Detection of Cross-Instance Cloud Data Remanence via Sector-Level Differential Analysis and Fragment Source Attribution

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

by

Bradley Lee Snyder
Master of Science
Virginia Polytechnic Institute and State University, 2008
Master of Business Administration
Virginia Polytechnic Institute and State University, 2007
Bachelor of Science
Virginia Polytechnic Institute and State University, 2004

Director: James Jones, Associate Professor
Department of Electrical and Computer Engineering

Spring Semester 2019
George Mason University
Fairfax, VA

DEDICATION

For Stephani, Alexander, and Adam.

ACKNOWLEDGEMENTS

Throughout the process of writing this dissertation I have benefited greatly from the knowledge and expertise of my committee. In particular my Director, Dr. James Jones, has been critical in the process since inception. If it were not for his initial class in digital forensics with a heavy emphasis in Python, this dissertation may have taken a completely different course. His guidance regarding the structuring of my experiments and methodology was also invaluable. Most importantly was his patience and understanding of a student with too many of life's obligations attempting to fulfill a life-long dream.

I also wish to extend deepest gratitude to Professor Bob Osgood for recognizing the academic potential within me and giving me the opportunity to achieve.

Finally, I would like to thank my Mother and Father for inspiring me to be an engineer and continuing to push myself beyond what I thought possible. Thank you for believing in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

Amazon Machine Image ..... AMI
Application Programming Interface ..... API
Amazon Web Services ..... AWS
Central Processing Unit ..... CPU
Contiguous Remanence Event ..... CRE
Defense Information Systems Agency ..... DISA
Downloaded Virtual Machine ..... DVM
Elastic Computing Cloud EC2
Infrastructure as a Service ..... IaaS
Information System ..... IS
Information Technology ..... IT
Master File Table ..... MFT
National Institute of Standards and Technology ..... NIST
New Technology File System ..... NTFS
Operating System ..... OS
Other Virtual Machine Type ..... OVMT
Random Access Memory ..... RAM
Remote Desktop Protocol ..... RDP
Source Marked Remanence ..... SMR
Secure Shell ..... SSH
Source Unmarked Remanence ..... SUR
The Sleuth Kit ..... TSK
Tested Virtual Machine Type ..... TVMT
Veterans Affairs ..... VA
Virtual Computing Lab VCL
Virtual Machine ..... VM
Virtual Machine Manager ..... VMM
Virtual Machine Type ..... VMT

ABSTRACT

DETECTION OF CROSS-INSTANCE CLOUD DATA REMANENCE VIA SECTOR-LEVEL DIFFERENTIAL ANALYSIS AND FRAGMENT SOURCE ATTRIBUTION

Bradley Lee Snyder, Ph.D.

George Mason University, 2019

Dissertation Director: James Jones

Modern cloud providers provision virtual machines for different customers from a common infrastructure of persistent storage, volatile memory, and processors. The hard disk space, RAM, and processor resources allocated to a new instance were previously in use by one or more other instances, where these other instances may have been used by other customers. If the cloud provider does not adequately sanitize data resident in these resources between allocations, then resources allocated to a new instance may include data from a previous instance. Such leakage across cloud virtual machine instances is an example of data remanence and may reveal personal or sensitive information to unauthorized parties. Detection of this kind of data remanence on hard disk cloud resources is the subject of this work.

Data remanence concerns date back at least to the United States government-issued “Rainbow Series” books of the early 1990s, and concerns of cross-instance data

remanence in cloud environments were raised as far back as 2009 (Mather, S., & Latif, 2009). To date, efforts to detect cross-instance cloud remanence have consisted of searching current instance unallocated space for fragments easily attributable to a prior user or instance, and results were necessarily dependent on the specific instances tested and the search terms employed by the investigator. In contrast, this work developed, tested, and applied a general method to detect cross-instance cloud remanence that does not depend on specific instances or search terms. This method collects unallocated space from multiple instances based on the same cloud provider template. Empty sectors and sectors which also appear in the allocated space of that instance are removed from the candidate remanence list, and the remaining sectors are compared to sectors from instances based on other templates from that same provider; a matching sector indicates likely cross-instance remanence. Matching sectors are further evaluated by considering contiguous sectors and mapping back to the source file from the other instance template, providing additional evidence that the recovered fragments are in fact from another instance.

This work first found that unallocated space from multiple cloud instances based on the same template is not empty, random, nor identical - in itself an indicator of possible cross-instance remanence. This work also found sectors in unallocated space of multiple instances that mapped directly to contiguous portions of files from instances created from other templates, definitively proving cross-instance remanence. This work also identified multiple sectors in unallocated space which were not mapped to other

known instances; such sectors could be from instance templates not included in these tests, from infrastructure operations, or from user data in another instance.

This work contributes a general method to detect cross-instance cloud data remanence which is not dependent on a specific provider or infrastructure, instance details, or the presence of specific user-attributable remnant fragments. The method was based on the known operation of cloud environments, and a tool to implement the method was developed, validated, and then run on two enterprise cloud environments: a university and Amazon's AWS cloud services. Cross-instance remanence was found in both cases.

1 INTRODUCTION

Cloud service providers deliver computing resources to customers from a common infrastructure. It is projected that by 2020, 83% of enterprise workloads will be hosted in the Cloud (LogicMonitor, 2018). Not only has it become an essential part of most corporations' business models, but also part of many governments' information technology infrastructures. There is a perception that the providers of cloud services have solved many of the security issues plaguing on-premise systems. However, research in the area of cloud security has shown repeatedly that security gaps remain, leaving these organizations and their data vulnerable (Gemalto, 2017) (Mather, S., & Latif, 2009). One of these areas is cross-instance cloud data remanence in IaaS (Infrastructure-as-a-Service) clouds, i.e., data from one cloud virtual machine instance is available in a different cloud virtual machine instance. This work specifically addresses subsequent virtual machine instances which use some of the same cloud infrastructure resources at different points in time. In a multi-tenant cloud, cross-instance cloud data remanence means the potential to view another user's data. Currently, no general and comprehensive mechanisms exist to test for cross-instance cloud data remanence; development of such a capability is the focus of this work.

1.1 Background of the Problem

Remanence in the context of information processing systems is "residual information remaining on storage media" (Regenscheid, Feldman, & Witte, 2015), i.e., data that remains after attempts to delete or overwrite the data. Historically, as a security

concern, this referred to magnetic storage media and specifically meant *magnetic remanence*, i.e., the residual magnetic field that persisted after a new field was applied. The ability to recover that residual field meant the ability to recover data that had been overwritten. Due to the advent of virtual and cloud computing, storage media is now shared by multiple virtual machines (VMs) over time, where different virtual machines may be allocated to different users. Each time a VM is decommissioned, the media once used to store the previous owner's data will soon be used by someone else with a new VM. If the previous user's data is not removed, a subsequent cloud user may extract and exploit data without physical access to the media. With this potential exploitation in mind, the security of cloud computing becomes increasingly important as more users, corporations, and governments rely on cloud services for storage, processing, and infrastructure. Determining data remanence in the cloud is challenging since any cross-instance data leakage is unpredictable. Finding overt user data from one cloud instance in another cloud instance is a clear indicator of cross-instance remanence, but such occurrences are rare in practice and the absence of such a finding does not preclude the existence of cross-instance remanence to any meaningful degree. One challenge that this work addresses is to develop a method which does not rely on finding such rare events.

Recent and current research in the area of cloud data remanence falls into three main categories: how remanence occurs, exploitation of remanence discovered, and solving the problem. The literature discussing how remanence occurs makes the reader aware that data remanence can occur in cloud environments, explains that a shared infrastructure is the reason why, and multi-tenancy creates the risk of a data

confidentiality violation. Exploitation research presents various ways to extract data exposed by a shared infrastructure and cloud multi-tenancy. Such exploitation includes side channel attacks, which leverage concurrent instances, and inspection of the raw data in unallocated space and file slack for confidential information, which leverages sequential instances. Much of the literature related to existence or exploitation also attempts to present various solutions to the issues discussed. A common recommendation is to encrypt the virtual drive and then simply not store the key so that any surviving data will not be in plaintext. However, this technique requires additional computational resources during each VM allocation and use, which increases time and resources to provision and operate, potentially increasing cloud provider or end user costs. (DevTiw, 2017) Disk encryption may be implemented by the end-user, in some cases as a provider-supplied option, although users have historically not used such capabilities even when available and free.

1.2 Statement of the Problem

Cross-instance cloud data remanence is a significant risk to users, corporations, and governments using multi-tenant or multilevel security cloud environments. Exposed data may be sensitive, including health records, financial data (e.g., sales, banking, and stock transactions), proprietary corporate data (e.g., technical designs, new product characteristics, corporate records, and internal book-keeping), intellectual property, critical infrastructure data, and government law enforcement and national security data. Exploitation and abuse of such data could have significant consequences to individuals, corporations, and governments. Historical methods and tools for determining data

remanence are not easily utilized in cloud environments, and do not provide unequivocal determination of data remanence. A comprehensive, definitive, and broadly applicable method for identifying cross-instance cloud data remanence is needed.

1.3 Purpose of the Research

The purpose of this work is to develop a theoretical foundation, methodology, and validated tool for determining whether data remanence exists in VMs instantiated by cloud service providers, and to apply that methodology and tool to a commercial cloud provider's environment.

1.4 Significance of the Research

Typical cyber security activities are often handled by the cloud providers due to the complexity of remote configuration, patch management, malware detection and protection, access and permission, authentication, encryption, and other security controls, and the relative ease and efficiency with which cloud providers can provide these services. Corporations depend on the systems of the large providers, such as Amazon and Microsoft, to perform these tasks, with 64.9% of IT leaders believing that the Cloud is as secure as, or more secure than, on-premises infrastructure (McAfee, 2017). However, according to the same document, 67.8% of IT leaders stated that the main barrier of entry to cloud computing is the lack of a capability to enforce security requirements. This finding coincides with another article stating that companies that utilize cloud computing have minimal to no capability to check on whether the cloud providers are applying security requirements at an adequate level (Corbin, 2015). One of the security controls,

unique to a cloud computing environment, is that prior instance data has been removed when a virtual machine is decommissioned, yet data remanence is a serious security concern that can be exploited with substantial consequences (Garfinkel & Shelat, 2003). The fact that private information could be seen by another party is, at least, an invasion of privacy to the average user, and could expose passwords for financial transactions, health records, encryption keys, etc. For private industry that uses these same cloud services, intellectual property and proprietary information could be exposed, leading to a potential loss of competitive edge. For government agencies, the leaked information could escalate to a potential threat to national security. The Cloud's footprint will only increase in the coming years, and so will the number of participants with sensitive information. Therefore, it is critical that a method is developed for determining whether cloud providers are removing previous customers' data before reassigning the physical infrastructure (e.g., hard drives) to new customers. Such a method would provide verification that cross-instance data remanence is being properly addressed by cloud providers, increase confidence in usage of the Cloud, and reduce potential exploitation of data stored in the Cloud.

1.5 Research Goals and Expected Contributions

Research Goals:

1. Establish a theoretical basis for the possible existence of cross-instance data remanence within a virtual machine, based on the inherent characteristics of storage media and virtual machine hosts and cloud infrastructures.

2. Develop a practical methodology to positively determine the existence of remanence which is not infrastructure, instance, or user data dependent.
3. Develop and validate a tool to implement the methodology.
4. Apply the tool to evaluate a commercial cloud service provider.

Expected Contributions.

1. A methodology and associated tool(s) for establishing the existence of cross-instance data remanence in a cloud environment, where the methodology does not depend on a specific infrastructure, instance details, or user data.
2. The results of applying this tool to a commercial cloud provider.

1.6 Research Design

The Research Design comprises five parts: theoretical foundation, data remanence detection approach, data remanence detection tool, data remanence detection tool validation, and cloud provider evaluation. These parts are covered in detail in their respective sections, and are briefly summarized in the following paragraphs.

1: **Theoretical Foundation:** Develop the theoretical foundation to determine data remanence in a VM based on technical knowledge of VMs and the method by which data is stored on hard drives within a cloud environment. After conducting research and analysis, it was determined that VMs in a cloud environment are consecutively provisioned which potentially creates overlapped data within the unallocated space.

2: **Data Remanence Detection Approach:** Develop the approach encompassing three general steps for positively identifying data remanence in the Cloud: setup,

detection, and file extraction. During the setup phase, VM types are chosen from the ones offered by the cloud provider. Sector hashes of the offered VM types are generated and referenced in a configuration file for use in comparisons during detection. During the detection phase, an instance of one of these VM types is created and the disk contents are collected (remotely imaged to a local storage drive) for testing. Sector hashes of the unallocated space are then generated for this VM type and compared with the hashes in the configuration file. Thus, sector hashes of the tested instance are compared against sector hashes of the known sectors in the offered VM types (in the configuration file). If unique hashes are found in unallocated space of the tested instance from known VM types, other than the VM type of the tested instance, then these hashes represent potential data remanence. The sectors associated with the identified hashes are then inspected to positively identify if the VM contains previous instance data by ensuring that the data within the sector cannot be found within the allocated space of the tested VM. This determination is further verified during the file extraction phase when the actual file associated with the potential remanence is extracted from the image and checked to see if it contains the raw data remanence string. If it does, then the file associated with the data remanence has been positively identified as well. The process of collecting an instance and examining the VM for remanence is executed multiple times for each cloud provider.

3: **Data Remanence Detection Tool:** Develop a data remanence detection tool to implement the approach using Python.

4: **Data Remanence Detection Tool Validation:** Validate the tool by conducting controlled testing in a laboratory environment using a) artificially-induced data remanence in local VMs, and b) a local cloud environment.

5: **Cloud Provider Evaluation:** Verify the method by running a small-scale test of a representative cloud service provider (Amazon). Amazon Web Service (AWS) has the largest market share with revenues reaching twice those of the next three competitors combined (Tung, 2016).

1.7 Scope

1. Only unallocated virtual machine hard disk space is tested; RAM, volatile processor data, and hard disk file slack space are not checked. Regarding hard disks, data remanence could be found within file slack space. However, there is typically much more unallocated space than file slack space, so the likelihood of cross-instance data remanence detection in file slack space would be small. Since a positive result (detected cross-instance data remanence) in disk unallocated space answers the primary question (is data from one instance available in a subsequent instance), further effort does not provide additional benefit. A similar argument, i.e., the chance of remanence detection is small compared to disk unallocated space, can be made for RAM and volatile processor data. Should the goal of a particular activity be to capture *all* remanence events, then all possible sources should be checked, including file slack space, RAM, and volatile processor data.

2. Only VMs will be evaluated that run either: 1) Windows operating systems that are capable of Remote Desktop Protocol (RDP), or 2) Linux operating system variants capable of Secure Shell (SSH). Modern operating systems from these two families will be the focus of the research, however the method applies to any operating system that meets these access criteria (RDP or SSH). Further, the method applies to platforms without RDP or SSH, but changes to the tool implementation would be necessary.

1.8 Limitations

While the methodology and associated tool identify all data remanence, the method and tool are not designed to positively identify user generated remanence.

This restriction means that previous user data will not be automatically detected using this method. However, if positive data remanence is found within the VM, it means that user data is more likely to be found within the unallocated space of the VMs provisioned by the provider if other techniques are utilized to find the information. The tool identifies the locations of data remanence within the unallocated space on which other techniques can be used to detect previous user data.

1.9 Assumptions

The VM image provided by the cloud provider will not change frequently upon each given provisioning of the requested VM as numbered in the cloud provider's catalog.

As discussed in the theoretical foundation section, it is believed that once a cloud provider has assigned a provisioning number to the VM, due to the practice of templating, the allocated space within the VM will not change upon each provisioning within the short time window necessary for testing. For example, two VMs can be extracted from the cloud provider within hours of each other, and then compared for data remanence. The cloud provider would not have changed the image of the OS and applications for a given version (template) of the VM within that short period. If the image were changed in a short period, then the user would not have stability of system configuration, patch versions, and capabilities in order to employ the VM.

1.10 Definition of Terms

Allocated Space: The clusters on a partition that have been formally assigned files and are enumerated by the operating system.

Cloud: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell & Grance, 2011).

Contiguous Remanence Event (CRE): CRE is where a 512 byte chunk of remanence is found with another remanence event immediately contiguous. A CRE is a single event, irrespective if it is a set of two or more contiguous chunks.

Data Remanence: The residual representation of digital data that remains even after attempts have been made to remove or erase the data (Gallagher, 1991).

Host: The base hardware that the virtual machine runs on.

Hypervisor: "The virtual machine manager (VMM) (also known as a hypervisor) creates and runs virtual machines by providing an interface that is identical to the host. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others" (Silberschatz, Galvin, & Gagne, 2012).

Instance Data: Data that is contained within the allocated space of a provisioned VM.

Multiplexing: See Multitenancy. (Can have multiple meanings in the context of VMs, but is being used synonymously with Multitenancy for the purpose of this paper.)

Multitenancy: "The cloud characteristic of resource sharing. Several aspects of the [Information System] are shared including, memory, programs, networks and data. Cloud computing is based on a business model in which resources are shared (i.e., multiple users use the same resource) at the network level, host level, and application level. Although users are isolated at a virtual level, hardware is not separated" (Zissis & Lekkas, 2012).

Provision: Establish an instance of a VM type within a hypervisor.

Sector Hashing: Hashing an image file in the number of bytes that equals the sector size of the image being evaluated. A common sector size is 512 bytes.

Side-Channel Attack: Attacks that rely on information that is retrieved from the encryption device itself instead of the algorithm utilized by the device (Kelsey, Schneier, Wagner, & Hall, 1998).

Slack Space: The unused space at the end of a file in a file system that uses fixed clusters (so if the file is smaller than the fixed block size then the unused space is simply left) (Kent, Chevalier, Grance, & Dang, 2006).

Source Marked Remanence (SMR): Remanence detected from known data derived from previously downloaded VMs from the specific cloud provider being tested.

Source Unmarked Remanence (SUR): Remanence detected with unknown origins.

Unallocated Space: "The unallocated space is the group of clusters not in active use by any file; data within this space could have come from any file" (Phillip, Cowen, & Davis, 2009).

Virtual Machine: "The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer" (Silberschatz, Galvin, & Gagne, 2012).

VM Type: An image or template of a specific VM offered by a Cloud Provider. Each VM type can differ from other images or templates by instantiating differing operating system, software packages, or updates and patches.

2 REVIEW OF THE LITERATURE

Cloud storage and processing providers and users continue to grow (Gupta, Seetharaman, & Raj, 2013). Along with this growing adoption, research specifically focused on cloud data remanence, under the general topic of cloud forensics, has seen some attention. However, as shown in this work, gaps remain in terms of sanitizing shared cloud resources and positively detecting remanence when sanitization is incomplete.

A review follows of the relevant research broken into two broad categories: hard drive analysis and cloud forensics.

2.1 Hard Drive Analysis

General research in data remanence has been an on-going field of investigation since at least the 1990s (Gallagher, 1991). For most of that time, the primary focus has been deleted files stored on local magnetic or solid-state storage devices. For example, utilizing sector hashing, researchers are able to locate fragments of files once deleted (Garfinkel & McCarrin, 2015). Other researchers use a sector hashing method to quickly sift through hard drive images for specific target data (Taguchi, 2013). The use of sector hashing is useful for scrutinizing large amounts of data in a short amount of time, however the main use is only when there is a known database of hash values to reference. The tool *hashdb* was developed to assist with matching these known hash values (Allen, 2014). A related technique is "hash-based carving", where each hashed sector is compared to hashes of known target files (Collange, Daumas, Dandass, & Defour, 2009).

Bulk_extractor (Garfinkel, 2013) is another tool that is widely used to extract files and strings from an image. *Bulk_extractor* uses underlying regular expressions to find strings of interest with no dependence on the underlying operating system (Alherbawi, Shukur, & Sulaiman, 2016).

Sector hashing is used extensively in this dissertation to break an image into multiple hashed parts and is an example of the concept of "piecewise hashing". Essentially, this is taking a single file and breaking it into blocks (often sectors) where each block is hashed separately. Piecewise hashing has been used in multiple applications since first developed by Nick Harbour (Harbour, 2002), and by researchers afterward (Young, Foster, Garfinkel, & Fairbanks, 2012). The majority of research though is centered around "Context Triggered Piecewise Hashing". Also known as "fuzzy hashing", the algorithm matches inputs that have a certain number of homologies that would otherwise be lost with traditional hashing techniques. (Kornblum, 2006) Additional research in this field has been leveraged to expand on traditional computer forensics such that a file that has been slightly altered can still be detected during an investigation (Chen & Wang, 2008). While interesting, a contextual based approach could be detrimental to the research presented here, since an undeniable positive match is what is desired. For the purposes of this research, piecewise hashing 512 bytes at a time (essentially sector or sector-fraction hashing) is appropriate.

While these techniques are useful for many forensic applications, including cloud forensics and the detection of specific fragments or remanence, they have not been applied to the problem of detecting cross-instance cloud remanence generally.

Furthermore, most carving methods attempt to extract most, or all, of the file to determine contents or reconstruct a deleted file. The research presented in this dissertation is not concerned with specific contents, but rather the existence of any cross-instance remnants in a cloud provider's environment. This research does require post-processing of identified fragments to find disk locations (offsets) and associated source files. This function uses tools from The Sleuth Kit by Brian Carrier (Carrier, n.d.).

2.2 Cloud Remanence

2.2.1 How Remanence Occurs

There are a number of articles that raise the issue of cloud data remanence and discuss how such remanence might occur in a cloud environment, as well as raising awareness of its possible existence. The National Institute of Standards and Technology (NIST) has created Special Publication 800-125A which describes Security Recommendations for Hypervisor Deployment on Servers (Chandramouli, 2018). The publication discusses risks exposed by shared memory between virtual machines and raises concerns about this issue.

Multitenancy is the practice of multiple cloud customers sharing a common infrastructure and is fundamental to cloud computing. However, it exposes a number of risks with regard to confidentiality as discussed in *Security Issues in Cloud Computing* (Tianfield, 2012). One such risk is data remanence due to "virtual separation of logical drives and the lack of hardware separation between multiple users." As noted in the Tianfield work, this situation could lead to the disclosure of confidential data, either unknowingly or through deliberate action of an attacker. The paper asserts that a

malicious attack could occur by a user requesting a large amount of disk space from the cloud provider and then rummaging through the unallocated space for sensitive information. While this technique is a valid attack method, there is no discussion in the paper as to how to detect whether specific content is a case of cross-instance data remanence or whether such leaks are occurring on a large scale.

Multitenancy raises legitimate concerns for cloud providers and users. For example, researchers have experimentally proven within a specific, localized, cloud environment that remanence can occur (AlBelooshi, Salah, Martin, & Damiani, 2015). In Experimental Proof: Data Remanence in Cloud VMs, researchers established their own Xen hypervisor environment to determine if they could generate remanence. The Xen hypervisor was not chosen at random, but rather because Amazon, and many other cloud providers, utilize the Xen hypervisor as the basis of their architecture. (Amazon, 2019) According to the researchers, there is little information on how Amazon sanitizes their instances before reallocation of physical memory, except for a reference to NIST 800-88 ("Guidelines for Media Sanitization") which is cited as part of the decommissioning process (Amazon, 2017). This reference is not a descriptive statement since the guidelines are generic and, as will be discussed later in this dissertation, the exact implementation is crucial to determine if data from one user is actually removed before another user is able to access the information. Amazon's security document goes on to describe a proprietary method of virtualization within the Xen hypervisor architecture, but does not discuss how or where it is deployed.

Not only is multi-tenancy a possible generator of cross-instance data remanence within the cloud, but so are the templates that are commonly used and distributed within the cloud itself. Templates offer a "perfect, model copy of a virtual machine from which an administrator can clone, convert or deploy more virtual machines" (AlBelooshi, Salah, Martin, & Damiani, 2015). AlBelooshi, et al, state that these templates could potentially contain data that is sensitive to the creators which, in turn, would then be accessed by the cloud users that launch them. Their work in producing a process of extracting this accidentally leaked data from AMIs instantiated on Amazon's AWS yielded clear results establishing that templates should be properly sanitized before being made available for customer use. Similar findings were found in (Kirda, 2012) and (Bugiel, Numberger, Poppelmann, Sadeghi, & Schneider, 2011) where sensitive user data was leaked through user generated templates. In the former, even SSH keys were found within the templates that could be used to establish unauthorized access.

2.2.2 Exploitation of Remanence

Other published work discusses the implications of data remanence in the Cloud. These works go into detail regarding the different methods of attack, and the impact it would have if data remanence did occur in a cloud environment. The most significant article in this category was *Hey, You, Get Off of My Cloud* (Ristenpart, Tromer, Shacham, & Savage, 2009) which described a clever attack method on Amazon's AWS. If a cloud environment utilizes multiplexing as an approach to multi-user virtualization, then it is possible to launch cross-VM side-channel attacks to extract certain information from a victim's VM. This attack can occur only if the attacker and victim are on the same

physical machine, and the article does not discuss how to detect if data remanence exists. In fact, the data leakage discussed is not from information on the hard drive, but rather detecting resource usage of the victim's VM. There are a number of other articles that discuss extracting cryptographic secrets utilizing the same cache-based side channel attacks. However, even though these attack methods theoretically exist, they were challenging to exploit at the time (Osvik, Tromer, & Shamir, 2006). Published work describing these attacks do not address detection of data remanence.

Inspecting hard drive slack space and unallocated space is not a new phenomenon. There are a number of articles discussing how to exploit data that has been left on a system when system administrators thought hard drives had been sanitized. For example, one such article tested a number of hard drives that were removed from the United States Veterans Administration (VA) Medical Center in Indianapolis (Garfinkel & Shelat, 2003). Researchers using relatively unsophisticated techniques were able to extract information from the hard drives that were thought devoid of confidential information. While the methods used to search for remaining files cannot be applied directly to cloud data remanence detection, they can be used to search resources for data of interest regardless of the reason that data exists on the target resources. None of the works in this section address positively determining the existence of cross-instance cloud data remanence.

2.2.3 Solving the Problem

While the VA study above (Garfinkel & Shelat, 2003) does not discuss remanence in the Cloud, it does make valid points about solving the issue of sanitization

in general. The study discusses three different methods of sanitization: destruction, degaussing, and overwriting. In a cloud environment, destruction and degaussing are not a viable option due to multitenancy and the need for fast reuse of cloud infrastructure. Overwriting could work, but the article continues to discuss why sanitization efforts are overlooked in many organizations. One reason is due to "Failure to properly estimate the risk." (ibid, p. 31) Essentially, the current owner does not believe that any future owner would find the information that is left behind of value, or would use such data for malicious purposes.

The VMware ESXi Server 5.0 Security Technical Implementation Guide discusses remanence within a VMware based infrastructure in Finding ID V-39353 (Defense Information Systems Agency, 2017). The guide, published by the Defense Information Systems Agency (DISA), specifically states that the VMDK files housed by a system must be zeroed prior to deletion to ensure no sensitive information is left. Furthermore, there are multiple guides that discuss how to automate the process of zeroing a VM upon deletion from the VMware ESXi host.

Other published works provide various methods on how to solve data remanence in the Cloud as well as security controls to prevent it from occurring. The primary question that many companies have is "What happens to my data in the cloud?" This question is addressed in *Cloud Computing Security – Trends and Research Directions*, (Sengupta, Kaulgud, & Sharma, 2011) where the authors specifically address the data remanence problem by suggesting the cloud provider encrypt the entire drive before deleting the drive's contents. This approach seems to be a reliable method, if

implemented correctly, and is discussed in other work such as File System Design with Assured Delete (Perlman, 2006). However, encrypting the entire hard drive of the VM, even if not deleted, may not be a viable option for cloud VMs due to resource requirements and the need for fast reuse of cloud infrastructure.

2.3 Prior Work Summary

There has been much discussion of how traditional forensics can be applied to studying hard drive images collected from a cloud environment but no work on generally detecting cross-instance cloud data remanence. This work leverages some of this prior work and tools, but in a novel way. While research in cloud computing is ongoing in multiple areas, discovering remanence does not seem to be a major concern as evidenced by the lack of research being conducted in this area (Bayramusta & Nasir, 2016). While the security and reliability of the large-scale cloud providers have increased over the years, data remanence could still exist and expose cloud VM data to exploitation. Prior research does provide various ways to solve the problem of data remanence, with some solutions coming from the cloud providers themselves. However, many of these methods may be impractical or not scalable. While data remanence in a cloud environment is an obvious concern in the cybersecurity world, little has been done to determine the extent to which it exists in the wild, or to develop methodology and tools to locate it.

3 THEORETICAL FOUNDATION

This section provides the theoretical foundation of the research methodology and developed tool. The ideas presented here are agnostic to a specific implementation and include technical background, types of remanence, associated file recovery, and Contiguous Remanence Events.

3.1 Technical Background VMs

The research hinges on how VMs function at a fundamental level. Essentially, VMs, whether in the Cloud or operated on-premises, create abstraction layers where the hardware typically found on a solitary computer (e.g., CPU, RAM, hard drive) is split into multiple abstraction layers. These layers operate independently in a manner whereby the VM believes that the hardware is solely owned by the operating system controlling it. There are two types of hypervisors: Type 1 and Type 2. For a Type 1 hypervisor, the operating system installed directly onto the bare-metal server is the hypervisor itself. This installation gives the hypervisor the most control over the virtual machines it hosts. While in a Type 2 hypervisor configuration, the hypervisor is simply an application that is installed within an operating system. This application performs the functions of provisioning and allocating resources to the VMs. The operating systems installed on each VM is known as a Guest OS.

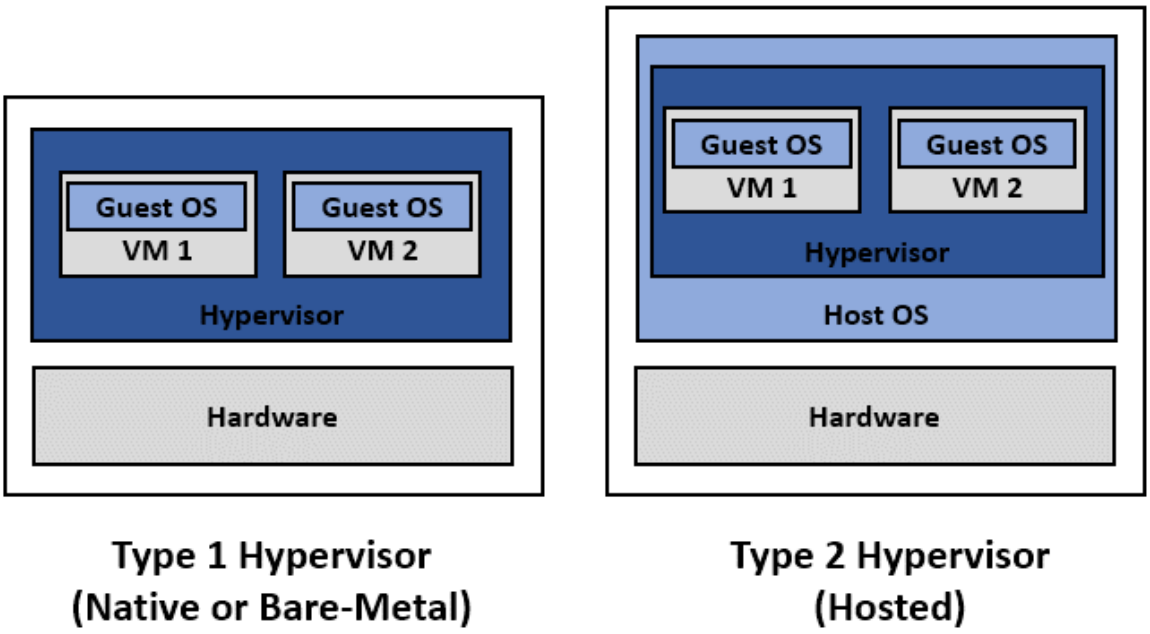


Figure 1 Types of Hypervisor Architectures

Cloud providers utilize, almost exclusively, Type 1 hypervisors to host provisioned VMs for their customers. The different environments are created using two main components: the host and the hypervisor. The host is the physical hardware of the system on which the VM is instantiated, while the hypervisor provides the interpretation between the host and the VM.

Theoretically, the hypervisor is providing an interface to the VM that is identical to the host (Silberschatz, Galvin, & Gagne, 2012). This configuration means that multiple guest operating systems can concurrently utilize the same resources provided by the host, which leads to increased resource efficiency, cost benefits, elasticity of performance, templating, duplication of VMs, and infrastructure scalability.

One major benefit of VMs is the use of templating, in which one standard VM image, including an installed and configured guest operating system and applications, is saved and used as a source for provisioning multiple running VMs (Silberschatz, Galvin, & Gagne, 2012). Templating is how cloud computing providers operate and create their VMs on the fly. When a user requests a new VM, the user is actually requesting a specific template, or VM Type, that is provided by the cloud provider. This templating means that as long as the same VM Type is being requested, the information found within the allocated space on the VM hard drive will remain the same. Also, if the unallocated space on the VM hard drive is initialized properly, it will also contain the same information as the template or, more accurately, be blank or otherwise sanitized.

For the methodology developed and reported in this work, it does not matter which type of hypervisor is used or the implementation method of the virtualization infrastructure. The method is simply a black-box approach and detects what is presented to the VM from the hard drive space allocated. This approach is necessary in order to be applicable across all cloud providers and since most cloud providers keep their infrastructure and method of implementation proprietary, preliminary assumptions are made regarding the method of provisioning for VMs in a cloud environment. For any given VM type, there is an associated OS (including OS type, version, and installed patches) and possibly applications.

3.2 Notional Origination of Data Remanence

The method used to locate data remanence is based on sector hashing of the allocated space for various VMs. It is anticipated that the underlying cloud environment is conceptually as shown in Figure 2.

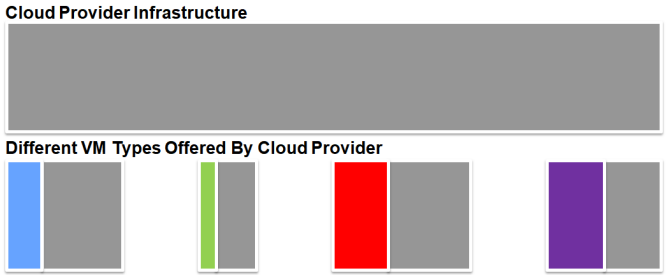


Figure 2 Notional Representation of the Provider's Cloud Environment Before Provisioning (top) and of Available VM Types (bottom)

The grey space is the entire cloud provider's available unallocated space used to provision VMs as requested by customers. Each of the colored objects in the figure represents different VM types, and typically operating systems, that the service provider offers and instantiates. The colored portion in Figure 2 is the allocated space of the VM while the associated grey section is the unallocated space of the hard drive. As customers request VMs based on VM types provided by the cloud service provider, the cloud hard drive space fills with VMs as shown in Figure 3.

Figure 3 Cloud Infrastructure Filled with Various VM Types.

At some point, customers will eventually request VMs to be removed, or moved, from the environment when the VMs are no longer needed. When the VMs are removed, it is anticipated that the allocated space remains on the cloud's hard drives, if data remanence were to occur. New VMs are added as users continually request new VMs. The remaining allocated space, if not initialized, will overlap into the new VMs unallocated space. Figure 4 represents this behavior where the patterned colors, indicated by arrows, are allocated space from previous VM instances that have remained on the hard drive. These places, where previously allocated space overlaps with the unallocated space of a current instance, are potential locations for cloud data remanence.

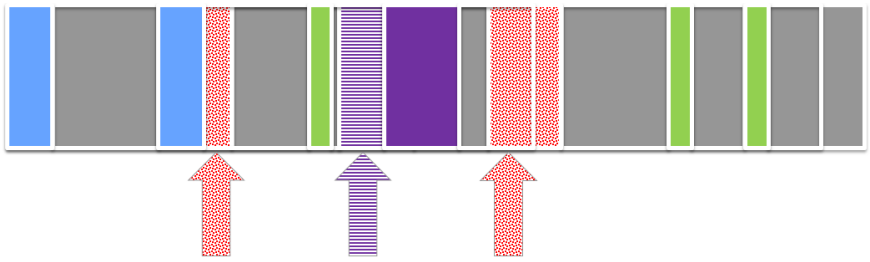


Figure 4 Patterned Allocated Space Represents Data Remanence in the Cloud.

Over time, as VMs are removed and re-provisioned, the amount of potential data remanence in a new VM single instance increases in both size and complexity. Instead of

coming from a single previous instance, the allocated space from multiple operating systems could be found within the unallocated space of a newly created VM. If one of the VMs were examined closely, it might look like Figure 5.

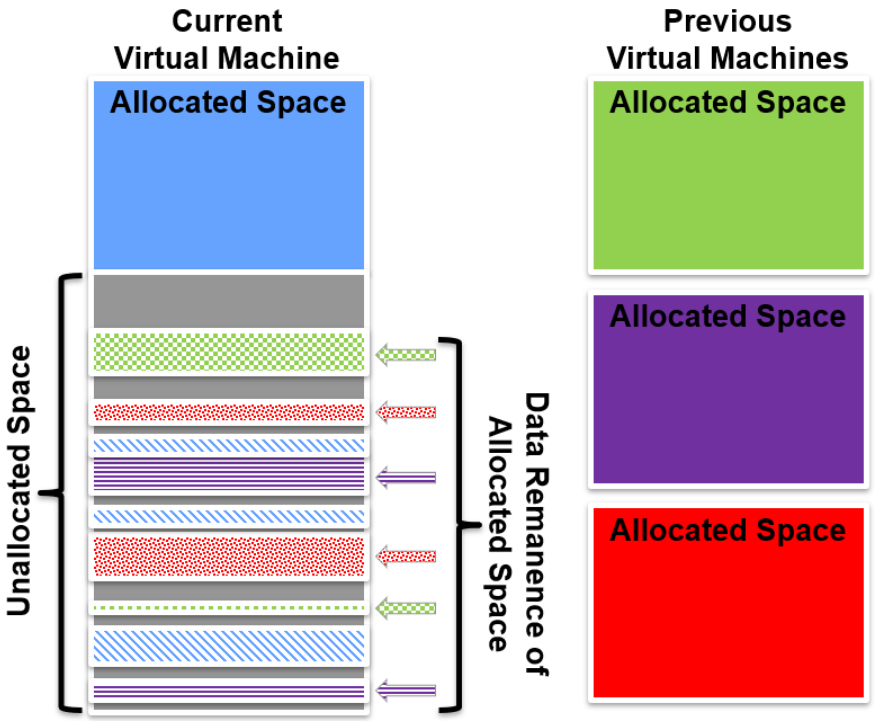


Figure 5 A Representation of Data Remanence of a Single VM In the Cloud.

As in the previous figures, the colored blocks represent the allocated spaces of the respective VMs. The left instance is seen to contain multiple pieces of other template instances found within the Cloud, as well as some pieces of its own. While finding fragments of its own template instance could be potential data remanence, it is not definitive and should be ignored. However, if unique sectors from any of the other template instance files are found within the unallocated space, then positive data

remanence has likely been discovered. Residual fragments of other template instances in unallocated space of the current VM are the basis of the methodology presented in this work.

3.3 Key Detection Concept

The underlying concept behind this method of determining data remanence is having known data to detect, where the known data is derived from previously downloaded VMs from the specific cloud provider being tested. Similar to Figure 5, Figure 6 depicts this key concept in how remanence is detected in a cloud provider's environment.

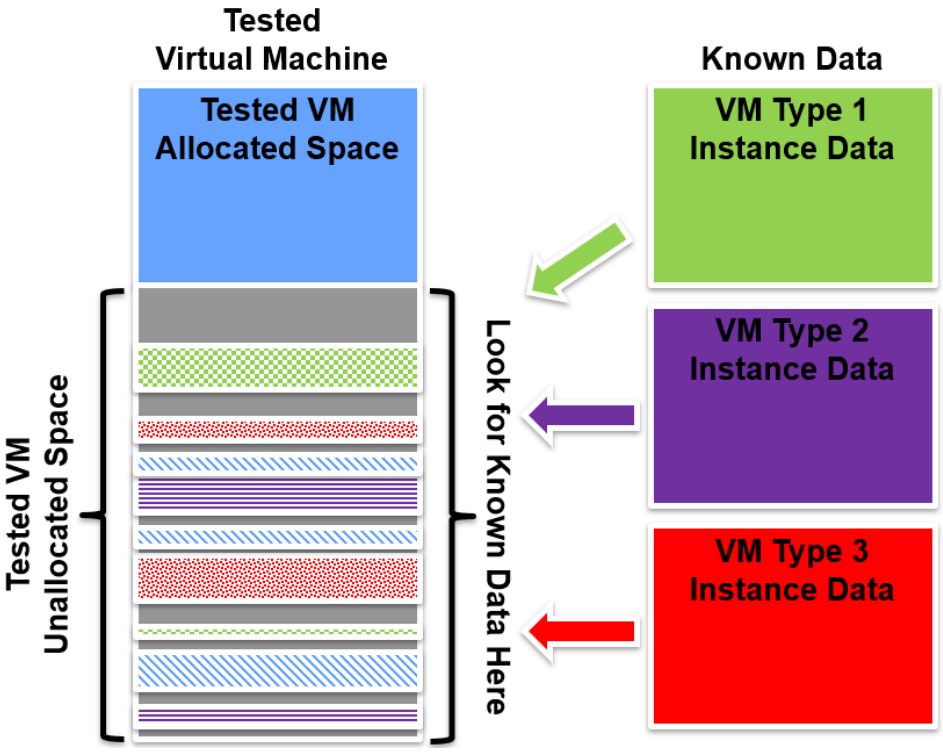


Figure 6 Depiction of Key Concept Where Known Data Is Used for Detection

Remanence detected in this manner that can be traced back to a known VM Type is Source Marked Remanence (SMR). Remanence detected with unknown origins is Source Unmarked Remanence (SUR). SUR is especially important since it is within this type of remanence that usernames, passwords, and other sensitive data could be found. For example, when using the methods described against a production cloud environment, the startup scripts used to initialize the virtual machines requested by users were discovered as SUR. Data remanence within the script included privileged account information, username and password, which could be used by an attacker to log into any machine provisioned by a major university's cloud environment. Appendix B shows the script that was extracted from the cloud provider with the username and password

intentionally post-obfuscated by the researcher. Appendix F.2 shows the program that was used to collect the virtual machines from the cloud environment.

3.4 Determining File Associated with SMR

All operating systems must maintain a detailed knowledge of where each file is located. This knowledge is used to reference the object for everything from opening a file, executing a program, or deleting an unwanted picture. Since SMR is derived from not only a known VM Type but also from a previously downloaded image, then the specific location where the SMR was found is known. With this information, it is theoretically possible to determine which file the SMR instantiated from utilizing the respective file system for the operating system from which the SMR derived.

For example, in most modern Windows operating systems utilizing New Technology File System (NTFS), a Master File Table (MFT) is utilized to keep track of every file in the system. It is essentially a database containing at least one record for each file and directory that maintains metadata including creation time, last access time, and critically, where the file is located physically on the drive. With this knowledge, it is a trivial task of performing a reverse lookup of which file is associated with a specific location on a drive. Similarly, for Linux based operating systems, the inode (index node) is used to keep track of where files are located on a drive along with various metadata. This reverse lookup is only possible with SMR detection and maintaining an image of the VM Type from which it was derived.

3.5 Contiguous Remanence Event

A Contiguous Remanence Event (CRE) is where a 512 byte chunk of remanence was discovered with at least one other remanence event immediately following. Figure 7 shows two notional separate CREs found in an experiment.

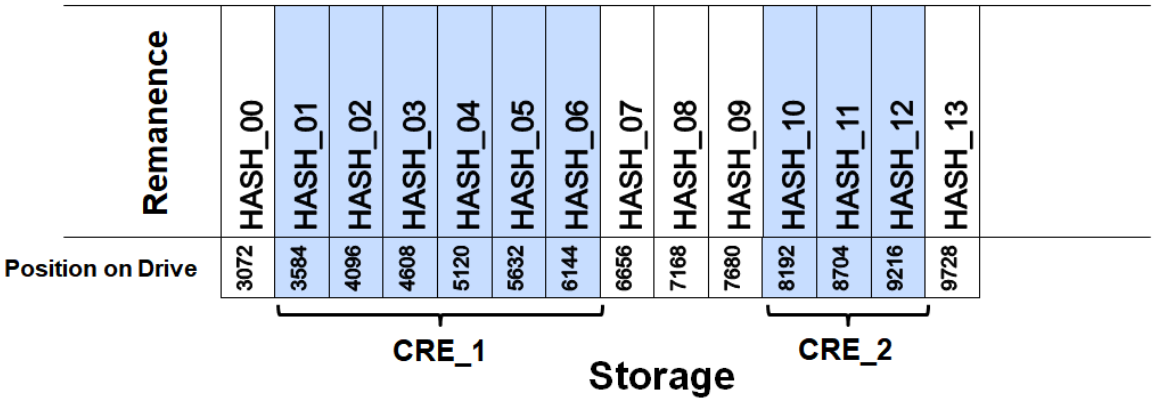


Figure 7 Graphical Depiction of Contiguous Remanence Events

CREs are a critical aspect in determining whether a cloud provider is successfully removing historical user data before reassigning memory to another user. If remanence is detected randomly, and non-consecutively, throughout the drive, then it is less likely that it originated from a file that was left behind. Therefore, determining the amount of CREs discovered during each run, and for each type of remanence (SMR or SUR) discovered, is central to the process. While the idea of CREs gives credence to discovery of non-random remanence, further investigation into the remanence and potential files associated with each CRE should be conducted.

3.5.1 Files Associated with a Single CRE

If the supposition is that a CRE is derived from a single file, then each single SMR event found within a CRE must contain the same associated file. Therefore, a simple test can be conducted to determine whether all associated files within a single CRE are the same. This idea of a single file for each CRE is illustrated in Figure 8.

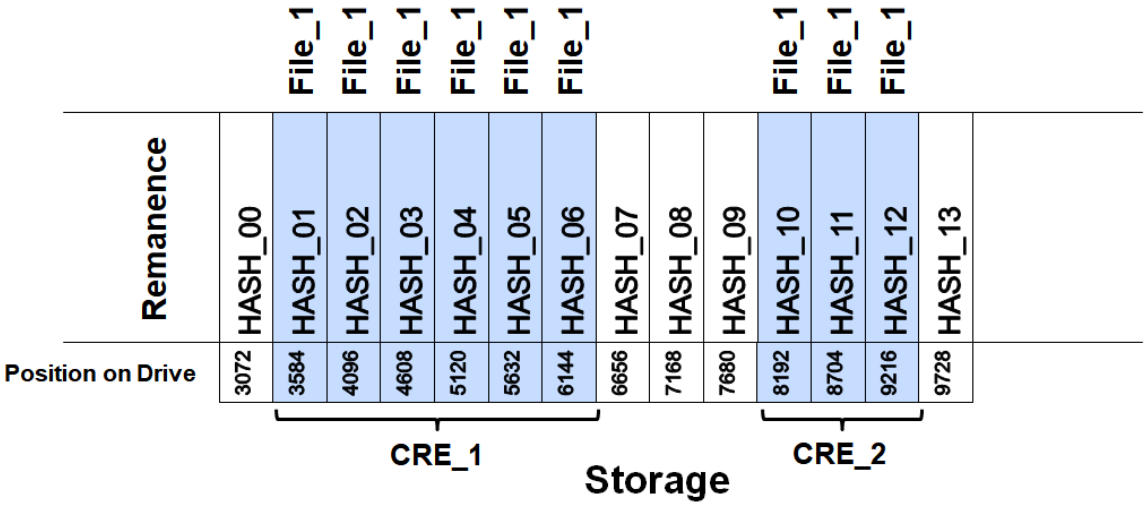


Figure 8 Contiguous Remanence Events Sharing the Same Associated File

Even with the associated file matching all SMR within a CRE, additional investigations must be made in order to definitely determine if the concept of a CRE holds significance for positively finding data remanence.

3.5.2 Sequential Order

While discovering whether all SMR associated files are the same within a CRE is essential in establishing the significance of discovered CREs within a cloud provider, it would not matter if the values found were out of sequence with the extracted associated

file. Therefore, the next test is to determine whether the CREs are in sequential order with the associated file by hashing the file 512 bytes at a time and then checking if each 512 byte remanence in each CRE is discovered in the same order as the file. Figure 9 is a notional depiction of the SMR discovered along with the associated file and the order in which the file was discovered.

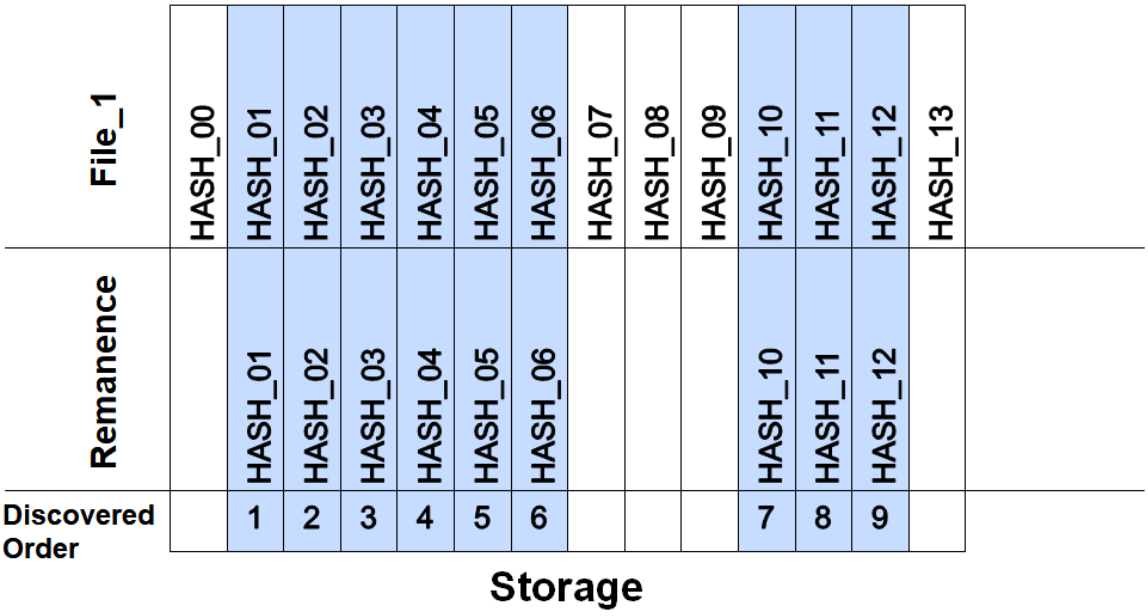


Figure 9 Sequential Discovery of Remanence in Respect to the Associated file

As seen in Figure 9, if the hash values are found in the correct order, even with gaps between the CREs, it increases the probability of the SMR originating from the associated file dramatically. With this information, even more inferences, discussed in subsequent paragraphs, can be made in regards to additional remanence of the associated file on the drive not detected as SMR.

3.5.3 Data on Drive

If the file was placed on the drive, in the same cluster contiguously with little fragmentation, it is possible to determine the sector offset of the start and end of the file on the drive where the remanence was found. This sector offset can be found by determining the lowest sector offset of all CREs associated with a single file, determining the offset of that hash value within the associated file, and subtracting those bytes from the sector offset to determine the start. The ending sector offset is the start offset plus the file size in the original drive's image. With the start and end offset, that part of the disk can be sector hashed and compared, sequentially, with the associated file. Figure 10 depicts the idea of detecting remanence by directly inspecting the drive and comparing the hash values discovered to that found within the SMRs associated file.

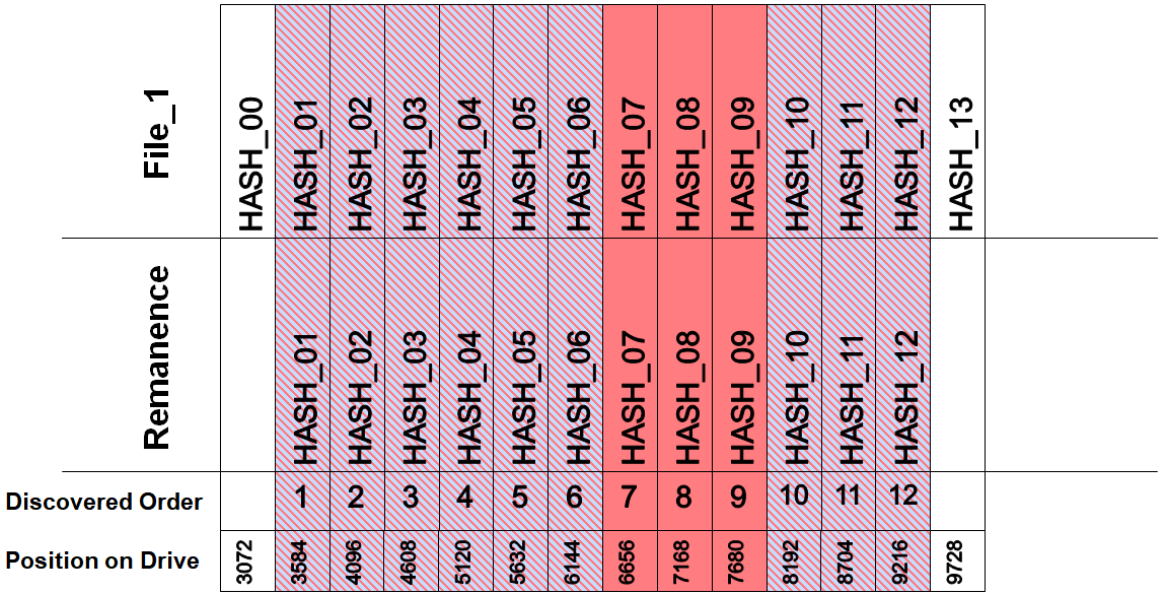


Figure 10 Discovering Additional Remanence through Direct Inspection of Drive

The hash values in red would not be initially detected as SMR, but discovered during the process of inspecting the drive directly. The hash values in blue and red stripes would be detected by both the SMR detection and direct drive inspection processes. Having the values detected by both processes is important because it strengthens the argument that the SMR is not random, but the actual remanence of the associated file.

3.5.4 Graphical Depiction of Recovered File

With the associated file extracted and hashed, as well as the respective CREs' hash values, it is possible to graphically show the specific locations in which the remanence was discovered with respect to the associated file. This graphical depiction is accomplished by matching hash values with the location of the hash in the associated file. Furthermore, a percentage of the file discovered can also be determined. Figure 11 is a notional representation of a graphical depiction of the remanence discovered as SMR and remanence discovered by inspecting the drive directly for each associated file.

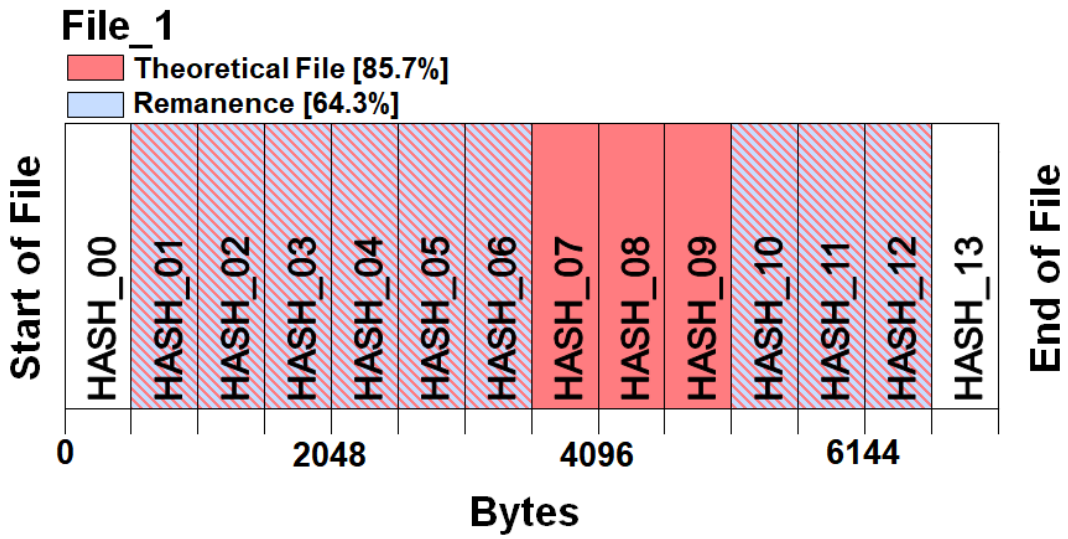


Figure 11 Notional Depiction of The Results of Graphing Remanence Discovered by Both Methods

Hash values in red are parts of the associated file that were not initially discovered by the remanence detection methodology. The figure's legend labels the red values as the "Theoretical File" since discovering these values are based on a number of theoretical assumptions that the file is still not fragmented and in the specific location on the drive. While hash values in blue were discovered previously as SMR. Theoretically, all SMR should also be discovered using the direct method as well, therefore blue and red stripes are used to reflect this state in the figure.

4 DATA REMANENCE DETECTION PROCESS

This chapter describes the method of detecting data remanence, and determining the operating system and the file to which the remanence belongs. The overall process encompasses three main steps: Setup, Detection, and File Recovery. All steps except the Setup portion are built into the developed data remanence detection tool.

4.1 Setup

The Setup step was developed for speed and efficiency. While the data remanence detection tool could perform all steps automatically, it is a reasonable assumption that the amount of changes between downloads of a specific VM Type within a short time span is extremely minimal; especially since most cloud providers mark when a VM Type's image has been altered by changing the reference number of that VM Type. Thus, performing these repetitive, bandwidth intensive and time-consuming functions prior to starting the detection process dramatically decreases detection time while maintaining the integrity of the tool.

4.1.1 Download VM Image and Hash

As noted in the previous chapter, the known data used to determine data remanence is derived from previously downloaded VMs from the cloud provider being evaluated. Therefore, the first step is to collect a number of representative images from the cloud provider and generate sector hashes of the allocated space. The unique sector hashes in these data sets are the basis of our remanence detection approach. This step is completed for each cloud provider at the beginning of the assessment process. It involves

selecting the VMs that are going to be potentially detected as well as which will be tested for remanence at a later date. A configuration file is created with a line for each selected VM that references the data set generated. Appendix F.4 shows the program used to generate all files referenced in the configuration file.

After selecting a cloud provider, a VM type (template), with included operating system, is chosen to test for data remanence. The operating systems can be of the same family, both Windows and Linux, but different versions of an OS are necessary in order to increase the likelihood of data remanence detection. It is recommended that at least some of the operating systems come from separate families. The process of downloading a virtual machine of the Tested VM Type (TVMT) and detecting remanence within the unallocated space is repeated as many times as desired.

4.1.2 Creating the Configuration File

While it is possible to perform all tasks dynamically, it can be time and computing power intensive. Therefore, a configuration file is created where each line signifies one of the VM types that have been selected from the cloud providers in the previous step. Each line is semi-colon delimited where each column references a file that was derived from the raw image of the downloaded virtual machine. Table 1 describes what each object in the configuration file represents.

Table 1 Description of Items Found within the Configuration File

Header	Example Value	Description
Tag	<blank>, !, or #	Used to designate whether the line is used in the process and for what purpose.
Directory	./Path/	The location of where the subsequent files are located.
Raw Image	FileName	The raw image file name.
Allocated Space	FileName_ALLOC	The raw allocated space of the image.
Unallocated Space	FileName_UNALLOC	The raw unallocated space of the image.
Allocated Hash Key File	[KEY]hash_key_FileName_ALLOC	MD5 hash list of the allocated space 512 bytes at a time.
Allocated Super Key File	[KEY]hash_key_super_FileName	Mapping of the sector hash to position in the image.
Unallocated Key File	[KEY]hash_key_FileName_UNALLOC	MD5 hash list of the unallocated space 512 bytes at a time.

The "Tag" header is used to provide greater flexibility with the detection tool. It allows the user to choose which VM types to detect by toggling lines "on" and "off" with a "#" symbol. If a line contains a "#" symbol as the first character, then it is completely ignored and that VM type is not used with the process. The lack of a "#" symbol is also used to designate which VM types are used by the tool and known as Other VM Types (OVMT). This designation is critical for the initial differential detection process used by the tool and described later. The VM type with the "!" tag before the line in the configuration file is considered to be the TVMT.

The Directory header is simply the path where all of the configuration files for that VM type are being kept. This method makes it possible to easily store various VM types in separate folders in an organized manner.

The Raw Image header is the raw image file taken of each operating system that includes both allocated and unallocated space. The method of collecting the image differs by cloud environment and VM type. These images are downloaded straight to the local machine's hard drive so that the Cloud's environment is altered as little as possible. It should be noted that further processing and calculations are performed on the downloaded images and not in the Cloud in order to keep the images free from possible contamination of the unallocated space.

The Allocated and Unallocated headers are the allocated and unallocated space, respectively, extracted from the raw image file. Allocated space within an image contains the operating system and any other active files within the VM. Unallocated space is the remaining portion of the image which does not contain any active files and to which data can be written freely.

The allocated space for each image is extracted from the raw image and then sector hashed, using MD5, at 512 byte intervals. Sector hashing increases the likelihood of finding data remanence since files are sector aligned and data for two different files should not occupy the same 512 byte chunk. The file created from this process is known as the Allocated Hash Key File. The same process is performed for the unallocated space for each VM, and is known as the Unallocated Hash Key File.

The Allocated Super Key File header is a byte location mapping of 512 bytes, sector aligned, MD5 hashes of each raw image file (Figure 12).

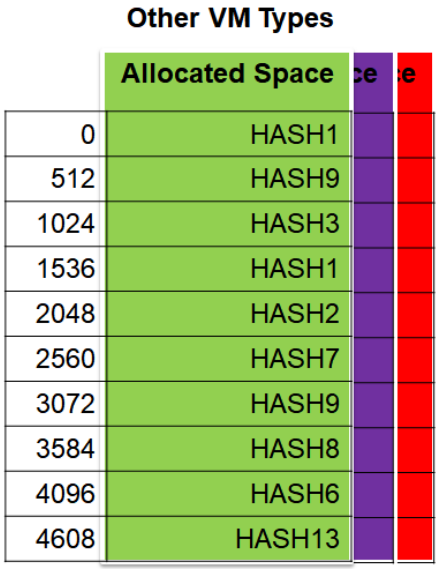


Figure 12 OVMT's Sector Hashed Allocated Space with the Associated Byte Offset

This dictionary of byte offsets mapping to respective hash values is central to file recovery when remanence is detected. Once the configuration file and all of the referenced files within it have been created, the next step in the process can begin: Detection.

4.2 Detection

The Detection step begins with collecting a VM image chosen from the list of previously collected VMs within the configuration file. This newly downloaded VM is the image that will be checked for remanence within its unallocated space. However, before testing for remanence can begin, differential hash key sets must be made in order to narrow down potential remanence. The key sets used to create this subset of hash values used for testing were created and collected during the generation of the configuration file.

The TVMT is the same VM type that was recently downloaded from the cloud provider where remanence may be found within its unallocated space. Each Allocated Hash Key File within the configuration file is used to create a set of hash keys. The hashes for the OVMTs are compared to the TVMT where values unique to the OVMTs are saved for later use (Figure 13). The unique values found in the OVMTs are the basis for potential SMR detection. Additionally, a combined unique set of hash values, comprising Allocated Hash Key Files within the configuration file, is saved to be used later for SUR detection.

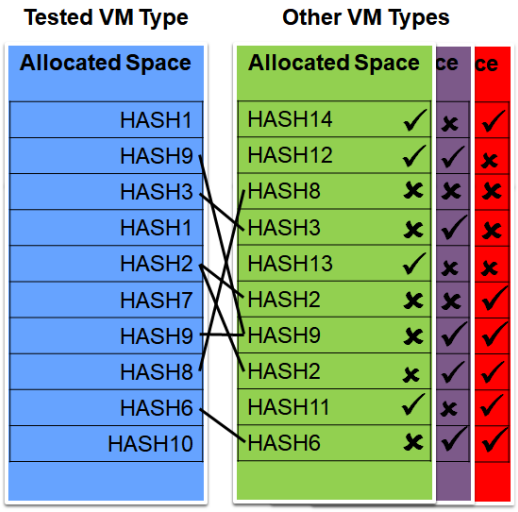


Figure 13 Determining Unique Hashes between the TVMT and OVMTs

At this point, the full images of the TVMT and OVMTs have been downloaded, a unique hash list of OVMTs have been created, and a hash dictionary mapping the byte offset to the 512 byte hashes found in the allocated space of the OVMTs have been

generated. With this information, the newly Downloaded VM (DVM), of the same VM type as the TVMT, can be tested to determine if it contains any data remanence from the OVMTs.

In order to pare down the amount of data tested for remanence by the tool, the unallocated space of the TVMT (previously collected, processed, and referenced in the configuration file) is utilized. Only unique hash values found in the DVM's unallocated space when compared to the TVMT's unallocated space are used in the bulk of the tool's processes. If there is no difference between the unallocated space of the DVM and TVMT, then remanence is not occurring. However, any differences found means that the potential for remanence exists and should be explored by the tool. By only utilizing the differences, the tool's efficiency is increased dramatically.

With these unique values known, the unallocated space is sector hashed 512 bytes at a time, known as chunks, in order to quickly determine the content. Before the chunks are compared to unique hash values, the program filters specific content. If the chunk is composed entirely of 1's or 0's, or is NULL, then it is ignored and the next chunk is ingested for inspection. The chunk is also ignored if the hash value is found within the unallocated space of the TVMT as discussed in the previous paragraph. Each hash is then checked against the unique values of the OVMTs found in the previous steps (Figure 14).

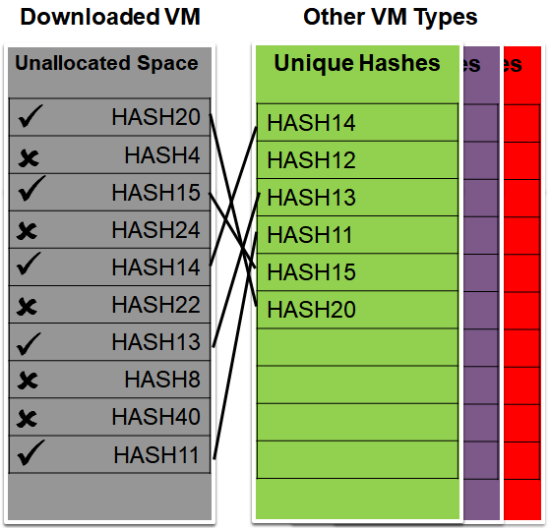


Figure 14 Unallocated Space of DVM Checked against OVMT's Unique Hash Lists

If a match is found, then potential data remanence has been discovered. In order to positively identify data remanence, further testing is required.

After flagging each matching hash value, the raw 512 byte chunk of data associated with the hash is extracted from the unallocated space of the DVM. For each extracted chunk, the allocated space of the DVM is checked to see if it contains the string of data; whether sector-aligned or not (Figure 15).

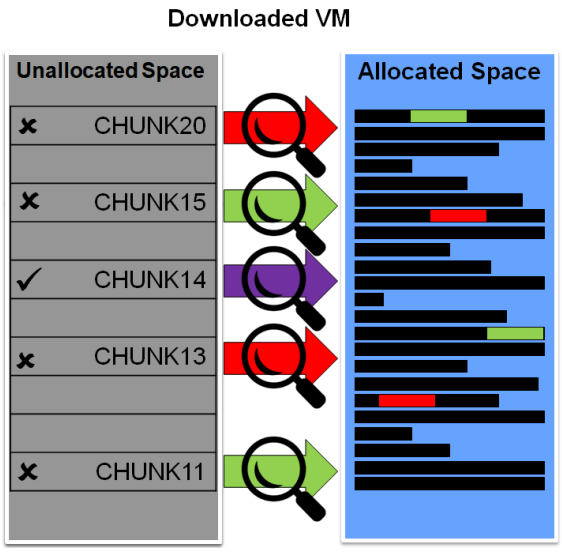


Figure 15 The Allocated Space Is Checked for the 512 Byte String of Potential Data Remanence

If the 512 byte chunk is found within the allocated space, then it is discarded as unverifiable data remanence. However, if the string is not found within the allocated space of the DVM, then this string is data that should not be within the VM, and positive data remanence has been discovered. Once data remanence has been confirmed, and attributed to a specific VM and hash value, this attribution can be utilized to recover the file to which it belongs.

4.3 File Recovery

The File Recovery process utilizes a combination of the files generated during the Setup phase, and locations derived during the Detection phase. The hash values listed as confirmed data remanence are compared to the unique hash value list for the OVMT to which it is supposedly associated. If it is within the list, then the hash values are located

within the hash dictionary for the same OVMT. Once found, the byte offset location within the OVMT image is known. (Figure 16)

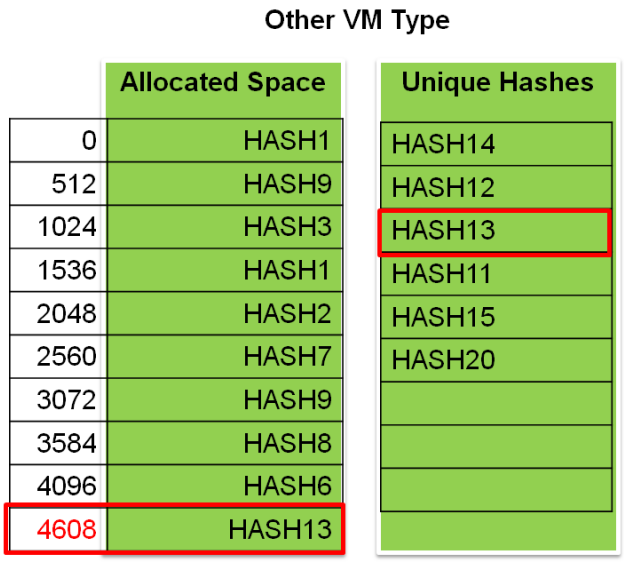


Figure 16 Determining the Byte Offset Location within the OVMT

With the byte offset known, a forensic toolset known as The Sleuth Kit (Carrier, n.d.) can be utilized to determine and extract the file where the remanence, probably, originated. The first step is determining the cluster size of the image in question by utilizing the fsstat tool. Once the cluster size is known, ifind is utilized to determine the file ID for Windows, or inode for Linux, that is associated with the byte offset provided for the image. To determine the files associated with the result of ifind, ffind is called. The results of ffind provides a full path listing to each file linked to the file ID or inode. For each file listed, fcat is called to pull those files out of the image and into a separate folder. Each file is then searched for the raw bytes associated with hash of the data

remanence found. If the string is found within the file, then the data remanence origination file has been located and verified.

5 DATA REMANENCE DETECTION TOOL

Due to the specific approach for determining cloud data remanence, new tools were developed to implement the approach presented here. The automated system for determining cloud data remanence was programed using Python 2.7 (Python Software Foundation, n.d.). Python is currently the programming language of choice for digital forensics and has many of the modules required to develop the tool necessary to test the methodology. Appendix F.5 shows the program developed for the data remanence detection tool.

A few Python modules of critical importance are Boto (Amazon Web Services, n.d.), Selenium (Barantsev, n.d.), and Subprocess. The Boto module is used to connect to the AWS Application Programming Interface (API) and perform a variety of tasks in an automated fashion. It is used to create, terminate, and delete volumes for each VM tested.

The Sub-process module is used to call a number of executables that are not found within the Python module libraries. Most notably are Plink (Tatham, n.d.), DD, and TSK tools. Plink is used to automate SSH commands while DD is used to create raw image files of the VMs. TSK executables, discussed in the File Recovery section, are used to extract the files to which the located data remanence belongs.

5.1 Data Remanence Detection Tool Validation Testing

The data remanence detection process, and resultant tool based on the process, require validation that it performs as expected. Since the tool implements the process,

validation of the tool implicitly validates the process. The tool was validated using two distinct methods: Artificial Remanence and Local Cloud Environment. The Artificial Remanence method placed known amounts of remanence within the unallocated space of the simulated DVM and attempted to detect all traces of remanence injected in the image. The Local Cloud Environment mimicked a cloud provider's hypervisor by intentionally generating remanence within an instance of VMware's ESXi suite and detecting remanence (VMware, 2018). For each method, representative Windows- and Linux-based operating systems were used -- which were Windows 10 and Windows 7, and Ubuntu 17 and Fedora 19, respectively.

5.1.1 Validity Test 1 – Artificial Remanence

To ensure that the tool is detecting data remanence, artificially placed remanence is inserted into various operating systems which are then tested to find the stamped data. The process tests both the accuracy and validity of the detection model and the program that executes it. This test is accomplished by first instantiating a clean representative operating system within a hypervisor for each virtual machine used during the test.

5.1.1.1 VMware Workstation 12

VMware Workstation 12, a Type 2 Hypervisor (Silberschatz, Galvin, & Gagne, 2012), is used to create the virtual machines for each operating system. The stamp file is of no type with the first 512 bytes containing a repeating pattern that is unique to each operating system. Each image was searched for the MD5 hash of the 512 byte chunk of remanence to ensure the values did not exist in any of the images. Each file was placed on the Desktop of their respective operating system's virtual machines. Table 2 describes

each operating system and their respective stamp file attributes created using the program shown in Appendix F.1.

Table 2 Operating Systems Used for Validating the Developed Tool

Operating System	Repeated Hex Value	Stamp File Name	Stamp File Size (MB)
Windows 10	8765432187654321…	win10stamp_DISSER	2
Windows 7	1234567812345678…	win7stamp_DISSER	1
Ubuntu 17	1122334455667788…	ubuntu64stamp_DISSER	3
Fedora 19	1234567887654321…	fedora19stamp_DISSER	5

An image of each virtual machine is then collected and stored to be used with the tool's comparison algorithms described in the Data Remanence Detection Process section. This image represents the initial baseline image that would be pulled from a cloud provider; the TVMT and OVMTs. A duplicate copy of each image is made to represent virtual machines with remanence within the unallocated space of the image.

The artificial remanence matches the data in each file placed on the virtual machine's desktop. An additional type of remanence data, not associated with any operating system, is also placed within the unallocated space of each operating system representing a previous user's data, or even startup information; also known as SUR. For example, the Windows 10 operating system would contain artificial remanence from the Windows 7, Ubuntu 17, and Fedora 19 images, as well as remanence not associated with any image. Different stamped data was used for each operating system remanence to keep track from which operating system the remanence originated; also known as SMR. Table 3 displays the remanence data type with their matching stamped remanence data.

Table 3 Operating Systems Used for Validating the Developed Tool with Their Corresponding Hash Values

Remanence Data Type	Repeated Hex Value	MD5 of 512 Byte Chunk
Windows 10	8765432187654321…	30D2DFA2A144A5534BC16D7056BFEAD5
Windows 7	1234567812345678…	67C44D63E0F50A85E81C58EC942CD300
Ubuntu 17	1122334455667788…	FB69067AA76E261791F74A760044C3BA
Fedora 19	1234567887654321…	591215D50D9423BD884BAA0617340F3B
User Data	8877665544332211…	E604533E9E73F67AFC98354B02420BED

For each image, a known amount of different types of stamped remanence data, unique to each operating system, is placed in various locations of each duplicate image's unallocated space using a hex editing tool. The placed remanence is a repeating value that is 512 bytes, sector-aligned, in length at a time. Table 4 shows the number of 512 byte chunks of remanence placed into the unallocated space of each operating system along with the data type.

Table 4 The Amount of Artificial Remanence Placed within the Respective Operating Systems

Operating System	Number of 512 Chunks of Remanence Data Type				
	Windows 10	Windows 7	Ubuntu 17	Fedora 19	User Data
Windows 10	7	15	10	6	5
Windows 7	7	9	15	5	5
Ubuntu 17	8	12	20	5	5
Fedora 19	6	15	10	7	5

This duplicate copy, containing the artificial remanence in the unallocated space, represents an image collected from the cloud provider during testing; the DVM. Furthermore, for my experiments, it is just as important to keep record of the location

where the artificial remanence was placed within the unallocated space. As with the remanence amount, the location placed are also different for each operating system. For example, Table 5 shows the location of remanence for the Windows 10 image.

Table 5 Location of Artificial Remanence Placed in Windows 10 Image

Remanence Data Type	Locations/Offsets In Image (GB)
Windows 10	12.44420096,17.969316352,17.969318912,17.969321472,17.969324032,17.969327616,17.969329152
Windows 7	17.96931584,17.969316864,17.969317376,17.969317888,17.969319936,17.969320448,17.96932096,17.969321984,17.969322496,17.969323008,17.969324544,17.969325056,17.96932608,17.969326592,17.969327104
Ubuntu 17	17.969319424,17.969325568,17.969328128,17.96932864,17.969329664,17.969330688,17.9693312,17.969331712,17.969332224,17.969333248
Fedora 19	17.969335296,17.969335808,17.969336832,17.969337344,17.969338368,17.96933888
User Data	17.9693184,17.96932352,17.969330176,17.969332736,17.969334272

If working properly, the tool should be able to locate all the stamped data artificially placed throughout the operating system's unallocated space, excluding the remanence data type matching the image's operating system, which verifies its capability to locate differential operating system data. It should also accurately determine the location the remanence was found within the unallocated space. After verifying detection of artificial remanence, the tool can be tested on a locally created cloud environment.

5.1.1.2 Validity Test 1: Artificial Remanence - Results

To determine the accuracy of the Data Remanence Detection Tool, artificial remanence was placed in strategic and known locations on the virtual machines being tested. Therefore, it is essential for the output of the tool to keep track of not only the amount of remanence discovered, but the raw data and location on the virtual machine.

With this in mind, a remanence file is created which contains all remanence discovered during the experiment or run. As discussed previously, remanence is discovered 512 bytes at a time and must be logged and documented individually. Each discovered 512 byte chunk of artificial remanence generates a section within a remanence file.

Each section describes the image from which the remanence originated, the location in bytes (offset) where the remanence was found within the unallocated space, the MD5 hash, and whether the raw byte string could be found within the allocated space of the DVM. Once potential remanence reaches this stage of the validation, the 512 byte chunk has already been found to not be within the allocated space of the DVM or the unallocated space of the TVMT. A "PASS" means that the raw byte string was found within at least one of the files recovered, while a "FAIL" means otherwise and remanence could not be verified. The output of the tool also displays the recovered file where the remanence is believed to have originated and whether the raw byte string of the remanence could be found within the file. A "PASS" value before the recovered file means that the raw bytes were found within the file. There could potentially be multiple files recovered for each section with each file recovered checked for remanence data. Finally, the raw byte string, in ASCII, is displayed to be easily read. A sample section of a remanence file, containing SMR information, from the artificial remanence test is shown in Figure 17.

```
*****************************REMANENCE INFORMATION****************************
Image: ./_stampTest_DISSER/win10stamp_DISSER/   Offset:6386782720
Hash: 30d2dfa2a144a5534bc16d7056bfead5  Verified:PASS
******************************************************************************
2   PASS    /Users/brad/Desktop/stamp_win10
******************************BEGIN OF REMANENCE******************************
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
*******************************END OF REMANENCE*******************************
```

Figure 17 A Sample Section of the Generated Remanence File

This sample section of a SMR file comes from testing a Windows 7 VM (DVM) that was intentionally injected with a unique 512 byte sequence of data that matches a file that resides only on a Windows 10 machine (OVMT). Another Windows 7 machine that did not contain the unique identifier represented the TVMT. Therefore, in this test, there would be as many sections generated within the remanence file as there are 512 bytes of artificial remanence injected. Using this section as an example, the following can be determined from the various values discovered:

- Image: The associated file derived from a Windows 10 VMT
- Hash: The MD5 hash of the 512 byte chunk is 30D2DFA2A144A5534BC16D7056BFEAD5
- Offset: The SMR was discovered at byte offset 6386782720 in the Unallocated file
- Verified: The 512 byte chunk was successfully discovered in the recovered associated file

- File: The associated file within the Windows 10 image was found at location /Users/brad/Desktop/stamp_win10
- Data of Interest: The raw data, in ASCII, shows a specific pattern

All values found within the section of remanence were compared to what was expected within the Windows 7 machine down to the exact location and hash value. While testing SMR is important, SUR is equally significant.

A separate file, known as the SUR file, is generated at the same time the SMR file is created in the process. All remanence that is not discovered within any of the other OVMT or the TVMT are cataloged within this file. Along the same lines as the SMR file, each distinct remanence discovered is placed into its own individual section. **Figure 18** shows a sample section of the SUR file for the artificial remanence test.

```
***************************DATA OF INTEREST INFORMATION***********************
Offset:6386780160   Hash: e604533e9e73f67afc98354b02420bed
***************************BEGIN DATA OF INTEREST*****************************
^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3
"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^
wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"
DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^w
fUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"
DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^w
fUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"
DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1^w
fUD3"DC1^wfUD3"DC1^wfUD3"DC1^wfUD3"DC1
****************************END DATA OF INTEREST******************************
```

Figure 18 A Sample Section of the Generated SUR File

Similar to the SMR file, the sections within the SUR file display the offset and MD5 hash of the SUR found. Since it is unknown from where the data came, no other

information is given except for the raw bytes. However, the raw bytes can be critical to discovering remanence that is user specific and not from a loading script or other potentially innocuous source. Therefore, another file is created that contains only the raw bytes discovered in the SUR file. This file can be easily searched for user information such as usernames, passwords, phone numbers, credit cards, etc.

While the remanence files are formatted and organized so that it is relatively easy for a human to interpret each section independently, it can be cumbersome to parse the files by hand. With this in mind, another program was developed specifically to read, compile, and display the content within the remanence files created by the Data Remanence Detection tool; the Report Generator (Appendix F.6). With this tool, and the known values for each remanence data type, one can easily determine the accuracy and validity under these specific conditions. Using the Data Remanence Detection Tool on virtual machines injected with artificial remanence yielded the results shown in Table 6, organized by examined operating system.

Table 6 Results of Windows 10 Virtual Machine with Artificially Injected Remanence

Windows 10			
Remanence Data Type	**Chunks Injected**	**Chunks Found**	**Percentage Found (%)**
Windows 10	7	0	0
Windows 7	15	15	100
Ubuntu 17	10	10	100
Fedora 19	6	6	100
User Data	5	5	100

All SMR is represented by the rows containing Windows 10, Windows 7, Ubuntu 17, and Fedora 19, while the SUR is represented by the User Data row. Upon first inspection, one would be concerned that the Data Remanence Detection Tool did not report any Windows 10 remanence even though seven chunks of remanence were injected. However, the tool is working as designed and expected. To ensure a positive assertion of remanence, the Data Remanence Detection Tool ignores all remanence it finds associated with the same operating system as that being tested. Therefore, while all other SMR and SUR were detected at a rate of 100% tracing back to the appropriate Virtual Machine Type (VMT) and associated file, the Windows 10 SMR was not reported as discovered, but rather ignored.

It is not enough to determine whether the remanence was discovered, but also to ensure that it was discovered at the appropriate offset. A unique feature of the Report Generator program developed is its ability to visually display the remanence discovered, including location on the drive for the virtual machine being tested. Figure 19 shows a graphical representation of the SMR and SUR discovered in the Windows 10 virtual machine injected with multiple artificial remnants from other VMTs.

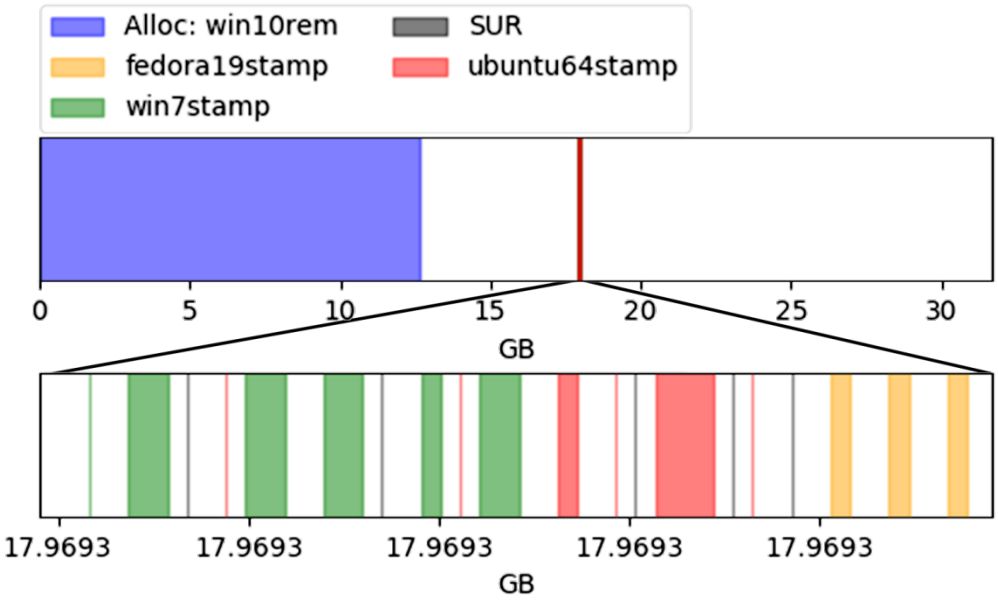


Figure 19 Visualization of Remanence Found Within Windows 10 Image

As can be seen, the artificial SMR and SUR placed within the image is contained within a very small area, therefore the specific location where remanence was discovered is zoomed to easily visualize the remanence. It was also spread throughout so that there were multiple CREs and non-CREs alike. Furthermore, with the offset for each remanence location displayed within the SMR and SUR files, and the known locations within the tested virtual machine, it was determined that all remanence was discovered at the correct positions on the drive.

The outcome of testing the Windows 7 virtual machine with artificial remanence yields similar results. Table 7 describes the aggregation of remanence injected as well as discovered for each VMT in the experiment. See Appendix A.1 for the raw output from the Data Remanence Detection Tool.

Table 7 Results of Windows 7 Virtual Machine with Artificially Injected Remanence

Windows 7			
Remanence Data Type	**Chunks Injected**	**Chunks Found**	**Percentage Found (%)**
Windows 10	7	7	100
Windows 7	9	0	0
Ubuntu 17	15	15	100
Fedora 19	5	5	100
User Data	5	5	100

All SMR and SUR were discovered at a rate of 100%, except for Windows 7. As discussed previously, this is working as designed and yielding results as expected. Furthermore, all VMTs and associated files were found to match correctly. A graphical depiction of the SMR and SUR is shown in Figure 20.

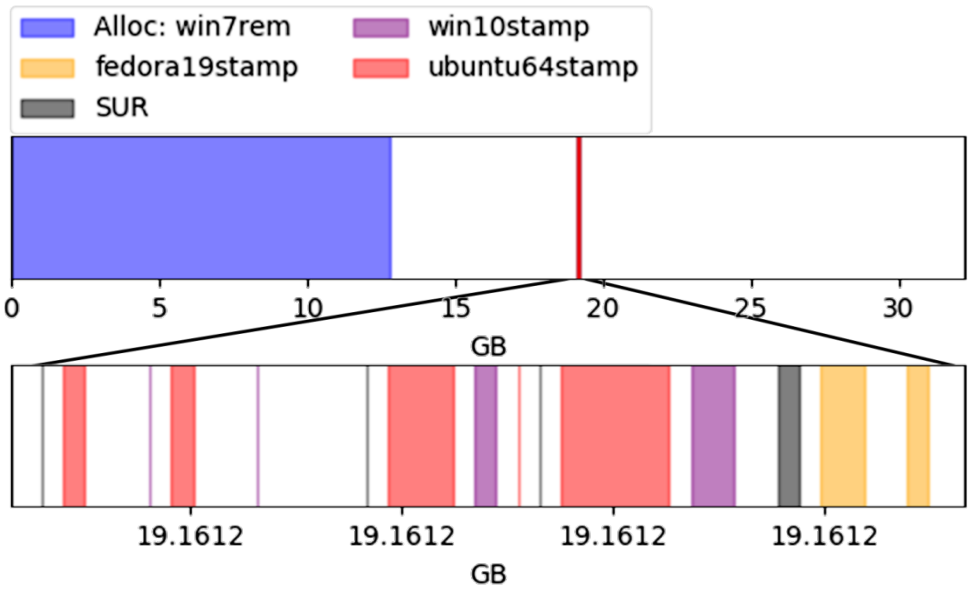


Figure 20 Remanence Found Within Windows 7 Image

As with the Windows 10 run, remanence was randomly placed in a small area on the drive with varied amounts of CREs. It should be noted that for each CRE, different locations were chosen to ensure that no matter where remanence was placed within the unallocated space, the Data Remanence Detection Tool would discover it accurately. All SMR and SUR were also discovered at the correct locations on the drive. See Appendix A.2 for the raw output from the Data Remanence Detection Tool.

The Ubuntu 17 virtual machine contained the most artificial remnants injected and, as expected, did not have any discovered by the Data Remanence Detection Tool that was traced back to the same VMT. All other remanence was discovered with 100% accuracy. Table 8 shows the results for artificially injected remanence.

Table 8 Results of Ubuntu 17 Virtual Machine with Artificially Injected Remanence

	Ubuntu 17		
Remanence Data Type	**Chunks Injected**	**Chunks Found**	**Percentage Found (%)**
Windows 10	8	8	100
Windows 7	12	12	100
Ubuntu 17	20	0	0
Fedora 19	5	5	100
User Data	5	5	100

When placing artificial remanence within the unallocated space of the Ubuntu 17 virtual machine, the location of the remanence was changed drastically from the Windows 10 and Windows 7 locations. As with the other experiments, this change was to ensure there was no location bias in the Data Remanence Detection Tool. Figure 21 shows the remanence found within the Ubuntu 17 image.

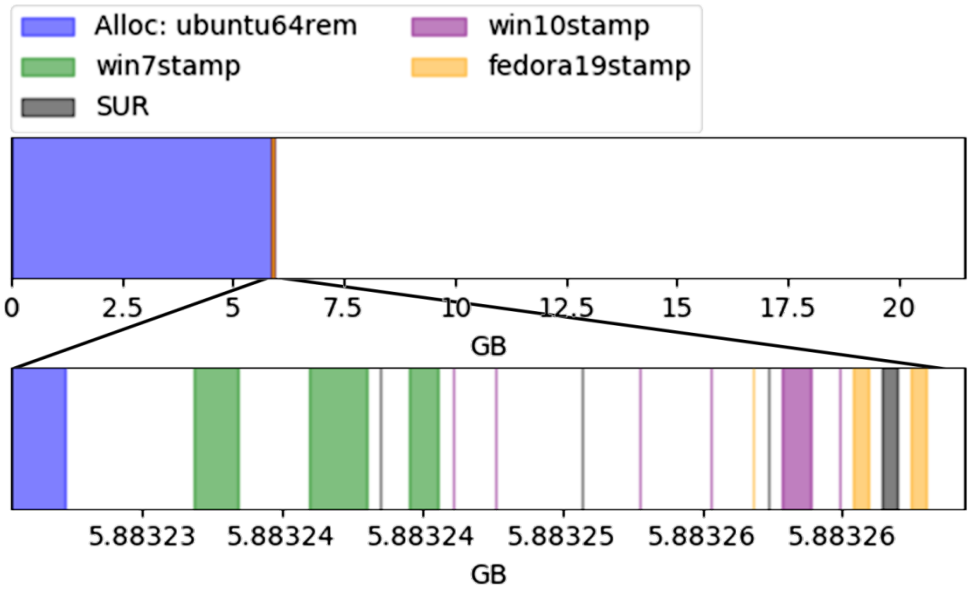


Figure 21 Remanence Found Within Ubuntu 17 Image

Even with the remanence placed in a radically different location, nearly sequential to the allocated space on the drive, all remanence was discovered in the correct location. See Appendix A.3 for the raw output from the Data Remanence Detection Tool.

The Fedora 19 virtual machine was the last of those in the test used for artificial remanence. All SMR and SUR were discovered with 100% accuracy with the Fedora 19 remembrance being ignored as expected including the respective remanence data type and associated files. Table 9 shows the results for the Fedora 19 VM.

Table 9 Results of Fedora 19 Virtual Machine with Artificially Injected Remanence

Fedora 19			
Remanence Data Type	**Chunks Injected**	**Chunks Found**	**Percentage Found (%)**
Windows 10	6	6	100
Windows 7	15	15	100
Ubuntu 17	10	10	100
Fedora 19	7	0	0
User Data	5	5	100

Along the same thought process as the Ubuntu 17 VMT, the remanence was placed near the end of allocated space in varied amounts and sizes of CREs. Figure 22 shows the distribution of remanence for Fedora 19 images.

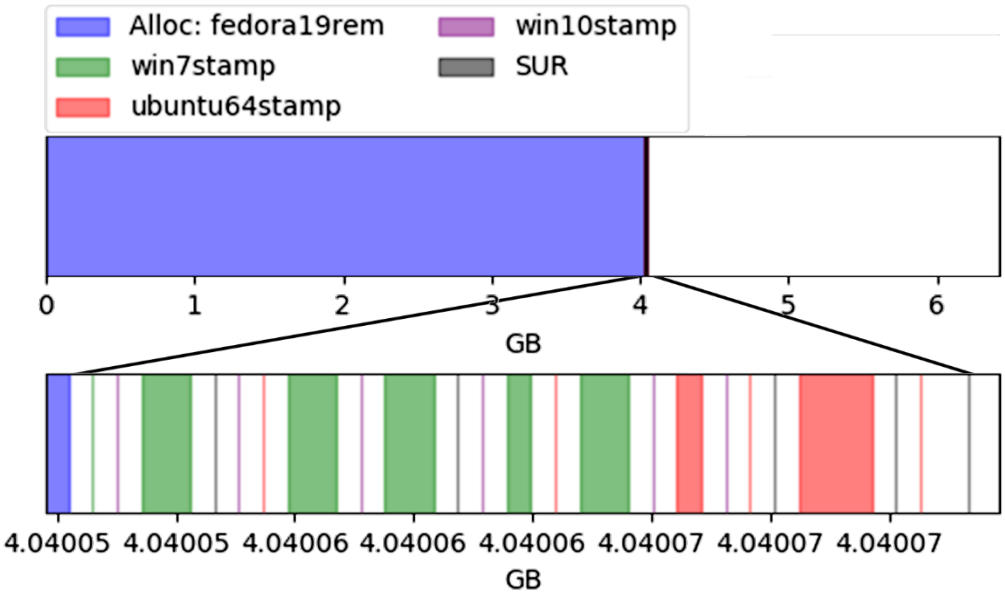


Figure 22 Remanence Found Within Fedora 19 Image

Each SMR and SUR graphed was mapped to the correct location of the original injected remanence. See Appendix A.4 for the raw output from the Data Remanence Detection Tool.

In summary, all remanence data types were accurately traced to the originating remanence file and the offset in the raw image, and then verified that the raw 512 byte remanence chunk was found within the associated file.

5.1.2 Validity Test 2 – Local Cloud Environment

Before testing the data remanence detection tool on real world cloud environments, a local test environment was created. The environment was used to simulate the various, potential, cloud environments that the tool would test for data remanence. While there are many different ways by which to deploy a local cloud environment, only one was necessary to validate the developed tool, as long as remanence could be generated in a controlled environment. It would be possible to emulate the remanence within VMware Workstation 12, but the Workstation line of VMware is typically only used for testing purposes and not for a full cloud installation. Furthermore, the methods used to create the remanence, for example, sharing virtual disks, would be performed out-of-band and not using the native tools provided within the software suite. Therefore, a different software package was required to perform this stage of testing. Keeping with VMware, the vSphere line was utilized to create the local cloud environment.

5.1.2.1 VMware vSphere 6.5

VMWare is one of the most popular hypervisors used throughout the virtualization industry. It is feature rich and allows for a multitude of virtual disk types. VMware vSphere, the localized virtualization suite used for many private cloud environments, is generally composed of two parts: ESXi (VMware, 2018) and vCenter Server (VMware, 2018). VMware ESXi is a bare-metal hypervisor, also known as a type 1 Hypervisor (Silberschatz, Galvin, & Gagne, 2012), since it runs directly on the host's hardware and also manages the guest operating systems. VMware's vCenter Server runs on a separate piece of hardware and is used to interface with the ESXi server to manage all the guest operating systems, data stores, and general maintenance of the ESXi server. Two new virtual machines are created within VMware Workstation 12; ESXi 6.5 U1 and Windows Server 2012 R2, with vCenter Server 6.5 U1. While this implementation is utilizing nested virtualization, it should not impact the experiment.

While many of the tasks can be performed using the graphical user interface provided by vCenter Server, connected to ESXi, some operations can only be completed directly through the ESXi command line. Typically, these actions are performed using the command set found within *vmkfsktools* (VMware, 2017). All actions performed to create the data remanence to be detected later were completed by using *vmkfsktools* commands.

The type of virtual disk utilized by the virtual machine is critical as to which will create remanence. Therefore, it is important to understand the various types of disks that can be used by the chosen hypervisor, which creates an understanding as to what actions

would generally create remanence on the disk. Table 10 lists the supported virtual disk formats with a brief description (VMware, 2016).

Table 10 List of Supported Disk Formats Utilized by VmWare's vCenter Server Suite

Disk Format	Description
zeroedthick (default)	Space required for the virtual disk is allocated during creation. Any data remaining on the physical device is not erased during creation, but is zeroed out on demand at a later time on first write from the virtual machine. The virtual machine does not read stale data from disk.
eagerzeroedthick	Space required for the virtual disk is allocated at creation time. In contrast to zeroedthick format, the data remaining on the physical device is zeroed out during creation. It might take much longer to create disks in this format than to create other types of disks.
thin	Thin-provisioned virtual disk. Unlike with the thick format, space required for the virtual disk is not allocated during creation, but is supplied, zeroed out, on demand at a later time.
rdm	Virtual compatibility mode raw disk mapping.
rdmp	Physical compatibility mode (pass-through) raw disk mapping.
2gbsparse	A sparse disk with 2 GB maximum extent size. You can use disks in this format with other VMware products, however, you cannot power on sparse disk on an ESX host unless you first re-import the disk with vmkfstools in a compatible format, such as thick or thin.

These descriptions provide the basis for the assumptions as to which disk format would, theoretically, leave data remanence and which would not. Based on Table 10, *zeroedthick*, the default disk format for VMWare, should remove any data left on the medium at the time of creation. Therefore, the *zeroedthick* disk format will be used to provide the image baseline used for the experiment; the TVMT and OVMTs. The next

two formats, *eagerzeroedthick* and *thin*, would not create remanence since they zero disk space when initiated and on demand respectively. However, the *rdm* and *rdmp* formats most likely will produce data remanence because they both use a form of raw disk mapping for the virtual disk. This mapping means that the disk itself is shared between the current virtual machine and any virtual machine that utilized the disk before it. Therefore, if the entire medium, not just the size presented to the virtual machine, is not properly sanitized before reuse, then remanence from previous virtual machines could be recovered. Raw disk mapping is also commonly used throughout production virtualized environment for the format's speed and efficiency, as well as some software limitations that require a raw disk. With this theory in mind and that this type of virtual disk format would be encountered in the wild, *rdm* was chosen to be utilized for the Local Cloud Environment experiment.

The same operating systems used in the previous validity test with artificial remanence were used with testing a local cloud environment: Windows 10, Windows 7, Ubuntu 17, and Fedora 19. These selections were used for the same reasons in that they are different operating system types making it easier for the developed tool to determine if remanence occurs.

Traditional virtual machines were made of each operating system with standard *zeroedthick* vmdk drives and stored within the local ESXi datastore. As conducted in the artificial remanence validity test, an image of each newly created virtual machine is then collected and stored to be used with the tool's comparison algorithms. These collected

images represent the baseline virtual machine of a cloud provider; the TVMT and OVMTs.

It is critical that only tools native to ESXi are used throughout the experiment and no out-of-band processes are employed to create or detect remanence. Therefore, in order to deploy the virtual machines to an *rdm* disk, the *vmkfsktools* command was used. The generic command used for each virtual machine is as follows.

```
Vmkfstools –I srcfile –d rdm:/vmfs/devices/disks/identifier /vmfs/volumes/datastore/vmdir/vmname.vmdk
```

The command takes a vmdk source file and clones the virtual machine to an *rdm* mapped virtual machine that directly accesses the chosen drive. The virtual machines with their respective disk size are shown in Table 11.

Table 11 The Operating Systems' Disk Size and The Order Converted To An *rdm* Mapped Disk

Operating System	Disk Size (GB)	Order Cloned
Windows 10	30	1
Windows 7	30	2
Ubuntu 17	20	3
Fedora 19	8	4

In order to potentially maximize the chance of remanence within the drive, the cloning process was performed in order of largest to smallest virtual disk size. Employing this sequence ensures that the least amount of data possible is overwritten during the cloning process performed by the *vmkfsktools* command. Once each cloning process is complete, an image of the entire physical medium is taken for later inspection. After the last operating system, Fedora 19, is cloned to an *rdm* virtual drive, the newly created virtual machine is run with the local disks inspected for what is available for

copying via a dd process. Figure 23 depicts the results of a *parted* command via an SSH session with the Fedora 19 virtual machine.

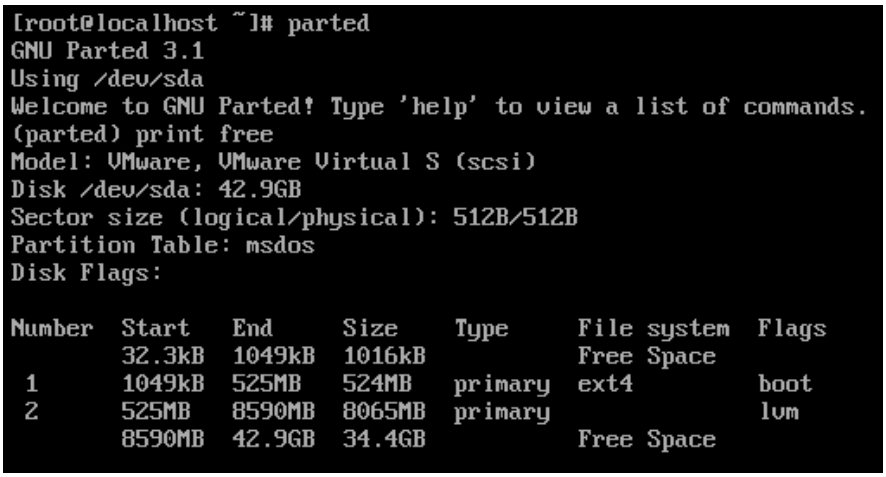


Figure 23 Results of the Parted Command Showing Size of Free Space

It is important to notice that there is 34.4 GB of unallocated space within the sda disk. This space would, theoretically, possibly contain remanence from the previous three operating systems installed on the *rdm* mapped disk as well as the unallocated space of the operating system itself. However, being a Linux based operating system, unallocated space within the operating system is generally zeroed. With this information, the virtual machine is accessed via an SSH session exactly like a user would access a virtual machine provisioned by a cloud provider. The SSH command line tool Plink (Tatham, n.d.), from the same developers as Putty, is called using an Administrator Command prompt to create the full disk image of the Fedora 19 virtual machine with the command shown below.

cmd /c echo yes | .\plink.exe -ssh -pw PASSWORD USERNAME@IP_ADDRESS sudo "dd if=/dev/sda | gzip -9 -" | dd of=directory/image_name

This single command combines a number of functions into a single line. The command calls the Plink executable to create an SSH session, with username and password credentials, to the Fedora 19 IP address. Once the SSH session is established, the dd command is executed with super user (su) privileges, piping all contents of the sda drive to gzip which compresses the information before sending the "zipped" contents to a file on the local drive. It is important that the files are not being generated on the virtual machine, but rather on the local computer so as not to alter any of the original data.

With the Gzip compressed image file transferred to the local computer, the entire image needs to be uncompressed in order to extract the parts of the image that is searched for remanence. As stated previously, the most important partition to search is the unpartitioned space from the collected drive. Figure 24 shows the various partitions of the collected image as presented by WinHex.

Name	Ext.	Size	Created	Modified	Record changed	Attr.	1st sector
Start sectors		1.0 MB					0
Partition 1	Ext4	500 MB					2,048
Partition 2	LVM2 Container	7.5 GB					1,026,048
Partition 3	Linux Swap	1.5 GB					1,028,096
Partition 4	Ext4	6.0 GB					4,239,360
Unpartitioned space		32.0 GB					16,777,216

Figure 24 The Fedora 19 Raw Image File Displayed Using WinHex

The 32 GB of unpartitioned space is extracted and isolated from the rest of the image as a separate file using WinHex. This file can be put through the developed tool to

determine whether remanence exists within the collected image of the Fedora 19 virtual machine. Creating the Configuration File section describes the configuration used for the remanence detection tool. Table 12 shows the configuration file parameters used in the local cloud environment validation testing.

Table 12 The Configuration File Parameters Used for Local Cloud Environment Validation

Tag	Raw Image	Allocated Space	Unallocated Space	Allocated Hash Key File
	win7	win7_ALLOC	win7_UNALLOC	[KEY]hash_key_win7_ALLOC
	win10	win10_ALLOC	win10_UNALLOC	[KEY]hash_key_win10_ALLOC
	ubuntu17	ubuntu17_ALLOC	ubuntu17_UNALLOC	[KEY]hash_key_ubuntu17_ALLOC
!	fedora19	fedora19_ALLOC	fedora19_UNALLOC	[KEY]hash_key_fedora19_ALLOC

Allocated Super Key File	Unallocated Hash Key File
[KEY]hash_key_super_win7	[KEY]hash_key_win7_UNALLOC
[KEY]hash_key_super_win10	[KEY]hash_key_win10_UNALLOC
[KEY]hash_key_super_ubuntu17	[KEY]hash_key_ubuntu17_UNALLOC
[KEY]hash_key_super_fedora19	[KEY]hash_key_fedora19_UNALLOC

A full description of what each column represents within the configuration file has already been discussed in Table 1 of this paper. The only column not shown that is in the raw configuration file is the directory location of the files. With this configuration file, and the file containing the unpartitioned space of the collected image, the remanence detection tool can search for any remaining, unique, operating system data from the previous cloned virtual machines. However, to determine whether the tool is accurately detecting remanence within this environment, a prediction of how the remanence might appear is required.

After careful inspection of the provisioning process used for this part of the experiment, it was determined that the virtual machine templates were provisioned via copying the entire virtual drive to the *rdm* mapped disk. This process seemed to overwrite all data previously stored on the *rdm* mapped disk including zeroed space within the template's drive. An outside program was written to simulate the provisioning process and keep track of which operating system the data belonged for each step. With this knowledge, a visual map of the hard drive can be depicted for each stage of the process in the order described previously. Figure 25 shows the steps in the process.

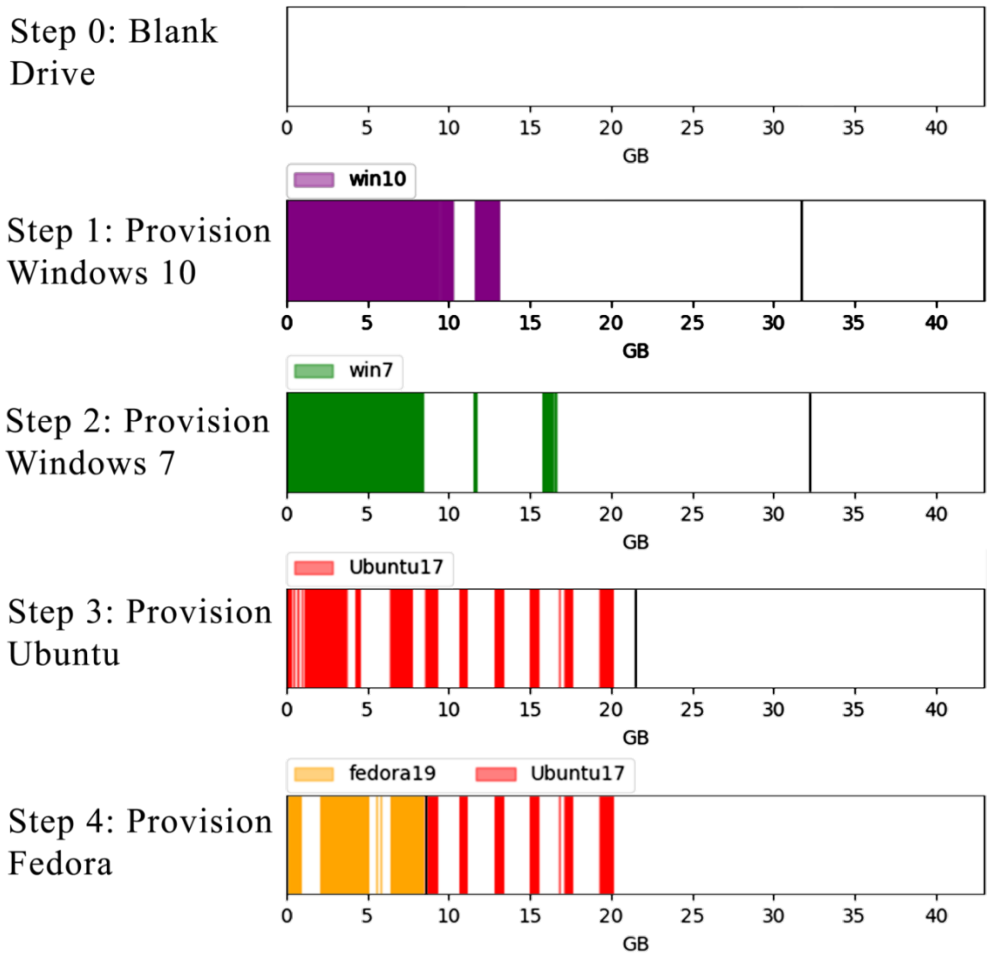


Figure 25 Data Progression of rdm Mapped Drive

The white space represents 512 byte chunks of zeroed data, while colored portions contain at least one non-zero byte. While all disks represent the same, approximately, 40 GB *rdm* mapped drive, not all templates are of the same size. Therefore, the black line within the drive represents the end of the virtual drive being provisioned.

As the graphic shows, the only drive that should contain any amount of remanence would be the drive collected after provisioning the Fedora template, while all

others should not contain any amount of remanence. Furthermore, any remanence detected should resemble that which is depicted in Step 4 of Figure 25. With this knowledge, the tool can be accurately validated using the created local cloud environment.

5.1.2.2 Validity Test 2: Local Cloud Provider - Results

While it is was essential to ensure that the Data Remanence Detection Tool was both accurate and valid in a heavily controlled and artificial environment, validating the tool's ability in a more realistic setting is also important.

After provisioning the various virtual machines in the order shown in Figure 25, the Data Remanence Detection Tool was applied to the retrieved Fedora 19 image searching for remnants of the other virtual machines: Windows 10, Windows 7, and Ubuntu 17. The order provisioned and remanence recovered, in number of test events, is shown in the Table 13.

Table 13 Local Cloud Environment Remanence Detected

	Remanence Data Type		
Provisioning Step	**Windows 10**	**Windows 7**	**Ubuntu 17**
Step 1	0	0	0
Step 2	0	0	0
Step 3	0	0	0
Step 4	1282	579	2612828*

As anticipated, the only step that yielded any remanence was Step 4 after provisioning the Fedora template. The results of Step 4 can be visually inspected with all

four remanence data types shown together, and also broken apart for ease of examination, as shown in Figure 26.

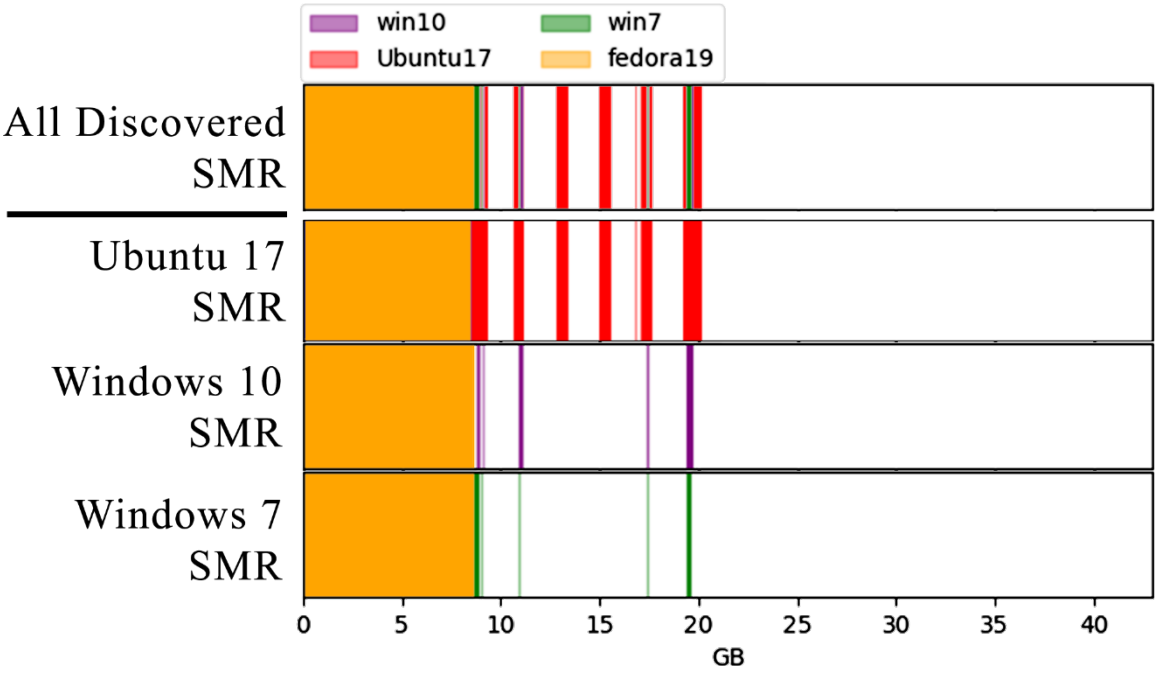


Figure 26 Stacked View of Remanence Discovered

It is important to note that the Ubuntu SMR was discovered with less vetting as the Windows 10 and Windows 7. Ubuntu 17 SMR was determined via differential hash values and does not include string detection or file extraction verification. This vetting procedure is only due to the amount of time it would take to verify all SMR discovered being infeasible. Furthermore, even though Windows 10 and Windows 7 remanences were detected, the tool is still working as intended because the unique values detected are only unique between each OVMT within the configuration file and the current TVMT.

Therefore, any hash values shared between the potential remanence VM Types will be detected as remanence separately. Continuing to follow the predicted remanence's example of what data should be found, only the SMR that can be traced back to the Ubuntu operating system is used for Step 4 validation. Visually matching the results from the drive extracted after provisioning the Fedora template at Step 4 to the anticipated results derived previously yields the following results, as shown in Figure 27.

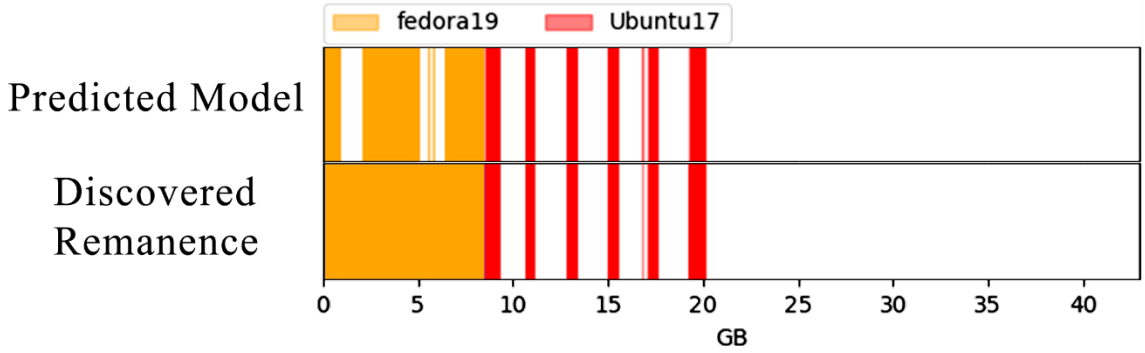


Figure 27 Visual Comparison of Predicted Model (top) with Discovered Remanence (bottom)

As can be seen, the remanence detected using the Data Remanence Detection Tool coincides with the predicted model. With the two data sets matching, along with the anticipated results from the other provisioning steps, the Data Remanence Detection Tool has been validated using the Local Cloud Provider experiment.

5.2 Data Remanence Detection Tool Validation Complete

The data remanence detection tool has been validated to detect data remanence using two different methods. Therefore, the tool is validated for its intended purpose;

i.e., to detect data remanence in cloud service providers in the wild. Furthermore, since the tool directly implements the detection process, this process is also validated.

6 CLOUD PROVIDER EVALUATION

Once the tool has been fully tested and validated, the next step is to determine if remanence can be detected against a commercial cloud provider. The cloud provider tested is Amazon's AWS. Specifically, Amazon's Elastic Computing Cloud (EC2) is being evaluated for data remanence. Amazon EC2 is a web server that provides secure, resizable compute capacity in the cloud (Amazon, 2018). This evaluation is an application of all the techniques and methods discussed in previous sections and applying them to a real-world environment. While it is not necessary for remanence to be discovered to validate the detection method previously described, it is important that the method is demonstrated against a real-world environment to prove that it is a scalable and agnostic method of evaluation. Also, it is important to determine that the results are easily interpreted even in a full-scale production environment.

6.1 Amazon Web Services

According to the Synergy Research Group, AWS is the clear leader in enterprise public cloud adoption. Figure 28 (Synergy Research Group, 2018) and Figure 29 (Synergy Research Group, 2018) describe market share and competitive position of cloud providers.

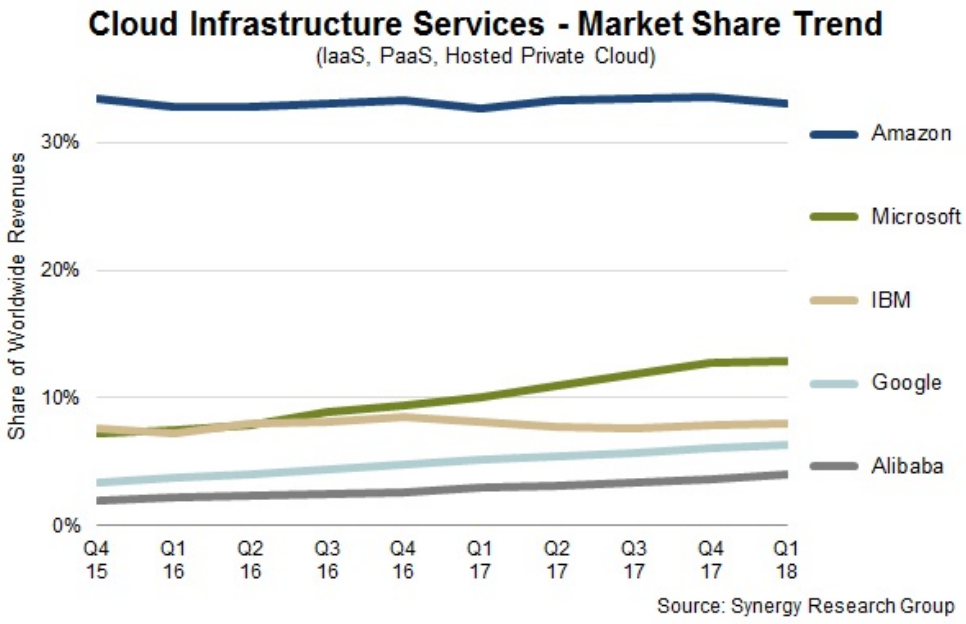


Figure 28 Market Share Trend of Cloud Infrastructure Services

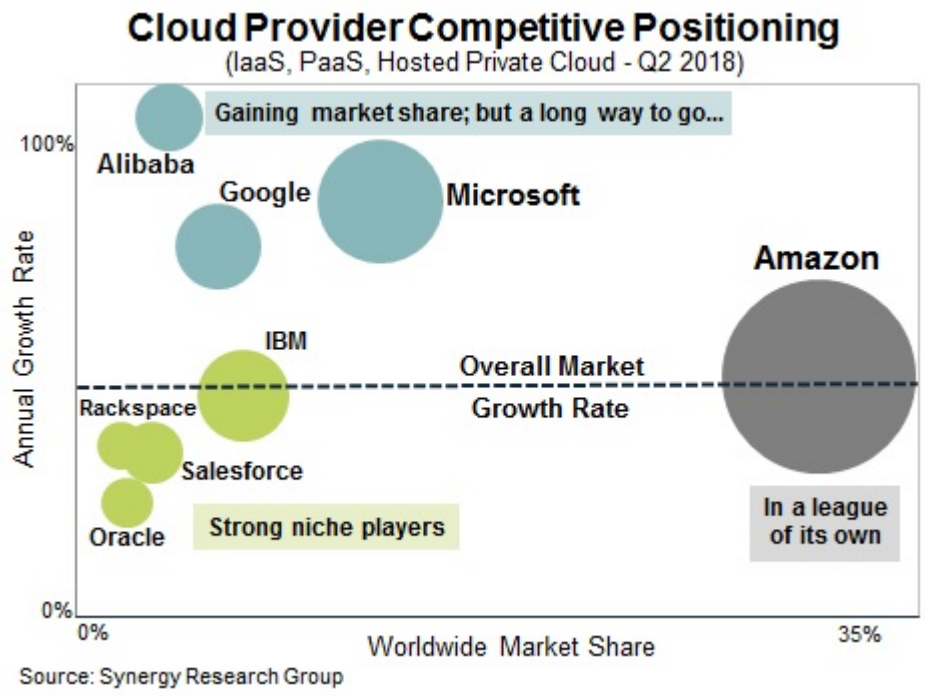


Figure 29 Competitive Positioning of Cloud Providers

With Amazon's AWS being the most widely used cloud provider, it seemed natural to utilize this approach to determine how adequately the company is preventing data remanence.

6.1.1 Detecting Remanence in AWS

EC2 allows a user to deploy templates of virtual machines of various types. This offering of different VM templates is considered a form of Infrastructure as a Service (IaaS). This method of deployment is specifically vulnerable to data remanence, and its subsequent detection, due to the direct access of the virtual hard drive and ability to run tools which collect and inspect the data residing on the VM. Therefore, EC2 instances were used to determine if data remanence occurred within AWS.

To start the data remanence detection process, a list of VM types are chosen for the experiment from the available Amazon Machine Images (AMI) that Amazon offers from its EC2 service. An AMI consists of a template for the root volume which consists of an operating system, patches, and applications (Amazon, 2018). This configuration of an AMI means that it is critical to know the AMI ID that matches the VM type to ensure that, theoretically, the exact same AMI is downloaded each time for the experiment. Table 14 lists all of the AMIs that were selected for evaluating AWS.

Table 14 AMI Chosen for the Experiment

Name	AMI ID
Amazon Linux AMI 2017.09.1 (HVM)	ami-97785bed
SUSE Linux Enterprise Server 12 SP3	ami-a03869da
Ubuntu Server 16.04 LTS (HVM)	ami-41e0b93b
Microsoft Windows Server 2016 Base	ami-603b1c1a
Microsoft Windows Server 2012 R2 Base	ami-013e197b
Microsoft Windows Server 2008 R2 Base	ami-ea391e90
Microsoft Windows Server 2003 R2 Base	ami-acd034d6

As an additional step, a custom security group is created in order to connect via Remote Desktop Protocol (RDP), Secure Shell (SSH), and other ports. This security group, shown in Table 15, is applied to all virtual machines so that the images can be collected.

Table 15 Security Group Applied to AWS Virtual Machines

Type	Protocol	Port Range
SSH	TCP	22
RDP	TCP	3389
ALL ICMP - IPv4	ALL	N/A

6.1.2 Determining Differences Between Templates

While it is not critical that the cloud provider utilizes templating, or some other method that creates virtually identical VMs, it does provide more accurate results by reducing the number of false positives of various remanence types. Therefore, an

approach to determine the similarities between two VMs of the same VM Type was developed.

6.1.2.1 Difference Detection Method

As will be discussed later in this section, remanence is capable of being traced back to its originating file. With this knowledge, a good method of determining the difference between the two VMs would be to compare filenames, with location, and the hash of each file. The two most critical comparisons are files not found in both VMs and files found in both with different hash values. This comparison will not only accomplish the goal of assessing the similarities between the two VMs, but also provide a method of comparison such that any remanence found could be cross-referenced with the results.

6.1.2.2 Difference Detection Results

Using this process of comparing files, and their respective hashes, between the TVMT and the DVM for each run, a determination can be made on how similar the images are from one provisioning to another. Therefore, in an effort to determine if there is a significant difference between images downloaded during the experiment, TSK's FLS tool was utilized to gather all allocated files, with hashes, from all downloaded images. Table 16 shows a summary of the results of the differences between VMs.

Table 16 Difference in Images for Each AWS Run

Run	TVMT Files Found	DVM Files Found	DVM Exclude	Same Hash	Different Hash	%Diff	%DVM Exclude
1	130854	130880	375	130007	498	0.38%	0.29%
2	130854	130852	360	129999	493	0.38%	0.28%
3	130854	130875	373	130003	499	0.38%	0.29%
4	130854	130877	366	130009	502	0.38%	0.28%
5	130854	130858	344	130018	496	0.38%	0.26%
6	130854	130853	346	130009	498	0.38%	0.26%
7	130854	130856	350	130008	498	0.38%	0.27%
8	130854	130856	350	130008	498	0.38%	0.27%
9	130854	130854	348	130010	496	0.38%	0.27%
10	130854	130873	370	130005	498	0.38%	0.28%

Each row describes the differences between the TVMT and the DVM for each run. The DVM Exclude column describes how many files were recovered in the DVM and not in the TVMT. The Same Hash and Different Hash columns show how many files were recovered with the same and different hash values respectively. Out of all the files that were found to have different hash values with the same file name and location, 490 were found in all images downloaded with only 21 files found in only some of the images.

Adding the percentage difference and files not found in the TVMT yields less than one percent difference between each provisioning. The majority of the files that were found in the DVM and not in the TVMT are GUID tagged, meaning that the file name would almost certainly be unique to the VM for the limited scope of this experiment. Therefore, in respect to data remanence detection, there is only a 0.38%

difference between the TVMT and DVM. The files discovered to be different can be utilized later in the process to add an extra layer of confidence that positive remanence is being discovered.

6.2 Evaluation

With the AMIs chosen, including their respective AMI IDs, a representative image of each is downloaded. Appendix F.3 show the program used to programmatically collect the images from AWS. Depending on the operating system of the virtual machine, the method of downloading varies. For Linux-based operating systems, the *dd* command is used over an SSH connection, while Windows-based operating systems are more challenging. For these, an RDP session is established mapping to the local hard drives with the built-in Terminal Server Client (tsclient) capability. The program Winhex (X-Ways, 2017) is then called from a remotely connected drive to create the image. No matter which method is used, the most important aspect is to alter the downloaded image as little as possible. Representative images downloaded are used to generate files for detecting remanence within AWS; the TVMT and OVMTs.

With all representative AMIs downloaded and processed, one of the AMIs is chosen to repeatedly download and search for remanence. For this set of experiments, the AMI ID matching AWS's Microsoft Windows Server 2008 R2 Base, ami-ea391e90, was selected. This specific AMI ID was downloaded and tested for remanence ten times.

6.3 Evaluation Results

After testing the Microsoft Windows Server 2008 R2 Base image provided by AWS ten times for remanence, various amounts of remanence were discovered. This remanence included both SMR and SUR. Table 17 displays the amount of 512 byte chunks of remanence discovered within the unallocated space of the tested virtual machine.

Table 17 Raw Remanence Results from AWS Runs

	1	2	3	4	5	6	7	8	9	10
Win 2012	218	54	1			135		429		14
Win 2016	83							270		
SUR	950	2390	1731	5323	1086	1101	1076	1329	628	174

As Table 17 shows, each downloaded image of the Windows Server 2008 R2 virtual machine yielded different results. Out of the six VM templates tested, the only SMR discovered was from Microsoft Windows Server 2012 R2 Base and Microsoft Windows Server 2016 Base images, while SUR was discovered in each run.

While the individual chunks of remanence are curious, Contiguous Remanence Events (CRE) are potentially the most interesting. CREs are where a 512 byte chunk of remanence was found with another remanence event immediately contiguous. A contiguous event is a single event whether it is a set of 2 or more contiguous chunks. In other words, contiguous remanence chunk locations are consolidated into runs of remanence events. The numbers of CREs are shown in Table 18.

Table 18 Contiguous Remanence Events from AWS Runs

	1	2	3	4	5	6	7	8	9	10
Win 2012	13	7				12		48		4
Win 2016	1							35		
SUR	131	248	198	92	150	137	156	174	114	85

These events are critical because they represent a higher potential for the remanence to derive from a single file, meaning that it is more likely that true remanence vs. a false positive is occurring. The files discovered, including the amount of experiments they were found, across all ten tests are shown in Table 19.

Table 19 Summary of Files Found Containing SMR

Windows_Server_2012R2	Occurrences
System.Xml.ni.dll	2
SMDiagnostics.ni.dll	4
System.ServiceModel.Channels.ni.dll	2
System.Runtime.Remoting.ni.dll	1
energy-report-latest.xml	1
energy-report-2017-12-13.xml	1
energy-report-2018-01-06.xml	1
energy-report-2017-11-29.xml	1
energy-report-2018-01-13.xml	1
PresentationFramework.ni.dll	1
XsdBuildTask.ni.dll	1
Windows_Server_2016	**Occurrences**
System.Xml.ni.dll	2
PresentationFramework.ni.dll	1

An interesting aspect about the files recovered is that while some files were found in multiple runs, many recovered files are only discovered in a single run. These differences give additional credence to the files being true remanence. Furthermore, referencing the Difference Detection Results, none of the associated files recovered are part of the 0.38% difference between the TVMT and DVM. Thus, all files recovered were not a collision with a slightly altered file between the TVMT and DVM, reaffirming positive remanence detection.

6.4 Run 8: A Closer Look

In order to obtain a more granular understanding of each of these runs, Run 8 was chosen for further inspection due to its high amount of both SMR and SUR detection. This granular inspection can be performed for each run. However, providing a focused examination on a single run provides enough insight into the process that it can easily be extrapolated to other runs.

6.4.1 Summary of Remanence Discovered

Run 8 resulted in not only the largest amount of SMR and SUR, but also CREs for each remanence type. For reference, Table 20 shows the raw results from the run.

Table 20 Summary of Results from Run 8

VM Type	512 Byte Chunks of Remanence	Contiguous Remanence Events
Win. 2012 R2	429	48
Win 2016	270	35
SUR	1329	174

Approximately 351KB of SMR and 666KB of SUR were discovered. From the SMR the Table 21 shows the files that were identified as containing the remanence data, potentially being the source of the remanence.

Table 21 Proposed Files Where Remanence Originated

VM Type	Files
Win 2012 R2	System.Xml.ni.dll System.ServiceModel.Channels.ni.dll PresentationFramework.ni.dll SMDiagnostics.ni.dll
Win 2016	System.Xml.ni.dll PresentationFramework.ni.dll

With the combined knowledge of CREs and associated files containing SMR, it is possible to demonstrate that CREs represent a higher probability of deriving from a single file.

6.4.2 Granular CRE Inspection

As discussed in the Theoretical Foundation section, CREs are pivotal to the data remanence detection process. CREs further validate that discovered SMR is not random data discovered by either chance or hash collisions, but rather positive assertions that remanence exists within the evaluated space. In order to make this claim, multiple tests were performed on all runs, not only Run 8, to determine the assertion's validity.

The first, and most basic, requirement is that all SMR CREs must be matched with the same associated file. If they were not, then one would not be able to state that the CRE is representing an entire file structure on the drive. Appendix D lists SMR

CREs that are associated with the same file for Run 8. All SMR CREs for Run 8 contained the same file association. In fact, the same file association was the case for all ten runs of the experiment.

If the remanence was truly a part of the associated file, the next requirement must also hold true in that the remanence should be discovered sequentially and in the correct order in respect to the associated file. All SMR CREs were found to be in the correct order not only for Run 8 but for all experiments performed. However, the process can be taken one step further by inspecting the drive directly for remanence.

Assuming the associated file is not heavily fragmented, meaning continuous, on the VM's drive, then performing a simple calculation results in determining the theoretical beginning and end of the file on the drive.

With the gathered information, it is possible to determine, and visualize where the remanence was gathered from within the file itself and the amount of remanence discovered for each associated file. The results of all remanence discovered within the associated files for Run 8 are displayed in Figure 30 through Figure 33.

Figure 30 shows that 0.48% of the file was recovered using the data remanence detection method, while 1.31% was able to be recovered by directly inspecting the drive from the theoretical start and end of the file.

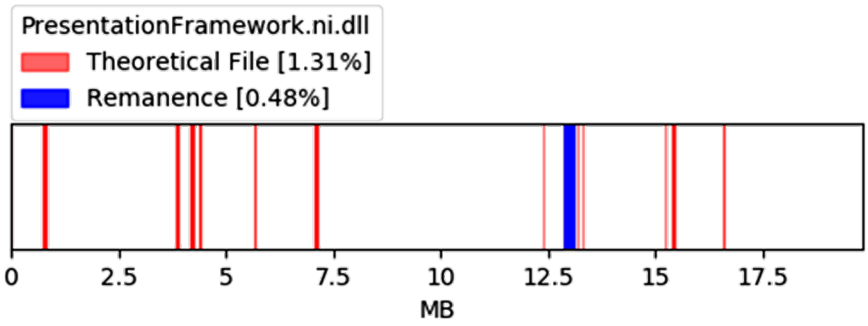


Figure 30 Graphical depiction of remanence discovered for the associated file named PresentationFramework.ni.dll

Figure 31 shows a similar amount of remanence with 0.56% and 1.87% for the different methods. While these values are not especially high, the fact that any validated, by multiple means, remanence is discovered at all is interesting.

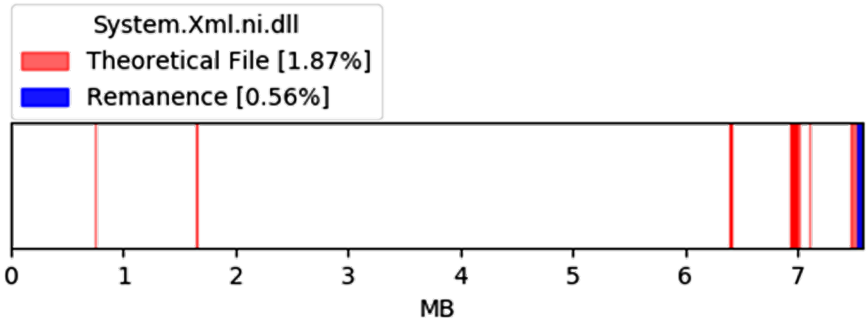


Figure 31 Graphical depiction of remanence discovered for the associated file named System.Xml.ni.dll

Figure 32 shows that 58.44% of the file was discovered as remanence and virtually the entire file was discovered on the drive with 96.1% found. With almost the entire file found with all checks passed, it would be difficult to argue that the remanence was not directly part of the associated file.

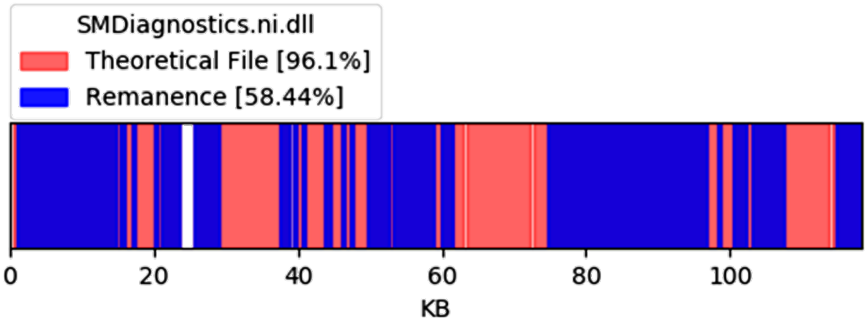


Figure 32 Graphical depiction of remanence discovered for the associated file named SMDiagnostics.ni.dll

While the other associated file's remanence detected via direct inspection of the drive and using the data remanence detection method have a general ratio of 2-to-1, Figure 33 shows a different result. Only 2.28% of SMR remanence discovered, while 25.98% of remanence was discovered via direct inspection of the drive. While not as much as shown in Figure 32, it is still a significant amount recovered and leads additional credence to the idea that CREs are telling factors in true data remanence discovery.

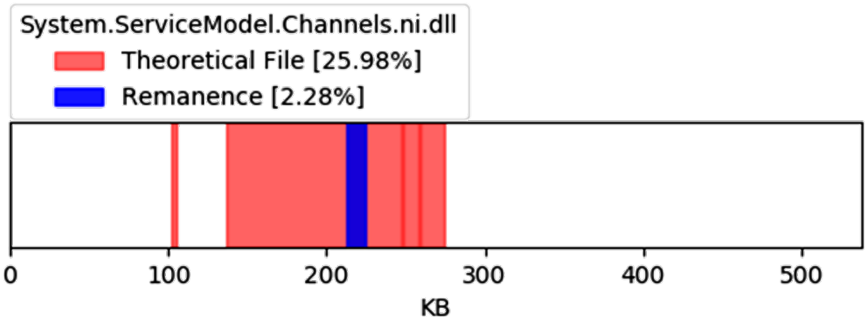


Figure 33 Graphical depiction of remanence discovered for the associated file named System.ServiceModel.Channels.ni.dll

Critically, it was discovered that all values in remanence (blue), overlapped all values in the theoretical file (red), which is the direct inspection of the drive. By using the associated file as a template, a one-to-one check of the offset within the file to 512 byte chunk hash value creates the values in red. Having these values match randomly would be improbable to consider.

While at first it would seem disconcerting that the values in red were not initially discovered by the data remanence detection method, it actually is not. After further inspection, it was discovered that these 512 bytes of remanence are common between multiple operating systems. Since the algorithm ignores values that are not unique to a specific VM type, the algorithm is working as intended. However, it clearly proves that CREs are a critical aspect in determining whether a cloud provider is removing historical user data. Appendix E contains all CREs discovered, along with their associated files, for each run.

6.4.3 Data Remanence Visualization

The raw results can be further leveraged by visualizing the image collected, as shown in Figure 34. To inspect the events effectively, the graphic has been sequentially enhanced (zoomed) at the interesting locations.

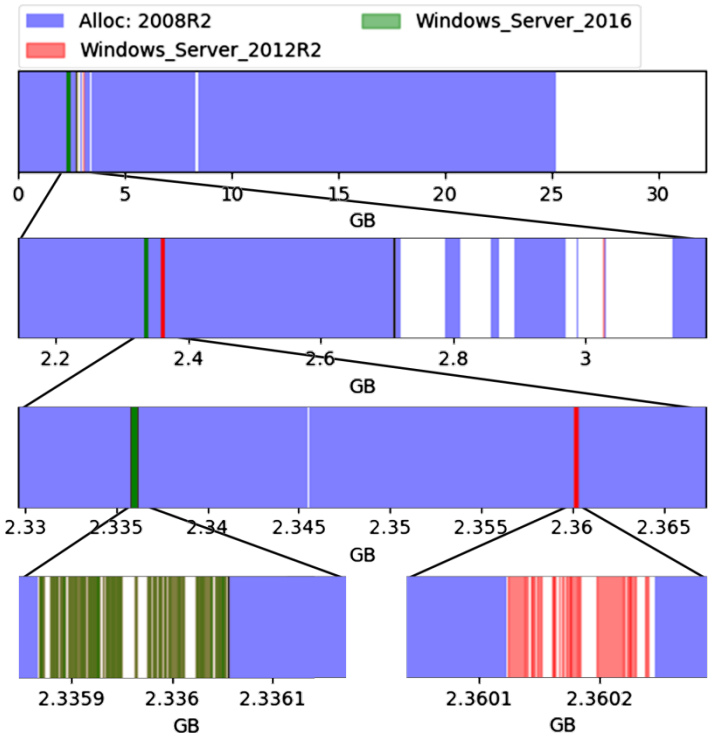


Figure 34 Visualization of SMR Found in Run 8

Once visualized, the placement of the SMR can be seen as occurring throughout the small amounts of unallocated space within the first and last clusters of the allocated space of the DVM as well as the points where CREs exist. The same visualization can be performed for the SUR, as shown in Figure 35.

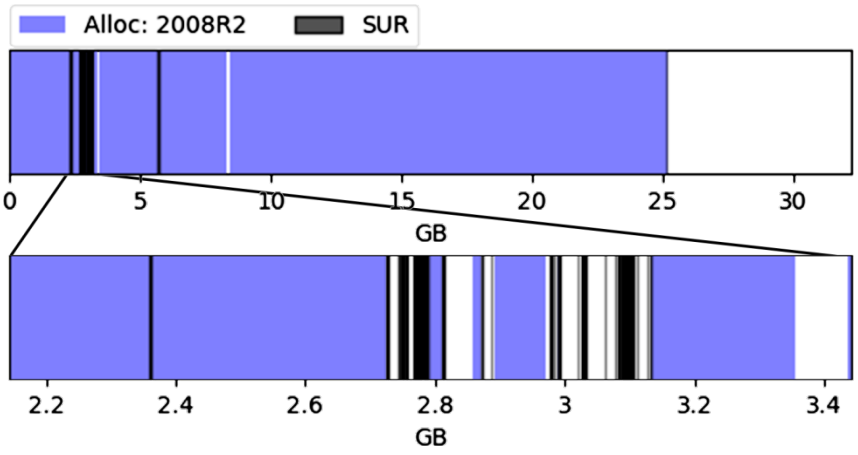


Figure 35 Visualization of SUR Found in Run 8

The SUR discovered is randomly placed. However, like the SMR found, it is found "within" the allocated space of the DVM, meaning that it is possible AWS' sanitization methods target a specific location in the disk after the last cluster of allocated space. Manual inspection of the SUR shows a range of differing data that appears to include startup script information. As discussed previously, depending on how the scripts are written, this remanence could potentially lead to leaked usernames and passwords used by the system to perform initialization steps and other tasks. See Appendix C for the visualization of SMR and SUR found within all ten AWS experiments conducted.

6.5 Summary of Findings, Possible Explanations, and Solutions

One of the foundational goals of this dissertation is to develop an infrastructure agnostic method for establishing the existence of cross-instance data remanence in a cloud environment. In order to prove that the tool implementing this method could be

applied to a commercial cloud provider, Amazon's AWS (specifically EC2) was selected as a test case.

Multiple analyses were performed including a differential analysis between two VMs instantiated from the same AMI. This testing was designed to detect any differences in the allocated space of VMs derived from the same AMI, as such differences could affect the accuracy of my method as well as indicate the template (or not) nature of Amazon's VM allocation scheme. This difference was found to be approximately 0.38%. My method accounts for this small difference, as I use only sectors that are common across multiple VMs. This small difference also verifies that Amazon is attempting to create templated images that are as similar to each other as possible. An explanation for the existence of any difference, though minimal, could be in the method of template creation. Amazon does not seem to utilize a "cloning" method, where a byte-for-byte duplicate is made from an image, but rather an installation script is used to generate and deploy an operating system with a specific configuration.

The developed method and associated tool were successfully applied to the cloud provider yielding cross-instance cloud data remanence. Furthermore, it was determined that different amounts of SMR and SUR were discovered for every run. Finding variable amounts of remnants with varying contents suggests that data from a previous instance is a likely source of the discovered remnants. If both the same amount and type of remanence were discovered for each run, then the remanence would still be a possible concern but such remanence would likely be from a contaminated template file instead of a prior VM instance.

Upon further inspection, there are also multiple CREs found in many of the runs. The importance of CREs has been described at length, in particular the fact that CRE findings support the existence of a cross-instance remanence event. Directly inspecting the VM's drive contents to locate additional fragments of the recovered associated files confirms that the remanence discovered was not a random incident, but rather a clear indication that the files identified are the likely source of SMR, due in part to the additional and contiguous file fragments located.

The root cause of remanence is difficult to determine, and beyond the scope of this dissertation. According to Amazon's own security document, the company claims that "AWS proprietary disk virtualization layer automatically resets every block of storage used by the customer" and continues stating that the "memory allocated to guests is scrubbed (set to zero)" before it is allocated to another user (Amazon, 2017). In the same document, and same section under sanitization, they recommend that end-users utilize full-disk encryption to protect their data. No further details regarding Amazon's resource sanitization procedures are publicly available. This work was inspired, in part, to compensate for that lack of information.

Further investigation to determine the cause of remanence could be based on the graphical depictions of the drives with remanence and allocated space shown. All remanence was discovered before the last cluster of allocated space. Therefore, it is possible that Amazon decided, in the interest of time and processing power, to simply sanitize the drive only after the last byte of allocated space instead of sanitizing the entire drive. Given the amount of instances Amazon is standing up and tearing down, this

approach could potentially save a significant amount of time and processing power (money).

An obvious solution is for the cloud provider, in this instance Amazon's AWS, to change their method of sanitizing the VM's drive before the next user is able to access the system. If the assumption is correct in that Amazon is only clearing data after the last allocated cluster, then expanding their sanitization to all unallocated space would solve the issue. Another approach is for AWS to employ full disk encryption, using an adequate key size. Upon tear-down of the VM, any remnants would be unusable by an unauthorized party. Furthermore, as noted elsewhere in this work, Amazon could use a cloning technique to create the instances, if the clones are thoroughly sanitized before being released and used. The deployment of any solution depends in part on how Amazon is cleaning the data currently. Any change or new technique employed will likely yield an additional overhead cost to AWS. Another solution not depending on Amazon is that end users could utilize full disk encryption or even encrypted containers to avoid leaving sensitive data as clear text in remanence.

7 CONCLUSIONS AND FUTURE WORK

Cross-instance cloud remanence detection is currently based on finding arbitrary user-generated data; this approach is haphazard, unpredictable, and incomplete. The work presented here finds remnant sectors definitively from another cloud provider instance (Source Marked Remanence, or SMR) to provide a more reliable and repeatable indication of cross-instance cloud data remanence. The method presented here applies regardless of cloud provider infrastructure and does not depend on specific instance properties or user data. The tool implementing this method was validated using artificially created remanence and a locally created cloud environment, verifying both the methodology and theoretical foundations for identifying whether cross-instance data remanence exists in a cloud environment.

The approach was applied to Amazon's AWS cloud service and positively identified cross-instance data remanence. While the quantity of both SMR and SUR (Source Unmarked Remanence, i.e., fragments not tied to another instance template) is relatively small, the fact that any amount exists indicates that infrastructure resources are not being cleansed before reallocation. The specific data exposed and associated implications are situation-dependent. Furthermore, with many of the runs yielding different values across all categories, including multiple CREs (Contiguous Remanence Events, i.e., sequential blocks from the same source file), it appears that different instances yield different unallocated space contents. Such an exposure opens the door for a Heartbleed-like attack (Durumeric, et al., 2014), where repeated exploitations are

conducted to accumulate a large volume of leaked data and increase the likelihood of sensitive data exposure.

This work contributes an infrastructure, instance, and data-independent method, along with a verified implementation, to find cross-instance data remanence in a cloud environment. Cloud providers as well as their users and customers can use this method and tools to test any cloud-based virtual machine environment. Such testing may occur at periodic intervals, or to validate the effectiveness and impact of software or infrastructure changes, and is particularly relevant for multi-tenant and multi-level security environments.

Assessing the root cause of the identified cross-instance data remanence is beyond the scope of this work, but it is safe to assume that the underlying persistent storage resources are not being fully sanitized before reallocation. This may be due to a timing issue, as unallocated space captures were conducted immediately after a new instance was provisioned. It may be that the providers tested are sanitizing unallocated space post-instantiation, although this seems unlikely since users might begin writing new data immediately after a new instance is instantiated. Future work will focus on exploring the causes of the detected cross-instance remanence and possible solutions. Additional future work will explore the effect of a Heartbleed-like exploitation (mentioned above), will examine SUR for overt user data, and will attempt to trace the sources of the SUR findings.

APPENDIX A
OUTPUT FROM DATA REMANENCE DETECTION TOOL ARTIFICIAL REMANENCE TEST (NO DUPLICATE HASH VALUES)

Appendix A.1: Output of Windows 10 Virtual Machine with Artificial Remanence

Appendix A.1.1: Generated SMR File

```
****************************REMANENCE INFORMATION***************************
Image: ./_stampTest_DISSER/win7stamp_DISSER/          Offset:5307420672
Hash: 67c44d63e0f50a85e81c58ec942cd300        Verified:PASS
***************************************************************************
0           PASS      /Users/Brad/Desktop/stamp_win7
*****************************BEGIN OF REMANENCE*****************************
4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4V
x4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4
Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx
4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4V
x4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx
******************************END OF REMANENCE******************************
***************************************************************************
****************************REMANENCE INFORMATION***************************
Image: ./_stampTest_DISSER/ubuntu64stamp_DISSER/    Offset:5307424256
Hash: fb69067aa76e261791f74a760044c3ba        Verified:PASS
***************************************************************************
1           PASS      /home/brad/.cache/vmware/drag_and_drop/Fw8LWa/stamp_ubuntu
2           PASS      /home/brad/Desktop/stamp_ubuntu
*****************************BEGIN OF REMANENCE*****************************
"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUf
w^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3D
Ufw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3
DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^
"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUfw^"3DUf
w^
******************************END OF REMANENCE******************************
***************************************************************************
****************************REMANENCE INFORMATION***************************
Image: ./_stampTest_DISSER/fedora19stamp_DISSER/     Offset:5307440128
Hash: 591215d50d9423bd884baa0617340f3b        Verified:PASS
***************************************************************************
3           N/A       * /home/fedora/.config/gconf/.testing.writeability
4           PASS      /home/fedora/Desktop/stamp_fedora
*****************************BEGIN OF REMANENCE*****************************
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
******************************END OF REMANENCE******************************
***************************************************************************
SUMMARY OF OPERATING SYSTEMS FOUND:
[Potential Data Remanence Detection]
----------------------------------
./_stampTest_DISSER/fedora19stamp_DISSER/
```

./_stampTest_DISSER/win7stamp_DISSER/
./_stampTest_DISSER/ubuntu64stamp_DISSER/

SUMMARY OF HASH VALUES FOUND:
[Potential Data Remanence Detection]
---
fedora19stamp_DISSER:591215d50d9423bd884baa0617340f3b-6
win7stamp_DISSER:67c44d63e0f50a85e81c58ec942cd300-15
ubuntu64stamp_DISSER:fb69067aa76e261791f74a760044c3ba-10

Appendix A.1.2: Generated SUR File

**********DATA OF INTEREST INFORMATION**********
Offset:5307423232 Hash: e604533e9e73f67afc98354b02420bed
**********BEGIN DATA OF INTEREST**********
ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD
3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfU
D3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwf
UD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆ
wfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3
"
**********END DATA OF INTEREST**********
****
SUMMARY OF HASH VALUES FOUND:
[SUR Detection]
---
e604533e9e73f67afc98354b02420bed-5

Appendix A.2: Output of Windows 7 Virtual Machine with Artificial Remanence

Appendix A.2.1: Generated SMR File

**********REMANENCE INFORMATION**********
Image: ./_stampTest_DISSER/ubuntu64stamp_DISSER/ Offset:6386780672
Hash: fb69067aa76e261791f74a760044c3ba Verified:PASS
****
0 PASS /home/brad/.cache/vmware/drag_and_drop/Fw8LWa/stamp_ubuntu
1 PASS /home/brad/Desktop/stamp_ubuntu
**********BEGIN OF REMANENCE**********
"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUf
wˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3D
Ufwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3
DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ
"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUf
wˆ
**********END OF REMANENCE**********
****
**********REMANENCE INFORMATION**********
Image: ./_stampTest_DISSER/win10stamp_DISSER/ Offset:6386782720
Hash: 30d2dfa2a144a5534bc16d7056bfead5 Verified:PASS
****
2 PASS /Users/brad/Desktop/stamp_win10
**********BEGIN OF REMANENCE**********
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!

‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!
*******************************END OF REMANENCE*******************************
****
*****************************REMANENCE INFORMATION**************************
Image: ./_stampTest_DISSER/fedora19stamp_DISSER/ Offset:6386798592
Hash: 591215d50d9423bd884baa0617340f3b Verified:PASS
****
3 N/A * /home/fedora/.config/gconf/.testing.writeability
4 PASS /home/fedora/Desktop/stamp_fedora
******************************BEGIN OF REMANENCE******************************
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
*******************************END OF REMANENCE*******************************
****
SUMMARY OF OPERATING SYSTEMS FOUND:
[Potential Data Remanence Detection]

./_stampTest_DISSER/fedora19stamp_DISSER/
./_stampTest_DISSER/win10stamp_DISSER/
./_stampTest_DISSER/ubuntu64stamp_DISSER/

SUMMARY OF HASH VALUES FOUND:
[Potential Data Remanence Detection]
----
fedora19stamp_DISSER:591215d50d9423bd884baa0617340f3b-5
ubuntu64stamp_DISSER:fb69067aa76e261791f74a760044c3ba-15
win10stamp_DISSER:30d2dfa2a144a5534bc16d7056bfead5-7

Appendix A.2.2: Generated SUR File

***************************DATA OF INTEREST INFORMATION***********************
Offset:6386780160 Hash: e604533e9e73f67afc98354b02420bed
***************************BEGIN DATA OF INTEREST*****************************
^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD
3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfU
D3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wf
UD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^
wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3"^wfUD3
"
****************************END DATA OF INTEREST******************************
****
SUMMARY OF HASH VALUES FOUND:
[SUR Detection]
----
e604533e9e73f67afc98354b02420bed-5

Appendix A.3: Output of Ubuntu 17 Virtual Machine with Artificial Remanence

Appendix A.3.1: Generated SMR File

****************************REMANENCE INFORMATION****************************
Image: ./_stampTest_DISSER/win7stamp_DISSER/ Offset:4608
Hash: 67c44d63e0f50a85e81c58ec942cd300 Verified:PASS
****
0 PASS /Users/Brad/Desktop/stamp_win7
******************************BEGIN OF REMANENCE******************************
4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4V
x4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4
Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx
4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4V
x4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx
*******************************END OF REMANENCE*******************************
****
****************************REMANENCE INFORMATION****************************
Image: ./_stampTest_DISSER/win10stamp_DISSER/ Offset:13824
Hash: 30d2dfa2a144a5534bc16d7056bfead5 Verified:PASS
****
1 PASS /Users/brad/Desktop/stamp_win10
******************************BEGIN OF REMANENCE******************************
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!
*******************************END OF REMANENCE*******************************
****
****************************REMANENCE INFORMATION****************************
Image: ./_stampTest_DISSER/fedora19stamp_DISSER/ Offset:24576
Hash: 591215d50d9423bd884baa0617340f3b Verified:PASS
****
2 N/A * /home/fedora/.config/gconf/.testing.writeability
3 PASS /home/fedora/Desktop/stamp_fedora
******************************BEGIN OF REMANENCE******************************
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!4Vx‡eC!
*******************************END OF REMANENCE*******************************
****
SUMMARY OF OPERATING SYSTEMS FOUND:
[Potential Data Remanence Detection]

./_stampTest_DISSER/fedora19stamp_DISSER/
./_stampTest_DISSER/win10stamp_DISSER/
./_stampTest_DISSER/win7stamp_DISSER/

SUMMARY OF HASH VALUES FOUND:
[Potential Data Remanence Detection]
----
fedora19stamp_DISSER:591215d50d9423bd884baa0617340f3b-5
win7stamp_DISSER:67c44d63e0f50a85e81c58ec942cd300-12

win10stamp_DISSER:30d2dfa2a144a5534bc16d7056bfead5-8

Appendix A.3.2: Generated SUR File

**********DATA OF INTEREST INFORMATION**********
Offset:11264 Hash: e604533e9e73f67afc98354b02420bed
**********BEGIN DATA OF INTEREST**********
ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD
3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfU
D3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwf
UD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆ
wfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3
"
**********END DATA OF INTEREST**********
****
SUMMARY OF HASH VALUES FOUND:
[SUR Detection]
---
e604533e9e73f67afc98354b02420bed-5

Appendix A.4: Output of Fedora 19 Virtual Machine with Artificial Remanence

Appendix A.4.1: Generated SMR File

**********REMANENCE INFORMATION**********
Image: ./_stampTest_DISSER/win7stamp_DISSER/ Offset:512
Hash: 67c44d63e0f50a85e81c58ec942cd300 Verified:PASS
****
0 PASS /Users/Brad/Desktop/stamp_win7
**********BEGIN OF REMANENCE**********
4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4V
x4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4
Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx
4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4V
x4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx4Vx
**********END OF REMANENCE**********
****
**********REMANENCE INFORMATION**********
Image: ./_stampTest_DISSER/win10stamp_DISSER/ Offset:1024
Hash: 30d2dfa2a144a5534bc16d7056bfead5 Verified:PASS
****
1 PASS /Users/brad/Desktop/stamp_win10
**********BEGIN OF REMANENCE**********
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!
‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡eC!‡e
C!‡eC!‡eC!‡eC!‡eC!
**********END OF REMANENCE**********
****
**********REMANENCE INFORMATION**********
Image: ./_stampTest_DISSER/ubuntu64stamp_DISSER/ Offset:4096
Hash: fb69067aa76e261791f74a760044c3ba Verified:PASS
****
2 PASS /home/brad/.cache/vmware/drag_and_drop/Fw8LWa/stamp_ubuntu
3 PASS /home/brad/Desktop/stamp_ubuntu
**********BEGIN OF REMANENCE**********

"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUf
wˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3D
Ufwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3
DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ
"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUfwˆ"3DUf
wˆ
**********END OF REMANENCE**********
****
SUMMARY OF OPERATING SYSTEMS FOUND:
[Potential Data Remanence Detection]

./_stampTest_DISSER/win10stamp_DISSER/
./_stampTest_DISSER/win7stamp_DISSER/
./_stampTest_DISSER/ubuntu64stamp_DISSER/

SUMMARY OF HASH VALUES FOUND:
[Potential Data Remanence Detection]
---
win7stamp_DISSER:67c44d63e0f50a85e81c58ec942cd300-15
ubuntu64stamp_DISSER:fb69067aa76e261791f74a760044c3ba-10
win10stamp_DISSER:30d2dfa2a144a5534bc16d7056bfead5-6

Appendix A.4.2: Generated SUR File

**********DATA OF INTEREST INFORMATION**********
Offset:3072 Hash: e604533e9e73f67afc98354b02420bed
**********BEGIN DATA OF INTEREST**********
ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD
3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfU
D3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwf
UD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆ
wfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3"ˆwfUD3
"
**********END DATA OF INTEREST**********
****
SUMMARY OF HASH VALUES FOUND:
[SUR Detection]
---
e604533e9e73f67afc98354b02420bed-5

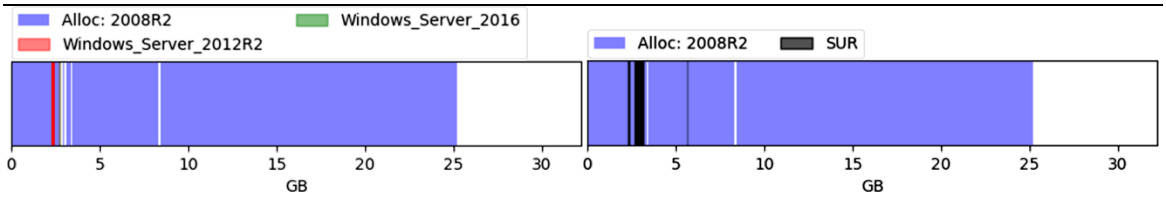
APPENDIX B
PORTION OF SCRIPT DISCOVERED IN ACADEMIC CLOUD ENVIRONMENT

```
rem must be called during this stage because Sysprep will disable
rem AutoAdminLogon if it has been configured previously.
set /A STATUS=0
rem Get the name of this batch file and the directory it is running from
set SCRIPT_NAME=%~n0
set SCRIPT_FILENAME=%~nx0
set SCRIPT_DIR=%~dp0
rem Remove trailing slash from SCRIPT_DIR
set SCRIPT_DIR=%SCRIPT_DIR:~0,-1%
echo ================================================================================
echo %SCRIPT_FILENAME% beginning to run at: %DATE% %TIME%
echo Directory %SCRIPT_FILENAME% is running from: %SCRIPT_DIR%
echo --------------------------------------------------------------------------------
set USERNAME=**********†
set PASSWORD=**********†
echo Setting DisableTaskOffload to 1...
reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters" /v
"DisableTaskOffload" /t REG_DWORD /d "1" /f
echo ERRORLEVEL: %ERRORLEVEL%
set /A STATUS+=%ERRORLEVEL%
echo Setting AutoAdminLogon to 1...
reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon" /v
"AutoAdminLogon" /t REG_SZ /d "1" /f
echo ERRORLEVEL: %ERRORLEVEL%
set /A STATUS+=%ERRORLEVEL%
echo Setting DefaultUserName to %USERNAME%...
reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon" /v
"DefaultUserName" /t REG_SZ /d "%USERNAME%" /f
echo ERRORLEVEL: %ERRORLEVEL%
set /A STATUS+=%ERRORLEVEL%
echo Setting DefaultPassword...
reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon" /v
"DefaultPassword" /t REG_SZ /d "%PASSWORD%" /f
echo ERRORLEVEL: %ERRORLEVEL%
set /A STATUS+=%ERRORLEVEL%
echo --------------------------------------------------------------------------------
echo %SCRIPT_FILENAME% finished at: %DATE% %TIME%
echo exiting with status: %STATUS%
"%SystemRoot%\system32\eventcreate.exe" /T INFORMATION /L APPLICATION /SO %SCRIPT_FILENAME%
/ID 555 /D "exit status: %STATUS%" 2>&1
exit /B %STATUS%
```

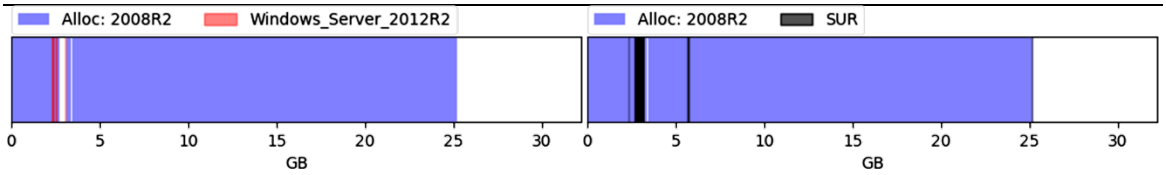
† Both the plaintext USERNAME and PASSWORD recovered during the experiment have been intentionally obfuscated in the appendix to protect the Academic Environment it was recovered from.

APPENDIX C
VISUALIZATION OF SMR AND SUR FOR ALL AWS RUNS

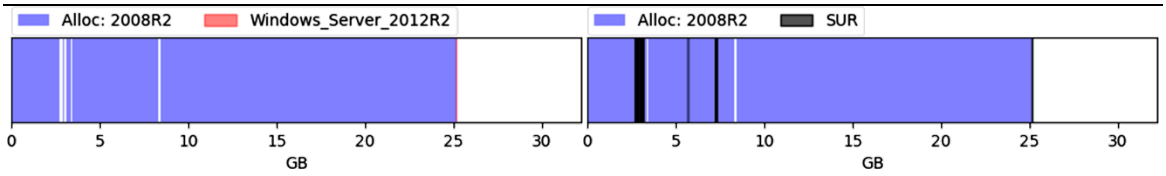
Run 1



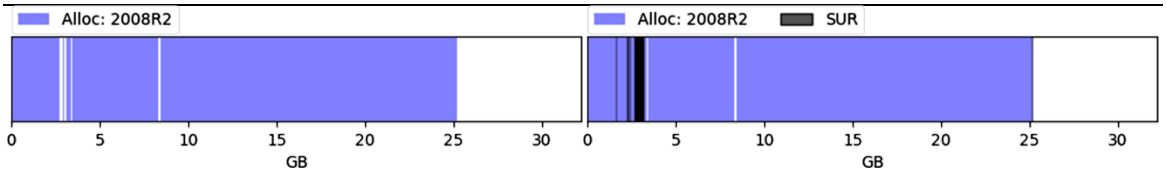
Run 2



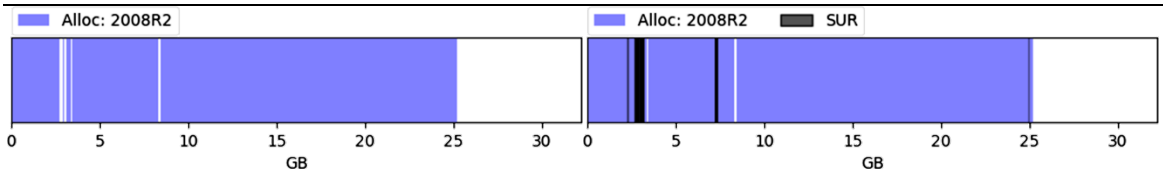
Run 3



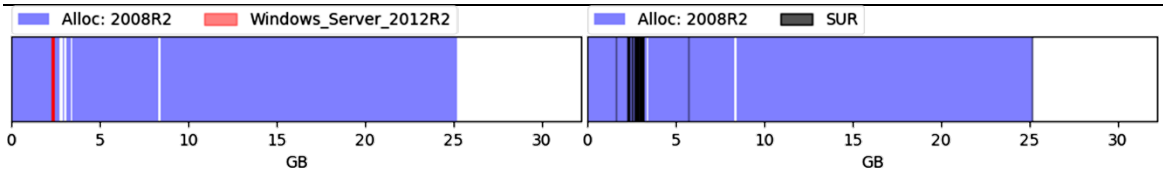
Run 4



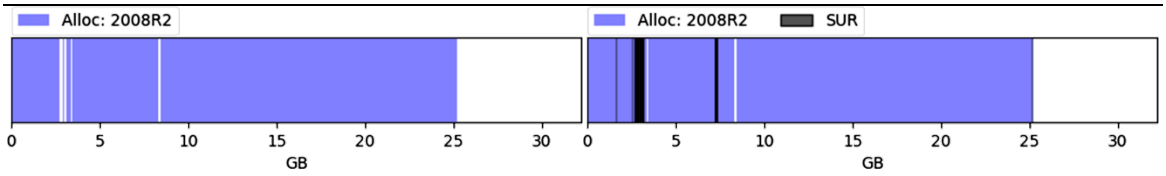
Run 5



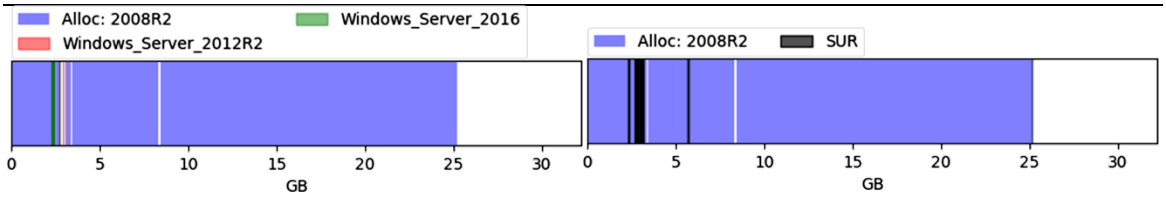
Run 6



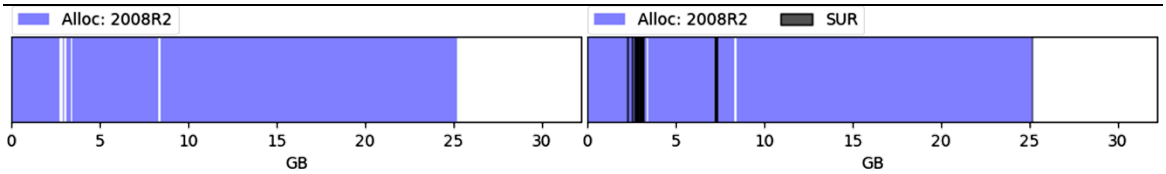
Run 7



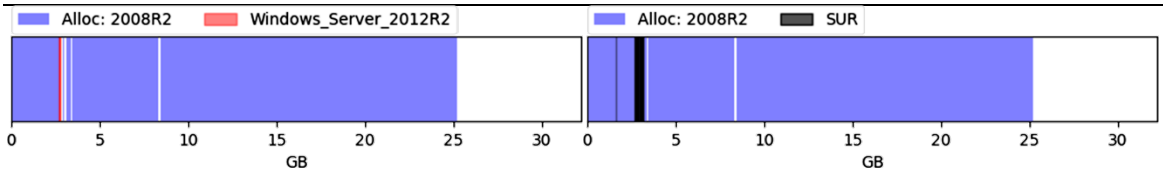
Run 8



Run 9



Run 10

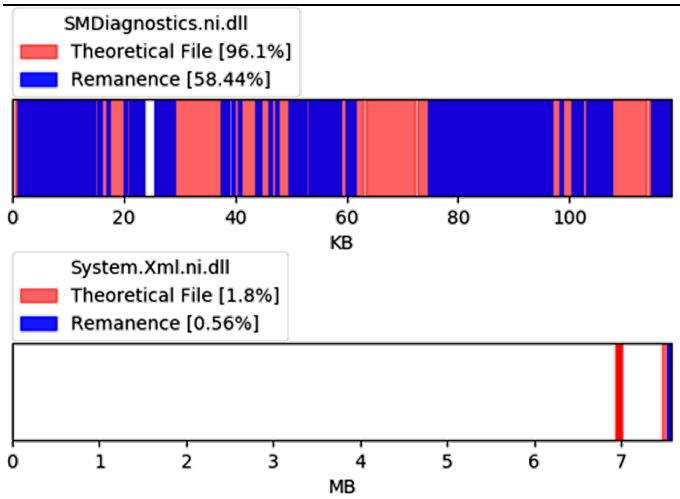


APPENDIX D
FILES FOUND WITH CORREPSONDING FIRST BYTE OFFSET LOCATION OF CRE FOR RUN 8 OF AWS EXPERIMENT

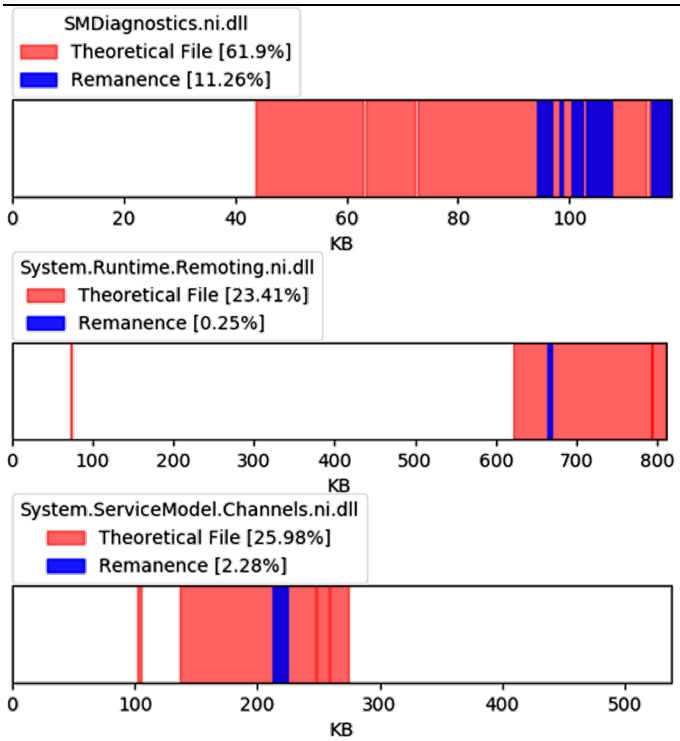
2335872000	PresentationFramework.ni.dll
2335879168	PresentationFramework.ni.dll
2335884800	PresentationFramework.ni.dll
2335887872	PresentationFramework.ni.dll
2335890944	PresentationFramework.ni.dll
2335900672	PresentationFramework.ni.dll
2335903232	PresentationFramework.ni.dll
2335909888	PresentationFramework.ni.dll
2335912960	PresentationFramework.ni.dll
2335915520	PresentationFramework.ni.dll
2335917056	PresentationFramework.ni.dll
2335919616	PresentationFramework.ni.dll
2335923200	PresentationFramework.ni.dll
2335926272	PresentationFramework.ni.dll
2335934464	PresentationFramework.ni.dll
2335936000	PresentationFramework.ni.dll
2335941120	PresentationFramework.ni.dll
2335944704	PresentationFramework.ni.dll
2335947264	PresentationFramework.ni.dll
2335963648	PresentationFramework.ni.dll
2335974912	PresentationFramework.ni.dll
2335981056	PresentationFramework.ni.dll
2335986688	PresentationFramework.ni.dll
2335992320	PresentationFramework.ni.dll
2335995904	PresentationFramework.ni.dll
2335997440	PresentationFramework.ni.dll
2336007168	PresentationFramework.ni.dll
2336023552	PresentationFramework.ni.dll
2336026112	PresentationFramework.ni.dll
2336027648	PresentationFramework.ni.dll
2336030720	PresentationFramework.ni.dll
2336043520	PresentationFramework.ni.dll
2336046592	PresentationFramework.ni.dll
2336050176	PresentationFramework.ni.dll
2360124416	SMDiagnostics.ni.dll
2360144384	SMDiagnostics.ni.dll
2360148992	SMDiagnostics.ni.dll
2360160768	SMDiagnostics.ni.dll
2360166912	SMDiagnostics.ni.dll
2360173056	SMDiagnostics.ni.dll
2360176640	SMDiagnostics.ni.dll
2360183296	SMDiagnostics.ni.dll
2360198144	SMDiagnostics.ni.dll
2360223744	SMDiagnostics.ni.dll
2360226304	SMDiagnostics.ni.dll
2360238080	SMDiagnostics.ni.dll
2709893120	System.Xml.ni.dll
3028106752	System.ServiceModel.Channels.ni.dll

APPENDIX E
GRAPHICAL REPRESENTATION OF CRES DISCOVERED FOR ALL RUNS

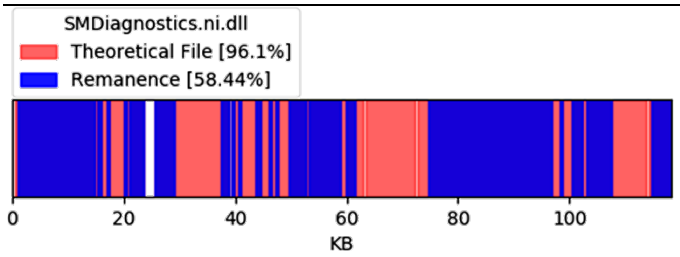
RUN 1



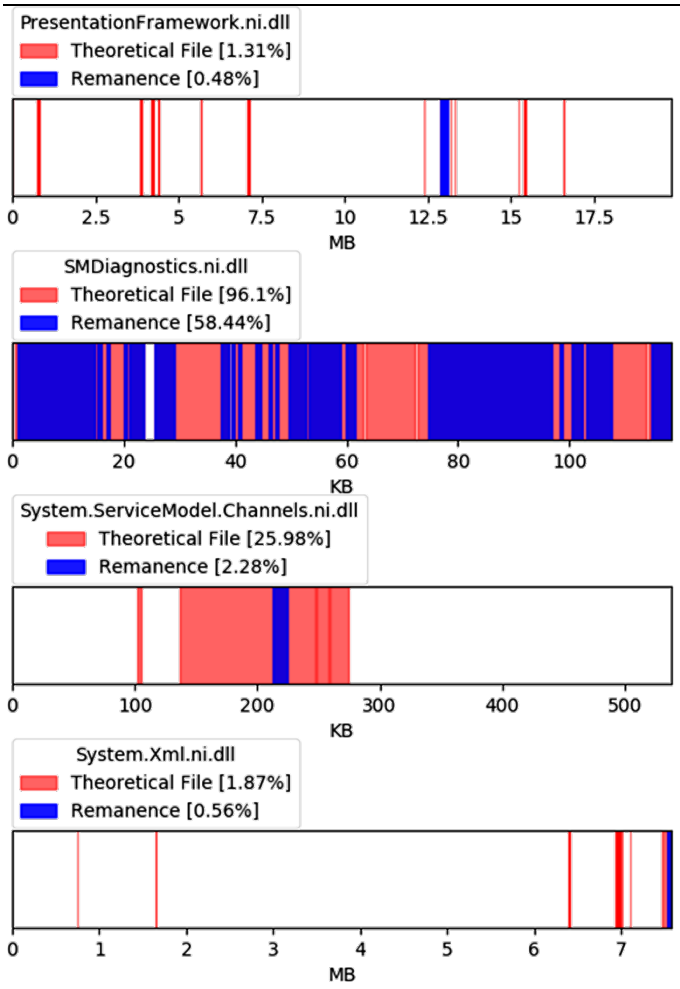
RUN 2



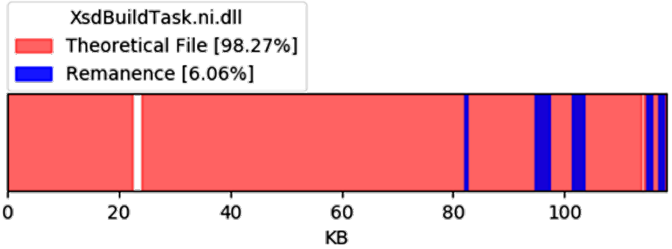
RUN 6



RUN 8



RUN 10



APPENDIX F
CUSTOM PYTHON CODE WRITTEN TO CONDUCT RESEARCH

Appendix F.1: Used to Create the Artificial Remanence Files

```
import os

statinfo = 0
#file_size = 1048576 #1Mb
#file_size = 2097152 #2Mb
#file_size = 3145728 #3Mb
#file_size = 4194304 #4Mb
file_size = 5242880 #5Mb
#file_size = 16106127360 #15GB
#file_size = 5368709120 #5GB
#file_size = 1073741824 #1GB
#file_size = 524288000 #500MB

#file_name = 'stamp_file'
#file_name = 'stamp_file_2'
#file_name = 'stamp_file_3'
#file_name = 'stamp_file_4'
#file_name = 'stamp_file_5'
file_name = 'stamp_file_5_FULL'

k = 0

with open(file_name,'wb') as sf:
    while (statinfo < file_size):
        if k < 64:
            #sf.write(b'\x12\x34\x56\x78\x12\x34\x56\x78') #1
            #sf.write(b'\x87\x65\x43\x21\x87\x65\x43\x21') #2
            #sf.write(b'\x11\x22\x33\x44\x55\x66\x77\x88') #3
            #sf.write(b'\x88\x77\x66\x55\x44\x33\x22\x11') #4
            sf.write(b'\x12\x34\x56\x78\x87\x65\x43\x21') #5
            statinfo = os.stat(file_name).st_size
        else:
            #sf.write(b'\x00\x00\x00\x00\x00\x00\x00\x00')
            #sf.write(b'\x12\x34\x56\x78\x12\x34\x56\x78') #1
            #sf.write(b'\x87\x65\x43\x21\x87\x65\x43\x21') #2
            #sf.write(b'\x11\x22\x33\x44\x55\x66\x77\x88') #3
            #sf.write(b'\x88\x77\x66\x55\x44\x33\x22\x11') #4
            sf.write(b'\x12\x34\x56\x78\x87\x65\x43\x21') #5
            statinfo = os.stat(file_name).st_size

        k+=1
print 'File Created of size ',file_size
```

Appendix F.2: Used to Create, Download and Decommission Virtual Machines from a University's Private Cloud

(Note that some variables have been obfuscated)

```
#!/usr/bin/env python

from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support.ui import Select
from selenium.webdriver.common.keys import Keys
import time
import re
import subprocess as sp

def Create_Reservation():

        # The phantomjs executable is assumed to be in your PATH:
        #driver =
webdriver.PhantomJS(executable_path=r'.\PYTHON_MODULES\phantomjs-2.0.0-
windows\bin\phantomjs.exe')
        driver = webdriver.Firefox()
        driver.get('https://www.****.edu/')

        Username = '******'
        Password = '******'
        #Click the Proceed to Login button
        procToLoginXpath = "//input[@value='Proceed to Login']"
        #Waits to make sure that this element is there
        procToLoginElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_xpath(procToLoginXpath))
        #clicks the Proceed to Login button
        procToLoginElement.click()
        #look for the id of the field
        userFieldID = 'userid'
        passwordFieldID = 'password'
        #check for the input Xpath
        loginButtonXpath = "//input[@value='Login']"
        #the webdriver waits a max of 10 seconds to see if the field is
displayed. if not it displays an exception
        userFieldElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_name(userFieldID))
        passFieldElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_name(passwordFieldID))
        loginButtonElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_xpath(loginButtonXpath))

        #clears the element first and then sends the values to the
fields
        userFieldElement.clear()
        userFieldElement.send_keys(Username)

        passFieldElement.clear()
        passFieldElement.send_keys(Password)
```

```
        #clicks the login button
        loginButtonElement.click()

        #Link Text for New Reservation
        New_Res = "New Reservation"
        #Checks for the new reservation link to be available before
trying to click it
        WebDriverWait(driver,10).until(lambda driver:
driver.find_element_by_link_text(New_Res))
        link = driver.find_element_by_link_text(New_Res)
        link.click()

        NewSubmitElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_id("newsubmit"))

        #imageName = 'Linux: Ubuntu Desktop 12.04 LTS'
        #imageName = 'Class: CSI 500'
        imageName = 'Blue Hill'
        #imageName = 'Linux: Ubuntu Server 13.04 with XEN Kernel'
        imageNameElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_id("imagesel"))
        imageNameElement.clear()
        imageNameElement.send_keys(imageName)
        imageNameElement.send_keys(Keys.RETURN)

        Select_Duration =
Select(driver.find_element_by_id('reqlength'))
        # select by visible text
        Select_Duration.select_by_visible_text('24 hours')

        NewSubmitElement.click()

        time.sleep(2)

        WebDriverWait(driver,10).until(lambda driver:
driver.find_element_by_xpath("//span[@id='dijit_form_Button_0_label']")
)

        body = driver.find_element_by_xpath('html').text

        #continues to check if Pending... is found and if so it is not
done
        while bool(re.search('Pending...',body)) == True:
                time.sleep(20)
                body = driver.find_element_by_xpath('html').text
                print 'CHECKING!'

        print 'DONE!!!'

        time.sleep(1)

        driver.refresh()
```

```
        time.sleep(1)

        ConnectXpath = "//span[@id='dijit_form_Button_0_label']"
        ConnectElement = WebDriverWait(driver,10).until(lambda driver:
driver.find_element_by_xpath(ConnectXpath))
        ConnectElement.click()

        WebDriverWait(driver,10).until(lambda driver:
driver.find_element_by_id("connectdiv"))

        time.sleep(5)

        body = driver.find_element_by_xpath('html').text

        #displays the following information for use later
        IP_ADDRESS_PATTERN = 'Remote Computer: (.*)'
        USERNAME_PATTERN = 'User ID: (.*)'
        PASSWORD_PATTERN = 'Password: (.*)'

        matchObj = re.search(IP_ADDRESS_PATTERN,body)
        IP_ADDRESS = matchObj.group(1)

        matchObj = re.search(USERNAME_PATTERN,body)
        USERNAME = matchObj.group(1)

        matchObj = re.search(PASSWORD_PATTERN,body)
        PASSWORD = matchObj.group(1)

        print IP_ADDRESS
        print USERNAME
        print PASSWORD


        '''dijit_form_Button_0_label'''
        '''dijit_form_Button_2_label'''

        driver.quit()

        return IP_ADDRESS,USERNAME,PASSWORD

def dd_image(IP_ADDRESS,USERNAME,PASSWORD,directory):
    '''Given the IP, USERNAME, and PASSWORD, this function collects the
entire image of the virtual machine
    using the program DD.  Note that unix utils is required to perform
this function.'''

    print 'Collecting DD Image'
    image_name = 'image.gz'

    #The next two lines are required so that the command window does
not show.
    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
    #Calls plink to create an SSH session and collect the desired data
```

```
    dd_args = 'cmd /c echo yes | .\plink.exe -ssh -pw '+PASSWORD+'
'+USERNAME+'@'+IP_ADDRESS+' sudo "dd if=/dev/vda2 | gzip -9 -" | dd
of='+directory+image_name
    #Calls plink and the dd program
    dd = sp.Popen(str(dd_args), startupinfo=startupinfo,
stdout=sp.PIPE, stderr=sp.PIPE)

    #prints the output from the command called
    for x in dd.communicate():
        print x

    #returns the downloaded image name
    return image_name

def Delete_Reservation():

        # The phantomjs executable is assumed to be in your PATH:
        #driver =
webdriver.PhantomJS(executable_path=r'.\PYTHON_MODULES\phantomjs-2.0.0-
windows\bin\phantomjs.exe')
        driver = webdriver.Firefox()
        driver.get('https://www.******.edu/')


        Username = '******'
        Password = '******'
        #Click the Proceed to Login button
        procToLoginXpath = "//input[@value='Proceed to Login']"
        #Waits to make sure that this element is there
        procToLoginElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_xpath(procToLoginXpath))
        #clicks the Proceed to Login button
        procToLoginElement.click()
        #look for the id of the field
        userFieldID = 'userid'
        passwordFieldID = 'password'
        #check for the input Xpath
        loginButtonXpath = "//input[@value='Login']"
        #Link Text for Current Reservations
        Current_Res = "Current Reservations"
        #the webdriver waits a max of 10 seconds to see if the field is
displayed. if not it displays an exception
        userFieldElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_name(userFieldID))
        passFieldElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_name(passwordFieldID))
        loginButtonElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_xpath(loginButtonXpath))

        #clears the element first and then sends the values to the
fields
        userFieldElement.clear()
        userFieldElement.send_keys(Username)
```

```
        passFieldElement.clear()
        passFieldElement.send_keys(Password)

        #clicks the login button
        loginButtonElement.click()

        #Checks for the current reservation link to be available before
trying to click it
        WebDriverWait(driver,10).until(lambda driver:
driver.find_element_by_link_text(Current_Res))
        link = driver.find_element_by_link_text(Current_Res)
        link.click()

        body = driver.find_element_by_xpath('html').text

        if 'Reservation has timed out' in body:
            #Checks for the Remove button to be available before trying
to click it
            RemoveID = "dijit_form_Button_0_label"
            RemoveElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_id(RemoveID))
            RemoveElement.click()

            time.sleep(1)

            #Checks for the Remove popup window to appear before trying
to click it
            RemovePopID = "remResDlgBtn_label"
            RemovePopElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_id(RemovePopID))
            RemovePopElement.click()

        elif 'Delete Reservation' in body:
            #Checks for the Delete Reservation button to be available
before trying to click it
            DeleteResXpath = "//span[@id='dijit_form_Button_1_label']"
            DeleteResElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_xpath(DeleteResXpath))
            DeleteResElement.click()

            time.sleep(1)

            #Checks for the Delete Reservation Popup windows to appear
before trying to click it
            DeleteResPopID = "endResDlgBtn_label"
            DeleteResPopElement = WebDriverWait(driver,10).until(lambda
driver: driver.find_element_by_id(DeleteResPopID))
            DeleteResPopElement.click()

        driver.quit()

        return
```

```
if __name__ == '__main__':
        directory = './GMU/'
        IP_ADDRESS,USERNAME,PASSWORD = Create_Reservation()
        dd_image(IP_ADDRESS,USERNAME,PASSWORD,directory)
        Delete_Reservation()
```

Appendix F.3: Used to Create, Download and Decommission Virtual Machines from Amazon's AWS EC2

(Note that some variables have been obfuscated)

```
import boto, boto.ec2
import time
import subprocess as sp
from subprocess import Popen, PIPE
import os
import base64, binascii, sys
from Crypto.PublicKey import RSA
import shutil
import datetime
import psexec
import itertools

def AM_connect():


'''http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/TroubleshootingIn
stancesConnecting.html#TroubleshootingInstancesConnectingMindTerm
    -For an Amazon Linux AMI, the user name is ec2-user.
    -For a RHEL AMI, the user name is ec2-user or root.
    -For an Ubuntu AMI, the user name is ubuntu or root.
    -For a Centos AMI, the user name is centos.
    -For a Fedora AMI, the user name is ec2-user.
    -For SUSE, the user name is ec2-user or root.
    -Otherwise, if ec2-user and root don't work, check with the AMI
provider.
    '''

    password = ''
    windows = ''
    state = ''

    conn = boto.ec2.connect_to_region("us-east-1",
        aws_access_key_id='*******',
        aws_secret_access_key='********')

    '''#Amazon Linux AMI 2017.09.1 (HVM), SSD Volume Type ***NEW
02/10/2018***
    username = '*******'
    image_id='ami-97785bed'
    #dev_xvda = boto.ec2.blockdevicemapping.EBSBlockDeviceType()
```

```
    #dev_xvda.size = 20 # size in Gigabytes
    #bdm = boto.ec2.blockdevicemapping.BlockDeviceMapping()
    #bdm['/dev/xvda'] = dev_xvda
    reservation = conn.run_instances(
        image_id='ami-97785bed',
        key_name='ec2-prod-key',
        instance_type='t2.micro',
        security_groups=['disser_group'],
        )
        #block_device_map = bdm)'''

    '''#Red Hat Enterprise Linux 7.4 (HVM), SSD Volume Type ***NEW
02/10/2018***
    username = '*******'
    image_id='ami-26ebbc5c'
    #dev_xvda = boto.ec2.blockdevicemapping.EBSBlockDeviceType()
    #dev_xvda.size = 20 # size in Gigabytes
    #bdm = boto.ec2.blockdevicemapping.BlockDeviceMapping()
    #bdm['/dev/xvda'] = dev_xvda
    reservation = conn.run_instances(
        image_id='ami-26ebbc5c',
        key_name='ec2-prod-key',
        instance_type='t2.micro',
        security_groups=['disser_group'],
        )'''

    '''#SUSE Linux Enterprise Server 12 SP3 (HVM), SSD Volume Type
***NEW 02/10/2018***
    username = '*******'
    image_id='ami-a03869da'
    #dev_xvda = boto.ec2.blockdevicemapping.EBSBlockDeviceType()
    #dev_xvda.size = 20 # size in Gigabytes
    #bdm = boto.ec2.blockdevicemapping.BlockDeviceMapping()
    #bdm['/dev/xvda'] = dev_xvda
    reservation = conn.run_instances(
        image_id='ami-a03869da',
        key_name='ec2-prod-key',
        instance_type='t2.micro',
        security_groups=['disser_group'],
        )'''

    '''#Ubuntu Server 16.04 LTS (HVM), SSD Volume Type ***NEW
02/10/2018***
    username = '*******'
    image_id='ami-41e0b93b'
    #dev_xvda = boto.ec2.blockdevicemapping.EBSBlockDeviceType()
    #dev_xvda.size = 20 # size in Gigabytes
    #bdm = boto.ec2.blockdevicemapping.BlockDeviceMapping()
    #bdm['/dev/xvda'] = dev_xvda
    reservation = conn.run_instances(
        image_id='ami-41e0b93b',
        key_name='ec2-prod-key',
        instance_type='t2.micro',
        security_groups=['disser_group'],
```

```
        )'''

    '''#Microsoft Windows Server 2016 Base [ami-603b1c1a] ***NEW
02/10/2018***
    username = '*******'
    image_id='ami-603b1c1a'
    windows = 1
    reservation = conn.run_instances(
        image_id='ami-603b1c1a',
        key_name='ec2-prod-key',
        instance_type='t2.micro',
        security_groups=['disser_group'])'''

    '''#Microsoft Windows Server 2012 R2 Base [ami-013e197b] ***NEW
02/10/2018***
    username = '*******'
    image_id='ami-013e197b'
    windows = 1
    reservation = conn.run_instances(
        image_id='ami-013e197b',
        key_name='ec2-prod-key',
        instance_type='t2.micro',
        security_groups=['disser_group'])'''

    #Microsoft Windows Server 2008 R2 Base [ami-ea391e90] ***NEW
02/10/2018***
    username = '*******'
    image_id='ami-ea391e90'
    windows = 1
    reservation = conn.run_instances(
        image_id='ami-ea391e90',
        key_name='ec2-prod-key',
        instance_type='t2.micro',
        security_groups=['disser_group'])

    for r in conn.get_all_instances():
        if r.id == reservation.id:
            instance = r.instances[0]
            while instance.state != 'running':
                time.sleep(5)
                instance.update()
                if state != instance.state:
                    print "Instance state: %s" % (instance.state)
                    state = instance.state


            while
conn.get_all_instance_status(instance_ids=[instance.id])[0].system_stat
us.details["reachability"] != 'passed':
                time.sleep(5)
                instance.update()
                instance_status =
conn.get_all_instance_status(instance_ids=[instance.id])[0].system_stat
us.details["reachability"]
```

```
                if state != instance_status:
                    print "Instance Status: ",instance_status
                    state = instance_status
            '''


            startupinfo = sp.STARTUPINFO()
            startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
            output =
sp.Popen(["ping.exe",instance.ip_address],startupinfo=startupinfo,stdou
t = sp.PIPE).communicate()[0]
            while ('unreachable' in output) or ('timed out' in output):
                time.sleep(5)
                output =
sp.Popen(["ping.exe",instance.ip_address],startupinfo=startupinfo,stdou
t = sp.PIPE).communicate()[0]

            '''

            time.sleep(10)

            print "Instance %s done!" % (instance.id)
            print "Instance IP is %s" % (instance.ip_address)

    #The lines here are only needed if trying to get a windows
instance.
    #Pulls in the key to be used to get the username
    if windows == 1:
        with open('ec2-prod-key.pem','rb') as keyFile:
            keyLines = keyFile.readlines()
        key = RSA.importKey(keyLines)

        print "Checking for Password..."
        while password == '':
            try:
                password = decryptPassword(key,
conn.get_password_data(instance.id))
            except:
                time.sleep(5)
        print "Password Found: ",password

    return image_id,instance.id,instance.ip_address,username,password

def AM_terminate(instance_id):

    conn = boto.ec2.connect_to_region("us-east-1",
        aws_access_key_id='*******',
        aws_secret_access_key='*******')

    conn.terminate_instances(instance_ids=[instance_id])

    '''
    for vol in conn.get_all_volumes():
        while vol.status == 'in-use':
```

```
            vol.update
            print '1',vol.status
            time.sleep(5)
    for vol in conn.get_all_volumes():
        while vol.status == 'available':
            vol.update
            print '2',vol.status
            conn.delete_volume(str(vol).split(':')[1])'''

    return

def AM_Delete_Volumes(instance_id):

    state = ''

    conn = boto.ec2.connect_to_region("us-east-1",
    aws_access_key_id='*******',
    aws_secret_access_key='*******')

    #volumes = conn.get_all_volumes(filters={'attachment.instance-id':
[instance_id]})

    #while conn.get_all_volumes():
    while conn.get_all_volumes(filters={'attachment.instance-id':
[instance_id]}):
        #for vol in conn.get_all_volumes():
        for vol in conn.get_all_volumes(filters={'attachment.instance-
id': [instance_id]}):
            if vol.status == 'available':
                conn.delete_volume(str(vol).split(':')[1])
            else:
                if state != vol.status:
                    print "Volume state: %s" % (vol.status)
                    state = vol.status
                time.sleep(10)

    print 'Volumes Deleted'

    return


def AM_dd_image(IP_ADDRESS,USERNAME,directory):
    '''Given the IP, USERNAME, and PASSWORD, this function collects the
entire image of the virtual machine
    using the program DD.  Note that unix utils is required to perform
this function.'''

    x='ERROR'

    print 'Collecting DD Image'
    image_name = 'image.gz'
    #drive = '/dev/xvda1' #AMI,SUSE,Ubuntu
    #drive = '/dev/xvda2' #RH
    #drive = '/dev/sda'
```

```
    drive = '/dev/xvda'

    #The next two lines are required so that the command window does
not show.
    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
    #Calls plink to create an SSH session and collect the desired data
    dd_args = 'cmd /c echo yes | .\plink.exe -ssh -i ec2-prod-key.ppk
'+USERNAME+'@'+IP_ADDRESS+' sudo "dd if='+drive+' | gzip -9 -" | dd
of='+directory+image_name
    print dd_args
    #Calls plink and the dd program
    dd = sp.Popen(str(dd_args), startupinfo=startupinfo,
stdout=sp.PIPE, stderr=sp.PIPE)

    #prints the output from the command called
    for x in dd.communicate():
        print x

    #returns the downloaded image name
    return image_name

def launch_rdp(IP_ADDRESS,USERNAME,PASSWORD):

    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW

    print 'Launching RDP Session'
    #Calls the program launchrdp.exe in order to create the RDP session
    rdp_args = './launchrdp.exe '+IP_ADDRESS+' 3389 '+USERNAME+'
localhost '+PASSWORD+' 0 1 0'
    sp.Popen(rdp_args, startupinfo=startupinfo, stdout = None,
shell=False).wait()

    return

def
AM_blkls_image(IP_ADDRESS,USERNAME,PASSWORD,directory,image_id,instance
_id):
    '''Automatically launches the RDP session for the created virtual
machine'''

    image_name = 'image.gz'

    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW

    launch_rdp(IP_ADDRESS,USERNAME,PASSWORD)

    time.sleep(10)

    print 'Determining Session ID'
    session_id = -1
    sess_result = ''
```

```
    #while session_id == -1:
    sess_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' cmd /c "query session '+USERNAME+'"'
    sess_proc = sp.Popen(sess_args, startupinfo=startupinfo,
stdout=sp.PIPE)
    while ('Administrator' not in sess_result) and (session_id == -1):
        sess_result = sess_proc.stdout.readlines()
        for sess in sess_result:
            if (sess.strip()[:3] == 'rdp') and ('Active' in sess):
                session_id = sess.strip().split()[2]
                print 'Session ID: ',session_id
                break
        time.sleep(5)

    print 'Collecting Version Information'
    AMI_args = r'cmd /c "echo AMI ID: '+image_id+' >
.'+directory[1:]+'Info.txt"'
    print AMI_args
    AMI_proc = sp.Popen(AMI_args, startupinfo=startupinfo,
stdout=sp.PIPE).wait()
    AMI_args = r'cmd /c "echo INSTANCE ID: '+instance_id+' >>
.'+directory[1:]+'Info.txt"'
    print AMI_args
    AMI_proc = sp.Popen(AMI_args, startupinfo=startupinfo,
stdout=sp.PIPE).wait()
    #Uses psexec to collection information on the system
    info_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' '+session_id+' cmd /c "systeminfo >>
\\\\tsclient\\Z\\DISSERTATION\\'+directory[1:]+'Info.txt"'
    info_proc = sp.Popen(info_args, startupinfo=startupinfo,
stdout=sp.PIPE).wait()
    #info_result = info_proc.stdout.readlines()

    time.sleep(5)

    print datetime.datetime.now().time()

    print 'Launching BLKLS Process'
    #query_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' -s -i '+session_id+' cmd /c
"\\\\tsclient\Z\Collection.bat"'
    query_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' -s -i '+session_id+' cmd /c
"\\\\tsclient\\Z\\DISSERTATION\\blkls.exe -e \\\\.\\C: |
\\\\tsclient\\Z\\DISSERTATION\\7z.exe a
\\\\tsclient\\Z\\DISSERTATION\\'+directory[1:]+'image.gz -tgzip -
siimage"'
    print query_args
    #query_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' -s -i 1 cmd /c "ipconfig /all"'
    #query_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' -s -i 1 cmd /c "//tsclient/J/DISSERTATION/Collection.bat"'
    #query_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' -s -i '+session_id+' cmd /c
```

```
"//tsclient/J/DISSERTATION\blkls.exe -e //./c: |
//tsclient/J/DISSERTATION/gzip.exe -9 - > //tsclient/J/image.gz"'
    query = sp.Popen(query_args, stdout=sp.PIPE, stderr=sp.PIPE).wait()
    #print query.communicate()

    time.sleep(5)

    print datetime.datetime.now().time()

    #shutil.move('.\\image.gz','')


    print 'Closing RDP Session'
    logoff_args = r'.\psexec.exe \\'+IP_ADDRESS+' -u '+USERNAME+' -p
'+PASSWORD+' -s -i '+session_id+' cmd /c logoff '+session_id
    out = sp.Popen(logoff_args, stdout=sp.PIPE, stdin=sp.PIPE, shell =
False).wait()

    return image_name

#realtime check of subprocess. will use for debugging
def myrun(cmd):
    """from http://blog.kagesenshi.org/2008/02/teeing-python-
subprocesspopen-output.html
    """
    p = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr=sp.STDOUT)
    stdout = []
    while True:
        line = p.stdout.readline()
        stdout.append(line)
        print line,
        if line == '' and p.poll() != None:
            break
    return ''.join(stdout)

########################################################################
#These functions are used to decrypt the username from the EC2 instance
########################################################################
def pkcs1_unpad(text):
    #From http://kfalck.net/2011/03/07/decoding-pkcs1-padding-in-python
    if len(text) > 0 and text[0] == '\x02':
        # Find end of padding marked by nul
        pos = text.find('\x00')
        if pos > 0:
            return text[pos+1:]
    return None

def long_to_bytes (val, endianness='big'):
    # From http://stackoverflow.com/questions/8730927/convert-python-
long-int-to-fixed-size-byte-array

    # one (1) hex digit per four (4) bits
    try:
        #Python < 2.7 doesn't have bit_length =(
```

```
        width = val.bit_length()
    except:
        width = len(val.__hex__()[2:-1]) * 4

    # unhexlify wants an even multiple of eight (8) bits, but we don't
    # want more digits than we need (hence the ternary-ish 'or')
    width += 8 - ((width % 8) or 8)

    # format width specifier: four (4) bits per hex digit
    fmt = '%%0%dx' % (width // 4)

    # prepend zero (0) to the width, to zero-pad the output
    s = binascii.unhexlify(fmt % val)

    if endianness == 'little':
        # see http://stackoverflow.com/a/931095/309233
        s = s[::-1]

    return s

def decryptPassword(rsaKey, password):
    #Undo the whatever-they-do to the ciphertext to get the integer
    encryptedData = base64.b64decode(password)
    ciphertext = int(binascii.hexlify(encryptedData), 16)

    #Decrypt it
    plaintext = rsaKey.decrypt(ciphertext)

    #This is the annoying part.  long -> byte array
    decryptedData = long_to_bytes(plaintext)
    #Now Unpad it
    unpaddedData = pkcs1_unpad(decryptedData)

    #Done
    return unpaddedData

def result_dir():
    '''Creates the directory location all of the files are sent to.
    Dynamically creates the directory depending on how many folders
    are in the 'RESULTS' directory.  Therefore, each run has its
    own folder and can be reviewed later.'''

    #This is the path where all of the folders will reside
    path = './Amazon/RESULTS'
    #path = './_StampTest_DISSER/RESULTS'
    #If the path exists, then count all of the
    #folders in the path and creates a new folder that is one
    #value larger than the last to keep the results separate
    if os.path.exists(path):
        count = 0
        count = len(next(os.walk(path))[1])
    else:
        count = 0
    directory = path+'/RUN_'+str(count)+'/'
```

```
    os.makedirs(directory)

    return directory

if __name__ == '__main__':

    #How many times the tool should run
    N = 1

    #Repeats the code below N times
    for _ in itertools.repeat(None, N):

        image_id,instance_id,IP_ADDRESS,USERNAME,PASSWORD =
AM_connect()

        #Determines the correct directory the files should be placed
        #directory = result_dir()

        launch_rdp(IP_ADDRESS,USERNAME,PASSWORD)

        image_name =
AM_blkls_image(IP_ADDRESS,USERNAME,PASSWORD,directory,image_id,instance
_id)
        #image_name = AM_dd_image(IP_ADDRESS,USERNAME,directory)

        AM_terminate(instance_id)
        AM_Delete_Volumes(instance_id)
```

Appendix F.4: Generates All Files Referenced in the Configuration File for each of the Tested Virtual Machine Types (TVMT)

```
'''This program is designed to create the files required for the config
file.  It accepts a location, and image
and then creates the Allocated, Unallocated, Allocated Hash Key File
and Allocated Super Key File.'''

import subprocess as sp
import sys
import os
import re
import gzip
import re, hashlib, time, datetime, sys, fileinput
from functools import partial
from collections import defaultdict


def unzip(directory,image_name):
    '''Given the image_name that was collected via DD earlier, this
function extracts the image file using 7zip.'''

    print 'Unzipping Downloaded Image'

    #used to suppress the command prompt
    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW

    #location of 7z.exe
    unzip_args = 'C:\\Program Files\\7-Zip\\7z.exe x -o'+directory+'
'+directory+'\\'+image_name
    #Unzips into the same directory just created
    unzip_process = sp.Popen(unzip_args, startupinfo=startupinfo,
stdout=sp.PIPE, stderr=sp.PIPE,shell=False).wait()

    return

#Defines the blklsicollect function
def Unallocated (directory,image_name):
    '''Accepts a single logical image file (image), in order to collect
all slack space found in the image.
    The image file must be logical in that there is no offset to the
logical drive of the system imaged.
    The end result is a file that contains all of the slack found using
blklsi.exe with the slack
    separated by the inode location the slack was found in.'''

    #Collects only the name of the image to be used for the output file
names
    image_name = image_name.split(".")[0]

    print 'Collecting Unallocated Space From: ',directory+image_name
```

```
    #Location of blklsi.exe
    blkls_process = './blkls.exe'

    #Switches used when calling blkls.exe
    #Used to collect all of the unallocated space
    blkls_1 ='-A'
    #The image that the slack is collected from
    blkls_2 =directory+image_name

    #The outfile that the slack is written to after collected
    unalloc_file = directory+str(image_name)+'_UNALLOC'

    blkls_args = './blkls.exe -A '+directory+image_name

    print blkls_args

    #Calls blkls.exe as a subprocess, collects all of the slack and
writes it out to the outfile
    with open(unalloc_file,'w+') as f:
        startupinfo = sp.STARTUPINFO()
        startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
        #sp.Popen([blkls_process,blkls_1,blkls_2],
startupinfo=startupinfo, stdout = f, shell=False).wait()
        sp.Popen(blkls_args, startupinfo=startupinfo, stdout = f,
shell=False).wait()
        f.seek(0)

    #Returns the name of the blkls_file
    return unalloc_file

def Allocated (directory,image_name):
    '''Accepts a single logical image file (image), in order to collect
all slack space found in the image.
    The image file must be logical in that there is no offset to the
logical drive of the system imaged.
    The end result is a file that contains all of the slack found using
blklsi.exe with the slack
    separated by the inode location the slack was found in.'''

    #Collects only the name of the image to be used for the output file
names
    image_name = image_name.split(".")[0]

    print 'Collecting Allocated Space From: ',directory+image_name

    #Location of blklsi.exe
    blkls_process = './blkls.exe'

    #Switches used when calling blkls.exe
    #Used to collect all of the allocated space
    blkls_1 ='-a'
    #The image that the slack is collected from
    blkls_2 =directory+image_name
```

```
    #The outfile that the slack is written to after collected
    alloc_file = directory+str(image_name)+'_ALLOC'

    blkls_args = './blkls.exe -a '+directory+image_name

    print blkls_args

    #Calls blkls.exe as a subprocess, collects all of the slack and
writes it out to the outfile
    with open(alloc_file,'w+') as f:
        startupinfo = sp.STARTUPINFO()
        startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
        #sp.Popen([blkls_process,blkls_1,blkls_2],
startupinfo=startupinfo, stdout = f, shell=False).wait()
        sp.Popen(blkls_args, startupinfo=startupinfo, stdout = f,
shell=False).wait()
        f.seek(0)

    #Returns the name of the blkls_file
    return alloc_file

def no_dups(seq, idfun=None):
   # order preserving
   if idfun is None:
       def idfun(x): return x
   seen = {}
   result = []
   for item in seq:
       marker = idfun(item)
       # in old Python versions:
       # if seen.has_key(marker)
       # but in new ones:
       if marker in seen: continue
       seen[marker] = 1
       result.append(item)
   return result

def Hash_Key_File(directory,image_file):

    print 'Creating Image Hash Key'

    BLOCKSIZE = 512

    #hash_key = {}
    hash_key = []

    k=0

    out_file = directory+'[KEY]hash_key_'+image_file.split("/")[-1]

    with open(out_file, 'wb') as out, open(image_file, 'rb+') as fp:
        for block in iter(lambda: fp.read(BLOCKSIZE), ''):
            #Calculates the MD5 hash of the slack
            hash_object = hashlib.md5(block)
```

```
            hashed_block = hash_object.hexdigest()
            #Writes out the inode location and the hash of the 512
chunk slack space
            #out.write(str(hashed_block)+'\n')
            #k+=1
            #hash_key[k]=str(hashed_block)
            # inside your loop to add items:
            hash_key.append(str(hashed_block))

        hash_key_2=no_dups(hash_key, idfun=None)
        for hash_line in hash_key_2:
            out.write(str(hash_line)+'\n')
    #print len(hash_key_2)

    print 'Operation Completed'
    return

class Dictlist(dict):
    def __setitem__(self, key, value):
        try:
            self[key]
        except KeyError:
            super(Dictlist, self).__setitem__(key, [])
        self[key].append(value)

def Hash_Key_Super_File(directory,image_file):

   print 'Creating Super Hash Key File'

   BLOCKSIZE = 512

   #hash_key = {}
   hash_key = Dictlist()

   k=0

   out_file = directory+'[KEY]hash_key_super_'+(image_file.split('/')[-
1]).split('.')[0]

   image_file = directory+image_file

   with open(out_file, 'wb') as out, open(image_file, 'rb+') as fp:
      for block in iter(lambda: fp.read(BLOCKSIZE), ''):
         #Calculates the MD5 hash of the slack
         hash_object = hashlib.md5(block)
         hashed_block = hash_object.hexdigest()
         #Writes out the inode location and the hash of the 512 chunk
slack space
         #out.write(str(hashed_block)+'\n')
         #hash_key[k]=str(hashed_block)
         #inside your loop to add items:
         #offset = k*512
         hash_key[str(hashed_block)]=int(k)
         k+=1
```

```
      for h_val in hash_key:
         out.write(str(h_val)+"\t"+str(hash_key[h_val])+"\n")

   #print len(hash_key_2)

   print 'Operation Completed'
   return out_file

def hash_dictionary(out_file):

    hash_key = Dictlist()

    with open(out_file, 'rb') as out:
        for line in out:
            #print (line.split('\t')[0]).strip()
            for offset in
((line.split('\t')[1]).replace("[","")).replace("]","").split(","):
                #print offset.strip()
                hash_key[(line.split('\t')[0]).strip()]=offset.strip()
            #print "\n"

    print 'Operation Completed'

    return hash_key


def find_hash(out_file,h_val):

    with open(out_file, 'rb') as f:
        content = f.read()
        for line in content.splitlines():
            if h_val in line:
                for offset in
((line.split('\t')[1]).replace("[","")).replace("]","").split(","):
                    print offset.strip()
                break

    return

if __name__ == '__main__':

    #directory = './Amazon_Linux_AMI/'
    #directory = './SUSE_Linux_Enterprise/'
    #directory = './Ubuntu_Server_LTS/'
    #directory = './Windows_Server_2003R2/'
    #directory = './Windows_Server_2008R2/'
    #directory = './Windows_Server_2012R2/'
    directory = './Windows_Server_2016/'

    #image_file = 'Amazon_Linux_AMI'
    #image_file = 'SUSE_Linux_Enterprise'
    #mage_file = 'Ubuntu_Server_LTS'
    #image_file = 'Windows_Server_2003R2'
    #image_file = 'Windows_Server_2008R2'
```

```
    #image_file = 'Windows_Server_2012R2'
    image_file = 'Windows_Server_2016'
    image_name = image_file
    print datetime.datetime.now().time()
    #unzip(directory,image_name)
    unalloc_file = Unallocated(directory,image_name)
    #unalloc_file = directory+image_file+'_UNALLOC'
    print datetime.datetime.now().time()
    alloc_file = Allocated(directory,image_name)
    print datetime.datetime.now().time()
    Hash_Key_File(directory,alloc_file)
    print datetime.datetime.now().time()
    Hash_Key_File(directory,unalloc_file)
    print datetime.datetime.now().time()
    Hash_Key_Super_File(directory,image_file)
    print datetime.datetime.now().time()
```

Appendix F.5: Main Data Remanence Detection Tool

```
#Import the following modules
import re
import hashlib
import binascii
from os import walk
import os
import math
from time import time
from time import sleep
import time
import mechanize
import sys
import win32crypt
import subprocess as sp
import gzip
import datetime
import signal
#from EDU_CONNECT import Create_Reservation
#from EDU_CONNECT import Delete_Reservation
import collections
import itertools
from itertools import groupby, count
#from Amazon_Connect_v2 import AM_connect
#from Amazon_Connect_v2 import AM_terminate
#from Amazon_Connect_v2 import AM_dd_image
#from Amazon_Connect_v2 import AM_blkls_image
#from Amazon_Connect_v2 import AM_Delete_Volumes

#This changes the dictionary class
class Dictlist(dict):
    def __setitem__(self, key, value):
        try:
            self[key]
        except KeyError:
```

```
            super(Dictlist, self).__setitem__(key, [])
        self[key].append(value)

#This changes the dictionary class
class Dictset(dict):
    def __setitem__(self, key, value):
        try:
            self[key]
        except KeyError:
            super(Dictset, self).__setitem__(key, set())
        self[key].add(value)

def read_config(config_file):

    image_info = Dictlist()
    t_image_info = []

    with open(config_file,'rb') as cf:
        for line in cf:
            if (line[0]!="#") and (line[0]!="!"):

loc,raw_image,alloc_file,unalloc_file,key_file,hash_map,key_file_unallo
c = line.strip().split(';')
                image_info[loc] = raw_image
                image_info[loc] = alloc_file
                image_info[loc] = unalloc_file
                image_info[loc] = key_file
                image_info[loc] = hash_map
                image_info[loc] = key_file_unalloc
            elif (line[0]=="!"):

loc,raw_image,alloc_file,unalloc_file,key_file,hash_map,key_file_unallo
c = line.strip().split(';')
                t_image_info.append(loc)
                t_image_info.append(raw_image)
                t_image_info.append(alloc_file)
                t_image_info.append(unalloc_file)
                t_image_info.append(key_file)
                t_image_info.append(hash_map)
                t_image_info.append(key_file_unalloc)
            else:
                pass

    return image_info,t_image_info

def merge_two_dicts(x, y):
    '''Given two dicts, merge them into a new dict as a shallow
copy.'''
    z = x.copy()
    z.update(y)
    return z

def dd_image(IP_ADDRESS,USERNAME,PASSWORD,directory):
```

```
    '''Given the IP, USERNAME, and PASSWORD, this function collects the
entire image of the virtual machine
    using the program DD.  Note that unix utils is required to perform
this function.'''

    #value of concatenated output string
    dd_output = ''

    #While there is an error for the DD of the image, restart the
download.
    while True:

        print 'Collecting DD Image'
        image_name = 'image.gz'

        #The next two lines are required so that the command window
does not show.
        startupinfo = sp.STARTUPINFO()
        startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
        #Calls plink to create an SSH session and collect the desired
data
        #You may need to install ssh using 'sudo apt-get install
openssh-server'
        #May need to add username ALL=NOPASSWD: ALL to the /etc/sudoers
file
        #May have to replace 'Defaults requiretty' with 'Defaults
!requiretty' in /etc/sudoers file (vi -> crtl-x -> ESC -> :wq!)
        #The drive that is copied depends on what is shown using 'sudo
lsblk'
        #dd_args = 'cmd /c echo yes | .\plink.exe -ssh -pw '+PASSWORD+'
'+USERNAME+'@'+IP_ADDRESS+' sudo "dd if=/dev/vda2 | gzip -9 -" | dd
of='+directory+image_name
        #dd_args = 'cmd /c echo yes | .\plink.exe -ssh -2 -load EDU -pw
'+PASSWORD+' '+USERNAME+'@'+IP_ADDRESS+' sudo "dd if=/dev/vda2 | gzip -
9 -" | dd of='+directory+image_name
        dd_args = 'cmd /c echo yes | .\plink.exe -ssh -pw '+PASSWORD+'
'+USERNAME+'@'+IP_ADDRESS+' sudo "dd if=/dev/sda1 | gzip -9 -" | dd
of='+directory+image_name
        #dd_args = 'cmd /c echo yes | .\plink.exe -ssh -pw '+PASSWORD+'
'+USERNAME+'@'+IP_ADDRESS+' sudo "dd if=/dev/sda | gzip -9 -" | dd
of='+directory+image_name
        #dd_args = 'cmd /c echo yes | .\plink.exe -ssh -pw '+PASSWORD+'
'+USERNAME+'@'+IP_ADDRESS+' sudo "dd if=/dev/sda3 | gzip -9 -" | dd
of='+directory+image_name
        #dd_args = 'cmd /c echo yes | .\plink.exe -ssh -pw '+PASSWORD+'
'+USERNAME+'@'+IP_ADDRESS+' sudo "dd if=/dev/vda1 | gzip -9 -" | dd
of='+directory+image_name

        #if a windows machine using cygwin, you can use this.  Sudo is
not used and the C drive is targeted
        #dd_args = 'cmd /c echo yes | .\plink.exe -ssh -2 -pw
'+PASSWORD+' '+USERNAME+'@'+IP_ADDRESS+' "dd if=\\\\\\\.\\\C: | gzip -9
-" | dd of='+directory+image_name
```

```
        #dd_args = 'cmd /c echo yes | .\plink.exe -ssh -2 -load EDU -pw
'+PASSWORD+' '+USERNAME+'@'+IP_ADDRESS+' "dd if=\\\\\\\.\\\C: | gzip -9
-" | dd of='+directory+image_name

        #Calls plink and the dd program
        dd = sp.Popen(str(dd_args), startupinfo=startupinfo,
stdout=sp.PIPE, stderr=sp.PIPE)

        #prints the output from the command called
        for x in dd.communicate():
            #print x
            #concatenate x so that it is a complete string
            dd_output = dd_output + str(x)

        print dd_output

        if "error" not in dd_output.lower():
            break
        else:
            print "ERROR DETECTED: ",datetime.datetime.now()
            dd_output = ''
            time.sleep(60)

    #returns the downloaded image name
    return image_name

def unzip(directory,image_name):
    '''Given the image_name that was collected via DD earlier, this
function extracts the image file using 7zip.'''

    print 'Unzipping Downloaded Image'

    #used to suppress the command prompt
    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW

    #location of 7z.exe
    unzip_args = 'C:\\Program Files\\7-Zip\\7z.exe x -o'+directory+'
'+directory+'\\'+image_name
    #Unzips into the same directory just created
    unzip_process = sp.Popen(unzip_args, startupinfo=startupinfo,
stdout=sp.PIPE, stderr=sp.PIPE,shell=False).wait()

    return

#Defines the blklsicollect function
def blklscollect (directory,image_name):
    '''Accepts a single logical image file (image), in order to collect
all slack space found in the image.
    The image file must be logical in that there is no offset to the
logical drive of the system imaged.
    The end result is a file that contains all of the slack found using
blklsi.exe with the slack
    separated by the inode location the slack was found in.'''
```

```
    #Collects only the name of the image to be used for the output file
names
    image_name = image_name.split(".")[0]

    print 'Collecting Unallocated Space From: ',directory+image_name

    #Location of blklsi.exe
    blkls_process = '.\\blkls.exe'

    #Switches used when calling blkls.exe
    #Used to collect all of the unallocated space
    blkls_1 ='-A'
    #The image that the slack is collected from
    blkls_2 =directory+image_name

    #The outfile that the slack is written to after collected
    blkls_file = directory+'blkls_'+str(image_name)+'.txt'

    blkls_args = '.\\blkls.exe -A '+directory+image_name

    print blkls_args

    #Calls blkls.exe as a subprocess, collects all of the slack and
writes it out to the outfile
    with open(blkls_file,'w+') as f:
        startupinfo = sp.STARTUPINFO()
        startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
        #sp.Popen([blkls_process,blkls_1,blkls_2],
startupinfo=startupinfo, stdout = f, shell=False).wait()
        sp.Popen(blkls_args, startupinfo=startupinfo, stdout = f,
shell=False).wait()
        f.seek(0)

    #Returns the name of the blkls_file
    return blkls_file

def key_maker(key_file):
    '''Iterates over the previosly made file that contains the 512 byte
hashed key file of the
    operating system that might be found in the unallocated space and
places every hash into
    a set called 'key_list'. The same is done with the 512 byte hashed
file of the
    allocated space of the operating system you are testing, which is
then placed into a set
    called 'diff_list'.'''

    #Set that contains all of the hashes for the key file
    key_list = set()

    #Iterates over the key file to determine if a hash match is found
    with open(key_file, 'rb') as kf:
        for h_key in kf:
```

```
            #This is a list of all the keys in the key file
            key_list.add(h_key.strip())

    return key_list

def read_image(blkls_file, image_name,directory,image_info,
t_image_info):
    '''Accepts the key_list(list of hashes in potentional remanence
OS), diff_list
    (list of hashes in tested OS),blkls_file(Unallocated space file of
tested OS),
    alloc_file(raw allocated space file of tested OS),image_name(name
of gzipped
    file downloaded by DD),and directory(location files are saved to
generated
    dynamically by function).
    This function reads the unallocated space of the image 512 bytes at
a time,
    converts to binary, determines if all 1's or 0's, if so then ignore
the string.
    Then the MD5 hash of the 512 byte string is calculated and checked
to see if
    it is contained in the key_list(key file) and not in rem_spills
set(no dups).
    If it is contained in the key_list, then it is possible that
remanence did occur.
    It is then checked against the allocated space of the Os being
tested.  If
    the 512 bytes is not found in the allocated space, and was found in
the
    remanence OS, then there is a high probability that it is
remanence.'''

    print 'Remanence File: ',blkls_file

    alloc_file = t_image_info[0][1:]+t_image_info[2]

    print 'Creating Tested Key Set [ALLOC]:
',t_image_info[0][1:]+t_image_info[4]
    t_key_list = key_maker(t_image_info[0][1:]+t_image_info[4])
    print 'Creating Tested Key Set [UNALLOC]:
',t_image_info[0][1:]+t_image_info[6]
    t_key_list_unalloc = key_maker(t_image_info[0][1:]+t_image_info[6])

    key_list_dict = Dictlist()

    operating_systems = set()

    key_list_same = set()

    #This is a dictionary set with the hash value of the block as the
    #key and the value is a set of files written
    files_writ = Dictset()
```

```
    for image in image_info:
        print 'Creating Key Set: ',image+image_info[image][3]
        image_key_list = key_maker(image+image_info[image][3])
        key_list_diff = image_key_list - t_key_list
        #print len(image_key_list)
        #print len(key_list_diff)
        key_list_dict[image] = key_list_diff
        #print len(key_list_dict[image][0])
        #All of the hash values for allocated space accross all images
collected
        #key_list_same = key_list_same.union(set(image_key_list))
        key_list_same |= set(image_key_list)

    #Add the tested image key list to the sample set as well
    #key_list_same = key_list_same.union(set(t_key_list))
    key_list_same |= set(t_key_list)

    #Remanence found broken up by 512 byte chunks.  Also displays the
offset
    #in the blkls file, the hash, and raw bytes found.
    rem_file = 'remanence_found.txt'
    #Only the raw remanence bytes found with no breaks in between
    raw_rem_file = 'raw_remanence_found.txt'
    #The reverse lookup file of data found that does not match any of
the
    #collected images' allocated space hash values
    reverse_rem_file = 'reverse_remanence_found.txt'
    #Only the raw reverse remanence bytes found with no breaks in
between
    raw_rev_rem_file = 'raw_reverse_remanence_found.txt'

    #Simply a hashed list of all the remanence found
    #Used to make sure duplicates are not listed in the remanence file.
    rem_spills = set()
    #Same as rem_spills, but for reverse lookup
    rev_rem_spills = set()

    #Keeps track of which hash values were passed per OS
    passed = Dictset()

    #This is just the test remanence hash values.  Can remove any time.
    #67c44d63e0f50a85e81c58ec942cd300 = 87654321 for 512 bytes
    #30d2dfa2a144a5534bc16d7056bfead5 = 12345678 for 512 bytes
    hash_remanence =
['67c44d63e0f50a85e81c58ec942cd300','30d2dfa2a144a5534bc16d7056bfead5']
    #Testing amount of times test remanence is discovered and then
printing
    hash_remanence_test = {}
    #Tracks the amount of reverse remanence hash files discovered
    rev_remanence_test = {}

    #The hash values of the spills found
    hash_spills = []
```

```
    #How many bytes read at a time
    BLOCKSIZE = 512

    #counter used to keep track of location in unallocated file
    k=0

    writ_dict_total = Dictlist()
    verify_overall = ""

    #Determines the size of the unallocated file
    fpsize = os.path.getsize(blkls_file)
    print "Size of unallocated space: ",fpsize

    #Opens all of the files required to determine remanence
    with open(blkls_file, 'rb+') as fp, open(directory+rem_file, 'wb')
as rf, open(directory+raw_rem_file, 'wb') as rrf, open(alloc_file,
'rb') as f, open(directory+reverse_rem_file, 'wb') as revf,
open(directory+raw_rev_rem_file, 'wb') as rrevf:

        #Reads in the allocated file to be searched multiple times
later
        print 'Reading in allocated space (This could take some time)'
        alloc_data = f.read()
        print datetime.datetime.now().time()

        #Creates a progressbar to keep track of how far in the process
of writing
        #remanence the program is
        pb = progressbar(100,"*")

        #Reads in the entire unallocated space file 512 bytes at a time
        while True:
            chunk=fp.read(BLOCKSIZE)
            #if there is nothing in the chunk variable, break loop
            if not chunk: break
            #converts the chunk into binary for inspection
            bin_chunk = bin(int(binascii.hexlify(chunk), 16))
            #counts the number of ones in the binary string
            ones = sum(c=='1' for c in str(bin_chunk)[2:])
            #counts the number of zeroes in the binary string
            zeros = sum(b=='0' for b in str(bin_chunk)[2:])
            #Determines the length of the chunk
            chunk_len = len(str(bin_chunk)[2:])
            #Calculates the MD5 hash of the slack
            hash_object = hashlib.md5(chunk)
            hashed_block = hash_object.hexdigest()
            #If the chunk is NULL, all 1's or all 0's, then it is
ignored.
            #Else if is written to the spill files
            #Also checks if hash is in the stored tested image's
unallocated key file
            #if it is not, then it is something different and should be
investigated
```

```
            if str(bin_chunk) != '0b0' and (ones != chunk_len) and
(zeros != chunk_len) and (hashed_block not in t_key_list_unalloc):
                for image in image_info:
                    #Resets the files that have been written so that it
will pick up the same
                    #file in different images
                    files_writ = Dictset()
                    #print 'Testing: ',image+image_info[image][3]
                    key_list_diff = key_list_dict[image][0]
                    #The offset of the potential remanence in the
unallocated file
                    offset = int(k)*512
                    #If the hash is found in the key file then continue
since it
                    #is potential remanence
                    #if (hashed_block in key_list_diff) and
(str(image+hashed_block) not in rem_spills):
                    if (hashed_block in key_list_diff):
                        #Just for testing to make sure it is picking up
all test remanence
                        #Can remove after testing!
                        if image.split('/')[2]+'*'+hashed_block in
hash_remanence_test:

hash_remanence_test[image.split('/')[2]+'*'+hashed_block]=hash_remanenc
e_test[image.split('/')[2]+'*'+hashed_block]+1
                        else:

hash_remanence_test[image.split('/')[2]+'*'+hashed_block]=1
                        if (str(image+hashed_block) not in rem_spills):
                            #if (hashed_block in key_list_diff):
                            #print
image+image_info[image][3],hashed_block
                            #print offset
                            #Adds the hashed block to the set so that
duplicate values
                            #are not added to the remanence file
                            #rem_spills.add(image+hashed_block)
#COMMENTED OUT SO THAT MULTIPLES ARE WRITTEN
                            #Determines if the potential remanence can
be found in the
                            #allocated data of the tested OS
                            #print 'Searching....'
                            response = fnd(alloc_data, f,chunk,start=0)
                            #print 'Search Complete!'
                            #If it cannot be found, a value of -1 is
returned and means that
                            #remanence occurred and writes it out to
the file
                            if response == -1:
                                hash_map = image+image_info[image][4]
                                #print image+image_info[image][4]
                                with open(hash_map, 'rb') as hm:
```

```
                                        h_offset_list =
find_hash(hm,hashed_block)
                                        h_offset_list_grouped =
group_offset(h_offset_list)
                                        #print "\nOFFSET LIST:
",h_offset_list_grouped
                                        for h_offset in
h_offset_list_grouped:
                                            #print "hashed block:
",hashed_block
                                            #print "\noffset: ",h_offset
                                            raw_image =
image+image_info[image][0]
                                            #print 'Finding files....'
                                            ffind_result =
file_find(raw_image,h_offset)
                                            #print 'Files Found!'

                                            if ffind_result:
                                                for ffind in ffind_result:
                                                    if (hashed_block not in
files_writ) or (ffind not in files_writ[hashed_block]):
                                                        writ_dict =
file_retrieve(raw_image,h_offset,directory,hashed_block,chunk,ffind)
                                                        writ_dict_total =
merge_two_dicts(writ_dict_total,writ_dict)

                                                        #print
'writ_dict',writ_dict
                                                        for writ_file in
writ_dict:
files_writ[hashed_block]=(writ_dict[writ_file])[0]

                                                        for file_key in
writ_dict_total:
                                                            if
verify_overall == "FAIL":
                                                                pass
                                                            else:
                                                                if
(writ_dict_total[file_key])[1] == "FAIL":
verify_overall = "FAIL"
                                                                else:
verify_overall = "PASS"
passed[(str(image).split('/'))[-2]] = hashed_block
                                            elif not writ_dict_total:
                                                verify_overall = 'N/A'
                                                #print 'writ dict
before',writ_dict_total
```

```
                                        writ_dict_total =
Dictlist()
                                        writ_dict_total['N/A']="NO
FILES FOUND"
                                        writ_dict_total['N/A']=" "
                                        #print 'writ dict
after',writ_dict_total

                                    #print ffind_result_total

rf.write('****************************REMANENCE
INFORMATION****************************\n')
                                    rf.write('Image:
'+str(image)+'\tOffset:'+str(offset)+'\n')
                                    rf.write('Hash:
'+str(hashed_block)+'\t'+'Verified:'+verify_overall+'\n')

#rf.write('Hash:'+str(hashed_block)+'\n')

rf.write('************************************************************
****************\n')

                                    for file_key in writ_dict_total:

rf.write(str(file_key)+'\t'+(writ_dict_total[file_key])[1]+'\t'+(writ_d
ict_total[file_key])[0]+'\n')
                                        #print
(writ_dict_total[file_key])[1],'\t',(writ_dict_total[file_key])[0]


rf.write('****************************BEGIN OF
REMANENCE****************************\n')
                                    rf.write(str(chunk)+'\n')

rf.write('****************************END OF
REMANENCE****************************\n')

rf.write('************************************************************
****************\n')
                                    #Raw remanence is written to this
file
                                    rrf.write(str(chunk)+'\n')
                                    h_offset_list = []
                                    writ_dict_total = Dictlist()
                                    verify_overall = ""
                                    #print
'************************************************************************
****\n'
                                    operating_systems.add(str(image))

                    elif (hashed_block not in key_list_same):
                        if hashed_block in rev_remanence_test:

rev_remanence_test[hashed_block]=rev_remanence_test[hashed_block]+1
```

```
                        else:
                            rev_remanence_test[hashed_block]=1
                        if (str(offset)+str(hashed_block) not in
rev_rem_spills):
                            #Adds the hashed block to the set so that
duplicate values
                            #are not added to the remanence file

rev_rem_spills.add(str(offset)+str(hashed_block))

                            revf.write('**************************DATA
OF INTEREST INFORMATION**********************\n')
                            revf.write('Offset:'+str(offset)+'\tHash:
'+str(hashed_block)+'\n')

revf.write('**************************BEGIN DATA OF
INTEREST*****************************\n')
                            revf.write(str(chunk)+'\n')
                            revf.write('**************************END
DATA OF INTEREST*****************************\n')

revf.write('************************************************************
******************\n')
                            #Raw reverse remanence is written to this
file
                            rrevf.write(str(chunk)+'\n')

            #Increments the counter to keep track of the location in
the unallocated file
            k+=1

            #Calculates how far in the unallocated file the program is
and displays
            #the progress on the screen
            percent = ((float(k)*512)/int(fpsize))*100
            #print (float(k)*512),' ',int(fpsize),' ',percent
            pb.progress(percent)

        #print 'operating_systems',operating_systems,'
',bool(operating_systems)
        #print 'passed',passed

        if bool(operating_systems):
            rf.write('SUMMARY OF OPERATING SYSTEMS FOUND:\n')
            rf.write('[Potential Data Remanence Detection]\n')
            rf.write('-----------------------------------\n')
            print '\nSUMMARY OF OPERATING SYSTEMS FOUND:'
            print '[Potential Data Remanence Detection]'
            print '-----------------------------------'
            for o in operating_systems:
                if (str(o).split('/'))[-2] in passed:
                    rf.write(o+'\n')
                    print o
            rf.write('\n')
```

```
        if bool(hash_remanence_test):
            rf.write('SUMMARY OF HASH VALUES FOUND:\n')
            rf.write('[Potential Data Remanence Detection]\n')
            rf.write('-----------------------------\n')
            print '\nSUMMARY OF HASH VALUES FOUND:'
            print '[Potential Data Remanence Detection]'
            print '-----------------------------'
            if passed:
                for hrem in hash_remanence_test:
                    if str(hrem.split('*')[1]) in
passed[str(hrem.split('*')[0])]:

rf.write((hrem.split('*')[0]+':'+str(hrem.split('*')[1])+'-
'+str(hash_remanence_test[hrem])+'\n'))
                        print
hrem.split('*')[0],':',hrem.split('*')[1],'-',hash_remanence_test[hrem]

        if bool(rev_remanence_test):
            revf.write('SUMMARY OF HASH VALUES FOUND:\n')
            revf.write('[Reverse Remanence Detection]\n')
            revf.write('-----------------------------\n')
            print '\nSUMMARY OF HASH VALUES FOUND:'
            print '[Reverse Remanence Detection]'
            print '-----------------------------'
            for hrev in rev_remanence_test:
                #Just for testing. Can remove!
                if str(hrev) == 'e604533e9e73f67afc98354b02420bed':
                    print str(hrev),'-
',str(rev_remanence_test[hrev]/len(image_info)),'\n'
                revf.write(str(hrev)+'-
'+str(rev_remanence_test[hrev]/len(image_info))+'\n')
            #    print str(hrev),'-',str(rev_remanence_test[hrev])
            #
revf.write((hrev.split('*')[0]+':'+str(hrev.split('*')[1])+'-
'+str(rev_remanence_test[hrev])+'\n'))
            #    print hrev.split('*')[0],':',hrev.split('*')[1],'-
',rev_remanence_test[hrev]

    return

def group_offset(h_offset_list):
    '''Used to group all of the offset numbers so that the same file
    is not processed multiple times'''
    h_offset_list_grouped = []
    h_offset_list[:] = [x / 512 for x in h_offset_list]
    groups = groupby(h_offset_list, key=lambda item, c=count():item-
next(c))
    tmp = [list(g) for k, g in groups]
    for t in tmp:
        h_offset_list_grouped.append(t[0])

    h_offset_list_grouped[:] = [x * 512 for x in h_offset_list_grouped]
```

```
    return h_offset_list_grouped


def fnd(data, f, s, start=0):
    '''Used to find a binary string in a file.'''
    f.seek(0)
    result = data.find(s)
    return result


class progressbar(object):
    '''Creates the progress bar object when called and displays the
progress
    given the percentage completed in the process.'''
    def __init__(self, finalcount, block_char='.'):
        self.finalcount = finalcount
        self.blockcount = 0
        self.block = block_char
        self.f = sys.stdout
        if not self.finalcount: return
        self.f.write('\n------------------ % Progress ----------------
---\n')
        self.f.write('    1    2    3    4    5    6    7    8    9
0\n')
        self.f.write('----0----0----0----0----0----0----0----0----0----
0\n')
    def progress(self, count):
        count = min(count, self.finalcount)
        if self.finalcount:
            percentcomplete = int(round(100.0*count/self.finalcount))
            if percentcomplete < 1: percentcomplete = 1
        else:
            percentcomplete=100
        blockcount = int(percentcomplete//2)
        if blockcount <= self.blockcount:
            return
        for i in range(self.blockcount, blockcount):
            self.f.write(self.block)
        self.f.flush()
        self.blockcount = blockcount
        if percentcomplete == 100:
            self.f.write("\n")

def result_dir():
    '''Creates the directory location all of the files are sent to.
    Dynamically creates the directory depending on how many folders
    are in the 'RESULTS' directory.  Therefore, each run has its
    own folder and can be reviewed later.'''

    #This is the path where all of the folders will reside
    path = './_StampTest_DISSER/RESULTS'
    #path = './_StampTest_DISSER/RESULTS'
    #If the path exists, then count all of the
    #folders in the path and creates a new folder that is one
```

```
    #value larger than the last to keep the results separate
    if os.path.exists(path):
        count = 0
        count = len(next(os.walk(path))[1])
    else:
        count = 0
    directory = path+'/RUN_'+str(count)+'/'
    os.makedirs(directory)

    return directory

def file_find(raw_image,offset):

    #print "FINDING FILE"

    fsstat_args = "fsstat "+raw_image

    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
    fsstat_proc = sp.Popen(fsstat_args, startupinfo=startupinfo,
stdout=sp.PIPE)
    fsstat_result = fsstat_proc.stdout.read()

    for line in fsstat_result.splitlines():
        if ("Cluster Size" in line) or ("Block Size" in line):
            cluster_size = int((line.split(":")[1]).strip())

    cluster_loc = int((float(offset)/cluster_size))

    ifind_args = "ifind -d "+str(cluster_loc)+" "+raw_image
    ifind_proc = sp.Popen(ifind_args, startupinfo=startupinfo,
stdout=sp.PIPE)
    ifind_result = ifind_proc.stdout.read().strip()
    if ifind_result != 'Inode not found':
        ffind_args = "ffind -a "+raw_image+" "+ifind_result
        ffind_proc = sp.Popen(ffind_args, startupinfo=startupinfo,
stdout=sp.PIPE)
        ffind_result = ffind_proc.stdout.read().splitlines()
        #print "raw_image",raw_image
        #print "cluster_loc",str(cluster_loc)
        #print "ifind_result",ifind_result
        #print "ffind_result",ffind_result
    else:
        ffind_result = False

    return ffind_result


def file_retrieve(raw_image,offset,directory,hashed_block,s,ffind):

    verify = ""
    writ_files = []
    writ_dict = Dictlist()
```

```
    ret_files = directory+'Files/'
    if os.path.exists(ret_files):
        pass
    else:
        os.makedirs(ret_files)

    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW

    #if ffind_result:
    #    for ffind in ffind_result:
    path, dirs, files = os.walk(ret_files).next()
    num_files = len(files)
    file_name = ffind.split("/")[-1].strip()
    writ_name = ret_files+str(num_files)+'_'+hashed_block+'_'+file_name
    ffind = '"%s"'%ffind
    fcat_args = "fcat -R "+ffind+" "+raw_image
    with open(writ_name, 'wb') as f:
        fcat_proc = sp.call(fcat_args, startupinfo=startupinfo,
stdout=f)

    file_size = os.path.getsize(writ_name)
    if file_size > 0:
        with open(writ_name, 'rb') as f:
            data = f.read()
            result = data.find(s)
            if result != -1:
                verify = "PASS"
            else:
                #print 'ffind',ffind
                f.seek(0)
                #print 'length of slack',len(s)
                #raw_slack = s.rstrip(' \t\r\n\0')
                raw_slack = s[:s.rfind('\n')]
                length_raw_slack = len(raw_slack)
                #print 'length of raw slack',length_raw_slack
                f.seek(len(data)-length_raw_slack-1)
                data = f.read()
                #print 'length of data string',len(data)
                result = data.find(raw_slack)
                #print 'where string was found',result
                if result != -1:
                       verify = "PASS*"
                else:
                    verify = "FAIL"
    else:
        verify = "N/A "
    #print verify
    #print '############'

    writ_dict[num_files]=ffind[1:-1]
    writ_dict[num_files]=verify

    '''
```

```
    print u'\u2713'
    ✓
    print u'\u2717'
    ✗
    '''

    return writ_dict

def find_hash(hm,h_val):

    hm.seek(0)

    offset_list = []

    content = hm.read()
    for line in content.splitlines():
        if h_val in line:
            for offset in
((line.split('\t')[1]).replace("[","")).replace("]","").split(","):
                #print offset.strip()
                offset_list.append(int(offset.strip())*512)
            break

    return offset_list


if __name__ == '__main__':

    #How many times the tool should run
    N = 1

    #Used as a counter
    n = 0

    #Repeats the code below N times
    for _ in itertools.repeat(None, N):

        #*****************************************
        #REQUIRED DECLARATIONS

        #Assigns the configuration file that shows where all of the
images
        #are located
        #config_file = "./Amazon/Config.txt"
        config_file = "./_StampTest_DISSER/Config.txt"
        #config_file = "./_RemSim/Machine_State_Physical/Config.txt"

        #*****************************************
        #NON_REQUIRED DECLARATIONS

        #image_name = 'image.gz'

        #directory = './_StampTest_DISSER/RESULTS/RUN_3/'
```

```
        #blkls_file =
'./_StampTest_DISSER/RESULTS/RUN_3/blkls_fedora19rem_DISSER.txt'

        #-----EDU STUFF-----
        #config_file = "./EDU/Config.txt"

        #image_name = 'Win_10_Mathmatica'

        #directory = './EDU/RESULTS/RUN_1/'

        #blkls_file = './EDU/RESULTS/RUN_2/blkls_Base_Fall_2017.txt'
        #-------------------

        #*****************************************
        #STRICTLY USED FOR TESTING!!!
        #Used to iterate through folders since using
        #all of config file as intended is too slow
        #for my computer.
        n+=1
        #Each folder to place the results in selected
        #by the value of n
        d = {}
        d.update({1: '0_Linux/', 2: '1_2003/', 3:
'2_2012/',4:'3_2016/'})
        #directory = directory + d[n]
        #Each config file used for each folder selected
        #by the value of n
        c = {}
        c.update({1: './Amazon/0_Linux_Config.txt', 2:
'./Amazon/1_2003_Config.txt', 3:
'./Amazon/2_2012_Config.txt',4:'./Amazon/3_2016_Config.txt'})
        #config_file = c[n]
        #*****************************************

        #Reads the config file for the images
        image_info,t_image_info = read_config(config_file)

        #Determines the correct directory the files should be placed
        #directory = result_dir()

        #just for testing to keep it straight for now
        #print IP_ADDRESS
        #print directory

        print datetime.datetime.now().time()

        #Creates the VM and collects the IP, user and pass
        #IP_ADDRESS,USERNAME,PASSWORD = Create_Reservation()
        #image_id,instance_id,IP_ADDRESS,USERNAME,PASSWORD =
AM_connect()

        print datetime.datetime.now().time()
```

```
        #Collects the image of the VM as a GZ file
        #image_name = dd_image(IP_ADDRESS,USERNAME,PASSWORD,directory)
        #image_name = AM_dd_image(IP_ADDRESS,USERNAME,directory)
        #image_name =
AM_blkls_image(IP_ADDRESS,USERNAME,PASSWORD,directory,image_id,instance
_id)

        print datetime.datetime.now().time()

        #print 'Deleting Reservation'

        #Deletes the VM reservation
        #Delete_Reservation()
        #AM_terminate(instance_id)

        print datetime.datetime.now().time()

        #Unzips the image file downloaded
        #unzip(directory,image_name)

        print datetime.datetime.now().time()

        #Uses blkls to create the unallocated file of the image
        blkls_file = blklscollect(directory,image_name)

        print datetime.datetime.now().time()

        print 'Writing Remanence to File'

        #Determines remanence and writes it to various files
        #read_image(key_list,blkls_file,
alloc_file,image_name,directory,raw_image,hash_map)
        read_image(blkls_file, image_name, directory, image_info,
t_image_info)

        print datetime.datetime.now().time()

        try:
            os.remove(blkls_file)
        except:
            pass
        try:
            os.remove(str(directory)+image_name.split('.')[0])
        except:
            pass

        print datetime.datetime.now().time()

        AM_Delete_Volumes()
```

Appendix F.6: The Report Generator Tool Used to Aggregate and Analyze the Remanence Files as well as Create the Graphical Depictions of Remanence Discovered in Various Forms

```
import sys
import os
from itertools import groupby, count
import matplotlib.pyplot as plt
import re
import matplotlib.ticker as ticker
import subprocess as sp
import collections
import fnmatch
import hashlib

#This changes the dictionary class
class Dictlist(dict):
    def __setitem__(self, key, value):
        try:
            self[key]
        except KeyError:
            super(Dictlist, self).__setitem__(key, [])
        self[key].append(value)

#This changes the dictionary class
class Dictset(dict):
    def __setitem__(self, key, value):
        try:
            self[key]
        except KeyError:
            super(Dictset, self).__setitem__(key, set())
        self[key].add(value)

class progressbar(object):
    '''Creates the progress bar object when called and displays the
progress
    given the percentage completed in the process.'''
    def __init__(self, finalcount, block_char='.'):
        self.finalcount = finalcount
        self.blockcount = 0
        self.block = block_char
        self.f = sys.stdout
        if not self.finalcount: return
        self.f.write('\n------------------ % Progress ----------------
---\n')
        self.f.write('    1    2    3    4    5    6    7    8    9
0\n')
        self.f.write('----0----0----0----0----0----0----0----0----0----
0\n')
    def progress(self, count):
        count = min(count, self.finalcount)
        if self.finalcount:
```

```
            percentcomplete = int(round(100.0*count/self.finalcount))
            if percentcomplete < 1: percentcomplete = 1
        else:
            percentcomplete=100
        blockcount = int(percentcomplete//2)
        if blockcount <= self.blockcount:
            return
        for i in range(self.blockcount, blockcount):
            self.f.write(self.block)
        self.f.flush()
        self.blockcount = blockcount
        if percentcomplete == 100:
            self.f.write("\n")

def group_offset(image_list,diff):
    '''Used to group all of the offset numbers so that the same file
    is not processed multiple times'''
    image_list_grouped = []
    image_list[:] = [x / diff for x in image_list]
    groups = groupby(image_list, key=lambda item, c=count():item-
next(c))
    tmp = [list(g) for k, g in groups]
    for t in tmp:
        image_list_grouped.append([t[0],t[-1]])

    image_list_grouped[:] = [[y * diff for y in x] for x in
image_list_grouped]

    return image_list_grouped

def allUnique(x):
    seen = set()
    return not any(i in seen or seen.add(i) for i in x)

def
blklsb_datapoints(image_file,blklsb_file_alloc,allocated_mapping_file,b
lklsb_file_unalloc,unallocated_mapping_file):

    blklsb_dict = {}
    alloc = 0
    unalloc = 0
    unalloc_512 = 0
    blklsb_alloc = collections.OrderedDict()
    #blklsb_unalloc = {}
    blklsb_unalloc = Dictlist()

    directory = '/'.join(blklsb_file_alloc.split('/')[:-1])

    fsstat_args = "fsstat -o 2048 "+directory+image_file #For Amazon
2008 Images

    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
```

```
    fsstat_proc = sp.Popen(fsstat_args, startupinfo=startupinfo,
stdout=sp.PIPE)
    fsstat_result = fsstat_proc.stdout.read()

    for line in fsstat_result.splitlines():
        if ("Cluster Size" in line) or ("Block Size" in line):
            cluster_size = int((line.split(":")[1]).strip())
    try:
        print 'The Cluster Size: ',cluster_size
    except:
        raise Exception('No Cluster Size Found!')


    #***********************************
    #This code is for the allocated space
    #***********************************

    #Creates the blocked allocated space file
    if os.path.isfile(blklsb_file_alloc):
        print 'Blklsb Allocated File Exists...'
    else:
        blklsb_file = blklsb_file_alloc
        blklsb_args = './blklsb.exe -a -o 2048 '+directory+image_file
#For Amazon 2008 Images
        #blklsb_args = './blklsb.exe -a '+directory+image_file #For
Fedora Images
        print 'Creating Blklsb Allocated File: ',blklsb_args

        with open(blklsb_file,'w+') as f:
            startupinfo = sp.STARTUPINFO()
            startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
            sp.Popen(blklsb_args, startupinfo=startupinfo, stdout = f,
shell=False).wait()

    #Creates the mapping file for allocated space
    if os.path.isfile(allocated_mapping_file):
        print 'Blklsb Allocated Mapping File Exists: Generating Alloc
Dictionary'
        with open(allocated_mapping_file,'rb') as amf:
            for am in amf:
                #blklsb_dict[alloc] = "Allocated"
                #print am.strip()
                cluster_loc = am.strip().split('\t')[0]
                byte_loc = am.strip().split('\t')[1]
                blklsb_alloc[str(cluster_loc)] = str(byte_loc)
                alloc+=1
    else:
        print 'Creating Blklsb Allocated Mapping File'
        with open(allocated_mapping_file,'wb') as amf,
open(blklsb_file_alloc,'rb') as bstat:
            for line in bstat:
                if line.startswith('write block'):
                    #print
(line.strip().split())[0].strip(),(line.strip().split())[2].strip()
```

```
                    #blklsb_dict[(line.strip().split())[2].strip()] =
"Allocated"

amf.write(str(alloc)+'\t'+str(int((line.strip().split())[2].strip())*cl
uster_size)+'\n')
                    blklsb_alloc[str(alloc)] =
str(int((line.strip().split())[2].strip())*cluster_size)
                    alloc+=1

    #**********************************
    #This code is for the unallocated space
    #**********************************

    #Creates the blocked unallocated space file
    if os.path.isfile(blklsb_file_unalloc):
        print 'Blklsb Unllocated File Exists...'
    else:
        blklsb_file = blklsb_file_unalloc
        blklsb_args = './blklsb.exe -A -o 2048 '+directory+image_file
#For Amazon 2008 Images
        #blklsb_args = './blklsb.exe -A '+directory+image_file #For
Fedora Images
        print 'Creating Blklsb Unllocated File: ',blklsb_args

        with open(blklsb_file,'w+') as f:
            startupinfo = sp.STARTUPINFO()
            startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW
            sp.Popen(blklsb_args, startupinfo=startupinfo, stdout = f,
shell=False).wait()

    #Creates the mapping file for unallocated space
    if os.path.isfile(unallocated_mapping_file):
        print 'Blklsb Unllocated Mapping File Exists: Generating
Unalloc Dictionary'
        with open(unallocated_mapping_file,'rb') as umf:
            for um in umf:
                #print um.strip()
                sector_loc = um.strip().split('\t')[0]
                byte_loc = um.strip().split('\t')[1]
                clust_loc = um.strip().split('\t')[2]
                blklsb_unalloc[str(sector_loc)] = str(byte_loc)
                blklsb_unalloc[str(sector_loc)] = str(clust_loc)
                unalloc_512+=1

        unalloc = unalloc_512 / 8
    else:
        print 'Creating Blklsb Unllocated Mapping File'
        with open(unallocated_mapping_file,'wb') as umf,
open(blklsb_file_unalloc,'rb') as bstat:
            for line in bstat:
                if line.startswith('write block'):
                    #print
(line.strip().split())[0].strip(),(line.strip().split())[2].strip()
```

```
                    #blklsb_dict[(line.strip().split())[2].strip()] =
"Not"
                    clust_loc = 1
                    for u in range(8):

umf.write(str(unalloc_512)+'\t'+str((int((line.strip().split())[2].stri
p())*cluster_size)+(u*512))+'\t'+str(clust_loc)+'\n')
                        blklsb_unalloc[str(unalloc_512)] =
str((int((line.strip().split())[2].strip())*cluster_size)+(u*512))
                        blklsb_unalloc[str(unalloc_512)] =
str(clust_loc)
                        unalloc_512+=1
                        clust_loc+=1
                    unalloc+=1

    print 'blklsb_alloc [clusters]',len(blklsb_alloc)
    print 'blklsb_unalloc [sectors]',len(blklsb_unalloc)

    #for (k,v), (k2,v2) in zip(blklsb_alloc.items(),
blklsb_unalloc.items()):
        #print 'alloc',k,type(k),v,type(v)
        #print 'unalloc',k2,type(k2),v2,type(v2)
        #print 'unalloc',k2,v2

    print "Size of
Allocated:",alloc,"[Clusters]",alloc*cluster_size,"[Bytes]"
    print "Size of
Unallocated:",unalloc,"[Clusters]",unalloc_512,"[Sectors]",unalloc*clus
ter_size,"[Bytes]"
    print "Size of
Blklsb:",alloc+unalloc,"[Clusters]",(alloc+unalloc)*cluster_size,"[Byte
s]"

    return blklsb_alloc,blklsb_unalloc


def locations(directory,rem_file,blklsb_unalloc):

    alloc_list = []
    alloc_list_grouped = []

    SMR_unalloc_clus_loc = []
    SUR_unalloc_clus_loc = []

    offsets_found = set()
    image_list = Dictlist()
    image_list_Dict = Dictlist()
    del_list=[]
    img_isolated = ''
    img_isolated = 'Windows_Server_2012R2'
    #img_isolated = 'ubuntu17_DISSER'
    #img_isolated = 'win10_DISSER'
    #img_isolated = 'win7_DISSER'
    #img_isolated = 'fedora19_DISSER'
```

```
    #img_isolated = 'SUR'
    print img_isolated
    #img_deleted = 'ubuntu17_DISSER'
    #img_deleted = 'SUR'
    img_deleted = ''
    print img_deleted
    files_set = Dictset()
    #files_total = Dictset()
    files_total = collections.OrderedDict()
    files_total_list = []
    files_dict_list = Dictlist()
    files_dict_list_grouped = Dictlist()
    CRE_File_List = {}
    #This is for the new method of tracking hashes in file
    files_hash_tracker = collections.OrderedDict()
    files_hash_tracker_dictlist = Dictlist()

    for r in rem_file:
        with open(directory+r,'rb') as f:
            first_line = (f.readline()).strip()
            print first_line
            if first_line == '*****************************REMANENCE
INFORMATION*****************************':
                seg_rem =
(f.read()).split('*****************************REMANENCE
INFORMATION*****************************')
                for s in seg_rem:
                    if (('Verified:PASS' in s) and ('\tPASS' in s)):
                        lines = s.splitlines()
                        for line in lines:
                            if line.startswith('Image:'):

offsets_found.add(int((line.strip().split()[2]).split(':')[1]))
                                image_name =
((line.strip().split()[1]).split('/'))[-2]
                                #print
(line.strip().split()[1]).split('/')[2],
int((line.strip().split()[2]).split(':')[1])
                                #image_list[image_name] =
int((line.strip().split()[2]).split(':')[1])
                                image_list[image_name] =
int(blklsb_unalloc[str(int((line.strip().split()[2]).split(':')[1])/512
)][0])
                                #print
(blklsb_unalloc[str(int((line.strip().split()[2]).split(':')[1])/512)][
1])
                                #print
int((line.strip().split()[2]).split(':')[1]),int(blklsb_unalloc[str(int
((line.strip().split()[2]).split(':')[1])/512)][0])
                                offset_save =
int(blklsb_unalloc[str(int((line.strip().split()[2]).split(':')[1])/512
)][0])
```

```
SMR_unalloc_clus_loc.append((blklsb_unalloc[str(int((line.strip().split
()[2]).split(':')[1])/512)][1]))
                                elif ('Hash:' in line):
                                    hash_val_save =
(line.strip().split()[1]).strip()
                                    #print hash_val_save
                                elif ('\tPASS' in line):
                                    files_total[offset_save] =
((line.split())[-1]).split('/')[-1]
                                    files_set[image_name] =
((line.split())[-1]).split('/')[-1]
                                    files_hash_tracker[offset_save] =
hash_val_save

files_hash_tracker_dictlist[((line.split())[-1]).split('/')[-1]] =
hash_val_save
                                    #print offset_save,hash_val_save
                                    #print ((line.split())[-
1]).split('/')[-1]
            elif first_line == '**************************DATA OF
INTEREST INFORMATION**********************':
                seg_rem =
(f.read()).split('**************************DATA OF INTEREST
INFORMATION**********************')
                for s in seg_rem:
                    lines = s.splitlines()
                    for line in lines:
                        if line.startswith('Offset:'):

offsets_found.add(int((line.strip().split()[0]).split(':')[1]))
                            #print
(line.strip().split()[1]).split('/')[2],
int((line.strip().split()[2]).split(':')[1])
                            #image_list['SUR'] =
int((line.strip().split()[0]).split(':')[1])
                            image_list['SUR'] =
int(blklsb_unalloc[str(int((line.strip().split()[0]).split(':')[1])/512
)][0])

SUR_unalloc_clus_loc.append(blklsb_unalloc[str(int((line.strip().split(
)[0]).split(':')[1])/512)][1])

    #Used to show CRE file names and that they are in order
    #Also placed into a Dictlist to then group by file name to graph
later
    print '*****************'
    print "Files Found with Offsets"
    for k,v in files_total.items():
        files_total_list.append(k)
        files_dict_list[v] = k
        #print k,'\t',v
    #print files_dict_list
```

```
    #This part takes the inidividual lists and grouped the values for
plotting
    for k,v in files_dict_list.items():
        files_dict_list_grouped[k] = group_offset(v,512)
    #print files_dict_list_grouped

    print '*****************'

    for i in image_list:
        CRE = 0
        print "Number of Remanence Found[",i,"]: ",len(image_list[i])
        #print i,image_list[i]
        image_list_grouped = group_offset(image_list[i],512)
        for j in image_list_grouped:
            image_list_Dict[i] = j
        #print image_list_Dict[i]
        for k in image_list_Dict[i]:
            if int(k[0])-int(k[-1]) != 0:
                CRE+=1
                if i != 'SUR' and files_total[k[0]] == files_total[k[-
1]]:
                    CRE_File_List[k[0]] = files_total[k[0]]
                    #prints the first and last offset of CRE with
filename
                    #print k[0],k[-1],files_total[k[0]]
        print "Number of CRE[",i,"]: ",CRE
        print '*****************'
    print 'CRE w/ File Matching[ALL IMAGES]:',len(CRE_File_List)
    print '*****************'

    for (k,v) in blklsb_alloc.items():
        alloc_list.append(int(v))

    alloc_list_grouped = group_offset(alloc_list,4096)

    #print 'LIST',len(alloc_list_grouped)
    #for l in alloc_list_grouped:
    #    print l

    #Used to delete all but a single key in dictionary
    #Used to isolate remanence locations
    if img_isolated:
        print "ISOLATING TARGETED IMAGE"
        for x in image_list_Dict:
            if x != img_isolated:
                del_list.append(x)
        for y in del_list:
            del image_list_Dict[y]

    #Used to delete one single key in dictionary
    #Used to isolate remanence locations
    if img_deleted:
        print "DELETING TARGETED IMAGE"
        del image_list_Dict[img_deleted]
```

```
    print
'*********************************************************************
********'
    for z in files_set:
        print "Files Found in[",z,"]: ",",".join(files_set[z])
    print
'*********************************************************************
********'

    print 'Offsets Found:',len(offsets_found)

    #print image_list_Dict

    return
image_list_Dict,alloc_list_grouped,SMR_unalloc_clus_loc,SUR_unalloc_clu
s_loc,files_dict_list_grouped,files_hash_tracker,files_hash_tracker_dic
tlist

def unalloc_cluster_graph(SMR_unalloc_clus_loc,SUR_unalloc_clus_loc):

    #print 'Length of Unallocated Cluster
Locations',len(unalloc_clus_loc)

    SMRcounter=collections.Counter(SMR_unalloc_clus_loc)
    print 'Frequency of SMR Cluster Locations',SMRcounter

    SURcounter=collections.Counter(SUR_unalloc_clus_loc)
    print 'Frequency of SUR Cluster Locations',SURcounter

    return


def
hd_plot(image_size,blkls_size,image_list_Dict,image_file,all_drive,allo
c_list_grouped):

    image_file = (image_file.split('/')[-1]).split('.')[0]

    dict_list_count = 0
    count = 0

    colors = ['red','green','purple','yellow','brown','orange']
    #colors =
{'ubuntu64stamp_DISSER':'red','win7stamp_DISSER':'green','win10stamp_DI
SSER':'purple','fedora19stamp_DISSER':'orange'}
    #colors =
{'ubuntu17_DISSER':'red','win7_DISSER':'green','win10_DISSER':'purple',
'fedora19_DISSER':'orange'}
    #keep track of the colors used
    c = 0
    #used for no duplicate labels
    l = True
    b = True
```

```
    fig, ax = plt.subplots()
    ax.set_aspect('equal')
    #fig = plt.gcf()
    #fig = plt.figure(1)

    #size of allocated
    allocated_size = image_size - blkls_size
    #print allocated_size
    #print allocated_size
    ax.plot(alpha=0)

    if all_drive == True:
        for al in alloc_list_grouped:
            if b:
                ax.axvspan(al[0], al[1]+4096, alpha=.5, color='blue',
linewidth=0,label='Alloc: '+image_file.split('.')[0])
                b = False
            else:
                ax.axvspan(al[0], al[1]+4096, alpha=.5, color='blue',
linewidth=0)
        #allocated space shaded, uncomment to put back in
        #ax.axvspan(0, allocated_size, alpha=.5,
color='blue',label='Alloc: '+image_file[:-7])
        #Use this set for the whole disk drive
        plt.ylim(0,image_size/6.8)
        plt.xlim(0, image_size)

    elif all_drive == False:
        #Use this set for only the blkls drive
        plt.ylim(0,blkls_size/4)
        plt.xlim(allocated_size, image_size)

    for x in image_list_Dict:
        dict_list_count = len(image_list_Dict[x]) + dict_list_count

    pb = progressbar(100,"*")

    for x in image_list_Dict:
        #print colors[c],c
        l = True
        for y in image_list_Dict[x]:
            #print colors[c]
            #print y[0],y[1]
            #allocated size added for offsetting for image size view
            if l:
                if x != 'SUR':
                    #Use this one for the entire disk drive
                    ax.axvspan(y[0], y[1], alpha=.5,
color=colors[c],label=x)
                    #ax.axvspan(y[0], y[1], alpha=.5,
color=colors[x],label=x[:-7])
                    l=False
                else:
```

```
                    c+=-1
                    ax.axvspan(y[0], y[1], alpha=.5,
color='black',label=x)
                    l=False
            else:
                if x != 'SUR':
                    #Use this one for the entire disk drive
                    ax.axvspan(y[0], y[1], alpha=.5, color=colors[c])
                    #ax.axvspan(y[0], y[1], alpha=.5, color=colors[x])
                else:
                    ax.axvspan(y[0], y[1], alpha=.5, color='black')
            count+=1
            percent = ((float(count))/dict_list_count)*100
            pb.progress(percent)
        c+=1

    plt.axis()

    lgd = ax.legend(loc=9, bbox_to_anchor=(0.5,-0.02))

    #ax.set_xticks([])
    ax.set_yticks([])

    #Used to scale it to view as GB instead of Bytes
    scale_x = 1e9
    ticks_x = ticker.FuncFormatter(lambda x, pos:
'{0:g}'.format(x/scale_x))
    ax.xaxis.set_major_formatter(ticks_x)

    ax.set_xlabel("GB")

    print '\nPLOTTING GRAPH'

    plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, borderaxespad=0.)
    fig.savefig('samplefigure', bbox_inches='tight')

    plt.show()

    return

def
CRE_File_Info(directory,files_dict_list_grouped,files_hash_tracker,file
s_hash_tracker_dictlist,image_file):

    print '*****************************'
    print '*****GRANULAR CRE CHECKS*****'
    print '\t',(directory.split('/'))[-2]
    print '*****************************'

    image_file = directory+image_file

    directory = directory+'Files/'
    #print directory
```

```
    shared_items_dict = {}
    file_offset_mapping_2_dict = Dictlist()
    file_size_dict = {}

    recovered_file = Dictlist()
    file_offset_mapping_2 = Dictlist()
    percent_dict = Dictlist()

    with open(image_file,'rb') as img_f:

        for cre_file in files_dict_list_grouped:

            print 'Checks for ',cre_file
            dict_list_count = 0
            count = 0

            hash_list = collections.OrderedDict()
            #Determines how much of the file has been recovered to
calculate percentage
            amount_recovered = 0
            #Make new list for checking if order is sequential when
compared to original file
            seq_check = []

            #How many 512 byte chunks were found
            pieces = 0

            #Maps the offset within the image to the offset within the
file
            file_offset_mapping = {}

            file_list = []
            shared_items_list = []
            shared_items_list_grouped = []
            file_offset_mapping_2_list = []
            file_offset_mapping_2_grouped = []

            #Makes sure that hash values are only counted once
            duplicates = set()
            duplicates_check = []

            orig_file_hash_list = []

            #Keeps track on if the theoretical discovered values are
cluster aligned
            cluster_aligned = set()

            #List of hashes for the extracted file to see if it matches
what was theoretically found
            extracted_file_dict = collections.OrderedDict()

            #for file in glob.glob(directory+'*'+cre_file):
```

```
            file_list = [file for file in os.listdir(directory) if
fnmatch.fnmatch(file, '*'+cre_file)]
            for fnd_file in file_list:
                if os.stat(directory+fnd_file).st_size !=0:
                    cre_file_orig = directory+fnd_file

            print "\tOriginal CRE File: ",cre_file_orig
            file_size = os.stat(cre_file_orig).st_size
            print "\tFile Size: ",file_size

            file_size_dict[cre_file] = file_size

            cre_offsets = files_dict_list_grouped[cre_file]
            #print cre_offsets

            k=0

            BLOCKSIZE = 512

            with open(cre_file_orig, 'rb+') as fp:
                for block in iter(lambda: fp.read(BLOCKSIZE), ''):
                    #Calculates the MD5 hash of the slack
                    hash_object = hashlib.md5(block)
                    hashed_block = hash_object.hexdigest()
                    #print k, hashed_block
                    hash_list[k] = hashed_block
                    if hashed_block in
files_hash_tracker_dictlist[cre_file] and hashed_block not in
duplicates:
                        file_offset_mapping_2[cre_file]=k*512
                        duplicates_check.append(hashed_block)
                        duplicates.add(hashed_block)
                        #print cre_file,hashed_block,k
                    try:
                        reverse_hash =
files_hash_tracker.keys()[files_hash_tracker.values().index(hashed_bloc
k)]
                        reverse_location =
files_hash_tracker.keys().index(reverse_hash)
                        #print
reverse_location,":",k*512,"\t",reverse_hash,hashed_block
                        file_offset_mapping[reverse_hash]=k*512
                        seq_check.append(reverse_location)
                    except:
                        pass
                    k+=1

            #print len(hash_list)*512

            for cre_o in cre_offsets[0]:
                #Needed a way to deal with the image offset vice file
offset
                #so created a mapping dictionary that is called
                start = file_offset_mapping[cre_o[0]]
```

```
                end = file_offset_mapping[cre_o[1]]+512
                amount_recovered = amount_recovered + (end - start)
                pieces = pieces + ((end - start)/512)
                recovered_file[cre_file] = [start,end]

            print "\tNo Duplicates Found:",allUnique(duplicates_check)

            print "\tSuccessfully Recovered", amount_recovered,
"bytes(",pieces,'parts,',
round((float(amount_recovered)/file_size)*100,2),"%) of",cre_file
            #Determine if hashes were found sequentially, meaning that
the file in the correct order
            seq_check_group = group_offset(seq_check,1)[0]
            is_list_sequential = (seq_check[0]==seq_check_group[0]) and
(seq_check[-1]==seq_check_group[1])
            print "\tSequential Check:",is_list_sequential
            #print seq_check
            #Determine the theoretical offset in the raw image as to
where the entire file should be located
            theor_start = min(file_offset_mapping)-
file_offset_mapping[min(file_offset_mapping)]
            theor_end = theor_start + file_size + 512 #The 512
subtraction might not be neccesary.Depends on loop.
            print "\tStart of offset:",theor_start,"End of
offset:",theor_end

            #Seeking to the theoretical start of the file recovered if
still found on disk
            #The offset value '1048576' was used since the entire disk
image was used instead of just the logical image
            #This value was discovered manually through winhex
            img_f.seek(theor_start+1048576)

            #extracted_file = img_f.read(file_size)
            #print 'length of extracted_file',len(extracted_file)

            kk = 0
            break_num = file_size / 512
            extracted_file_raw = directory+'_EFR_'+cre_file
            orig_file_hash_list = hash_list.values()

            #with open(extracted_file_raw,'wb') as efr:
            for block in iter(lambda: img_f.read(BLOCKSIZE), ''):
                efr_k = 0
                #Calculates the MD5 hash of the slack
                hash_object = hashlib.md5(block)
                hashed_block = hash_object.hexdigest()
                #if hashed_block in orig_file_hash_list:
                #    efr.write(block)
                #else:
                #    while (efr_k < 512):
                #        efr.write(b'\x00\x00\x00\x00\x00\x00\x00\x00')
                #        efr_k+=8
                #print kk, hashed_block
```

```
                extracted_file_dict[kk] = hashed_block
                kk+=1
                if kk == break_num:
                    break

            #print '\tkk values: ',kk,kk*512

            shared_items = {k: extracted_file_dict[k] for k in
extracted_file_dict if k in hash_list and extracted_file_dict[k] ==
hash_list[k]}
            print '\tOriginal File Matches:',len(shared_items),'out
of',kk,'parts (',round((len(shared_items)/float(kk))*100,2),'%)'

percent_dict[cre_file]=round((float(amount_recovered)/file_size)*100,2)

percent_dict[cre_file]=round((len(shared_items)/float(kk))*100,2)
            #for key,val in hash_list.items():
            #    print key,'\t',val,'\t',extracted_file_dict[key]

            img_f.seek(0)

            for k,v in shared_items.items():
                shared_items_list.append(k*512)
            shared_items_list.sort()
            shared_items_list_grouped =
group_offset(shared_items_list,512)
            shared_items_dict[cre_file] = shared_items_list_grouped

            #print shared_items_list

            #This checks to make sure that all remanence found is
cluster aligned
            for k in shared_items_list_grouped:
                if k[0]%4096 == 0 and (k[1]+512)%4096 == 0:
                    cluster_aligned.add('True')
                else:
                    cluster_aligned.add('False')
            if 'False' not in cluster_aligned:
                print "\tTheoretical Cluster Aligned: True"
            else:
                print "\tTheoretical Cluster Aligned: False"

            print '*****************************'

        #This puts the remanence found in grouped lists for graphing
        for k,v in file_offset_mapping_2.items():
            v.sort()
            file_offset_mapping_2_dict[k] = group_offset(v,512)



    '''
    for k,v in file_offset_mapping_2_dict.items():
```

```
            for x in v:
                print k,'\t',x

    '''

    #for k,v in shared_items_dict.items():
    #    for x in v:
    #        print k,'\t',x



    return
file_offset_mapping_2_dict,shared_items_dict,file_size_dict,percent_dic
t

def
CRE_hd_plot(directory,file_offset_mapping_2_dict,shared_items_dict,file
_size_dict,percent_dict):

    plots = len(percent_dict)

    #fig = plt.figure()

    for idx,cre_file in enumerate(percent_dict):

        file_size = file_size_dict[cre_file]

        rem_percent = (percent_dict[cre_file])[0]
        theor_percent = (percent_dict[cre_file])[1]

        colors =
['blue','red','green','purple','yellow','brown','orange']
        #colors =
{'ubuntu64stamp_DISSER':'red','win7stamp_DISSER':'green','win10stamp_DI
SSER':'purple','fedora19stamp_DISSER':'orange'}
        #colors =
{'ubuntu17_DISSER':'red','win7_DISSER':'green','win10_DISSER':'purple',
'fedora19_DISSER':'orange'}
        #keep track of the colors used
        c = 0
        #used for no duplicate labels
        b = True
        l = True

        #print len(hash_list)*512
        #ax=plt.subplot(plots,1,idx+1)
        fig, ax = plt.subplots()

        ax.set_aspect('equal')
        #fig = plt.gcf()
        #fig = plt.figure(1)

        ax.plot(alpha=0)
```

```
        for shr_o in shared_items_dict[cre_file]:
            start = shr_o[0]
            end = shr_o[1]+512
            if l:
                ax.axvspan(start, end, alpha=.5,
color='red',label='Theoretical File ['+str(theor_percent)+'%]')
                l = False
            else:
                ax.axvspan(start, end, alpha=.5, color='red')

        for cre_list in file_offset_mapping_2_dict[cre_file]:
            for cre_o in cre_list:
                start = cre_o[0]
                end = cre_o[1]+512
                if b:
                    ax.axvspan(start, end, alpha=.75,
color='blue',label='Remanence ['+str(rem_percent)+'%]')
                    b = False
                else:
                    ax.axvspan(start, end, alpha=.75, color='blue')

        #allocated space shaded, uncomment to put back in
        #ax.axvspan(0, allocated_size, alpha=.5,
color='blue',label='Alloc: '+image_file[:-7])
        #Use this set for the whole disk drive
        plt.ylim(0,file_size/6.8)
        plt.xlim(0, file_size)

        plt.axis()

        lgd = ax.legend(loc=9, title = cre_file, bbox_to_anchor=(0.5,-
0.02))

        #ax.set_xticks([])
        ax.set_yticks([])

        '''

        if file_size >= 1000000:
            #Used to scale it to view as MB instead of Bytes
            scale_x = 1e6 #For MB
            ax.set_xlabel("MB")
            ticks_x = ticker.FuncFormatter(lambda x, pos:
'{0:g}'.format(x/scale_x))
            ax.xaxis.set_major_formatter(ticks_x)
            print 'MB',cre_file
        elif 1000 <= file_size < 1000000:
            scale_x = 1e3 #For KB
            ax.set_xlabel("KB")
            ticks_x = ticker.FuncFormatter(lambda x, pos:
'{0:g}'.format(x/scale_x))
            ax.xaxis.set_major_formatter(ticks_x)
            print 'KB',cre_file
        else:
```

```
            scale_x = 1
            ax.set_xlabel("B")
            ticks_x = ticker.FuncFormatter(lambda x, pos:
'{0:g}'.format(x/scale_x))
            ax.xaxis.set_major_formatter(ticks_x)
            print 'B',cre_file

        '''

        scale_x = 1e3 #For KB
        ax.set_xlabel("KB")
        #scale_x = 1e6 #For MB
        #ax.set_xlabel("MB")
        ticks_x = ticker.FuncFormatter(lambda x, pos:
'{0:g}'.format(x/scale_x))
        ax.xaxis.set_major_formatter(ticks_x)



        print '\nPLOTTING GRAPH'

        plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
               ncol=1, title = cre_file, borderaxespad=0.)
        fig.savefig('samplefigure', bbox_inches='tight')

    plt.show()

    return

def ni_file_find():

    file_set = set()
    file_set_2 = set()
    diff_set = set()
    diff_set_2 = set()
    diff_set_sym = set()

    #Files found in both images
    sim_set = set()
    #File found in both images, but different hash values
    sim_set_diff = set()

    file_dict = {}
    file_dict_2 = {}
    diff_dict = {}
    diff_dict_2 = {}
    diff_dict_sym = {}

    #image_file = './_StampTest_DISSER/win7_DISSER/win7_DISSER'
    #image_file =
'./Amazon_NEW/Windows_Server_2016/Windows_Server_2016'
    #image_file =
'./Amazon_NEW/Windows_Server_2012R2/Windows_Server_2012R2'
```

```
    #image_file =
'./Amazon_NEW/Windows_Server_2008R2/Windows_Server_2008R2'
    #image_file = './Amazon_NEW/RESULTS/RUN_1/2008R2.001'
    #image_file = 'K:/Windows_Server_2012.001'

    image_file =
'./Amazon_NEW/Windows_Server_2008R2/Windows_Server_2008R2'
    image_file_2 = './Amazon_NEW/RESULTS/RUN_8/2008R2.001'


    #search_for =
['SMDiagnostics.ni.dll','PresentationFramework.ni.dll','System.Xml.ni.d
ll','System.ServiceModel.Channels.ni.dll','System.Runtime.Remoting.ni.d
ll','energy-report-','XsdBuildTask.ni.dll']
    search_for = ['energy-report-']

    startupinfo = sp.STARTUPINFO()
    startupinfo.dwFlags |= sp.STARTF_USESHOWWINDOW

    flsstat_args = "fls -Fhrp -m ./ "+image_file
    print flsstat_args

    flsstat_proc = sp.Popen(flsstat_args, startupinfo=startupinfo,
stdout=sp.PIPE)
    flsstat_result = flsstat_proc.stdout.read()

    for line in flsstat_result.splitlines():
        #if any(s in line for s in search_for) and (('(deleted)' and
'($FILE_NAME)' and '($') not in line):
        #    print image_file.split('/')[-1],line.split('|')[1]
        #file_set.add((line.strip()).split()[-1])
        if ('(deleted)' and '($FILE_NAME)' and '($') not in line:
            file_dict[line.split('|')[1]]=line.split('|')[0]
            #print line

    #for k,v in file_dict.items():
        #print k,'\t',v

    #print len(file_dict)


    flsstat_args = "fls -Fhrp -m ./ -o 2048 "+image_file_2
    print flsstat_args

    flsstat_proc = sp.Popen(flsstat_args, startupinfo=startupinfo,
stdout=sp.PIPE)
    flsstat_result = flsstat_proc.stdout.read()

    for line in flsstat_result.splitlines():
        #if any(s in line for s in search_for) and (('(deleted)' and
'($FILE_NAME)' and '($') not in line):
        #    print image_file_2.split('/')[-1],line.split('|')[1]
        #file_set.add((line.strip()).split()[-1])
        if ('(deleted)' and '($FILE_NAME)' and '($') not in line:
```

```
            file_dict_2[line.split('|')[1]]=line.split('|')[0]
            #print line

    #for k,v in file_dict_2.items():
        #print k,'\t',v

    print 'Files found in',image_file.split('/')[-1],':',len(file_dict)
    print 'Files found in',image_file_2.split('/')[-
1],':',len(file_dict_2)

    print image_file.split('/')[-1],':',len(set(file_dict.keys()) -
set(file_dict_2.keys())),' files not found in
',image_file_2.split('/')[-1]
    print image_file_2.split('/')[-1],':',len(set(file_dict_2.keys()) -
set(file_dict.keys())),' files not found in ',image_file.split('/')[-1]

    shared_items = {k: file_dict[k] for k in file_dict if k in
file_dict_2 and file_dict[k] == file_dict_2[k]}
    diff_items = {k: file_dict[k] for k in file_dict if k in
file_dict_2 and file_dict[k] != file_dict_2[k]}

    print 'Similar Files Found:',len(shared_items)
    print 'Different Files Found:',len(diff_items)

    for k in diff_items:
        print k
    #    if k in file_dict_2 and file_dict[k] != file_dict_2[k]:
    #        print
k.strip(),'\t',file_dict[k].strip(),'\t',file_dict_2[k].strip()

    '''


    flsstat_proc = sp.Popen(flsstat_args, startupinfo=startupinfo,
stdout=sp.PIPE)
    flsstat_result = flsstat_proc.stdout.read()

    for line in flsstat_result.splitlines():
        #if any(s in line for s in search_for):
        file_set_2.add((line.strip()).split()[-1])
        #print line

    diff_set = file_set - file_set_2
    diff_set_2 = file_set_2 - file_set
    diff_set_sym = file_set.symmetric_difference(file_set_2)

    print image_file,len(file_set),len(diff_set)
    print image_file_2,len(file_set_2),len(diff_set_2)
    print len(diff_set_sym)

    for x in diff_set_2:
        if any(s in line for s in search_for):
            print x
```

```
    sim_set = file_set & file_set_2
    total_sim = len(sim_set)
    print total_sim
    '''

    return

if __name__ == '__main__':

    #matplotlib.use('GTKAgg')

    directory = './Amazon_NEW/RESULTS/RUN_9/'
    print directory

    rem_file = ['remanence_found.txt','reverse_remanence_found.txt']

    blkls_file = 'blkls_image.txt'

    image_file = './2008R2.001'


#**********************************************************************
********************
    #*********************************BLKLSB
STUFF*****************************************

#**********************************************************************
********************

    #blklsb_file_alloc =
'Z:/DISSERTATION/Amazon_NEW/RESULTS/RUN_7/blklsb_ALLOC.txt'
    blklsb_file_alloc = directory+'blklsb_ALLOC.txt'

    #blklsb_file_unalloc =
'Z:/DISSERTATION/Amazon_NEW/RESULTS/RUN_7/blklsb_UNALLOC.txt'
    blklsb_file_unalloc = directory+'blklsb_UNALLOC.txt'

    #allocated_mapping_file =
'Z:/DISSERTATION/Amazon_NEW/RESULTS/RUN_7/blklsb_ALLOC_map.txt'
    allocated_mapping_file = directory+'blklsb_ALLOC_map.txt'

    #unallocated_mapping_file =
'Z:/DISSERTATION/Amazon_NEW/RESULTS/RUN_7/blklsb_UNALLOC_map.txt'
    unallocated_mapping_file = directory+'blklsb_UNALLOC_map.txt'


#**********************************************************************
********************

    blkls_size = os.stat(directory+blkls_file).st_size
    #blkls_size = os.stat(blkls_file).st_size
    #print blkls_size
    #image_size = os.stat(image_file).st_size
    image_size = os.stat(directory+image_file).st_size
```

```
    #print image_size


    #Creates the two dictionarys for allocated and unallocated space
    #blklsb_alloc: The key is the cluster location and the value is the
sector location/byte offset
    #blklsb_unalloc: The key is the blkls file offset and the value is
the absolute offset in the image
    blklsb_alloc,blklsb_unalloc =
blklsb_datapoints(image_file,blklsb_file_alloc,allocated_mapping_file,b
lklsb_file_unalloc,unallocated_mapping_file)


image_list_Dict,alloc_list_grouped,SMR_unalloc_clus_loc,SUR_unalloc_clu
s_loc,files_dict_list_grouped,files_hash_tracker,files_hash_tracker_dic
tlist = locations(directory,rem_file,blklsb_unalloc)
    #print len(image_list_Dict)
    #print files_hash_tracker

    ###unalloc_cluster_graph(SMR_unalloc_clus_loc,SUR_unalloc_clus_loc)

    #Select True or False.  True means print whole drive
    #False means to brink only unallocated space
    all_drive = True


file_offset_mapping_2_dict,shared_items_dict,file_size_dict,percent_dic
t =
CRE_File_Info(directory,files_dict_list_grouped,files_hash_tracker,file
s_hash_tracker_dictlist,image_file)


CRE_hd_plot(directory,file_offset_mapping_2_dict,shared_items_dict,file
_size_dict,percent_dict)

    #ni_file_find()
```

REFERENCES

AlBelooshi, B., Salah, K., Martin, T., & Damiani, E. (2015). Experimental Proof: Data Remanence in Cloud VMs. *IEEE 8th International Conference on Cloud Computing*, 1017-1020.

AlBelooshi, B., Salah, K., Martin, T., & Damiani, E. (2015). Inspection and Deconfliction of Published Virtual Machine Templates' Remnant Data for Improved Assurance in Public Clouds. *IEEE*.

Alherbawi, N., Shukur, Z., & Sulaiman, R. (2016). A Survey on Data Carving in Digital Forensics. *Asian Journal of Information Technology 15*, 5137-5144.

Allen, B. (2014). hashdb. Retrieved from https://github.com/simsong/hashdb.git

Amazon. (2017). Amazon Web Services: Overview of Security Processes. Retrieved from https://d1.awsstatic.com/whitepapers/Security/AWS_Security_Whitepaper.pdf

Amazon. (2018). *Amazon EC2*. Retrieved from Amazon: https://aws.amazon.com/ec2/

Amazon. (2018). *Amazon Machine Images (AMI)*. Retrieved March 06, 2018, from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

Amazon. (2019). *Amazon Elastic Compute Cloud User Guide for Windows Instances.* Retrieved from https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/ec2-wg.pdf

Amazon Web Services. (n.d.). *AWS SDK for Python (Boto3)*. Retrieved from https://aws.amazon.com: https://aws.amazon.com/sdk-for-python/

Barantsev, A. (n.d.). *SeleniumHQ*. Retrieved from http://github.com: https://github.com/SeleniumHQ/Selenium

Bayramusta, M., & Nasir, V. A. (2016). A fad or future of IT?: A comprehensive literative review on the cloud computing research. *International Journal of Information Management*, 635-644.

Bugiel, S., Numberger, S., Poppelmann, T., Sadeghi, A.-R., & Schneider, T. (2011). AmazonIA: When Elasticity Snaps Back. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, (pp. 389-400). New York, NY.

Carrier, B. (n.d.). *Open Source Digital Forensics*. Retrieved from www.sleuthkit.org: http://www.sleuthkit.org/

Chandramouli, R. (2018, January). *Security Recomendations for Hypervisor Deployment on Servers*. Retrieved from http://nvlpubs.nist.gov: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-125A.pdf

Chen, L., & Wang, G. (2008). An Efficient Peicewise Hashing Method for Computer Forensics. *IEEE Computer Society Workshop on Knowledge Discovery and Data Mining*, 635-638.

Collange, S., Daumas, M., Dandass, Y., & Defour, D. (2009). Using graphics processors for parallelizing hash-based data carving. *Proceedings of the 42nd Hawaii International Conference on System Sciences.*

Corbin, K. (2015, OCT 1). *Government cloud adoption efforts lag as security concerns persist*. Retrieved from www.cio.com: http://www.cio.com/article/2988495/cloud-computing/government-cloud-adoption-efforts-lag-as-security-concerns-persist.html

Defense Information Systems Agency. (2017, January 06). *VMware ESXi Server 5.0 Security Technical Implementation Guide*. Retrieved from https://www.stigviewer.com: https://www.stigviewer.com/stig/vmware_esxi_server_5.0/2017-01-06/

DevTiw. (2017, 11 01). *Azure Disk Encryption for Windows and Linux IaaS VMs*. Retrieved from https://docs.microsoft.com/en-us/azure/security/azure-security-disk-encryption

Durumeric, Z., Li, F., Amann, J., Beekman, J., Payer, M., & Halderman, J. A. (2014). The Matter of Heartbleed. *Internet Measurement Conference* (pp. 475-488). ACM.

Gallagher, P. R. (1991). *A Guide to Understanding Data Remanence in Automated Information Systems.*

Garfinkel, S. L. (2013). Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security 32*, 56-72.

Garfinkel, S. L., & McCarrin, M. (2015). Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigation, 14*, S95-S105.

Garfinkel, S. L., & Shelat, A. (2003). Remembrance of Data Passed: A Study of Disk Sanitization Practices. *IEEE Security & Privacy*, 17-27.

Gemalto. (2017). *2017 The Year of Internal Threats and Accidental Data Breaches.* Gemalto. Retrieved from https://breachlevelindex.com/assets/Breach-Level-Index-Report-2017-Gemalto.pdf

Gupta, P., Seetharaman, A., & Raj, J. R. (2013). The usage and adoption of cloud computing by small and medium businesses. *International Journal of Information Management*, 861-874.

Harbour, N. (2002). *Dcfldd.* Retrieved from Defense Computer Forensics Lab: http://dcfldd.sourceforge.net/

Kelsey, J., Schneier, B., Wagner, D., & Hall, C. (1998). Side Channel Cryptanalysis of Product Ciphers. *In European Symposium on Research in Computer Security* (pp. 97-110). Berlin, Heidelberg: Springer.

Kent, K., Chevalier, S., Grance, T., & Dang, H. (2006, August). Guide to Integrating Forensic Techniques into Incident Response. Gaithersburg, MD.

Kirda, E. (2012). A Security Analysis of Amazon's Elastic Compute Cloud Service. *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops.* Boston: IEEE.

Kornblum. (2006). Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *The Digital Forensic Research Conference*, (pp. S91-S97). Lafayette, IN.

LogicMonitor. (2018). *Cloud Vision 2020: The Future of the Cloud.* LogicMonitor. Retrieved from https://www.logicmonitor.com/wp-content/uploads/2017/12/LogicMonitor-Cloud-2020-The-Future-of-the-Cloud.pdf

Mather, T., S., K., & Latif, S. (2009). Cloud Security and Privacy: An Enterprise Perspective On Risks and Compliance. O'Reilly Media, Inc.

McAfee. (2017). *Custom Applications and IaaS Trends 2017.* McAfee. Retrieved from http://info.skyhighnetworks.com/rs/274-AUP-214/images/wp-csa-survey-custom-apps-iaas-survey-report.pdf

Mell, P., & Grance, T. (2011, September). The NIST Definition of Cloud Computing. Gaithersburg, MD.

Osvik, D. A., Tromer, E., & Shamir, A. (2006). Cache attacks and countermeasures: the case of AES. *In Cryptographers' Track at the RSA Conference* (pp. 1-20). Springer Berlin Heidelberg.

Perlman, R. (2006). File system design with assured delete. *Third IEEE International Security in Storage Workshop*.

Phillip, A., Cowen, D., & Davis, C. (2009). Hacking Exposed: Computer Forensics. McGraw Hill Professional.

Python Software Foundation. (n.d.). *Python Language Reference*. Retrieved from http://www.python.org: http://www.python.org

Regenscheid, A. R., Feldman, L., & Witte, G. A. (2015). *NIST Special Pubication 800-88, Revision 1: Guidelines for Media Sanitization.* National Institute of Standards and Technology.

Ristenpart, T., Tromer, E., Shacham, H., & Savage, S. (2009). Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. *16th ACM conference on Computer and Communications security* (pp. 199-212). ACM.

Sengupta, S., Kaulgud, V., & Sharma, V. S. (2011). Cloud computing security -- trends and research directions. *IEEE World Congress on Services*.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2012). Operating System Concepts 9th Edition. John Wiley & Sons.

Synergy Research Group. (2018). *Cloud Infrastructure Services - Market Share Trend.* Reno: Synergy Research Group.

Synergy Research Group. (2018). *Cloud Provider Competitive Positioning.* Reno: Synergy Research Group.

Taguchi, J. K. (2013). *Optimal sector sampling for drive triage.* Monterey: Naval Postgraduate School.

Tatham, S. (n.d.). *PuTTY Download Page*. Retrieved from www.chiark.greenend.org.uk: http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Tianfield, H. (2012). Security Issues in Cloud Computing. *IEEE International Conference on Systems, Man, and Cybernetics*.

Tung, L. (2016, November 1). *AWS public cloud is twice as big as Microsoft, Google, IBM combined*. Retrieved from ZDNet: http://www.zdnet.com/article/aws-public-cloud-is-twice-as-big-as-microsoft-google-ibm-combined/

VMware. (2016, December 21). *Supported Disk Formats*. Retrieved from https://docs.vmware.com: https://docs.vmware.com/en/VMware-vSphere/6.5/com.vmware.vsphere.storage.doc/GUID-9CC98802-82D9-44B0-9735-97675258FAAE.html

VMware. (2017, June 28). *Using vmkfstools* . Retrieved from https://docs.vmware.com: https://docs.vmware.com/en/VMware-vSphere/6.5/com.vmware.vsphere.storage.doc/GUID-A5D85C33-A510-4A3E-8FC7-93E6BA0A048F.html

VMware. (2018). *ESXi*. Retrieved from https://www.vmware.com: https://www.vmware.com/products/esxi-and-esx.html

VMware. (2018). *vCenter Server*. Retrieved from https://www.vmware.com: https://www.vmware.com/products/vcenter-server.html

X-Ways. (2017, November 27). *WinHex: Computer Forensics & Data Recovery Software*. Retrieved from https://www.x-ways.net/winhex/

Young, J., Foster, K., Garfinkel, S., & Fairbanks, K. (2012). Distinct Sector Hashes for Target File Detection. *IEEE Computer Society*, 28-35.

Zissis, D., & Lekkas, D. (2012). Addressing cloud computing security issues. *Future Generation Computer Systems, 28(3)*, 583-592.

BIOGRAPHY

Bradley Lee Snyder graduated from Spotsylvania High School, Spotsylvania, Virginia, in 2000. He received his Bachelor of Science from Virginia Polytechnic Institute and University in 2004. He continued his education at Virginia Tech where he received his MBA, 2007, followed by a Master of Science in Systems Engineering in 2008.