Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS USA 2016 — Proceedings of the 16th annual USA digital forensics research conference

# Rapid differential forensic imaging of mobile devices

M. Guido*, J. Buttner, J. Grover

*The MITRE Corporation, 7515 Colshire Drive, Mclean, VA, 22102, USA*

*Keywords:*
Mobile forensics
Rapid acquisition
Differential analysis
Android

A B S T R A C T

Commercial mobile forensic vendors continue to use and rely upon outdated physical acquisition techniques in their products. As new mobile devices are introduced and storage capacities trend upward, so will the time it takes to perform physical forensic acquisitions, especially when performed over limited bandwidth means such as Universal Serial Bus (USB). We introduce an automated differential forensic acquisition technique and algorithm that uses baseline datasets and hash comparisons to limit the amount of data sent from a mobile device to an acquisition endpoint. We were able to produce forensically validated bit-for-bit copies of device storage in significantly reduced amounts of time compared to commonly available techniques. For example, using our technique, we successfully achieved an average imaging rate of under 7 min per device for a corpus of actively used, real-world 16 GB Samsung Galaxy S3 smartphones. Current commercially available mobile forensic kits would typically take between one to 3 h to yield the same result. Details of our differential forensic imaging technique, algorithm, testing procedures, and results are documented herein.

## Introduction

Performing a physical forensic acquisition for Android™ devices usually requires the device to be booted into one of the following environments:

- Custom bootloader
- Custom recovery mode
- Normal mode with root access.

In any of these modes, physical acquisition techniques need to execute code on a target mobile device, which creates in-memory, bit-for-bit copies of the device that are sent to a receiving service. In most commercial toolkits, data is typically sent over a Universal Serial Bus (USB) interface to a connected hardware device or server (e.g., laptop or desktop). USB 2.0 has a maximum transmission rate of 480 Mb/s but rarely achieves speeds of more than 320 Mb/s (Spector, 2014).

Full physical forensic acquisitions can be time-intensive, sometimes taking hours to complete. Acquisition times are largely dependent on device processor speeds, cable types used, and the amount of data transferred. At the time of writing, the largest Android devices available on the market were 128 GB (Florin, 2015). As devices continue to grow in size, the times to physically acquire them will likely increase.

In some circumstances, such as time-sensitive operations at crime scenes or border crossings, having a rapid physical acquisition capability for mobile devices could be critical in resolving a situation. Currently, in these situations, a forensic investigator may instead opt to perform a logical device acquisition to save time. Not having a physical image available during an examination may open questions about missing data, as logical acquisitions do not capture disarranged or deleted files, and in some cases may not preserve file timestamps.

Our research focuses on reducing the amount of data that needs to be transferred during the physical acquisition

* Corresponding author. Tel.: +1 703 983 5130; fax: +1 703 983 1002.
*E-mail address:* mguido@mitre.org (M. Guido).

http://dx.doi.org/10.1016/j.diin.2016.04.012

process, thus decreasing the overall acquisition time. The final product can be the same as that of a traditional acquisition tool: a complete physical forensic image. We utilized prior research to develop a prototype software agent named *hawkeye*, which uses differential analysis and runs within an Android custom bootloader or recovery mode to acquire a physical forensic image.

The remainder of this paper is structured as follows: Section (Related work) covers related work; Section (Corpus of phone images) discusses the phone image corpus used by *hawkeye*; Section (Hawkeye) contains *hawkeye* implementation details, including the algorithm used; Section (Experimentation) contains experiment procedures and results; Section (Discussion) includes a discussion area; and Section (Conclusion) concludes with a summary and some proposed future work.

## Related work

The Hawkeye project is an extension of the Periodic Mobile Forensics (PMF) system (Guido et al., 2013); however, it targets a different use case (e.g., a crime scene) and is specifically designed to improve device acquisition speeds. The *hawkeye* agent is designed to operate as a client within the overall PMF system architecture, which is referenced heavily within this work. Both systems use differential forensic analysis, as formally defined by Garfinkel, Nelson, and Young (Garfinkel et al., 2012). The original PMF agent has operated on a variety of mobile devices running Android 2.2+, and there is no reason foreseen that the *hawkeye* agent, using the lessons learned building PMF compatibility, should not be considered equally compatible with modern mobile devices.

Laurenson et al. applied and automated the work done by Garfinkel, Nelson, and Young to collect and distribute application software artifacts in a reference set that they termed *application profiles* (Laurenson et al., 2015). While their purpose and implementation differ, there are many similarities found in building Hawkeye's baseline hash list and corresponding data storage in PMF. We will describe several mechanisms built into PMF (Guido et al., 2013) for generating these hashes.

Gurjar et al. (2015) compared the runtime efficiency of common hashing algorithms (MD5, SHA-family) and their implementations on Windows® and Linux®. In their work, they found that MD5 performed best on both Windows and Linux. Hawkeye uses MD5 as its hash algorithm because of its speed. The risk of constructed collisions using MD5 is not relevant for the scope of our work.

The method of using hash maps to discriminate known-good files is well known in the forensics community, although it is not typically implemented in physical acquisition tools. This fact holds true in the commercial mobile forensic acquisition tools that were tested as part of our work. One of the notable contributions of the Hawkeye work is the implementation of differential analysis, enabling Hawkeye to only hash a fraction of a device's storage, leading to significant time savings.

Watkins et al. (2009) previously developed Teleporter. The *hawkeye* agent incorporates a hashing comparison technique similar to that of Teleporter; however, both the purpose and overall system design differ significantly. Teleporter's purpose is to enable transport of a minimal amount of data from hard drives when faced with limited bandwidth environments, sometimes over large distances, and it was designed to identify both known files and previously recorded blocks of data. Hawkeye's purpose is to acquire a full disk image from a target mobile device and does not have requirements to interpret any filesystems or storage content. Some partitions of an Android device are structured in proprietary or undocumented formats; Hawkeye acquires them all and makes no attempt to understand them; that is left for the PMF system (Guido et al., 2013). Similarly, Grier and Richard (2015) use sifting collectors to identify and acquire only the regions of a disk that have forensic value. Their research limits the amount of device data imaged and does not result in complete bit for bit copies produced by Hawkeye or other mobile forensic tools. Grier and Richard's approach is similar to Teleporter in that they interpret filesystems to identify files and they note that their methods are not suitable for unknown filesystems.

Garfinkel et al. (2010) performed forensic analysis at the block and sector level. They developed algorithms to identify fragments of file formats on a storage device and showed that contents of a storage device can be determined with high accuracy using statistical sampling. Hawkeye uses hash representations of much larger blocks of storage compared to (Garfinkel et al., 2010), primarily to tradeoff the number of required hash comparisons performed to the amount of data transmitted over the wire. Statistical sampling to determine mobile storage contents could be complementary during forensic analysis of the mobile device images that Hawkeye collects.

Mobile forensic acquisitions are often considered “live acquisitions” because they rely on a target device's running kernel. Many commercially available mobile phone kits use live acquisition techniques to take one-time logical or physical images of a target device (Lessard and Kessler, 2010). Vidas, Zhang, and Christin (2011) developed a more generalized acquisition method that requires no prior knowledge of phone content. Son et al. (2013) studied the recovery mode method formally introduced by Vidas, Zhang, and Christin and found that a device's userdata partition maintained its integrity during the “recovery mode” acquisition. Recovery mode is a preferred environment for *hawkeye* to execute in because it enables the tool to operate on many different Android devices, provides access to Android API functions, and temporarily disables all wireless functionality of the mobile device.

Yang et al. (2015) demonstrated a new method of acquiring a device through the Android update protocols of some devices' bootloaders. They tested a variety of 32 GB Android devices[1] and found that their method was significantly faster than the Cellebrite® UFED 4PC. They stated that their method took 30 min and UFED 4PC took 120 min on average. Their results inspired the optimization and

[1] Devices used by Yang et al.: LG® G3™ (F400S, D851), Optimus G™ (F180S, E975), R3 (IM-A850S), Iron2 (IM-A910S), and Nexus™ 4/5 (E960, D821).

application of PMF's automated differential forensic techniques into Hawkeye. Note that the devices used by Yang et al. were newer and more powerful than the phones used by the Hawkeye research team. We chose to use Samsung Galaxy S3s for availability reasons: we already had access to a corpus of Galaxy S3 device images populated with real-world data.

## Corpus of phone images

The PMF project originally sought to improve the state of enterprise mobile device monitoring. It was designed to discover malicious users, masquerading users, and malicious applications in an enterprise (Guido et al., 2013).

In the fall of 2013, in a collaborative effort with Purdue University's Department of Computer and Information Technology (CIT), PMF's capabilities were deployed in a simulated enterprise environment (Guido et al., in press). The environment consisted of 34 mobile devices, each operated by a unique human subject volunteer on the Purdue campus over a period of three months. Each volunteer was provided with an acceptable use policy, some deceptive instructions designed to steer normal behavior, and a Samsung Galaxy S3 mobile device with the original PMF software agent installed. The agent, named *tractorbeam*, was configured to communicate wirelessly with a cloud instance every 24 h, transmitting changed device storage as encrypted binary data. The server de-duplicated and stored the data until it was collected in a secure enclave for analysis purposes.

During the Purdue experiment, we were able to build a corpus of forensic images of the mobile devices used by the volunteers. Over the three-month period, we regularly stored the transferred forensic image data as the volunteers continued to use their provided devices. The final corpus consisted of more than 1100 images from the 34 devices. The volunteers were instructed to use the devices normally. We assume that they followed instructions and that their data is largely uninhibited. Therefore, we feel this corpus represents a rich dataset of normal device usage and behavior. We were able to provide Hawkeye with this “real-world” data to influence design decisions and verify results.

The corpus was stored in an efficient, de-duplicated format within PMF's database, delivering significant storage space savings. Utilities were developed for PMF that can interpret this format and convert images to their raw forms. Among those utilities was PMF's automated write-back capability, which was used in the Hawkeye experimentation (Section Experimentation) to re-populate some of the devices to physical states recorded during the Purdue experiment.[2]

## Hawkeye

The *hawkeye* agent is a multi-threaded Native C program compiled for ARM that is temporarily installed into volatile space on a device. The agent uses lists of pre-computed baseline hashes and partitions in its automated differential forensic imaging technique. The agent's goal is to identify and send only necessary storage device blocks to the backend PMF architecture as quickly as possible. The “bottleneck” in this process is the transmission of blocks over the USB link. Alternative ordering of procedures in Hawkeye's algorithm involving more client and server communications was explored, but it did not reduce and sometimes even increased the amount of data having to traverse the link.

### *Agent work-flow*

Hawkeye is meant for a one-time physical acquisition. When at a crime scene (or other applicable environment), the following approach should be used:

1. Identify the target mobile device model and verify Hawkeye support for it.
2. Flash a Team Win Recovery Project (TWRP) custom recovery image[3] onto the device and boot the custom recovery kernel.
3. Connect the target mobile device to the PMF architecture (e.g., laptop) via USB cable.
4. Execute Hawkeye from the laptop, which temporarily installs *hawkeye* to the device's volatile memory, sets up communications between the client and PMF queuing service, and starts the *hawkeye* agent.

The setup process in step 4 above provides the *hawkeye* agent with two key pieces of information: a baseline hash list (discussed in Section Adding device support / baseline imaging) sent from the laptop, and partition data interpreted from reading and processing the device's GUID Partition Table (GPT). The partition data is provided in a comma-separated format and includes:

- Path
- Size in bytes

By default, the partition list contains a single entry with the path /dev/block/mmcblk0. This causes the tool to perform a full device acquisition. The list can be altered during the setup process to instead include individual partitions (e.g., /dev/block/mmcblk0p3, /dev/block/mmcblk0p20) if the operator only wants to acquire a subset of the device storage. The on-device agent parses the partition list information and calculates individual block offsets, which are added to a stack for processing. Section (Per thread algorithm) covers the algorithm used to process the blocks.

[2] The corpus of images collected during the Purdue experiment were acquired while the devices were in active use. The images written back to the devices were potentially not in a stable state. For our purposes, the state was irrelevant because it was only used to populate the devices with real-world data.

[3] We compiled and installed our own custom version of a TWRP recovery image since we found that TWRP modifies several partitions of a device by default (i.e., it is not forensically sound). We removed TWRP's persistence mechanisms and added “read-only” and “no-load” mount options. The “no-load” option prevents journal loading on ext4 partitions. TWRP can be found here: https://twrp.me/. Our forensically sound changes can be provided upon request.

### *Adding device support /baseline imaging*

Adding device support to Hawkeye is a relatively simple task since it only requires a previously taken image of a baseline device. A baseline device should be the same or similar model to that of the target device. A baseline image can be used to build a baseline set of hashes (further referred to as a "gold hash list"). These are hashes that a forensic investigator would expect to find on a target device. This list can typically include hashes representing portions of stock operating system (OS) files, application data, and other common "read-only" storage areas of a device (e.g., the baseband, bootloader, boot image, etc.). The gold hash list is pushed to a target device during the Hawkeye setup process and is loaded into an in-memory hash table, which is referenced heavily during execution.

For the Hawkeye experiment conducted in Section (Hawkeye benchmarks), we built our gold hash list from the baseline images of the Purdue experiment (Section Corpus of phone images). We built the list by calculating all the unique MD5 hashes from each image using blocksizes of 64 KiB and 1 MiB.[4] This resulted in roughly 66,000 distinct hashes.

When Hawkeye is used in an operational setting, it is recommended that the hash list be prebuilt to include one or more Android versions per particular mobile device hardware. Having a wider variety of baseline devices and OS versions represented in the gold hash list will improve the overall imaging times. For example, the Galaxy S3 SGH-I747 model used in our Purdue experiment can run Android 4.04, 4.1.1, 4.1.2, 4.3, and 4.4, as well as custom third-party OSs such as Cyanogenmod.[5] Using baseline images from each of these versions to form a gold hash list allows *hawkeye* to skip sending data from certain storage areas of any Galaxy S3 SGH-I747 target device since those storage areas are already known.

To create a gold hash list, forensic images of the mobile devices' stock images need to be stored in the PMF database. These images can be acquired directly through PMF/Hawkeye or imported if another tool (e.g., Cellebrite UFED Touch, UFED 4PC) was used and that tool produced a raw (i.e., *dd*) image file. A list of MD5 hashes are then pulled from the PMF database and used by Hawkeye.

It is believed that a gold hash list could be produced and included by commercial mobile forensic kit vendors. Vendors have access to a wide variety of devices and already use device-specific acquisition procedures. We theorize that inclusion of a gold hash list and *hawkeye* algorithm would require relatively little additional effort on their parts and would significantly decrease/reduce their products' physical acquisition times.

### *Per thread algorithm*

The *hawkeye* agent uses partition information from a device's GUID Partition Table (GPT) table to populate a stack of 1 MiB block offsets. The block offsets are popped from the stack[6] and processed by a thread pool. Hawkeye is configured by default to use a thread pool with 10 threads.[7] Each thread in the pool performs the algorithm as seen in Fig. 1.

The Hawkeye algorithm initially calculates the MD5 of individual blocks with a small blocksize[8] and then uses the *search* and *insert* functions to interface with an in-memory hash table. The hash table contains MD5s from a gold hash list (Section Adding device support / baseline imaging). Searches are performed to check for the existence of MD5s in the hash table. When a hash is not found, it is immediately inserted into the table and its corresponding full 1 MiB data block is sent.

Using a small initial blocksize to hash a block saves processing time, as a full block hash is not always needed to determine if the block has been seen before. This results in a significant performance improvement worth noting. The following cases present themselves:

- If the hash of the first portion of a block is new, it is guaranteed that the full block has never been seen before.
- If the hash of the first portion of a block is not new (i.e., it has been seen before and results in a hash match), one cannot assume that the full block is already known.

In the latter case, the full block hash would be needed for the determination. When calculating the full block hash, Hawkeye makes use of the fact that it has already operated on part of the block. Since it previously calculated the small block hash, it continues hashing the remaining portion of the full 1 MiB block without having to rehash the first 64 KiB. If the full block hash matches an entry in the table, the data block is not sent (only its reference data is transmitted).

A by-product of the Hawkeye algorithm is the allowance for a limited amount of duplicate block data to be sent. We do not consider it a design flaw since the number of

```
procedure SCAN(block_offset)
  blocksize ← 1MiB
  buf ← read(block offset, blocksize)
  if customcmp(buf) = all zeros then return SEND HASH ONLY(md5_zeros)
  else
    md5_64KiB ← md5sum(buf, blocksize ← 64KiB)
    if SEARCH(md5_64KiB) = false then
      INSERT(md5_64KiB)
      SEND DATA(buf)
    else
      md5_1MiB ← md5sum(buf, blocksize ← 1MiB)
      if SEARCH(md5_1MiB) = true then return SEND HASH ONLY(md5_1MiB)
      else
        INSERT(md5_1MiB)
        SEND DATA(buf)
```

**Fig. 1.** *hawkeye* per thread scanning algorithm.

[4] The blocksize values of 64 KiB and 1 MiB are not arbitrary; an explanation for their use is provided in Section (Blocksize comparison).

[5] More information about Cyanogenmod is available from: http://www.cyanogenmod.org/.

[6] Hawkeye uses a stack instead of a queue, as the order of processing block offsets does not matter.

[7] Limited testing on a single Galaxy S3 device confirmed that a range of 10–15 threads was optimal in achieving the fastest runtimes.

[8] Smaller blocksizes of 4 KiB and 64 KiB were compared and tested to see which would yield a faster runtime. We chose to use 64 KiB for reasons explained in Section (Blocksize comparison).

occurrences were found to be negligible; however, it is noted herein. A duplicate block may be sent because of our choice to use a small block hash to check if a large block has been observed previously. Upon first occurrence of a small block hash not previously observed by Hawkeye, the large block is transmitted but no large block hash is calculated. Upon encountering a duplicate block, the large hash will be calculated, but Hawkeye will also unnecessarily transmit the same block again. This scenario is further broken down and happens when the following events occur:

1. A 64 KiB block hash is identified as new (not matching any hash previously in the hash list), is inserted into the hash lookup table, and has its corresponding 1 MiB data block transmitted to the server. The 64 KiB hash is added to the hash lookup table, but to save processing time, no 1 MiB hash is produced.
2. A duplicate 1 MiB block is encountered later. Its 64 KiB hash will match an existing hash in the hash lookup table, which will cause Hawkeye to calculate the 1 MiB hash. Its 1 MiB hash will not match any hash in the hash lookup table, prompting the hash to be identified as new. The 1 MiB hash will be inserted into the hash lookup table, and its corresponding 1 MiB block will be transmitted for the second time.

By not performing a full 1 MiB block hash of every block, we improved our overall performance even when some duplicate blocks were present. Inserting a calculation of the 1 MiB hash upon first observance of a block added over 30 s on average to Hawkeye's execution time.

Hawkeye does not use compression, nor does it transform the data in any way. Contents are transmitted in a raw block form. While investigating using Zlib for compression, the added time to compress and send the data negatively increased Hawkeye's overall execution time. As such, we decided to exclude it. We leave investigating alternative compression utilities for future work.

### Server architecture

As mentioned in previous sections, Hawkeye relies on a PMF server to operate. The server is not a main focus of this paper; however, its components are briefly covered here for the sake of completeness.

The PMF server architecture is composed of a PostgreSQL server, MongoDB server, and RabbitMQ service. Raw blocks of data are stored single-instanced in PostgreSQL. Full physical images of devices' blockdevices can be rebuilt[9] from PostgreSQL. MongoDB stores analytical results of forensic processes run against rebuilt images from PostgreSQL. RabbitMQ provides message queuing, which continuously listens for data from a device. For the Hawkeye experimentation, RabbitMQ was tuned to use 2 MiB buffers, which improved overall system performance. The PMF system (Fig. 2) has been designed to allow for the maximum number of USB devices running on the same host at the same time. The maximum number varies per machine, depending on the number of buses and the amount of bandwidth utilized by each device. To date, we have only tested four devices simultaneously and left more expansive testing for future work. In our limited experience, running multiple devices simultaneously has not noticeably affected acquisition times.

## Experimentation

Several experiments were performed to determine and improve the agent's runtime. The first experiment compared the use of several blocksizes. The second experiment details queries performed against the Purdue corpus, findings from which were used to tune Hawkeye performance. Overall Hawkeye benchmark testing was performed and the results are compared to existing products and techniques. Finally, specific features used by Hawkeye were measured for their contributions to the overall speed improvement.

### Blocksize comparison

The blocksizes used in the *hawkeye* algorithm (Section Per thread algorithm) are not arbitrary values. The algorithm makes use of two different blocksizes: a small blocksize and a full blocksize. Tests were performed to determine which set of blocksizes yielded the fastest overall runtimes.

In determining an optimal small blocksize, we produced lists of 64 KiB hashes and 4 KiB hashes from the Purdue dataset (Section Corpus of phone images). Each list was provided and used by the *hawkeye* agent. We found that 64 KiB blocksizes slightly increased the total amount of time spent hashing a device but also reduced the amount of data needed to be sent to the PMF server. In most cases, we found the 64 KiB size to be more efficient due to the reduced network impact.

Tests to determine an optimal full blocksize were also performed. We compared blocksizes of 128 KiB, 256 KiB, 512 KiB, 1 MiB, 2 MiB, and 4 MiB. The tests were run using an empty gold hash list on a heavily populated 16 GB Samsung Galaxy S3 and a sparsely populated 8 GB Nexus 4. Results were largely inconsistent and appeared to be data, device, and server platform dependent. For example, runtimes were all nearly identical on the Nexus 4 regardless of the blocksize used. On the Samsung Galaxy S3, there were some indications that a smaller blocksize (128 KiB – 512 KiB) may potentially result in slightly faster runtimes, but we were unable to identify a single optimal value. We found that when the same device sends to different server platforms,[10] the optimal blocksize value also changes.

While we recognize the need for further testing and research in determining the best blocksize, we chose to use 1 MiB for several reasons:

[9] PMF post-acquisition rebuilds are done on demand and typically take less than 5 min to produce a complete 16 GB image file from PostgreSQL on a bare metal Ubuntu® machine (see specifications described in Section Hawkeye benchmarks). Optimizations to the rebuild process were left for future work. The rebuild time was not included in any reported Hawkeye results because PMF can output this format as needed as part of a post-acquisition process.

[10] We configured two Ubuntu 14.04 servers to listen for hawkeye data, each with 2 GB RAM. One was a virtual machine; one was bare metal.

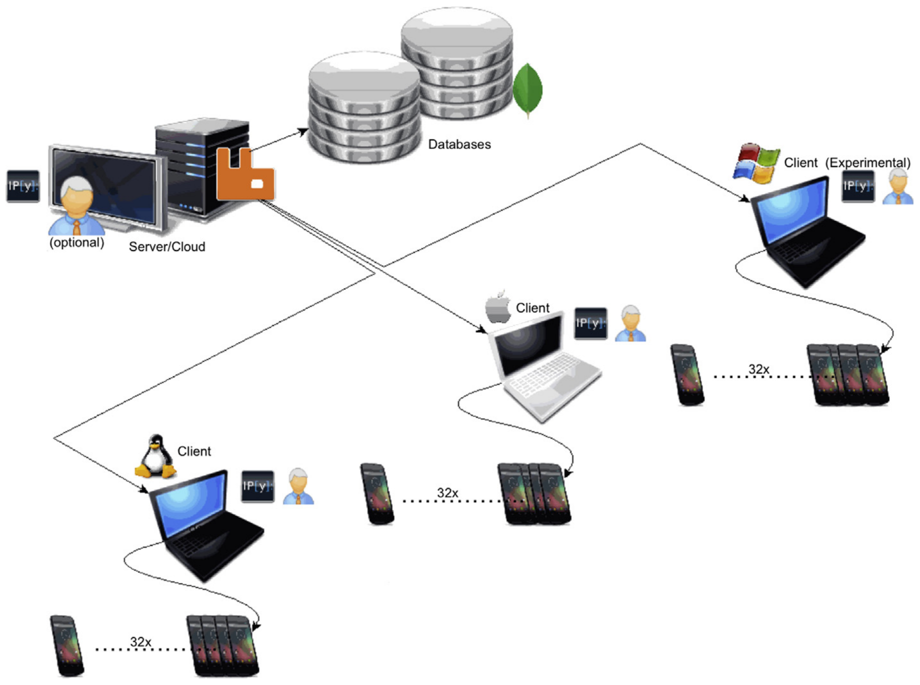

**Fig. 2.** PMF architecture.

- Based on our limited test results, we conjecture that 1 MiB represents a “sweet spot” between the runtimes of sparsely and heavily populated devices.
- PMF's legacy client uses 1 MiB blocksizes, allowing us backwards-compatibility. 1 MiB blocksizes previously proved to be optimal for wireless transmission.
- Some current commercial mobile forensic tools use 1 MiB blocksizes during acquisitions. Keeping a consistent blocksize may allow for a smoother transition if an approach like Hawkeye were to be implemented.

## *Corpus examination*

An analysis of the data in the Purdue corpus influenced design decisions in the *hawkeye* algorithm. In order to tune it for maximum performance, queries were performed on the dataset to determine the:

- Prevalence of “all-zero” blocks
- Frequency of blocks with a 4 KiB/64 KiB gold hash list match that also needed to be transmitted
- Frequency of duplicate blocks that have not been previously observed (i.e., the duplicate block sending scenario discussed in Section Per thread algorithm)

Analysis of storage across all devices in the Purdue corpus revealed a large amount of blocks (blocksize = 1 MiB) containing all zeros (i.e., NULL characters). We found that 620,277 of 1,406,104 (44%) blocks were all-zeros.[11] When examining the initial baseline images alone, we found that the all-zeros percentage was higher at 71% (543,508 of 765,866). The decrease in “all-zero” blocks over time is predictable, since the volunteers started with near-stock (baseline) devices and gradually added their unique content over the course of the experiment. Because there were still a significant number of all-zero blocks on many devices at the end of the experiment, we decided to improve the *hawkeye* algorithm by leveraging a zero block comparison function.

The comparison function iterates over the entire block until it finds a non-zero 4-byte word or determines the block is all zeros. The function compares 16 bytes at a time and implements loop unrolling to increase its speed.

We found that the zero block comparison function evaluates a block of all-zeros significantly faster than any hashing technique. On average, it classifies a block as all-zeros 252 times faster than an MD5 hash calculation. This finding led us to conclude that even if there is no support in Hawkeye for an unknown mobile device (i.e., no gold hash list available), Hawkeye can still achieve a speed improvement during the acquisition process, since it will likely encounter all-zero blocks.

In the *hawkeye* algorithm (Fig. 1), the most time-intensive portion is triggered when an examined block is non-zero, has an initial gold hash list match, and then is unable to find a 1 MiB hash match. These situations are considered expensive because of the number of lengthy operations involved: two hashes performed and 1 MiB of data sent over USB to the server. Findings showed that when 4 KiB was used as the small blocksize, an average

[11] The number of repeated 1 MiB block patterns drastically declined after discovering the high prevalence of all-zero blocks in the corpus. No other block pattern occupied more than 1% of the total blocks observed.

range of 300–700 blocks would need to be hashed twice and transmitted. When the 64 KiB small blocksize was used, the average range was reduced to 100–300 blocks. As a result of these findings, we used a 64 KiB hash size instead of 4 KiB in the *hawkeye* algorithm.

Additionally, we calculated the amount of duplicate blocks that were not part of the gold hash list. Processing these blocks would cause Hawkeye to transmit data twice (i.e., the duplicate block sending scenario discussed in Section Per thread algorithm). We found that the amount of occurrences across the Purdue corpus was relatively small (average of less than 100 occurrences). On some images, duplicate blocks were not present. This provided assurances that the presence of these duplicates should be considered corner cases that should have negligible effect on Hawkeye performance.

## *Hawkeye benchmarks*

A series of tests to measure the runtime of Hawkeye were performed against devices with varying amounts of data.

The first set of tests were done on Samsung Galaxy S3 SGH-I747 model devices while connected via USB 2.0 to an Ubuntu 14.04 virtual machine (VM) with one processor and 2 GiB of RAM. The VM was hosted on a Windows 7 Enterprise machine with an Intel® Core™ i7 2.70 GHz processor and 4 GiB of RAM.

These experiments dealt with real-world devices and data. We took advantage of the smartphone image corpus collected during the Purdue experiment (Section Corpus of phone images). PMF's write-back capability was used to write images to devices, which were then imaged with Hawkeye. We tested six selected images, each originating from a different device. The date of collection factored into the image selection decisions: we wanted images that were collected toward the end of the Purdue experiment, maximizing the amount of usage each device endured.

Table 1 contains a list of the benchmarks seen when Hawkeye was used to perform physical acquisitions on these devices.

For the benchmarks in Table 1, the timer was started when the *hawkeye* binary began executing and stopped when all necessary blocks were received by the server. Each device contained 15,023 total blocks (blocksize = 1 MiB). Traditional forensic acquisition tools would normally send all 15,023 blocks to a listening service, taking a significant amount of time. Based on the send times from Table 1, *hawkeye*, on average, needs only 389 s (6 min, 29 s) with a standard deviation of 86 s to transfer all necessary data and achieve the same result as that of a traditional forensic acquisition tool. Note that the images acquired by Hawkeye were validated through hash comparisons and matched with those of the original images from the corpus.

**Table 1**
Hawkeye acquisition benchmarks with real data.

| Phone ID | # Blocks sent w/Hawkeye | Hawkeye time in secs (Avg. 3 trials) |
|---|---|---|
| 4 | 797 | 324 |
| 15 | 932 | 333 |
| 29 | 2143 | 412 |
| 30 | 3797 | 528 |
| 33 | 309 | 302 |
| 34 | 2567 | 434 |

Known blocks and all-zero blocks were not sent to the server; however, all blocks did have reference data transmitted. The time to send all the references was minimal (less than 1 s). Reference data included items such as an MD5 hash and an index value.

The second set of tests compared Hawkeye acquisition times to existing tools and techniques; results are listed in Table 2. Existing tools and techniques include Cellebrite UFED 4PC v4.5.0.307, Cellebrite UFED Touch v4.1.0.367, MSAB XRY v6.16, Magnet Acquire v2.0.0.0699, Oxygen Forensic® Extractor v8.0.3.199, and dd/netcat over ADB.[12] Each tool was used to physically acquire the same well-populated[13] 16 GB Samsung Galaxy S3. While the same phone was used throughout these tests, the state of it varied per tool. Some tools required the phone to be booted into normal mode, which inherently makes changes to a device's storage. Other tools ran in the phone's recovery mode or through the bootloader, which are more forensically sound. Due to the variable phone states, some of the acquired images slightly differed from each other. We were, however, able to verify exact hash matches among images produced by Cellebrite, XRY, and Hawkeye/PMF, since they similarly produced a raw bit-for-bit copy of the device storage without making changes to the target device. The collection platforms also varied by tool. Most tools were run on the aforementioned Windows machine; Hawkeye was executed on a bare metal 64-bit Ubuntu 14.04 laptop with 2 GB RAM and a 1.73 GHz Intel Core i7 processor. The times in Table 2 were provided by each tested tool after a successful acquisition. If the acquisition time was not reported by a tool (as was the case for the non-commercial tools/techniques), the UNIX *time* command or Windows PowerShell *Measure-Command* output was used.

A third set of tests were performed on Samsung Galaxy S3 and Nexus 4 devices to measure Hawkeye's best and worst case scenarios. For the best case scenario, the devices were baselined (99% of the unused userdata and cache partitions were filled with NULL blocks). This enabled us to measure the core speed of the *hawkeye* algorithm. No raw block data was sent to the server since the gold hash list

**Table 2**
Comparison of existing tool acquisition times of a Samsung Galaxy S3.

| Tool/technique | Time in seconds |
|---|---|
| dd/netcat | 11,050 |
| Cellebrite UFED Touch | 10,800 |
| Magnet Acquire | 7260 |
| Cellebrite UFED 4PC | 6942 |
| Oxygen Forensic Extractor | 4415 |
| MSAB XRY | 3240 |
| Hawkeye | 635 |

[12] A blocksize of 1 MiB was used with dd. Instructions on how to use dd and netcat over ADB can be found here: http://freeandroidforensics.blogspot.com/2014/08/live-imaging-android-device.html.

[13] The tested device's data partition was over 70% full of user-generated content, most of which consisted of unique JPG files.

contained hash entries for all the non-zero blocks on the devices. The results of the test can be found in Table 3. Note that the speeds listed in the “Baseline” column in Table 3 are not representative of real-world scenarios, since no user-generated content was present on the devices.

In measuring the worst-case scenario for Hawkeye, the devices were filled with random data. Nearly every block of these devices would need to be sent (excluding blocks containing known OS data). The devices were prepped by filling 99% of their unused userdata and cache partitions with random data from /dev/urandom. Results of these tests outperformed all existing solutions and can also be found in Table 3.

Finally, three additional tests were performed to specifically measure the speed improvement gained through our use of threads, Hawkeye's hash comparison function, and its zero block comparison function. The tests were performed using a baseline Galaxy S3 with a capacity of 16 GB (this was the same baseline device used in Table 3). The first test involved the removal of the hash comparison function, zero block comparison function, and threading from Hawkeye, resulting in an acquisition time of 1164 s. In the second test, threads were reinstated but the hash comparison and zero block comparison functions were removed. This resulted in a runtime of 957 s. In the last test, only the zero block comparison function was removed from Hawkeye (threading and the hash comparison function were used). The resulting time to acquire the image was 348 s. These results are listed in Table 4 and analyzed in Section (Analysis of results).

## Discussion

Discussion points herein include an analysis of experiment results, device encryption implications, and the applicability of Hawkeye to other platforms.

### *Analysis of results*

The Hawkeye runtimes displayed in Tables 1–3 mark significant improvements over current approaches used by mobile forensics tools to perform physical acquisitions. Table 1 shows that acquisitions of a common device containing real-world data can be acquired in under 7 min. Table 2 emphasizes that Hawkeye is at least 5X faster than other compared commercial tools or techniques. Table 3 highlights the theoretical minimum and maximum runtimes for two different devices.

**Table 3**
Hawkeye best-case /worst-case benchmarks.

| Device | Capacity | Baseline (secs) | Random-filled (secs) |
|---|---|---|---|
| Nexus 4 | 8 GB | 93 | 401 |
| Galaxy S3 | 16 GB | 265 | 891 |

**Table 4**
Measuring speed gain of various Hawkeye features.

| Device | All features removed (secs) | Threading only (secs) | Threading and hash comparison function (secs) |
|---|---|---|---|
| Galaxy S3 | 1164 | 957 | 348 |

The maximum runtime results in Table 3 greatly exceeded our initial expectations. We hypothesized that devices filled with random data would perform at or near the speeds of current tools. The results show otherwise and further emphasize that differential analysis is not the only major factor in improving the agent's runtime. In this case, differential analysis was minimally effective against the userdata and cache partitions since they were mostly filled with random data. Although the impact of using differential analysis was minimized, Hawkeye was still found to achieve significantly faster acquisitions than current tools. We surmise that Hawkeye's native code implementation and code optimizations led to the majority of the speed improvements achieved.

The final three tests from Section (Hawkeye benchmarks) allowed us to specifically measure the performance gained through the use of threading, the hash comparison function, and the zero block comparison function in our baseline Samsung Galaxy S3. The results emphasize how effective these techniques are in reducing the physical acquisition time. We found that by using a combination of all three features, Hawkeye achieved a speed improvement of 15 min. This was calculated by subtracting the time of the full featured run (265 s—Table 3 baseline test) from the time of the run that had all features removed (1164 s). Furthermore, we calculated each feature's individual contribution to the speed improvement. Threading accounted for 207 s, the hash comparison function reduced the time by 609 s, and the zero block comparison function caused an 83-s improvement. Note that the time gained from the zero comparison function depends entirely on the number of all-zero blocks in a device. In this case, the device tested contained many all-zero blocks. While these three factors represent significant time-saving elements, they do not account for the full difference in speed between Hawkeye and other compared tools and techniques. This fact lends support to our aforementioned theory that our native code implementation and code optimizations were the most effective factors in achieving faster acquisition times.

### *Working with encryption*

Mobile devices with encryption enabled do not prevent Hawkeye from performing physical acquisitions. Hawkeye makes no distinction on a device's encryption status during its operation. Note that on Android and iOS devices, encryption settings only apply to the userdata partitions (Mayer and Suarez; Edwards, 2016). Partitions such as system (Android and iOS), cache (Android only), and others remain unencrypted.

With the release of TWRP 3.0, a user's passcode can be entered to decrypt the userdata partition of an Android device in recovery mode.[14] This action would enable Hawkeye to acquire the userdata partition in an unencrypted form. If Hawkeye acquires encrypted data, the

[14] TWRP's userdata decryption feature is device dependent. Full TWRP 3.0 release notes can be found here: https://twrp.me/site/update/2016/02/05/twrp-3.0.0-0-released.html.

decryption process would need to occur in a post-acquisition phase, which may pose some challenges depending on the target device's hardware and OS version.

### *Applicability to non-Android platforms*

While the majority of our work on Hawkeye has been tested on Android, the tool's underlying technical acquisition techniques can be applied to a wider variety of environments and platforms such as iOS, system-on-chip (SoC) devices, and hard drives. The approach should be viable as long as a Hawkeye implementation has the ability to read the storage on a device of interest.

A Hawkeye implementation for iOS would need to read the blockdevice files in the /dev directory. On the iPhone 5, the /dev/disk0s1s1 and /dev/disk0s1s2 blockdevices represent the system and data partitions, respectively. A jailbroken iPhone would allow the permissions necessary for Hawkeye to perform a physical acquisition. Similar approaches could be applied to other platforms and environments.

## Conclusion

Using a variety of techniques which deliver significant time savings, we were able to develop an imaging agent that can physically acquire a mobile device faster than any current tool found commercially or in research. The applications for our approach include crime scenes, border crossings, and any other situation where performing a mobile forensic acquisition is time-sensitive.

Our results indicate that commercial vendors that have developed mobile acquisition capabilities would benefit from adding functionality similar to Hawkeye. Furthermore, it is theorized that commercial forensic kit makers can perform all necessary preparation for their supported mobile devices in a similar fashion to what is currently done today, leading to an easy adoption of the methods described herein.

The potential impact on mobile forensics in general would be significant, as dramatic speed-ups during the acquisition phase would allow data to get into the hands of analysts more quickly. Analysts would then have the opportunity to perform a more thorough examination using a physical image instead of a logical file set.

Future work involves more testing and expansion of several areas within the Hawkeye project. Acquisitions performed on devices that support USB 3.0 were not tested as part of our experiments; future research should be conducted to discover any performance differences over such a mechanism. Testing devices larger than 16 GB were also left for future experimentation. Work to acquire and provide more thorough baseline data (i.e., gold hash lists) remains. Further exploration of the use of full disk encryption should be explored as it becomes more ubiquitous. Also, as devices grow in storage size, compression should be reexamined. Finally, the techniques found in Hawkeye can be expanded and applied to other forensic areas besides mobile forensics.

## Acknowledgments

We share the credit for the work with Kyle Anthony, Jared Ondricek, Christina Johns, Eric Katz, and Lauren Sharpe for their contributions both with development of PMF and Hawkeye and review of the findings. The authors would also like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

## References

Edwards, S. iOS imaging on the cheap!. Mac4n6com, 23 March 2016. Web. http://mac4n6.com/blog/2016/3/23/ios-imaging-on-the-cheap.

Florin, T. The Kings of Storage Space-Smartphones with 128 GB of Internal Memory PhoneArena. N.p., 31 Mar. 2015. Web. http://www.phonearena.com/news/The-kings-of-storag-space-smartphones-with-128-GB-of-internal-memory_id67790.

Garfinkel S, Nelson A, White D, Roussev V. Using purpose-built functions and block hashes to enable small block and sub-file forensics. Digit Investig 2010;7:S13–23.

Garfinkel S, Nelson AJ, Young J. A general strategy for differential forensic analysis. Digit Investig 2012;9:S50–9.

Grier J, Richard G. Rapid forensic imaging of large disks with sifting collectors. Digit Investig 2015;14:S34–44.

Guido M, Ondricek J, Grover J, Wilburn D, Nguyen T, Hunt A. Automated identification of installed malicious android applications. Digit Investig 2013;10:S96–104.

Guido, M.,Brooks, M.,Grover, J.,Katz, E., Ondricek, J., Rogers, M., Sharpe, L. (in press). Generating a corpus of mobile forensic images for masquerading user experimentation. J Forensic Sci.

Gurjar S, Baggili I, Breitinger F, Fischer A. An empirical comparison of widely adopted hash functions in digital forensics: does the programming language and operating system make a difference?. In: Proceedings of the conference on digital forensics, security and law; 2015, May. p. 57–68.

Laurenson T, MacDonell S, Wolfe H. Towards a standardised strategy to collect and distribute application software artifacts. 2015.

Lessard J, Kessler G. Android forensics: simplifying cell phone examinations. Small Scale Digital Device Forensics J 2010;4(1):1–12.

Mayer, D and Suarez D. Faux disk encryption: realities of secure storage on mobile devices. Web. https://www.blackhat.com/docs/eu-15/materials/eu-15-Mayer-Faux-Disk-Encryption-Realities-Of-Secure-Storage-On-Mobile-Devices-wp.pdf.

Son N, Lee Y, Kim D, James JI, Lee S, Lee K. A study of user data integrity during acquisition of android devices. Digit Investig 2013;10:S3–11.

Spector, L. USB 3.0 speed: real and imagined. PC World, 26 June 2014. Web. http://www.pcworld.com/article/2360306/usb-3-0-speed-real-and-imagined.html.

Vidas T, Zhang C, Christin N. Towards a general collection methodology for android devices. Digit Investig 2011;8(Suppl.):S14–24.

Watkins K, McWhorte M, Long J, Hill B. Teleporter: an analytically and forensically sound duplicate transfer system. Digit Investig Sept, 2009;6(Suppl.):S43–7.

Yang SJ, Choi JH, Kim KB, Chang T. New acquisition method based on firmware update protocols for android smartphones. Digit Investig 2015;14:S68–76.