5-2012

Forensic Carving of Wireless Network Information from the Android Linux Kernel

Brendan D. Saltaformaggio
University of New Orleans

Forensic Carving of Wireless Network Information

from the Android Linux Kernel

An Honors Thesis

Presented to

the Department of Computer Science

of the University of New Orleans

In Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science, with

Honors in Computer Science

by

Brendan D. Saltaformaggio

May 2012

Acknowledgement

I would like to thank my professors for the wealth of knowledge that I have gained since beginning my studies in Computer Science at the University of New Orleans. Most importantly, I give special thanks to my advisor, Dr. Golden G. Richard III, for passing on to me his knowledge, attitude, and love of security research and also for his support and advice. I am extremely thankful to Dr. Daniel Bilar who got me started in Information Assurance research and who remains a close friend. And finally, I must give credit to Dr. Jaime Niño, for teaching an arrogant student who thought he knew how to program that he was completely wrong. To these teachers I owe everything.

I also want to thank everyone in our Information Assurance group, especially Joe Sylve. Joe taught me everything he knew about this topic and patiently answered my endless questions.

Finally, I would like to thank my family and friends who have always supported me. My love and gratitude go to my mother, Jody, and my girlfriend, Kristen, for never getting tired of hearing about kernels, bits, and segments despite not understanding what they are.

Table of Contents

List of Figures

Abstract

Modern smartphones integrate ubiquitous access to voice, data, and email communication and allow users to rapidly handle both personal and corporate business affairs. This is possible because of the smartphone's constant connectivity with the Internet. Digital forensic investigators have long understood the value of smartphones as forensic evidence, and this thesis seeks to provide new tools to increase the amount of evidence that one can obtain and analyze from an Android smartphone. Specifically, by using proven data carving algorithms we try to uncover information about the phone's connection to wireless access points in a capture of the device's volatile memory.

Keywords: Android forensics, memory forensics, memory analysis, Linux.

Chapter 1 - Introduction

1.1 Motivation

Modern smartphones such as the Apple iPhone and Google Android handsets are powerful handheld computers that have become an everyday part of many people’s lives. These devices are often by our side at all times, all the while handling vast amounts of data. In a criminal investigation this data becomes evidence and it is up to digital forensics analysts to retrieve and examine as much of it as possible.

In order to inspect or even obtain evidence from a smart phone, an investigator must have a large arsenal of tools at his or her disposal. The development of these tools is still largely driven by research-grade work done in academia, and this has created a race among forensics researchers to design tools and methodologies to investigate smart phones as evidence. This initiative to find, acquire, and study as much evidence as possible hidden in a smart phone is what motivated this work.

1.2 Android Operating System

In this honors thesis, we will concentrate on the examination of Android devices exclusively for a number of reasons. First, Android devices have a majority share of the mobile market [1]. Secondly the Android operating system is built on a slightly modified Linux kernel. This gives an added bonus that any tool designed for use on an Android device can, with some modification, be used on a Linux computer. This importance will be revisited in chapter 4. Finally, Android devices were made available by the Greater

New Orleans Center for Information Assurance (GNOCIA) at the University of New Orleans, where the research was conducted.

1.3 Contributions

The work in this thesis provides a step forward in increasing the amount of evidence that one can obtain and analyze from an Android smartphone. The goal of this research was, given only a context-free memory image from an Android device, to identify which wireless network access points the device was near when the image was taken. This information could also lead investigators to geolocate where the device was when the image was taken.

This thesis makes two contributions to the field of Android device forensics. First, it provides two new toolsets to an investigator to obtain and analyze Android or Linux kernel network structures in memory dumps. These tools will be discussed in detail in chapter 3. Secondly, the investigation that went into developing these tools has provided interesting insight into the utility and limitations of using the method described in this thesis. As with most research in the field of smartphone forensics, there were a number of difficulties and limitations that make these tools useful under most, but not all, investigative scenarios. The particulars of these limitations are also discussed in chapter 3.

1.4 Organization

The remainder of this thesis is organized into five additional chapters. Chapter 2 will cover other works related to network structure carving and memory forensics.

Chapter 3 will detail the investigation undertaken to develop our forensic tools to aid an investigator in discovering wireless network data in a memory capture. Chapter 4 will discuss some limitations which were faced while developing these tools. Results from testing the forensic tools against actual memory images taken from both a Linux workstation and a Droid2 smartphone will be shown in chapter 5. Finally, chapter 6 will conclude and list future goals related to this research.

Chapter 2 – Related Works

2.1 Network Structure Carving

A similar approach to a different problem set was undertaken by Beverly et. al. [2]. In [2] the authors attempted to gain information about the network activity of a target computer. To do this, the authors used data carving techniques to find, extract, and analyze IP packets, socket structures, and ethernet frames from non-volatile storage devices. In the process they found that a large amount of contextual information about the machine could be determined such as local ethernet settings, geolocation, and conversations with remote machines.

The work in [2] was later extended by Gregory Cardwell (one of the original authors) and applied to the Android smartphone's storage devices [3]. Again the author focused on finding network metadata in the device's non-volatile storage. Because of the many similarities between Android smartphones and modern computers, the same approach from [2] works in [3], and again allows the author to discover network artifacts in the non-volatile storage of Android devices.

The main difference between their work and this research is that [2] and [3] primarily rely on structures created during a conversation between the target computer and other machines. This introduces a high probablilty of false negatives (i.e. entirely missing a network device because it did not communicate recently with the target). In contrast, our work relies on information that is constantly maintained by the kernel and wireless NIC drivers in an Android operating system. Thus if the phone can detect a wireless access point (i.e. which it has stored in the device's memory) then our implementation will be able to find it.

Additionally, our work differs from theirs in the storage medium that we are interested in. In [2] and [3] the authors chose to apply their structure carving technique to only non-volatile storage such as hard disk images, hibernation files, and swap files. Again, we believe that this leads to a high probablilty of false negatives. This is because it is possible that all IP packets and ethernet frames for a given network conversation do not get written to a non-volatile storage device. In this scenario, the approach taken in [2] and [3] would be unable to detect that the conversation took place. In this research we focus on searching snapshots of an Android device's volatile memory for the kernel structures of interest. If the structures exist then they must be in the device's memory and so they are guaranteed to be found in that snapshot.

The final difference between the two cited works and this thesis is the scope of the investigation. The authors of [2] and [3] are primarily interested in finding network metadata associated with network conversations between two machines. The scope of our investigation is much smaller, but more detailed. The goal of this research is to identify the wireless access points that the device was in contact with when the memory image was taken.

2.2 Mobile Memory Forensics

The research done for this thesis was made possible by the prior Android device forensics conducted at the University of New Orleans and the Greater New Orleans Center for Information Assurance. Specifically, the research done in [4] was the first known successful process for extracting a copy of physical memory from an Android device. Without this work, there would be no reliable method for obtaining a copy of physical memory and so this research would only be applicable to Linux computers with wireless NICs.

All captures of physical RAM (from both Android devices and Linux computers) were obtained using dmd [4]. This tool inserts a simple kernel module into the running Android/Linux kernel and copies the contents of physical memory out to a file. This file, containing a copy of the physical memory, is then used for analysis.

Also demonstrated in [4] is some basic analysis of Android memory images using the Volatility framework [5], but this was mainly done to demonstrate the functionality of dmd as a memory extractor. This research should be considered an extension of the analysis shown in [4] because we focus instead on providing tools to extract additional information from a memory image – information that has not, until now, been recoverable.

The only other known tool which is capable of analyzing Android kernel objects in memory captures is Volatilitux [6]. This program provides only limited analysis capabilities such as enumeration of running processes, memory maps, and open files. It does not include the functionality to display network information of any kind. One main contribution of this thesis is a plugin that implements the carving of wireless access point data and adds this to the already extensive functionality of the Volatility framework [5].

2.3 Volatile Memory Forensics

In the last few years, the amount of research being done in the analysis of volatile memory has grown exponentially. To attempt to cover even a fraction of this research would be beyond the scope of work related to this thesis. For this reason we choose to only include the work done in this field that is directly used in this thesis.

The Volatility framework [5] is a program (more specifically, a collection of programs) that allows for easy analysis of memory captures. The primary advantage of using the Volatility framework is its division of duties. Volatility allows plugins to implement certain analysis functionality independent of the memory image specific details. This allows a plugin, for example, to be written to analyze Linux memory captures without the knowledge of which specific version of Linux will be inside the memory image. Volatility also allows users to design “address spaces” for different platforms. This allows the Volatility framework to handle, parse, and run its plugins on memory captures from a new Linux kernel version, for example.

One contribution of this thesis is a Volatility framework that finds, carves, and analyzes wireless access point data from an Android/Linux memory image. Because Android is built on a regular Linux kernel, the plugin for Volatility will work without modification on both platforms. A user would only need to implement a new “address space” for a given phone or Linux distribution.

Chapter 3 – Implementation

3.1 Methodology

The primary goal of this research was, given only a context-free memory image from an Android device, to identify which wireless network access points the device was near when the image was taken. To do this we first needed to find how that information gets into the device's RAM.

We began by reviewing the Linux kernel source that handles requests by processes for wireless access point data. It turns out that this is an I.O. control (`ioctl`) function call against an internet socket handle. This `ioctl` function call switches into kernel context, traces through the higher level socket I.O. control code, and finally calls down into the Wireless Extensions [7] [8] I.O. control handling code. The exact kernel code flow is shown in Figure 8 in Appendix A.

The `ioctl` function of interest is called on an internet socket handle with a command argument of `SIOCGIWSCAN` (corresponding to hexadecimal value of 0x8B19) and a pointer to a caller supplied buffer to fill with the wireless access point data. The full list of possible `ioctl` function commands that Wireless Extensions can handle is listed in `wireless.h` [9] from the Linux kernel source, but `SIOCGIWSCAN` in particular is the command for "get scanning results" as documented in [9].

When the `ioctl` function finally calls down to the `ioctl_standard_iw_point` function in `wext_core.c` [10]from the Linux kernel source, the request is handed off to driver specific code. The driver specific code is then tasked with filling a kernel supplied buffer with the wireless access point data that has been detected by the device's most recent scan. Since potentially every different Android device could have a different wireless NIC driver we do not assume access to or knowledge of how the request is handled beyond the `ioctl_standard_iw_point` function. However, we do not need to be concerned with the driver specific implementation that handles this request. This is because the kernel expects the answer from the driver to be in a predefined, specific format. Additionally, once the kernel receives its answer from the driver, the same format is expected by the process that originally invoked the `ioctl` function call.

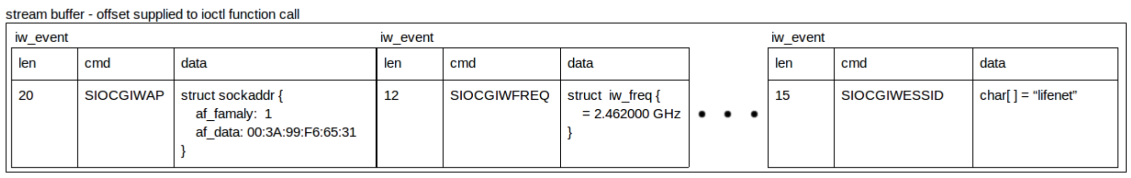
This specific format that the driver uses to return its answer is a combination of kernel objects. The outermost object is the buffer that the kernel passes into the driver specific request handling function. This is referred to as `char* stream` in the code and so we will refer to it as "the stream" in this thesis. The stream is then loaded by calls to the `iwe_stream_add_event` or `iwe_stream_add_point` functions[1] in `iw_handler.h` [11] from the Linux kernel source.

Starting at the beginning, the stream is filled with `iw_event` objects. An `iw_event` (defined in `wireless.h` [9]) is a structure that contains three fields, in this order: a length, a command, and the data. The length field contains the entire length in bytes of the `iw_event`. The command field contains a command from the list given in `wireless.h` [9]. Some possible values of this field are `SIOCGIWFREQ` which

[1] We ignore the `iwe_stream_add_value` function because not only does its use make that `iw_event` impossible to parse, but also we never found it called in any drivers used in this experiment. Perhaps this is because the function is commented by "Be careful, this one is tricky to use properly" [10].

corresponds to "get channel/frequency (Hz)" or SIOCGIWAP which corresponds to "get access point MAC addresses." The final field is the data field, more specifically this field is an iwreq_data object (defined in wireless.h [9]). This field can contain a large number of concrete values, but luckily for us the type of this field's data is governed by the preceding command field. For the previous examples: If the command field contained SIOCGIWFREQ then the iwreq_data object would be of type iw_freq (a simple object defined in wireless.h [9] that describes a wireless broadcast/scanning frequency. If the command field contained SIOCGIWAP then the iwreq_data object would simply be a string of characters containing the access point's MAC address. Figure 1 contains a depiction of how a stream would look once it has been filled by the driver.

These iw_event objects are placed one after another (with no space in between) inside the stream until either the stream is full or the driver has described all the features of every access point it has detected. In the former case (the stream gets filled) the driver will return the E2BIG error code. If all the iw_event objects fit inside the stream then the driver returns the stream to the kernel full of data. The kernel then copies the stream from its buffer into the buffer supplied by the process to the original ioctl function call. This discovery is of significant value because we now have two potential copies of the data that we want to carve in memory.


stream buffer - offset supplied to ioctl function call

iw_event			iw_event				iw_event		
len	cmd	data	len	cmd	data		len	cmd	data
20	SIOCGIWAP	struct sockaddr { af_famaly: 1 af_data: 00:3A:99:F6:65:31 }	12	SIOCGIWFREQ	struct iw_freq { = 2.462000 GHz }	• • •	15	SIOCGIWESSID	char[ ] = “lifenet”

Figure 1. *iw_event* stream in memory.

3.2 Design

Once we figured out the structure of the stream, we could then design an algorithm to find and carve the structure out of memory images. The algorithm needs to find the streams in memory by finding and validating consecutive `iw_event` objects. If enough `iw_event` objects are found one after another in a consecutive block of memory then it would be considered a stream and the collection of `iw_event` objects is carved out of the memory image. This should be generic enough to handle streams of `iw_event` objects without prior knowledge of which types of `iw_event` objects the driver had added to the stream.

The process begins with scanning the memory image in 16 bit increments. These segments must be within the range of possible `iw_event` commands. If the segment is not, then the next 16 bits are checked. Once a segment of the memory image is found within that range, then the preceding 16 bits are tested to be between the minimum and a maximum possible length for an `iw_event` object. If these two checks pass then the section of memory is considered a possible `iw_event` object. For example: If the scanner finds a section of memory at address Y that contains the value 0x8B1B (corresponding to `SIOCGIWESSID`, commented as “get ESSID”) then the scanner checks that the value of the section at Y - 2 bytes is between 5 and 60 (5 is the smallest an `iw_event` object can be and 60 is a configurable upper bound). If the value at address Y - 2 is within range then everything between addresses Y - 2 and Y - 2 + value(Y - 2) is carved out and considered a possible `iw_event` object.

Once a possible `iw_event` object is found, the algorithm uses the command field to look up the appropriate validation code. These functions all check that the `iwreq_data` object in the possible `iw_event` object is valid. For the example above: The command is `SIOCGIWESSID`, so the `iwreq_data` object should be a string containing the ESSID associated with the wireless access point. This string is extracted easily since it must be in the memory between addresses Y + 2 bytes (just past the command field) and Y - 2 + value(Y - 2) (the end of the `iw_event` object). Then, each character in the string is checked to be in the range of printable characters[2]. If those tests pass then the given `iw_event` object is considered valid.

If a possible `iw_event` object is found to be valid then we have a possible beginning of a stream. If the validated `iw_event` object begins at memory address X then the algorithm increments to address X + value(X) and checks for another `iw_event` object[3] [4]. This process repeats until the current `iw_event` object of interest is considered invalid. If enough `iw_event` objects in a row are found to be valid then

[2] We also allow for nulls and a few other values to accommodate string termination, etc.

[3] Note that if the `iw_event` object begins at address X then the first 16 bits at address X contain the length field for that `iw_event` object.

[4] In reality the algorithm jumps to address X + value(X) + 2 in the memory image and recursively calls the checking function. The "+2" is because that function will be first looking for valid `iw_event` commands.

the entire group is considered a valid stream. This stream is carved from the memory image and displayed to the user. Figure 2 contains pseudocode for the algorithm described in this section.

```
Let *p denote "the value at offset p in the
input file."

Let p->q where p is a pointer to a struct
denote "the value of field q in the struct
pointed to by p."


Variable Definition:
struct iw_event pointer S = null;

array V[] = validation functions for each
specific iw_event command type.


Code:
FOR EACH offset x IN input file:
    IF check(x) = True THEN:
        output(x);

FUNC check(offset x):
    LOCAL return_value = FALSE;
    IF *x is a possible iw_event command THEN:
        IF 5 <= *(x - 2 bytes) <= 60 THEN:
            S = x;
            return_value = V[S->command](S);
    RETURN return_value;
```

Figure 2. Pseudocode Algorithm.

3.3 `wext_stream_scan` plugin

The first implementation of our algorithm was a plugin for the Volatility framework [5] named `wext_stream_scan`. Implementing the algorithm as a plugin first was extremely helpful because it provided an easy to write proof of concept[5] to verify that the algorithm worked and also allowed for any minor implementation

[5] Volatility plugins are written in Python 2.x

changes to be tested. Additionally, because the Android operating system is built on the Linux kernel, we were able to test our plugin against a variety of memory captures from Linux computers that we had previously analyzed with Volatility.

The format of Volatility plugins made it easy to adapt our algorithm to fit the framework's required plugin interface. By default a plugin must export a `calculate` and a `render_text`[6] function. The `render_text` function is (as the name suggests) simply a means of displaying in a human-readable form any output that the program determines is valid. An investigator would then need to analyze this output for false-positives or potentially important evidence.

In the `calculate` procedure, we used the built in Volatility Scanner object to quickly walk through the memory image looking for possible `iw_event` commands at each legitimate memory address. We did this by overloading the Scanner's `check` function (which takes an offset in the memory image as a parameter) to perform our algorithm. Specifically, we performed the validation on the command and length fields to find a possible `iw_event` object at the given offset. If those checks passed, we then used the command field to index into a dictionary of functions to find the appropriate validation procedure for the data field of the `iw_event` object. In the event that an `iw_event` object is validated at a given offset we then continue to check the offsets immediately after the end of the previously verified `iw_event` object. If the

[6] Volatility can accept rendering functions for other data types besides "text", but we only needed to display textual answers.

implementation finds and verifies a steam of `iw_event` objects then it returns a boolean True to the Scanner and that offset is marked for rendering.

To simplify the manual inspection of `wext_stream_scan`'s output that a forensic investigator would have to perform, we designed a second plugin. The second plugin, named `wifi_mac_list`, will perform the same validation to find `iw_event` streams, but will only render the wireless access point's MAC - hence the name. By contrast the output of `wext_stream_scan` will list all information that can be gleaned from the carved `iw_event` streams.

3.4 `iwe_pull` and `iwe_carve`

After verifying that our algorithm worked as expected[7], we then developed two more tools - `iwe_pull` and `iwe_carve`.

`iwe_pull`, a small (~140 lines) C program, was designed to aid an investigator with devices (either Linux machines with wireless NICs or Android devices) that would later be analyzed for wireless access point data. Because of the limitation discussed in chapter 4 section 1, an investigator may want to improve the quality of a memory capture before analysis. The tool guarantees that, with as much certainty as possible, at least one copy of the most current Wi-Fi access point information is contained in memory. Assumably this tool would be run immediately before taking a snapshot of the device's physical memory contents. Two scenarios were considered when designing this tool and they are described below.

[7] … and running into the limitation described in chapter 4 section 1

First, an investigator is faced with a time sensitive investigation of a device. In this type of environment, he or she would want to save as much information as possible to analyze later before losing physical access to the device in question. It is common practice for an investigator to have on hand at all times a number of investigative tools and we envision that `iwe_pull` will be one of them. Rather than spending valuable time to mark which wireless access points the device was in contact with, an investigator could simply run `iwe_pull` (and possibly other similar tools) to very rapidly fill the device's memory with as much valuable evidence as possible. This would be followed immediately by capturing a copy of the device's physical memory. Note that these tools (including `iwe_pull`) are not producing any evidence that was not already contained in the operating system; they are simply making investigation of the device faster by allowing an investigator to save the information in a memory image and recover it at a later, less stressful, time.

Second, an investigator may be unsure whether any complete and up-to-date `iw_event` streams are contained in memory. Indeed it may be the case that the device has not generated an `iw_event` stream (as a result of the function calls described in section 1 of this chapter) in a considerable amount of time. Additionally, rather than having to locate and document a device's wireless access point information for the specific device being investigated (which could range from a Linux laptop to an Android smartphone to an Amazon Kindle Fire) an investigator could simply run `iwe_pull` to place a copy of the most current Wi-Fi access point information in the device's memory. Again the investigator could then use one of our tools to extract the wireless access point data from the memory image at a later time.

Note that using `iwe_pull` to improve the quality of a memory capture is in no way creating new evidence. It is simply making the information that is already contained in the kernel easier to find during later analysis. In fact, it is nearly impossible to find this information if it is not organized into `iw_event` streams because this data is normally held in driver specific data structures. Should the investigator be in a situation where running `iwe_pull` is not possible (e.g. a previously acquired memory capture) then our analysis tools will still be able to find the evidence contained in the memory image. Additionally running `iwe_pull` will only make the most current wireless access point information easier to find in memory. Our forensic tools will always be able to find any legacy information (i.e. left over from past calls to the functions described in section 1 of this chapter) regardless of if `iwe_pull` was run on the device.

`iwe_carve` is an implementation of our algorithm as a standalone C program. This program is designed to give the same functionality as the Volatility plugin, but without needing the memory image to work with the other features of Volatility (which, as described in chapter 2 section 3, would require the implementation of an "address space" if one is not already available for the kernel version within the memory image). `iwe_carve` also gives an investigator the benefit of being able to work on any file, not just memory images. Conceivably an investigator may find `iw_event` streams in a swap or hibernation file[8].

`iwe_carve`'s implementation is similar to our `wext_stream_scan` plugin, but differs in two main ways: it, obviously, cannot use any of the built-in functionality of the Volatility framework and it has the advantage of being able to include the C header files that define many of the constants being used in the kernel's code. Again we use the fact that the Android operating system is built on a nearly identical copy of the regular Linux kernel.

`iwe_carve` begins by rapidly reading through the input file scanning for possible `iw_event` commands. Then when a possible `iw_event` object is found, switch cases are used to appropriately examine the data field of the possible `iw_event` object. Like in our `wext_stream_scan` plugin, if the first possible `iw_event` object is found to be valid then the program continues checking at the address immediately after the end of the newly validated `iw_event` object in an attempt to find a stream. If a stream is found then the program will output its contents for an investigator to review.

[8] Though we did not test this for this thesis.

Chapter 4 - Limitations

4.1 `iw_event` streams in memory

One main limitation of our approach is the availability of `iw_event` streams in memory on an arbitrary device. As noted in chapter 3 section 1, `iw_event` streams are produced by an `ioctl` function call with a command argument of `SIOCGIWSCAN`. This command argument corresponds to the “get scanning results” handler code. Unfortunately for our research, this is not the only way for a process to query wireless network information.

As we mentioned in chapter 3 section 1, `wireless.h` [9] from the Linux kernel source defines the full list of possible `ioctl` function commands that Wireless Extensions can handle. The benefit of the `SIOCGIWSCAN` command is that all the known wireless network information is returned, but most of these can be used as a command argument to the `ioctl` function to query just one piece of the wireless network device’s information. The problem that these other commands pose to our research is that their returns are not in a predefined, specific format.

These other commands only query the NIC driver for one specific piece of information, and so the information returned from that query is simply the raw data that the process asked for. For example, a process can call the `ioctl` function against an internet socket and with a command of `SIOCGIWAP` which corresponds to “get access point MAC addresses.” The buffer supplied to the `ioctl` function would then be filled with just the hexadecimal values of the access point’s MAC address. Obviously, this type of data (without any significant formatting or characteristics) cannot be found in or carved from a memory image without prior knowledge of its value.

The bigger problem that these alternate `ioctl` commands pose is that we have found that they are more commonly used than `SIOCGIWAP`. We performed a manual code review of two common programs where one would expect wireless network information to be queried: the Linux NetworkManager [12] component and the Android operating system's WPA_supplicant [13]. We found that both of these programs used individual `ioctl` function calls to retrieve each piece of the wireless network information that they display.

This limitation was overcome by implementing `iwe_pull` to allow an investigator to place a copy of the most current `iw_event` stream in memory. Additionally, we found that there is a high level of determinability when scanning for `iw_event` streams (because of the multiple dependencies between the fields in the structures). Thus even if an investigator does not get a chance to run `iwe_pull` on a target machine, he or she can be assured that if a stream is in memory then our algorithm will be able to find it.

4.2 Volatility framework

One limitation that we faced when testing our design was that the Volatility framework's support for memory dumps taken from ARM processors is still experimental. This made it very difficult to develop "addresses spaces" for the Android smartphones that we were testing with. As a result we were only able to test our Volatility plugin on memory images taken from a laptop computer running Ubuntu

Linux and a Linux desktop with a WI-FI USB adapter.

Analysis of memory captures taken from our Android smartphones was done using `iwe_carve`. We also performed analysis of the Linux computer's memory images with `iwe_carve` to ensure that both implementations of our algorithm produced identical results.

4.3 Methodology and Testing

Another challenge we faced both while developing our algorithm and testing it was the limited options we had for kernel analysis. As is obvious from chapter 3 section 1, the information that we are interested in (specifically `iw_event` streams) is created and handled in the operating system kernel. Therefore, when beginning our analysis we had to inspect how the Android operating system kernel handles requests by processes for wireless access point data. This would generally be very easy thanks to virtualized kernel debugging (the process of debugging a running kernel inside a virtual machine from a gdb instance on the host machine). However, we ran into multiple problems with this approach.

The Android emulator [14], which is basically a virtual machine running a specially modified Android kernel, does not support Wi-Fi access. In fact, the emulator does not even emulate a NIC card for the virtual device, and instead it treats the host's Internet connection (regardless of if that connection is wired or wireless on the host) as its own internet connection. Thus, any analysis or testing that we performed on the Android emulator was useless as the kernel did not support wireless networking.

The kernel debugging savvy reader may think (as we did) that the obvious next step is to debug a physical, live phone via a USB cable. This would have been a perfect solution, but would have required recompiling the phone's kernel and using the new kernel on the debugged phone. This would turn out to be impossible since the Droid2 smart phones that we were using have signed boot loaders and boot partitions. This meant that any changes to boot loader (such as a using a different kernel) would cause the signature to change and so the phone would refuse to boot.

In the end, we again relied on the fact that the Android operating system is built on a regular Linux kernel. We used virtual debugging on a virtual machine running Ubuntu Linux then verified via manual code inspection that our findings would be the same for the Android operating system.

Chapter 5 – Results

5.1 Acquiring Viable Memory Captures

The first step in evaluating the functionality of our algorithm was to collect viable memory captures. These memory captures would serve as test evidence as if it had been collected in a digital forensics investigation. This was done, as mentioned in chapter 2 section 2, using `dmd` [4].

To use `dmd`, one must first compile it (it being a kernel module) using the target kernel source. For our analysis of an Android smartphone, we used the Droid2 given to us by GNOCIA which was running Android 2.2 with the 2.6.32.9 kernel. Luckily, previous researchers from the University of New Orleans were able to give us a copy of the kernel source (because Motorola has removed it from its open source repositories). We used a virtual machine running Ubuntu 11.04 Linux with a 2.6.38-8 kernel as our Linux evidence device and acquired that kernel's source easily.

Once `dmd` was compiled and working on our test devices, we could begin taking memory captures. In order to ensure that evidence (specifically `iw_event` streams) would be present in memory when we took our memory images, we also needed to compile `iw_pull` for both the Linux computer and Droid2 smartphone. This was considerably simpler that compiling `dmd` because Google provides the toolchains necessary to compile usermode programs for use with the Android operating system on ARM processors. Figure 3 shows the output from executing `iw_pull` on the Droid2 evidence smartphone. Now able to capture memory images that were guaranteed to contain `iw_event` streams, we then tested our analysis tools.

```
# adb push iwe_pull /sbin/
# adb shell
$ su
# chmod 777 /sbin/iwe_pull
# /sbin/iwe_pull
[+] Using Interface: eth0
[+] Opening Socket
[+] Performing IOCTL
[+] Success!
#
```

Figure 3. iwe_pull on Droid2.

5.2 Analysis of Memory Captures

First the `wext_stream_scan` plugin was used to find and analyze any `iw_event` streams in the Linux evidence computer's memory image. This produced a very large output with, as expected, multiple copies of the `iw_event` stream in memory (recall that the kernel will copy its buffer into the user-space buffer). Part of this output is shown in Figure 4. This output was verified by checking the information found in the memory dump against wireless network information found by other (control) devices[9] in the same area. All the information matched between the devices and the carved `iw_event` streams.

Because the output of `wext_stream_scan` was very detailed and an investigator may not care to know a lot of the information contained in the `iw_event` stream (for example the channel frequency is unlikely to be of much evidentiary value) we designed and then tested the `wifi_mac_list` plugin. The `wifi_mac_list`

[9] The control devices consisted of a Laptop and a Motorola Atrix smartphone.

plugin drastically simplified the output to make investigation much simpler for an investigator. This output is shown in Figure 5.

Next we analyzed the memory captures from the Linux computer using `iwe_carve`. This output, as expected, reported the same information as the previous analysis using the `wext_stream_scan` plugin. This proved that our algorithm worked correctly on both implementations. A portion of this output is shown in Figure 6.

Finally, the memory images from the Android smartphone were analyzed with `iwe_carve`. Again this output was verified by checking the information found in the memory dump against wireless network information found by the same control devices. Again all the information was successfully matched. Some output from this execution of `iwe_carve` is shown in Figure 7.

```
[4] Stream - Starting Offset:0x2D02E744
 -iwevent
   Offset:0x2D02E744   - 0x2D02E757
   Length:20
   Command:0x8B15
   af_famaly:  1
   af_data:00:3A:99:F6:65:31
 -iwevent
   Offset:0x2D02E758   - 0x2D02E763
   Length:12
   Command:0x8B05
   freq: Channel: 11.000000
 -iwevent
   Offset:0x2D02E764   - 0x2D02E76F
   Length:12
   Command:0x8B05
   freq:  2.462000 GHz
 -iwevent
   Offset:0x2D02E770   - 0x2D02E777
   Length:8
   Command:0x8C01
   Quality: 47
   Signal level: 193 dBm
 -iwevent
   Offset:0x2D02E778   - 0x2D02E77F
   Length:8
   Command:0x8B2B
 -iwevent
   Offset:0x2D02E780   - 0x2D02E78E
   Length:15
   Command:0x8B1B
   SSID:lifenet
```

Figure 4. wext_stream_scan against Linux computer memory capture.

```
iw_event streams...
   MAC:D0:C2:82:06:EC:52   SSID:0x00
   MAC:00:3A:9A:04:2F:87   SSID:0x00
   MAC:20:37:06:F6:63:63   SSID:wetr-guest
   MAC:02:7F:D0:09:00:92   SSID:HPC4A9FE
   MAC:06:01:70:16:FE:96   SSID:HPC4128D
   MAC:D0:C2:82:06:EC:50   SSID:WData
   MAC:00:3A:9A:04:2F:87   SSID:0x00
   MAC:00:3A:99:F6:56:D1   SSID:lifenet
   MAC:00:3A:9A:04:2F:81   SSID:lifenet
   MAC:10:9A:DD:81:E3:EF   SSID:Haedicke Law
   MAC:02:20:E0:EE:AE:D3   SSID:HPC4EA69
   MAC:00:12:17:04:3E:88   SSID:CKB
   MAC:00:3A:99:F6:56:D4   SSID:5WiFiWlan5
```

Figure 5. wifi_mac_list against Linux computer memory capture.

```
[+] Opening input file /home/knowbs/memdumps/DMDdump.dump
[+] Opened input file for processing.
[+] Beginning Scan.
 .
 .
 .
[4] Stream
[=]  iwevent
[-]   Length:20
[-]   Command:0x8B15
[-]   af_famaly:  1
[-]   af_data:00:3A:99:F6:65:31
[=] iwevent
[-]   Length:12
[-]   Command:0x8B05
[-]   freq: Channel: 11.000000
[=] iwevent
[-]   Length:12
[-]   Command:0x8B05
[-]   freq:  2.462000 GHz
[=] iwevent
[-]   Length:8
[-]   Command:0x8C01
[-]   Quality: 47
[-]   Signal level: 193 dBm
[=] iwevent
[-]   Length:8
[-]   Command:0x8B2B
[=] iwevent
[-]   Length:15
[-]   Command:0x8B1B
[-]   SSID:lifenet
```

Figure 6. iwe_pull against Linux memory capture.

```
[+] Opening input file /home/knowbs/memdumps/DMDdroid.dump
[+] Opened input file for processing.
[+] Beginning Scan.
 .
 .
 .
[8] Stream
[=]  iwevent
[-]   Length:20
[-]   Command:0x8B15
[-]   af_famaly:  1
[-]   af_data:00:21:55:61:B2:E0
[=] iwevent
[-]   Length:12
[-]   Command:0x8B05
[-]   freq: Channel: 6.000000
[=] iwevent
[-]   Length:12
[-]   Command:0x8B05
[-]   freq:  2.437000 GHz
[=] iwevent
[-]   Length:8
[-]   Command:0x8C01
[-]   Quality: 38
[-]   Signal level: 184 dBm
[=] iwevent
[-]   Length:8
[-]   Command:0x8B2B
[=] iwevent
[-]   Length:15
[-]   Command:0x8B1B
[-]   SSID:unosecure
```

Figure 7. iwe_pull against Droid2 memory capture.

Chapter 6 – Conclusion

6.1 Wrap-Up

The central question that we set out to answer was: Given only a context-free memory image from an Android device, can an investigator be able to identify which wireless network access points the device was near when the image was taken?

This thesis thoroughly answered that question. In chapter 3, we presented our findings regarding how wireless network information gets put into a device's memory in the first place. We then presented three distinct tools that investigators can now have at their disposal for finding and analyzing that information. In chapter 4, we discussed the limitations of this approach which were discovered during our research. Finally, chapter 5 proved that, despite any limitations, the solution presented in this thesis is as effective as could be performed. We anticipate that this work, in a small way, has made a difference in the amount of information one can expect to glean during a digital forensics investigation of an Android mobile device.

6.2 Future Work

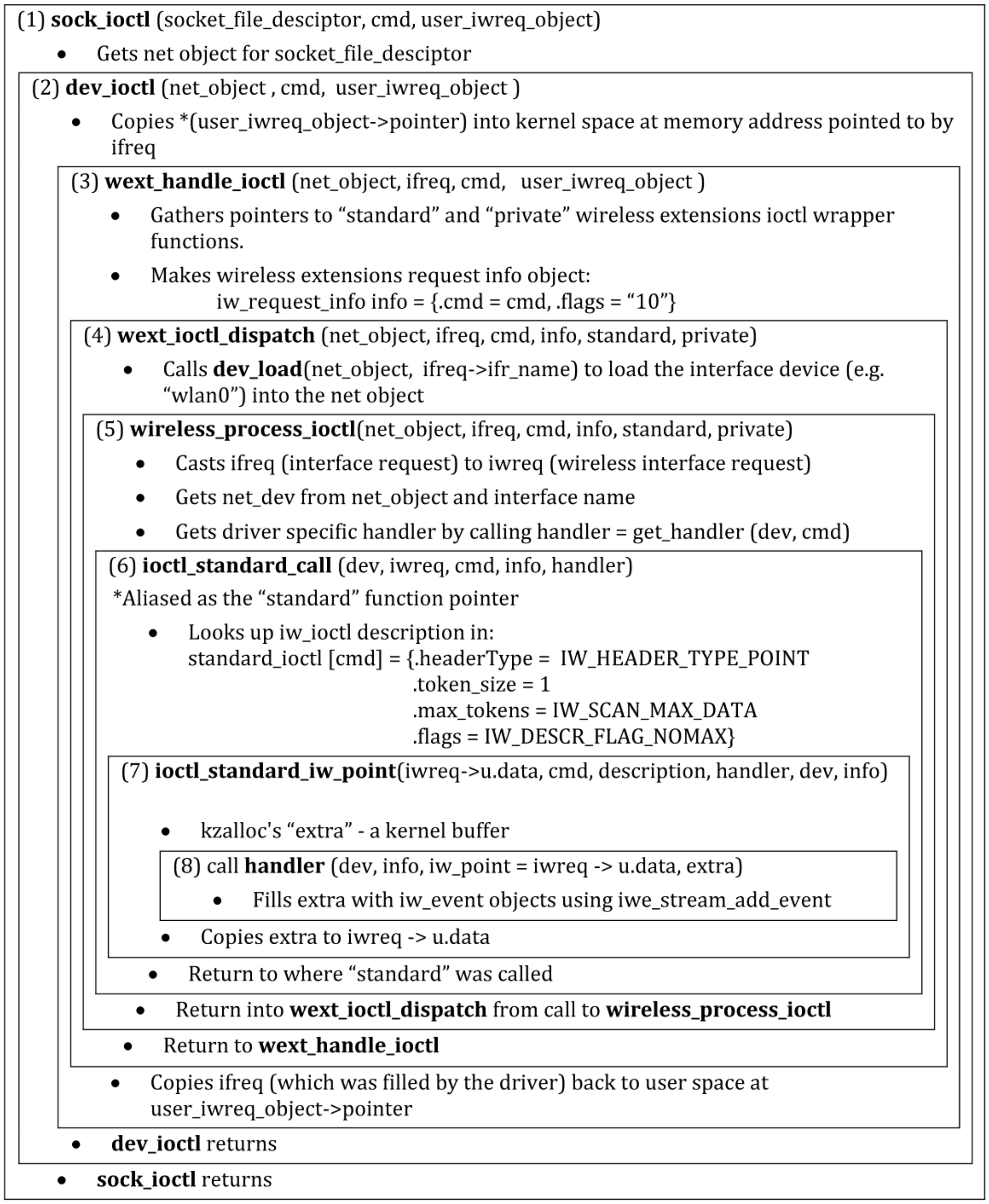
One main objective that is left as future work of this research is the analysis of non-volatile memory. We hope to thoroughly test our `iwe_carve` implementation for use with hibernation and swap files (as mentioned briefly in chapter 3 section 4). Since both of these operating system files contain copies of volatile memory, it is probable that any `iw_event` streams that are in memory at the time may be saved in those files. Additionally, (as mentioned in chapter 3 section 4) since `iwe_carve` does not require its input file to have any particular formatting, there is a high likelihood that it would be able to find and carve `iw_event` streams in those non-volatile files. The author feels that there is no reason to suspect the presence of `iw_event` streams in any other portion of a device's non-volatile storage.

The second objective being left for future work is to have the `wext_stream_scan` plugin and `wifi_mac_list` plugin added to the Volatility framework. This objective is currently underway and we hope that future releases of Linux compatible Volatility will include our plugins in the default plugin library.

Appendix A

Below is a depiction of the Wireless Extensions I.O. control handling code. Starting at (1) in the image, the figure walks through the important steps taken in the kernel to handle and return an answer from an ioctl function called on an internet socket handle with a command argument of SIOCGIWSCAN (corresponding to hexadecimal value of 0x8B19) and a pointer to a caller supplied buffer to fill with the wireless access point data.

The following are the arguments supplied to the original ioctl function call:
socket_file_desciptor = an internet socket file desctiptor
cmd = SIOCGIWSCAN
user_iwreq_object = {interface name = "wlan0", pointer = pointer to userspace buffer}



(1) **sock_ioctl** (socket_file_desciptor, cmd, user_iwreq_object)

- Gets net object for socket_file_desciptor

(2) **dev_ioctl** (net_object , cmd, user_iwreq_object)

- Copies *(user_iwreq_object->pointer) into kernel space at memory address pointed to by ifreq

(3) **wext_handle_ioctl** (net_object, ifreq, cmd,   user_iwreq_object)

- Gathers pointers to “standard” and “private” wireless extensions ioctl wrapper functions.
- Makes wireless extensions request info object:
  iw_request_info info = {.cmd = cmd, .flags = “10”}

(4) **wext_ioctl_dispatch** (net_object, ifreq, cmd, info, standard, private)

- Calls **dev_load**(net_object, ifreq->ifr_name) to load the interface device (e.g. “wlan0”) into the net object

(5) **wireless_process_ioctl**(net_object, ifreq, cmd, info, standard, private)

- Casts ifreq (interface request) to iwreq (wireless interface request)
- Gets net_dev from net_object and interface name
- Gets driver specific handler by calling handler = get_handler (dev, cmd)

(6) **ioctl_standard_call** (dev, iwreq, cmd, info, handler)
*Aliased as the “standard” function pointer

- Looks up iw_ioctl description in:
  standard_ioctl [cmd] = {.headerType = IW_HEADER_TYPE_POINT
  .token_size = 1
  .max_tokens = IW_SCAN_MAX_DATA
  .flags = IW_DESCR_FLAG_NOMAX}

(7) **ioctl_standard_iw_point**(iwreq->u.data, cmd, description, handler, dev, info)

- kzalloc's “extra” - a kernel buffer

(8) call **handler** (dev, info, iw_point = iwreq -> u.data, extra)

- Fills extra with iw_event objects using iwe_stream_add_event

- Copies extra to iwreq -> u.data
- Return to where “standard” was called
- Return into **wext_ioctl_dispatch** from call to **wireless_process_ioctl**
- Return to **wext_handle_ioctl**
- Copies ifreq (which was filled by the driver) back to user space at user_iwreq_object->pointer
- **dev_ioctl** returns
- **sock_ioctl** returns

Figure 8. Wireless Extensions I.O. Control Handler.

References

[1] L. Goasduff and C. Pettey, "Gartner," Gartner, Inc., 15 February 2012. [Online]. Available: http://www.gartner.com/it/page.jsp?id=1924314.

[2] R. Beverly, S. Garfinkel and G. Cardwell, "Forensic carving of network packets and associated," *Digital Investigation 8 (Supplement 1),* pp. S78 - S89, 2011.

[3] G. S. Cardwell, "Residual Network Data Structures in Android," Naval Postgraduate School, Monterey, California, 2011.

[4] J. Sylve, A. Case, L. Marziale and G. G. Richard, "Acquisition and analysis of volatile memory from android devices," *Digital Investigation 8,* p. 175–184, 2012.

[5] "Volatility," 2011. [Online]. Available: https://www.volatilesystems.com/default/volatility.

[6] E. Girault, "Volatilitux," 2010. [Online]. Available: http://www.segmentationfault.fr/projets/volatilitux-physical-memory-analysis-linux-systems/.

[7] "About Wireless-Extensions," 2011. [Online]. Available: http://wireless.kernel.org/en/developers/Documentation/Wireless-Extensions.

[8] J. Tourrilhes, "Wireless Extensions for Linux," HP, 1997. [Online]. Available: http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Linux.Wireless.Extensions.html.

[9] "LXR - wireless.h," [Online]. Available: http://lxr.linux.no/linux+v2.6.38.8/include/linux/wireless.h.

[10] "LXR - wext_core.c," [Online]. Available: http://lxr.linux.no/linux+v2.6.38.8/net/wireless/wext-core.c.

[11] "LXR - iw_handler.h," [Online]. Available: http://lxr.linux.no/linux+v2.6.38.8/include/net/iw_handler.h.

[12] "Linux Network Manager Code Repository," 2011. [Online]. Available: http://cgit.freedesktop.org/NetworkManager/NetworkManager/tree/src/wifi/wifi-utils-wext.c.

[13] "Android WPA Supplicant Code Repository," 2012. [Online]. Available: https://github.com/android/platform_external_wpa_supplicant/blob/master/driver_wext.c.

[14] "Android Emulator," Google, 2012. [Online]. Available: http://developer.android.com/guide/developing/tools/emulator.html.

APPROVAL SHEET

This is to certify that Brendan D. Saltaformaggio has successfully completed his Senior Honors Thesis, entitled:

Forensic Carving of Wireless Network Information from the Android Linux Kernel

[illegible] Director of Thesis
Golden Richard

[illegible] for the Department
Jaime Nino

Carl D. Malmgren for the University Honors Program
Carl D. Malmgren

April 30, 2012
Date