI) with some basic features such as loading a particular image and applying address spaces and views to that image. Additionally, we have implemented two visualization modules for presenting address spaces and objects within them. These modules are discussed further in Section 4.5.

3.3 Profile Extraction from C Programs

Automation and generalization are among the most difficult aspects of forensic analysis. To date, the most useful tools for volatile memory analysis have relied on hand-coded, low-level implementations of operating system data structures and algorithms [2, 22, 4]. While these tools provide excellent information about kernel data structures, they suffer from loss of generality when seeking to analyze new data structures in the same kernel, or when changing to a new system with a different kernel version and kernel configuration. Typically, the changes between version releases are minimal and simple [32], but changing even a single member of a data structure will frequently affect the low-level representation of that structure. FATKit deals with version differences by allowing users to define different profiles for different versions. These profiles can be stored and shared for common builds of a particular piece of software, as in the case of closed-source commercial products and binary GNU/Linux distributions. For those systems with custom kernels, or even different configurations of the same kernel, the analyst will be required to either manually produce profiles for each new scenario or develop a mechanism for automating profile generation.

When an analyst has access to C program source and a compiler, it is possible to automate the generation of FATKit profiles. The C programming language supports three ways for users to define new types: the aliasing of already defined types using the `typedef` keyword, the definition of enumeration (`enum`) types, and the definition of compound types (`struct` or `union`). The required profile information for compound types is:

1. The high-level name of the data structure (sometimes anonymous in C)
2. Whether the type is defined as struct or a union
3. The name of each member
4. The type of each member

5. The offset of each member from the start of the type in its compiled form
6. The size of each member in its compiled form

The profile can be viewed as a mapping between the high-level representation of a type and its low-level format. While the C source can provide the first four, only the compiler (or an emulation of it) can produce the final two. One possible solution would be to modify the compiler to produce this mapping as an additional output. A shortcoming of this solution is that it would not be portable to software built with other compilers. Rather than modify the compiler, we have implemented our solution using the C Intermediate Language (CIL) program analysis tool.

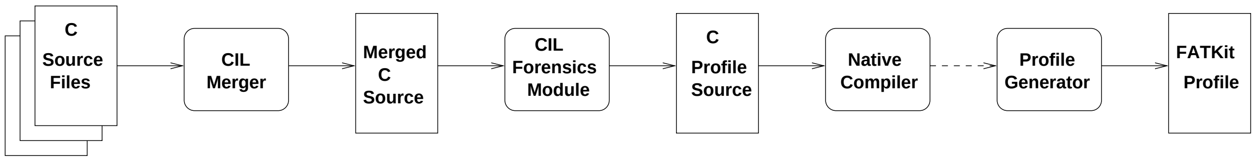


Figure 3: C source profile extraction.

The CIL tool can be used to perform a large number of C source analyses. Its primary feature is the use of an “intermediate language” that represents a normalized subset of the C language. The reduced syntax of CIL allows for simpler constructs and therefore less-complex analyses. The CIL tool has two additional features from which our profile generation can benefit. First, CIL provides a C source “merger” that turns multi-file program builds into a single, large C program. The advantage of merging as it relates to our task is that each data structure to be profiled will only appear once in the merged source, rather than multiple times in the case of a structure defined in a header file included by multiple source files. Second, CIL’s normalized syntax inserts names for anonymous structures, fills in indices for array initializations, and simplifies nested compound type definitions. The resulting data structures compile to the same format, but are much easier to analyze.

```
procedure GENERATEPROFILEGENERATOR($Merged\ source$)
  // Output is a C source file
  $T \leftarrow$ set of all type definitions in$Mergedsource$
  for each definition $d \in T$
    do { Print $d$        // print all type definitions
  Start new main() function
  for each definition $d \in T$
    do { if $d$ isa $struct$ or $d$ isa $union$
           then { Create new variable $v$ of type $d$
                  Print $nameof(d)$        // record the name of the type
                  Print $sizeof(v)$        // have the compiler record the size of the type
                  for each member $m \in d.members$
                    do { Print $nameof(m)$        // record the name of the member
                         Print $typeof(m)$        // record the type of the member
                         Print $addressof(m) - addressof(v)$        // have the compiler calculate the offset
  end main() function
```

Figure 4: Algorithm for generating the profile-generating program.

As shown in Figure 3, our profile generator requires a number of transformations and steps. Fortunately, this process is completely automated. The steps required for profile generation are as follows:

1. Run the CIL merger and translator on the target software. The output of this step produces a single, large CIL source file. For our test environment, the Linux 2.4.32 kernel, the resulting file is over 530,000 lines.

2. Run the CIL profile extraction module on the merged CIL source. The role of the module is to create a new C program with only the original program's type definitions and a main function that prints out the required information for each type. The algorithm for this module is defined in Figure 4. For our Linux test case, the output of this step is over 26,000 lines.
3. Compile the output C file using the same compiler and flags as the program being analyzed. The output will be an executable.
4. Run the newly created executable and capture the output. This output is the profile taken as input to FATKit. For our Linux source example, the output has over 1100 data structures, described in almost 11,000 lines.

The key observation behind our profile automation system is that a source code analysis tool (CIL) can be used to produce a *new program* that, once compiled with a native compiler and run, produces information about *both* the original source's structures and their compiled formats, complete with offsets and sizes. Performing the same task using manual analysis would take far more effort and time.

4 Extending the Analyst's Toolkit with FATKit Modules

As an extensible framework, the ultimate indicator of FATKit's usefulness is the quality of the modules that can be written for it. In this section, we provide an overview of a representative subset of the current set of FATKit modules. While we hope these examples demonstrate the potential capabilities of the FATKit framework, more important than the functionality we have implemented is the ease with which high-level modules can be created within our infrastructure. As an example, our Linux task identification function is a mere 12 lines of Python (Figure 7), including comments and white space. Giving analysts such high-level access to low-level objects is the one of the primary features of FATKit. We now turn our attention to the details of our example modules, starting with the our address space reconstruction module and working our way up through higher levels of abstraction.

4.1 Intel IA-32 Virtual Memory Module

In order to extract structure from physical memory, it is necessary to first understand the programming model provided by the CPU for access to that memory. The IA-32 processor architecture provides two memory management mechanisms, segmentation and paging, which may be used in combination [28]. Segmentation is a mandatory mechanism for dividing a single linear address space into different sections of variable length. Segments may overlap or may partition the linear address space completely. IA-32 memory locations are addressed using a 16-bit segment selector, which identifies a particular segment descriptor, and a 32-bit offset into the specified segment. The segment descriptor is an in-memory data structure that defines the beginning, end, and permissions for a given segment. While segmentation is mandatory, most commodity operating systems simply define a set of overlapping segments to create the appearance of a single "flat" linear address space [3]. Because most operating systems do not make use of IA-32 segmentation (beyond the required minimum), we have focused our implementation efforts on paging.

The use of paging, the IA-32's second memory mechanism, allows each 32-bit linear address space to be broken up into fixed-length sections, called pages, and mapped to physical memory in an arbitrary order. When the default 4KB page size is used, this mapping, shown in Figure 5, utilizes two in-memory data structures known as "page directories" and "page tables." In the translation process, the 32-bit virtual address is broken into three sections, each of which is used as an index into the paging data structures or the physical page. Pages of size 4MB are also supported (but not shown), in which case only the page directory is required. By using different page directories and/or tables for different processes, an operating system

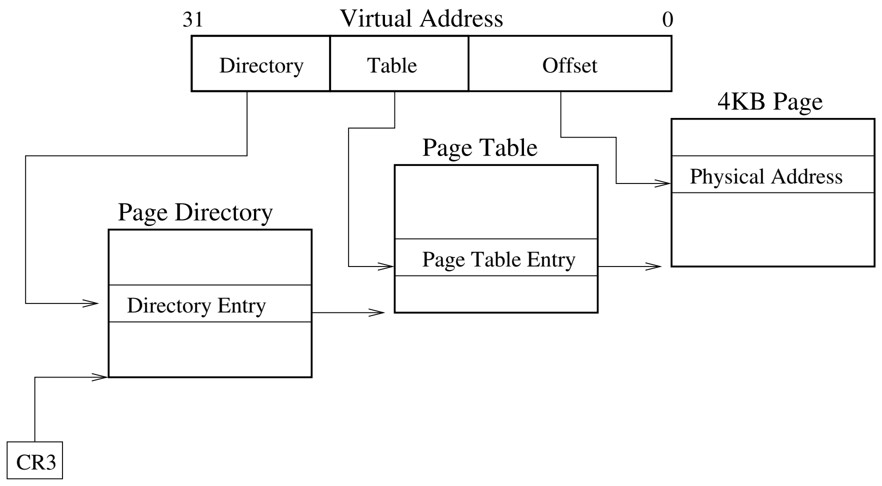


Figure 5: Virtual to physical translation in IA-32 with 4KB pages (adapted from [28]).

can provide the appearance of a single-programmed environment to each process while fully supporting multi-programming.

Our Python implementation of IA-32 paging supports both 4KB and 4MB pages and correctly performs virtual to physical address translation, as tested on physical memory images collected from multiple versions of Linux and Windows kernels running on different IA-32 machines. In addition to simple address translation and page file support, the interface also provides a generic `read()` function for reading arbitrary amounts of data within or across virtual page boundaries. This feature allows higher-level analyses to interact with the underlying physical memory completely within the context of the emulated virtual address space. Additionally, we have implemented functions to provide a list of virtual pages that are currently in physical memory, thereby providing higher-level modules a clear view of the in-memory footprint of a given virtual address space. The only required inputs to our `IA32PagedMemory` class are an underlying physical address space and a value for `cr3`, the register used to locate the page directory within physical memory. Finally, when used in conjunction with a swap or paging file, the IA-32 module also provides the ability to access pages that were not memory-resident at the time of data acquisition.

4.2 Linux Kernel View Module

Built on top of our address space module and our generic object framework, our Linux View module provides access to kernel data objects at a level of abstraction comparable to that of the Linux kernel's C source code. Thus far, we have implemented a number of accumulator functions such as obtaining tasks in the all-tasks list and run/wait queues, listing all modules in the module list, gathering virtual filesystem data, and aggregating network socket information.

Machine Type:	IBM ThinkPad T30
RAM:	512MB
Processor:	Intel Pentium 4 Mobile CPU 1.80GHz
Operating System:	Redhat GNU/Linux 9.0, Linux 2.4.32 kernel
Collection mechanism:	GNU `dd` from `/dev/mem`

Table 1: FATKit test machine summary.

The test system used for this discussion is described in Table 1. A RedHat GNU/Linux 9.0 default installation was performed and a "vanilla" Linux 2.4.32 kernel was built and installed using the default RedHat kernel configuration file. Using this configuration file, we ran our CIL profile extraction tool on the Linux 2.4.32 source, as described in Section 3.3. The `dd` [23] system utility was used to extract a memory image from the Linux `/dev/mem` physical memory interface. The inputs to FATKit were therefore:

1. The automatically extracted profile
2. The Linux `System.map` file from the built 2.4.32 kernel
3. The `dd` image file

To demonstrate the expressibility of our object model and the ease with which views can be implemented, we now present a code example from our Linux View. Before introducing our function's source code, we first briefly introduce the Linux kernel's internal representation of system tasks.

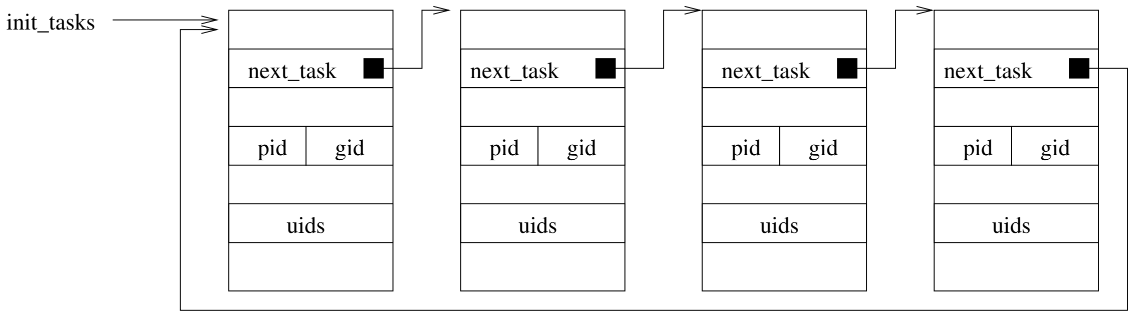


Figure 6: Linux all-tasks list.

In Linux 2.4 kernel versions, tasks are represented using the `task_struct` C structure. Every running task within the system has exactly one instance of this structure defined somewhere in kernel memory. The all-tasks list, depicted in Figure 6, is represented in kernel memory as a circular linked-list. The `next_task` member of the structure is a pointer to the next task in the list. The Linux symbol `init_tasks` provides the location of a static pointer containing the address of the first member of this list. If able to traverse the Linux all-tasks list in a collected memory image, a forensics investigator is able to list all (not otherwise hidden) tasks that were active in the running system. As we now demonstrate, FATKit is well-suited for extracting this type of data.

```python
def getTasks(view):
    addr_space = view.get_address_space()
    symtab = view.get_symtab()
    profile = view.get_profile()

    thread_start_vaddr = symtab.lookup('init_tasks')
    # the first member of init_tasks is a pointer to the first task
    init_task = profile.create_object(['pointer', ['task_struct']], \
                                       addr_space, thread_start_vaddr)
    # Obtains the first member through dereferencing
    head = init_task.dereference()
    return linked_list_collect(profile, head, "next_task",init_task.value)
```

Figure 7: Linux View `getTasks()` function (Python).

Figure 7 presents the `getTasks()` function, as implemented in our current Linux View module. The goal of this function is to traverse the Linux all-tasks list and return the set of Python objects that describes the low-level task instances in the kernel's virtual address space. Typically, the list returned by this function would be used as input to either a higher level analysis or to one of of the GUI/visualization modules for presentation to the analyst. When the function is called, it is passed a single argument called `view`, which is a Python object that has already been setup as a view of an IA-32 address space (recall that views are abstractions on a particular address space). This address space is accessed in line 2 above. While not shown in the figure, for our initial analysis the Linux symbol `swapper_pg_dir` has been used to identify the first set of page tables. This symbol represents the kernel's "master page tables," which do not provide any information about pages mapped within the "userspace" sections of a process' address space, but provides mappings for all kernel memory regions. Since all `task_struct` objects exist in kernel memory, this set of page tables is sufficient to begin our analysis. Lines 3 and 4 obtain the current view's symbol table and object profile respectively. After performing a symbol lookup in line 6, line 8 is the first demonstration of the usefulness of our object model. The profile function `create_object` is called with three parameters, the first of which describes the type of object to be created. In our type language, the notation `['pointer', ['task_struct']]` represents a pointer to a C task structure. The second and third arguments simply identify the address space in which the object is to be created and the offset at which the object exists (in this case extracted from the `System.map` file). In line 11, the pointer object is dereferenced in order to obtain the first `task_struct` in the all-tasks list. Because the profile contains information about all compound types in the Linux kernel, it is able to return the correct type of object. Finally, in line 12, our generic linked-list function is called and its result returned. This function iterates through a list starting with the object `head`, obtains its next value from the member named `next_task` in each object, and ends when it reaches the first member (represented by the `init_task` pointer object).

The above example has been tested and shown to correctly extract the Linux all-tasks list in a number of Linux 2.4 kernel versions. Additionally, we have implemented the same functionality for the slightly modified Linux 2.6 process accounting system, including support for Linux `list_head` embedded pointer lists. Similar functions have been implemented to list all system modules, extract the names of memory-mapped files, and gather information about open network connections. While we cannot exhaustively describe every function we have implemented and tested, we believe this example demonstrates the usefulness of our framework.

4.3 Windows Kernel View Module

Unlike the Linux kernel, source code for the Microsoft Windows operating system is not publicly available. While this precludes the use of FATKit's source code-based automatic profile extraction tool, it does not mean that FATKit is ineffective for analyzing Windows systems. Rather, profiles must be generated through some other means. For example, one source of FATKit profiles for binary code is through automated extraction from debugging information. Once created for a particular binary release, a FATKit profile can be distributed for use on any deployment utilizing that release. To demonstrate the generality of our approach, we have created profiles for three versions of the Microsoft Windows operating system: Windows 2000, Windows 2003 Server, and Windows XP.

As with Linux, using these profiles, we have implemented a significant number of operating system data accumulation functions for Windows kernel objects, such as processes, threads, modules, and drivers. To test our Windows functionality, we utilized the memory images provided for the Digital Forensics Research Workshop's (DFRWS) 2005 Memory Challenge [14] and validated our results with the outputs from the winning teams' tools. In fact, we have built interface modules that generate outputs identical to the tools submitted by Betz [2] and Garner [22]. These modules were built on top of the FATKit framework to demonstrate its versatility. In addition to the DFRWS images, we have also tested our Windows kernel view module on images taken from a range of Windows installations using a variety of data acquisition mechanisms.

4.4 Advanced Detection Data Analysis Module

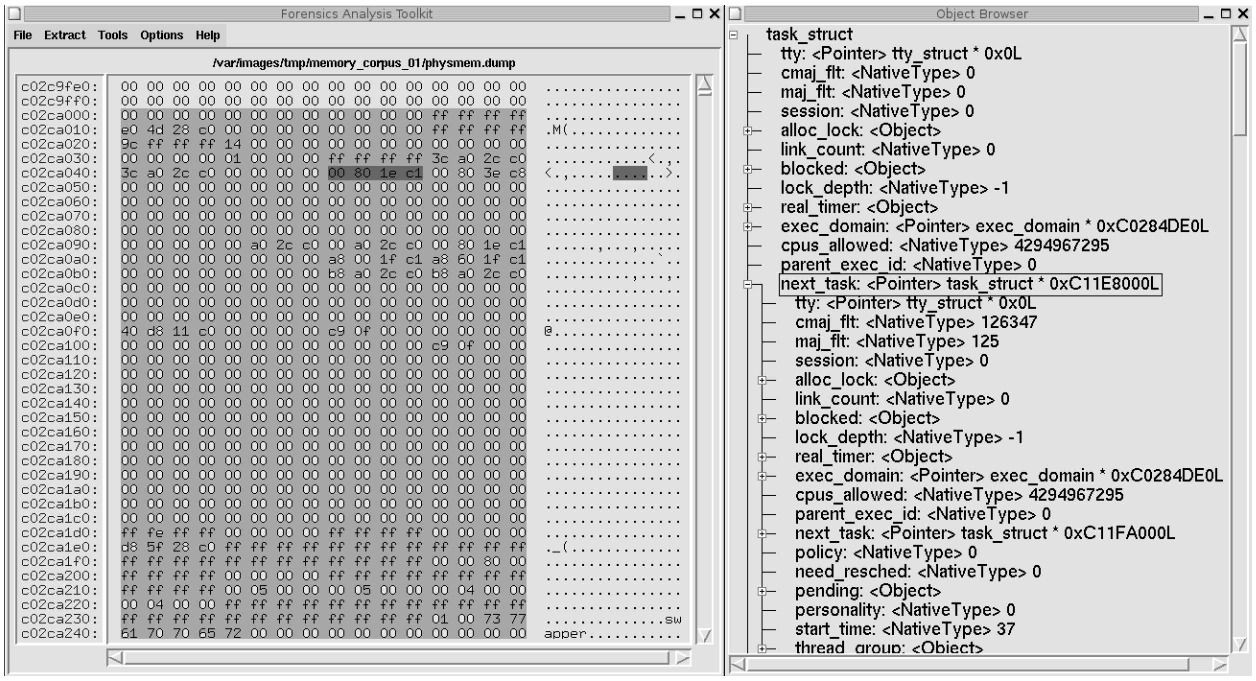
Built on top of our Linux and Windows kernel view modules, FATKit's advanced detection module provides mechanisms for automating and repeating the task of searching for anomalous or potentially interesting data conditions. While the ability to list processes or open files is useful in and of itself, the ability to automate analyses such as identifying signs of intrusion or detecting anomalous activity is far more valuable to the analyst. Because incident response is one of the primary applications of forensic analysis, we have implemented a wide range of detection algorithms that look for evidence of malicious or unexpected activity [36]. Among the targets of our current detection suite are kernel and userland malware (e.g., rootkits, viruses, trojans), injected or modified code and data, and inconsistent data conditions such as hidden processes or drivers. One example of the types of checks performed by our advanced detection module is a search for evidence of "anti-forensics" techniques such as DLL injection in the kernel and userspace portions of system processes [43]. These techniques, which are frequently employed by frameworks such as Metasploit [31], are commonly regarded as undetectable by traditional non-volatile forensic analysis procedures. By allowing experts to easily write detection algorithms for these techniques, FATKit raises the bar for adversaries seeking stealth and reduces the time it takes for analysts to determine what has occurred.

4.5 Visualization Modules

In addition to the expressive object model and low-level object extraction capabilities provided by FATKit, one of the most novel and useful features of the toolkit is its support for data visualization modules. It is our belief that data presentation, particularly if provided in an interactive manner, can greatly increase the productivity of analysts. While scripting high-level analyses is very useful, particularly for reproducing results or applying the same algorithms to multiple data sources, the ability to interact with a data source visually allows an analyst to get a feel for the relationships between digital objects and how they appear in memory. We present here two useful modules that we have implemented for visualizing data at two important layers of abstraction – an address space and the objects within it. We refer to these modules as the "address space browser" and "object browser" respectively.

Address Space Browser. The address space browser provides a linear representation of the raw data in a given address space. The primary features of this module are the decoding of raw data as hexadecimal and ASCII values, the color-coding of requested objects depending on the current context, and the pictorial representation of which virtual addresses are available and which are not. The screen image shown in Figure 8 displays the address space browser on the left and the object browser on the right. The leftmost column of the address space browser provides hexadecimal offsets into the virtual address space of the requested object, whose contents are displayed in the remaining two columns. The large, highlighted portion of the address space browser represents the Linux `task_struct` object currently being examined within the object browser. The small four-byte field that is darkened within that region is the `next_task` pointer of that object, which has been selected in the object browser.

Object Browser. The object browser provides a mechanism for navigating through memory objects similar to a conventional file system browser for a desktop user environment. Complex objects, such as structures and arrays, can be expanded to display their member objects, which can themselves be expanded if appropriate. Objects with values, so-called "native types" in our nomenclature, are displayed with their corresponding in-memory value. Pointers can be dereferenced by double-clicking on them in our interface,


Forensics Analysis Toolkit
File Extract Tools Options Help
/var/images/tmp/memory_corpus_01/physmem.dump
Object Browser

```
c02c9fe0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02c9ff0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca000:  00 00 00 00 00 00 00 00 00 00 00 00 ff ff ff ff   ................
c02ca010:  e0 4d 28 c0 00 00 00 00 00 00 00 00 ff ff ff ff   .M(.............
c02ca020:  9c ff ff ff 14 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca030:  00 00 00 00 01 00 00 00 ff ff ff ff 3c a0 2c c0   ............<.,.
c02ca040:  3c a0 2c c0 00 00 00 00 00 80 1e c1 00 80 3e c8   <.,...........>.
c02ca050:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca060:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca070:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca080:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca090:  00 00 00 00 00 a0 2c c0 00 a0 2c c0 00 80 1e c1   ......,...,.....
c02ca0a0:  00 00 00 00 00 00 00 00 a8 00 1f c1 a8 60 1f c1   ..............`..
c02ca0b0:  00 00 00 00 00 00 00 00 b8 a0 2c c0 b8 a0 2c c0   ..........,...,.
c02ca0c0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca0d0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca0e0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca0f0:  40 d8 11 c0 00 00 00 00 c9 0f 00 00 00 00 00 00   @...............
c02ca100:  00 00 00 00 00 00 00 00 00 00 00 00 c9 0f 00 00   ................
c02ca110:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca120:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca130:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca140:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca150:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca160:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca170:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca180:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca190:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca1a0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca1b0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca1c0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c02ca1d0:  ff fe ff ff 00 00 00 00 ff ff ff ff 00 00 00 00   ................
c02ca1e0:  d8 5f 28 c0 ff ff ff ff ff ff ff ff ff ff ff ff   ._(.............
c02ca1f0:  ff ff ff ff ff ff ff ff ff ff ff ff 00 00 80 00   ................
c02ca200:  ff ff ff ff 00 00 00 00 ff ff ff ff ff ff ff ff   ................
c02ca210:  ff ff ff ff 00 05 00 00 00 05 00 00 00 04 00 00   ................
c02ca220:  00 04 00 00 ff ff ff ff ff ff ff ff ff ff ff ff   ................
c02ca230:  ff ff ff ff ff ff ff ff ff ff ff ff 01 00 73 77   ..............sw
c02ca240:  61 70 70 65 72 00 00 00 00 00 00 00 00 00 00 00   apper...........
```

```
task_struct
  tty: <Pointer> tty_struct * 0x0L
  cmaj_flt: <NativeType> 0
  maj_flt: <NativeType> 0
  session: <NativeType> 0
  alloc_lock: <Object>
  link_count: <NativeType> 0
  blocked: <Object>
  lock_depth: <NativeType> -1
  real_timer: <Object>
  exec_domain: <Pointer> exec_domain * 0xC0284DE0L
  cpus_allowed: <NativeType> 4294967295
  parent_exec_id: <NativeType> 0
  next_task: <Pointer> task_struct * 0xC11E8000L
    tty: <Pointer> tty_struct * 0x0L
    cmaj_flt: <NativeType> 126347
    maj_flt: <NativeType> 125
    session: <NativeType> 0
    alloc_lock: <Object>
    link_count: <NativeType> 0
    blocked: <Object>
    lock_depth: <NativeType> -1
    real_timer: <Object>
    exec_domain: <Pointer> exec_domain * 0xC0284DE0L
    cpus_allowed: <NativeType> 4294967295
    parent_exec_id: <NativeType> 0
    next_task: <Pointer> task_struct * 0xC11FA000L
    policy: <NativeType> 0
    need_resched: <NativeType> 0
    pending: <Object>
    personality: <NativeType> 0
    start_time: <NativeType> 37
    thread_group: <Object>
```

Figure 8: FATKit address space and object browsers.

causing the referenced object (assuming it exists in a valid address range) to be added to the object browser list. Objects can be displayed in the browser by executing a view module function, such as our Linux task listing example, or by selecting a region in the address space browser and choosing "decode-as" with the desired object type. This latter functionality provides analysts with the ability to hypothesize about the structure of yet-to-be-identified data and to test those hypotheses interactively and immediately.

In Figure 8, the analyst is investigating the first member of the all-tasks list, the `swapper` kernel thread, in the object browser window. The window displays a list of all elements in the structure, including the type and value of those elements. Additionally, an ability to expand compound member elements is provided to the user. The selected item, `next_task`, is a pointer to the next `task_struct` in the all-tasks list. Note that the user has "expanded" this pointer to view the dereferenced object. In this way, the analyst can arbitrarily follow object pointers in kernel memory with the click of a button.

The functionality provided by these two modules is far more interactive than any existing approach to the forensic analysis of volatile memory proposed thus far. Additionally, we are currently investigating more advanced visualization modules such as depicting virtual to physical page mappings and representing data structure dependency relationships.

5 Related Work

While still an evolving field, the past few years have seen a marked increase in the amount of attention given to digital forensics from the point of view of both incident response and law enforcement. Much of the focus to date has been on defining and analyzing well-formed methodologies and processes for obtaining a level of sophistication analogous to that of the physical forensics world [34]. However, in addition to the questions of processes and models, the topics of low-level technical analysis have also made great strides, particularly in the area of non-volatile disk memory and their filesystems [16, 10, 17]. This work has laid the foundation for technical analysis within the field.

As described in Section 3.1, Carrier's work towards defining abstraction and the complexity problem within digital forensics [7] was one of the first attempts to view the creation of forensics tools such as FATKit in a more general manner. His explanations of abstraction within forensic data provide insight into the methodologies employed by FATKit and other examination and analysis tools.

Unlike the advanced tools available for filesystem forensic analysis, prior to FATKit, work in the area of volatile system memory was yet to produce a general tool for managing this complex data. However, foundational work on the analysis of operating systems and the manual extraction of in-memory structure has motivated and guided the development of FATKit. In [17], Farmer and Venema provide an excellent overview of the challenges and evidential potential of volatile memory forensics. Their techniques and experiments related to identifying files, or portions of them, and their discussion of the half-life of digital evidence in physical memory are particularly compelling. Recent work towards more detailed analysis of particular operating systems has been presented by Betz [2], Garner [22], and Burdach [4]. This work has largely taken the form of manually-coded programs focused on specific low-level objects.

Since we first developed FATKit, a number of others have produced a substantial amount of online work and discussion about specific techniques for extracting data from volatile memory dumps. FATKit was designed to generalize and improve on these types of tools, while still providing access to just as much low-level detail. To demonstrate the generality of our tool, we have implemented each of the following techniques without any change to our underlying framework and without much implementation effort. Much of this work, which we now summarize, has provided valuable insight to how particular systems work rather than to the general problem of volatile memory analysis. Schuster has discussed a number of topics related to converting virtual to physical addresses, reconstructing program binaries, and linearly searching for `EPROCESS` and `ETHREAD` objects in memory [1]. He has also created a tool, PTfinder, for linearly searching memory for `EPROCESS` and `ETHREAD` structures in memory images of systems running Microsoft Windows 2000 SP4 that is capable of generating a process graph. Carvey recently released tools for linearly searching memory for processes and threads, dumping process details, and dumping a process' memory to a file. Additionally, he has released a tool for parsing PE headers [26]. Stewart has created a tool, pmodump, for reconstructing a processes virtual memory space once it has been collected from a physical memory dump. Unlike the previous tools, which scan for `EPROCESS` structures, this tool works by scanning for page directory blocks [30]. Memory Parser (MMP) is yet another tool that was designed to analyze process memory dumps. It allows the ability to enumerate the DLLs in a process and performs a hash over the first 1024 bytes of the DLLs PE image [40].

In addition to the volatile memory analysis techniques described above, some have focused on the development of tools and techniques for preserving and collecting digital evidence from memory. These approaches are outlined by Carrier and Grand [12] and include hardware-based, firmware-based, and a number of software-based approaches. Tribble [12] is a PCI-based hardware mechanism that is pre-positioned in a machine as a precaution for an eventual incident. Once an incident has occurred, but before the system is powered down, the Tribble device may be activated. The device uses direct memory access to collect the contents of physical memory in an isolated and trustworthy manner. Copilot takes a similar approach, but utilizes DMA to perform online analysis in addition to forensic evidence gathering [35]. Potential software-based solutions include virtual machine introspection [20] and a number of low-level copy programs that utilize operating system-specific interfaces [21, 23, 42, 25].

Finally, a number of approaches have been taken towards automating various aspects of forensics analysis, particularly in the area of forming and testing hypotheses about events [41, 15, 38, 6, 13, 11]. Much of this research has focused on using automation to demarcate systems or identify evidence for further analysis through a number of outlier detection and clustering techniques. Recent work by Garfinkel attempts to automate the analysis of large data sets in the form of filesystems [19]. In this work, "cross-drive analysis" is used to compare corpora of large numbers of disk drives to identify "interesting" and "related" data. To our knowledge, we are the first to utilize automated source code tools to aide in post-mortem system analysis.

Other fields of related work that must be considered are the areas of debugging and crash dump analysis. There are a number of clear similarities between the work performed by a debugger and that required for volatile memory forensics. For example, debuggers offer the ability to traverse through an address space and dump typed objects found in memory. At least one debugger, ddd, even offers the ability to visually traverse data structures in memory [24]. In fact, FATKit leverages these similarities by using the information found in debugging data to augment its profiles when necessary and by utilizing common debugger interfaces in order to provide a more familiar "look and feel" to users. On the other hand, FATKit was also designed as a forensic analysis toolkit and offers features designed to support the needs of the forensic analyst. Unlike conventional debuggers, FATKit is cross-platform and version independent based on automatically generated profiles. The platform and operating system are simply inputs into the system. Finally, FATKit offers the ability to analyze multiple images concurrently and to correlate information across machines regardless of dissimilarities between operating systems or hardware.

6 Future Work

As we continue to develop our methodologies, we intend to maintain focus on the goals of automation, reuse, and visualization. Among the first tasks we plan to undertake is the extension of our automatic profile extraction to applications running in process virtual address spaces. While operating systems are an increasingly important target for malicious activities, attackers continue to penetrate userspace applications for initial entry into the system. Our per-process address space extraction mechanism provides a solid foundation for extending this work to the analysis of user programs. Additional automation tasks that we intend to pursue are the extraction of algorithmic and semantic constructs from program source, attempting to perform binary analysis in order to identify in-memory structure, and extending our automation to other high-level programming languages.

In addition to our automation goals, we will continue to extend our toolkit by investigating possible interactions between filesystem forensics tools and our system. Specifically, there are clear relationships between main memory and secondary storage, such as identifying and analyzing the programs and libraries running within a process' address space and locating swapped memory pages on disk for a more complete process address space analysis. While FATKit currently has modules for rebuilding memory-resident programs and libraries [43], more advanced analysis is required for evaluating the non-static portions of these files. Although FATKit provides modules for integrating swap and page files, more research is needed with regard to acquisition methods for volatile memory and its associated page files; existing methodologies have the potential to lead to temporal inconsistencies.

Finally, since FATKit is a tool intended to be used by analysts and investigators, we will engage real-world practitioners in providing feedback and testing of our techniques. Specifically, we intend to identify additional visualization modules that may help investigators perform analyses we have not yet considered.

7 Conclusion

To address the problem of analyzing low-level forensic data collected from volatile memory in the wake of an incident or crime, we have introduced the Forensic Analysis ToolKit (FATKit). FATKit is a framework for processing system memory images that facilitates the extraction and analysis of digital objects at various levels of abstraction in a manner far more modular and automatic than any system proposed to date. Designed to facilitate high-level analysis by forensics investigators, FATKit provides a set of tools for quickly describing, extracting, and visualizing relevant data from memory images. In particular, the ability to automatically produce digital object definitions from C source code and use those definitions in online analysis, rather than relying on manually written routines, provides an analyst with immediate access to data formats for all objects used within a program or operating system. Additionally, the modular nature of the toolkit allows analysts to quickly extend its functionality in a high-level language to augment their analyses. We have implemented a set of useful modules including IA-32 virtual address space reconstruction, Linux and Windows data extraction, and interactive data visualization.

8 Acknowledgments

The authors would like to thank Michael Hicks and Iulian Neamtiu for their guidance and examples with writing CIL modules. The authors would also like to thank Brian Carrier for his insightful discussions.

References

[1] Andreas Schuster. International Forensics Blog, June 2006. Available at: http://computer.forensikblog.de.

[2] Chris Betz. *memparser*, 2005. Available at: http://www.dfrws.org/2005/challenge/memparser.html.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., second edition, December 2002.

[4] Mariusz Burdach. *Windows Memory Forensic Toolkit (WMFT)* , 2006. Available at: http://forensic.seccure.net/.

[5] CAIDA. *Analysis of Code-Red*, July 2001. Available from: http://www.caida.org/analysis/security/code-red/.

[6] Megan Carney and Marc Rogers. The Trojan Made Me Do It: A First Step in Statistical Based Computer Forensics Event Reconstruction. *International Journal of Digital Evidence (IJDE)*, 2(4), Spring 2004.

[7] Brian Carrier. Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers. *International Journal of Digital Evidence (IJDE)*, 1(4), Winter 2003.

[8] Brian Carrier. *The Autopsy Forensic Browser*, 2005. Available at: http://www.sleuthkit.org.

[9] Brian Carrier. *File System Forensic Analysis*. Addison Wesley, 2005.

[10] Brian Carrier. *The Sleuth Kit*, 2005. Available at: http://www.sleuthkit.org.

[11] Brian Carrier and Blake Matheny. Methods for Cluster-Based Incident Detection . In *IEEE Information Assurance Workshop (IAW)*, Charlotte, NC, USA, April 2004.

[12] Brian D. Carrier and Joe Grand. A Hardware-Based Memory Aquisition Procedure for Digital Investigations. *Journal of Digital Investigations*, 1(1), 2004.

[13] Brian D. Carrier and Eugene H. Spafford. Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence. In *Proceedings of the 2005 Digital Forensic Research Workshop (DFRWS)*, 2005.

[14] Digital Forensic Research Workshop (DFRWS). *Memory Analysis Challenge*, 2005. Available at: http://www.dfrws.org/2005/challenge/index.html.

[15] Christopher Elsaesser and Michael C. Tanner. Automated Diagnosis for Computer Forensics. Technical report, The MITRE Corporation, August 2001.

[16] Dan Farmer and Wietse Venema. *The Coroner's Toolkit (TCT)*, 2004. Available at: http://www.porcupine.org/forensics/tct.html.

[17] Dan Farmer and Wietse Venema. *Forensic Discovery*. Addison-Wesley, 2005.

[18] French Institue for Research in Computer Science and Control (INRIA). *Objective Caml*, 2006. Available at: http://caml.inria.fr/ocaml/index.en.html.

[19] Simson L. Garfinkel. Cross-Drive Analysis and Forensics. In *2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006. Submitted.

[20] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *The 10th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, February 2003.

[21] George Garner. *Forensic Acquisition Utilities*, 2004. Available at: http://users.erols.com/gmgarner/forensics/.

[22] George M. Garner Jr. *kntlist*, 2005. Available at: http://www.dfrws.org/2005/challenge/kntlist.html.

[23] GNU Coreutils. *dd*, 2005. Available at: http://www.gnu.org/software/coreutils/.

[24] GNU Project. *Data Display Debugger*, 2006. Available at: http://www.gnu.org/software/ddd.

[25] Nick Harbour. *dcfldd*, 2005. Available at: http://dcfldd.sourceforge.net/.

[26] Harlan Carvey. Windows Incident Response, June 2006. Available at: http://windowsir.blogspot.com.

[27] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, July 2005.

[28] Intel Corporation Volume 3: System Programming Guide. IA-32 Intel Architecture Software Developer's Manual, September 2004. Order Number: 253668.

[29] ISS X-Force. *SQL Slammer worm propagation*, January 2003. Available from: http://xforce.iss.net/xforce/xfdb/11153.

[30] Joe Stewart. pmodump.pl v0.1, June 2006. Available at: http://www.lurhq.com/truman.

[31] H D Moore and Skape. Metasploit Project, June 2006. Available at: http://www.metasploit.com.

[32] Iulian Neamtiu, Jeffrey Foster, and Michael Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *MSR 2005: International Workshop on Mining Software Repositories*, Saint Louis, MO, May 2005.

[33] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Computational Complexity*, pages 213–228, 2002.

[34] Gary Palmer. A Road Map for Digital Forensic Research. Technical Report DTR-T001-01, DFRWS, November 2001. Report From the First Digital Forensic Research Workshop (DFRWS).

[35] Nick L Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[36] Nick L Petroni, Timothy Fraser, AAron Walters, and William A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *15th USENIX Security Symposium*, Vancouver, B.C., Canada, August 2006.

[37] Python Software Foundation. *The Python programming language*, 2006. Available at: http://www.python.org.

[38] Tye Stallard and Karl Levitt. Automated Analysis for Digital Forensic Science: Semantic Integrity Checking. In *Proceedings of the 2003 Annual Computer Security Applications Conference (ACSAC)*, 2003.

[39] Sam Stover and Matt Dickerson. Using Memory Dumps in Digital Forensics. *;login: The USENIX Magazine*, 30(6), December 2005.

[40] Tobias Klein. Memory Parser (MMP), June 2006. Available at: www.trapkit.de/research/forensic/mmp.

[41] M. W. Tyson. DERBI: Diagnosis, Explanation and Recovery from Computer Break-ins. Technical report, SRI Artificial Intelligence Center, January 2001. Available from: http://www.dougmoran.com/dmoran/PAPERS/DerbiFinal.pdf.

[42] Wietse Venema. *memdump*, 2003. Available at: http://www.porcupine.org/.

[43] AAron Walters. FATKit: Detecting Malicious Library Injection and Upping the Anti, July 2006. Available at: http://www.4tphi.net/fatkit/.