# Investigations into Decrypting Live Secure Traffic in Virtual Environments

**Peter William Lindsay McLaren**

A thesis submitted in partial fulfilment of the requirements of Edinburgh Napier University, for the award of Doctor of Philosophy

July, 2019

## COPYRIGHT

## DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

**ACKNOWLDEGMENTS**

The PhD research has been a thoroughly stimulating experience with many peaks and troughs along the way. I would not have completed the work without support from my family, friends, and my supervisory team at Edinburgh Napier University.

I would like to acknowledge my late parents who instilled my strong desire for learning. I am most grateful to my wife, Penny McLaren, for her commitment to my reaching this stage. I would like to acknowledge her encouragement throughout as well as her shouldering significant household activities during this process. Other family members and friends, such as 'Oxford comma Phil', have also provided encouragement and recommendations when I might otherwise have yielded.

I have had the benefit of an extended supervisory team as well as an independent panel chair, Professor Hazel Hall. I would like to thank you all. I appreciate the continued encouragement, support, and reflection over many supervisory sessions from Doctor Gordon Russell, in particular once I returned to Australia. Thanks to Doctor Zhiyuan Tan for seeding the VMI idea, providing University of Technology Sydney contacts, and for his interesting, thought-provoking viewpoints. Finally, my profound thanks to Professor William Bill Buchanan OBE for his energy, excitement, and lateral thinking abilities that encouraged me to explore new opportunities.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# ABSTRACT

Malicious agents increasingly use encrypted tunnels to communicate with external servers. Communications may contain ransomware keys, stolen banking details, or other confidential information. Rapid discovery of communicated contents through decrypting tunnelled traffic can support effective means of dealing with these malicious activities.

Decrypting communications requires knowledge of cryptographic algorithms and artefacts, such as encryption keys and initialisation vectors. Such artefacts may exist in volatile memory when software applications encrypt. Virtualisation technologies can enable the acquisition of virtual machine memory to support the discovery of these cryptographic artefacts.

A framework is constructed to investigate the decryption of potentially malicious communications using novel approaches to identify candidate initialisation vectors, and use these to discover candidate keys. The framework focuses on communications that use the Secure Shell and Transport Layer Security protocols in virtualised environments for different operating systems, protocols, encryption algorithms, and software implementations. The framework minimises virtual machine impact, and functions at an elevated level to make detection by virtual machine software difficult.

The framework analyses Windows and Linux memory and val-

idates decrypts for both protocols when the Advanced Encryption Standard symmetric block or ChaCha20 symmetric stream algorithms are used for encryption. It also investigates communications originating from malware clients, such as bot and ransomware, that use Windows cryptographic libraries.

The framework correctly decrypted tunnelled traffic with near certainty in almost all experiments. The analysis durations ranged from sub-second to less than a minute, demonstrating that decryption of malicious activity before network session completion is possible. This can enable in-line detection of unknown malicious agents, timely discovery of ransomware keys, and knowledge of exfiltrated confidential information.

# Chapter 1

# Introduction

## 1.1 Context

The Internet is a primary communications medium for businesses, organisations, and individuals. It also supports malicious activity. Internet communications may be sent in the clear where an initiator, *a client*, and a receiver, *a server*, are unconcerned with other parties knowing contents of the exchange. Increasingly, the privacy of communications is a key concern for Internet users [1] that is generally realised with secure communications protocols. These protocols use encryption to provide information confidentiality, and with the growth of security awareness amongst users and developers, encryption is an increasingly common communications feature so that now more Internet traffic is encrypted than unencrypted [2] [3]. Encryption is, thus, a key facet of Internet communications for benign or malign intent.

Benign and malicious users may benefit from strong encryption in secure communications protocols. For benign users, encryption provides confidence that third parties cannot access the confidential information contained in on-line activities such as banking, purchasing of goods, and emailing. Encrypted channels also enable malicious actors to hide nefarious activities. Examples of malicious activities are attackers who obtain access to a company's

servers and exfiltrate confidential data [4] [5], malware applications running on desktops or servers communicating banking information with external controllers [6], or criminals using messaging applications to conduct conversations [7].

Information misuse is not restricted to malicious actors, as an authorised user may upload confidential data via a secure channel to an external server for later investigation [8]. Although such a transfer may be benign, the transfer might be against company policy in case, for instance, they result in General Data Protection Regulation (GDPR) breaches.

While security analysts may be able to identify harmful communications without knowledge of encrypted contents [9] [10], decoding such channels and understanding message contents may enable information owners to take measures that minimise possible damage. Obtaining message contents is problematic and is the focus of these investigations.

Principal protocols used in secure communications are Transport Layer Security (TLS) and Secure Shell (SSH). Both TLS [11] and SSH [5] enable the types of misuse described earlier, are well-defined, and extensively used [12]. Decryption of malicious communications using these protocols can therefore benefit security analysts. Dealing with misuse in live TLS and SSH communication sessions may require rapid decryption with little information as malicious users seek to evade detection with short network sessions [13] [14] [10]. An ability to discover the message contents of terse TLS and SSH sessions in live environments in a timely manner may, therefore, offer security analysts an opportunity to identify potential misuse.

## 1.2 Significance

Discovering message contents requires *decryption*. Whereas *encryption* takes information that is intelligible in some sense (*plain-*

*text*) and applies a cryptographic algorithm to generate obfuscated data (*ciphertext*), decryption is the reverse process of deriving plaintext from ciphertext. Secure communications contain ciphertext so decryption requires knowledge of the cryptographic algorithm.

Publicly-known encryption algorithms are commonly used in secure communications protocols. Kerckhoffs' Second Principle asserts that the security of an encrypted message should not rely on secrecy of the algorithm [15]. Indeed, reverse-engineering techniques may enable a proprietary algorithm's functionality to be discovered. Although such algorithms remain a subject of research [16], they may lack robustness without extensive independent verification [17].

With public encryption algorithms, encryption key secrecy is paramount. Consequently, to decrypt secure communications, decryption keys must be discovered. Many decryption techniques identify probable encryption keys through the application of statistical algorithms, ranging from simple to complex. Trial-and-error is the simplest and most direct approach in which decryption is attempted with each possible key variation. This approach succeeded in discovering the 56-bit key of the Digital Encryption Standard (DES), at the time a commonly used encryption algorithm, in 56 hours [18] and a year later in 22 hours 15 minutes [19] resulting in the algorithm's subsequent discontinuance for American Federal communications [20]. With encryption algorithm key lengths now commonly exceeding 128-bits, the probability of successful key discovery with a trial-and-error approach in a useful time frame is much reduced. As a result, researchers generally focus on more efficient approaches, typically investigating key management, or vulnerabilities in an algorithm's design or implementation.

Decryption approaches that investigate algorithmic design apply statistical analysis techniques to discover keys. Through an

iterative process of analysing discrepancies in ciphertext associated with specific plaintext, individual key bits can be determined. The process requires knowledge, and in many cases selection, of the plaintext for each iteration. Furthermore, the number of plaintext-ciphertext pairs required for accurate analysis is substantial. For instance, the linear analytic approach required $2^{43}$ known plaintext-ciphertext pairs to obtain an 85% probability of discovering a DES algorithm key [21].

Knowledge of linear analytic and differential analytic approaches enable researchers to design encryption algorithms to be highly resistant to such techniques. Although approaches to investigating algorithmic weaknesses continue to be explored, success in live scenarios is difficult as plaintexts associated with ciphertexts are generally unknown, and even if obtained, may not be available in sufficient quantities. Consequently, algorithmic design analysis techniques may be ineffective for decrypting live secure communications.

Side-channel attacks that investigate implementations of encryption algorithms are actively researched to discover encryption keys. By monitoring the physical device during the encryption process, sufficient evidence may be produced for keys to be identified. Many such approaches require physical access or proximity to the encrypting device so that, for example, PCI cards must be installed in the target [22] or target power levels measured with oscilloscopes [23]. Unless there is prior knowledge of misuse, such access or proximity may be unfeasible in practice. Even with proximity, the number of required monitoring units may be impractical to uncover malicious activities in live situations. However, memory acquisition techniques may support the discovery of data used as input to encryption and decryption processes, known as *cryptographic artefacts*.

## 1.3 Approach

Memory acquisition techniques vary in appropriateness for discovering cryptographic artefacts associated with wide-scale malicious activity in real-world scenarios. Memory forensic analysis criteria are: image copy accuracy; invulnerability to tampering [24]; the absence of signs of concurrent activity [25] [26]; and integrity where the target device is uncontaminated [27]. Extending the criteria to this investigation, techniques should be scalable, undetectable, and non-invasive.

Scalability ensures multiple suspect network sessions can be monitored simultaneously. This can be useful when many infected clients communicate with one or more controllers. Scalability can also assist where malicious activity crosses multiple devices, such as peer-to-peer botnets.

Undetectability is an awareness by target device software, or device users, of the acquisition activity. Undetectable memory acquisition reduces the opportunities for virtual machine entities to manipulate acquisition results.

Invasiveness is the degree to which memory acquisition transforms the target environment. As multiple memory acquisitions may be required for decryption, temporal consistency may be required, which can be problematic with large memory extracts [28]. Techniques permanently altering the target should be avoided to ensure comparability in repeated memory acquisitions. Also, highly invasive techniques may render the target inoperable. Undetectability and invasiveness are not independent, as highly invasive techniques may be detectable. For example, malware may detect the presence of software monitors executing on the target device.

From a technological perspective, forensic researchers classify memory acquisition methods as target hardware insertion, target direct memory access, emulation, virtualisation, kernel-level ac-

quisition user-level acquisition, Windows hibernation file, or cold-boot attack [25] [27] [29]. A less granular categorisation of hardware access, software applications, and hosted access suffices for thesis purposes. Hardware access includes target device hardware insertion, direct memory access, and cold-boot attacks; software applications include kernel and user-level acquisition tools; and hosted access includes emulation as well as virtualisation. Windows hibernation file availability implies the target is not operational and is not investigated.

**Hardware Access** Hardware access methods necessitate physical connectivity between target and collection devices. Examples of hardware access include PCI card insertion [22], USB device insertion [30], and cabled connection via Ethernet or Firewire ports [31]. Connecting new devices can be detected - for instance, Windows pop-up windows commonly notify device users when new hardware is added. Even if the hardware can be surreptitiously attached, practicality limits its wide-scale implementation unless access is already enabled, as attempted with the "Clipper" chip [32]. Cold-boot attacks, where the target memory is artificially cooled, rebooted and inspected [33], do not scale and are highly invasive as the target is permanently altered. However, such attacks are potentially undetectable as the reboot may not restart previously executing target applications. Hardware access methods have scalability, undetectability, and invasibility memory acquisition issues, which make them challenging for the discovery of cryptographic artefacts.

**Software memory acquisition** Software memory acquisition applications also have limitations. The applications execute at user or kernel level, resulting in different memory extraction capabilities. User-level applications, such as ProcDump [34] and Process Hacker [35], obtain memory of a target device process and write

memory contents to disk. The acquisition application executes on the target making it detectable by target software. Furthermore, executing the application may impact target memory including target process memory if, for example, memory is paged to virtual memory. Kernel level applications, such as FTK Imager [36], DumpIt [37], Memoryze [38], and WinPmem [39] obtain complete target volatile memory images. These tools also execute on the target and can be detected. Further, the generated image copy is either written to a local target device folder or, optionally for WinPmem, across a network. The image copy activity may not contaminate copy contents but the activity impacts the target, temporarily at least. If the image is written to a local folder, extraction for external analysis requires copying to a physical device such as a USB device. So, executing user-level or kernel-level applications for memory acquisition are detectable and partially invasive.

**Hosted Access** Hosted access may offer a scalable, non-invasive opportunity to acquire memory. In these environments, multiple virtual machines, each comprising an operating system and applications, use computer resources, including disk, memory, and network of the host physical machine. This allows programs executing on the physical host to access virtual machine memory. With physical hosts supporting multiple virtual machines, such programs have the potential for simultaneous virtual machine access and may, therefore, access the network traffic and memory of encrypting programs in any virtual machine running on that host [40]. Furthermore, by executing on the physical host, the programs may be more resistant to detection and manipulation than virtual machine programs.

Virtualisation is a key technology because it offers enhanced utilisation of computing resources so desktops and servers are frequently virtualised. For example, virtualisation technologies

underpin many cloud architectures [41]. Investigating effective approaches to decrypting secure communications traffic in virtualised environments may be useful in assisting defences against secure channel misuse.

This thesis focuses on developing a framework, MemDecrypt, to investigate the decryption of potentially malicious communications that use SSH and TLS protocols in virtualised environments as illustrated in Figure 1-1. MemDecrypt analyses traffic to predict when cryptographic artefacts used in suspect virtual machine network sessions have been generated. The framework then extracts and analyses virtual machine memory to discover candidate artefacts enabling the decryption of the suspect sessions.

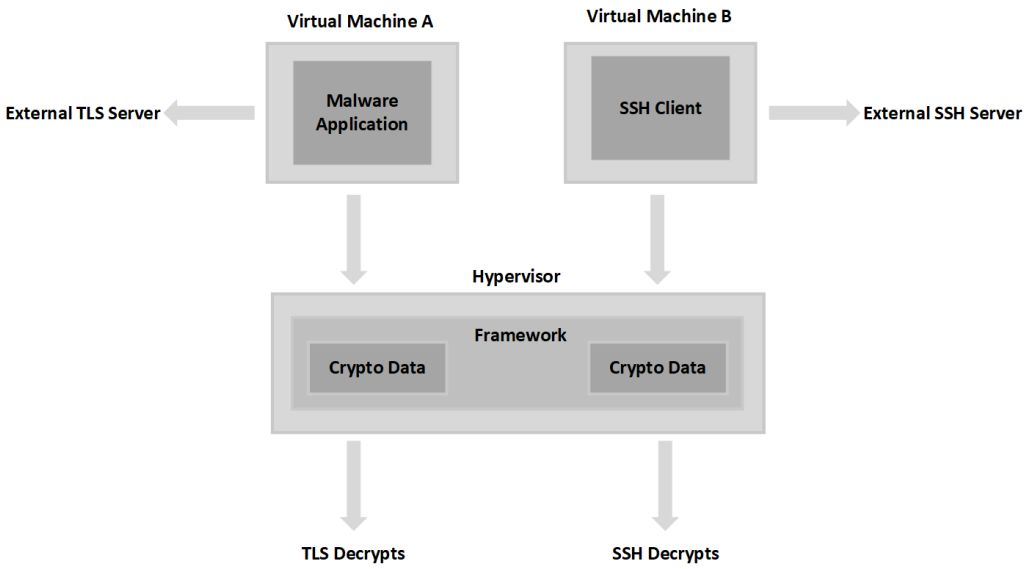



Figure 1-1: Investigative Approach

## 1.4 Research Questions

This thesis presents a framework design and construction to investigate effective and efficient decryption of TLS and SSH protocol

traffic through access to live virtual machine resources. The main research questions to be answered are:

- RQ1 How effective are existing methods for decrypting secure communications originating from live machines?
- RQ2 What new methods can be developed to decrypt secure communications originating from live machines in virtualised environments?
- RQ3 Can the new methods decrypt secure communications originating from a client in a virtualised machine efficiently and effectively?

## 1.5 Ethics

Decrypting communications poses an ethical question. There is increasing conflict between the privacy rights of citizens and the rights of society to protect itself from adversaries with some advocates favouring public safety [42] and others more equivocal [43] [44]. Also, legislative disparity exists between jurisdictions protecting user privacy and anonymity, such as the European Union [45], and those requiring the active involvement of intermediaries in enabling decryption by state agencies, such as Australia [46]. Eavesdropping by decrypting secure communications may find common acceptance depending on the who, why, what, and how of the activity [47].

Inter-state conflict provides grounds for eavesdropping. For example, decrypting enemy communications in military conflicts can change the status quo, enabling resources to be deployed to maximise advantage as occurred when the French broke the German ADFGX cipher in World War I [48]. In such instances, an intention to minimise loss of life may be the justification. Outside of military conflict, eavesdropping on supposedly unfriendly

states may also be justified, as when the American Black Chamber decrypted Japanese shipping tonnage demands in disarmament negotiations [49]. Although media reports highlight attacks on infrastructural computational capabilities, such as against Iranian nuclear facilities, Estonian national communications network, and Ukrainian power-grid attacks, cyber espionage may be more prevalent [50]. For example, the National Security Agency (NSA) is believed to have intercepted and decrypted confidential communications of allied states using a weakness in the implementation of the Diffie-Hellman algorithm in protocols such as VPN, TLS, and SSH [51].

The NSA and partnering organisations are reported to have also eavesdropped on the personal communications of foreign individuals [52] despite legislation and covenants, such as Article 12 of the United Nations declaration safeguarding personal privacy [53]. Following the Snowden revelations, the effectiveness of surveillance is openly discussed [54], suggestive of its continued practice. With encryption of email, messages, and phone calls becoming more common, eavesdropping necessitates decryption.

Intra-state eavesdropping on citizens' communications may be less contentious. Local legislation can identify processes for state organs to obtain data access. However, with end-to-end message and phone call encryption used by terrorists [55] and criminals [7], simple data access may be insufficient. In Australia, the Telecommunications and Other Legislation Amendment (Assistance and Access) Bill has been approved requiring digital communications providers to facilitate decryption of personal communications. By contrast, in the United States the FBI has failed to coerce Apple to decrypt smartphone data of alleged criminals despite judicial direction [56]. The balance between user privacy rights and crime prevention and detection has become blurred [44] and the Australian approach may be followed with interest [57].

At an institutional level, successful eavesdropping can prevent

cyber-crime. With the growth in encrypted communications between malware and controllers [58] [59] and confidential data exfiltration using TLS, SSH, and DNS by insider attackers [60], future defences may require content knowledge. Header information permits the distinguishing of malicious activity from benign activity [61] [9] [10]. Although this enables session disruption, such an approach may be counter-productive. For example, when a TLS payload contains the ransomware key, disruption may lead to loss of essential information. With SSL proxies enabling institutions' knowledge of encrypted communications content, institutional agreement to decrypt employee communications may be acceptable. The problem is more contentious for cloud vendors [62] but retention of the cryptographic artefacts enabling later decryption may assist in legislative adherence.

Knowledge or suspicion that confidential information is decryptable may lead to preventative measures. For example, the Digital Encryption (DES) algorithm was decommissioned federally after knowing that information could be decrypted within a useful time-frame [19]. More recently, TLS version 1.3 removes support for algorithm variations that encrypt after authenticating. This is believed to be consequential [63] [64] on publicised BEAST, POODLE and Lucky13 attacks [65] [66]. With evolving technologies, other cryptographic measures may be considered. For instance, in the commonly used Advanced Encryption Standard (AES) algorithm, where key lengths of 128, 192 and 256 bits have sufficed to resist current decryption techniques, key length increases are expected if quantum computer cryptographic potential is realised [67].

Generally, decrypting secure communications remains an area of conflict, with agencies seeking to discover methods that provide access to possible malicious activity while applications with vulnerabilities that enable decryption are corrected. A further concern is that techniques that support decrypting suspect com-

munications may also be employed by malicious actors to obtain the confidential data of benign actors.

The thesis aims to benefit protectors against malicious actors by decrypting their suspect communications. Although malicious actors could arguably also use the proposed approach, the results can be used to improve the robustness of secure communications applications to prevent this. The framework also presents opportunities for privacy campaigners. For instance, identified candidate cryptographic artefacts may be retained along with network traffic and decryption deferred until such time that legal sanction is obtained.

## 1.6 Contributions

The MemDecrypt framework is developed to investigate the decryption of secure communications in virtualised environments. The framework executes at a privileged level in virtualised environments which s limit misuse. Consequently, it may benefit agents with hypervisor access, such as operators of virtualised environments, as well as providing a foundation for researchers and security analysts. This thesis makes the following original contributions:

- Constructs the MemDecrypt framework that accesses virtual machine network and memory resources to rapidly identify small sets of candidate cryptographic artefacts including encryption keys. The framework decrypts efficiently with minimal impact to the target device. MemDecrypt is implemented on the Xen hypervisor but may be extended to apply to alternative environments, such as more commonly used hypervisors, as well as other applications, protocols, and algorithms. Thus, the framework can benefit researchers seeking to develop comprehensive security tools that defend against

malicious activity taking place within the target devices including virtual machines.

- Develops an extension for MemDecrypt to rapidly decrypt secure communications that use the SSH protocol. Knowledge of SSH enables candidate AES cryptographic artefacts to be discovered with minimal interruption to live virtual machines running Windows or Linux operating systems. The correct artefacts are then rapidly identified with high levels of certainty. As SSH has been used as a medium for data exfiltration [5], this capability can assist security analysts in developing defences against these attacks, which are increasingly common and difficult to detect [68].

- Builds an extension for MemDecrypt to rapidly decrypt the TLS communications of Linux and Windows virtual machines. Application of TLS protocol knowledge, including generation of cryptographic artefacts, enables small sets of AES candidate artefacts to be identified. Although TLS supports other application protocols such as SMTP, HTTP-over-TLS is commonly used for benign on-line transactions between clients and web servers and malicious communications such as malware applications communicating with external controllers [9]. The rapid decryption of HTTP-over-TLS traffic can determine the nature of malicious activity and assist security analysts in developing countermeasures.

- Constructs an extension for the rapid decryption of secure communications emanating from client bot or ransomware applications. When executing on Windows platforms, malware can use Windows cryptographic libraries, so specific features of the library data structures can enable communications to be decrypted even when the malware hijacks a benign application. This approach can benefit analysts con-

cerned with detecting new malware, tracing malware activity and security application providers interested in intercepting and preventing such breaches.

- Develops and proves an approach to decrypting TLS and SSH secure communications that use a recently approved stream encryption algorithm, ChaCha20. Two common ChaCha20 implementations leave memory traces that facilitate the rapid decryption of SSH and TLS secure communications. As well as benefiting security analysts with defences, this knowledge may enable the adoption of enhanced implementation approaches to protect ChaCha20 cryptographic artefacts.

Although this thesis focuses on virtual environments, the framework is extensible. Secure communications from other technologies that support memory access to live encrypting devices may be vulnerable and, so, benefit from the application of this framework. As memory access is important to forensic investigations [69] software tools and libraries already exist to support such capability for technologies such as desktops, servers, the Internet of Things (IoT), Android smartphones, and virtualized environments. For example, as Android smartphones run the Linux operating system, non-commercial memory acquisition tools such as the Linux Memory Extractor application ('LiME') [70], AMExtractor [71] and TrustDump [72] or commercial tools such as Cellebrite [73] may support memory access enabling possible decryption of secure network traffic.

## 1.7 Aims and Objectives

This thesis aims to deliver a framework that enables the secure communications of live virtual machines to be decrypted. The framework should minimally impact target virtual machines and

also provide security against misuse by virtual machine actors. To attain this aim, the following objectives are to be achieved:

- Review related literature. This determines the encryption algorithms for analysis through a review of cryptographic mechanisms used in secure protocols.
- An exploration of techniques for discovering cryptographic artefacts. Also included is a review of existing methodologies for accessing live virtual machine network traffic and memory.
- Design of a framework, executing at a privileged level, that can capture live virtual machine memory and encrypted network traffic, discover small sets of candidate cryptographic artefacts in the captured memory, and rapidly decrypt the encrypted network traffic.
- Implementation of the framework to decrypt the encrypted traffic originating from Windows clients and Ubuntu servers.
- Extension of the framework to SSH. Experiments are conducted to evaluate the framework capability for decrypting SSH traffic for different operating systems, encryption algorithms, key lengths, and volumes of exchanged data.
- Extension of the framework to TLS. Experiments are conducted to evaluate the framework capability for decrypting TLS traffic with different operating systems, encryption algorithms, and key lengths. A further extension investigates framework decryption of malware traffic using Windows encryption libraries.
- Proposals for future work to address any gaps in investigation and suggestions for potential interesting research areas where the framework can be adopted.

## 1.8 Organisation of Thesis

This thesis is structured as follows:

- This chapter has provided a brief introduction to the problems addressed by the research. It explains the research motivation, offers justification for investigating decryption of SSH and TLS protocol traffic in virtualised environments, proposes research questions, discusses the ethics of the thesis topic, states the thesis contributions, aim and objectives. It also presents the organisation of the thesis.

- Chapter 2 provides context for this thesis. Background on encryption algorithms, particularly symmetric block and stream algorithms with a focus on AES, its Cipher Block Chaining and Counter modes, and ChaCha20 is presented.

- Chapter 3 reviews and discusses prior research into discovery of cryptographic artefacts. This confirms that virtual machine memory analysis may be an effective mechanism for secure traffic decryption as large quantities of data, specialised equipment, or physical device access or proximity are not required. It considers studies that obtain access to live virtual machine resources, focussing on methods to extract live virtual machine network and memory data without disrupting normal operations. Although researchers have discovered encryption keys in virtual machine memory, study scopes were limited and decryption not achieved where additional cryptographic artefacts are required.

- Chapter 4 describes the framework for investigating the decryption of secure communications in virtual environments. Framework requirements focus on effectiveness, efficiency, and security. A detailed design to meet requirements is proposed followed by a consideration of implementation options. The

framework is intended to fulfil the generic purpose of decrypting secure communications in virtualised environments so extensions are supported.

- Chapter 5 investigates decrypting SSH communications that use AES encryption. Salient aspects of the Internet Engineering Task Force (IETF) SSH specifications are presented as well as features of SSH cryptographic artefacts. These enable development of an SSH extension to the MemDecrypt framework. Experiments are performed with file transfers on Windows client and Linux server virtual machines to evaluate the framework extension. Experimental results are discussed and extension limitations considered.

- Chapter 6 investigates decrypting TLS communications that use AES decryption. A MemDecrypt extension is developed to address TLS 1.2 and 1.3 protocol versions. Experiments are performed principally with TLS 1.2 on Windows client and Linux server virtual machines and an additional experiment conducted with TLS 1.3. Experimental results are discussed and extension limitations considered.

- Chapter 7 presents an investigation into decrypting secure communications originating from malware applications on Windows clients. Although the TLS protocol is used for malware-controller communications, the TLS extension has performance limitations. A MemDecrypt extension to accommodate the use of Windows cryptographic libraries and prioritising memory extracts is constructed. Experiments are performed with real ransomware and bot malware applications on Windows clients. Experimental results are discussed and limitations to the Windows library cryptographic extension considered.

- Chapter 8 investigates SSH and TLS use of the ChaCha20-

Poly1305 symmetric stream algorithm. A framework extension is developed to discover cryptographic artefacts in common implementations. Experiments are carried out using Windows and Linux operating systems for each protocol. ChaCha20 experimental results are discussed and countermeasures considered.

- Chapter 9 reviews the experimental results and discusses implications of the investigation. It reflects on whether contributions have been provided, reviews research questions, thesis aim and objectives, and proposes future research using the framework as a platform.

## 1.9 Publications

During these studies, the following papers have been published in, or have been submitted to, peer-reviewed journals. All except the first paper relate to specific experiments conducted for this thesis.

- McLaren, P., Russell, G., & Buchanan, B. Mining malware command and control traces. In 2017 Computing Conference (pp. 788-794). IEEE [10].
- Buchanan, B., Tan, Z., McLaren, P., & Russell, G. Decrypting Live SSH traffic in virtual environments Digital Investigation. Elsevier [74]. This paper is expanded on in thesis Chapter 5.
- Buchanan, B., Tan, Z., McLaren, P., & Russell, G. Discovering Encrypted Bot and Ransomware Payloads Through Memory Inspection Without A Priori Knowledge. ACM Transactions on Privacy and Security (submitted). This paper is expanded on in thesis Chapter 7.
- Buchanan, B., Tan, Z., McLaren, P., & Russell, G. Deriving ChaCha20 Key Streams From Targeted Memory Analysis.

Journal of Information Security and Applications. Elsevier (2nd draft submitted after first review). This paper is expanded on in thesis Chapter 8.

# Chapter 2

# Background and Theory

## 2.1 Introduction

This chapter provides a background on cryptography with a focus on the application of symmetric encryption algorithms in secure communications encryption and decryption processes. It discusses the importance of *initialisation vectors* and *nonces* to these processes through a description of two major encryption algorithms, AES and ChaCha20.

Cryptography enables secure transmission of information across insecure digital networks. Security generally requires transmitted information to be inaccessible to an eavesdropper, uncorrupted by a man-in-the-middle, to reach the intended recipient, or more technically, digital network security requires confidentiality, integrity, and authentication.

Information is encrypted to achieve confidentiality [75]. When a cryptographic algorithm generates ciphertext from the plaintext, an eavesdropper should be prevented from discovering the plaintext in encrypted communications. Insecure digital networks are a standard communications transport medium so, for privacy, secure communications protocols employ encryption algorithms. Cryptographic hash algorithms provide potential for message integrity whereby the receiver verifies that received information is

what was transmitted. Authentication commonly employs cryptographic algorithms which enable verification of the respondent's identity.

## 2.2 Cryptography in Digital Networks

In digital communications, asymmetric and symmetric algorithms support confidentiality, integrity, and authentication. Following Kerckhoffs' Second Principle [15], these algorithms are generally publicly known so key secrecy is paramount. Whereas asymmetric algorithms require two keys, a public and private key pair, symmetric algorithms require one key. This characteristic enables different, useful cryptographic requirements to be addressed.

Uses for asymmetric algorithms in digital communications include authentication and digital signing. In these instances, the sender encrypts information with the sender's private key and the recipient uses the public key to decrypt the message. If an eavesdropper intercepts an exchange and interposes a replacement message, the receiver's decrypt fails.

Asymmetric encryption keys are typically generated using complex mathematical operations such as exponentiation and modular arithmetic. These operations make discovering asymmetric algorithms' private keys challenging. The complexity takes processor time, so asymmetric algorithm encryption is considerably less efficient than with symmetric algorithms [76]. Consequently, for secure communications, asymmetric algorithms, such as Elliptic Curve and El-Gamal, are commonly used in the initial phase of a network session to securely distribute session encryption and message authentication keys for subsequent message integrity and confidentiality.

Message integrity is achieved by comparing message hashes. Application of a publicly known hash algorithm, agreed in the initial phases, to the transmitted message generates a hash value. As

obtaining the same hash for different messages is unlikely, equivalence between the received hash and the receiver-generated hash provides confirmation that the message has not been corrupted.

Network protocols commonly use symmetric encryption algorithms for confidentiality. In symmetric encryption, the same key encrypts and decrypts, so in a secure network session encryption keys for transmitting and receiving must be mutually known. Symmetric algorithms encrypt data element-by-element (where an element is a bit or byte), known as stream algorithms, or in blocks of a defined size, known as block algorithms. Despite stream algorithms generally encrypting faster than block algorithms, their implementation has been considered difficult [77] or less adaptable [78]. However, concerns with the lack of alternatives should vulnerabilities be discovered in, and the software performance of, symmetric block algorithms have led to the adoption of specific symmetric stream algorithms in secure communications [79].

This thesis investigates decrypting secure communications traffic encrypted with symmetric key block and stream algorithms.

## 2.3 Symmetric Block Algorithms

Symmetric block encryption with a session key alone may be vulnerable. Plaintext is generally segmented into blocks of 64 or 128 bytes. Encrypting two identical data blocks with the same key and algorithm produces two identical encrypted blocks, which in many circumstances is an unacceptable weakness [80]. To avoid this, the US National Institute of Standards and Technology (NIST) recommendation is that symmetric block algorithms use initialisation vectors for confidentiality [81]. The initialisation vector differs for each block encrypted with the same key so that through incorporating them in the encryption process, identical plaintext blocks encrypted with the same key produce different ciphertext.

### 2.3.1 Modes of Operation

For symmetric block algorithms, the application of initialisation vectors in encrypting successive blocks of data is known as *mode of operation*, or simply *mode*. The NIST reviewed five confidentiality modes for symmetric block algorithms: Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), Counter (CTR), and the Electronic Codebook (ECB). ECB does not use an initialisation vector, and as a result, exhibits the vulnerability mentioned before. An example of ECB's weakness was demonstrated in an analysis of the 2013 Adobe data loss [82] where passwords were encrypted with a symmetric block encryption algorithm in ECB mode. Derivation of a single user's password through alternate means enabled all accounts with the same password to be accessible. Of the other modes, CBC and CTR have been in common use [83] [84] and are supported by secure protocols, such as SSH and TLS 1.2, when used with the Advanced Encryption Standard. However, as TLS 1.3 has removed support for CBC because vulnerabilities were discovered when used with TLS, use of that mode may decline.

The application of initialisation vectors varies per mode. For CBC, the initialisation vector for encrypting and decrypting each block of data after the first block is the encrypted output of the previous block, i.e. the previous ciphertext. A bitwise exclusive-or operation of the initialisation vector and plaintext provide the input for the encryption process, and for decryption the exclusive-or operation on the output of the decryption process and initialisation vector yields the plaintext. The NIST requires the initialisation vector for the first block in CBC mode to be unpredictable, i.e. random. Figure 2-1 illustrates the encryption and decryption processes for two consecutive blocks using CBC mode including the chaining between them.

In CTR mode, after its initial value is generated, the initialisa-

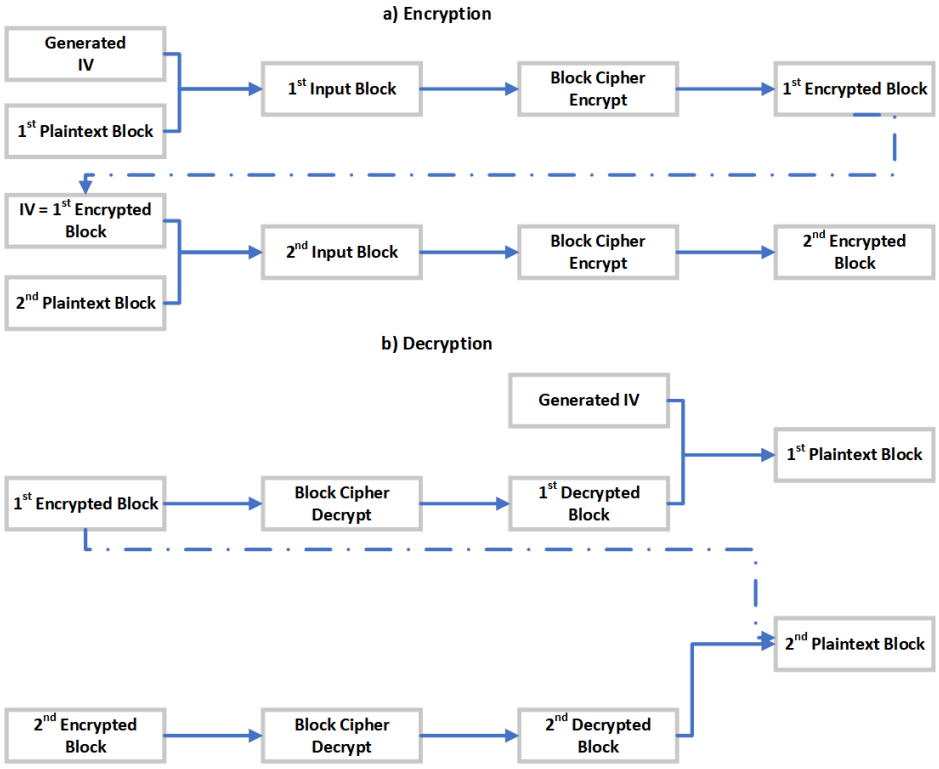



Figure 2-1: CBC Mode Encryption and Decryption Process

tion vector increments by a defined value for each successive block of data. Normally, the incremental value is 1, although other values are possible. As with CBC mode, the initialisation vector for encrypting and decrypting the first block should be unpredictable. Unlike the chaining modes, such as CBC, the initialisation vector for each data block after the first block can be calculated independently of previous encryptions. This allows for possible parallelisms in encryption and decryption processes. In CTR mode, the initialisation vector is encrypted and then an exclusive-or operation is performed with the unencrypted block of data to obtain the ciphertext as shown in Figure 2-2.

With the ability to encrypt blocks in parallel, the CTR mode encrypts large amounts of data considerably faster than other modes, which use chaining, so that CTR is commonly preferred for

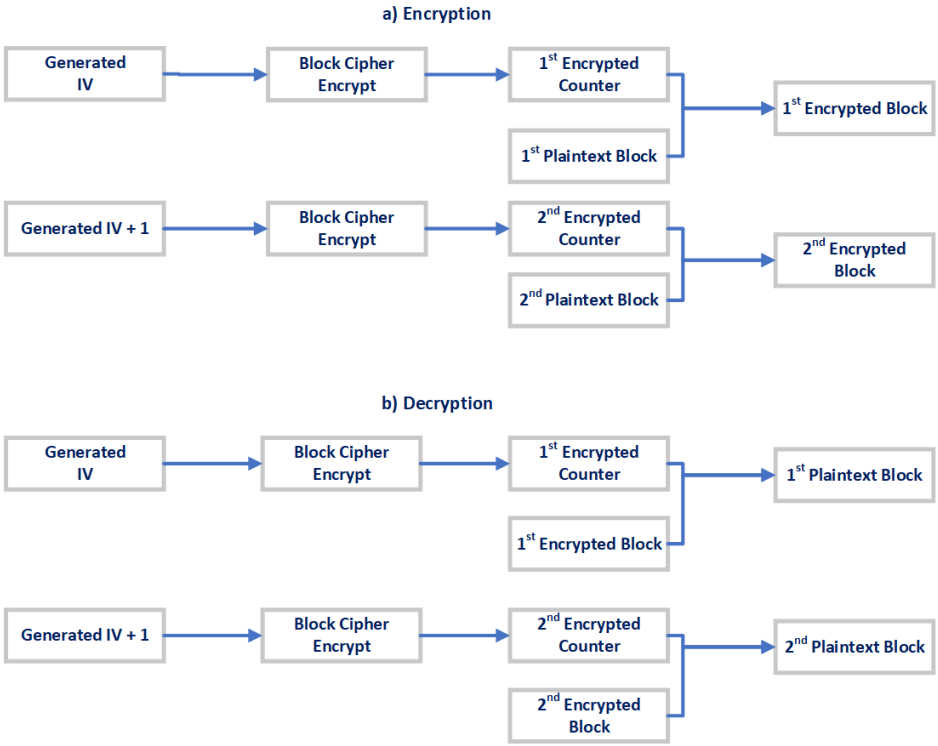



Figure 2-2: CTR Encryption and Decryption Process

secure communications [85]. CTR has also been recommended for confidentiality when compared with the other modes [86] although a lack of CTR initialisation vector uniqueness may weaken security [17]. Many symmetric block encryption algorithms, including the widely adopted Advanced Encryption Standard, support CTR mode, and the CTR mode is approved with that Standard for major secure communications protocols.

### 2.3.2 Algorithms

With increased demand for information confidentiality in digital networks [58], confidence in an algorithm's invulnerability to key discovery is essential. A vulnerability can result from shortness of key length or algorithm weaknesses. For example, the Data Encryption Standard (DES), previously used for protecting financial

transactions inter alia, had an effective key length of 56 bits. With the availability of increased computational power, keys were inevitably discovered in less than a day using the exhaustive, brute-force method of checking every possible key [19]. Keys may be discovered faster than brute-force by applying statistical analysis to plaintext-ciphertext pairs [87] [21] as will be discussed in the next chapter.

Concerns with DES and its successor, TripleDES, led to the NIST search for a replacement algorithm to become known as the Advanced Encryption Standard. Five of the fifteen submissions, namely Mars, Rijndael, RC6, Twofish, and Serpent, were shortlisted on the basis of security, speed and memory usage, implementation flexibility, algorithm simplicity, and public comments with a particular focus on the first three qualities. Because of its versatility across software and hardware platforms, low memory requirements, and fast key set-up times, the NIST selected the Rijndael algorithm, with restrictions, to be the Advanced Encryption Standard (AES) for symmetric block ciphers [88]. The restrictions are that the block size must be 128 bits, i.e. 16 bytes, and key lengths must be 128, 192 or 256 bits [89]. For AES, a segment of unencrypted data is divided into blocks of 16 bytes for encryption and decryption. As plaintext must be a multiple of 16 bytes, padding is added to the last block where necessary.

Despite AES being called the 'gold standard' for encryption [79], limitations exist. Although shortlisted AES candidate implementations were evaluated on floating-point gate arrays, for newer technologies such as sensors, radio-frequency identification (RFID), and embedded devices, which are limited in memory, processor and/or power availability, alternate lightweight block algorithms, such as PRESENT, may be preferred [90]. As virtualised systems hardware is generally unrestricted by these limitations, this thesis focuses on investigating AES as the symmetric block algorithm.

### 2.3.3 Advanced Encryption Standard

AES encryption and decryption processes perform repeated sequences of mathematical operations on the plaintext. Each repeated sequence is known as *a round* and the number of sequences depends on the key size: 10 rounds for 128-bit keys, 12 rounds for 192-bit, and 14 rounds for 256-bit keys where the output from the final round of encryption is the ciphertext. Decryption reverses the process.

The encryption process is illustrated in Figure 2-3. The input to each round is constructed as an array. Substitution output is produced by matching input bytes with a predefined substitution array, known as an *S-Box* as shown in Algorithm 2.1. The Shift operation diffuses the array by moving row elements by varying offsets as shown in Algorithm 2.2 and the Mix operation diffuses the array further by performing column wise calculations over an algebraic field as shown in Algorithm 2.3. The Add Round Key, as shown in Algorithm 2.4, is a bitwise exclusive-or operation, *XOR*, between a round key and either the output from the Mix operation, for rounds 1 to $n-1$, or the output of the shift operation, for round $n$. Round keys are derived from the encryption key and the assembly of all round keys is commonly known as a *key schedule*.

**Algorithm 2.1:** AES Substitute

**Data:** $S$ is State Array
**Result:** $S^{'}$ is new State Array
**for** $i$ = *0 to 3* **do**
  **for** $j$ = *0 to 3* **do**
    $S'_{ij} \Longleftarrow Substitute(S_{ij})$ ;
  **end**
**end**
where Substitute is the S-Box value matching the $S_{ij}$ element

AES is frequently used in secure communications for security and ease of implementation. The Rijndael proposal demonstrated

**Algorithm 2.2:** AES Shift Rows

**Data:** $S$ is State Array
**Result:** $S^{'}$ is new State Array
**for** $i$ = *0 to 3* **do**
  **for** $j$ = *0 to 3* **do**
    $S^{'}_{ij} \Longleftarrow S_{ij} << i$ ;
  **end**
**end**

**Algorithm 2.3:** AES Mix Column

**Data:** $S$ is State Array
**Result:** $S^{'}$ is new State Array
**for** $j$ = *0 to 3* **do**
  $S^{'}_{0j} = (2 \bullet s_{0j}) \oplus (3 \bullet s_{1j}) \oplus s_{2j} \oplus s_{3j}$;
  $S^{'}_{1j} = s_{0j} \oplus (2 \bullet s_{1j}) \oplus (3 \bullet s_{2j}) \oplus s_{3j}$;
  $S^{'}_{2j} = s_{0j} \oplus s_{1j} \oplus \bullet s_{2j}) \oplus (3 \bullet s_{3j})$;
  $S^{'}_{3j} = (3 \bullet s_{0j}) \oplus s_{1j} \oplus s_{2j} \oplus (2 \bullet s_{3j})$;
**end**
where $\bullet$ is multiplication over GF($2^8$)

**Algorithm 2.4:** AES Add Round Key

**Data:** $S$ is State Array, $R$ is Round Key Array
**Result:** $S^{'}$ is new State Array
**for** $j$ = *0 to 3* **do**
  **for** $j$ = *0 to 3* **do**
    $S^{'}_{ij} \Longleftarrow S_{ij} \oplus \ R_{ij}$ ;
  **end**
**end**

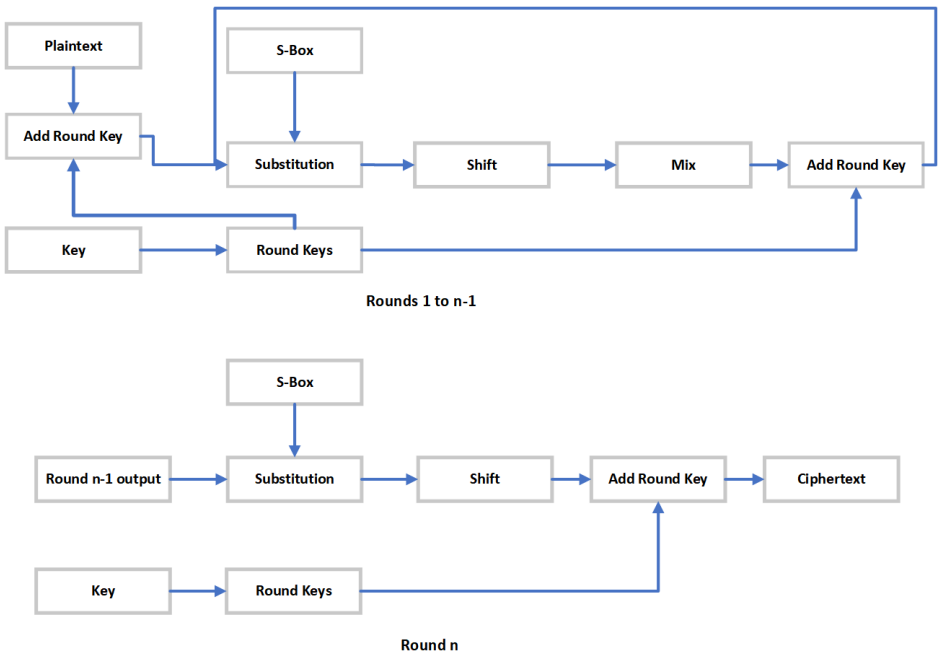



Figure 2-3: AES Encryption Process

resistance to known attacks [91]. Subsequent research has not established any overt vulnerabilities in the algorithm so AES is currently considered secure. Because the same set of mathematical calculations is used in each round, the software code is reusable, allowing for compact AES implementations. Where AES implementations support CTR, CBC, OFB or CFB modes, an initialisation vector the size of the AES block is used. This thesis investigates decrypting secure communications that implement the AES symmetric block encryption algorithm in CTR and CBC modes.

## 2.4 Symmetric Stream Algorithms

This thesis also investigates symmetric stream decryption. Symmetric stream algorithms substitute plaintext elements, bits or bytes, with ciphertext elements. The substitution changes for each

element to make frequency analysis ineffective. Typically, stream algorithms generate continuous pseudorandom key streams [92], which are melded with plaintext using mathematical operations to obtain ciphertext. With symmetric streaming algorithms, cryptographic artefacts are shared by secure communications parties. The generated key stream is typically XORed with the plaintext to generate ciphertext [92] so the decryption process is identical to encryption.

Stream algorithm software implementations are demonstrably faster than block algorithm equivalents although possibly more difficult to implement [77]. Symmetric stream algorithms are useful where power consumption and memory is restricted. So, stream algorithms have typically been used in embedded technologies such as Internet of Things (IoT) devices and smartphones [93].

### 2.4.1 Algorithms

Stream algorithms used in IoT devices, for example, are Trivium and Enocoro [90]. The RC4 stream cipher was applied in mobile communications, such as the weak WEP protocol, as well as environments with no significant power and memory concerns. As the algorithm is vulnerable [94], RC4 is deprecated for TLS [95] and SSH [96]. New stream algorithms are now available for these protocols.

Concerns with AES software implementation performance, and the consequences if AES vulnerabilities are discovered, has led to an interest in alternatives [79]. As a result, ChaCha20 with the Poly1305 authenticator has been adopted for SSH [97] and TLS [98], and is the default encryption algorithm for Google mobile phone connectivity using Chrome [99]. ChaCha20 design makes it resistant to timing attacks [100] [101]. Furthermore, algorithmic and side-channel attacks require large quantities of plaintext-

ciphertext pairs for success [102].

### 2.4.2 ChaCha20

ChaCha20 [101] is an extension to the Salsa20 stream algorithm [100]. The algorithm takes as input a constant, a key, a nonce, and a counter. (Stream algorithms refer to nonces rather than initialisation vectors but, in context, the terms are interchangeable) [79]. This creates a key stream, which is melded with the plaintext stream to generate ciphertext. In software-only implementations, it is more than three times faster than AES [98] and is well suited to lower-powered devices and real-time communications. The encryption process is illustrated in Figure 2-4.

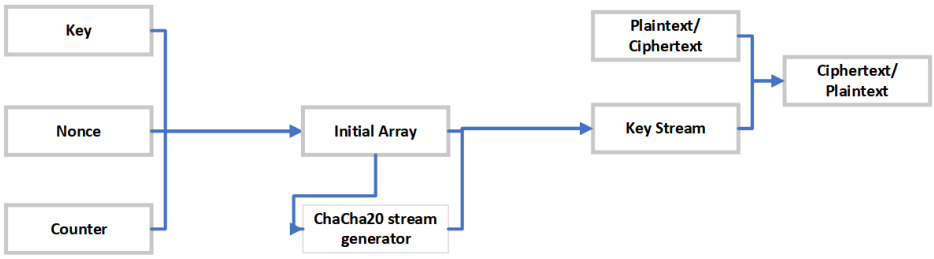



Figure 2-4: ChaCha Encryption Process

ChaCha20 operates on 32-bit words with a key of 256 bits (K = ($k_0$, $k_1$, $k_2$, $k_3$, $k_4$, $k_5$, $k_6$, $k_7$). This outputs 512-bits blocks for a key stream ($Z$) which is XORed with the plaintext stream. The state of the encryption is stored with 16 x 32-bit word values within a 4x4 matrix as shown in Equation 2-1:

$$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{bmatrix} \tag{2-1}$$

The initial state contains 16 x 32-bit values with constant values (`0x61707865`, `0x3320646e`, `0x79622d32`, `0x6b206574`), the key ($k_0$, $k_1$, $k_2$, $k_3$, $k_4$, $k_5$, $k_6$, $k_7$), the counter ($c_0$) and the nonce ($n_0$, $n_1$, $n_2$) as shown in Equation 2-2:

$$\begin{bmatrix} \texttt{0x61707865} & \texttt{0x3320646e} & \texttt{0x79622d32b} & \texttt{0x6b206574} \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ c_0 & n_0 & n_1 & n_2 \end{bmatrix} \tag{2-2}$$

The counter thus has 32-bits (1 x 32 bits), and the nonce has 96-bits (3 x 32 bits). ChaCha20 then defines a quarter round (QR), which is shown in Algorithm 2.5.

**Algorithm 2.5:** ChaCha20 Quarter round

**Result:** QR(a,b,c,d)

$a = a + b$;
$d = d \oplus a$;
$d = (d) << 16$;
$c = c + d$;
$b = b \oplus c$;
$b = (b) << 12$;
$a = a + b$;
$d = d \oplus a$;
$d = (d) << 8$;
$c = c + d$;
$b = b \oplus c$;
$b = (b) << 7$;

20 rounds are carried out (10 for column rounds and 10 for diagonal rounds) where the QR function is invoked 4 times on each

iteration as shown in Algorithm 2.6.

**Algorithm 2.6:** ChaCha20 Keystream

**Data:** X is created with K, c and n
**Result:** Z is the resultant key stream.
$y \longleftarrow X$;
**for** *i = 0 to 9* **do**
  $(x0, x4, x8, x12) \longleftarrow QR(x0, x4, x8, x12)$;
  $(x1, x5, x9, x13) \longleftarrow QR(x1, x5, x9, x13)$;
  $(x2, x6, x10, x14) \longleftarrow QR(x2, x6, x10, x14)$;
  $(x3, x7, x11, x15) \longleftarrow QR(x3, x7, x11, x15)$;
  $(x0, x5, x10, x15) \longleftarrow QR(x0, x5, x10, x15)$;
  $(x1, x6, x11, x12) \longleftarrow QR(x1, x6, x11, x12)$;
  $(x2, x7, x8, x13) \longleftarrow QR(x2, x7, x8, x13)$;
  $(x3, x4, x9, x14) \longleftarrow QR(x3, x4, x9, x14)$;
**end**
$Z \longleftarrow X + y$

The counter increments for repeated iterations to generate additional keystreams. As the plaintext is not required to be a 512-bit multiple, unused key stream elements are discarded. Of note, is that ChaCha mathematical operations are: addition; bitwise XOR; and bitwise rotation, so that software implementations can be fast. ChaCha and Salsa variants allow for reduced rounds of 8 and 12, such as ChaCha12 which implements 12 rounds. XChaCha is an extension with a 192-bit nonce [103] proposed for Google Android disk encryption [104].

## 2.5 Conclusions

Symmetric block and stream algorithms have been adopted for encrypting network traffic in digital network security protocol standards. Both AES and ChaCha20 use keys and initialisation vectors (or nonces) in encryption and decryption processes so both

artefact types must be discovered to reveal the original plaintext.

Alternative symmetric algorithms for encrypting secure communications exist. However, the pre-eminence of ChaCha20 and AES in TLS, and to a lesser extent, SSH, protocols in confidential client-server interaction, suggests that focusing on these algorithms in this thesis may be beneficial.

# Chapter 3

# Literature Review

## 3.1 Introduction

With increased usage of secure protocols in the communications of malicious actors [105], an ability to uncover exchanged information speedily enough to prevent damage is important. Rapidity is critical as malware infections can spread or banking details be lost and transacted upon in seconds.

Encryption keys may be discovered using techniques that include attacks on algorithm's design, its implementation, or key management. The review established that typical approaches targeting implementation, known as side-channel attacks, are unlikely to succeed in the rapid decryption of terse live network communication sessions, encrypted with algorithms such as AES and ChaCha20. However, research suggests that virtual machine monitoring may provide opportunities for discovering cryptographic artefacts.

Virtual machine monitoring enables full access to virtual machine resources. To prevent monitor manipulation, monitoring from outside, such as the hypervisor, may be preferred. Differences between the hypervisor and the monitored virtual machine introduces the issue of the *semantic gap*, which is discussed.

External virtual machine monitoring has been employed for

various purposes, frequently associated with detection or analysis of malware activity. The virtual machine resources employed in such studies are considered with a particular focus on the discovery of encryption keys in virtual machine memory. This review establishes that decrypting secure traffic from live virtual machines may be a productive and, as yet, unresearched topic.

## 3.2 Implementation Attacks

Decryption, where plaintext is obtained from the ciphertext, is the partnering process to encryption so that decryption commonly requires knowledge of the encryption algorithm, key, and frequently the initialisation vector. Where public encryption algorithms are employed, only key and initialisation vector need be ascertained. Prior research has focused solely on key discovery.

Implementation attacks discover keys from information leaked by a device during the encryption process [89]. When a process executes a set of instructions, physical events may occur that can be measured with appropriate technologies. Sufficient measurements allow partial or complete key identification. Principal implementation attack approaches, also known as side-channel attacks, include differential fault analysis, timing attacks, power analysis, and electromagnetic radiation analysis [106]. Although memory analysis also investigates the encryption process implementation, it is not commonly considered as a side-channel attack, perhaps because physical process measurements are absent. A high-level classification of implementation attacks is illustrated in Figure 3-1.

**Differential fault analysis** Differential fault analysis manipulates a bit or byte on an encrypting device by external interference and uses differential analysis on resultant ciphertexts to discover keys [107]. A single generated fault may suffice to discover a

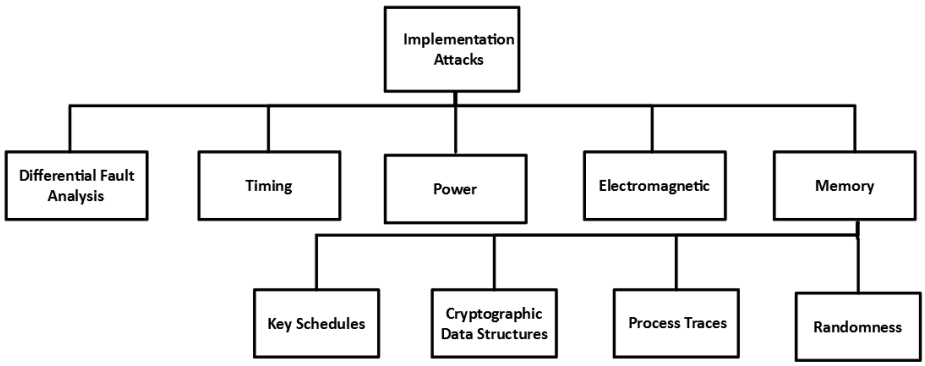



Figure 3-1: Decryption Approaches

128-bit AES key under selected conditions [108] and a 256-bit AES key may be discovered with two pairs of correct and faulty plaintexts and associated ciphertexts [109]. So, key recovery may be achieved with little computational effort. Analysis requires physical access to manipulable computing devices, such as smart cards, and electronic equipment to engineer bit or byte changes. Differential fault analysis may be more attuned to test situations than live cloud and mobile network environments.

**Timing** Timing attacks measure instruction execution time to identify keys [110]. Because of significant clock cycle differences between accessing cached and non-cached memory, measuring cache access is a common type of timing attack [111] [112], although branch instruction timing has also been successfully studied [113].

As block algorithms, such as AES, typically use S-Boxes, cache access times can be measured through a displacement of elements prior to encryption. Although hardware access requirements are absent, timing attacks require repeated tests with chosen plaintext for key recovery. Moreover, software [114] and hardware [115] counter-measures use specific assembler instructions that mask AES encryption processes. ChaCha20 is resistant to cache timing attacks as look-up tables are not used [100] [101].

**Power** Power analysis techniques sample electrical data to measure power consumed by an encrypting device. Whereas simple power analysis involves a visual inspection of measurements, differential power analysis compares power consumption for iterations with different plaintexts and a fixed key [116]. Both forms require large quantities of plaintexts to obtain meaningful accuracy [23]. Also, measuring and encrypting devices require physical connectivity or proximity. These approaches can succeed with access to proximate small devices such as embedded hardware, field-programmable gate arrays, or smart cards, but may be challenging in real-world scenarios such as cloud environments.

**Electromagnetic** Measuring electromagnetic radiation is possible without device connectivity perhaps offering greater flexibility than other techniques [117]. A related approach uses microphones for measuring acoustic signals emitted when encryption operations are performed [118]. Bandwidth and noise challenges can make effective measurement more expensive and difficult but forms of exponentiation used in asymmetric encryption have been attacked to discover partial keys. As with similar hardware attacks, measurement equipment should be in close proximity to the encrypting device and chosen plaintexts may be required.

### 3.2.1 Memory

Cryptographic artefacts may be discovered in volatile memory. As the artefacts are parameters of software encryption processes, they may be resident in encrypting device memory. The observation, regarding implementation attacks, that exploiting such weaknesses may seem unfair, but makes the impact more important [119] could apply to searching memory for these artefacts. Memory acquisition methods for key discovery vary but, in the absence of a taxonomy, a classification might be key-schedule

matches, program data structures, process traces, and randomness tests.

**Key Schedules** Encryption programs can improve performance by generating cryptographic data structures for frequent re-use such as key schedules comprised of round keys. Halderman et al. [33] investigated whether blocks of memory in an artificially cooled, rebooted computer might contain full or partial key schedules for symmetric and asymmetric algorithms. For AES, each memory block was treated as a possible key schedule. Round keys were generated for the first block segment and identified as potential keys if round keys closely matched memory block contents. A similar approach was extended to the AES candidate, Serpent [120].

These studies aimed to discover keys used by disk encryption tools such as BitLocker. However, when applied to network session key discovery in, for example, Skype, the researchers were unsuccessful [120]. In these studies, for disk encryption, the elapsed key discovery durations ranged from 75 seconds with pre-processing [120] through to 2 hours [33]. Although the duration for key recovery using cold-boot techniques has been improved with more efficient search algorithms [121] and the use of specialised hardware [122], such approaches may not scale in real-world scenarios.

**Cryptographic Data Structures** Key discovery performance can be improved by finding identifiers for cryptographic data structures. For example, known malware key delimiters assisted in determining the RC4 keys used in enciphered botnet command and control traffic [123]. Implementation knowledge also enabled the identification of Twofish key structures, another AES candidate, used in TrueCrypt [120]. As RSA keys are frequently stored as ASN.1 objects, private asymmetric keys can also be found in memory [124] [125]. Cryptographic data structure specifications

are not necessarily available, as, for instance, for Windows cryptographic libraries.

**Process Traces** Encryption artefacts have been discovered by determining encryption sub-routine parameters. One technique is to attach tracing or debugging tools to an encrypting process to access program data. For example, the Linux *ptrace* command can identify the probable encryption key parameter, using known plaintext to differentiate data segments [126]. Although effective for experimentation, the use of tracing and debugging tools may be considered too invasive in live environments. Furthermore, as the approach is detectable by software executing on the encrypting device, malicious actors may cease their activity.

**Randomness Measures** Measuring randomness can assist in encryption key discovery. Entropy is a commonly used randomness measure that has distinguished potential key regions from other memory regions. As an example, Shamir and Someren [127] counted the number of unique byte values in a string to differentiate memory regions and identify potential RSA key locations. An entropy-like measure also facilitated the discovery of TLS master keys in memory [128]. Here, researchers evaluated bit-level randomness assuming a random string should have the same number of '0's as '1's. Non-random counter-examples, such as '0000111', '1111000', '0000111',... '0000111', '1111000' may defeat this approach.

Standard byte-level entropy measures, such as Shannon entropy, are also commonly applied. For example, Shannon entropy enabled detection of probable Twofish algorithm key schedules when searching for S-Box values [120]. Shannon entropy has also been used as a filtering mechanism to identify potential AES key schedules [33] [123]. In this thesis, randomness measurements will be used to discover keys in virtual machine memory.

### 3.2.2 Virtualised Environments

As virtualised environments enable virtual machine volatile memory access, studies have been undertaken to discover keys. For example, a virtual machine environment supported testing in unsuccessful searches for session keys [120]. In other studies, entire virtual machine images were copied to support RSA key searches [125] [123]. TLS encryption keys have also been discovered in virtual machine environments by the TLSkex system [128].

The TLSkex study is suggestive of a method that could support wide-scale decryption of malicious traffic, as in-line memory extracts were acquired to enable encryption key discovery [128]. Initialisation vectors were not obtained, so its applicability was restricted to a specific algorithm and mode. However, the virtual machine monitoring approach could be employed to support rapid decryption of secure communications.

## 3.3 Virtual Machine Monitoring

Plaintext can be obtained from virtual machine memory without decryption. For example, hooked system calls, where hooks are program modifications that interpose activities before the intended function is invoked, in a controlled target server can log information received from a client [129]. Such circumstances may be exceptional. Generally, additional virtual machine resources are required to obtain the plaintext. Resources such as memory, network traffic, and file activity can be monitored externally by a host-based monitor or internally by a virtual machine application. For convenience, the process of examining a virtual machine from the outside has been called virtual machine introspection, or VMI for short [130].

Host-based virtual machine monitoring may be challenging. It offers strong potential for VMI monitor application security, but

as it executes on a different platform from the virtual machine, comprehending virtual machine activity may be difficult because of the *semantic gap*, which is discussed below. Early VMI researchers often addressed the challenge by installing internal virtual machine agents to communicate with the VMI application [131] [132]. However, newer approaches may enable VMI applications to monitor effectively without the need for internal agents [133].

**Monitor Security** The avoidance of malware tampering is a cogent reason to prefer VMI over internal monitoring. Code running on an unprivileged virtual machine typically runs at a lower privilege than the hosting hypervisor [134] so malware on an unprivileged virtual machine must elevate privilege to manipulate a VMI application [130]. This is notwithstanding that hypervisor vulnerabilities can enable virtual machine break-out [135]. By contrast, malware on an unprivileged virtual machine can detect and subvert internal virtual machine monitoring tools with ease [136]. Nevertheless, malware can avoid VMI detection and analysis through kernel data manipulation [132] [137], or termination on detection of sandboxing, debugging operations, or virtualisation [138].

**Semantic Gap** The semantic gap is an issue because of operating system differences. Analysing virtual machine activity often requires high-level information, such as process lists or open file tables. However, data extracted from a virtual machine is raw, that is, in the form of bits and bytes [139]. Even if the hypervisor and the hosted virtual machine are running the same versions of the same operating system, system data structures, such as process lists, differ. VMI derivation of high-level knowledge from low-level virtual machine data has been termed 'closing the semantic gap' [140]. Various techniques have been developed to address the

issue.

Manual derivation of virtual machine semantic structures has been the most commonly used approach [141]. For this, VMI applications are re-configured for each monitored virtual machine operating system and operating system version. With frequent operating system updates, maintaining such VMI applications can become onerous. For example, the Windows 10 operating system at one stage had 11 base builds with between 2 and 151 updates [142]. As a result, more automated solutions have been developed.

Image cloning has been implemented to close the semantic gap. In Virtuoso, a training environment used the same operating system as the monitored virtual machine [140]. Introspection tool profiles were programmatically developed on the copied image. An inability to analyse all system calls, or generate tools for file or network introspection, and a requirement to clone and train each virtual machine were acknowledged as limitations by the researchers. Although more efficient than manual derivation, this approach may be inappropriate for general use.

The semantic gap can be addressed through hooking virtual machine system calls that invoke hypervisor functions. A VMI application hook can obtain relevant virtual machine semantic data structures. For instance, in HyperShell, when specific virtual machine system calls invoked functions in the KVM hypervisor and its Linux kernel, a virtual machine helper process provided VMI access to virtual machine memory [143] [144]. This approach required mapping virtual machine calls to hypervisor functions, known as para-virtualisation, and also that the virtual machine operating system and the hypervisor use compatible operating systems. So, although HyperShell successfully monitored Linux virtual machines on a KVM hypervisor, analysing a Windows virtual machine running on, say, the VMware ESXi hypervisor might be difficult.

Similarly, approaches can combine internal and VMI monitor-

ing. An internal virtual machine process can extract requisite system data structures and communicate them to the VMI monitor. In ShadowContext, a randomly selected virtual machine process was cloned to fulfil the monitor helper role [145]. Cloned process calls were intercepted by the VMI monitor for semantic data extraction. However, virtual machine helper processes are detectable and potentially manipulable by malware.

VMI integration with high-level semantic derivation tools can also bridge the semantic gap. Memory forensics tools such as Volatility [146] and Rekall [147] commonly extract semantic information from memory images for the investigation of potentially malicious activity. Integrating these tools with virtual machine memory extraction libraries, such as LibVMI [148], provides an opportunity for the extraction of higher-level information from live virtual machines. Although researchers still create custom VMI access routines to derive semantic data [149], integrating forensic memory tools and VMI applications is common [133] [150] [151]. With better security and a solution to the semantic gap issue, VMI provides greater confidence in the validity of virtual machine monitoring results.

### 3.3.1 Virtual Machine Introspection

VMI inspects the resources of virtual machines. A VMI classification, comprising memory introspection, I/O Introspection, and process introspection, where I/O includes file system, interrupt request and system call introspection, has been proposed [152] and expanded on [153]. However, a VMI classification of memory, network, and disk may enable easier transfer of forensic techniques between physical and virtual environments. VMI researchers frequently study a mixture of these classes as shown in Table 3-1, which is a summary of reviewed major VMI research. VMI techniques for analysing each resource class are discussed in

the following subsections.

Table 3-1: VM Introspection Research Summary

| **Authors** | **System Name** | **Approach** | **VM Resources** |
|---|---|---|---|
| Garfinkel & Rosenblum, 2003[130] | Livewire | Polling dumps and triggers | Memory |
| Kourai & Chiba, 2005[154] | HyperSpector | Mirroring | Memory, Network, Disk |
| Joshi, King, Dunlap, & Chen, 2005[155] | IntroVirt | Vulnerability via predicate checks | Memory, Disk |
| Jiang, Wang, & Xu, 2007[139] | VMwatcher | Memory state reconstruction | Disk |
| Yin, Song, Egele, Kruegel, & Kirda, 2007[156] | Panorama | Taint and flow tracking | Memory |
| Payne, Carbone, Sharif, & Lee, 2008[131] | Lares | Hook process create calls | Memory (code) |
| Hay & Nance, 2008[157] | VIX | Memory copy | Memory |
| Lanzi, Sharif, & Lee, 2009[158] | K-Tracer | Dynamic kernel analysis | Memory (code) |
| Dolan-Gavitt, Leek, Zhivich, Giffin, & Lee, 2011[140] | Virtuoso | Instruction Tracing | Memory (code) |
| Benninger, Neville, Yazir, Matthews, & Coady, 2012[159] | Maitland | Hook memory calls | Memory |
| Harrison, Cook, McGraw, & Hamilton, 2012[160] | - | Forensic analysis of polled data | Memory |
| Fu, Rd, & Lin, 2013[161] | EXTERIOR | Hooks & memory copy | Memory |
| Chung, Khatkar, Xing, Lee, & Huang, 2013[162] | NICE | Network traffic analysis | Network |
| Lengyel, Maresca, Payne, Webster, Vogl, & Kiayias, 2014[163] | DRAKVUF | Kernel debugging | Memory, File |
| Hizver & Chiueh, 2014[149] | RTKDSM | Memory write hooks | Memory |
| Fattori, Lanzi, Balzarotti, & Kirda, 2015[164] | AccessMiner | Model File activity hooks | File + registry |
| Saxon, Bordbar, & Harrison, 2015[165] | - | Entropy analysis | Memory |
| Shi, Yang, Li, & Wang, 2015[133] | SPEMS | Kernel debugging | Memory |
| Taubmann, Frädrich, Dusold, & Reiser, 2016[128] | TLSkex | Intercept packets & memory extract | Memory, Network |
| Zhan, Ye, Fang, Du, & Su, 2016[150] | CFWatcher | Triggers on dentry | File |
| Tien, Liao, Chang, & Kuo, 2017[166] | - | Sandbox on VM copy | Memory |
| Sentanoe, Taubmann,& Reiser, 2017[129] | - | System call analysis | Memory |
| Upadhyay, Gohel, Pons, & Lagos, 2018[167] | - | Kernel data monitoring | Memory |
| Zhan, Li, Ye, Zhang, Fang, & Du, 2019[168] | - | Kernel data monitoring | Memory |

### 3.3.2 Disk Introspection

VMI disk monitoring is not much researched. Technically possible as a virtual machine file system is accessible, remotely, through the creation of a shadow copy and tracking of file operations, it could be an interesting research area. In preliminary work for this thesis, virtual machines were monitored for unusually high quantities of file operations, as such activity might indicate the presence of ransomware. This approach rapidly discovered Hidden Tear [169] ransomware keys in memory.

Justifications given for the dearth of research are that such techniques may be trivial [141], or the host file system structure is unknown [139]. Nevertheless, cloning and kernel level system call intercept techniques have been developed.

Cloning has enabled detection of suspicious access to sensitive files. For example, in HyperSpector the Tripwire file integrity manager [170] executed on a virtual machine shadow copy to discover whether specific files had changed [154]. Similarly, a high-level abstraction of the virtual machine file system was generated from a shadow copy in VMwatcher and anti-virus software applied to the abstraction to detect malware [139]. Potentially malicious file modifications such as permission changes or deletions were detected by performing vulnerability analysis on a virtual machine copy in Introvirt [155]. Generating and monitoring multiple virtual machines shadow copies frequently or continuously may not scale.

Instead of file system monitoring, disk activity can be ascertained through analysing virtual machine kernel memory. To illustrate, file system data structures such as buffer caches were examined for potential malware in Livewire [130]. Other systems have inspected the kernel heap to identify artefacts including open files [171] [160] [163]. Kernel file activity can also be used in buildings models of benign activity, which then enable the detection

of anomalous behaviour [164]. A potential disadvantage to that approach is that such models require frequent retraining as benign activities evolve. CFWatcher intercepted system calls and triggered alerts when the directory entry objects of identified files were accessed [150]. Disk activity monitoring may signal the presence of malware but does not aid decryption.

### 3.3.3 Memory Introspection

Memory introspection describes the analysis of virtual machine memory [152]. As malware is unobfuscated on execution, VMI research into memory introspection has concentrated on malware detection and analysis as shown in Table 3-1. Researchers can also derive requisite file system and network structures from memory data structures. Generally, research has focused on image, kernel, or process introspection.

**Image** Introspecting complete virtual machine memory images may be problematic. Each extraction interrupts virtual machine activity so large acquisitions can degrade its performance. Consequently, when multiple extractions are needed, researchers typically extract the full image once and then maintain consistency by applying changes from the virtual machine to the copy.

In HyperSpector, this approach supported an intrusion detection system that executed on the copy [154]. In another study, the copy maintained a taint status for each byte and status anomalies with benign taint baselines indicated potential malware [156]. Typically, a virtual machine helper process communicates changes to the VMI monitor.

Single complete images have been analysed to discover cryptographic artefacts. For instance, RSA keys were discovered by searching for ASN.1-like objects in an image copy [125]. Similarly, a complete image was searched for large prime numbers and their

products enabling Apache server RSA keys to be discovered in Linux servers [165]. For traffic decryption, multiple image copies may be required, potentially through the use of virtual machine helper processes, which are detectable, and therefore manipulable.

**Kernel** Being smaller, kernel data structure acquisition has less virtual machine impact. Furthermore, as malware sometimes manipulates kernel structures for concealment, determining whether sensitive kernel data has been overwritten may be useful. For instance, in Livewire, kernel memory regions susceptible to malware manipulation, such as Windows system call tables, were set to read-only [130].

Kernel data has been the focus of VMI malware studies. K-Tracer traced malware activity through identifying those instructions associated with sensitive kernel data [158]. It required a hypervisor that emulated virtual machine system calls, which is less common in modern hypervisors. In EXTERIOR, a managed virtual machine replicated the virtual machine kernel. Changes in the managed kernel data structures were written to the virtual machine to enable recovery from malicious activity such as rootkits [161], similarly to Nixer [172]. VMI can also support the protection of kernel data [168]. However, cryptographic artefacts are not commonly stored in system data structures.

Cryptographic artefacts may be system call parameters that can be obtained with breakpoints or kernel function hooks. In Lares, hooks were inserted into kernel process creation system calls [131]. A hook invoked a local helper process, which requested monitor assessment of the created process's potential maliciousness. Malware has been detected by hooking memory management system calls and checking whether data regions have been made executable [159] but these calls are required to invoke hypervisor functions directly. Researchers have identified malicious activity by causing suspect writes to sensitive kernel data regions to gener-

ate page faults causing alerts [149], however page fault monitoring incurred an excessive performance overhead. DRAKVUF tracked malware manipulation of kernel memory with breakpoints on internal kernel instructions, such as heap memory allocations, as kernel mode rootkits can bypass system calls [163].

Kernel memory techniques, such as hooks and breakpoints, may be unacceptably invasive for real-world scenarios. Furthermore, encryption processes may not be kernel functions.

**Process** Process introspection can assist in discovering cryptographic artefacts. This term is meant as the inspection of process user space. Thus, kernel data investigations, such as the analysis of process headers to determine suspect process creation [173], or the discovery of hidden processes through comparing processor usage times [174], are kernel introspection, whereas finding changes in program executable code across the temporal frame is process introspection [175]. The distinction may be unclear as, for instance, when user-space system calls are trapped to appropriate stack contents for analysis, an active area of research [176] [152] [177] [178].

Process user-space data regions, such as the process stack, heap, or static variables can contain useful information. For example, introspecting process heaps has discovered large quantities of no-op instructions, often an indicator of malicious code being inserted on the heap [179], or detected stack and heap buffer-overflow attacks [180]. Process data structures can also be constructed when other information sources such as binary symbol tables are available [181].

Cryptographic artefacts may exist in process user-space data regions. In one study, Windows virtual machine memory was searched for software application virtual disk and session encryption keys when systems were in specific operating system states such as: live; hibernated; user logged off; and rebooted [120]. No

session keys were discovered for stand-alone encryption applications, such as WinZip, or secure network communication applications, such as Skype or Simp Lite MSN. This was thought to result from proprietary cryptographic data structures or obfuscation mechanisms, but with ephemeral data structures, extraction timing is critical.

TLS AES-CBC keys have been discovered by introspecting process user-space data regions [128]. After identifying the Linux process through searching the file descriptor table for associated socket details, process memory was scanned for potential keys. Memory segments were assessed for randomness using bit counts, and prospective keys, then evaluated by comparing decrypted and transmitted authentication codes. That study analysed Linux memory so Windows virtual machines, considered more prone to malicious activity [9], were not addressed. Furthermore, AES-CBC key discovery depended on obtaining memory extracts at a specific moment.

### 3.3.4 Network Introspection

Virtual machine network traffic must be captured so it can be decrypted. Virtualised networking implements virtual software switches at the hypervisor level, enabling virtual machines to communicate internally and externally. A VMI monitor accessing traffic passing through the switch, is arguably not *introspection*. Analysing memory network structures is possible, but for complete network sessions, memory acquisition might be required for every packet, with a potentially deleterious impact on virtual machine performance. Nevertheless, this approach was adopted in Livewire where the network monitor checked for unusual port usage [130]. Generally, network introspection studies examine virtual network switch traffic either through port mirroring, promiscuous monitoring, or packet interception.

**Port Mirroring** Port mirroring has been used for intrusion detection. For instance, virtual machine server ports were mapped to VMI intrusion detection system ports to detect incoming malware in HyperSpector [154]. A customised operating system was used so that technique may not be transferable. The Open vSwitch software bridge [182] supports port mirroring, and was used in the NICE system [162] to scan for malware traffic using the Snort intrusion detection tool [183]. Virtual machine network profiling produced alerts, to which the monitor responded by blocking all traffic from the suspect, diverting traffic for deep packet inspection, or re-configuring the virtual machine's network configuration. Packet interception might provide more timely control over malicious activities. Open vSwitch was also implemented in DRAKVUF in order to provide virtual machine isolation to limit cross-contamination rather than provide network introspection [163].

**Promiscuous Monitoring** Promiscuous monitoring captures all accessible network traffic. Promiscuous network traffic monitoring tools, such as 'tcpdump' [184] and VMware ESXi 'pktcap-uw' [185], are network introspection enablers. For instance, the tcpdump libpcap library was used to capture network traffic from untrusted virtual machines to detect network intrusion [186]. The monitor's passive nature prevents action, such as dropping packets, being taken ex post facto.

**Interception** Network traffic interception interposes the monitor between the source and destination and so offers active control opportunities [162]. To illustrate, a proxy application inspected packets to identify specific TLS messages and trigger memory extractions in TLSkex [128]. With this approach virtual machine operation is unaffected until a specific packet is detected but also allows for immediate action, such as interacting with the virtual

machine. Triggered extraction is an example of monitoring frequency.

### 3.3.5 Monitor Frequency

The frequency of VMI monitor information gathering is largely dependent on its purpose. In real-world environments, excessive monitor activity can degrade virtual machine performance, whereas for dynamic malware analysis in laboratory conditions, VMI activity levels may be inconsequential. Monitoring frequency can be categorised as being: polled; semi-continuous; or triggered.

**Polled** Polled monitoring reduces virtual machine impact by introspecting at intermittent intervals. With polling, essential virtual machine data may be absent precisely when required, should the data be ephemeral and the poll interval exceed the data lifetime. Nevertheless, researchers have used polled VMI monitoring.

Livewire periodically checked virtual machine integrity by comparing monitor results with user-level program output, comparing virtual machine process hashes, searching for malware signatures and checking raw sockets usage [130]. Other researchers extracted memory, file, and network data at polled intervals to reconstruct the virtual machine states for detecting anomalous behaviour [160]. Polled inspection of virtual switch traffic was employed for detecting potential malware activity in NICE [162].

For certainty of detection, frequent polling with degraded virtual machine performance might be required. Polling to detect malicious activity may be particularly risky especially when malware uses counter analysis techniques, so despite reduced virtual machine impact, polled monitoring may be insufficiently determinate.

**Semi-continuous** Semi-continuous monitoring interrupts virtual machine operations frequently, either through breakpoints or hooking. Despite its potential virtual machine performance impact, the approach can be useful. To illustrate, setting break-points on virtual machine internal kernel calls enabled the DRAKVUF monitor to analyse malware activity [163]. AccessMiner activated the monitor for each file operation but the performance impact was sufficiently severe that it was restricted to specific applications [164]. This approach risks missing malicious activity when malware employs techniques such as process hollowing to evade detection.

**Triggered** The impact of triggered monitoring impact depends on how often the trigger is activated. Typically, monitoring is activated when identified virtual machine events occur. For instance, the CFWatcher monitor was activated when virtual machine directory entries associated with sensitive files were accessed [150]. When the quantity of monitored sensitive files increased, virtual machine performance degraded so triggers need judicious selection.

In this discussion, semi-continuous and triggered monitoring are distinguished by the former activating when specific functions are called, and the latter on the detection of specific data. An illustration of a trigger is the detection of a TLS Change Cipher Specification message [128].

Triggers may indicate appropriate moments for extracting virtual machine memory as the memory might contain information needed for decryption. Triggering may, therefore, require less computational resources than semi-continuous monitoring, and so have less impact on virtual machine activity. Furthermore, it is more deterministic than polled monitoring.

## 3.4 Conclusions

Side-channel attack approaches are limited in their capacity to rapidly decrypt terse network communication sessions, encrypted with algorithms such as AES and ChaCha20, in live environments. Such attacks commonly require substantial quantities of selectable plaintext-ciphertext pairs and specialised measurement equipment. However, malware command and control communication sessions may be composed of few packets, plaintext may not be selectable, and the placement of equipment may not be possible without prior knowledge.

Volatile memory, and in particular, virtual machines offer less restrictive opportunities for discovering cryptographic artefacts to enable decryption of real-world network traffic. Virtual machine monitoring provides access to virtual machine resources. External monitoring, VMI, limits monitor manipulation and through integration with forensic tools, addresses the semantic gap issue. Furthermore, a triggered approach to deciding when to obtain memory ensures that virtual machine impact is minimised.

Although studies into discovering encryption keys in virtual machine memory exist, their scope has been limited to key discovery. Encryption keys may not suffice to obtain plaintext from ciphertext. Despite initialisation vectors, or nonces, being essential elements in the encryption and decryption processes for major symmetric block and stream algorithms, there are no published investigations into their discovery in volatile memory. Publication of this thesis addresses the knowledge gap. Further research will continue to build on the approaches developed to construct the framework.

Approaches for identifying initialisation vector and nonce artefacts differ from those of key discovery. For example, with shorter artefact lengths, randomness measurements may be limited for the efficient identification of artefacts. This thesis investigates

the construction of these artefacts for encryption algorithms commonly used in SSH and TLS protocols and develops novel approaches to speedily identify small sets of candidate artefacts in memory. With this knowledge new approaches to discovering candidate keys are incorporated in the framework leading to the rapid decryption of live secure network sessions.

Decryption provides useful knowledge of the confidential information exchange by malicious actors including their malware, and the timeliness of this knowledge is important. So, this thesis investigates efficient methods of discovering the initialisation vectors and encryption keys of encrypted communications in the volatile memory of live virtual machines. Although the investigation focuses on AES and ChaCha20 algorithms, the techniques may be applicable to cryptographic artefacts discovery for other encryption algorithms.

# Chapter 4

# MemDecrypt: A Framework for Decrypting Secure Communications

## 4.1 Introduction

Knowledge that decrypting secure communications can be beneficial, cryptographic artefacts may be found in computer memory, and VMI offers a scalable, stealthy approach for obtaining virtual machine memory, motivates the proposed line of research. This chapter presents a framework for decrypting virtual machine secure communications through memory acquisition and analysis.

The framework, MemDecrypt, contains memory acquisition, protocol decomposition, memory search, and decrypt analysis modules. Although focused on specific technologies, encryption algorithms, modes, and operating systems, the framework is intended to be extensible. The base framework implementation follows a rapid application development cycle [187] to enable the addition of extensions for protocol, algorithm, and operating system variants. The development approach proceeds through the following stages: requirements definition; design; and construction, including evaluation.

## 4.2 Requirements Definition and Terms

MemDecrypt requirements can be derived from the research objective *'a framework to capture live virtual machine memory and encrypted network traffic, discover small sets of candidate cryptographic artefacts in the captured memory, and rapidly decrypt the encrypted network traffic'*. This objective can be divided into five essential activities:

- Extraction: capturing live virtual machine memory
- Interception: capturing live virtual machine traffic
- Interpretation: inspecting virtual machine traffic contents
- Identification: discovering small sets of candidate artefacts
- Decryption: rapidly decrypt the encrypted network traffic

The choice of virtual machine environments was predicated on memory acquisition criteria of scalability, undetectability, and non-invasiveness so these requirements are considered in the framework design. Rapidity is also a consideration as it is a determiner of MemDecrypt's utility. In the following paragraphs each requirement is described in further detail and relevant terms defined for consistency.

**Extraction** acquires the memory of an untrusted virtual machine process from outside the virtual machine. A *virtual machine* is a software system including operating system and applications with the appearance of a system on a physical machine and that runs on software that supports access to physical machine hardware. The software that a virtual machine runs on is a *hypervisor*. As hypervisor consoles manage many modern hypervisors, for brevity, the term hypervisor will also reference its console. *Untrusted* virtual machines generally execute at lower privilege levels than hypervisors or *trusted* virtual machines. As a

result, untrusted virtual machines may be compromised by malicious activity, such as malware, so their activities may be suspect. *From outside the virtual machine* means that virtual machine resources are accessible by an application executing on the hypervisor. *Memory acquisition* is the activity of obtaining the memory contents of a process executing on the virtual machine in a form accessible to an external application, where the form is, typically, a binary file or raw memory blocks.

**Interception** captures network traffic originating from, or received by, an untrusted virtual machine. In all instances, the initiator in a network session, *a client*, is an untrusted virtual machine within the monitored environment. In many instances, the receiver in a network session, *a server*, is also an untrusted virtual machine, although the client may also communicate with external servers. All virtual machine traffic is required to traverse a monitored virtual switch, enabling capture.

**Interpretation** inspects network packet contents to determine whether further analysis is required. Fields for IP (OSI layer 3), TCP or UDP (OSI layer 4) and application protocols including SSH, TLS and DNS (OSI layer 7) may warrant examination. For example, IP source and destination addresses can identify specific untrusted virtual machines, while TCP fields can determine whether a supported application protocol is being used.

**Identification** discovers small sets of cryptographic artefacts in memory obtained from the extraction activity. *Cryptographic artefacts* are encryption keys, initialisation vectors (or nonces), and counters. An *encryption key* is the random sequence of bits used by a symmetric algorithm to perform an encryption or decryption process. An *initialisation vector* (IV) is a sequence of bits used in AES modes to ensure identical plaintext produces different ciphertext, while a *nonce* is a sequence of bits used in ChaCha20 to generate a key stream.

The smallness of a *small set* is contextual and determined by

computational resources, the mode of operation, and framework efficacy. A multi-threaded VMI monitor on a parallel processing device may investigate larger sets than an equivalent single processor device. Mode set sizes may also vary. For example, AES-CBC mode IVs are obtained from the network packet or preceding encrypted block so that the IV set size is one, whereas AES-CTR set sizes are less determinate and set sizes exceeding one thousand are possible. Implementation and evaluation approaches should consider these factors.

**Decryption** identifies correct session cryptographic artefacts by validating the decrypt for various protocols. Generally, the correct key and IV are identified for successful decryption. Validation may depend on message contents. For example, decryption that correctly produces binary strings unintelligible only to session parties could be difficult to validate. However, protocol fields, such as HTTP header fields, can provide clues to valid decrypts.

**Scalability** requires the framework to investigate multiple virtual machine sessions simultaneously. In this thesis, scalability is achieved by creating framework copies. Multi-threading of activities is likely to be a more efficient solution.

**Non-invasiveness** is the degree of impact on virtual machine normal operations. A significant impact might contaminate the virtual machine's run state and disrupt process activity such as live network sessions. Contamination can invalidate memory and network session extracts limiting analysis so this requirement aims to minimise virtual machine impact.

For **undetectability**, monitor operations should be segregated from untrusted virtual machine programs. Malware can detect and modify other programs executing on a virtual machine [188] so as to evade detection or analysis. Any tampering with the monitor application may invalidate framework results so monitor software should execute at a more privileged level than an untrusted virtual machine.

**Rapidity** requires the period between session packet capture and valid decryption to be sufficiently brief for MemDecrypt to have application in real-world scenarios. Applicability is contextual as the impact of malicious behaviour may be almost instantaneous when a spam campaign is launched by a botnet command, whereas exfiltration of large databases may take seconds, minutes, hours, or, perhaps, days [189]. In general, rapidity is a key design consideration.

## 4.3 Design

VMI provides access to virtual machine resources from outside the virtual machine. A virtual machine monitor application running on the hypervisor implements VMI. This thesis focuses on VMI monitoring of memory and network traffic virtual machine resources. The following sub-sections describe an architecture for monitoring these resources and present details of individual architectural components.

### 4.3.1 Description

A simple VMI architecture comprises a hypervisor and a monitored virtual machine. For these investigations, the architecture is extended to enable one virtual machine, *the client*, to communicate securely with another virtual machine, *the server*. Although the server could reside external to the virtual environment, a local virtualised server is generally preferred.

One reason for using internal servers is that, in specific experiments, malware programs are executed on the client. Aside from legal considerations and Internet Service Providers possibly blocking such communications, malware activity introduces unjustifiable risks to co-located devices. In non-malware experiments the framework also investigates decrypting secure communications

by analysing server memory. It is expected that more accurate results can be obtained by excluding extraneous factors such as traffic bottlenecks or server shutdowns.

Virtual machine traffic generally traverses a hypervisor virtual software bridge. A hypervisor application, such as a VMI monitor, can intercept and analyse each packet. If the VMI monitor suspects unusual behaviour, activities such as: manipulation of the virtual machine's environment; the collection of virtual machine memory; or disruption of the network connection may be performed. For framework analysis, virtual machine memory is acquired. Figure 4-1 depicts a data flow between untrusted virtual machines through the hypervisor console.

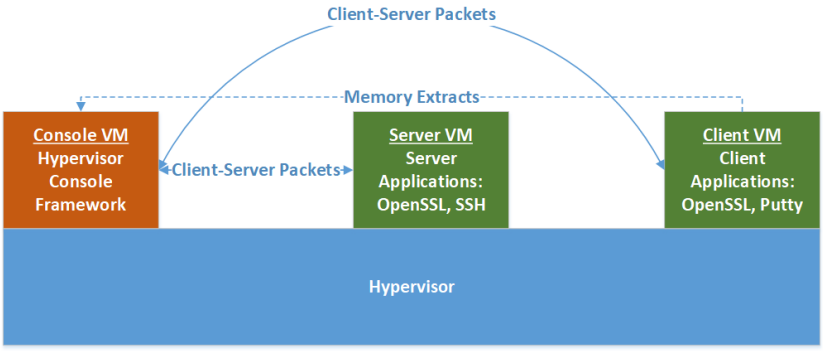



Figure 4-1: High-level architecture

The framework requirements of interception, interpretation, extraction, memory analysis, and decrypt analysis can be mapped directly to five logical components as shown in Table 4-1. This decomposition enables individual components to be extended or replaced for different technologies, protocols, and algorithms. Functionally the components can be described as:

- **packet interception** captures each virtual network packet
- **packet interpretation** inspects packet send and receive IP addresses, and TCP and UDP port numbers. Interesting

packets are retained for implemented protocols, and protocol fields parsed to obtain relevant details

- **memory extraction** determines the virtual machine process associated with the network transaction and acquires useful process memory when appropriate
- **memory analysis** discovers sets of candidate cryptographic artefacts such as encryption keys and IVs (or nonces)
- **decrypt analysis** checks packet stream decryption with sets of candidate cryptographic artefacts until a valid combination is discovered

Table 4-1: Logical Component Mapping to Requirements

| Requirement | Component |
|---|---|
| extraction | memory extraction |
| interception | packet interception |
| interpretation | packet interpretation |
| identification | memory analysis |
| decryption | decrypt analysis |

In the framework, interception and interpretation components are linked to enable real-time analysis. Maintaining separation is possible if subsequent activities such as memory extraction are performed without the need for interception or interpretation. In this thesis, the interception component invokes interpretation yielding the framework activity flow diagram shown in Figure 4-2.

In virtual environments, further efficiencies are obtained by integration. Packet interception, packet interpretation, and memory extraction relate to a virtual machine whereas memory and decrypt analysis are technology independent. Integrating virtual machine-related components may enhance framework performance, albeit with a possible loss in extensibility. Consequently, MemDecrypt consists of a data collection component incorporating inspection, interpretation, and memory extraction elements, a

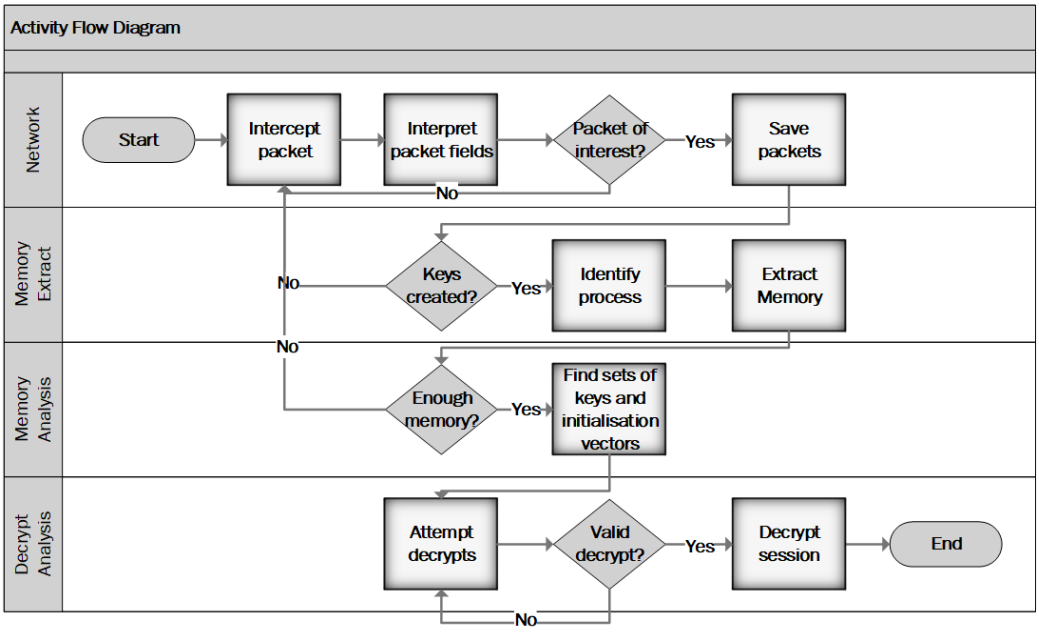



Figure 4-2: Framework Activity Flow Diagram

memory analysis component, and a decrypt analysis component. Figure 4-3 illustrates the chaining between these three MemDecrypt components. For enhanced performance, component output quantities are minimised to reduce subsequent component processing. The following sub-sections detail individual component designs.

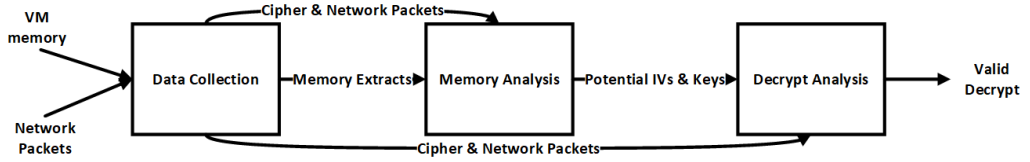



Figure 4-3: Analysis Framework Component Interaction

### 4.3.2 Data Collection Component

The component monitors an untrusted client or server virtual machine and acquires data for the analysis components. Framework

efficacy is achieved by restricting data acquisition to relevant network packets and potentially useful memory. This approach minimises the virtual machine impact.

Each packet is analysed for possible suspect activity. Indicators can be: DNS requests for unknown domain names; unknown protocol packets; and packets associated with known protocols, such as HTTP, VPN, SSH, and TLS, targeting non-whitelisted IP addresses. All packets are ultimately forwarded to the destination IP address. When activity is suspect, packets are retained for subsequent analysis. For SSH and TLS protocols, packet fields provide important information for analysis and, also, the appropriate stage for memory extraction. For instance, encryption algorithms, authentication algorithms, modes of operation (for AES), and encryption key lengths are important. The information is gleaned from specific protocol fields in messages such as SSH 'Key Exchange Initialisation' and TLS 'Hello'.

Essential MemDecrypt information is obtained from virtual machine memory acquisition. Identifying virtual machine processes associated with suspect activity supports restricting memory extract sizes. As open network connections are associated with specific ports and processes, matching virtual machine connections and network packet port numbers can identify the process. However, if different processes perform encryption and packet transmission, then the memory of an incorrect process may be extracted. So, as a precaution, virtual machine process lists at network session start are compared with lists at the time of memory extraction. New processes may suggest possible encryption processes, although other processes not associated with encryption activity may also have been created.

Process memory extracts are restricted. Because framework performance is highly correlated with memory extract sizes, only memory potentially containing encryption artefacts is acquired. As artefacts are commonly negotiated and generated by client

and server endpoints during the network session, a generating process will store these in writable memory. Furthermore, the artefacts are process data fields and, so, found in user-level, as opposed to kernel, memory. To limit memory extract size, the component extracts user-level writable memory of the identified process. Memory extracts are acquired when the probability that extracts contain cryptographic artefacts is high. Generally, this occurs after completion of a protocol client-server handshake, during which parameters used in generating cryptographic artefacts are exchanged. Secure communications protocols incorporate different handshake processes so extensions are developed for each and described in later chapters. Extracted memory, network packets, and algorithm details are retained for memory and decrypt analysis.

### 4.3.3 Memory Analysis Component

The component searches for candidate cryptographic artefacts in extracted memory. IVs vary in composition and size for protocols and encryption modes so component protocol extensions address variations. By contrast, encryption key characteristics are similar across protocols, encryption algorithms, and modes. Core component features of candidate artefact identification are prioritising searches for candidate IVs ahead of candidate keys and techniques to determine candidate encryption keys.

Candidate IV locations are identified before candidate key locations. For each variation, IV characteristics enable the discovery of smaller sets of candidate IVs than keys. Cryptographic applications may store the artefacts in common data structures. Consequently, identifying candidate IVs in memory and using their memory locations to ascertain candidate keys from nearby memory segments may improve framework efficiency.

Key characteristics restrict the number of memory segments

being candidate keys. One characteristic is randomness. Encryption algorithm key randomness aims to reduce the potential for key discovery by trial-and-error. For instance, RSA key certificates used for TLS and SSH protocols were discovered when keys were insufficiently random [190]. Various interpretations of information theoretic randomness exist. Von Mises proposed that an event sequence is random if each sequence element is unpredictable [191]. With this meaning, the probability of a sequence element in a binary sequence being 0 or 1 is equally likely. This interpretation was applied in a search for TLS keys [128]. However, the example of a sequence of alternating 0s and 1s suggest a possible weakness in von Mises' definition.

Instead, specifying a sequence, or string, of events to be random if there is no description shorter than the string itself [192] may offer greater rigour. Restated, this means a string of bits is random if it cannot be described algorithmically as a shorter string of bits. Random strings, such as encryption keys, generated by computer programs are algorithmic and so, may not be purely random, or, as eloquently yet brutally stated:
"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" [193].

Nevertheless, computer pseudo-random number generators are frequently used. Randomness measures may distinguish memory segments containing keys from other memory segments so measures that differentiate segment randomness are useful.

Entropy measures the randomness of memory segments. In information theory, entropy calculates the uncertainty of predicting the value of a string. In other words, entropy measures how difficult it is to predict an observation [194]. Shannon entropy, min-entropy and guessing entropy are considered the most useful information theoretic entropy measures for cryptography [195]. Min-entropy and guessing entropy both evaluate the probable number of attempts required to discover a sequence. Shannon

entropy evaluates the degree to which a key can be compressed. As min-entropy and guessing entropy adopt similar approaches, the benefits of Shannon entropy and guessing entropy only are compared for thesis purposes.

Shannon entropy provides a measure of the average uncertainty of a string of bits. It calculates the average number of bits which can describe the string [195]. As an example, if the sequence of bits consists of a character repeated many times, it is predictable and the entropy is 0. The Shannon entropy formula is given by [196]:

$$H = -\sum_{i=1}^{n} p(i) \log_2 p(i) \tag{4-1}$$

where $p(i)$ is the normalized frequency, $f(i)$, of the $i$th element in the sequence i.e. $p(i) = f(i)/n$. For encryption keys to be unpredictable, their H values should exceed a threshold, where longer sequences have higher thresholds.

Guessing entropy evaluates the probability of guessing the value of a discrete random variable [197]. A definition of guessing entropy is given by [195]:

$$E[G(x)] = \sum_{i=1}^{n} ip(i) \tag{4-2}$$

where $G(x)$ is the number of guesses to identify the correct value $x$ and $p(i)$ is the probability of an erroneous guess at the $ith$ attempt. The greater the number of guesses before obtaining the correct value, the greater the randomness.

For the framework, small candidate key sizes are preferred as the set size is proportional to computational processing effort and hence analysis durations. The maximum entropy measure that always identifies key segments, *the threshold*, determines candidate set size. The graphical comparison in Figures 4-4 and 4-5, which maps the entropy measure (Y-axis) against memory offsets

(X-axis), suggests that either entropy measure might suffice. In experiments with 256-bit keys, Shannon key entropies exceeded 4.5 and guessing entropies 38,000. However, with these thresholds, the guessing set size is ten times the Shannon entropy size making the Shannon entropy measure more efficient for comparing segment randomness.

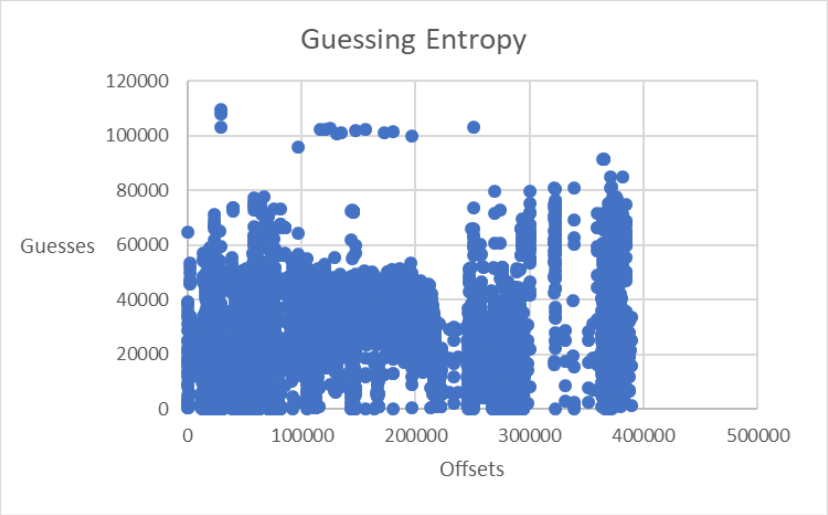



Figure 4-4: Guessing Entropy

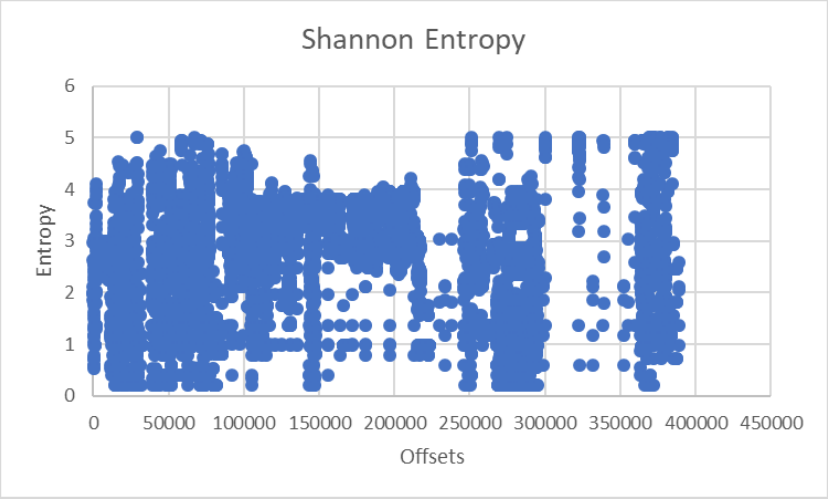



Figure 4-5: Shannon Entropy

Entropy measurements are useful for key discovery only if key entropy is sufficiently different from non-key memory segment entropy. An experiment was conducted to analyse the entropy distribution for sets of secure communications applications that use encryption. Figure 4-6 illustrates a typical distribution, where the count of memory segments exceeding an entropy (X-axis) maps against that entropy level (Y-axis) so that, for example, whereas out of 264,813 segments exceeding 0.0 entropy, 188,602 (i.e. 72.1%) exceed 2.0, 2,628 (i.e. 0.99%) exceed 4.5. Segments with 0 entropy are approximately 95% of all sets and are not shown for illustration purposes.

So, for 256-bit keys, an entropy threshold of 4.5 provides differentiation between candidate keys and other memory regions. While the Shannon entropy maximum for 256-bit keys is 5.0, for shorter key lengths the maximum and the minimum threshold decrease leading to larger candidate key set sizes. Other characteristics can reduce these set sizes.

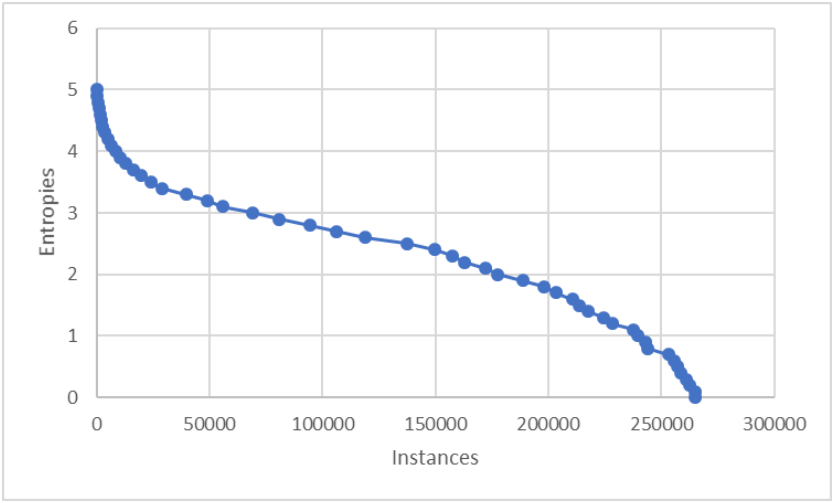



Figure 4-6: Cumulative Entropy Distribution Example

Staticity in a network session is an encryption key characteristic. After completion of the protocol handshake, the generated keys support encryption and decryption until client and server

negotiate generation of new keys. Keys are generally input parameters to encryption processes, and therefore may be memory resident in a cryptographic data structure throughout the session. So, memory extracts taken at different stages in a session may contain keys. Furthermore, if memory extracts are acquired on subsequent iterations of a virtual machine process function executing, such as network packet transmission, keys may be at the same location for each extract.

Another key characteristic is the distance between the memory extract locations of encryption keys and IVs. During the protocol handshake, the parties agree on the algorithm, mode of operation, and key length for encrypting confidential information. This enables a parameter exchange so that cryptographic artefacts can be generated. Cryptographic data structures are therefore often created at run-time to store fields such as encryption/decryption flag, key size, key, and IV (or nonce). Consequently, keys and IVs may be present in memory locations a short distance apart and thus in the same read/write memory extract.

These characteristics can enable the prioritisation of valid keys over other candidates. Specifically, memory segments near the memory locations of candidate IVs, with contents that are static between memory extracts, and with a Shannon entropy above the threshold are candidate encryption keys. The memory analysis component uses these insights to identify candidate cryptographic artefacts for processing by the decrypt analysis component.

### 4.3.4 Decrypt Analysis

The component searches for valid artefacts through an iterative process of decryption and verification. It generates decrypts for the first encrypted block in network packets by iterating through candidate artefact sets produced by memory analysis. This process terminates when the decrypt is verified. Although invalid

decrypts are generally random strings and therefore have high entropy, valid decrypts such as binary file blocks or dynamically generated URLs may also exhibit high entropy so decrypt entropy measures may not be safe for verification. Protocol features can provide higher confidence in decrypt validity and are built into protocol extensions. A valid decrypt identifies cryptographic artefacts enabling other session packets to be decrypted and reviewed.

## 4.4 Construction

MemDecrypt construction implements the proposed design using existing platforms and tools with bespoke software. Fully bespoke solutions may provide increased security for the monitor [164] and also be more efficient, but the additional development effort may not offset investigative benefits. *Bespoke* means that the software is written specifically for the framework.

A review was conducted of available hypervisors, packages, and tools for each component. Key selection criteria for each element were that they be non-commercial and able to integrate seamlessly where necessary. Most reviewed packages and tools permitted integration with Python programs so bespoke MemDecrypt software was written in the Python 2 programming language and is anticipated to be upgradeable to Python 3 with minor alterations.

### 4.4.1 Hypervisors

Custom hypervisors are not used. Although researchers have developed instances, e.g. Hyperspector [154] and AccessMiner [164], the review of major VMI studies, summarised in Appendix B, found this to be atypical. For MemDecrypt, the hypervisor should: support full virtualisation; be minimally exposed to tampering; and support application-level memory mapped memory extraction from live virtual machines. The principal non-commercial

hypervisors [198], VMware vSphere/ESXi, Microsoft's Hyper-V Server 2012, Xen, and KVM, are reviewed after a discussion of these criteria.

**Full virtualisation** is more suited to MemDecrypt requirements than para-virtualisation and operating-system level virtualisation. Whereas full virtualisation translates virtual machine instructions for hardware access into host machine instructions using software or hardware translation, para-virtualisation modifies the virtual machine operating system to provide direct hardware access. As an example, a para-virtualised Windows virtual machine operating system requires specific drivers [199]. Formerly, para-virtualisation performed significantly better than full virtualisation but with Intel VT-x and AMD-V instruction sets, full virtualisation performance is considered no longer to be a significant limitation [200] and for servers, full virtualisation is more common [201]. In operating-system level virtualisation, sets of applications run independently on a common operating system hosting a container platform, so for experiments with Windows and Linux, two virtualisation environments are necessary. Thus, only full virtualisation enables multiple, unmodified virtual machine operating systems to co-exist on a single host.

**Bare-metal** hypervisors are preferred. An aspect of hypervisor security is limiting the potential for tampering, in particular by virtual machine software detecting and disrupting hypervisor applications, called 'VM escape' [188]. For analysis, hypervisors can usefully be classified as bare-metal or hosted. Bare-metal hypervisors are installed directly onto the physical hardware while hosted hypervisors execute as an application on a base operating system, such as Windows or Unix, which itself runs on the physical hardware. Hosted hypervisors are considered less secure as their base operating systems are large, complex products with vulnerabilities that can facilitate hypervisor compromise [202]. A counter-argument exists that a bare-metal trusted computing

base, comprising hypervisor and hypervisor console, may exceed the trusted computing base of a hosted hypervisor [203]. Operating system vulnerabilities may be more prevalent than hypervisor console vulnerabilities. Owing to the additional layer between the virtual machine and the hardware, hosted hypervisor performance is also poorer than bare-metal [204]. VMI studies using hosted hypervisors, such as Livewire [130] and HyperSpector [154], are therefore exceptions.

**Memory mapping** is desirable. MemDecrypt requires live virtual machine memory extraction. Possible extraction techniques include virtual machine snapshotting or memory mapping. Memory snapshotting writes virtual machine memory to file, which generally requires the virtual machine to be in a powered off or suspended state. Memory mapping translates virtual machine memory to local, hypervisor memory. While snapshotting extracts all virtual machine memory, memory mapping accesses only virtual machine memory pages of interest. For performance, extractions should only extract the user-level read/write memory of identified processes making memory mapping a preference.

**vSphere/ESXi** The VMware hypervisor supports a wide range of virtual machine operating systems including Windows and Linux. The vSphere/ESXi 6.7 hypervisor component has a 129 MB footprint [205]. Most functionality, including a client which provides management capability, is delivered by separate applications for improved performance [206]. Virtualised memory can be acquired by powering-off or suspending the virtual machine. The minimum ESXi hypervisor size is 2GB of physical memory.

**Hyper-V Server** The Microsoft hypervisor supports all Windows and selected Linux virtual machine operating systems [207]. A privileged 'root' partition performs management functions such as virtual machine creation and the enabling virtual machine hard-

ware access [208]. Hyper-V needs a minimum hardware configuration, such as a 64-bit processor with nested paging [209]. Although virtual machine memory can be acquired, for example by using LiveKD [34] program [210], Windows debugging is invoked so the technique applies only to Windows virtual machines. Furthermore, the full virtual machine image is extracted.

**Xen Project** The Xen open-source hypervisor [211] supports virtual machines running a wide range of Windows and Linux variant. The hypervisor supports full virtualisation as well as paravirtualisation. Approximately 1 MB in size, it delivers scheduling, memory management and interrupt handling for hosted virtual machines [211]. A privileged virtual machine (Dom0) manages the hypervisor and also provides hardware functionality such as virtual disk device and network access to other hosted virtual machines. Xen supports LibVMI, a virtual machine memory mapping library [148], so VMI applications can directly access the memory and registers of live Linux or Windows virtual machines. Process details are obtained by mapping to the virtual machine's kernel data to access the process table. Xen Project requires 1 GB of physical memory.

**KVM & QEMU** The KVM [212] hypervisor is a Linux kernel module. As a result, Linux kernel security enhancements automatically apply to KVM. It is argued that KVM is a hosted hypervisor because of the Linux kernel presence [41]. The implementation leads to a large code base, which exposes a larger attack surface. KVM does not provide hardware emulation and relies on user-space implementations such as QEMU to provide this functionality. QEMU [213] provides full virtualisation by emulating the virtual machine for processor and hardware. To emulate, virtual machine binary code is dynamically translated into blocks of instructions for host execution. For performance, KVM

and QEMU are used together to provide processor virtualisation and hardware virtualisation, respectively. The LibVMI library is supported by KVM-QEMU providing virtual machine memory mapping after patching the hypervisor.

The Xen Project hypervisor appears most consistent with the MemDecrypt construction as shown in Table 4-2. vSphere/ESXi and Hyper-V Server map virtual machine memory internally so their functionalities may be available to hypervisor applications, e.g. with vSphere Software Development Kit for vSphere/ESXi [214]. Xen's low minimum memory requirement, support for Windows and Linux operating systems, and access to a memory mapping library, justifies further investigation into its appropriateness for the framework.

Table 4-2: Hypervisor Comparison

| **Hypervisor** | **Memory map** | **Full virtual'n** | **Bare-metal** |
|---|---|---|---|
| vSphere ESXi | - | Y | Y |
| Hyper-V Server | - | Y | Y |
| Xen | Y | Y | Y |
| KVM-QEMU | Y | Y | - |

For MemDecrypt construction, the Xen Project hypervisor is installed on a physical device and configured with recommended partition sizes, the xenbr0 software network bridge, which acts as a switch, and an SSH server to provide isolated communication with management devices [211]. MemDecrypt components execute or are initiated from Dom0. Build details for each component are described in the following subsections.

### 4.4.2 Data Collection

The data collection component obtains virtual memory network traffic and memory extracts. For network traffic, an iptables rule

routes traffic traversing xenbr0 to a NetFilterQueue queue [215] for interpretation. The Scapy Python package [216] deconstructs IP, TCP, and UDP fields in each packet. If a packet indicates suspect activity, such as an unusual destination IP address, bespoke protocol-specific modules analyse each session packet for supported protocols. All packets, including suspect packets, are forwarded to the destination to maintain the network session.

During the set-up process of a secure session for each protocol, the client and server agree on algorithms to be used to enable the secure exchange of data. The encryption algorithm including the mode of operation and key size are retained for use by the memory analysis and decrypt analysis components. For each protocol, a stage is reached during the set-up process when keys and IVs are likely to be in the memory of the process performing encryption.

Memory extraction is implemented differently for Windows client virtual machines and Linux server virtual machines. These approaches are described in the following paragraphs.

**Windows Clients** For Windows virtual machines, where kernel data structures are complex and changeable, a semantic analyser assists in process identification and memory extraction. This enables specific operating system and process structures to be examined to determine the process associated with the unusual event and the extraction of read/write user-level process memory. For reasons of availability, maintainability, and comprehensiveness, memory forensics analysis frameworks are preferred to other semantic analysers as discussed in the Literature Review. Volatility and Rekall are free, open-source memory forensics analysis frameworks that provide semantic analysis.

Volatility [146] derives high-level semantic information from memory image files, principally for forensic investigation. When integrated with LibVMI [148] and PyVMI [217], a Python LibVMI wrapper, Volatility provides live virtual machine semantic

information. This is achieved by invoking low-level calls to obtain kernel symbols, read memory segments, and convert virtual to physical addresses, for example. Volatility does not require virtual machine agents, reducing potential for detection and manipulation. Although Volatility derives Windows and Linux semantic information, there are limitations on its use with Linux images. While the Volatility distribution includes profiles for most supported Windows operating systems, because Linux variations are more plentiful, Linux kernel profiles must frequently be generated. Volatility has fewer memory extraction features for Linux than for Windows. As Linux process memory management structures are simpler than Windows, deriving Linux semantic information may not require a semantic analyser tool.

The Rekall framework [147] branched from Volatility, and although rewritten [218], similarities exist. Functionally, both extract semantic information from image files and live systems. Operationally, both frameworks use plugins and can be used as libraries. Rekall generates operating system profiles using a different approach [219], but, more importantly, requires a virtual machine agent for data collection [220], so Rekall agents may be detected by virtual machine malware. To limit the potential for data corruption, Rekall is less appropriate for MemDecrypt purposes than Volatility.

The data collection component invokes Volatility through installed Python scripts, rather than as a stand-alone executable or an imported library. Volatility functions are accessed through supplied or user plugins [221] so the scripting approach allows for implementation of bespoke plugins and also makes Volatility upgrades, including the addition of operating system profiles and plugins, simpler [146]. Bespoke plugins are implemented for framework efficiency to obtain specific process details as well as extract read/write process memory.

The bespoke *netscantbl* plugin, based on the supplied Volatility

*netscan* plugin, obtains the names, process identifiers, port numbers, and physical offsets of processes with open network ports. An extract of sample plugin output is shown in Appendix C. Matching plugin output network addresses and ports identifies the virtual machine process associated with the network activity.

The *vadanalyse* plugin, based on the supplied Volatility *vaddump* plugin, extracts user-level process read/write memory. The plugin traverses the process Virtual Address Descriptors (VADs) tree that is implemented by the Windows Memory Management system for rapid access to program memory regions [34]. The plugin identifies those VADs with read/write protection flags as potential containers of encryption artefacts and writes the memory to time-stamped files for memory analysis component examination. Only memory-resident VADs are extracted so the virtual machine should be suspended during the plugin operation to prevent memory pages being unavailable.

**Linux Servers** The Linux server implementation uses LibVMI for process identification and memory extraction. LibVMI requires knowledge of specific operating system global symbols and process values. For Linux, the global symbol values are extracted from the system map. Process values are discovered by executing a temporary kernel module on the virtual machine prior to operation. Process values identify field offsets in the process task structure. For example, the *linux_pid* value is the process identifier offset from the process task structure start. Process code, stack, and heap start and end offset values were obtained in addition to those required for LibVMI configuration. Bespoke process identification and memory extraction scripts invoke PyVMI API functions, which call associated low-level LibVMI routines to read virtual machine memory.

Process names are generally known because server applications are started to listen to client requests. Bespoke code traverses the

double-linked list containing process header pointers until process names are matched. Process details, such as identifiers and memory structure pointers, are then extracted using the known process value offsets.

Linux memory management is less complex than Windows so semantic abstraction is not required for process read/write memory extraction. For user-level read/write memory extraction, bespoke code reads the process heap in 4Kb blocks, the smallest Linux memory allocation, and writes them to a time-stamped file. Swapped out blocks cannot be read and are ignored.

### 4.4.3 Memory Analysis

The memory analysis component is bespoke and customised for each operating system, protocol, and the modes within each protocol. In particular, candidate IVs have different characteristics for each protocol and mode. However, candidate key identification is common in each variant.

The randomness characteristic of encryption keys asserts that their Shannon entropies exceed a threshold. Any memory blocks with entropies above the threshold are of interest. The threshold setting is important as a high threshold limits the candidate keys for analysis. Small sets of artefacts such as candidate keys reduce the time required by the decrypt analysis. However, setting the threshold too high may lead to an artefact being missed and invalidate the framework.

Artefact lengths determine entropy threshold. Thresholds settings are determined for possible encryption key lengths (256, 192, and 128 bits) and TLS CTR mode IVs (32-bits). Random strings are commonly produced by pseudo-random number generators such as the OpenSSL toolkit [222] rand function. A 99.99% confidence level for threshold settings can be ascertained by generating 10,000 keys with *'openssl rand nnn'* where *nnn* is requisite byte

count and obtaining entropies. The values are detailed in Table 4-3 together with the actual framework threshold settings.

Table 4-3: Entropy Thresholds

| **Artefact length** | **99.99% confidence** | **Threshold** |
|---|---|---|
| 256 bits | 4.52 | 4.5 |
| 192 bits | 4.08 | 4.0 |
| 128 bits | 3.45 | 3.4 |
| 32 bits | 1.5 | 1.5 |

Memory analysis also identifies candidate key locations by proximity to candidate IV locations. Commencing from candidate IV locations, entropies for key length segments, located at increment multiples from the IV locations, are calculated and segments retained as candidate keys if the entropy threshold is exceeded. As cryptographic artefact sizes are 4-byte multiples, encryption programs probably align the artefacts on word boundaries so the incremental value is generally 4. For Linux, a single heap memory file is analysed for each extraction, whereas for Windows, every VAD file is examined for each memory extraction .

### 4.4.4 Decrypt Analysis

For each protocol, mode, and operating system variant the component carries out iterative decryption using the sets of candidate encryption artefacts produced by memory analysis and retained network packets. The process terminates on the correct decrypt validation or exhaustion of the candidate artefacts. As performance is critical, data such as candidate encryption artefacts and network packets are memory resident throughout the iterative process.

The component comprises bespoke code and two decryption packages. For decrypting AES in CTR and CBC modes with 128, 192, and 256-bit keys, the Python 'PyCrypto' package [223] is

recommended [224], has minimal Python package dependencies, and is easily integrated into MemDecrypt without modification. For decrypting ChaCha20, the Python Chacha20poly1305 package [225] was modified to integrate with the framework.

Framework effectiveness relies on the accuracy of decrypt validation. One approach is evaluating decrypt entropy. The rationale is that valid decrypts may be less random than invalid decrypts and hence have lower entropy. However, in preliminary testing, the decrypt entropy approach was unable to distinguish valid and invalid decrypts with sufficient accuracy, so the approach was discontinued.

Another approach calculates lengths of decrypt strings matching regular expressions containing alphanumeric or common special characters, based on an assumption that valid decrypts will be largely alphanumeric. When 3-byte strings matched a regular expression, large quantities of potential decrypts were generated, while matching longer strings could exclude valid decrypts. Although this approach could probably be improved, protocol-specific decrypt validation results in greater accuracy. Generally, valid decryption identifies the encryption artefacts, which enables complete session decryption and deconstruction.

## 4.5 Evaluation

Experiments were conducted with various protocols, encryption algorithms, and operating systems. Success can be measured by decrypt precision and rapidity. For consistency and comparability, experiments were executed in a standard environment. Evaluation criteria and environment details are discussed in the remainder of this section.

Of known information security measurement philosophies [226], the adopted approach is formulation of a hypothesis and conducting experiments to investigate its validity. For framework utility

in live scenarios, decryption must complete correctly and rapidly, that is, effectively and performantly.

### 4.5.1 Test Criteria

In this thesis, *effectiveness* requires that the probability that sessions are correctly decrypted exceeds a nominal value. Because related domain studies do not decrypt, a commonly accepted value does not exist and is therefore defined within the context of framework usefulness. A tolerance of 1 session in 10,000 to decrypt incorrectly may still miss a vital malicious communication session but compares favourably with network intrusion detection systems. The boundary condition of 99.99% may be challenging in short secure communications sessions that comprise one or two encrypted blocks. With longer sessions, and therefore more encrypted blocks, invalid decrypts are progressively less likely as protocol deconstructs become meaningless.

*Performant* requires that the elapsed time taken to discover the correct combination is sufficiently short for the framework to be useful in real-world scenarios. If decryption is achieved during a live network session, actions, such as session disruption, are possible. Decryption after session completion may nevertheless be beneficial, if, for instance, the decrypt contains a ransomware key, or identifies confidential information that has been exfiltrated. In the absence of published malicious activity network session durations, a target decryption time-period can be set through a consideration of network session lengths.

User-initiated and software-initiated malicious activity session durations are assessed separately, as the former allows for user decision making. Analysis of a publicly available HTTPS traffic dataset established that the network session duration of a very small quantity was below 1 second, and the duration for the remainder exceeded 1 minute [227]. As only the second set con-

tained encrypted payloads, 1 minute may be indicative of typical user-initiated benign or malicious network session lengths.

An analysis of malicious software-initiated traffic flows discovered that sessions may be less than 100 packets [10], which is suggestive of 1 second network session durations. Optimally, then, decrypts should be obtained within 60 seconds for user-initiated and 1 second for software-initiated malicious activities. Durations for non-optimal solutions where the decrypt is obtained after the session terminates depend on the scenario and impact of the malicious activity.

### 4.5.2 Test Approach

Experiments were conducted by establishing network sessions between Windows clients and Ubuntu server virtual machines. In each instance, a client application was executed at the command level on the Windows client after a server listener process was started. Destination server IP addresses and ports were provided as client application parameters, where supported. In other instances, such as client malware, a DNS application running on a virtual machine responded to suspect requests with the IP address of the destination server.

Once client and server awaited input, data collection was initiated from the Dom0 console command line. Monitored virtual machine network traffic and memory extracts were retained, together with identified encryption algorithms and extract message numbers for further analysis. Monitor commands were of the form:

*python intercept.py -p DPORT PROFILE SRC DST*

where DST and DPORT are the server destination IP address and port respectively, SRC is the client IP address, and PROFILE identifies the operating system and version of the monitored virtual machine.

After completion of a monitored network session, the memory

analysis process was initiated from the Dom0 console command line. Using data retained by the data collection component, it identifies and retains sets of candidate cryptographic artefacts. Memory analysis commands were of the form:

*python PROT_PROFparse.py FOLDER*

where PROT identifies the protocol, such as SSH or TLS, PROF identifies the operating system, and FOLDER the location of memory analysis input and output data.

After completion of a memory analysis process instanc, decrypt analysis was initiated from the Dom0 console command line. Using the sets of candidate cryptographic artefacts retained by the memory analysis component, and captured network traffic, it tests decrypts for protocol compliance. Decrypt analysis commands were of the form:

*python PROT_PROFdecrypt.py FOLDER*

where PROT identifies the protocol, such as SSH or TLS, PROF identifies the operating system, and FOLDER the location of retained data.

MemDecrypt effectiveness was measured by executing the framework to obtain decrypts and then calculating the theoretic probability that correct decrypts had been obtained by chance. The assumption is made that each possible result is equally likely. Thus, the likelihood of obtaining a specific three character decrypt, such as 'GET', is 1 in $3^{256}$ as any ASCII value is possible for each character position.

Experiments were conducted varying specific parameters such as user input data, encryption algorithms, and modes. This aimed to ensure that framework effectiveness was not restricted to specific scenarios. For example, for file uploads, multiple text files were generated, and Adobe Portable Document files and Excel files downloaded from the Internet.

Framework performance was measured by obtaining the time difference between the start and completion of specific operations.

For data collection, measured operations were memory extract, while for memory and decrypt analysis, primary measurement was of the complete process. Invocations to the Python logging package were used to generate times to an accuracy of 1/1000 seconds. Component durations were obtained in each experiment to assess framework performance relative to the evaluation criteria and identify anomalies.

### 4.5.3 Test Environment

The physical environment used for thesis experiments is a Core 2 Duo Dell personal computer with 40 GB of disk storage and 3GB of RAM. The hypervisor, which supports all virtual machines, is Xen Project 4.4.1. Virtual machine configurations are provided in Table 4-4. Anti-virus software is not installed on untrusted virtual machines and firewall rules are implemented solely for testing purposes. A single software network bridge provides network connectivity between all virtual machines and any external communications. To enable test runs, packages installed on Dom0 are LibVMI, PyVMI, Volatility and their dependencies (listed in Appendix A), NetFilterQueue, Scapy, and the non-essential tcpdump used for display purposes.

## 4.6 Extensibility

MemDecrypt extensibility can be considered from both a practical and a conceptual perspective. The former determines the requirements for in-scope protocol extensions, while the latter discusses how the framework might apply to technologies, algorithms, and protocols outside the scope of this thesis.

Practically, the data collection component requires analysers for each protocol. Protocol analysis extensions indicat e when cryptographic artefacts may be memory-resident as well as agreed

Table 4-4: Virtual Machine Configurations

| **Role** | **Operating system** | **Memory** | **Disk** | **Installed s/w** |
|---|---|---|---|---|
| VMI | Debian 3.16.0-4-amd64 version 1 | 512 MB | 15 GB | sshd, Volatility, LibVMI |
| Test Client | Windows 7 SP1 | 512 MB | 30 GB | PuTTY, OpenSSL, nasm |
| Test Client | Windows 10 (10.0.16299) | 2 GB | 40 GB | As above |
| Test Server | Ubuntu 14.04 ("Trusty") | 512 MB | 4 Gb | OpenSSL, sshd |
| DNS | Ubuntu 14.04 ("Trusty") | 512 MB | 4 GB | dnsmasq |

encryption algorithms and related information. Memory extraction is a commonly performed technique for Xen hypervisors with Windows and Ubuntu virtual machines. Conceptually, memory extraction may be possible for other hypervisors, as well as other technologies, such as Android smartphones. For instance, where Android smartphone volatile memory is acquired using a Linux tool such as the Linux Memory Extractor ('LiME') application [70], the framework could support decryption of TLS sessions using ChaCha20 without modifying memory analysis or decrypt analysis components.

The memory analysis component focuses on firstly identifying candidate IVs or other traces of cryptographic material and secondly using that knowledge to identify candidate keys. The first element may vary for: each encryption algorithm and mode; protocol, including its version; and software implementation of the protocol. Thus, both practically and conceptually, new extensions *may* be required where new variations are investigated. However, the second, candidate key, location element is common across variations where candidate keys have not been identified by

the first element.

Decryption in the decrypt analysis component varies with protocol and encryption algorithm to enable determination of successive IVs. Where these are determined with a common approach, conceptually a common component may be used. As decrypt validation is protocol specific, new extensions are constructed for each protocol type only. For example, TLS-over-HTTP validation is common for each TLS version, encryption algorithm, operating system, and protocol implementation. So, practically, validation is built into the protocol extension and would need to be constructed for additional protocols, such as IPsec.

## 4.7 Conclusions

This chapter's aim was to present a design and construct a framework that supports decryption of secure protocol communications in virtualised environments. Although the solution may not uniquely solve the challenge, MemDecrypt componentry does support the addition of extensions for different environments, algorithms, and protocols.

Construction selections, particularly regarding hypervisor and memory extraction approach, may require revision. Although Xen is a popular research hypervisor and its capabilities make it useful for investigative purposes, the commercial version ranked only third behind VMware's ESXi and Microsoft's Hyper-V in a 2017 survey of server virtualisation [228]. With VMware's pre-eminence in the market, Volatility's ability to extract process information from VMware snapshots suggest that MemDecrypt could plausibly be implemented on the ESXi hypervisor. The framework construction is also constrained by its application to a single hypervisor instance and therefore does not address the potential impact of virtual machines moving between hypervisor instances. The framework current analysis limit of a maximum of two memory

extracts may also be a mitigating factor.

MemDecrypt uses Volatility plugins for Windows extraction and LibVMI API calls for Linux extraction because of complexities in Windows memory management and kernel data structure changes between operating system versions. However, the MemDecrypt framework requires little semantic analysis. If process identification and memory extraction durations differ significantly between Windows and Linux, it may be expedient to replicate the requisite Volatility semantic analysis in MemDecrypt.

The constructed framework meets the research objective *a framework to capture live virtual machine memory and encrypted network traffic, discover small sets of candidate cryptographic artefacts in the captured memory, and rapidly decrypt the encrypted network traffic.* Furthermore, the solution scales, may be difficult to tamper with, and minimally impacts monitored virtual machines.

# Chapter 5

# Determining Insider Attack Data Exfiltration

## 5.1 Introduction

SSH provides secure communications between clients and servers. As a legacy, SSH is also an application providing protocol functionality in most Ubuntu systems. The protocol is supported on MAC OSX and Windows operating systems by other applications. Typically, SSH applications are employed to provide security for remote access to target servers from clients across potentially insecure networks. Types of remote access include: executing commands on a remote server; secure file transfers between client and a remote server; and providing a secure tunnel to a remote server [229]. SSH has been used as a medium for data exfiltration [5] so successful decryption can assist in identifying insider attacks, which are difficult to detect and increasingly common [68].

SSH encryption keys can be discovered by intercepting encryption function calls to extract parameters. For example, the Linux *ptrace* command can attach to the encrypting process enabling identification of keys and other artefacts [126]. This approach appears to have been used to discover SSH plaintext, ciphertext, and keys, although implementation details are unclear [230]. *Ptrace*, or the related *strace*, can also monitor server system calls to extract

SSH plaintext but this presumes control of the server and rapidity of tracing, both of which could be problematic in live scenarios. A similar technique obtained SSH client plaintext by extracting identified system call data from a modified SSH honeypot server [129]. These approaches are Linux-specific and, furthermore, may be detectable by virtual machine software. Consequently, they may be ineffective against malicious insiders, especially when the target device runs on Windows.

An SSH MemDecrypt extension implements a novel approach to find IVs in memory extracts that, in turn, enable rapid location of candidate keys, and the deciphering of live SSH traffic with high certainty. Protocol background, including SSH set-up and the message exchange process, is presented first, followed by details of the SSH framework extension design, construction, and evaluation. The final section discusses the implications of the investigation and observations on the adopted approach. Focus is restricted to AES encryption, with ChaCha20 encryption investigated in a later chapter.

## 5.2 SSH Protocol

MemDecrypt investigates the current SSH protocol, SSH-2. SSH-1 was defined in 1995 following development of the original SSH application. However, SSH-1 has security weaknesses and limited functionality. These factors led to the development of SSH-2, which uses different authentication and encryption algorithms, and defines additional functions such as secure file transfer [229]. SSH-2 is the most recent version, and is investigated in this chapter. SSH-2 will be referred to as SSH.

SSH is specified in four Internet Engineering Task Force (IETF) Request for Comments (RFCs), which are listed below. Of these, the second, third, and fourth items provide the technical details that form the basis for the framework extension.

- SSH Protocol Architecture (SSH-ARCH) [231] describes the SSH componentry and SSH architecture along with policy recommendations.
- SSH Transport Layer Protocol (SSH-TRANS) [232] specifies the set-up phase of a secure connection between a client and a server.
- SSH Authentication Protocol (SSH-AUTH) [233] defines the authentication process for a client once the connection is established.
- SSH Connection Protocol (SSH-CONNECT) [234] defines protocols for an SSH sub-system. Examples of sub-systems are starting a remote shell or transferring a file.

Although commonly layered, the protocols are considered to be independent. After completion of the set-up phase, authentication and connection services are requested by the client. Figure 5-1 illustrates the set-up, or handshake, phase.

| Protocol | Length | Info |
|---|---|---|
| SSHv2 | 82 | Client: Protocol (SSH-2.0-PuTTY_Release_0.70) |
| TCP | 54 | 2222 → 5488 [ACK] Seq=2885440383 Ack=1610535188 Win=29248 Len=0 |
| SSHv2 | 98 | Server: Protocol (SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.10) |
| SSHv2 | 1158 | Client: Key Exchange Init |
| SSHv2 | 430 | Server: Key Exchange Init |
| TCP | 54 | 5488 → 2222 [ACK] Seq=1610536292 Ack=2885440803 Win=16337920 Len=0 |
| SSHv2 | 134 | Client: Elliptic Curve Diffie-Hellman Key Exchange Init |
| SSHv2 | 294 | Server: Elliptic Curve Diffie-Hellman Key Exchange Reply, New Keys |
| TCP | 54 | 5488 → 2222 [ACK] Seq=1610536372 Ack=2885441043 Win=16276480 Len=0 |
| SSHv2 | 70 | Client: New Keys |
| TCP | 54 | 2222 → 5488 [ACK] Seq=2885441043 Ack=1610536388 Win=31424 Len=0 |
| SSHv2 | 166 | Client: Encrypted packet (len=112) |

Figure 5-1: SSH Handshake

### 5.2.1 Set-up Phase

The initial connection, packet protocol, server authentication, and basic encryption and integrity services are defined in the set-up

phase [229]. During this process, clients and servers exchange four messages: Protocol Version, Key Exchange Initialisation, Key Exchange, and New Keys. The New Keys message indicates set-up is complete for the sender.

In the initial exchange, SSH version compatibility is determined. The client initiates the conversation with the Protocol Version message identifying the supported SSH version and, optionally, the application and version implementing the connection, and the server responds in kind. The message contents are unencrypted ASCII strings in the format 'SSH-protocolversion-softwareversion comments' enabling the application or library, and probable operating systems to be inferred. For instance, *SSH-2.0-PuTTY_Release_0.70* indicates use of the PuTTY SSH client program [235], suggesting a probable Windows client. *SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.8* advises the receiver of the sender's operating system, Ubuntu, and library, OpenSSH, used in many applications and operating systems such as Ubuntu's *sshd* [97].

In the second message exchange, the client and server agree on algorithms for the session. The client transmits a Key Exchange Initialisation message, in which the contents are a set of unencrypted ASCII strings advising the server of supported algorithms for various purposes. For each purpose, the algorithms are listed in decreasing order of preference. Figure 5-2 illustrates the algorithm lists in a client Key Exchange Initialisation message.

The server Key Exchange Initialisation message identifies its algorithmic preferences for each purpose. SSH-TRANS stipulates that the agreed algorithm for each purpose is the most preferred client algorithm that is also supported by the server. For instance, if the client sends its encryption_algorithms_client_to_server list as a truncated string such as *aes256-ctr,aes256-cbc,rijndael-cbc@lysator.liu.se ...* and the server response is *aes128-ctr,aes192-ctr,aes256-ctr,arcfour256 ...*, then *aes256-ctr* becomes the agreed

```
Key Exchange
  Message Code: Key Exchange Init (20)
v Algorithms
    Cookie: 81696a2b0abd0cf9b7229ac406ee1245
    kex_algorithms length: 240
    kex_algorithms string [truncated]: curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ec
    server_host_key_algorithms length: 87
    server_host_key_algorithms string: ssh-ed25519,ecdsa-sha2-nistp256,ecdsa-sha2-nistp38
    encryption_algorithms_client_to_server length: 189
    encryption_algorithms_client_to_server string: aes256-ctr,aes256-cbc,rijndael-cbc@lys
    encryption_algorithms_server_to_client length: 189
    encryption_algorithms_server_to_client string: aes256-ctr,aes256-cbc,rijndael-cbc@lys
    mac_algorithms_client_to_server length: 155
    mac_algorithms_client_to_server string: hmac-sha2-256,hmac-sha1,hmac-sha1-96,hmac-md5
    mac_algorithms_server_to_client length: 155
    mac_algorithms_server_to_client string: hmac-sha2-256,hmac-sha1,hmac-sha1-96,hmac-md5
```

Figure 5-2: SSH Client Key Exchange

client to server encryption algorithm. Key Exchange algorithms of interest for constructing the MemDecrypt SSH extension are listed in Table 5-1. Whereas knowledge of the agreed symmetric algorithms is core to MemDecrypt, the hash algorithms are only used in the determination of encrypted message structures.

Table 5-1: Applied SSH Algorithm Types

| **Field** | **Description** |
|---|---|
| encryption_algorithms_ client_to_server | Symmetric algorithms for encrypting secure messages sent by the client |
| encryption_algorithms_ server_to_client | Symmetric algorithms for decrypting secure message received by the client |
| mac_algorithms_ client_to_server | Hash algorithms to authenticate messages sent by the client |
| mac_algorithms_ server_to_client | Hash algorithms to authenticate messages received by the client |

In the third message exchange, client and server send information required to generate the session cryptographic artefacts in Key Exchange messages. Asymmetric protocols such as Diffie-Hellman provide security for the exchange. The artefacts for the agreed algorithms are generated using the transmitted and received information. For example, if AES-CTR with 256-bit keys

using SHA2-256 hash was agreed, then encryption keys, IVs and authentication keys of the required size are generated.

In the fourth message exchange, the 'New Keys' message is transmitted to signify set-up completion for the sending party. All subsequent network messages in the session are encrypted using the generated symmetric encryption artefacts until another 'New Keys' message is transmitted or the session terminates. Authentication, connection, and sub-system exchanges are therefore encrypted.

### 5.2.2 Authentication Phase

Authentication with the server is commonly required before the client can perform activities on the remote server. The client initiates the authentication process by requesting support for the 'user-auth' service. Following a successful server response, the authentication method can be determined.

SSH-AUTH authentication methods are: 'public key', 'password', 'host based', and 'none'. Although the 'none' method is not recommended for actual authentication, the client can send it to discover supported server authentication methods. The only required method is 'public key' authentication, in which the client typically sends an encrypted message with a signature generated from a private key controlled by the client user. The server verifies the key using previously exchanged key data. For 'password' authentication, the encrypted message includes the user name and the password. In 'host based' authentication, the client device rather than the user is authenticated. For this method, a client sends an encrypted message with a signature generated by the client device private key for authentication.

For all methods, the server responds with a success or failure message. Success enables the session to progress to the connection stage.

### 5.2.3 Connection Phase

A client requires connection with a server for remote access. Multiple concurrent SSH sessions may be needed for client-server pairs as, for example, when a client user logs on for a shell session on the remote server and contemporaneously downloads files from the server. After successful authentication, a separate channel is established to support each SSH service. Client message types for the maintenance of SSH channels are 'Channel Open', 'Channel Request', 'Data Transfer', and 'Channel Close'. A Channel Open message specifies the channel identifier type of the channel required, which could be a session, x11 connection, or a direct or port-forwarded tunnel. This investigation focuses on session type channels.

Opening a session type channel advises the server that a client will request a server application. For a session type channel, the activity type to be executed on a server is identified with a Channel Request message. Common channel activity types request the creation of a command shell, or the enabling of an application or sub-system.

For the PuTTY client implementation, one Channel Request message identifies the server application. A string such as *simple@putty.projects.tartarus.org* advises the server that a single channel is required. A second Channel Request message identifies the sub-system. Sub-systems are applications configured to be supported by the SSH server. A common SSH server application sub-system is secure file transfer, which is described in the next sub-section.

The Data Transfer message type payload encapsulates the channel data. For the secure file transfer sub-system, all protocol messages are Data Transfer message payloads. The Channel Close type advises that no further messages will be sent for that channel.

### 5.2.4 Secure File Transfer

The secure file transfer sub-system has significant potential for malicious use by insiders stealing confidential files [5] making it worthy of investigation. Although the SSH file transfer protocol can be layered on other protocols, SSH is the norm. The SSH file transfer protocol (SFTP) is specified in IETF Draft 13 [236]. SFTP functions enable a number of file-related operations to be performed on the remote server such as writing, retrieving, deleting, renaming, setting and retrieving attributes, locking files, and reading directory contents. Of these, the writing of confidential files to a remote server is the activity most commonly associated with insider attackers using SSH and SFTP.

Common client SFTP message types involved in file uploads to remote servers are shown in Table 5-2. Though SFTP message lengths and request identifiers (except for Initialisation and Version message types) are message payload fields and therefore encrypted, these are intentionally omitted from the table for legibility.

The client initiates and, generally, terminates file uploads. The initial exchange determines the SFTP version for the session. The client transmits an Initialisation message with its supported SFTP protocol version and the server responds with a Version message. The next message exchange establishes server capacity to support the upload. A client Retrieve Attributes message requests information on the file system object. Of the available request attributes, a key request is determining whether the file size can be accommodated. A File Attributes message provides the server response. Following a client Open File message that contains the full path and file name for writing, the server responds with a Handle message. The client uses the handle to transmit source file contents in Write File messages, to which the server responds with Status messages. Once the upload has completed, a client

Close message advises that the server handle can be released and the server responds with its final Status message. A complete SSH message flow for connection, authentication, and connection for secure file upload is illustrated in Figure 5-3.

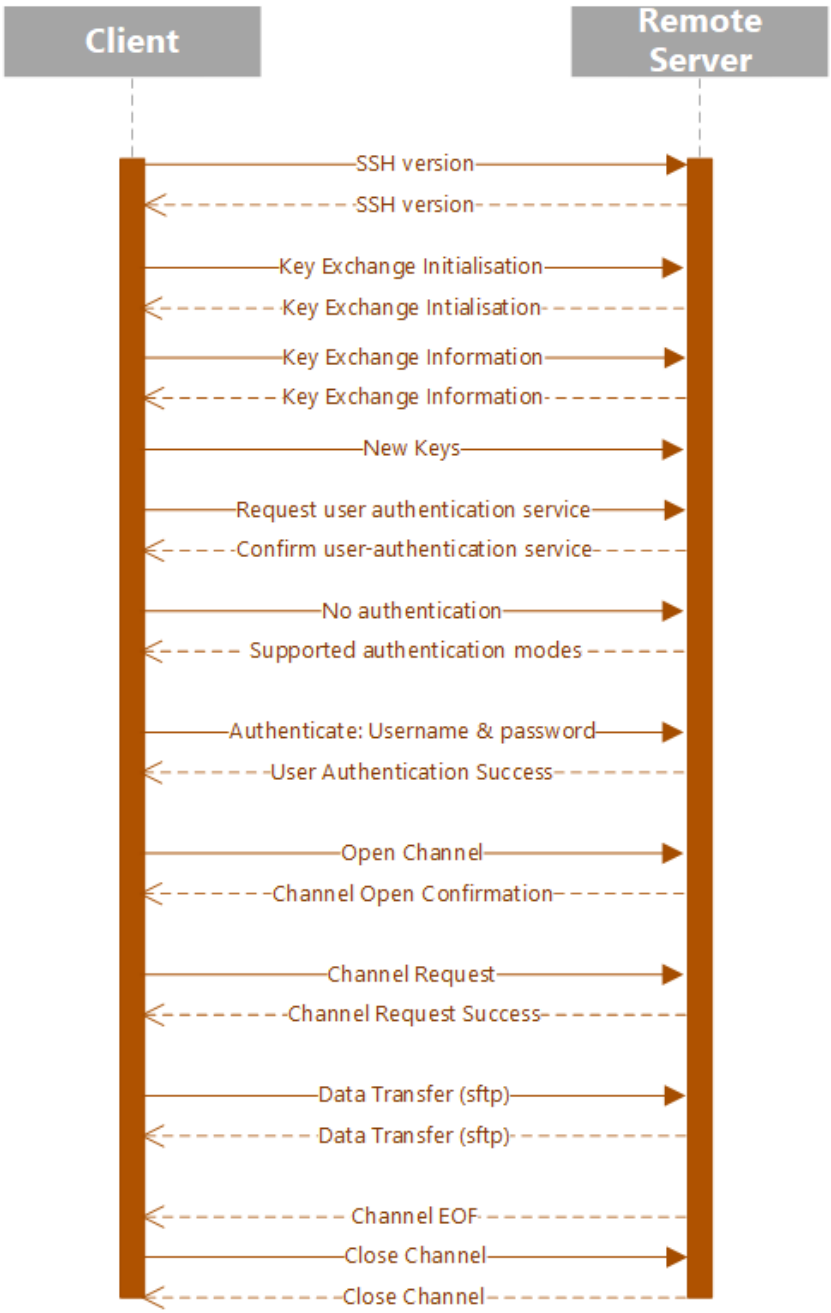



Figure 5-3: SSH Message Flow

Table 5-2: SFTP Write File Message Types

| **Code** | **Message Type** | **Purpose** | **Sample Payload** |
|---|---|---|---|
| SSH_FXP_STAT | Retrieve Attribute | Confirm file attributes e.g. file size, access | 4096 (file size) |
| SSH_FXP_CLOSE | Close Handle | (handle string) | - |
| SSH_FXP_STAT | File Attributes | Request file or directory attributes | /home/user/ |
| SSH_FXP_HANDLE | Handle | Confirm file open with handle identifier | (handle string) |
| SSH_FXP_INIT | Initialisation | Protocol version definition | 3 or 6 |
| SSH_FXP_OPEN | Open File | Request handle for folder & filename | /home/user/ creditcard.csv |
| SSH_FXP_STATUS | Status | Client operation outcome | 0-31 & associated message |
| SSH_FXP_VERSION | Version | Protocol version definition | 3 or 6 |
| SSH_FXP_WRITE | Write File | File contents | name, number, expiry, cvv |

## 5.3 SSH Extension Design

Each MemDecrypt component incorporates an SSH extension. Protocol features and associated packet formats are primary determinants. However, AES modes and virtual machine operating systems also require separate logic. Customisation improves performance further and provides opportunities for accurate validation of decrypts. The following sub-sections present extension details for each component and, where applicable, for modes of

operation and operating systems.

### 5.3.1 Data Collection

This component obtains virtual machine data including network packets and virtual machine process memory for analysis. SSH network packets are inspected to discover agreed algorithms and determine the timing for process identification and process memory extraction. Encryption and hash algorithms are derived by matching preferred client algorithms with server algorithms from the SSH Key Exchange messages.

Completion of the SSH session's TCP handshake phase presents a useful opportunity to identify the encrypting process. On Windows clients, the initiating SSH process is generally discovered by inspecting lists of processes with open network ports for a match with the network packet addresses. A second, weaker, test is by identifying new client processes. On Ubuntu servers, the SSH *sshd* service connects to an SSH port awaiting connection requests. On reception of such a request, the SSH service creates a child *sshd* process to service it. So, new server *sshd* processes are potential encrypting processes.

Memory extracts may contain cryptographic artefacts once New Keys messages are transmitted, as all subsequent SSH session messages have encrypted payloads. To discover candidate cryptographic artefacts, the read/write memory of each identified process is extracted for memory analysis when two different messages are transmitted. The messages are not required to be sequential.

With algorithms known, memory acquired, and network packets captured, memory analysis commences.

### 5.3.2 Memory Analysis

AES modes are treated differently in memory analysis. In AES-CBC, IVs are obtained from network packets so only candidate

keys are sought. Candidate keys are segments of process memory extracts taken at different times, that are static across the extracts, and with Shannon entropies exceeding the threshold.

In AES-CTR, candidate IVs and keys are sought. Candidate IV locations are discovered first with approaches that encompass an analysis of memory extracts, network packets, or both network packets and memory extracts. If program memory extracts are taken when the same activity is being performed, such as the transmission of outgoing messages, memory blocks containing IVs change, while other blocks remain static.

AES-CTR IVs increment by 1 for each encrypted or decrypted block so the IV delta between blocks in two outgoing SSH network packets is equal to the number of encrypted blocks transmitted between them. Algorithmically, suppose the value at location $p$ in extract file $y$ at time $a$ is compared with the value at location $p$ in extract file $y$ at time $b$. Then, for the values to be IVs, represented by $IV_{pya}$ and $IV_{pyb}$ respectively, equation 5-1 must be valid:

$$IV_{pyb} = IV_{pya} + n \tag{5-1}$$

where $n$ is the number of AES encrypted network blocks transmitted in the session between times $a$ and $b$. IV field locations in program data structures, and therefore in memory, may not change during a session. So, when a virtual machine process performs an action, such as transmitting a packet to the same destination address, values at the same memory address in different extracts can be usefully compared. Memory segments with values that change between extracts by the number of blocks in the payload of the intervening packets are candidate IVs.

As with AES-CBC, AES-CTR candidate encryption keys are segments of process memory extracts taken at different times, that are static and have entropies exceeding the threshold. Because keys and IVs are cryptographic artefacts and probably in common data structures, candidate key discovery commences from candi-

date IV locations. Candidate cryptographic artefacts are used in decrypt analysis.

### 5.3.3 Decrypt Analysis

The component iterates through candidate IVs and keys until a valid combination is found or all combinations exhausted. In AES-CBC mode, the IV for decrypting the first network packet block is contained in the network packet, and for subsequent blocks is the preceding ciphertext so its candidate IV set size is 1. The same validation approach is used for both AES-CTR and AES-CBC modes.

In SSH, the valid combination can be determined using encrypted data block fields. The encrypted segment in SSH data packets has the format shown in Table 5-3. As specified by SSH-TRANS, the sum of the payload and padding field sizes must be a multiple of 16 with a minimum padding of four bytes.

| Packet Length (4 bytes) | Padding Length (1 byte) | Payload (variable bytes) | Padding (variable bytes) | MAC (16/32 bytes) |
|---|---|---|---|---|

Table 5-3: SSH Encrypted Payload Format

The packet length is the sum of the padding length size, the payload, and padding fields. Equation 5-2 may be a good decrypt test for SSH messages as $2^{(8\times4)} - 21$ valid packet length decrypts are possible. The minimum SSH block size is 21 bytes comprising a packet length of four bytes, a padding length of one byte, and the payload and padding, which are at least one block. So, the probability of an incorrect decrypt producing the correct header data is 1 in 4,294,967,275 for SSH packets not extending beyond one network packet. Equation 5-2 is sound during the authentication, channel, and sub-service setup stages when SSH packet sizes are generally small. Reassembly is undertaken for SSH packet sizes

exceeding the network packet size. A modified version of Equation 5-2 is applied to reassembled SSH packets as the reassembled packet size must be equal to the encrypted SSH packet size less the packet length field size and authentication code size.

$$\begin{aligned} packet\ data\ length\ = \\ & decrypted\ packet\ length\ + \\ & size(packet\ length\ field)\ + \\ & size(MACfield) \end{aligned} \tag{5-2}$$

An additional decrypt test evaluates the padding length field. As specified in SSH-TRANS, a correct decrypt must comply with Equation 5-3:

$$4 <= padding\ length <= 255 \tag{5-3}$$

When both equations are correct, the desired cryptographic artefacts have been identified. The component uses these artefacts to decrypt the session. The decrypts are parsed to determine SSH authentication and connection fields, as well as all SFTP fields including file details and content.

## 5.4 SSH Extension Implementation

Implementation details for SSH extensions are presented in the following sub-sections. For each component, Windows client and Ubuntu server details are provided where differences exist. AES mode implementations also differ and are described.

### 5.4.1 Data Collection

Cryptographic algorithms are obtained from the Key Exchange Initialisation messages. The bespoke software for finding client preferences in server algorithm lists is identical for clients and

servers. This information is available in the retained network packets so could be obtained during the analysis phases.

Operating system process identification implementations differ. For Windows clients, the Volatility *netscantbl* plugin retrieves details of processes with open network ports and matches SSH client packet and destination addresses with plugin source and target address fields. For Ubuntu servers, process lists and their associated details are generated by calling a bespoke routine to detect recently created *sshd* processes. Candidate process identifiers and associated memory structure pointers are retained for later memory extraction.

Operating system memory extractions also adopt different approaches. A specific SSH extension is not required for extraction so, for Windows, the *vadanalyse* Volatility plugin identifies VADs with read/write protection attributes and writes the memory to a time-stamped file. For Ubuntu server virtual machines, the heap memory start and end locations are identified from the memory structure for each new *sshd* process, the memory read, and then written to a time-stamped file.

### 5.4.2 Memory Analysis

Memory analysis approaches differ between the AES modes. For AES-CTR mode, bespoke software first discovers candidate IVs. IVs increment between separate memory extractions, so identical files over different extracts cannot contain IVs, when IV memory locations are fixed over different extracts. For Windows analysis, files in the first memory extraction folder are compared with corresponding files in the second folder. If the memory files differ, a deeper analysis identifies candidate IVs.

Candidate IVs are discovered in these Windows extract files and Linux heaps by evaluating Equation 5-1. For example, if a 16-byte memory block value is 123,456 and two network packets of

10 and 5 encrypted blocks are transmitted before the next extract, then a memory block value of 123,471 at the same address in the second extract identifies a candidate AES-CTR IV. Algorithm 5.1 shows the AES-CTR IV location process.

**Algorithm 5.1:** SSH AES-CTR IV Memory Analysis

```
Data: extract folders fldr_a,fldr_b and packets pkt_a, pkt_b
Result: Z = candidate IVs
delta := blocks[pkt_a:pkt_b];
for file f_1 in fldr_a do
    f_2 = match (f_1, fldr_b);
    if f_1 <> f_2 then
        for i = 0 to size(f_1) inc 4 do
            if val(f_2[i:i+16]) - val(f_1[i:i+16]) = delta then
                Z += f_1[i:i+16];
            end
        end
    end
end
```

The increment of four assumes 16-byte IVs are word-aligned in memory. Lower increments would result in longer analysis times. The MemDecrypt extension accommodates big-endian and little-endian memory storage schemes to allow for different SSH implementations.

Key discovery is bespoke software. For AES-CTR, candidate IV locations assist in the discovery of candidate encryption keys. Extract files in the first folder with candidate IVs are inspected for candidate keys. The component iteratively evaluates entropies of memory segments and identifies candidate keys where the Shannon entropy threshold is exceeded. Keys are assumed to be word-aligned so iterations increment and decrement by four starting from candidate IV locations. Key length determines segment size and entropy threshold. Segments with entropies exceeding the threshold are compared with the segments at the same extract

file location in the second folder. Equality identifies the segment as a candidate key.

For AES-CBC mode, candidate keys are identified applying the same process. However, there is no preferential ordering of memory extract files. The identified candidate keys, and IVs for AES-CTR, provide input to the decrypt analysis component.

### 5.4.3 Decrypt Analysis

The component iterates through the sets of candidate keys and IVs until decrypts are validated for the first ciphertext block. Software is bespoke aside from using PyCrypto [223] to generate the decrypt. For performance, session network packets and cryptographic artefacts are held in MemDecrypt extension memory data structures. Decrypt validation evaluates Equation 5-2, or its modified version, to exclude decrypts with implausible payload lengths. Decrypted padding length compliance with Equation 5-3 provides additional validation. A valid decrypt identifies the key and IV combination enabling SSH plaintext to be uncovered.

Each encrypted block is deconstructed as specified in SSH-AUTH, SSH-CONN, and SFTP RFCs. For authentication, the 'password' authorisation method is implemented for client and server messages. For connection requests, complete Channel Open, Channel Request, Data Transfer, and Channel Close messages are decrypted for client and server messages. For the SFTP subsystem, all fields for client and server SFTP message types are decrypted. Decrypted messages, including file contents, are written to disk for off-line viewing.

## 5.5 Evaluation

The SSH MemDecrypt extension was evaluated by carrying out a series of experiments with variable file sizes, key lengths, AES

modes, operating systems, and operating system versions. The SSH experimental set-up is described, followed by the experimental results for Windows clients and Ubuntu servers. The testbed is comprised of the base MemDecrypt physical environment with additional software to support SSH communications.

### 5.5.1 Experimental Set-up

Windows supports a number of SSH clients, including the PuTTY suite [235]. PuTTY is widely used [237], so may be used by suspect actors. It is anticipated that other Windows SSH client applications may generate similar results. Two secure copy programs, *pscp* and *sftp*, are included in the PuTTY suite but, in modern implementations, these programs execute the same process to transfer files [229]. For testing, *pscp* was run from the Windows command line using commands of the form:

*pscp -P nnnn filename name@ipaddress:/home/name*

where *nnnn* is the target port, *filename* is the file being transmitted, *name* is a user account on the target Ubuntu server, *ipaddress* is the target server IP address and */home/name* is the Ubuntu server target folder for the transmitted file.

SSH server functionality is provided by *openssh-server*. An Ubuntu service is started from the bash command line to receive SSH client requests with commands of the form:

*/usr/sbin/sshd -f /root/sshd_ config -d -p nnnn*

where *nnnn* is the service receiving port number and *sshd_ config* contains configuration details such as server supported encryption algorithms.

Sets of experiments investigated decrypting SSH traffic en-

crypted with AES under different conditions. One set evaluated decrypt effectiveness for Windows 7 and Windows 10 clients. To evaluate file invariability, a second set uploaded 30 files in text, Adobe Portable Document Format (pdf), Microsoft Office Excel, and Microsoft Windows executable formats from Windows 10 clients in AES-CTR mode with 256-bit keys. A third set evaluated the effectiveness of 128-bit, 192-bit and 256-bit keys on Windows 10 clients in AES-CTR mode. A fourth set evaluated MemDecrypt effectiveness with 256-bit keys in AES-CBC and AES-CTR modes on Windows 10 clients. A final set assessed decrypt effectiveness for Ubuntu server memory extracts in AES-CBC and AES-CTR modes with 256-bit keys.

### 5.5.2 Experimental Results

In each experiment, the encryption keys, and for AES-CTR, the IVs, were discovered and valid plaintext produced for all SSH and SFTP fields. For example, the decrypted fields of interest from the command *'pscp -P 2222 plaintext.txt peter@192.168.137.85:/home/peter'* and plaintext.txt of *'An outcropping of limestone beside the path that had a silhouette...'*, are illustrated in Figure 5-4. Other decrypted fields such as request identifiers and file handles are omitted. As observed earlier, the probability of an incorrect combination generating a packet length complying with Equation 5-2 is 0.00000002%. As a result, no false positives were generated in the experiments.

Analysis durations for producing SSH plaintext determines the framework's utility. For example, if plaintext is produced during the network session, MemDecrypt can assist in detection or preventative measures. Component durations were obtained in each experiment to assess framework performance and identify anomalies.

The first experiment considered component durations for Win-

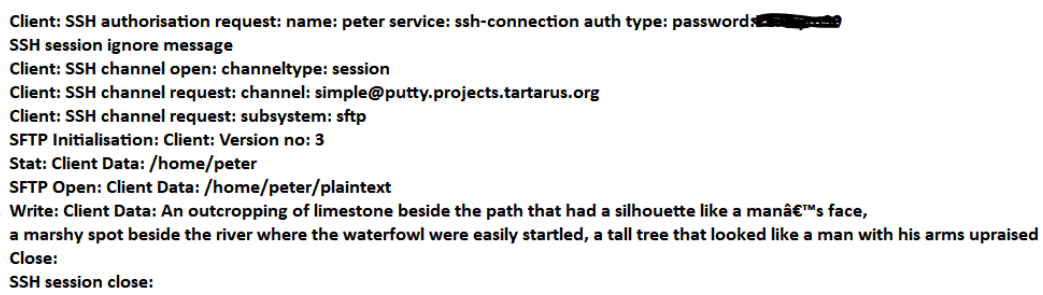



Figure 5-4: SSH Decrypt Output

dows 7 and Windows 10 clients. Fixed variables were AES-CTR encryption mode, 256-bit key length, and the uploaded text file. The results are summarised in Table 5-4. Windows 7 extracts were typically 4.1 MB, and Windows 10 6.9 MB. The size differences result from VAD structure changes between operating system versions, so Windows 7 memory analysis completed faster than Windows 10. Maximum and minimum sizes for IV candidate sets were 18 and 4, and for key candidate sets 525 and 267. Although combinatorial quantities are significant, the primary cause of differences in operating system version decrypt analysis durations was the IV candidate ordering, where correct Windows 7 IVs generally occurred later in the candidate set than Windows 10 IVs.

Table 5-4: Windows 7 vs Windows 10 Durations (secs)

| | | **Extract** | **Memory** | **Decrypt** |
|---|---|---|---|---|
| Windows 7 | Mean | 2.0 | 10.0 | 3.5 |
| | Std Dev | 0.1 | 1.4 | 0.5 |
| Windows 10 | Mean | 4.9 | 15.9 | 1.6 |
| | Std Dev | 0.5 | 1.4 | 1.5 |

A second experiment compared analysis time durations for different file sizes on Windows 10 clients using AES-CTR with 256-bit keys. File sizes ranged between 5 bytes and 660 KB and included text files, Excel spreadsheets, and Adobe pdf files. The results are presented in Table 5-5. Memory analysis durations

were invariant with file size. In decrypting different sized files with a single key and IV combination, larger files generally took longer, but durations were not determined solely by file size. So, a 660 KB file could be decrypted in 2.0 seconds whereas a 230 KB file required 3.9 seconds. Here, differentiators were the candidate IV set size and the ordering of IVs within the set.

Table 5-5: AES-CTR Upload File Size Analysis Durations (secs)

| | **Memory Analysis** | **Decrypt Analysis** |
|---|---|---|
| Maximum | 25.2 | 3.9 |
| Minimum | 12.3 | 0.1 |
| Mean | 16.7 | 1.4 |
| Standard Deviation | 3.0 | 1.2 |

The third experiment considered analysis durations for different AES-CTR key lengths on Windows 10 clients. Shorter key lengths required lower entropy thresholds, so more candidate encryption keys were discovered in memory analysis. For example, one test sequence with identical file sizes yielded 272 candidate keys with a 256-bit key length, 1123 candidate keys with a 192-bit key length, and 5,658 candidate keys with a 128-bit key length. These differences are reflected in decrypt analysis durations as illustrated in Figure 5-5.

The fourth experiment compared analysis time durations on Windows 10 clients for 256-bit key lengths in AES-CTR and AES-CBC modes. Memory analysis was generally more rapid in AES-CTR than AES-CBC although extract sizes were similar. The decrypt analysis component does not iterate through potential IVs for AES-CBC resulting in marginally shorter durations than for AES-CTR as shown in Figure 5-6.

A fifth experiment investigated the extension's ability to decrypt SSH traffic when analysing Ubuntu server memory. In tests with AES-CTR encryption, 256-bit key lengths, and file sizes between 200 bytes and 20 KB, client and server sessions were de-

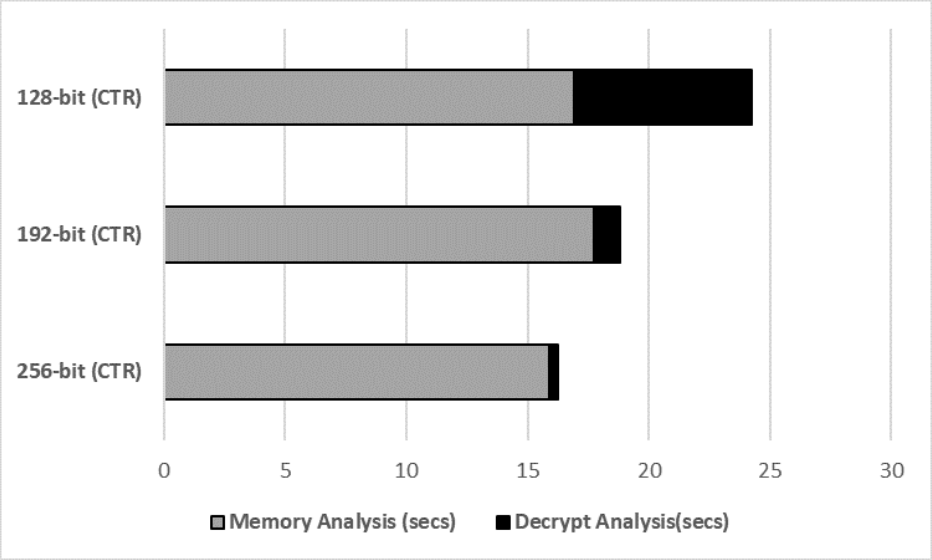



Figure 5-5: SSH Analysis Durations for Variable Key Lengths

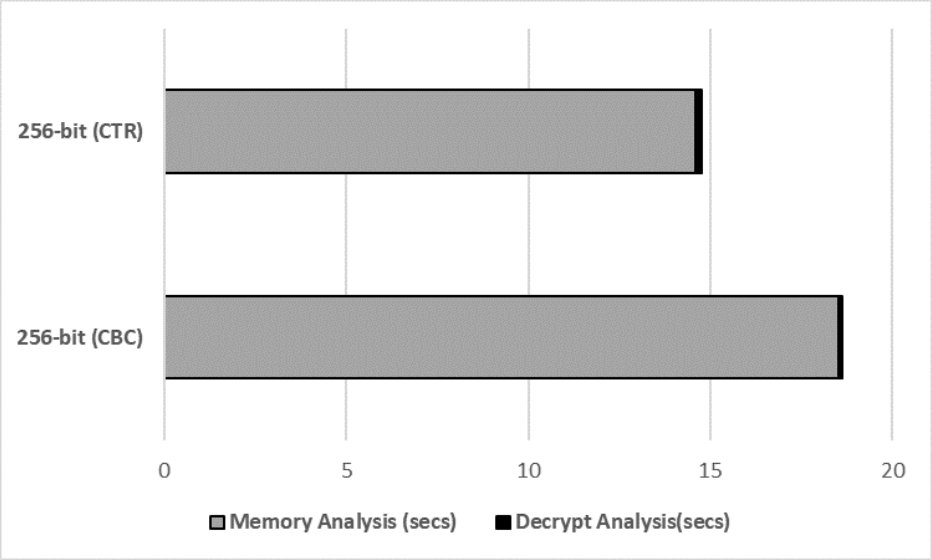



Figure 5-6: SSH Analysis Durations for Variable Modes

crypted. However, the approach was slightly different from Microsoft Windows. In particular, candidate IV and keys were discovered in extracts associated with messages later in the SSH session when compared with Windows. A possible reason for the discrepancy is that server process extracts were triggered on frame-

work interception of SSH *client* messages. Artefacts might be discovered in earlier extracts with SSH *server* message triggers. Figure 5-7 at the end of this chapter illustrates an example of a complete SSH session, including a file upload.

Also in the fifth experiment, the data collection component obtained Ubuntu server process lists and extracted process heaps in 0.3 seconds. The memory analysis component consistently discovered 2 candidate IVs and between 290 and 330 candidate keys. As encryption and decryption IVs should be independent, both candidates could be assumed to be valid IV instances. When combined, Ubuntu server analysis durations, which are shown in Table 5-6, are faster than the Windows client equivalents.

Table 5-6: AES-CTR Ubuntu Server Analysis Durations (secs)

| | **Memory Analysis** | **Decrypt Analysis** |
|---|---|---|
| Maximum | 2.0 | 1.9 |
| Minimum | 1.3 | 1.2 |
| Mean | 1.6 | 1.5 |
| Standard Deviation | 0.3 | 0.3 |

### 5.5.3 Analysis

Component duration analysis suggests opportunities for performance enhancement. Ubuntu memory extraction is more than an order of magnitude faster than Windows. MemDecrypt employs Volatility plugins to extract Windows process read/write memory. Volatility loads layered Python classes on each invocation to provide extensibility in supporting various operating systems and versions. A faster, but more complex, framework solution may be to invoke low-level PyVMI functions directly, as implemented for Ubuntu.

Operating systems memory management differences also impact memory extraction durations. While the *openssh-server* heap

is typically no greater than 168 KB, Windows read/write VADs number 16 files with a total size of 6.9 MB. In SSH experiments, Windows cryptographic artefacts were always discovered in one of three 1 MB VAD extracts, so extracts could possibly be limited by memory size. Deeper knowledge of undocumented VAD structure fields may also enable precise identification of which memory extracts contain cryptographic structures.

Memory and decrypt analysis durations can also be reduced. With PuTTY, distances between key and IV memory addresses were invariant for operating system version or transmitted file size, but not key length. For example, *pscp* distances are 968 bytes for 256-bit and 192-bit keys and 728 bytes for 128-bit keys. With this knowledge, memory and decrypt analysis durations reduced to one second.

As decrypt analysis for a single key-IV combination took about 0.0056 seconds and each decrypt can be checked independently, parallelising can reduce durations significantly. Implementing pre-testing, pipelining between components, multi-threading of components to use parallel processing, and translating the framework into low-level language offer opportunities for further performance enhancements.

## 5.6 Conclusions

Results show the MemDecrypt SSH extension identified small candidate sets of cryptographic artefacts in the memory of Windows clients and Ubuntu servers enabling SSH sessions to be decrypted with very high degrees of certainty. These results were independent of Windows client version, AES mode, key length, and uploaded file size. Performance sufficed for user-initiated sessions. Although the investigation was limited to specific authentication, SSH sub-system, encryption algorithm and modes, and client and server applications, these selections are not uncommon.

Assumptions were made regarding candidate encryption key and IV characteristics. Candidate encryption keys were assumed to have high Shannon entropy, static throughout a network session, and at the same memory location for multiple extracts. Lower entropy implies less randomness, which is considered to be an unlikely outcome. Two extractions were required for AES-CTR memory analysis and one for AES-CBC. As extractions can be sequential, non-static keys might require a new key exchange between encrypted messages, which is impractical. An experiment manually relocating keys within Linux heap extracts delayed detection by less than 0.5 seconds. The key characteristic assumptions appear valid.

Candidate AES-CTR IVs were assumed to be present at the same address for multiple extracts and to increment by 1 for each encrypted plaintext block. As with keys, relocation induced slight memory analysis delay. Obfuscating IV values in memory may delay framework decryption and so could be implemented in secure communications applications. Possible obfuscatory measures include IV encryption or splitting. The former requires doubled decryption with significant performance consequences, so IV splitting may be more promising. For example, randomised splits with variable interposed data offers potentially faster IV reassembly. This technique can also be applied to keys making MemDecrypt artefact discovery more challenging.

Completely decrypted SSH sessions offer opportunities to counter insider attacks, such as the exfiltration of data to external locations. Another potential application is, with candidate artefact retention, decryption can be delayed until required to avoid contravening privacy laws. Decryption also introduces the risk that malicious actors may obtain access to remote servers as well as legitimately uploaded confidential data.

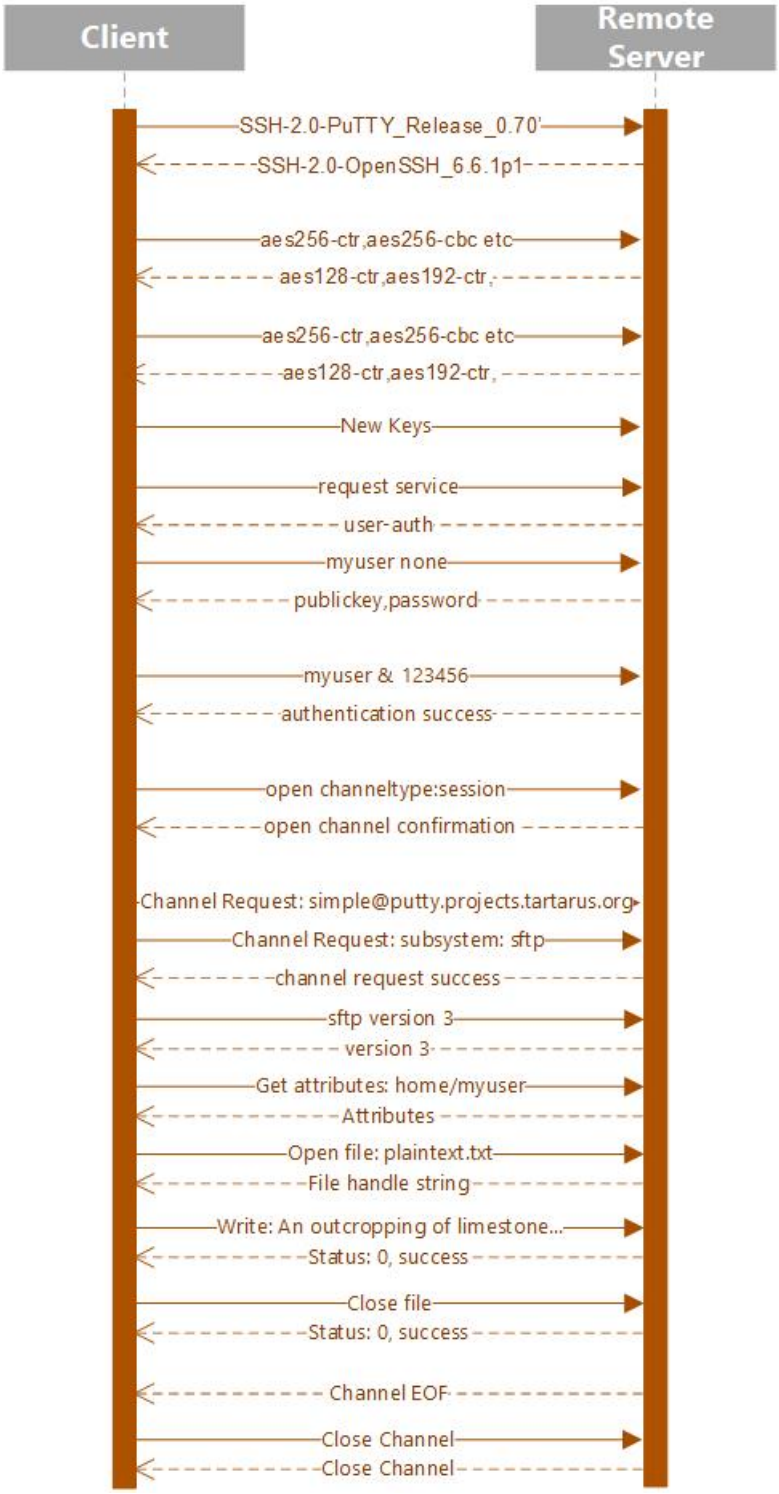



Figure 5-7: SSH Decrypted Session

# Chapter 6

# Decrypting Web Traffic

## 6.1 Introduction

TLS is a key protocol for Internet client-server communications [2] [3]. In this chapter a framework extension is constructed to decrypt TLS communications. Focus is restricted to AES-encrypted TLS communications as a later chapter will investigate ChaCha20-encrypted communications.

Analysts can discover plaintext or encryption keys used in TLS communications. For instance, plaintext can be obtained by interposing a man-in-the-middle proxy, such as SSLsplit [238], which transforms a single TLS session into two such sessions. Encryption keys have also been found for the AES-CBC encryption mode in Linux memory extracted upon detection of a specific client message available in the TLS 1.2 protocol [128]. However, these scenarios are restrictive.

The TLS MemDecrypt extension implements novel approaches to locate candidate IVs in memory extracts, thereby supporting the discovery of candidate keys. These artefact sets enable the rapid decryption of TLS 1.2 and 1.3 payloads. To provide context, the first section presents protocol background, including set-up and message exchange process details. The following sections discuss the TLS framework extension and its implementation, con-

duct experiments, and discuss experimental results.

## 6.2 Background

TLS versions are discussed first, followed by protocol version details where relevant. In particular, IV structures differ between TLS versions and, for TLS 1.2, between modes, and these differences were important for developing the framework extension. Unlike other variations, TLS 1.2 AES-GCM IVs may be found in generated key blocks and this mode is described first, followed by details of TLS 1.2 AES-CBC and TLS 1.3 IV constructs.

### 6.2.1 Protocol Versions

Various TLS versions are still in use. TLS 1.0 was specified as the successor to SSL by the Internet Engineering Task Force (IETF) in 1999, after which versions 1.1, 1.2 and 1.3 addressed perceived shortcomings of previous versions. Protocol goals are: to provide a secure end-to-end channel between two parties through server authentication; confidentiality of the exchanged information once the channel is secured; and integrity, meaning data cannot be manipulated without detection [239].

TLS 1.2 was published by IETF RFC 5246 in 2008 [240]. For improved performance and security, TLS 1.3 was specified, ratified [241], and issued as a new RFC [239] in 2018. Although individual browsers are reported to be TLS 1.3-ready, a large hosting company survey over a 24-hour period in December 2017, obtained the TLS version usages shown in Table 6-1 [242]. The survey was conducted prior to TLS 1.3 ratification so, though indicative, usage may have changed. A signficant challenge is believed to be the upgrading of intermediary and target devices to support the newer version [242].

Commonly used tools and libraries were still under development for TLS 1.3 when primary thesis experiments were conducted so TLS 1.2 was the focus of investigations. Nevertheless, the framework is constructed to decrypt TLS 1.2 and 1.3 traffic.

Table 6-1: TLS Version Usage Statistics

| **Version** | **Usage** |
|---|---|
| TLS 1.0 | 11.36% |
| TLS 1.1 | 00.38% |
| TLS 1.2 | 88.2% |
| TLS 1.3 | 00.06% |

TLS 1.3 introduces changes which impact framework design. One change affects AES modes. For enhanced TLS performance, message confidentiality and integrity can be achieved in the same transaction by extending an encryption mode, commonly known as authenticated encryption. For instance, when the AES-CTR encryption mode is followed with a Galois field hash [243], the mode is called Galois Counter (GCM). GCM generally offers better performance than other TLS modes [85] [243].

Though encryption followed by authentication is considered secure, authentication followed by encryption, as with TLS 1.2 AES-CBC, creates vulnerabilities. Padding attacks, such as POODLE and BEAST, and timing attacks, such as Lucky13, demonstrated that AES-CBC encrypted traffic could be decrypted with sufficient attempts [66]. As a result, while TLS 1.3 supports AES in GCM mode, AES with CBC is deprecated. As MemDecrypt investigates TLS 1.2, both AES-GCM and AES-CBC modes are analysed. Other TLS 1.3 changes that impact the framework are covered in later sections.

### 6.2.2 Handshake, Change Cipher Specification, Application Data

TLS 1.2 comprises a record protocol, which provides encryption and authentication, and four protocols layered on the record protocol: handshake, change cipher specification, application data, and alert protocols [240]. All except the alert protocol, which is unconnected with this research, are analysed by the MemDecrypt extension. Each layered protocol is identified by a defined message content type. The relevant details of each analysed, layered protocol are described followed by germane elements of the record protocol.

The outcome of a completed TLS handshake is the production of cryptographic artefacts and algorithms for use in a secure session. The client and server exchange the messages shown in Table 6-2 to reach agreement on TLS version, session algorithms, extensions, to validate compatibility, and verify the opposite party's credentials where required. Unlike earlier versions, in TLS 1.3 the TLS version is negotiated through the extensions.

Client and server Hello messages define random values for key generation, and establish TLS version, new session cryptographic algorithms, message compression algorithms, and additional requested server functionality, the extensions. The client Hello message contains its algorithmic and extension preferences and the server Hello response identifies the first supported client preference. The agreed TLS version is the most recent supported by client and server. The cryptographic algorithm value identifies the session key exchange mechanism, the encryption algorithm, and the authentication algorithm. For example, the value `0xc014` specifies that key exchange applies the Elliptic Curve Diffie-Hellman Ephemeral method, server authentication requires RSA certificates, encryption applies AES encryption in CBC mode with a 256-bit key size, and authentication uses the Secure Hash

Table 6-2: TLS 1.2 Handshake Phase Messages

| **Message Type** | **Description** |
|---|---|
| Client Hello | Preferred list of ciphers, compression algorithms, and extensions |
| Server Hello | Agreed cipher, compression algorithm, and extensions |
| Server Certificate | Server Certificate for authentication |
| Server Key Exchange | Server parameters for master secret |
| Client Key Exchange | Client parameters for master secret |
| Server Hello Done | Server has sent all messages for the handshake |
| Server New Session Ticket | Tickets for resumed TLS sessions |
| Client Change Cipher Specification | Further client data messages are encrypted |
| Client Encrypted Handshake | Encrypted agreed handshake parameters |
| Server Change Cipher Specification | Further server data messages are encrypted |
| Server Encrypted Handshake | Encrypted agreed handshake parameters |

Algorithm 1 (SHA-1) algorithm, which is now considered insecure.

The server Certificate message allows for client verification of the server credentials. The message includes a sequence of certificates starting with the server's own. In Internet TLS communications, third-party generated certificates are commonly required by the client. The message is not of interest for the current framework.

The Key Exchange message contents depend on the agreed key exchange mechanism. The objective is for sufficient material to be exchanged for client and server to generate pre-master keys and then master keys enabling the generation of key blocks. For instance, if Diffie-Hellman is the agreed mechanism, Diffie-Hellman public values are included in the client Key Exchange message, and potentially in a server Key Exchange message when the key exchange mechanism variation requires it. These messages are not

of interest for the current framework.

A server Done message signifies no further server data is required for key block generation. It generally also transmits a New Session Ticket message so that TLS sessions can be resumed without a new handshake. These messages are also not of interest for the current framework.

The Change Cipher Specification message is not a handshake protocol message. However, its interposition between the prior handshake protocol messages and the handshake completion message (Encrypted Handshake) suggest that it is appropriately discussed at this juncture. The Change Cipher Specification message exchange advises that all further transmitted data messages in the session are encrypted. The next messages exchanged are Encrypted Handshake, also known as Finish, messages which include encrypted handshake parameters. The message following completion of the handshake phase is generally the first Application Data Message, which carries an encrypted payload. Change Cipher Specification is not a defined TLS 1.3 protocol message. The keys and IVs used in the encryption and decryption processes are defined by the record protocol.

### 6.2.3 Record Protocol

In TLS 1.2, the record protocol ingests material exchanged in the handshake to generate key blocks. In particular, the hash algorithm uses the client and server random values and the generated master secret to generate a key block comprised of randomised values. Key block field quantities and sizes vary in accordance with encryption and hash algorithms. Table 6-3 shows the key block structure for AES-GCM with 256-bit keys and a 256-bit SHA hash.

IV implicit segments are not used in all encryption modes. For instance, TLS IVs are constructed differently for AES-GCM and

Table 6-3: TLS 1.2 GCM Key Block Fields

| **Field** | **Size (bytes)** |
|---|---|
| Client Authentication key | 32 (SHA256) |
| Server Authentication key | 32 (SHA256) |
| Client Encryption Key | 32 (AES 256) |
| Server Encryption Key | 32 (AES 256) |
| Client IV implicit segment | 4 (AES-GCM) |
| Server IV implicit segment | 4 (AES-GCM) |

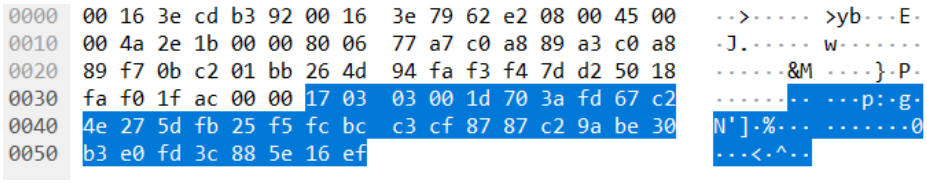
AES-CBC modes. In AES-GCM, a 12-byte IV is comprised of a 4-byte implicit and an 8-byte explicit segment [243] [244]. The implicit segment is the key block field value and the explicit segment is independently generated by the sender and included in each Application Data Message. The implicit field is a mutually agreed fixed value, so the explicit field is incremented for each packet to avoid identical plaintext producing identical ciphertext. The AES-GCM Application Data message format is shown in 6-4.

Table 6-4: AES-GCM Application Data Message Format

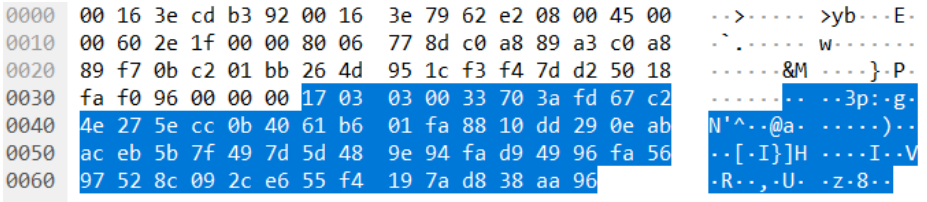
| Content Type | Version | Data Length | Explicit IV | Encrypted Data |
|---|---|---|---|---|

To illustrate, Figure 6-1 shows the payloads of two consecutive TLS 1.2 Application Data messages. In the highlighted sections, the content types `0x17` signify Application Data messages, versions `0x0303` that TLS 1.2 is the protocol, and the data lengths `0x001d` and `0x0033`, that is 29 and 51 decimal respectively, are the sizes of the encrypted data fields. The difference between the explicit IV segment values `0x703afd67c24e275d` and `0x703afd67c24e275e` is 1, the incremental value.

AES-CBC IVs are constructed differently. The 16-byte IV is generated by the sender and included as a field in the Application




(a) GCM Application Data Message




(b) Next GCM Application Data Message

Figure 6-1: TLS AES-GCM Application Data Messages

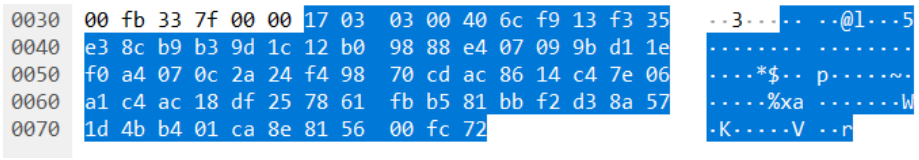
Data message as shown in Table 6-5.

Table 6-5: AES-CBC Application Data Message Format

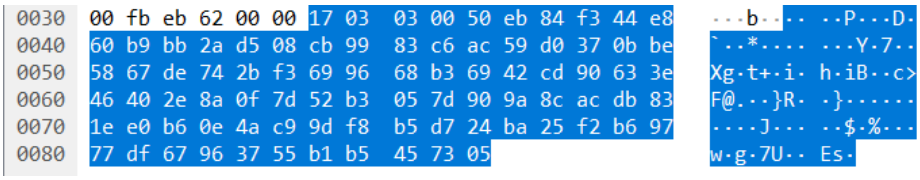
| Content Type | Version | Data Length | CBC IV | Encrypted Data |
|---|---|---|---|---|

The IV for each following block in a packet is the previous ciphertext block. Figure 6-2 illustrates two consecutive AES-CBC client Application Data messages. In the highlighted sections, the 16-byte segments from offset `0x3b` to `0x4a` are the randomly generated IVs for the first block and differ significantly. The second block's IV is the ciphertext of the first block. For example, in 6-2b, the IV for the second block is the 16-byte segment starting at offset `0x4b` with `0x59`.

TLS 1.3 AES-GCM Application Data messages do not include IV segments. The 12-byte initial IV, generated during the handshake process, is XORed with a counter to obtain the IV to be used in the AES-GCM encryption process. The counter value is initially set to zero and then incremented for each transmitted Application Data message.




(a) CBC Application Data Message




(b) Next CBC Application Data Message

Figure 6-2: TLS AES-CBC Application Data Messages

## 6.3 TLS Extension Design

The framework extension analyses: TLS protocol messages including handshake, Change Cipher Specification and Application Data message types; AES-CTR and AES-CBC IV characteristics; and properties of encryption keys. By integrating these features, the extension supports the rapid and accurate decryption of TLS communications. The following sub-sections describe extension details for data collection, memory analysis, and decrypt analysis components.

### 6.3.1 Data Collection

The component extracts useful virtual machine data when triggered. Data includes session network packets, cryptographic algorithms and characteristics, and memory extracts from virtual machine processes associated with the TLS session. Cryptographic algorithms and characteristics are obtained from the server Hello message. Fields extracted for the analysis stages are TLS version number, session identifier, and cipher suite. The cipher suite is decomposed into encryption algorithm, mode, and key length, and authentication algorithm and hash length.

Triggers for TLS 1.2 memory extraction are detection of Cipher Change Specification or Application Data messages, and for TLS 1.3, Application Data messages only. These messages indicate that the sender has generated cryptographic artefacts and a key block may exist. Although a single memory extract suffices for key block discovery, the component extracts memory twice to allow for their absence.

For Windows clients, the probable encrypting process is identified by discovering the virtual machine process associated with the network session client and server addresses. Figure 6-3 illustrates the logical steps for Windows client TLS data collection. Ubuntu processes that service client requests are user-initiated, so details for the known process are extracted.

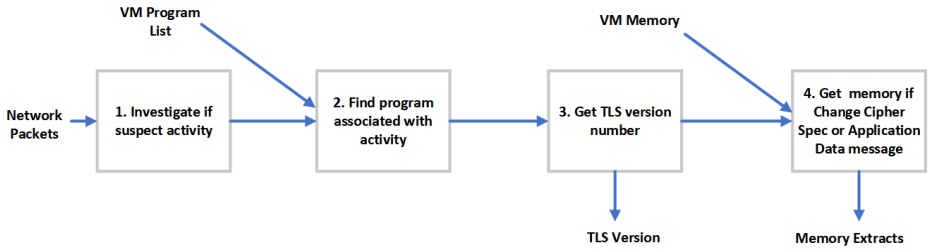



Figure 6-3: TLS Extension AES Data Collection Flow

### 6.3.2 Memory Analysis

This component analyses network packets and memory extracts to discover sets of candidate cryptographic artefacts. In TLS 1.2 AES-GCM, the process for obtaining candidate cryptographic artefacts is to obtain the explicit IV segment, discover the implicit IV segment, and discover key blocks as illustrated in Figure 6-4. If key blocks are not found, a fourth step discovers candidate encryption keys. In AES-CBC, the process is to identify candidate encryption keys and in TLS 1.3 AES-GCM, to discover candidate IVs, and then encryption keys.

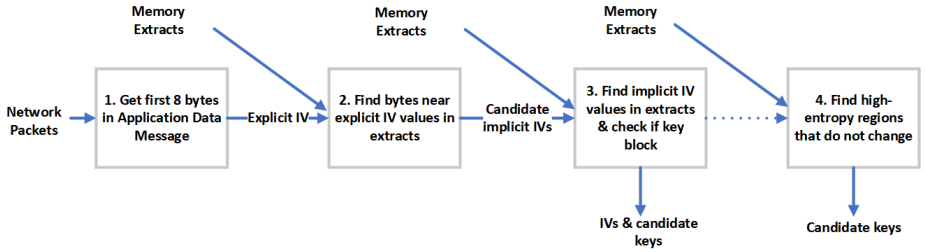



Figure 6-4: TLS Extension GCM Memory Analysis Flow

**TLS 1.2 AES-GCM** Initially, the component extracts the first 8 bytes from a transmitted Application Data message payload, the explicit segment of the 12-byte IV. In the second step, the component searches memory extracts for 8-byte segments closely matching the explicit IV allowing for IV counter increments. AES-GCM encryption uses the complete IV so explicit and implicit segments may be memory co-resident. Consequently, 4-byte segments before 8-byte explicit IV matching segments are identified as candidate implicit IV segments. The third step probes for potential key blocks by searching memory extracts for 4-byte memory regions matching each candidate implicit IV. Matches indicate possible key block presence. The two key length-sized memory segments in a key block antecedent to the client implicit IV are client and server encryption keys. So, the two antecedent segments to candidate implicit IVs are identified as candidate keys if the Shannon entropies of both segments exceed the key length threshold. The implicit IV segments and candidate keys are retained for decrypt analysis. Algorithm 6.1 shows the key block location process.

Complete key blocks may be ephemeral in memory. For instance, they may be partially or fully overwritten prior to a memory extraction. So, when no candidate key blocks are discovered, a final step discovers candidate encryption keys using entropies. Key length-sized memory segments in identical locations in separate extracts, that are static, and have entropies above the key-length threshold are candidate encryption keys. Segments that

**Algorithm 6.1:** TLS 1.2 AES-GCM Key Block Memory Analysis

**Data:** extract folder $fldr$ and Application Data message packet *pkt*
**Result:** X = client keys, Y = server keys, Z = Implicit IVs
ExpIV = $pkt[ExpIVstart : ExpIVstart + 8]$;
**for** *file* $f$ *in* $fldr$ **do**
  $fmem$ = memory($f$) **for** *ExpIV in fmem* **do**
    potentialImpIV = $fmem[ExpIVlocn - 4 : ExpIVlocn]$;
    **if** *entropy (potentialImpIV) > threshold* **then**
      $ImpIVs$ := fmem[ExpIVlocn-4:ExpIVlocn];
    **end**
  **end**
  **for** *ImpIV in ImpIVs* **do**
    **if** *entropy(fmem[ImpIVlocn-64:-ImpIVlocn-32]) > Threshhold & entropy(fmem[ImpIVlocn-32:ImpIVlocn]) > Threshhold* **then**
      X += $fmem[ImpIVlocn - 64 : ImpIVlocn - 32)$;
      Y += $fmem[ImpIVlocn - 32 : ImpIVlocn]$;
      Z += $ImpIV$;
    **end**
  **end**
**end**
where ExpIVstart is the encrypted data offset in an Application Data message, ExpIV is the explicit IV segment, ImpIV is the implicit IV segment, and *locn suffixes are memory locations of respective elements

are located near IVs are prioritised by commencing searches from IV memory locations in order to enhance the rapidity of correct key discovery by the decrypt analysis component.

**TLS 1.2 AES-CBC** In AES-CBC mode, memory analysis only discovers candidate keys. For improved decrypt analysis performance, IV traces are sought in memory extracts to enable probable candidate keys to be prioritised. For example, the last ciphertext block of an encrypted message, such as an Encrypted Handshake message or an Application Data message, may be memory-resident when the extraction takes place. So, the component searches for matches to the last ciphertext block in Application Data messages.

Matched memory locations provide an offset for commencement of candidate key searches and, if unmatched, the searches commence at the first block of each extract file. As with AES-GCM, key-length sized segments with entropies exceeding the key-length threshold, and static across extracts, are retained as candidate encryption keys for processing by the next analysis component.

**TLS 1.3 AES-GCM** The memory analysis component searches for IVs followed by encryption keys. As encrypt IVs result from an XOR between a fixed IV value and a strictly increasing counter, an XOR of two encrypt IVs in a session is equal to the quantity of records transmitted between the two extracts. The IV location process is shown in Algorithm 6.2.

Identified IV memory locations provide an offset for commencement of candidate key searches and if unmatched, the searches commence at extract starts. As with TLS 1.2 AES-GCM and AES-CBC modes, key-length sized segments with entropies exceeding the key-length threshold, and static across extracts are retained as candidate encryption keys for processing by the next analysis component.

**Algorithm 6.2:** TLS 1.3 AES-GCM IV Memory Analysis

```
Data: extract folders fldr_a,fldr_b and packets pkt_a, pkt_b
Result: Z = candidate IVs
delta := packetno[pkt_b] - packetno[pkt_a];
for file f_1 in fldr_a do
    f_2 = match (f_1, fldr_b);
    if f_1 <> f_2 then
        fmem_1 = memory(f_1);
        fmem_2 = memory(f_2);
        for i = 0 to size(f_1) inc 4 do
            if fmem_2[i : i + 12] ⊕ fmem_1[i : i + 12] = delta then
                Z += fmem_1[i : i + 12];
            end
        end
    end
end
```

### 6.3.3 Decrypt Analysis

The component decrypts by iterating through the cryptographic artefact sets and decrypting an Application Data Message until a valid decrypt is obtained or all combinations are exhausted. For AES-GCM in both TLS versions, cryptographic artefacts are sourced from memory analysis, while for AES-CBC, the IV is extracted from the network packet payload's first block and keys sourced from memory analysis. The decrypt validation process is common to all modes .

MemDecrypt analyses HTTP-over-TLS traffic. Although TLS can provide security for high-level protocols [245] such as SMTP, IMAP, POP, [246] and FTP [247], HTTP-over-TLS, also known as HTTPS, is commonly used [248]. The framework validates requests conforming with the HTTP 1.1 specification and is extensible to HTTP 2.0. The format of HTTP 1.1 request messages is:

*request-line:headers:body* [249]

A valid decrypt is inferred where the decrypt adheres to the HTTP 1.1 specification for 'request-line' and 'headers' elements. Validation identifies the session cryptographic artefacts, which then enables the remaining Application Data message payloads to be decrypted.

## 6.4 TLS Extension Implementation

This section presents implementation details for the TLS extension. For each component, the details for both Windows client and Ubuntu server are described where differences exist. Implementations are described for both TLS 1.2 modes of operation and TLS 1.3.

### 6.4.1 Data Collection

The identification of cryptographic algorithms, TLS versions, encrypting processes, and memory extraction are performed by bespoke software. The TLS cryptographic algorithms and TLS version are obtained from the server Hello message. The details for identifying processes and memory extraction are similar to those for SSH Data Collection and are only summarised here for completeness.

Bespoke process identification software differs between Windows clients and Ubuntu servers. For Windows clients, the Volatility *netscantbl* plugin retrieves details, such as process names, identifiers, physical offsets, and source and target addresses of processes with open network ports, and matches TLS client packet source and destination addresses with plugin output source and target fields. For Ubuntu servers, process lists and associated details are generated and recently created *openssl* processes iden-

tified. Candidate process identifiers and associated memory structure pointers are retained for memory extraction.

Bespoke memory extraction software also differs between Windows clients and Ubuntu servers. For Windows clients, the *vadanalyse* Volatility plugin identifies VADs with read/write protection attributes and writes identified memory to a time-stamped file. For Ubuntu servers, the heap memory start and end locations are identified from the memory structure for each new *openssl* process and written to a time-stamped file.

### 6.4.2 Memory Analysis

The memory analysis software is bespoke. For each of TLS 1.2 AES-GCM, AES-CBC, and TLS 1.3 AES-GCM, the processes described in Section 6.3.2 are implemented.

**TLS 1.2 AES-GCM** The implementation may be elucidated with an example using the Wireshark capture shown in Figure 6-5.

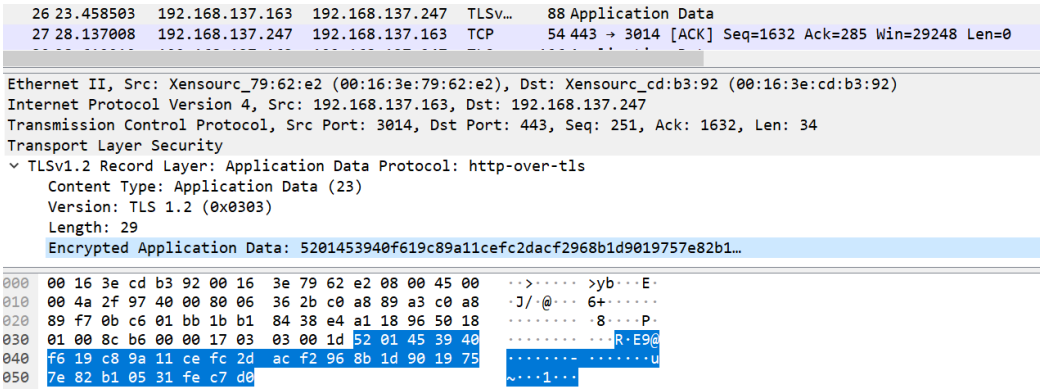



Figure 6-5: TLS 1.2 AES-GCM Application Data Message

The first 8-byte segment of the highlighted encrypted payload in the Application Data message, `0x5201...19c9`, is the client explicit IV. A memory extract search for an explicit IV close match, in this instance the first 7 bytes, discovers segments such as those

```
00055830  50 01 2D 16 03 03 00 28 52 01 45 39 40 F6 19 C7
00055840  87 00 D5 61 B8 13 77 CF 0B FB 0B 79 79 9D E3 00
```

(a) Low entropy candidate Implicit IV

```
0005BD30  E8 C1 34 C8 52 01 45 39 40 F6 19 C7 00 00 00 03
```

(b) High entropy candidate Implicit IV

Figure 6-6: TLS 1.2 Implicit AES-GCM IVs

```
00048E90  2C B0 8F F4 66 92 74 F4 DB 29 84 F9 79 0F 73 03
00048EA0  88 53 F8 C8 46 37 AA 6C 72 C5 22 68 13 CA CE 68
00048EB0  B1 89 BB 7F AB 2E A5 4D F7 86 70 58 4F 74 9E 3F
00048EC0  8F AC F8 8E 3E 03 6A 9F 0A 28 5B 84 A6 F7 30 66
00048ED0  E8 C1 34 C8 89 B0 FB 13 D5 6D 5F 7D 9B A8 4E BD
```

(a)

```
0005BCF0  67 1A C0 B2 DF 0A BF 2B B9 D2 EB 01 1F D7 B6 55
0005BD00  1D BA 51 FC DB 9F E7 D2 A7 E7 E1 DA 72 94 28 49
0005BD10  05 2E FB F2 DA 24 44 D9 63 F6 AF D8 7C 21 19 8D
0005BD20  0E 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
0005BD30  E8 C1 34 C8 52 01 45 39 40 F6 19 C7 00 00 00 03
```

(b)

Figure 6-7: TLS 1.2 AES-GCM Candidate Key Blocks

shown in Figure 6-6. The 4-byte antecedents, such as `0x03030028` in Figure 6-6a and `0xe8c134c8` in Figure 6-6b are temporary candidate implicit IVs. Candidates may be implicit IVs if their entropy exceeds the 4-byte field threshold so only 6-6b is retained as a candidate implicit IV.

As implicit IVs are key block fields, a search in memory extracts for candidate implicit IV segments such as `0xe8c134c8` may discover key blocks as illustrated in Figure 6-7. For each match, entropies of the preceding two key-length fields are calculated. Entropies of both preceding 32-byte segments exceed the threshold in Figure 6-7a only.

So, the concatenation of implicit and explicit segments yields a candidate IV, and the two key block segments are candidate client and server encryption keys. For this example, Table 6-6 shows the candidate cryptographic artefacts.

| | |
|---|---|
| Client IV | xe8c134c85201453940f619c7 |
| Client key | x2cb08ff4669274f4db2984f9790f7303<br>8853f8c84637aa6c72c5226813cace68 |
| Server key | xb189bb7fab2ea54df78670584f749e3<br>f8facf88e3e036a9f0a285b84a6f73066 |

Table 6-6: Key Block Field Example

If the component finds no key blocks, it searches for candidate keys. AES-GCM candidate keys are high-entropy, static segments near candidate IV memory locations. Using candidate IV memory locations, segments at the same address in two separate extracts are compared. Identical segments with an entropy exceeding the threshold identifies a candidate encryption key.

**TLS 1.2 AES-CBC** The memory analysis component discovers candidate encryption keys. After locating the last encrypted block in the network packet associated with the memory extract, the component searches memory extract files for block matches to identify probable extract files and start locations. Then, if segments at the same location in two separate extracts are identical and the entropy exceeds the key-length threshold, candidate encryption keys have been identified for use in decrypt analysis.

**TLS 1.3 AES-GCM** The component discovers candidate keys and then encryption keys. The IV discovery process is illustrated using the two segment extracts shown in Figures 6-8a and 6-8b. An XOR of the first and second segments from extracts taken from two consecutive transmitted Application Data messages is one, which is equal to the quantity of sent records so these are candidate IVs.

Discovering encryption keys follows the standard framework approach, namely that if segments at the same location in two

```
0006B810  52 F5 BB 5B 75 5F 98 A3 A2 58 AC 35
```

(a) 1st segment

```
0006B810  52 F5 BB 5B 75 5F 98 A3 A2 58 AC 34
```

(b) 2nd segment

Figure 6-8: TLS 1.3 AES-GCM IVs

separate extracts are identical and the entropy exceeds the key-length threshold, they are candidate encryption keys.

### 6.4.3 Decrypt Analysis

Component code is bespoke aside from use of the PyCrypto package for decryption. For all variations, decrypt validation is common. A decrypt is considered valid for HTTP-over-TLS if it is consistent with the specified HTTP 1.1 format. In particular, for request lines, the presence of a request method, such as 'GET', 'POST', or 'HEAD', and the HTTP version, such as 'HTTP/1.1' suffices. However, the presence of entity header information, such as 'Connection', 'Accept', 'Accept-Encoding', or 'User-Agent', provides additional validation.

## 6.5 Evaluation

The MemDecrypt extension was evaluated using a TLS library to perform a sequence of experiments with variable file sizes, key lengths, AES modes, operating systems, and operating system versions. The TLS experimental set-up is described followed by a presentation of results for Windows clients and Ubuntu servers. The base MemDecrypt physical environment is supplemented with software to support TLS communications.

### 6.5.1 Experimental Set-up

OpenSSL [222] supports both client and server testing. The software library is an open-source implementation of TLS that provides Unix and Windows command-line utilities incorporating the TLS cryptographic functions. OpenSSL 1.1.1, which supports TLS 1.3, was in beta state during the primary sequence of experiments, so OpenSSL Version 1.1.0g was implemented on client and server virtual machines to evaluate the extension's capacity for TLS 1.2 decryption. OpenSSL command line utilities were executed on the client and server. After generating Ubuntu server certificates and keys, the OpenSSL command line utility server emulated a web server with the command:

*openssl s_server -accept p -cert crt.pem -key key.pem -WWW*

where $p$ is the listening port number, *crt.pem* the server certificate, and *key.pem* the server private key. The Windows client connected to the OpenSSL server with a command of the form:

*openssl s_client [-cipher CIPHER] -connect a.b.c.d:p*

where the optional $CIPHER$ identifies the encryption algorithm, key exchange, and authentication algorithms (e.g. ECDHE-RSA-AES256-GCM-SHA384), *a.b.c.d* is the OpenSSL server IP address and $p$ the OpenSSL server port. Test request and header strings were manually entered at the OpenSSL client console. Request strings are typically of the form:

*GET /[xxx] HTTP/1.1*

where $xxx$ is empty, a server folder, or parameters. Header strings included: *Host: a.b.c.d*, *Accept-Encoding: gzip, deflate*,

and *Accept: */*.

### 6.5.2 Experimental Results

Experiments investigated decrypting TLS traffic for different operating systems, operating system versions, modes, and extract triggers. Except for one scenario, to be discussed below, valid decrypts were obtained for all experiments. The probability of generating an exact valid three character decrypt, such as 'GET', with incorrect cryptographic artefacts is 1 in $3^{256}$. Although other headers, such as 'POST' are possible, validation certainty exceeds 99.99%.

For performance analysis, Table 6-7 summarises the memory and decrypt analysis duration means for the different operating systems or operating system versions.Standard deviations were generally less than 0.01 and, so, are omitted from the table.

Table 6-7: TLS Analysis Duration Means - Operating Systems (secs)

| **Operating System** | **Memory** | **Decrypt** |
|---|---|---|
| Windows 7 | 10.0 | 0.1 |
| Windows 10 | 11.5 | 0.1 |
| Ubuntu 14.04 | 2.0 | 0.8 |

For the remaining experiments memory was extracted from Windows 10 clients using 256-bit key length, AES-GCM mode encryption unless indicated otherwise and Table 6-8 shows the results. The results of each experiment are discussed in the following paragraphs.

The first experiment investigated Windows 7 and Windows 10 client memory extracts taken when Change Cipher Specification and Application Data messages were detected in 256-bit AES-GCM mode. In each test, valid decrypts were discovered for different HTTP-like inputs. The component duration mean values are shown in Table 6-9.

Table 6-8: TLS Analysis W10 Duration Means - Other Variations (secs)

| Experiment | Variation | Memory | Decrypt |
|---|---|---|---|
| Key lengths | 128-bit | 12.1 | 0.9 |
| | 256-bit | 11.5 | 0.1 |
| Modes (C) | AES-CBC | 6.0 | 0.1 |
| | AES-GCM | 11.3 | 0.1 |
| Modes (ASM) | AES-CBC | 6.1 | 0.1 |
| | AES-GCM | 11.5 | 0.1 |

Table 6-9: AES-GCM 256-bit Key Analysis Duration Means (secs)

| Components | Windows 7 | Windows 10 |
|---|---|---|
| Data Collection/extract | 2.0 | 5.0 |
| Memory Analysis | 10.0 | 11.5 |
| Decrypt Analysis | 0.03 | 0.03 |

Although Windows OpenSSL extracts (3.9 MB) are smaller than the SSH application VAD extracts (6.9 MB), TLS data collection extract durations are similar to SSH durations and are not re-analysed. Memory analysis generally discovered 3 candidate explicit IVs in memory extracts. From these, the key entropy threshold filter reduced candidates to 1 as illustrated in the Figure 6-9 example. Each Windows 7 client memory extract is typically marginally over 3 MB, and Windows 10 marginally under 4 MB yielding similar memory analysis durations for each operating system and with a single encryption key and IV combination, decrypt analysis durations are identical.

The second experiment investigated AES-GCM and AES-CBC encryption modes with 256-bit keys on Windows 10 clients. C and assembly language OpenSSL build options were investigated because OpenSSL invokes different encryption modules for each mode. Valid decrypts were produced for both modes with various HTTP-like inputs, irrespective of build option. Over a series of test runs, AES-CBC memory analysis yielded a mean of 936 candidate encryption keys with a standard deviation of 63.2, taking

```
INFO:TLS Memory Analyser:Extract analysis started at 03:42:18PM
INFO:TLS Memory Analyser:Encrypted with AES-256-GCM
INFO:TLS Memory Analyser:3 potential key round(s) found
INFO:TLS Memory Analyser:Client and Server keys found in key round at 03:42:26PM
INFO:TLS Memory Analyser:Client key entropy = 4.9375
INFO:TLS Memory Analyser:Server key entropy = 4.875
INFO:TLS Memory Analyser:Entropies below threshhold
INFO:TLS Memory Analyser:Client key entropy = 3.68582038109
INFO:TLS Memory Analyser:Server key entropy = 0.200622324313
INFO:TLS Memory Analyser:Entropies below threshhold
INFO:TLS Memory Analyser:Client key entropy = 5.0
INFO:TLS Memory Analyser:Server key entropy = 3.3342822216
INFO:TLS Memory Analyser:1 IVs found
INFO:TLS Memory Analyser:Extract analysis complete at 03:42:29PM
```

Figure 6-9: TLS 1.2 AES-GCM Memory Analysis Log

6.0 seconds, in comparison with AES-GCM's 11.5 seconds. The decrypt analysis duration mean for AES-CBC was 0.10 seconds compared to AES-GCM's 0.03 seconds reflective of the increased candidate key set size. As illustrated in Figure 6-10, duration times were independent of the assembler (ASM) or C language (NOASM) build class. In summary, AES-GCM analysis completed within 11.6 seconds and AES-CBC 6.1 seconds.

The third experiment considered outcomes with different AES-GCM key lengths on Windows 10 clients. Valid decrypts were produced for the permitted 128-bit and 256-bit keys with various HTTP-like inputs. With a lower entropy threshold, 128-bit key memory analysis yielded a candidate encryption key set size mean of 11,019 with a standard deviation of 512.5. 128-bit key length analysis duration times were 12.1 and 0.9 seconds for memory and decrypt analysis respectively. Duration differences between the key lengths result from the lower entropy threshold and an absence of key blocks in 128-bit key length memory extracts.

A fourth experiment investigated 256-bit keys on Ubuntu server for both modes. Valid decrypts were produced with HTTP-like inputs for AES-GCM. Although the Ubuntu OpenSSL heap size (267 KB) is larger than the *ssh_server* heap size (176 KB) the

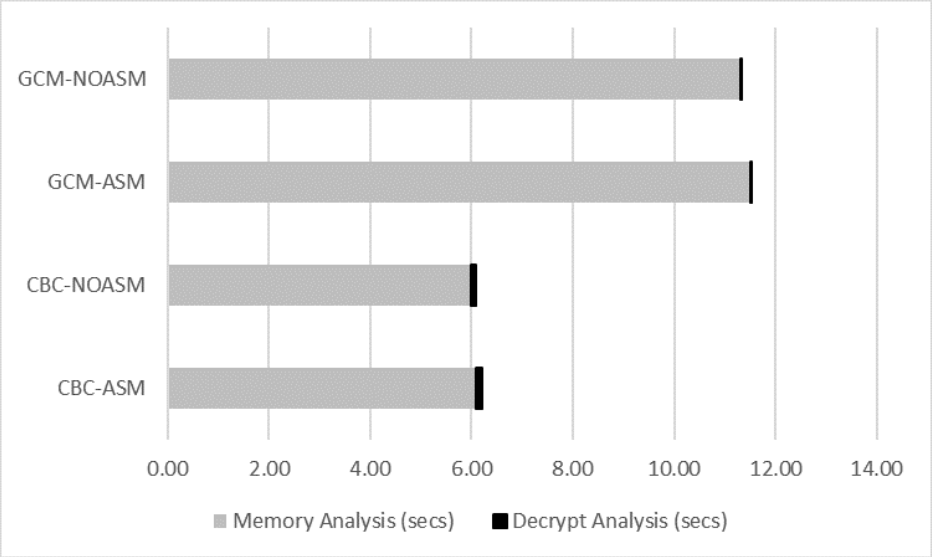



Figure 6-10: TLS 1.2 GCM and CBC Mode Analysis Durations

TLS memory extraction durations of 0.3 seconds are similar to SSH extraction durations and are not discussed further. Key blocks were not discovered in Ubuntu memory. The candidate key set size means were 1,707 and 1,654 for AES-GCM and AES-CBC respectively. Although the AES-GCM encrypted payload decrypted correctly, AES-CBC failed, indicating an absence of the encryption key from the candidate set.

Though TLS 1.3 does not support AES-CBC, this anomaly warranted further analysis. For performance, OpenSSL generates an enhanced AES-CBC key schedule when built with the assembler option. Although key schedules are discoverable in Ubuntu memory, the initial key is absent unlike Windows where key traces remain. However, the AES-CBC decrypt assembler routines themselves can be invoked using the enhanced key schedules.

Following the announcement of an OpenSSL TLS 1.3 compliant version [250], a fifth experiment investigated TLS 1.3 on a Windows 10 client in AES-GCM mode using OpenSSL 1.1.1.b installed on both Windows client and Ubuntu server. To enforce use of TLS 1.3, server commands were revised to be of the form:

*openssl s_server -tls1_3 -accept 443 -key key.pem -cert crt.pem -WWW*

where *tls*1_3 stipulates the TLS version. Memory analysis test runs yielded 1 candidate IV and approximately 2000 candidate keys. Using these artefacts, the decrypt analysis component identified the correct key through a similar process to TLS 1.2. The memory analysis and decrypt analysis duration means were 11.9 seconds and 0.8 seconds, respectively.

## 6.6 Conclusions

The experiments demonstrated that encrypted TLS traffic can be decrypted with a high degree of certainty when the well-known OpenSSL library is used. Implementation alternatives such as WolfSSL [251], a C-language library intended for embedded systems, libreSSL [252], a fork of OpenSSL which focuses on BSD systems but has been ported to Windows, GnuTLS [253], a C-language library running on Linux and Windows platforms, and NSS [254], used for Red Hat and Oracle server cryptographic libraries, may exhibit similar vulnerabilities.

The assumptions made in constructing the extension could limit effectiveness. For TLS 1.2 AES-GCM, candidate key block identification is predicated on discovering candidate implicit IV segments. Separating actual keys, explicit IV, and implicit IV in memory by secure protocol implementers may increase candidate set sizes, so these are potential decrypt analysis delay measures. For TLS 1.3, moving IV memory locations around slowed memory analysis, albeit by less than 1 second.

The results indicate that secure Internet communications between clients and servers may be vulnerable, particularly in virtualised environments. For HTTPS, the probability of correct

decrypts being generated exceeds 99.99%. Framework analysis decrypts takes place in approximately 12 seconds so performance suffices for user-initiated, but not software-initiated, sessions. By multi-threading, memory analysis completion in 1 second may be achievable. As TLS 1.3 progresses to becoming the default protocol version, MemDecrypt is already enabled.

# Chapter 7

# Discovering Malware Activity Without Prior Knowledge

## 7.1 Introduction

Malicious actors employ secure channels to hide communications from detection agents. These channels facilitate activities such as the installation of exploit kits, distribution of malware and adware, and communication of confidential information to controllers. Knowledge of channel contents is unknown, while use of secure channels for malicious purposes surges [105]. Malware classes that frequently use TLS secure channels for communications are bots and ransomware [58].

Modern bot malware is often multi-purpose [255]. Once installed on a client and a channel established, an external controller determines bot activities through issued commands. Encrypted controller-to-bot channels prevent defenders from discovering commands or knowing what information is being extracted.

By contrast, ransomware clients are generally single purpose in that users pay to recover documents or regain access to a device. A common variant is crypto-ransomware, where documents on an infected client are encrypted and a payment required, typically in Bitcoins, for the decryption key [256]. Communications between crypto-ransomware clients and controllers may include useful in-

formation such as encryption keys.

Existing methods that deal with the malicious use of secure channels have limitations. Unlike unencrypted channels, where payloads provides knowledge of malware activity, with encrypted channels, analysis relies on discovering anomalies between benign and malicious activity using data mining methods [257]. Differentiators such as channel request and response times [258], short packet sizes [259], TLS header information [9], or a combination of features [10] assist in the possible detection and prevention of malicious activity. Knowledge of the plaintext may enable the development of more effective methods for dealing with malware.

Plaintext of malware client encrypted communications can be obtained by logging client requests in an emulated botnet controller. Environment security, scalability for large botnets, and transparency, where malware detects the presence of a test environment and terminates, may present challenges [260]. Although controlled environments are useful for analysis [261], such as the detection of adversarial activities [129], drawbacks exist for decrypting real-world malware communications. There may be a knowledge gap in valid controller responses but, perhaps more importantly, prior knowledge of the malware is typically required for its execution in an emulator environment.

Memory analysis supports malware forensics. For instance, Patil et al. [175] define an investigation framework for analysing captured memory for the detection of malware using process information, running threads, opened registry keys, and user authentication details. Memory inspection can also detect malware using virtual machines to discover anomalous behaviour [262]. These approaches do not decrypt network sessions.

This chapter aims to discover the plaintext of encrypted communications sent by potentially unknown malware. Initially, real bot and ransomware samples are identified. The MemDecrypt TLS extension, constructed and evaluated in the previous chapter,

is applied to analyse malware sample communications. As framework performance challenges result from malware use of different cryptographic libraries, an additional extension is constructed to accommodate the Windows cryptographic library. Experiments evaluate decrypting real bot and ransomware command and control communications using the new extension.

## 7.2 Sourcing Malware Samples

Malware communications analysis ideally executes real malware samples. Samples of recent provenance are preferred as these reflect the approaches of modern malware authors. Maintained online databases provide guides to current malware usage. For TLS traffic, the SSL Blacklist website [263] lists bot and ransomware clients in reverse chronological order. For these investigations, 21 potential malware entities were manually identified.

For each entity, one or more executable samples were downloaded in compressed, password protected format from VirusShare [264]. The 36 downloaded executables and their MD5 hashes are listed in Appendix D. The compressed files were securely copied to the client, on which Windows Defender was disabled, the compressed files uncompressed and executed.

As the responder, acting as a malware controller, was not configured to respond appropriately for the malware client, only three executables successfully established a TLS connection including a data exchange. As shown in Appendix D, other samples exited for a variety of reasons including IP validation, use of UDP, and HTTP POST and GET failures, including one requesting access to a site hosting the US Constitution. The three malware executables that connected with the OpenSSL server, and their MD5 hashes, are shown in Table 7-1. To provide context, background information for these samples is provided.

Zbot, also known as ZeuS or Zeus, is a well-known bot mal-

Table 7-1: Malware Samples

| Type | Class | MD5 |
|---|---|---|
| Bot | Zbot | eeef1e062c8011cabb23b3c833ff766a |
| Ransomware | Torrentlocker | aeb5bb78ab442bc94bb94d968754e523 |
| Bot | Gozi | 67a775879d3664456cb6a5026c518ca0 |

ware instance. Detected in 2006 [265], Zbot is primarily known for stealing banking passwords by injecting code into a user browser as illustrated in Figure 7-1 [266]. Other Zbot functionality includes: extracting information such as browser history and cookies, certificates, and mail account information; manipulating local files; installing ransomware; logging keystrokes; taking screenshots; and managing botnets of other infected computers [267] [268]. Although Zbot once used the HTTP protocol for client-controller communications, information and commands are now generally concealed in TLS payloads.


Figure 7-1: Zbot Fake

Gozi, also known as Ursnif inter alia, is also an information-stealing bot. Detected in 2007 [269], Gozi is commonly used by malicious actors for stealing banking and other confidential information [270] [271] [272] as shown in Figure 7-2 [269]. Although functions include theft of cookies and email credentials, and logging of keystrokes and browsing activity [273], a key Gozi feature is intercepting network traffic to hijack financial transactions. For example, when a money transfer is detected, Gozi issues an encrypted message through its command and control server to prevent the valid transfer and redirect the funds to a controlled account [274].

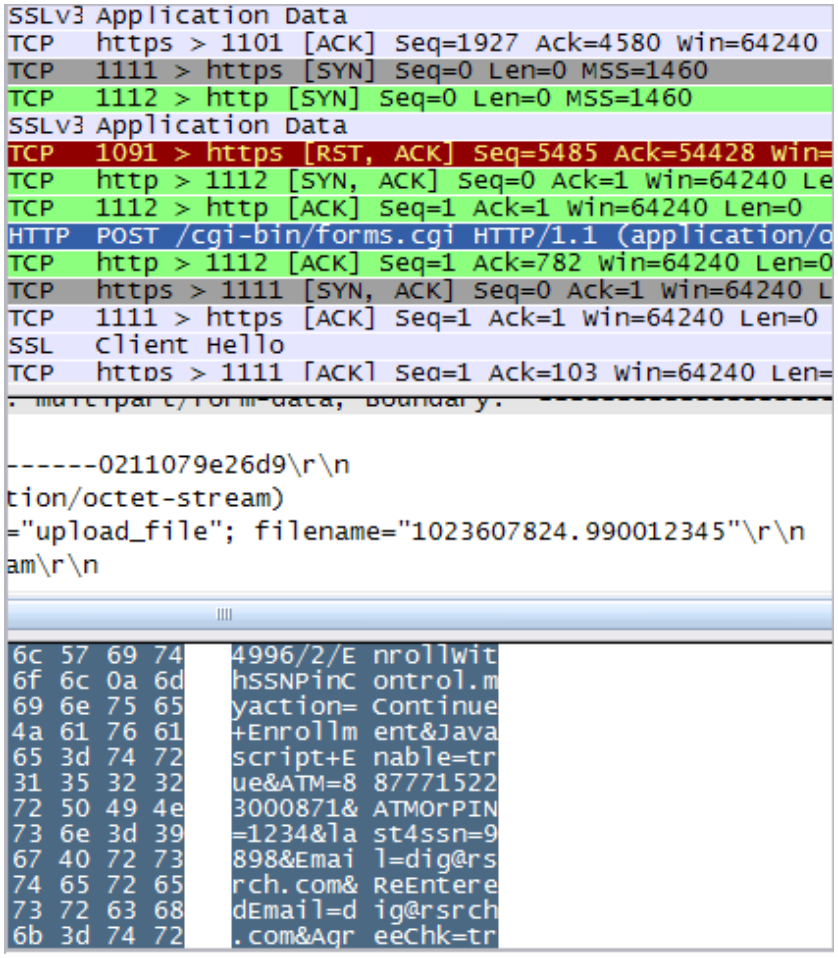



Figure 7-2: Gozi Data Theft

TorrentLocker is an example of crypto-ransomware. Known since 2014, and sufficiently similar to CryptoLocker ransomware to also be known as Crypt0L0cker, it encrypts user documents, advises the user of what has transpired, and demands payment [275] as indicated in Figure 7-3. Although client-controller communications were previously encrypted using XOR, TLS is now a common communications mechanism [276]. Information transmitted to the controller includes the ransom page, the encryption key, which is RSA-encrypted with a TorrentLocker public key, counts of encrypted files, address book contacts, email credentials, and logs [275].


Figure 7-3: TorrentLocker/Crypt0L0cker Warning

## 7.3 OpenSSL Extension Evaluation

Each of the three samples negotiated agreement on TLS version 1.2 using AES-GCM encryption with 256-bit keys. The TLS extension that was evaluated with OpenSSL supports this TLS version, encryption algorithm, and mode, and so can be applied.

The extension memory analysis component searched for key blocks adopting the three-step process: find the explicit IV segment in an Application Data message network packet, search for candidate implicit IVs, and then search for candidate key blocks. Candidate implicit IVs are memory extract segments co-located with explicit IV segment values. Candidate key blocks are memory extract segments containing candidate implicit IV segments, and where the Shannon entropies of the client key and server key fields exceed a threshold.

The test environment configuration differed in one crucial respect from the OpenSSL experiment testbed. The malware client was prevented from communicating with external servers, as the target might be a real malware controller, which could seek to corrupt other environments. So, a virtual machine was created and established as a DNS server with the 'dnsmasq' package. Responses to benign DNS requests, such as *.microsoft.com, returned the DNS server IP address, and to other requests, the IP address of the target TLS server. For the first experiment, debug mode was enabled for the server to log keys, IVs, and plaintext. The OpenSSL server commands entered on the Ubuntu server command-line were:

*openssl s_server -accept 443 -debug -cert crt.pem -key key.pem -WWW*

The OpenSSL TLS extension acquired memory, and attempted analysis and decryption with Zbot. In the first test run, decrypt analysis duration for the correct artefacts to be identified was projected to be approximately 34 hours. By excluding keys and IVs more than 1 KB apart in memory extracts, this duration reduced to 15 minutes with the result that the combination of memory and decrypt analysis required a total of 38 minutes when the test was re-run. Although performance exceeded brute-force, crypt-

analytic, and side-channel approaches, the duration may be insufficient for practical application in live scenarios. Furthermore, the duration was substantially longer than experiments with, for example, the OpenSSL library, warranting further analysis.

Two factors cause the increased duration. The first is the 8-byte explicit IV segment obtained from an Application Data Message. For Zbot, the first explicit IV segment was `x0000000000000001` as illustrated in the highlighted section of the Wireshark packet capture of Figure 7-4. This byte sequence occurs considerably more frequently in memory extracts than randomly generated explicit IVs.

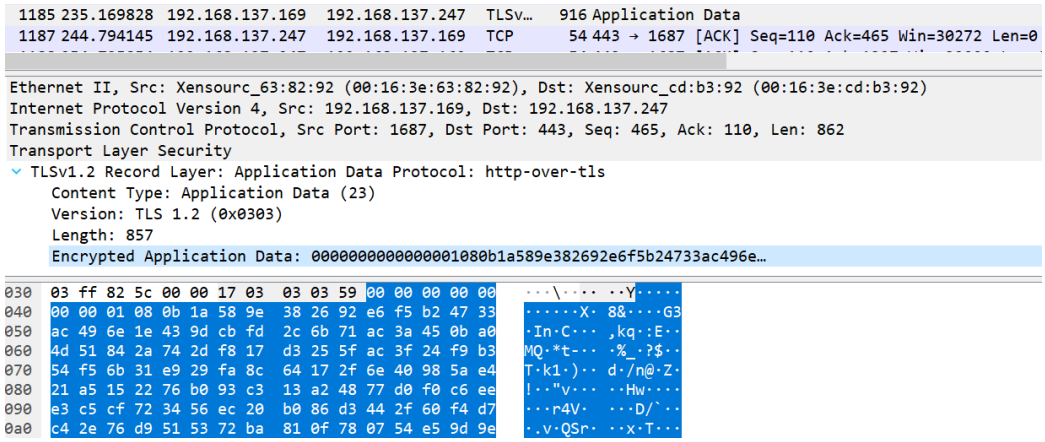



Figure 7-4: Zbot Application Data Message

The second factor is masquerading. To evade detection, malware applications may camouflage their activities and one such mechanism is masquerading as a benign application, sometimes known as 'process hollowing'. In the Windows environment, examples of benign applications used for masquerading include the Edge browser and Windows Explorer. When Zbot masqueraded as Windows Explorer, the data collection component extracted 265 read/write memory files totalling 73.2 MB for each separate extraction.

So, when the explicit IV was used in searching for possible 4-

byte implicit IV segments, 578,629 possible instances were found. Entropy measure thresholds reduced the candidate key block set size to 23,361. By comparison, an experiment with the OpenSSL client application yielded only 3 candidate implicit IVs and 79 candidate keys. Large sets of candidate keys and IVs take longer to analyse.

## 7.4 Windows Library Extension Design

Memory extract features suggested the existence of a more efficient alternative. Using the session key and IV logged by the OpenSSL server, a search of malware client application memory yielded interesting facts: the key occurs frequently in different extract files; memory extract files sizes containing the key are within specific ranges; and two unusual ASCII strings are present near encryption key locations in the memory extract files.

The repeated occurrence of the key in memory extracts may be due to protection of essential data or the absence of data cleansing. After the TLS handshake, when client and server keys have been generated by a pseudo-random generator, keys and implicit IVs may be copied to record data structures for simplified access by encryption processes. The malware may also copy cryptographic artefacts to multiple locations to ensure access. Alternatively, the malware writer may copy the artefacts for different purposes but fail to cleanse. This feature is not used in the new extension.

Sizes of memory extract files containing encryption keys ranged between 2 MB and 4 MB. Consistent with prior MemDecrypt SSH and TLS investigations, these sizes probably originate from application memory allocation requests ('malloc') for cryptographic data structures to hold the encryption and decryption fields, such as keys, encrypt/decrypt flags, key length, and mode. As illustrated in Figure 7-5 which maps the number of segments above a 4.5 threshold (Y-axis) against memory extract sizes (X-axis), the

distribution of high-entropy regions in malware application memory is uneven, which suggests that prioritising regions for memory analysis might improve the performance of the candidate IV and key discovery process.

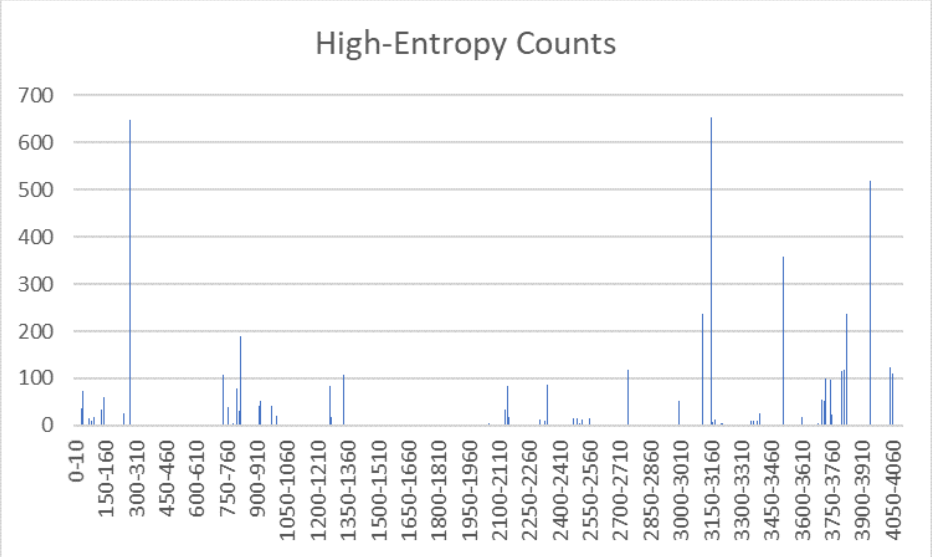



Figure 7-5: Windows Explorer High-entropy Regions

The presence of specific ASCII strings in memory extracts containing keys is more significant. The strings are '3LSS' and 'KSSM', or in big-endian format 'SSL3' and 'MSSK'. A researcher identified 'MSSK' in the Windows security policy application, Local Security Authority Subsystem Service (LSASS), as a possible acronym for 'MicroSoft Symmetric Key' or 'Microsoft Software Symmetric Key' [277]. 'SSL3' may refer to SSLv3, the deprecated forerunner of TLS. Possible fields identified in the undocumented LSASS data structure included encryption data structure sizes, TLS version, and the encryption key.

The field identified by that researcher as a probable IV field is inconsistent with memory extracted by MemDecrypt. In memory extracts, the implicit IV was located approximately 20 bytes after the '3LLS' string, and the key approximately 30 bytes after the 'KSSM' string. Although random occurrences are possible, these

ASCII strings can provide good indicators for the identification of candidate memory extracts containing cryptographic artefacts when Windows security libraries are used.

An extension was constructed to accommodate these features to support the decryption of TLS communications from executables that use Windows security libraries. These libraries are assumed to be employed when TLS Application Data messages contain the explicit IV value `x0000000000000001`. Additional techniques, such as the identification of executable linked libraries, could be used to validate this assumption. For the extension, extract file sizes are banded to prioritise medium-sized files and then searched for the ASCII strings. Memory segments near the string locations in the same extracts and with Shannon entropies above the threshold are identified as candidate keys and IVs.

The memory analysis process for the Windows library extension is shown in Algorithm 7.1. The Algorithm's banding is wider, and the maximum distances in memory between '3LSS' and a candidate IV, and 'KSSM' and a candidate key, larger than the empirically observed values. This allows for potential data structure changes such as may result from operating system upgrades. If the Windows library extension fails to find cryptographic artefacts, the TLS extension provides a slower alternative.

## 7.5 Windows Library Extension Evaluation

Experiments were executed to evaluate the Windows library extension. Each malware sample was executed on a Windows 10 client, virtual machine data extracted, and logs collected from the Ubuntu server with OpenSSL debugging disabled. Decrypts were validated by evaluating compliance with HTTP 1.1. Complying decrypts were also checked through comparison with the OpenSSL server logs.

The first client Application Data message for the Zbot, Gozi,

**Algorithm 7.1:** Windows Library TLS Memory Analysis

```
Data: Extracts folder, entropy thresholds, keysize
Result: Z = candidate keys, Y = candidate IVs
for file in folder do
    if 1 MB < size(file) < 8MB then
        Band1 += file;
    end
    if 0 MB < size(file) < 1MB then
        Band2 += file;
    end
    if 8 MB < size(file) then
        Band3 += file;
    end
end
IVsize = 4;
for each file in Bands 1-3 do
    if 'KSSM' in extract then
        start = location ('3LSS');
        for i = start to start + MaxIVdistance inc by 4 do
            s = extract[i:i+4];
            if entropy(s) > threshold(IVsize) then
                Y += s
            end
        end
        Start = location 'KSSM';
        for i = start to start + MaxKeydistance increment by 4 do
            s = extract[i:i+keysize];
            if entropy (s) > threshold(keysize) then
                Z += s;
            end
        end
    end
end
where MaxIVdistance and MaxKeydistance are the maximum search
 distances from the ASCII string locations for IV and keys
 respectively
```

and TorrentLocker samples decrypted with 100% success on each test run. An example of Zbot decrypt analysis output is illustrated in Figure 7-6. Further confirmation is provided by the OpenSSL server log shown in Figure 7-7.

```
INFO:TLS Decrypt Analyser:Encrypted with AES-256-GCM
INFO:TLS Decrypt Analyser:AES GCM Packet analysis started at 02:56:56PM
INFO:TLS Decrypt Analyser:No of client IVs = 3
INFO:TLS Decrypt Analyser:No of client keys = 79
INFO:TLS Decrypt Analyser:No of server keys = 79
INFO:TLS Decrypt Analyser:Session 1 packets from 1 to 38
GET
/images/iWjLmuDOy7/XYiI3c6Yc19CBsArB/rO6Z4SdMsmUI/_2FtdYcDlyk/LoNJcQlukRof2E/wEtt4hPTStlxvgVkkzKsg
/60jtn5ijt6M8aubv/YKQoZfCXiA4x01o/rjcmP9NhrPGMZrGeCb/mWLp7cs9z/83p8sChdoB_2FBMuejFT/IP3nyBFsFoV/U.jpeg HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 10.0; Win64; x64)
Host: aineiounfafjnafjwnaefoiajwnf.net
Connection: Keep-Alive
Cache-Control: no-cache
 has header information
INFO:TLS Decrypt Analyser:Valid client key at offset 270076 in VAD explorer.exe.082550.159907840-164102143.dmp at position 0
INFO:TLS Decrypt Analyser:Valid IV at offset 269928 in VAD explorer.exe.082550.159907840-164102143.dmp
INFO:TLS Decrypt Analyser:Decrypt analysis complete at 02:56:56PM
```

Figure 7-6: Zbot Decrypt Log

As previously observed, the probability of obtaining GET and POST with incorrect cryptographic artefacts is low (approximately 2 in $3^{255}$) so false positives are unlikely. Table 7-2 shows examples of decrypt output for all three malware samples. Host names and GET request image names varied for different test runs. Gozi decrypts also yielded POST as well as GET requests.

Analysis component durations for each malware sample confirmed the benefit of using a Windows cryptographic library-specific extension. As Table 7-3 shows, the maximum elapsed duration for both memory analysis and decrypt analysis completion was below one second. These results followed from the reduced candidate cryptographic artefact set sizes, as candidate IV set sizes now ranged between 3 and 6, and candidate encryption key set sizes between 79 and 483.

dec 383
47 45 54 20 2F 69 6D 61 67 65 73 2F 69 57 6A 4C
6D 75 44 4F 79 37 2F 58 59 69 49 33 63 36 59 63
31 39 43 42 73 41 72 42 2F 72 4F 36 5A 34 53 64
4D 73 6D 55 49 2F 5F 32 46 74 64 59 63 44 6C 79
6B 2F 4C 6F 4E 4A 63 51 6C 75 6B 52 6F 66 32 45
2F 77 45 74 74 34 68 50 54 53 74 6C 78 76 67 56
6B 6B 7A 4B 73 67 2F 36 30 6A 74 6E 35 69 6A 74
36 4D 38 61 75 62 76 2F 59 4B 51 6F 5A 66 43 58
69 41 34 78 30 31 6F 2F 72 6A 63 6D 50 39 4E 68
72 50 47 4D 5A 72 47 65 43 62 2F 6D 57 4C 70 37
63 73 39 7A 2F 38 33 70 38 73 43 68 64 6F 42 5F
32 46 42 4D 75 65 6A 46 54 2F 49 50 33 6E 79 42
46 73 46 6F 56 2F 55 2E 6A 70 65 67 20 48 54 54
50 2F 31 2E 31 0D 0A 55 73 65 72 2D 41 67 65 6E
74 3A 20 4D 6F 7A 69 6C 6C 61 2F 34 2E 30 20 28
63 6F 6D 70 61 74 69 62 6C 65 3B 20 4D 53 49 45
20 38 2E 30 3B 20 57 69 6E 64 6F 77 73 20 4E 54
20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36
34 29 0D 0A 48 6F 73 74 3A 20 61 69 6E 65 69 6F
75 6E 66 61 66 6A 6E 61 66 6A 77 6E 61 65 66 6F
69 61 6A 77 6E 66 2E 6E 65 74 0D 0A 43 6F 6E 6E
65 63 74 69 6F 6E 3A 20 4B 65 65 70 2D 41 6C 69
76 65 0D 0A 43 61 63 68 65 2D 43 6F 6E 74 72 6F
6C 3A 20 6E 6F 2D 63 61 63 68 65 0D 0A 0D 0A

GET images/iWjLmuDOy7/XYiI3c6Yc19CBsArB/rO6Z4SdMsmUI/_2FtdYcDlyk/
LoNJcQlukRof2E/wEtt4hPTStlxvgVkkzKsg/60jtn5ijt6M8aubv/YKQoZfCXiA4x01o/
rjcmP9NhrPGMZrGeCb/mWLp7cs9z/83p8sChdoB_2FBMuejFT/IP3nyBFsFoV/U.jpeg HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 10.0; Win64; x64)
Host: aineiounfafjnafjwnaefoiajwnf.net
Connection: Keep-Alive
Cache-Control: no-cache

Figure 7-7: Zbot Server Log

## 7.6 Conclusions

Valid decrypts were obtained with high probability within 1 second so the extension may be applicable in software-initiated scenarios. Furthermore, it can be inferred that MemDecrypt rapidly decrypts secure TLS communications between malware clients executing on Windows clients and their controllers despite the small

Table 7-2: Malware Decrypt Analysis Output Examples

| Malware | Decrypt |
|---|---|
| Zbot | GET /images/iWjLmuDOy7/XYiI3c6Yc19CBsArB<br>/rO6Z4SdMsmUI/_2FtdYcDlyk/LoNJcQlukRof2E/<br>wEtt4hPTStlxvgVkkzKsg/60jtn5ijtM8aubv/YKQoZ<br>fCXiA4x01o/rjcmP9NhrPGMZrGeb/mWLp7cs9z/83p8s<br>ChdoB_2FBMuejFT/IP3nyBFsFoV/U.jpeg HTTP/1.1<br>User-Agent: Mozilla/4.0 (compatible; MSIE 8.0;<br>Windows NT 10.0; Win64; x64)<br>Host: aineiounfafjnafjwnaefoiajwnf.net<br>Connection: Keep-Alive<br>Cache-Control: no-cache |
| TorrentLocker | POST /topic.php HTTP/1.1<br>Accept: */*<br>Host: orbfoz.drinkmilks.org<br>Content-Length: 176<br>Cache-Control: no-cache |
| Gozi | GET/images/SZK2b21jT4lHGlPQ/dzUZdQ7R6GQ9vDq/<br>_2Bubb8ji761TySLT2/0hh3OGhrh/Gp933q_2<br>BuLnl3Zz6zem/PlUvBGkYtrVJ8DUn8vN/LcmmaA101Yc<br>rEmAU5RwlEy/zK8Z9xa06n3R/g0EFUOhv/8sDfI2Goa38<br>03Voj6biFmRG/JDejp7Gffn/y7.jpeg HTTP/1.1<br>User-Agent: Mozilla/4.0 (compatible; MSIE 8.0;<br>Windows NT 10.0; Win64; x64)<br>Host: wjenqwdqwdwdqwd.com<br>Connection: Keep-Alive<br>Cache-Control: no-cache |

sample size. Although malware writers have implemented customised security software, errors have led to analysts developing resolutions [278], so, instead, known cryptographic libraries may be commonly used. The Windows library presents a good opportunity for malware writers as it is pre-loaded with the Windows operating system. Use of alternatives, such as the OpenSSL library, might require additional downloads thereby increasing the risk of detection.

The framework decrypts single malware requests, which may

Table 7-3: Malware Extension Analysis Durations (secs)

| | **Memory Analysis** | **Decrypt Analysis** |
|---|---|---|
| Maximum | 0.21 | 0.41 |
| Minimum | 0.18 | 0.03 |
| Mean | 0.17 | 0.16 |
| Standard Deviation | 0.03 | 0.18 |

not be conclusive. The lack of longer decrypted sessions results from the basic configuration as the OpenSSL test server responds with an HTTPS 'OK' message to most TLS requests. In the absence of an expected response, malware may terminate or cease further communications with the controller. Decryption may not necessarily provide immediately usable information, particularly where the plaintext includes a secondary encryption layer as with TorrentLocker. However, having discovered the key and IV, a complete session is decryptable.

Rapid decryption of live TLS malware traffic offers significant opportunities. For instance, by permitting the malware to communicate with its real controller and managing impact to local environments, deeper knowledge of client-controller interactions may be obtained and contribute to enhanced malware defences. More importantly, by acquiring memory and using it to decrypt suspect communications, it may be possible to discover ransomware keys or stolen banking details in communications without prior knowledge of the malware's existence.

# Chapter 8

# Deriving ChaCha20 Key Streams From Targeted Memory Analysis

## 8.1 Introduction

For encryption, ChaCha20 is currently the only alternative to AES that is supported by both SSH and TLS 1.3. Concerns with the lack of an AES alternative, should a major vulnerability be discovered, as well as the security and speed of ChaCha20, have led to its proposed adoption for other protocols such as IPSec [279]. The Google decision to use ChaCha20 in preference to RC4 for Chrome TLS communications on Android smartphones [99] may encourage further interest in use of the algorithm.

To date, ChaCha20 has resisted effective decryption. Researchers have been unable to develop effective cryptanalytic techniques for ChaCha20 key discovery [102]. Cache-timing attacks, which are principally reliant on S-Box access, are ineffective. ChaCha20 keys may be discovered through monitoring power or electromagnetic signals with sufficient resources [280]. This thesis presents the first study into decrypting live secure communications that use ChaCha20 encryption through memory analysis.

A ChaCha20 extension was designed and constructed to investigate decrypting SSH and TLS 1.2. Context for the extension is provided with a brief description of the ChaCha20 algorithm,

followed by a discussion of common ChaCha20 implementations for SSH and TLS protocols [99]. A description of the ChaCha20 extension design and construction for memory and decrypt analysis components follows. Experiments were conducted with both SSH and TLS 1.2 and these results are presented and discussed.

## 8.2 Background

Whereas block algorithms encrypt fixed blocks of plaintext, stream algorithms generate key streams, which are typically XOR-ed bytewise with plaintext to generate ciphertext [92]. This generally makes stream algorithm software-only implementations demonstrably faster than equivalent block algorithms [98], although possibly more difficult to implement [77]. The following subsections describe ChaCha20 and its features in SSH and TLS implementations.

### 8.2.1 ChaCh20 Description

Stream and block algorithms have both been used in secure protocols. However, with vulnerabilities leading to the planned deprecation of the RC4 stream algorithm for protocols such as SSH [96] and TLS [95], alternative stream algorithms have been sought. The ChaCha20 stream algorithm with the Poly1305 authenticator [79] has been adopted in secure protocol implementations, such as OpenSSH [97] and OpenSSL [222].

ChaCha20 generates key streams of 64-bytes from a key, nonce, and counter. For reasons of performance and security, inputs to ChaCha20 key stream generation are independent of plaintext or ciphertext [100], like other eSTREAM proposals [281], but unlike stream algorithms such as Helix.

ChaCha20 enables parallel ciphertext generation with consequent performance improvement. 20 rounds of mathematical cal-

culations are performed with XOR, addition, and rotation operations using as inputs, four 4-byte constants, a pseudo-random 32-byte key, a 4-byte counter, and a 12-byte nonce. The 4-byte constants are `0x61707865`, `0x3320646e`, `0x79622d32`, and `0x6b206574`, or, in ASCII, 'apxe', '3 dn', 'yb-2', and 'k et'. In ChaCha20 these strings are concatenated. The counter, which typically starts at 0 or 1, increments for each 64-byte plaintext block [79].

### 8.2.2 ChaCha20 Implementations

SSH and TLS implementations comply with the ChaCha20 and Poly1305 for IETF Protocol RFC [79]. The RFC is based on Bernstein's ChaCha20 algorithm proposal [101], a variation of the earlier Salsa20 algorithm [100]. However the number of key streams, and nonce and counter generation differ between SSH and TLS ChaCha20 implementations.

In the SSH *chacha20-poly1305@openssh.com* implementation, memory is allocated to hold a data structure comprising key stream input fields, the 64-byte key stream itself, and an index pointer. The packet lengths are encrypted separately from the remainder of the payload so four structures are required: two for encrypting outgoing lengths and payloads, and two more for incoming encrypted data. The memory contents of the concatenated constant string are 'expand 32-byte k'. The nonce is a sequence number, and counters for the encrypted packet lengths and payloads are zero and unity respectively.

The OpenSSL implementation of Chacha20-Poly1305, which follows the IETF RFC ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS) specification [98], has different features. As TLS does not encrypt packet lengths, two memory structures are used: one for encrypting outgoing messages, the other for decrypting incoming messages. Another difference is that the nonce is an XOR of the sequence number and the initialisation vector

generated during the initial handshake when the algorithm keys are obtained. Memory used for ChaCha20 encryption and decryption process parameters is temporary, as the encryption structure is assembled from other sources when required.

## 8.3 ChaCha20 Extension Design

The ChaCha20 extension design details are presented in the following paragraphs. For AES encryption, SSH and TLS MemDecrypt framework extensions were constructed for data collection, memory analysis, and decrypt analysis. The ChaCha20 data collection extensions are identical to the AES extensions for each protocol so only a brief description is included for completeness.

**Data Collection** The component captures network packets and extracts volatile memory of the targeted virtual machine. SSH and TLS sessions are analysed and packets retained for later examination. The trigger for extraction of target volatile memory is detection of a protocol message signalling completion of the protocol handshake phase. The messages are 'New Keys', and 'Change Cipher Specification' or 'Application Data', for SSH and TLS version 1.2, respectively.

When the messages are transmitted, the sending party advises it has sufficient material to generate cryptographic artefacts. The artefacts are likely to be memory-resident so memory can be extracted for any subsequent outgoing message in the network session. Volatility plugins and LibVMI routines enable process identification and memory extraction for Windows clients and Ubuntu servers, respectively.

**Memory Analysis** Candidate ChaCha20 cryptographic artefacts can be discovered in memory extracts. Initially, the component

searches for the constant string 'expand 32-byte k' to discover candidate ChaCha20 data structures. In view of its length, the string is unlikely to exist in memory segments that are not ChaCha20 structures.

Nevertheless, as a precautionary measure, a second step assesses whether the 32-byte memory segment after the constant string in a candidate base structure is a candidate key by evaluating its Shannon entropy. If the entropy exceeds the 32-byte entropy threshold, the data structure comprising key, nonce, and counter is a candidate cryptographic artefact group.

**Decrypt Analysis** Cryptographic artefact groups discovered by the memory analysis component are input parameters in decrypt analysis. In each instance, the candidate key and nonce group is used to decrypt and validate using similar approaches to those used in the AES SSH and TLS extensions. For TLS, decrypts are analysed to establish compliance with the TLS 1.2 protocol specifications.

For SSH, the groups are used to decrypt incoming and outgoing packet lengths and payloads. The packet length groups are identified when the decrypted packet length meets Equation 8-1 for short packets, typically in the authentication and channel set-up phases. This equation is identical to that used for AES encryption algorithms. For larger packets, a modified equation supports SSH packet reassembly.

$$\begin{aligned} packet\ data\ length\ = \\ & decrypted\ packet\ length\ + \\ & size(packet\ length field)\ + \\ & size(MAC field) \end{aligned} \tag{8-1}$$

## 8.4 ChaCha20 Extension Implementation

This section presents implementation details for the ChaCha20 extension. For each component, the details for both SSH and TLS are described where differences exist.

The data collection component inspects virtual machine network traffic and extracts memory. The ChaCha20 MemDecrypt extension identifies ChaCha20-Poly1305 as a valid algorithm for SSH and TLS and retains the details for memory analysis. In other respects, the implementation is similar to the SSH and TLS extension implementations when AES is the encryption algorithm.

The memory analysis extension implements bespoke code to obtain ChaCha20 candidate cryptographic artefacts. Each memory extract file is searched for the constant string. If the entropy of the following 32-byte block exceeds the threshold, a base structure is presumed to be identified and the key, nonce, and counter fields are retained as a group for decrypt analysis. The memory analysis approach is shown in Algorithm 8.1.

The decrypt analysis component iterates through key, nonce, and counter groups and verifies the decrypt in accordance with the protocol specification.

For SSH decrypt analysis, the first four encrypted bytes in packets with encrypted payloads are decrypted using the Python Chacha20poly1305 package with counter value zero [282]. For the valid packet length group, the first four bytes represent the packet length and Equation 8-1 holds. For the valid payload group, the decrypted padding length must lie between 4 and 255 as specified in the SSH Transport Layer Protocol [232]. When the payload group is valid, the remaining encrypted message payloads are decrypted, and examined for compliance with the SSH authentication and channel set-up specifications [233] [234].

For TLS decrypt analysis, the discovered cryptographic artefact groups are used to decrypt Application Data messages. De-

**Algorithm 8.1:** ChaCha20 Memory Analysis

```
Data: Extracts folder, entropy threshold
Result: Z = candidate artefacts
for extract in folder do
    i = 0;
    while not extract EOF do
        i := locate 'expand 32-byte k' in extract;
        if i > 0 then
            if entropy extract[i+16:i+48] > threshold then
                Z += key, nonce, and counter;
                i += 64;
            else
                i += 16;
            end
        end
    end
end
```

crypt compliance with the HTTP 1.1 specification [249] identifies the correct cryptographic artefacts. Once artefacts are found, encrypted payloads are ingested and decrypted, and the nonce incremented for each new message.

For both SSH and TLS 1.2, valid decrypts are written to file for user inspection. The decrypt analysis processes used to discover the valid candidate groups for SSH and TLS are shown in Algorithm 8.2.

## 8.5 Evaluation

Separate experiments were carried out to assess ChaCha20 extension effectiveness and performance with SSH and TLS 1.2 network sessions. For SSH, the extension was evaluated by performing a series of experiments with variable file sizes and operating systems, and, for TLS 1.2, by performing a series of experiments with different operating systems. The experimental set-up is de-

**Algorithm 8.2:** ChaCha20 Cryptographic Artefact Discovery

```
Data: candidate artefact sets
Result: Y = valid key, Z = valid nonce
if protocol = 'SSH' then
    for key, extract nonce in retained keys, nonces do
        delta = (extract application data packet – 1st application
         data packet);
        new nonce = extract nonce – delta;
        possible length = decrypt 1st 4 bytes with key, new nonce,
         and counter=0;
        if equation 2 valid then
            SSHlengthcryptoFound = True;
            if SSHlengthcryptoFound then
                plaintext = decrypt (payload, key, new nonce,
                 counter=1);
                if validate(plaintext) then
                    Y ⟵ key1;
                    Z ⟵ nonce;
                    break;
                end
            end
        end
    end
end
if protocol = 'TLS' then
    for key, extract nonce in retained keys, nonces do
        delta = (extract application data packet – 1st application
         data packet);
        new nonce = extract nonce – delta;
        plaintext = decrypt (payload, key, new nonce, counter=1);
        if validate(plaintext) then
            Y ⟵ key;
            Z ⟵ nonce1;
            break;
        end
    end
end
```

scribed briefly followed by a presentation of results for Windows clients and Ubuntu servers.

### 8.5.1 Experimental Set-up

The base MemDecrypt physical test environment was supplemented with additional software to support secure protocol communications. Experiments were executed on the Windows client, which connected to the Ubuntu server. For SSH, the PuTTY suite [235] provided SSH client functionality and *openssh-server* server functionality. For TLS 1.2, OpenSSL 1.1.0g provided both client and server functionality.

Similar to AES experiments, for SSH evaluation, the PuTTY *pscp* program was executed from the Windows command line using requests of the form:

*pscp -P nnnn 'filename' name@ipaddress:/home/name*

where *nnnn* is the target port, *'filename'* is the file being transmitted, *name* is a user on the target Ubuntu server, *ipaddress* is the target server IP address, and */home/name* is the Ubuntu server target folder for the uploaded file. An Ubuntu service was started from the command line to listen for client SSH requests with commands of the form:

*/usr/sbin/sshd -f /root/sshd_ config -d -p nnnn*

where *nnnn* is the listening port number and *sshd_ config* contains configuration details. To enforce ChaCha20, *sshd_ config* contained the line *ciphers chacha20-poly1305@openssh.com*

For TLS client and server experiments, OpenSSL command-line utilities were executed from the command line. The OpenSSL

server emulated a web server with the command:

*openssl s_ server -accept 443 -cert crt.pem -key key.pem -WWW*

where *crt.pem* and *key.pem* are server certificate and private key file, respectively. The client connected to the OpenSSL server with a command of the form:

*openssl s_ client -cipher CIPHER -connect a.b.c.d:443*

where *CIPHER* stipulates the algorithms for encryption, key exchange, and authentication, e.g. ECDHE-RSA-CHACHA20-POLY1305, and *a.b.c.d*, the OpenSSL server IP address. User input entered on the client command-line simulated browser requests, such as: *GET / HTTP/1.1*, *Host: a.b.c.d*, *Accept-Encoding: gzip, deflate*, *Accept: */*.

### 8.5.2 Experimental Results

ChaCha20 base structures were found in client and server volatile memory for SSH and TLS. In SSH experiments, four base structures were discovered in both Windows and Ubuntu application memory. An example of two such structures is illustrated in the highlighted section of Figure 8-1.

In TLS experiments, one base structure was discovered for ECDHE-RSA-CHACHA20-POLY1305. An example of this structure is illustrated in the highlighted section of Figure 8-2.

The base structure discoveries led to key stream generation and rapid decryption. Complete SSH sessions including user names, passwords, file names, and uploaded file contents were obtained. Only outgoing TLS traffic was decrypted. This difference between SSH and TLS results relates to their respective implementations.

PuTTY/OpenSSH ChaCha20 data structures are heap-resident


Figure 8-1: ChaCha20 SSH Base Structure in Memory

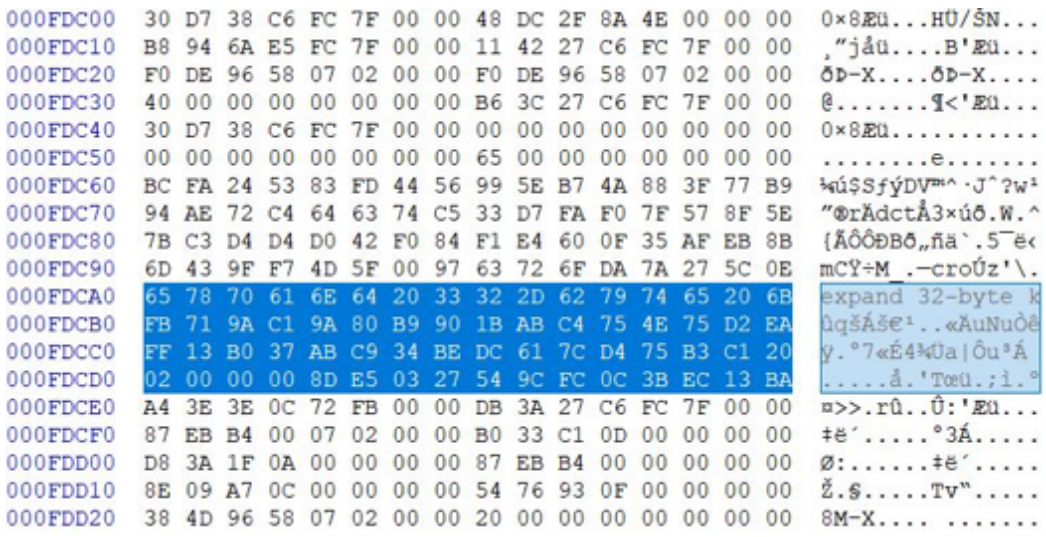



Figure 8-2: ChaCha20 TLS Base structure in Memory

so memory extracts for successful decryption are not necessarily linked to client SSH message transmission. By contrast, OpenSSL ChaCha20 base structures are stack-resident, so ephemeral, and may therefore be overwritten. Consequently, full SSH sessions

were decrypted whereas only outgoing TLS sessions were decrypted. The use of additional triggers for TLS memory extraction may lead to full TLS session decryption.

A sample decrypt for a complete SSH session is shown in Figure 8-3. Example TLS analysis component logs, including an Application Data Message decrypt, are shown in Figure 8-4.


Figure 8-3: ChaCha20 SSH Decrypt Output Example

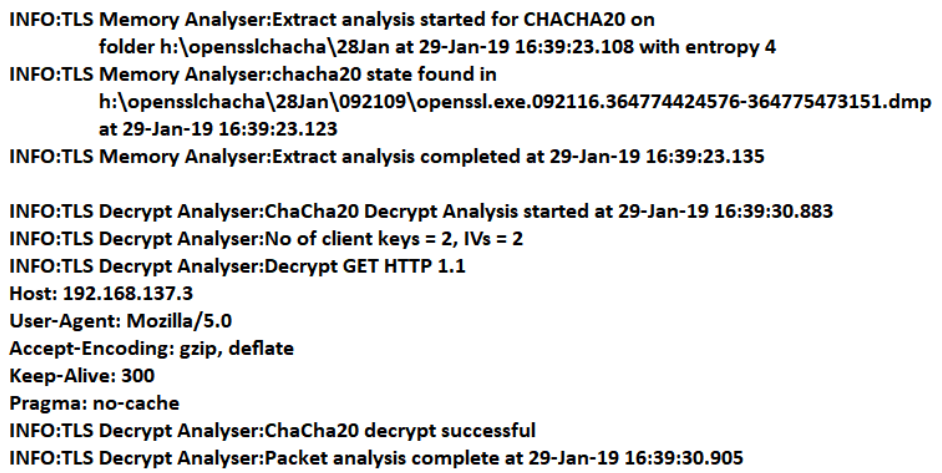



Figure 8-4: ChaCha20 TLS Memory & Decrypt Analysis Logs

ChaCha20 memory and decrypt analysis components completed rapidly. Memory analysis durations were less than 0.5 (SSH) and 0.1 (TLS) seconds as shown in Table 8-1. Although memory analysis was independent of file size, decrypt analysis durations were

roughly proportionate to the volume of encrypted traffic, which for SSH file transfer is a consequence of uploaded file size. The SSH analysis duration results are illustrated in the graph shown in Figure 8-5.

Table 8-1: ChaCha SSH & TLS Memory Analysis Durations

| | SSH | TLS |
|---|---|---|
| Maximum | 0.407 | 0.027 |
| Minimum | 0.007 | 0.011 |
| Mean | 0.144 | 0.021 |
| Standard Deviation | 0.153 | 0.006 |

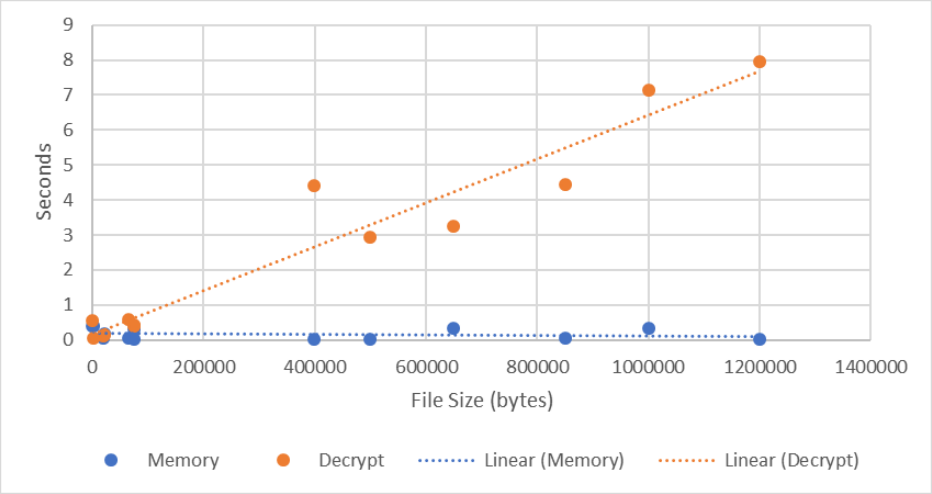



Figure 8-5: ChaCha20 SSH Analysis Durations vs File Size

## 8.6 Conclusions

Implementations of ChaCha20-Poly1305 encryption using commonly used applications and libraries for SSH and TLS communications are vulnerable to being decrypted with a single memory extract. Secure communications in other environments where ChaCha20 is used for encryption, such as Android smartphones [99], may also be vulnerable when memory is acquired.

A possible limitation occurs if implementations separate the constant string and the remainder of the ChaCha20 array. This can be achieved by copying the constant string segments to registers and delaying array assembly until required for encryption or decryption by encrypting the constant string, or randomly segmenting it. A more effective approach may be assembling the base structure on the stack, as for OpenSSL, and then ensuring stack contents are cleared immediately after the encryption process. Cryptographic artefacts can still be discovered by searching for high-entropy segments but the process is over 4 minutes slower because of the increased candidate artefact set sizes.

The ChaCha20 extension decrypts SSH and TLS protocol traffic with high probability and with durations under 1 second, so is able to decrypt software-initiated sessions. As with other MemDecrypt extensions, identification of small sets of cryptographic artefacts may allow for the artefacts to be retained along with network sessions for later decryption. This may benefit entities, such as cloud vendors, where it may be illegal to decrypt communications without sanction.

# Chapter 9

# Conclusions and Future Work

Increased use of encrypted tunnels for malicious activity motivated this research. This led to research questions being proposed regarding the decryption of secure communications. The questions encompassed reviewing the effectiveness of existing decryption methods; whether new decryption methods could be developed; and whether the new methods could usefully decrypt the secure communications of live virtual machines. The questions can be answered by reflecting on key conclusions of the investigations, the achievement of the aim and objectives, contributions made by the thesis, and potential future work based on the investigations and the constructed framework.

## 9.1 Key Conclusions

The primary thesis conclusion is that commonly-used encryption algorithm implementations leave cryptographic artefact traces in memory. The thesis developed a framework which applied novel approaches to determine small sets of candidate initialisation vectors or other cryptographic data for different protocols and encryption algorithms. This enabled small sets of candidate encryption keys to be identified. With small artefact set sizes confidential data exchanges were rapidly decrypted.

Rapid decryption enables the details of securely communicated malicious activity to be obtained during network sessions which may be of assistant to researchers and analysts. The detailed key conclusions are:

- Insider attackers can use SSH protocol implementations to extract confidential data. The MemDecrypt framework decrypts SSH secure file copy sessions for AES and ChaCha20 with different key lengths, and uploaded file sizes across Windows and Ubuntu operating systems in less than 16 seconds.
- For TLS version 1.2, MemDecrypt decrypts HTTP-over-TLS sessions that use a common open-source library for AES and ChaCha20 with different key sizes, Windows operating systems, as well as Ubuntu in most scenarios in less than 12 seconds. MemDecrypt also decrypts HTTP-over-TLS traffic where client and server negotiate TLS version 1.3 in less than 13 seconds.
- Windows malware clients, such as bots and ransomware, increasingly use TLS channels to beacon with controllers. Although process hollowing makes the TLS protocol approach more difficult, implementation traces enable the cryptographic artefacts to be discovered and communications to be decrypted. Outgoing botnet and ransomware traffic is decrypted in under 1 second.

## 9.2 Achievement of Aim and Objectives

The thesis aim was to deliver a framework that enabled decrypting the secure communications of live virtual machines. The implemented MemDecrypt framework has been applied to decrypting SSH and TLS secure sessions for different implementations, encryption algorithms, and operating systems. The thesis objectives

that supported this aim are considered briefly:

- The review of cryptographic mechanisms used in secure protocols identified AES and ChaCha20 to be algorithms of most interest.
- The survey of methods for discovering cryptographic artefacts in memory found that only encryption keys were discovered. Encrypting with the commonly used AES-CTR and ChaCha20 algorithms requires IVs so plaintexts were not obtained.
- The review of existing methodologies for accessing live virtual machine network traffic and memory established that virtual machine monitors can intercept and analyse packets travelling across the virtual network bridge and obtain semantically useful memory extracts.
- A framework was designed to decrypt encrypted network traffic. To enable the framework to be extensible to other technologies, protocols, and operating systems, the framework design has three interlocking components: network traffic and read/write memory capture; memory analysis to discover small sets of candidate cryptographic artefacts; and decrypt verification.
- A base framework was implemented on the Xen Project hypervisor using Python packages and bespoke software to support the decryption of network traffic emanating from Windows clients and Ubuntu servers.
- A framework extension was implemented to investigate the SSH protocol and the SFTP sub-system, In the experiments, complete sessions for all variations were decrypted with greater than 99.99% certainty.

- The framework was extended for TLS versions 1.2 and 1.3. Although experiments with OpenSSL decrypted HTTPS traffic with greater than 99.99% certainty in less than fifteen seconds, the analysis exceeded thirty minutes for malware clients. An additional extension was constructed for clients using Windows encryption libraries. Experiments with real malware clients successfully decrypted malware client communications in less than one second.

The final section of the current chapter proposes future work using the framework.

## 9.3 Key Contributions

The principal thesis contributions result from the construction of a framework which incorporates algorithms for extraction, memory analysis, and decrypt validation. The contributions are summarised in the following items:

- Development of a framework to access virtual machine resources, identify small sets of candidate cryptographic artefacts, and decrypt encrypted sessions efficiently with minimal impact on the target device. The framework can be extended to other target environments, applications, protocols, and algorithms.
- Determined that MemDecrypt can rapidly decrypt SSH communications from Windows client and Ubuntu server virtual machines. As SSH has been used as a medium for data exfiltration [5], the framework can assist in defending against such attacks, which are difficult to detect and increasingly common [68].
- Determined that MemDecrypt rapidly decrypts TLS communications from Ubuntu server and Windows client virtual ma-

chines. HTTPS is commonly used for benign on-line transactions between clients and web servers and malicious communications, such as malware applications communicating with external controllers [9]. Rapid decryption of HTTPS traffic can determine the nature of the malicious activity and assist in developing countermeasures.

- Constructed an extension to decrypt communications from Windows client malware, such as bots or ransomware. Specific features of Windows cryptographic library data structures enabled communications to be decrypted even when malware hijacked a benign application. This can benefit analysts concerned with tracing malware activity and security application providers interested in intercepting and preventing such breaches.

- Developed and proved that TLS and SSH secure communications using the ChaCha20 stream encryption algorithm are decryptable. Two common ChaCha20 implementations leave memory traces enabling rapid decryption. This knowledge may enable developers to adopt measures to protect ChaCha20 cryptographic artefacts.

## 9.4 Future Work

Future work is proposed to address gaps in these investigations and suggest potential research areas where the framework might be used.

### 9.4.1 Investigative Gaps

This thesis focused on constructing a framework to support the decryption of secure communications protocols in virtualised en-

vironments. Choices were made regarding the protocols, protocol applications, and encryption algorithms to be investigated.

Although SSH and TLS are extensively used, and are known to conceal malicious activities, other protocols could be usefully investigated. For example, there are alternative tunnelling protocols, such as IPSec, L2TP, PPTP, which may be used individually or together, as well as application protocols, such as OpenVPN, which is layered on TLS.

For each analysed protocol, commonly used applications were chosen for experiments. Although these implementations were found to be vulnerable, SSH and TLS capabilities are offered through a wide range of alternatives. Additional experiments can establish whether cryptographic artefacts traces can be discovered in the memory of these applications. As discovered in SSH experiments, pre-testing with alternative applications can also dramatically reduce analysis durations.

AES-CTR/AES-GCM and ChaCha20 appear to be the encryption algorithms of most current interest for secure communications. The only other algorithm currently permitted in TLS 1.3 is AES-CCM (Counter with Cipher Block Chaining - Message Authentication Code), which uses counter mode for encryption and block chaining in authentication [283] and so is expected to decryptable. SSH is presently not as restricted so further options could be analysed.

### 9.4.2 Potential Research Areas

The framework can assist in preventing traces of cryptographic artefacts being discovered in memory. A defined methodology for the construction of secure communications applications and libraries may be a useful research area. Such a methodology might include compiler or operating system considerations as well as application construction steps. In this scenario, the framework could

serve as a test harness to identify vulnerabilities before application release.

MemDecrypt can also provide a basis for improving defences against malicious activity. As the impact of such activity may be short-lived, high performance is essential. Implementation in a low-level compiled language such as C, faster memory extraction, pipelining between components, and multi-threading are all performance enhancement measures.

Framework utility is limited by being implemented on Xen. Investigating the practicality of migrating the framework to other hypervisors, such as ESXi, Hyper-V, and KVM/QEMU can improve defences for a wider population. Assessing MemDecrypt ability to function successfully with moving virtual machines may also be beneficial.

Although the thesis focus has been on virtualised environments, the framework can apply where memory can be acquired such as the volatile memory of Android smartphones. As they use Linux, memory acquisition tools such as the Linux Memory Extractor ('LiME') application [70] may suffice. However, LiME depends on specific Linux version compiled kernel modules, smartphone support, and kernel level execution. These requirements are challenging but alternatives such as AMExtractor [71], which requires kernel execution privilege but no compilation, are less restrictive. TrustDump [72] may also be appropriate but minimal testing has been carried out. Commercial tools, such as Cellebrite, claim to extract memory from Android devices without any target modification although scenarios for usage are restricted [73].

Internet of Things (IoT) devices also commonly run Linux [284]. For IoT, variations in device type and Linux versions pose potentially greater challenges than smartphones. Nevertheless, solutions that support live acquisition from Android smartwatches, as well as smartphones, have been proposed [285]. IoT device memory may also be acquired by flashing memory, running Linux

dump commands, or accessing device circuitry [286]. Furthermore, memory access with commercial tools, such as Cellebrite UFED Physical Analyzer, has also been demonstrated [287]. As IoT devices frequently communicate with cloud-based servers, memory acquisition of virtualised machines may present an easier option [284].

MemDecrypt may offer significant opportunities in defending against malware activity. Discovering the plaintext of encrypted malware traffic can support the development of tools to parse the traffic and generate responses that hoodwink the attacker. The framework could also support more powerful client defences. For instance, if memory is acquired from a suspect application running in a client anti-virus sandbox, plaintext knowledge may assist in terminating the application before it causes localised damage.

# Bibliography

[1] H. Chen, C. E. Beaudoin, and T. Hong, "Securing online privacy: An empirical test on internet scam victimization, online privacy concerns, and privacy protection behaviors," *Computers in Human Behavior*, vol. 70, pp. 291–302, 2017.

[2] K. Finley, "Half the Web Is Now Encrypted. That Makes Everyone Safer," 2017, [Online; accessed 15-Jan-2018]. [Online]. Available: https://www.wired.com/2017/01/half-web-now-encrypted-makes-everyone-safer/

[3] Google, "HTTPS encryption on the web," 2018, [Online; accessed 15-Dec-2018]. [Online]. Available: https://transparencyreport.google.com/https/overview?hl=en

[4] P. Chen, L. Desmet, C. Huygens, P. Chen, L. Desmet, C. Huygens, A. Study, A. Persistent, and T. Bart, "A Study on Advanced Persistent Threats," in *IFIP International Conference on Communications and Multimedia Security*. Springer Berlin Heidelberg, 2014, pp. 63–72.

[5] A. Duncan, S. Creese, and M. Goldsmith, "An overview of insider attacks in cloud computing," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 12, pp. 2964–2981, 2015.

[6] P. Black, I. Gondal, and R. Layton, "A survey of similarities in banking malware behaviours," *Computers & Security*, vol. 77, pp. 756–772, 2018.

[7] S. Khandelwal, "How Dutch Police Decrypted BlackBerry PGP Messages For Criminal Investigation," 2017, [Online; accessed 18-Nov-2018]. [Online]. Available: https://thehackernews.com/2017/03/decrypt-pgp-encryption.html

[8] M. Collins, "Common sense guide to mitigating insider threats," Carnegie-Mellon University Pittsburgh, Tech. Rep., 2016.

[9] B. Anderson, S. Paul, and D. McGrew, "Deciphering Malware's use of TLS (without Decryption)," *Journal of Computer Virology and Hacking Techniques*, pp. 1–17, 2016.

[10] P. McLaren, G. Russell, and B. Buchanan, "Mining Malware Command and Control Traces," in *2017 Computing Conference*. IEEE, 2017, pp. 788–794.

[11] D. Gooley, "The Rise in SSL-based Threats," 2017, [Online; accessed 5-Jul-2018]. [Online]. Available: https://www.zscaler.com/blogs/research/rise-ssl-based-threats-1

[12] P. Velan, M. Čermák, P. Čeleda, and M. Drašar, "A survey of methods for encrypted traffic classification and analysis," *International Journal of Network Management*, vol. 25, no. 5, pp. 355–374, 2015.

[13] Z. B. Celik, R. J. Walls, P. McDaniel, and A. Swami, "Malware Traffic Detection using Tamper Resistant Features," in *MILCOM 2015-2015 IEEE Military Communications Conference*. IEEE, 2015, pp. 330–335.

[14] N. Moustafa and J. Slay, "The Significant Features of the UNSW-NB15 and the KDD99 Data Sets for Network Intrusion Detection Systems," in *2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2015, pp. 25–31.

[15] A. Kerckhoffs, "La cryptographie militaire," *Journal des Sciences Militaires*, vol. 9, pp. 5–38, 1883.

[16] R. Verdult, "The ( in ) security of proprietary cryptography," Ph.D. dissertation, Radboud University Nijmegen, 2015.

[17] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography engineering: design principles and practical applications.* John Wiley & Sons, 2010.

[18] Electronic Frontier Foundation, *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly Media, 1998.

[19] The Electronic Frontier Foundation, "DES Challenge III Broken in Record 22 Hours," 1999, [Online; accessed 17-Oct-2018]. [Online]. Available: https://www.eff.org/effector/12/1

[20] Federal Register, "Announcing Approval of the Withdrawal of Federal Information Processing Standard (FIPS) 46-3, Data Encryption Standard (DES); FIPS 74, Guidelines for Implementing and Using the NBS Data Encryption Standard; and FIPS 81, DES Modes of Operation," Federal Register, Tech. Rep., 2005. [Online]. Available: https://www.federalregister.gov/documents/2005/05/19/05-9945/announcing-approval-of-the-withdrawal-of-federal-information-processing-standard-fips-46-3-data

[21] M. Matsui, "Linear cryptanalysis method for DES cipher," in *Advances in Cryptology – EUROCRYPT '93*, vol. 765 of Lecture Notes in Computer Science. Springer, 1994, pp. 386–397.

[22] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digital Investigation*, vol. 1, no. 1, pp. 50–60, 2004.

[23] E. Hess, N. Janssen, B. Meyer, and T. Schütze, "Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures:a Survey," in *EUROSMART Security Conference*, vol. 130, 2000, pp. 55–64.

[24] B. Schatz, "BodySnatcher : Towards Reliable Volatile Memory Acquisition by Software By BodySnatcher : Towards reliable volatile memory acquisition by software," in *DFRWS 2007 USA Proceedings*. Elsevier, 2007, pp. 127–134.

[25] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digital Investigation*, vol. 8, no. 1, pp. 3–22, 2011.

[26] S. Vömel and F. C. Freiling, "Correctness, atomicity, and integrity: Defining criteria for forensically-sound memory acquisition," *Digital Investigation*, vol. 9, no. 2, pp. 125–137, 2012.

[27] M. Gruhn and F. C. Freiling, "Evaluating atomicity , and integrity of correct memory acquisition methods," *Digital Investigation*, vol. 16, pp. S1–S10, 2016.

[28] F. Pagani, O. Fedorov, and D. Balzarotti, "Introducing the Temporal Dimension to Memory Forensics," *ACM Trans. Priv. Secur.*, vol. 22, no. 2, pp. 9:1–9:21, Mar. 2019.

[29] F. Freiling, T. Groß, T. Latzo, T. Müller, and R. Palutke, "Advances in Forensic Data Acquisition," *IEEE Design & Test*, vol. 35, no. 5, pp. 63–74, 2018.

[30] T. Vidas, "Volatile Memory Acquisition via Warm Boot Memory Survivability," in *2010 43rd Hawaii International Conference on System Sciences*. IEEE, 2010, pp. 1–6.

[31] L. Zhang, L. Wang, R. Zhang, S. Zhang, and Y. Zhou, "Live Memory Acquisition through FireWire," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, vol. 56, pp. 159–167, 2011.

[32] A. Froomkin, "The Metaphor Is the Key: Cryptography, the Clipper Chip, and the Constitution," *University of Pennsylvania Law Review*, vol. 143, no. 3, pp. 709–897, 1995.

[33] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[34] M. Russinovich and A. Richards, "ProcDump v9.0," 2017, [Online; accessed 10-Feb-2019]. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/procdump

[35] W. J. Liu, "Process Hacker," 2016, [Online; accessed 10-Feb-2019]. [Online]. Available: https://processhacker.sourceforge.io/

[36] AccessData, "FTK Imager," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: http://marketing.accessdata.com/ftkimager4.2.0

[37] Comae, "Comae Toolkit," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: https://my.comae.io/tools

[38] FireEye, "Memoryze," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: https://www.fireeye.com/services/freeware.html

[39] M. Cohen, "WinPMEM," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: https://github.com/google/rekall/tree/master/tools/windows/winpmem

[40] K. Nance, M. Bishop, and B. Hay, "Virtual Machine Introspection: Observation or Interference?" *IEEE Security & Privacy*, vol. 6, no. 5, pp. 32–37, 2008.

[41] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 386–393.

[42] R. Iphofen, "Safety is more important than privacy," Times Higher Education," https://www.timeshighereducation.com/features/safety-is-more-important-than-privacy, 2016, [Online; accessed 29-Jan-2019].

[43] G. R. Lucas Jr, "Privacy, Anonymity, and Cyber Security," *Amsterdam Law Forum*, vol. 5, p. 107, 2013.

[44] Z. Bauman, D. Bigo, P. Esteves, E. Guild, V. Jabri, D. Lyon, and R. B. Walker, "After Snowden: Rethinking the Impact of Surveillance," *International Political Sociology*, vol. 8, no. 2, pp. 121–144, 2014.

[45] W. B. Tesfay, P. Hofmann, T. Nakamura, S. Kiyomoto, and J. Serna, "PrivacyGuide: towards an Implementation of the EU GDPR on Internet Privacy Policy Evaluation," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*. ACM, 2018, pp. 15–21.

[46] A. Beech, L. Yaxley, and G. Roberts, "Facebook, Google obliged to decrypt online messages

to help Government fight terrorism," 2017, [Online; accessed 10-Nov-2018]. [Online]. Available: http://www.abc.net.au/news/2017-07-14/facebook-google-to-be-forced-to-decrypt-messages-fight-terrorism/8707748

[47] R. Iphofen, "Ethical Issues in Surveillance and Privacy," in *Hostile Intent and Counter-Terrorism*. CRC Press, 2017, pp. 59–71.

[48] D. Kahn, *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996. [Online]. Available: https://books.google.com.au/books?id=SEH_rHkgaogC

[49] H. O. Yardley, *The American Black Chamber*. Naval Institute Press, 2013.

[50] W. C. Banks, "Cyber espionage and electronic surveillance: Beyond the media coverage," *Emory Law Journal*, vol. 66, p. 513, 2016.

[51] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta *et al.*, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 5–17.

[52] J. Ball, "NSA monitored calls of 35 world leaders after US official handed over contacts," pp. 2–4, 2013, [Online; accessed 17-Sep-2018]. [Online]. Available: http://www.theguardian.com/world/2013/oct/24/nsa-surveillance-world-leaders-calls

[53] UN General Assembly, "Universal Declaration of Human Rights," *UN General Assembly*, 1948.

[54] M. Cayford and W. Pieters, "The effectiveness of surveillance technology: What intelligence officials are saying," *The Information Society*, vol. 34, no. 2, pp. 88–103, 2018. [Online]. Available: https://doi.org/10.1080/01972243.2017.1414721

[55] R. Graham, "How Terrorists Use Encryption," *CTC Sentinel*, vol. 9, no. 6, pp. 20–25, 2016. [Online]. Available: www.ctc.usma.edu/sentinel/

[56] J. McLaughlin, "Apple Faces 12 Other Requests to Break Into iPhones," 2016, [Online; accessed 10-Feb-2019]. [Online]. Available: https://theintercept.com/2016/02/23/new-court-filing-reveals-apple-faces-12-other-requests-to-break-into-locked-iphones/

[57] R. Crozier, "Govt's decryption bill can only lead to 'systemic weaknesses' - Security - iTnews," sep 2018, [Online; accessed 10-Nov-2018]. [Online]. Available: https://www.itnews.com.au/news/govts-decryption-bill-can-only-lead-to-systemic-weaknesses-512390

[58] Zscaler, "Zscaler ThreatLabZ Reveals Malicious Content Delivered Over SSL/TLS Has More Than Doubled in Six Months," 2017, [Online; accessed 29-Jan-2019]. [Online]. Available: https://www.zscaler.com/press/zscaler-threatlabz-reveals-malicious-content-delivered-over-ssltls-has-more-doubled-six-months

[59] S. Barker, "2018 sees surge in encrypted attacks, malware & ransomware," 2018, [Online; accessed 6-Sep-2018]. [Online]. Available: https://securitybrief.com.au/amp/story/2018-sees-surge-encrypted-attacks-malware-ransomware/

[60] T. Morgenstern, "Watch Out for These Two Data Exfiltration Channels," 2017, [Online; accessed 8-Jul-

2018]. [Online]. Available: https://www.cyberbit.com/blog/endpoint-security/data-exfiltration-channels/

[61] C. Rossow and C. J. Dietrich, "Provex: Detecting Botnets with Encrypted Command and Control Channels," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 7967 of Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2013, pp. 21–40.

[62] C. Macropoulos and K. M. Martin, "Balancing Privacy and Surveillance in the Cloud," *IEEE Cloud Computing*, vol. 2, no. 4, pp. 14–21, 2015.

[63] N. Sullivan, "Introducing TLS 1.3," 2016, [Online; accessed 3-Sep-2018]. [Online]. Available: https://blog.cloudflare.com/introducing-tls-1-3/

[64] A. Brodie, "Overview of TLS v1.3," 2018, [Online; accessed 10-Dec-2018]. [Online]. Available: https://www.owasp.org/images/9/91/OWASPLondon20180125_TLSv1.3_Andy_Brodie.pdf

[65] P. G. Sarkar and S. Fitzgerald, "Attacks on SSL a comprehensive study of BEAST, CRIME, TIME, BREACH, LUCKY13 & RC4 biases," *Internet: https://www. isecpartners. com/media/106031/ssl_attacks_survey. pdf [June, 2014]*, 2013.

[66] Y. Sheffer, R. Holz, and P. Saint-Andre, "RFC 7457 - Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)," *Internet Engineering Task Force*, 2015.

[67] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, *Report on Post-*

*Quantum Cryptography.* National Institute of Standards and Technology, 2016.

[68] S. Aditham and N. Ranganathan, "A System Architecture for the Detection of Insider Attacks in Big Data Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 974–987, 2018.

[69] N. Zhang, R. Zhang, K. Sun, W. Lou, Y. T. Hou, and S. Jajodia, "Memory Forensic Challenges Under Misused Architectural Features," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 9, pp. 2345–2358, 2018.

[70] J. Sylve, "LiME - Linux Memory Extractor," https://github.com/504ensicslabs/lime, 2019, [Online; accessed 29-Jan-2019].

[71] H. Yang, J. Zhuge, H. Liu, and W. Liu, "A Tool for Volatile Memory Acquisition from Android Devices," in *IFIP International Conference on Digital Forensics*, vol. 484. Springer,Cham, 2016, pp. 365–378.

[72] H. Sun, K. Sun, Y. Wang, and J. Jing, "Reliable and Trustworthy Memory Acquisition on Smartphones," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2547–2561, 2015.

[73] "Cellebrite", "Advanced Extraction Service," https://www.cellebrite.com/en/services/advanced-extraction-services, 2018, [Online; accessed 29-Jan-2019].

[74] P. McLaren, G. Russell, W. J. Buchanan, and Z. Tan, "Decrypting Live SSH Traffic in Virtual Environments," *Digital Investigation*, vol. 29, pp. 109–117, 2019.

[75] B. Schneier, *Secrets & Lies.* Wiley, 2000.

[76] D. Puthal, S. Nepal, R. Ranjan, and J. Chen, "A dynamic prime number based efficient security mechanism for big sensing data streams," *Journal of Computer and System Sciences*, vol. 83, no. 1, pp. 22–42, 2017.

[77] A. Klein, *Stream Ciphers*. Springer, 2013.

[78] K. M. Martin, *Everyday Cryptography:Fundamental Principles and Applications*, 2nd ed. Oxford University Press, 2017.

[79] Y. Nir and A. Langley, "RFC 8439 - ChaCha20 and Poly1305 for IETF Protocols," *Internet Engineering Task Force*, 2018.

[80] L. R. Knudsen and M. J. Robshaw, *The Block Cipher Companion*. Springer Science & Business Media, 2011. [Online]. Available: http://link.springer.com/10.1007/978-3-642-17342-4

[81] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques," National Institute of Standards and Technology, Tech. Rep., 2001.

[82] P. Ducklin, "Anatomy of a password disaster – Adobe's giant-sized cryptographic blunder," 2013, [Online; accessed 18-Nov-2018]. [Online]. Available: https://nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic-blunder/

[83] K. Bhargavan and G. Leurent, "On the Practical (In-) Security of 64-bit Block Ciphers Collision Attacks on HTTP over TLS and OpenVPN," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 456–467. [Online]. Available: https://eprint.iacr.org/2016/798{%}0Ahttps://eprint.iacr.org/2016/798.pdf

[84] C. Kowalczyk, "Modes of Block Ciphers," 2017, [Online; accessed 15-Jan-2019]. [Online]. Available: http://www.crypto-it.net/eng/theory/modes-of-block-ciphers.html

[85] D. A. McGrew and J. Viega, "The Security and Performance of the Galois/Counter Mode (GCM) of Operation," in *International Conference on Cryptology in India*. Springer, 2004, pp. 343–355. [Online]. Available: http://link.springer.com/10.1007/978-3-540-30556-9_27

[86] P. Rogaway, "Evaluation of Some Blockcipher Modes of Operation," *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011.

[87] E. Biham and A. Shamir, *Differential cryptanalysis of the Data Encryption Standard*. Springer Science & Business Media, 2012. [Online]. Available: http://link.springer.com/content/pdf/10.1007/978-1-4613-9314-6.pdf

[88] NIST, "Announcing the Advanced Encryption Standard (AES)," NIST, Tech. Rep., 2001.

[89] J. Daeman and V. Rijmen, *The Design of Rijndael*. Springer-Verlag Berlin Heidelberg GmbH, 2002.

[90] W. J. Buchanan, S. Li, and R. Asif, "Lightweight cryptography methods," *Journal of Cyber Security Technology*, vol. 1, no. 3-4, pp. 187–201, 2018. [Online]. Available: https://doi.org/10.1080/23742917.2017.1384917

[91] J. Daemen and V. Rijmen, "The Rijndael Block Cipher: AES Proposal," NIST, Tech. Rep., 2003.

[92] A. Biryukov, "Block Ciphers and Stream Ciphers: The State of the Art," *IACR Cryptology ePrint Archive*, vol. 2004/094, 2004.

[93] C. Manifavas, G. Hatzivasilis, K. Fysarakis, and Y. Papaefstathiou, "A survey of lightweight stream ciphers for embedded systems," *Security and Communication Networks*, vol. 9, no. 10, pp. 1226–1246, 2016.

[94] G. Paul and S. Maitra, "Permutation After RC4 Key Scheduling Reveals the Secret Key," in *International Workshop on Selected Areas in Cryptography*, vol. 4876 of Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2007, pp. 360–377.

[95] A. Popov, "RFC 7465 - Prohibiting RC4 Cipher Suites," *Internet Engineering Task Force*, 2015.

[96] L. Camara, "Deprecating RC4 in Secure Shell (SSH)," https://tools.ietf.org/id/draft-ietf-curdle-rc4-die-die-die-10.html, 2018, [Online; accessed 29-Jan-2019].

[97] OpenBSD, "OpenSSH," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: https://www.openssh.com/

[98] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)," NIST, Tech. Rep., 2016.

[99] Ianix, "ChaCha Usage and Deployment," https://ianix.com/pub/chacha-deployment.html, 2019, [Online; accessed 29-Jan-2019].

[100] D. J. Bernstein, "The Salsa20 Family of Stream Ciphers," in *New Stream Cipher Designs*. Springer,Berlin, Heidelberg, 2008, vol. 4986 of Lecture Notes in Computer Science, pp. 84–97.

[101] ——, "ChaCha, a variant of Salsa20," in *SASC: The State of the Art of Stream Ciphers*, vol. 8, 2008, pp. 3–5.

[102] KDDI Research Inc., "Security Analysis of ChaCha20-Poly1305 AEAD," https://www.cryptrec.go.jp/exreport/cryptrec-ex-2601-2016.pdf, 2017, [Online; accessed 29-Jan-2019].

[103] D. J. Bernstein, "Extending the Salsa20 nonce," in *Symmetric Key Encryption Workshop 2011*, 2011.

[104] P. Crowley and E. Biggers, "Adiantum: length-preserving encryption for entry-level processors," *IACR Transactions on Symmetric Cryptology*, pp. 39–61, 2018.

[105] D. Desai, "What's hiding in encrypted traffic? Millions of advanced threats," 2019, [Online; accessed 11-Feb-2019]. [Online]. Available: https://www.zscaler.com/blogs/research/whats-hiding-encrypted-traffic-millions-advanced-threats

[106] A. K. Khan and H. J. Mahanta, "Side Channel Attacks and Their Mitigation Techniques," in *1st International Conference on Automation, Control, Energy and Systems - 2014, ACES 2014*. IEEE, 2014, pp. 1–4.

[107] C. Giraud, "DFA on AES," in *Advanced Encryption Standard – AES*, vol. 3373 of Lecture Notes in Computer Science. Springer,Berlin,Heidelberg, 2004, pp. 27–41.

[108] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault," *IFIP International Workshop on Information Security Theory and Practices*, pp. 224–233, 2011.

[109] J. Takahashi and T. Fukunaga, "Differential Fault Analysis on AES with 192 and 256-Bit Keys," *IACR Eprint archive*, pp. 7–12, 2010. [Online]. Available: https://eprint.iacr.org/2010/023.pdf

[110] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures : the Case of AES," in *Topics in Cryptology – CT-RSA 2006*, vol. 3860 of Lecture Notes in Computer Science. Springer,Berlin,Heidelberg, 2005, pp. 1–25.

[111] Y. Lyu and P. Mishra, "A Survey of Side-Channel Attacks on Caches and Countermeasures," *Journal of Hardware and Systems Security*, vol. 2, pp. 33–50, 2018. [Online]. Available: https://link.springer.com/content/pdf/10.1007{%}2Fs41635-017-0025-y.pdf

[112] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–37, 2018.

[113] O. Aciiçmez, Ç. Koç, and J.-P. Seifert, "Predicting Secret Keys Via Branch Prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[114] E. Käsper and P. Schwabe, "Faster and Timing-Attack Resistant AES-GCM," *Cryptographic Hardware and Embedded Systems-CHES 2009*, pp. 1–17, 2009. [Online]. Available: https://cryptojedi.org/papers/aesbs-20090616.pdf

[115] "Intel® Data Protection Technology with AES-NI and Secure Key," [Online; accessed 10-Sep-2018]. [Online]. Available: https://www.intel.com.au/content/www/au/en/architecture-and-technology/advanced-encryption-standard--aes-/data-protection-aes-general-technology.html

[116] P. Kocher, J. Jaffe, and B. Jun, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[117] J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards," in *Smart Card Programming and Security. E-smart 2001*, vol. 2140 of Lecture Notes in Computer Science. Springer,Berlin,Heidelberg, 2001, pp. 200–210.

[118] D. Genkin, A. Shamir, and E. Tromer, "Acoustic Cryptanalysis," *Journal of Cryptology*, vol. 30, no. 2, pp. 392–443, 2017.

[119] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *Computer Security — ESORICS 98*, vol. 1485 of Lecture Notes in Computer Science, pp. 97–110, 1998.

[120] C. Maartmann-Moe, S. E. Thorkildsen, and A. Årnes, "The persistence of memory: Forensic identification and extraction of cryptographic keys," *Digital Investigation*, vol. 6, pp. 132–140, 2009.

[121] A. Tsow, "An Improved Recovery Algorithm for Decayed AES Key Schedule Images," in *Cryptography. SAC 2009*, vol. 5867 of Lecture Notes in Computer Science. Springer,Berlin,Heidelberg, 2009, pp. 215–230.

[122] H. Riebler, T. Kenter, C. Plessl, and C. Sorge, "Reconstructing AES key schedules from decayed memory with FPGAs," in *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*. IEEE, 2014, pp. 222–229.

[123] W. Lin and D. Lee, "Traceback attacks in cloud - Pebbletrace botnet," in *Proceedings - 32nd IEEE International Conference on Distributed Computing Systems Workshops, ICDCSW 2012*, 2012, pp. 417–426.

[124] T. Klein, "All your private keys are belong to us: Extracting RSA private keys and certificates from process memory," pp. 1–7, 2006.

[125] F. Rocha and M. Correia, "Lucy in the Sky without Diamonds: Stealing Confidential Data in the Cloud," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 129–134, 2011.

[126] Y. Nakano, A. Basu, S. Kiyomoto, and Y. Miyake, "Key Extraction Attack Using Statistical Analysis of Memory Dump Data," in *International Conference on Risks and Security of Internet and Systems*. Springer, 2014, pp. 239–246.

[127] A. Shamir and N. V. Someren, "Playing 'Hide and Seek' with Stored Keys," *Financial cryptography*, pp. 1–9, 1999. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-48390-X_9

[128] B. Taubmann, C. Frädrich, D. Dusold, and H. P. Reiser, "TLSkex: Harnessing virtual machine introspection for decrypting TLS communication," *Digital Investigation*, vol. 16, pp. S114–S123, 2016.

[129] S. Sentanoe, B. Taubmann, and H. P. Reiser, "Virtual Machine Introspection Based SSH Honeypot," in *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*. ACM, 2017, pp. 13–18.

[130] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," *Proceedings of the Network and Distributed System Security Symposium (NDSS'03)*, vol. 1, pp. 253–285, 2003.

[131] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtu-

alization," in *Proceedings - IEEE Symposium on Security and Privacy*. IEEE, 2008, pp. 233–247.

[132] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: Subverting Virtual Machine Introspection for Fun and Profit," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010, pp. 82–91.

[133] J. Shi, Y. Yang, C. Li, and X. Wang, "SPEMS: A Stealthy and Practical Execution Monitoring System Based on VMI," in *Cloud Computing and Security. ICCCS 2015*, vol. 9483 of Lecture Notes in Computer Science. Springer,Cham, 2015, pp. 147–156.

[134] F. Rodríguez-Haro, F. Freitag, L. Navarro, E. Hernánchez-sánchez, N. Farías-Mendoza, J. A. Guerrero-Ibáñez, and A. González-Potes, "A summary of virtualization techniques," *Procedia Technology*, vol. 3, no. February 2014, pp. 267–272, 2012.

[135] S. Zhang, X. Meng, L. Wang, L. Xu, and X. Han, "Secure Virtualization Environment based on Advanced Memory Introspection," *Security and Communication Networks*, vol. 2018, pp. 1–16, 2018.

[136] P. Chen, C. Huygens, L. Desmet, and W. Joosen, "Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware," *IFIP Advances in Information and Communication Technology*, vol. 471, pp. 323–336, 2016.

[137] S. T. King, P. M. Chen, Y. M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, "SubVirt: Implementing malware with virtual machines," in *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2006. IEEE, 2006.

[138] Y. Oyama, "Trends of anti-analysis operations of malwares observed in API call logs," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 69–85, 2018.

[139] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1315245.1315262

[140] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Proceedings - IEEE Symposium on Security and Privacy*, vol. 1. IEEE, may 2011, pp. 297–312.

[141] E. Bauman, G. Ayoade, and Z. Lin, "A Survey on Hypervisor-Based Monitoring," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–33, 2015.

[142] Microsoft, "Windows 10 release information," 2018, [Online; accessed 5-Nov-2018]. [Online]. Available: https://www.microsoft.com/en-us/itpro/windows-10/release-information

[143] Y. Fu, J. Zeng, and Z. Lin, "HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated in-VM Management," in *2014 USENIX Annual Technical Conference*, 2014, pp. 85–96. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643644

[144] Y. Fu, Z. Lin, K. W. Hamlen, L. Khan, and B. Thuraisingham, "Bridging the Semantic Gap in Virtual Machine Introspection via Binary Code Reuse," Ph.D. dissertation, University of Texas, 2016.

[145] R. Wu, P. Chen, P. Liu, and B. Mao, "System Call Redirection: A Practical Approach to Meeting Real-world Virtual Machine Introspection Needs," in *Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*. IEEE, 2014, pp. 574–585.

[146] Volatility Foundation, "Volatility," https://www.volatilityfoundation.org/, 2016, [Online; accessed 29-Jan-2019].

[147] M. Cohen", "Rekall Memory Forensic Framework"," http://www.rekall-forensic.com/, 2017, [Online; accessed 29-Jan-2019].

[148] LibVMI, "LibVMI Project," http://libvmi.com/, 2013, [Online; accessed 29-Jan-2019].

[149] J. Hizver and T.-c. Chiueh, "Real-Time Deep Virtual Machine Introspection and Its Applications," *ACM SIGPLAN Notices*, vol. 49, no. 7, pp. 3–14, 2014.

[150] D. Zhan, L. Ye, B. Fang, X. Du, and S. Su, "CFWatcher: A Novel Target-based Real-time Approach to Monitor Critical Files using VMI," in *2016 IEEE International Conference on Communications, ICC 2016*. IEEE, 2016, pp. 1–6.

[151] A. Cohen and N. Nissim, "Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory," *Expert Systems with Applications*, vol. 102, pp. 158–178, 2018.

[152] A. More and S. Tapaswi, "Virtual machine introspection: towards bridging the semantic gap," *Journal of Cloud Computing*, vol. 3, no. 1, pp. 1–14, 2014.

[153] M. Botacin, P. L. D. Geus, and A. Grégio, "Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms," *ACM Computing Surveys*, vol. 51, no. 4, pp. 69:1–69:34, 2018.

[154] K. Kourai and S. Chiba, "HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. ACM, 2005, pp. 197–207. [Online]. Available: http://dl.acm.org/citation.cfm?id=1065006

[155] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting Past and Present Intrusions through Vulnerability-Specific Predicates," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, p. 91, 2005.

[156] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*. ACM, 2007, pp. 116–127.

[157] B. Hay and K. Nance, "Forensics Examination of Volatile System Data Using Virtual Introspection," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 74–82, 2008.

[158] A. Lanzi, M. Sharif, and W. Lee, "K-Tracer: A System for Extracting Kernel Malware Behavior," in *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS '09)*, 2009, pp. 163–169.

[159] C. Benninger, S. W. Neville, Y. O. Yazir, C. Matthews, and Y. Coady, "Maitland: Lighter-Weight VM Introspection to Support Cyber-Security in the Cloud," in *Proceedings - 2012*

*IEEE 5th International Conference on Cloud Computing, CLOUD 2012*. IEEE, 2012, pp. 471–478.

[160] C. Harrison, D. Cook, R. McGraw, and J. A. Hamilton, "Constructing a cloud-based IDS by merging VMI with FMA," in *Proc. of the 11th IEEE Int. Conference on Trust, Security and Privacy in Computing and Communications, TrustCom-2012*. IEEE, 2012, pp. 163–169. [Online]. Available: http://ieeexplore.ieee.org/document/6295971/

[161] Y. Fu, W. C. Rd, and Z. Lin, "EXTERIOR: Using Dual-VM Based External Shell for Guest-OS Introspection, Configuration, and Recovery," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2013, pp. 97–110.

[162] C.-J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang, "NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 4, pp. 198–211, 2013.

[163] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. ACM, 2014, pp. 386–395.

[164] A. Fattori, A. Lanzi, D. Balzarotti, and E. Kirda, "Hypervisor-based malware protection with AccessMiner," *Computers & Security*, vol. 52, pp. 33–50, 2015.

[165] J. T. Saxon, B. Bordbar, and K. Harrison, "Efficient Retrieval of Key Material for Inspecting Potentially Malicious

Traffic in the Cloud," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 155–164.

[166] C. W. Tien, J. W. Liao, S. C. Chang, and S. Y. Kuo, "Memory Forensics using Virtual Machine Introspection for Malware Analysis," in *2017 IEEE Conference on Dependable and Secure Computing*. IEEE, 2017, pp. 518–519.

[167] H. Upadhyay, H. A. Gohel, A. Pons, and L. Lagos, "Windows Virtualization Architecture For Cyber Threats Detection," in *2018 1st International Conference on Data Intelligence and Security (ICDIS)*. IEEE, 2018, pp. 119–122.

[168] D. Zhan, H. Li, L. Ye, H. Zhang, B. Fang, and X. Du, "A Low-overhead Kernel Object Monitoring Approach for Virtual Machine Introspection," *arXiv preprint arXiv:1902.05135*, 2019.

[169] U. Sen, "Hidden Tear," Istanbul, 2015, [Online; accessed 8-Jan-2018]. [Online]. Available: https://github.com/goliate/hidden-tear

[170] "Tripwire," [Online; accessed 10-Jan-2018]. [Online]. Available: https://www.tripwire.com/

[171] G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen, and F. Zhao, "VMDriver: A Driver-based Monitoring Mechanism for Virtualization," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010, pp. 72–81.

[172] J. Grimm, I. Ahmed, V. Roussev, M. Bhatt, and M. Hong, "Automatic mitigation of kernel rootkits in cloud environments," in *International Workshop on Information Security Applications*. Springer, 2017, pp. 137–149.

[173] Y. Wen, J. Zhao, H. Wang, and J. Cao, "Implicit Detection of Hidden Processes with a Feather-Weight Hardware-

Assisted Virtual Machine Monitor," in *Information Security and Privacy. ACISP 2008*, vol. 5107 of Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2008, pp. 361–375.

[174] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based Hidden Process Detection and Identification using Lycosid," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE '08*. ACM, 2008, p. 91. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1346256.1346269

[175] D. N. Patil and B. B. Meshram, "Windows Physical Memory Analysis to Detect the Presence of Malicious Code," in *Recent Findings in Intelligent Computing Techniques*. Springer, 2019, pp. 3–13.

[176] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A Layered Architecture for Detecting Malicious Behaviors," in *Recent Advances in Intrusion Detection. RAID 2008*, vol. 5230 of Lecture Notes in Computer Science. Springer,Berlin,Heidelberg, 2008, pp. 78–97.

[177] C. Zheng, M. D. Preda, J. Granjal, S. Zanero, and F. Maggi, "On-Chip System Call Tracing: A Feasibility Study and Open Prototype," in *2016 IEEE Conference on Communications and Network Security, CNS 2016*. IEEE, 2017.

[178] S.-W. Hsiao, Y. S. Sun, and M. C. Chen, "Virtual Machine Introspection Based Malware Behavior Profiling and Family Grouping," *arXiv preprint arXiv*, vol. 1705.01697, pp. 1–13, 2017.

[179] S. Javaid, A. Zoranic, I. Ahmed, and G. G. Richard III, "Atomizer: Fast, Scalable and Lightweight Heap Analyzer for

Virtual Machines in a Cloud Environment," in *6th Layered Assurance Workshop*, vol. 6, 2012, p. 57.

[180] D. Tian, X. Xiong, C. Hu, and P. Liu, "Defeating buffer overflow attacks via virtualization," *Computers and Electrical Engineering*, vol. 40, no. 6, pp. 1940–1950, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.compeleceng.2013.11.032

[181] K. Leach, C. Spensky, W. Weimer, and F. Zhang, "Towards Transparent Introspection," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7476647/

[182] "Open vSwitch," 2017, [Online; accessed 8-Jan-2019]. [Online]. Available: http://openvswitch.org/

[183] "Snort," 2017, [Online; accessed 8-Jan-2019]. [Online]. Available: https://www.snort.org

[184] The Tcpdump team, "Tcpdump/Libpcap," 2018, [Online; accessed 8-Jan-2019]. [Online]. Available: http://www.tcpdump.org/

[185] F. Grehl, "ESXi Network Troubleshooting with tcpdump-uw and pktcap-uw," 2015, [Online; accessed 10-Sep-2018]. [Online]. Available: http://www.virten.net/2015/10/esxi-network-troubleshooting-with-tcpdump-uw-and-pktcap-uw/

[186] J. Shi, Y. Yang, J. He, C. Tang, and Q. Li, "Design of a comprehensive virtual machine monitoring system," in *CCIS 2014 - Proceedings of 2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems*. IEEE, 2014, pp. 510–513. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7175789

[187] R. Stephens, *Beginning software engineering.* John Wiley & Sons, 2015.

[188] C. Modi, D. Patel, B. Borisaniya, A. Patel, and M. Rajarajan, "A Survey on Security Issues and Solutions at Different Layers of Cloud Computing," *Journal of Supercomputing*, vol. 63, no. 2, pp. 561–592, 2013.

[189] T. Armerding, "The 18 biggest data breaches of the 21st century," 2018, [Online; accessed 20-Dec-2018]. [Online]. Available: https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html

[190] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in in Network Devices," in *Presented as part of the 21st USENIX Security Symposium USENIX Security 12)*, 2012, pp. 205–220.

[191] R. von Mises, "On the Foundations of Probability and Statistics," *Annals of Mathematical Statistics*, vol. 12, pp. 50–60, 1940.

[192] S. A. Terwijn, "The Mathematical Foundations of Randomness," in *The Challenge of Chance.* Springer International Publishing, 2016, pp. 49–66.

[193] J. von Neumann, "Various techniques used in connection with random digits," in *Monte Carlo Method*, A. Householder, G. Forsythe, and H. Germond, Eds., 1951.

[194] M. S. Turan, "Recommendation for the entropy sources used for random bit generation," NIST, Tech. Rep., 2018.

[195] C. Cachin, "Entropy Measures and Unconditional Security in Cryptography," Ph.D. dissertation, ETH Zurich, 1997.

[196] C. E. Shannon, "A Mathematical Theory of Communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[197] J. L. Massey, "Guessing and entropy," in *Proceedings of 1994 IEEE International Symposium on Information Theory*. IEEE, 1994, p. 204.

[198] D. Davis, "The Top 5 Enterprise Type 1 Hypervisors You Must Know," 2013, [Online; accessed 3-Sep-2018]. [Online]. Available: http://www.virtualizationsoftware.com/top-5-enterprise-type-1-hypervisors/

[199] J. Horalek, O. Marik, S. Neradova, and S. Zitta, "Virtualization tools analysis mapped into RING 0," in *Emerging eLearning Technologies and Applications (ICETA), 2014 IEEE 12th International Conference on*, vol. 2500. IEEE, 2014, pp. 151–156.

[200] B. Yenké, A. A. Abba Ari, C. Dibamou Mbeuyo, and D. A. Voundi, "Virtual Machine Performance upon Intensive Computations," *GSTF Journal on Computing*, vol. 4, no. 3, pp. 98–107, 2015.

[201] S. Nanz and C. A. Furia, "A Comparative Study of Programming Languages in Rosetta Code," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 778–788.

[202] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, Security Threats, and Solutions," *ACM Computing Surveys*, vol. 45, no. 2, pp. 1–39, 2013.

[203] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking Up is Hard to

Do : Security and Functionality in a Commodity Hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 189–202.

[204] L. YamunaDevi, P. Aruna, D. Sudha Devi, and N. Priya, "Security in Virtual Machine Live Migration for KVM," in *Proceedings of 2011 International Conference on Process Automation, Control and Computing, PACC 2011*. IEEE, 2011, pp. 1–6.

[205] F. Grehl, "VMware ESXi 3.5 - 6.7 Hypervisor Size Comparison," 2018, [Online; accessed 10-Sep-2018]. [Online]. Available: https://www.virten.net/2018/04/vmware-esxi-3-5-6-7-hypervisor-size-comparison/

[206] VMware, "vSphere ESXi Bare-Metal Hypervisor," 2016, [Online; accessed 10-Sep-2018]. [Online]. Available: http://www.vmware.com/products/esxi-and-esx.htmlhttps://www.vmware.com/products/esxi-and-esx/overview

[207] L. Poggemeyer, P. Short, N. Schonning, L. Iwer, and S. Cooley, "Supported Windows guest operating systems for Hyper-V on Windows Server," 2017, [Online; accessed 10-Sep-2018]. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/supported-windows-guest-operating-systems-for-hyper-v-on-windows

[208] S. Cooley, J. Terry, and H. Juarez, "Hyper-V Architecture," 2018, [Online; accessed 11-Sep-2018].

[209] K. Davies, P. Short, L. Iwer, and L. Poggemeyer, "System requirements for Hyper-V on Windows Server," 2016, [Online; accessed 10-Sep-2018]. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/system-requirements-for-hyper-v-on-windows

[210] W. Roersma, "Hyper-V 2012 and 2012 R2 live virtual machine memory acquisition and analysis," 2014, [Online; accessed 10-Sep-2018]. [Online]. Available: http://www.wyattroersma.com/?p=87

[211] "Xen Project", "Xen Project Software Overview"," https://wiki.xenproject.org, 2018, [Online; accessed 27-Nov-2018].

[212] "KVM," [Online; accessed 11-Sep-2018]. [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[213] "QEMU," [Online; accessed 11-Sep-2018]. [Online]. Available: http://wiki.qemu.org/Main_Pagehttp://www.qemu-project.org/

[214] VMware, "VMware API and SDK Documentation," 2018, [Online; accessed 10-Sep-2018]. [Online]. Available: https://www.vmware.com/support/pubs/sdk_pubs.html

[215] Kerkhoff Technologies, "NetFilterQueue," https://pypi.org/project/NetfilterQueue, 2017, [Online; accessed 29-Jan-2019].

[216] P. Biondi", "Scapy"," https://scapy.readthedocs.io/en/latest/, 2017, [Online; accessed 29-Aug-2018].

[217] B. D. Payne, "pyvmi – A Python adapter for LibVMI," https://github.com/libvmi/libvmi-backup/tree/master/tools/pyvmi, 2013, [Online; accessed 29-Jan-2019].

[218] F. Block and A. Dewald, "Linux memory forensics: Dissecting the user space process heap," in *DFRWS 2017 USA Proceedings of the Seventeenth Annual DFRWS USA*, vol. 22. Elsevier, 2017, pp. S66–S75.

[219] A. Socała and M. Cohen, "Automatic profile generation for live Linux Memory analysis," *Digital Investigation*, vol. 16, pp. S11–S24, 2016.

[220] M. Cohen, "Rekall Agent User Manual," Rekall, Tech. Rep., 2017.

[221] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.* John Wiley & Sons, 2014.

[222] OpenSSL Software Foundation, "OpenSSL: Cryptography and SSL/TLS toolkit," https://www.openssl.com/, 2018, [Online; accessed 29-Jan-2019].

[223] D. C. Litzenberger, "Python Cryptography Toolkit (pycrypto)," 2013, [Online; accessed 12-Feb-2018]. [Online]. Available: https://pypi.python.org/pypi/pycrypto

[224] K. Reitz, "Cryptography," 2016.

[225] H. Kario, "Chacha20poly1305"," https://github.com/ph4r05/py-chacha20poly1305, 2019, [Online; accessed 15-Jan-2019].

[226] S. Stolfo, S. M. Bellovin, and D. Evans, "Measuring Security," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 60–65, 2011.

[227] J. Muehlstein, Y. Zion, M. Bahumi, I. Kirshenboim, R. Dubin, A. Dvir, and O. Pele, "Analyzing HTTPS Encrypted Traffic to Identify User Operating System, Browser and Application," in *Consumer Communications & Networking Conference (CCNC), 2017 14th IEEE Annual.* IEEE, 2017, pp. 1–6.

[228] ComputerProfile, "VMware by far the largest in the server virtualisation market," 2017,

[Online; accessed 10-Jan-2019]. [Online]. Available: https://www.computerprofile.com/analytics-papers/vmware-far-largest-server-virtualisation-market/

[229] D. J. Barrett, R. E. Silverman, and R. Silverman, *SSH, the Secure Shell: the definitive guide*. O'Reilly Media, 2009.

[230] B. Hay and K. Nance, "Circumventing Cryptography in Virtualized Environments," in *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. IEEE, 2012, pp. 32–38.

[231] T. Ylonen and C. Lonvick, "RFC 4251 - The secure shell (SSH) protocol architecture," *Internet Engineering Task Force*, 2005.

[232] ——, "RFC 4253 - The secure shell (SSH) transport layer protocol," *Internet Engineering Task Force*, 2005.

[233] ——, "RFC 4252 - The secure shell (SSH) authentication protocol," *Internet Engineering Task Force*, 2005.

[234] ——, "RFC 4254 - The secure shell (SSH) connection protocol," *Internet Engineering Task Force*, 2005.

[235] S. Tatham, "PuTTY," https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html, 2018, [Online; accessed 29-Jan-2019].

[236] J. Galbraith and O. Saarenmaa, "SSH file transfer protocol," *Internet Engineering Task Force*, 2006.

[237] "SSH Communications", "SSH Client for Windows - Comparison"," https://www.ssh.com/ssh/client, 2018, [Online; accessed 29-Jan-2018].

[238] D. Roethlisberger, ""SSLsplit"," https://www.roe.ch/SSLsplit, 2018, [Online; accessed 29-Jan-2019].

[239] E. Rescorla, "RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3," *Internet Engineering Task Force*, 2018.

[240] T. Dierks and E. Rescorla, "RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2," *Internet Engineering Task Force*, 2008.

[241] K. McCarthy, "World celebrates, cyber-snoops cry as TLS 1.3 internet crypto approved," 2018, [Online; accessed 5-Sep-2018]. [Online]. Available: https://www.theregister.co.uk/2018/03/23/tls_1_3_approved_ietf/

[242] N. Sullivan, "Why TLS 1.3 isn't in browsers yet," 2017, [Online; accessed 4-Dec-2018]. [Online]. Available: https://blog.cloudflare.com/why-tls-1-3-isnt-in-browsers-yet/

[243] J. Salowey, A. Choudhury, and D. McGrew, "RFC 5288 - AES Galois Counter Mode (GCM) Cipher Suites for TLS," *Internet Engineering Task Force*, 2008.

[244] D. McGrew, "RFC 5116 - An Interface and Algorithms for Authenticated Encryption," *Internet Engineering Task Force*, 2008.

[245] I. Ristić, *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*, 1st ed. Feisty Duck, 2014.

[246] K. Moore and C. Newman, "RFC 8314 - Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access," 2018.

[247] P. Ford-Hutchinson, "RFC 4217 - Securing FTP with TLS," 2005.

[248] SSL Pulse, "SSL Pulse," 2019, [Online; accessed 18-Mar-2019]. [Online]. Available: https://www.ssllabs.com/ssl-pulse/

[249] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616 - Hypertext Transfer Protocol–HTTP/1.1," *Internet Engineering Task Force*, 1999.

[250] Caswell, M, "OpenSSL 1.1.1 Is Released," https://www.openssl.org/blog/blog/2018/09/11/release111/, 2017, [Online; accessed 29-Jan-2019].

[251] wolfSSL, "wolfSSL," https://www.wolfssl.com/, 2018, [Online; accessed 29-Jan-2019].

[252] LibreSSL, "LibreSSL," https://www.libressl.org/, 2018, [Online; accessed 29-Jan-2019].

[253] GnuTLS, "The GnuTLS Transport Layer Security Library," https://www.gnutls.org/, 2018, [Online; accessed 29-Jan-2019].

[254] Mozilla, "Network Security Services," https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS, 2018, [Online; accessed 29-Jan-2019].

[255] K.-K. R. Choo and P. Grabosky, "Cyber crime," *Oxford Handbook of Organized Crime*, 2013.

[256] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 3–24.

[257] A. S. Shekhawat, F. Di Troia, and M. Stamp, "Feature Analysis of Encrypted Malicious Traffic," *Expert Systems with Applications*, 2019.

[258] G. Gu, J. Zhang, and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[259] X. Ma, X. Guan, J. Tao, Q. Zheng, Y. Guo, L. Liu, and S. Zhao, "A Novel IRC Botnet Detection Method Based on Packet Size Sequence," in *2010 IEEE International Conference on Communications*. IEEE, 2010, pp. 1–5.

[260] B. Lin, Q. Hao, L. Xiao, L. Ruan, Z. Zhang, and X. Cheng, "Botnet Emulation: Challenges and Techniques," in *Emerging Technologies for Information Systems, Computing, and Management*. Springer,New York,NY, 2013, vol. 236 of Lecture Notes in Electrical Engineering, pp. 897–908.

[261] C. P. Lee, "Framework for botnet emulation and analysis," Ph.D. dissertation, Georgia Institute of Technology, 2009.

[262] E. Thioux, M. Amin, and O. A. Ismael, "System and method for analysis of a memory dump associated with a potentially malicious content suspect," Feb. 5 2019, US Patent App. 10/198,574.

[263] "SSL Blacklist," 2018, [Online; accessed 10-May-2018]. [Online]. Available: https://sslbl.abuse.ch

[264] "VirusShare," [Online; accessed 18-May-2018]. [Online]. Available: https://virusshare.com/

[265] Trend Micro, "Trend Micro," https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/

online-banking-trojan-brief-history-of-notable-online-banking-trojans, 2015, [Online; accessed 20-Mar-2019].

[266] R. PP, "Zeus/Zbot Trojan Attacks Credit Cards of Banks," 2016, [Online; accessed 13-Feb-2019]. [Online]. Available: https://techpp.com/2010/07/15/zeuszbot-trojan-attacks-credit-cards-of-banks/

[267] "Trojan.Zbot," 2016, [Online; accessed 5-Jul-2018]. [Online]. Available: https://www.symantec.com/security-center/writeup/2010-011016-3514-99

[268] Panda, "Zeus is Still the Base of Many Current Trojans," 2017, [Online; accessed 10-Feb-2019]. [Online]. Available: https://www.pandasecurity.com/mediacenter/panda-security/zeus-trojan/

[269] D. Jackson, "Gozi Trojan," 2007, [Online; accessed 10-Feb-2019]. [Online]. Available: https://www.secureworks.com/research/gozi

[270] M. Alvarez, T. Agayev, and T. Darsan, "Q1 2018 Results: Gozi (Ursnif) Takes Larger Piece of the Pie and Distributes IcedID," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: https://securityintelligence.com/q1-2018-results-gozi-ursnif-takes-larger-piece-of-the-pie-and-distributes-icedid/

[271] E. Brumaghin, H. Unterbrink, and A. Weller, "Gozi ISFB Remains Active in 2018, Leverages "Dark Cloud" Botnet For Distribution," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: https://blogs.cisco.com/security/talos/gozi-isfb-remains-active-in-2018

[272] M. Garnaeva, F. Sinitsyn, Y. Namestnikov, D. Makrushin, and A. Liskin, "Kaspersky Security Bulletin:

Overall Statistics for 2016," p. 31, 2016, [Online; accessed 10-Feb-2019]. [Online]. Available: https://kasperskycontenthub.com/securelist/files/2016/12/Kaspersky_Security_Bulletin_2016_Statistics_ENG.pdf

[273] A. Mohanta, A. Saldanha, and P. Kimayong, "The Gozi Sleeper Cell," 2018, [Online; accessed 10-Feb-2019]. [Online]. Available: https://forums.juniper.net/t5/Threat-Research/The-Gozi-Sleeper-Cell/ba-p/329691

[274] C. Weller, "CyberSecurity in 120 Secs_ The Comeback of Gozi Malware," 2016, [Online; accessed 6-Feb-2019]. [Online]. Available: https://blog.ensilo.com/cyber-security-in-120-secs-the-comeback-of-gozi-malware

[275] M.-E. M. Léveillé, "TorrentLocker," ESET, Tech. Rep., 2014. [Online]. Available: http://www.welivesecurity.com/wp-content/uploads/2014/12/torrent_locker.pdf

[276] M.-E. M.Léveillé, "TorrentLocker: Crypto-ransomware still active, using same tactics," 2016, [Online; accessed 8-Feb-2019]. [Online]. Available: https://www.welivesecurity.com/2016/09/01/torrentlocker-crypto-ransomware-still-active-using-tactics/

[277] J. M. Kambic, "Extracting cng tls/ssl artifacts from lsass memory," Ph.D. dissertation, Purdue University, 2016.

[278] B. Herzog and Y. Balmas, "Great crypto failures," *Virus Bulletin 2016*, 2016.

[279] Y. Nir, "ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec," *Internet Engineering Task Force*, 2015.

[280] B. Jungk and S. Bhasin, "Don't fall into a trap: Physical side-channel analysis of ChaCha20-Poly1305," in *2017 De-*

*sign, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1110–1115.

[281] M. Robshaw and O. Billet, "New Stream Cipher Designs," in *New Stream Cipher Designs*, vol. 4986 of Lecture Notes in Computer Science. Springer, 2008.

[282] D. Klinec, "Chacha20poly1305," https://github.com/ph4r05/py-chacha20poly1305, 2018, [Online; accessed 29-Jan-2019].

[283] D. McGrew and D. Bailey, "RFC 6655 - AES-CCM Cipher Suites for Transport Layer Security (TLS)," *Internet Engineering Task Force*, 2012.

[284] A. Case and G. G. Richard III, "Memory forensics: The path forward," *Digital Investigation*, vol. 20, pp. 23–33, 2017.

[285] S. J. Yang, J. H. Choi, K. B. Kim, R. Bhatia, B. Saltaformaggio, and D. Xu, "Live acquisition of main memory data from Android smartphones and smartwatches," *Digital Investigation*, vol. 23, pp. 50–62, 2017.

[286] B. P. Kondapally, "What is IoT Forensics and How is it Different from Digital Forensics?" https://securitycommunity.tcs.com/infosecsoapbox/articles/2018/02/27/what-iot-forensics-and-how-it-different-digital-forensic, 2018, [Online; accessed 29-Jan-2019].

[287] S. Alabdulsalam, K. Schaefer, T. Kechadi, and N.-A. Le-Khac, "Internet of Things Forensics–Challenges and a Case Study," *arXiv preprint arXiv:1801.10391*, 2018.

# Appendix A

# Package Dependencies

| Application | Dependency |
|---|---|
| LibVMI | libxc-dev |
| | bison |
| | flex |
| | libtool |
| | libxen-dev |
| | automake |
| | check |
| | make |
| | fuse |
| | libfuse-dev |
| | glib2.0 |
| PyVMI | LibVMI |
| Volatility | python-crypto |
| | python-pip |
| | distorm3 |
| | python-openpyxl |
| | python-dev |

# Appendix B

# Hypervisor Research Review

| **Authors** | **System Name** | **Hypervisor** |
|---|---|---|
| Garfinkel & Rosenblum | Livewire | VMware Workstation |
| Kourai & Chiba | HyperSpector | Custom |
| Joshi, King, Dunlap, & Chen | IntroVirt | User-Mode Linux |
| Jiang, Wang, & Xu | VMwatcher | VMware, QEMU, Xen, and UML |
| Yin, Song, Egele, Kruegel, & Kirda | Panorama | QEMU |
| Bryan D. Payne, Carbone, Sharif, & Lee | Lares | Xen |
| Hay & Nance | VIX | Xen |
| Lanzi, Sharif, & Lee | K-Tracer | QEMU |
| Dolan-Gavitt, Leek, Zhivich, Giffin, & Lee | Virtuoso | QEMU |
| Benninger, Neville, Yazir, Matthews, & Coady | Maitland | Xen |
| Harrison, Cook, McGraw, & Hamilton | - | Xen |
| Fu, Rd, & Lin | EXTERIOR | QEMU |
| Chung, Khatkar, Xing, Lee, & Huang | NICE | Xen |
| Lengyel et al | DRAKVUF | Xen |
| Hizver & Chiueh | RTKDSM | Xen |
| Fattori et al | AccessMiner | Custom |
| Saxon, Bordbar, & Harrison | - | VMWare Fusion |
| Shi, Yang, Li, & Wang | SPEMS | Xen |
| Taubmann et al | TLSkex | Xen |
| Zhan, Ye, Fang, Du, & Su | CFWatcher | Xen |
| Tien, Liao, Chang, & Kuo | - | Xen |
| Upadhyay, Gohel, Pons, & Lagos | - | Xen |
| Zhan, Li, Ye, Zhang, Fang, & Du | - | Xen |

# Appendix C

# NetScantbl Plugin Output

```
Offset, Proto, LocalAddress, ForeignAddress, State, Pid, Owner, PIDOffset(P), Created
246292728982720,UDPv4,0.0.0.0:65427,*:*,,788,svchost.exe,401614016,2018-04-05 12:46:00 UTC+0000
246292728982720,UDPv6,:::65427,*:*,,788,svchost.exe,401614016,2018-04-05 12:46:00 UTC+0000
246292728549392,TCPv4,192.168.137.169:1549,23.32.54.90:443,CLOSED,3852,OneDrive.exe,1910908416,2018-04-05 12:40:23 UTC+0000
246292735941424,UDPv4,0.0.0.0:5353,*:*,,788,svchost.exe,401614016,2018-04-05 12:43:28 UTC+0000
246292735961904,UDPv4,0.0.0.0:5355,*:*,,788,svchost.exe,401614016,2018-04-05 12:43:28 UTC+0000
246292736100416,UDPv4,0.0.0.0:5353,*:*,,788,svchost.exe,401614016,2018-04-05 12:43:28 UTC+0000
246292736100416,UDPv6,:::5353,*:*,,788,svchost.exe,401614016,2018-04-05 12:43:28 UTC+0000
246292744297664,UDPv4,0.0.0.0:0,*:*,,920,svchost.exe,375505024,2018-04-05 12:42:27 UTC+0000
246292745183248,UDPv6,-:546,*:*,,800,svchost.exe,360630336,2018-04-05 12:45:52 UTC+0000
246292745753824,UDPv4,0.0.0.0:5355,*:*,,788,svchost.exe,401614016,2018-04-05 12:43:28 UTC+0000
246292745753824,UDPv6,:::5355,*:*,,788,svchost.exe,401614016,2018-04-05 12:43:28 UTC+0000
246292746087504,UDPv4,0.0.0.0:58282,*:*,,788,svchost.exe,401614016,2018-04-05 12:45:30 UTC+0000
246292746087504,UDPv6,:::58282,*:*,,788,svchost.exe,401614016,2018-04-05 12:45:30 UTC+0000
246292754176192,TCPv4,192.168.137.169:139,0.0.0.0:0,LISTENING,4,System,2110113856,2018-04-05 12:38:45 UTC+0000
246292754313232,UDPv4,-:138,*:*,,4,System,2110113856,2018-04-05 12:38:45 UTC+0000
246292757002256,TCPv4,0.0.0.0:135,0.0.0.0:0,LISTENING,632,svchost.exe,325796800,2018-04-05 12:38:42 UTC+0000
246292757002624,TCPv4,0.0.0.0:135,0.0.0.0:0,LISTENING,632,svchost.exe,325796800,2018-04-05 12:38:42 UTC+0000
246292757002624,TCPv6,:::135,:::0,LISTENING,632,svchost.exe,325796800,2018-04-05 12:38:42 UTC+0000
246292757012160,TCPv4,0.0.0.0:1536,0.0.0.0:0,LISTENING,408,wininit.exe,2110349440,2018-04-05 12:38:42 UTC+0000
246292757085120,TCPv4,0.0.0.0:1536,0.0.0.0:0,LISTENING,408,wininit.exe,2110349440,2018-04-05 12:38:42 UTC+0000
246292757085120,TCPv6,:::1536,:::0,LISTENING,408,wininit.exe,2110349440,2018-04-05 12:38:42 UTC+0000
246292757735168,TCPv4,192.168.137.169:1553,204.79.197.200:443,CLOSED,3236,SearchUI.exe,847083584,2018-04-05 12:41:43 UTC+0000
```

Figure C-1: Netscantbl Output

# Appendix D

# Malware Client Downloads

| Class | MD5 | Status/Issue |
|---|---|---|
| Bergat | 5bfd12024dd4ef335d07306956a54026 | HTTP GET fails |
| Bergat | 8b39c7da3d444c0b5fdf25b136a6f65b | No Session |
| Cryptowall | a3c3a4acca0c8e88c23b1166c7438c92 | HTTP POST Fails |
| Deshacop | 4a6077a694df24b38adb816edcb0902e | Uses UDP |
| Deshacop | b097ed36451d0055dc06026050fdd8df | Uses UDP |
| DomaIQ | eebc815e0296787aca9052957259dfdf | HTTP |
| Dridex | 62e4f7cfa529ef63439e88ff176cc6c | No Session |
| Dridex | 9d75ff0e9447ceb89c90cca24a1dbec1 | No Session |
| Fareit | 1084b25aa5a709618146281cadfe2a41 | HTTP GET fails |
| Fareit | 1f45f04394e371b885ea16de6ba37097 | HTTP GET fails |
| Fareit | 311176f0e2708d3a388e7690263290d7 | HTTP GET fails |
| Gozi | aeb5bb78ab442bc94bb94d968754e523 | Succeeded |
| Kazy | 3f7a23acf65eb86ecc8e7492f70f6fdf | Did not start |
| Locky | 326fb6a1e746a4a299a03aa743ece109 | No Session |
| Locky | a61252e123e7fe72c5c8e7b560c89ede | No Session |
| Locky | d5412f708941c3950f9d3efa49cb0a34 | No Session |
| Papras | 1a210fd7bf7d465abe6fdb737262ce9a | HTTP GET fails |
| Razy | 0a6395e345fdf92a9b6f91fe775f28ef | No Session |
| Sality | 831a49ac7903be74524c632ada3f5cf5 | No Session |
| Symmi | 12a179ecf60c35ba474b8690fa9815be | No Session |
| Teslacrypt | 477957a9d5444dd4afa4fc01f3d8f510 | No Session |
| Teslacrypt | 90eb6fa9a801f5c125fd6816c5e4250e | No Session |
| Trickbot | 3d3e08ad3a8f3b35b9a10aa6c57b290f | HTTP IP Request Fails |
| Upatre | 0ba538e0ff0723f227b48611205d0e53 | HTTP POST Fails |
| Vawtrak | 7278ca09c39a2647b428b931cb9a0b23 | HTTP fails |
| Virtob | 070175ac1fa63c820f102cac820c1ca0 | No Session |
| WannaCryptor | 0291b0e8d72e728c5b7e5559b7493c25 | No Session |
| WannaCryptor | 3a1b0f7ee8a921a0aa24f19bab452fb9 | No Session |
| WannaCryptor | 7a3ddd634eea691850376105fb629318 | No Session |
| WannaCryptor | b581da8662097751690bb23658487c5c | No TLS |
| Yakes | 677836d62acfe363b6158227e5aeacb9 | No TLS |
| Yakes | c872b5fce8e65a701a1c5a19e3f387f5 | Starts but no TLS |
| Zbot | 2ed76f29535d897dec01e2b4fede5271 | Did not start |
| Zbot | b9e6c891dd76335b3f41f844442911e3 | Did not start |
| Zbot | eeef1e062c8011cabb23b3c833ff766a | Succeeded |
| Zbot | 389d73e184abc45e353e14ffd59b233f | Did not start |

# Glossary

**AES**

Advanced Encryption Standard, a symmetric block algorithm formerly called Rijndael, used in secure communications protocols.

**AES-CBC**

AES mode where the IV is typically the ciphertext of the previous block.

**AES-GCM**

AES mode where Galois field authentication follows AES-CTR for confidentiality and integrity.

**bespoke**

software that is specifically coded for the MemDecrypt framework.

**ChaCha20**

symmetric stream algorithm used in secure communications protocols.

**ciphertext**

obfuscated data that results from application of an encryption algorithm to plaintext.

**DES**

Digital Encryption Standard, a symmetric block algorithm used before AES.

**entropy**

the difficulty of predicting an observation. Shannon entropy is the average number of bits that describe a string.

**heap**

memory that is dynamically allocated at run-time for process data.

**hook**

technique used to modify behaviour of an operating system or application by intercepting function calls.

**HTTP**

HyperText Transfer Protocol, protocol used for World Wide Web client server communications.

**hypervisor**

software that creates and runs virtual machines enabling access to the underlying physical hardware.

**initialisation vector**

string used in symmetric block encryption to ensure that different ciphertext is produced for different instances of the same plaintext with an encryption key.

**IoT**

Internet of Things, embedding of Internet connectivity into physical devices.

**IV**

acronym for Initialisation Vector.

**kernel**

software associated with the operating system providing hardware access and other privileged functions.

**nonce**

similar to Initialisation Vector but commonly used in stream algorithms to generate key streams.

**PCI**

Peripheral Component Interconnect, mechanism for adding hardware device to computer.

**plaintext**

information that is intelligible in some sense on which an encryption algorithm operates to generate obfuscated data.

**process**

instance of a program executing in memory operating in user and kernel modes.

**process hollowing**

malware technique to evade detection where a legitimate process is created in a suspended state and legitimate code replaced with malicious code.

**RSA**

Rivest–Shamir–Adleman, an asymmetric public-key algorithm used in secure data exchange.

**SHA**

family of hash algorithms used for authentication.

**SSH**

Secure Shell Protocol, commonly used for server remote management.

**TLS**

Transport Layer Security Protocol, commonly used for Internet client server confidentiality and integrity.

**VAD**

Virtual Address Descriptor, data structures used in Windows memory management to track process virtual addresses.

**virtual machine**

emulation of a computer system that runs an operating system and applications.

**VMI**

Virtual Machine Introspection, monitoring virtual machines from the outside.

**VPN**

Virtual Private Network protocols to provide a tunnel between a client and server.

**WEP**

Wired Equivalent Privacy, a superseded Wireless Protocol.

**XOR**

bitwise operation where the output bit is 1 if and only if one input bit is 1.