Efficient Retrieval of Key Material for Inspecting Potentially Malicious Traffic in the Cloud

John T. Saxon, Behzad Bordbar, Keith Harrison
School of Computer Science
University of Birmingham
Birmingham, UK
Email: { j.t.saxon, b.bordbar, k.harrison }@cs.bham.ac.uk

Abstract—**Cloud providers must detect malicious traffic in and out of their network, virtual or otherwise. The use of Intrusion Detection Systems (IDS) has been hampered by the encryption of network communication. The result is that current signatures cannot match potentially malicious requests.**

A method to acquire the encryption keys is Virtual Machine Introspection (VMI). VMI is a technique to view the internal, and yet raw, representation of a Virtual Machine (VM). Current methods to find keys are expensive and use sliding windows or entropy. This inevitably requires reading the memory space of the entire process, or worse the OS, in a live environment where performance is paramount. This paper describes a structured walk of memory to find keys, particularly RSA, using as fewer reads from the VM as possible. In doing this we create a scalable mechanism to populate an IDS with keys to analyse traffic.

I. Introduction

The use of encryption has been seen as a major threat to Intrusion Detection Systems (IDS) for a long time [1]. Once the packet payloads are encrypted, the currently available signature based methods of detecting anomalous and harmful traffic becomes completely futile [2]. Within the cloud, where the infrastructure is shared between many users, cloud providers have an obligation to protect the infrastructure and other users from malicious behaviour. As this architecture accumulates large amounts of valuable data, the pay-off for a successful attack, and the damage incurred, could be substantially high. As a result, there is clear scope for developing technologies for decrypting data within the Cloud to allow IDSs to protect its users.

Virtual Machine Introspection (VMI) is a technique used for viewing, and potentially modifying, the internal state of a guest Virtual Machine (VM). It has been used successfully in developing novel security techniques which are tailored for use within the Cloud, for instance IDSs [3]–[5] and can also be used for managing cloud resources. VMI can be used to gather information regarding the VM, for instance its load, the raw byte stream and with correct traversal: operating system (OS) structures like the process table. This allows us to generate many programs to collate and use this information. Another use of VMI is found in Forensic Virtual Machines (FVMs) [6], [7]. These are small VMs that use this technique to find observables that may indicate the presence of malicious behaviour, for example the bot-net ZeuS changes a registry value to start at user login [8]. In this paper, we present a method of using VMI to gather keys to allow the decryption of data and assist an IDS. There has been suggestions to introduce Introspection-as-a-Service [9], which would allow such programs. The implementation, and usage, of this technology can be controversial and must be safeguarded by the rule of law, consent of the users and contractual obligations.

The focus of this paper is to provide a mechanism to find keys within memory, using VMI, to pass to an IDS such that it can analyse its traffic. Current methods are computationally expensive and require a sliding window of the entire memory space to find a sequence of high entropy bytes or signatures [10], [11]. This approach is very expensive when reading bytes from a live VM, instead we look at how these keys are stored in memory and provide a structured walk of the OS's internal representation. We demonstrate how our method can be applied to RSA and provide small inexpensive checks to decrease the amount of interaction, i.e. reading, with the VM for finding keys. Specifically looking at properties of the integral components of a private key, for example the length and entropy, and the relationships between them, derived from the way keys are generated. We use Apache2, with TLS enabled, to demonstrate and evaluate our approach.

The paper is organised as follows: we start by introducing some preliminaries in section II. We then describe the problems regarding finding keys in section III and show a sketch of the solution in section IV. Followed by a detailed view of our method in section V. A sketch of the implementation is provided in section VI and the evaluation follows in section VII. We then finish with a discussion of related work, section VIII, and a conclusion in section IX.

II. Preliminaries

A. Virtualisation and the Cloud

Virtualisation provides an abstraction for the hardware of a physical host. This abstraction is often called a Hypervisor or a Virtual Machine Monitor (VMM). This decoupling of physical hardware to a logical piece of software allows the complete emulation of an OS through a Virtual Machine (VM) [12]. It also allows a single host to run numerous versions of different OSs which share the resources available to it.

Cloud computing is a model that takes virtualisation a step further, rather than having virtualisation over a single host, it allows multiple hosts to pool their resources together. A common use for the Cloud is providing *infrastructure as a service* (IaaS) [13]. Opposed to having a set of default OSs that the user can choose and customise, consumers are generally able to provide their own images to be hosted for their systems. This is often beneficial to the customer as they have complete control over the OS they are using. However, this process can prove detrimental to the Cloud service provider as their security is in the hands of their users rather than themselves. If a user does not update a malware scanner, or indeed if he does not have one, other consumers could be affected as they reside on the same network and can aid in the spread of the malware. This with the pure mass of information collected in this environment provides large rewards for successful attackers.

B. Virtual Machine Introspection

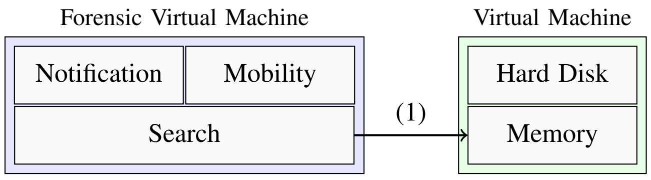
VMI allows applications to access the runtime state of a VM, including memory, registers and disks. This enables a VM with sufficient privileges to access another VM on the same server [3]. Providing a powerful forensic technique allowing analysis from outside of a "target" VM. This form of passive monitoring is advantageous when you wish the application, carrying out VMI, to remain hidden from the VM itself.

VMI has been used in digital forensics [14]–[16] and for IDSs [3]–[5] to isolate the forensics functionality from the host it is monitoring. Its use is widening and libraries are now being written [15], [17] and can be used to provide introspection-as-a-service [9].

Whilst introspection is a useful mechanism, knowledge bases regarding the internal structures of OSs vary in depth and quality depending on the availability of the source code. For example the Linux code base is open to the public, thus the internal structures are known and their use be deduced, whereas Windows is closed and requires some reverse engineering. The latter is being continually explored by the digital forensics community. Volatility is one of the largest open-source system analysers, managed by Volatile Systems [14]. As VMI works completely outside of the target VM, it does mean we do not have access to memory structures using native APIs. This means a program that uses it must be able to handle different memory layouts; this is especially difficult with closed source OSs like Windows.

C. Forensic Virtual Machines

Keith Harrison et al. [6] introduced the concept of an FVM. An FVM benefits from VMI to observe the internal state of a guest VM. Particularly looking for potential signs of malicious behaviour, e.g. unknown changes in the registry, etc. An FVMs prime objective is to act like a physician within a virtualised environment: to identify potentially infected VMs, within a large population, so that remedial action can start. For example, within the Cloud we can slow down, stop or snapshot the guest for further analysis. The key difference between the FVM and general practitioner is its patient base. An FVM can be part of a much larger neighbourhood of VMs, within the Cloud, that needs to be monitored concurrently. Each FVM is tailored for a single purpose to reduce its attack surface, thus if we want many of them, they need to be small.



(1): The FVM search module introspects into a VM's memory.
Fig. 1: The architecture of an FVM.

The structure of an FVM is illustrated within Figure 1, the architecture consists of three internal components. Due to the size of the Cloud and the number of VMs to introspect, the *mobility* module allows an FVM to move from one VM to another. This prevents the hypervisor from setting up another FVM with a different target, efficiently using resources as not to starve the guest VMs by absorbing more clock cycles. The *search* module allows for plug-and-play with different search strategies, for instance one strategy could look within the Windows registry, another the process table. For example an FVM could use VMI to check whether ZeuS [8] had changed a registry value to make it start at boot. We study how this module can be used to look within memory to find key material via a structured walk. This structured walk will reduce the total clock cycles needed to complete the analysis. The final module is the *notification* module: it allows outside communication and communication within the FVM network. In our scenario this is the mechanism to report the collected key material to the IDS, so intrusion detection can take place.

The implementation of FVMs is complex; however Shaw et al. [7] implemented and tested the performance of an FVM using the Xen hypervisor [18] and Mini-OS [19]. The use of paravirtualisation, i.e. whereby the guest VM calls to the hypervisor (*hypercalls*) for non-virtualisable instructions, improves the throughput of an FVM quite substantially.

D. TLS and RSA

TLS is a protocol that enables the use of cipher suites on network traffic. It is used to encrypt data between two parties that are communicating whilst it also has the ability to sign traffic to prove its authenticity. There has been a great move towards TLS, by default, by web service providers to protect their customer's data in transit [20]. The protocol provides this privacy with a mixture of asymmetric and symmetric key cryptography. Due to the performance of public key cryptography, symmetric key cryptography is used to encrypt the data itself using a *session key*. This unique key is used for communication between clients and servers.

The keys used within protocols such as TLS need to be exchanged such that the recipient, often the server, can decrypt the data within the message(s). There are three mechanisms for exchanging keys: 1) using the asymmetric cipher, i.e. the client encrypts it using the servers public key. If the server is the only one with the key, only that server should be able to decrypt it; 2) using an Ephemeral Diffie-Hellman key exchange, enabling each party to derive the key without actually sending it; and 3) using Elliptical-Curve Diffie-Hellman, which generates a session key based on finite fields [21]. The latter two provide perfect forward secrecy (PFS), that is if an attacker has access to the private key, they cannot decrypt communication traffic as they need the random values used by both parties to derive the session key and decrypt it. The former however transfers the session key within the communication, if the private key is known then the session key can be decrypted. Our system depends on the former being used, as this allows the IDS to decrypt the data. We feel this is feasible as at the time of writing, many websites, including banks still use this mechanism.

1) RSA: TLS requires an asymmetric cipher to prove authenticity to the client, such that they can sign the response. RSA is an asymmetric cipher that is well used for TLS communication over the Internet. This particular cipher is used with the above mentioned banks. The cipher contains three main components, a modulus n (the product of two primes p and q), a private exponent d and a public exponent e. The modulus and the public exponent form the public key (n, e) and the private exponent forms the private key. Operations that use the private key, signing and decryption, are computationally expensive. This can be optimised by the use of the Chinese Remainder Algorithm, which requires access to the original primes to speed up the two processes [22]. OpenSSL [23] is a library that keeps this information to optimise the use on high load servers. Our work heavily concentrates on finding RSA key material within memory.

2) Current Methods of Finding Keys in Memory: The core applications that gather key material have been in cold-boot attacks and general forensic [10], [11], [24], [25]. They both, in a general sense, work on snapshots of a machine's, virtual or otherwise, memory. The former concentrates more on recovery of keys as a cold-boot entails forcefully turning off a machine to avoid any clean-up of memory. The machine is then either booted from a USB device to dump the memory, warm-boot, or the RAM is moved to a controlled machine to boot and dump the memory. Memory will degrade in this setting as the transistors lose their energy, thus further reconstruction is needed. The latter however, particularly in the setting of Volatility [14] and libvmi [15], is based on snapshots of VMs or access to live VMs via a hypervisor's native libraries. Live access has no need for reconstruction as the transistors remain set. We concentrate on the recognition of keys within a live system so we assume no reconstruction is necessary.

T Klein provides a method to find certificates within memory which has been implemented as a Volatility [14] plug-in for use by the forensic community for Windows OSs. Volatility is a python library used for the introspection of memory blocks or files and can be extended for multiple inputs. This plug-in allowed Klein to find keys within memory by finding a signature of the Abstract Syntax Notation (ASN.1) used for storing RSA private keys, particularly those using PKCS#8, a generic representation of a private key. This approach uses a sliding window through a process's virtual memory to find the signature of this certificate. Requiring the plug-in to traverse the entire memory space of a process in order to find a certificate using a very specific representation. It also assumes that the cryptographic library does not scrub the ASN.1 certificate after it has generated its internal data structure. Klein's approach also calls the OpenSSL [23] command line tool to verify its findings, which is not suited for a live system.

Another approach for finding keys within memory is finding portions of high entropy within process' memory [25]. Entropy is the measure of the lack of order or predictability of a subject. In the case of RSA, and indeed most cryptographic ciphers, randomness is collected to increase the entropy of the key. The two primes, p and q, are populated with random data and then incremented or decremented until a probable prime has been found. This randomness is then inherited by the other components of the key.

III. Description of the Problem

Figure 2 shows the entropy of an Ubuntu 12.04-4 memory dump of size 512MB. The blue line shows the entropy of memory using a 2048-bit window, the size of the keys within. There are only two keys within memory, a key for SSHd and an Apache2 Virtual Host. The range specified in red shows where the keys *could* be based on key entropy (range discussed in subsection V-C). You'll notice that the red areas take less than half of the memory. The sliding window will cause many false positives without additional checks due to the number of potential locations.

The brute force approach of using a sliding window is expensive and can be reduced by a structured walk. Both methods require a sliding window of the VMs memory such that over a VM with memory size of vm_s bytes and a sliding window of w bytes, $vm_s - w$ samples must be taken. It can be reduced by a factor of four if 32-bit rounding were to take place, however the calculation of entropy still isn't included. This method also does not give definitive boundaries, as more than one piece of information could be in a single red block, thus the number of false positives can rise.

This sliding window can be very expensive in a virtualised environment, forcing the controller domain of the hypervisor to carry out operational tasks as well as enabling introspection from outside of the kernel. In this paper we present an efficient method of finding keys in an OS's memory.

IV. Sketch of the Solution

The computational requirements of finding keys within memory are high. This is based on looking for a sequence of high entropy bytes or a signature of a key container. To identify a sequence of high entropy bytes the application must read the entire memory space of a guest OS to find these traits. Forensics tools for virtualisation are often used post-mortem, i.e. a problem is found, a snapshot is made and then analysis is performed on that snapshot. This process can be moved to non-production equipment, for instance servers that aren't providing customer services, thus performance isn't as important. When using VMI on a live guest, performance

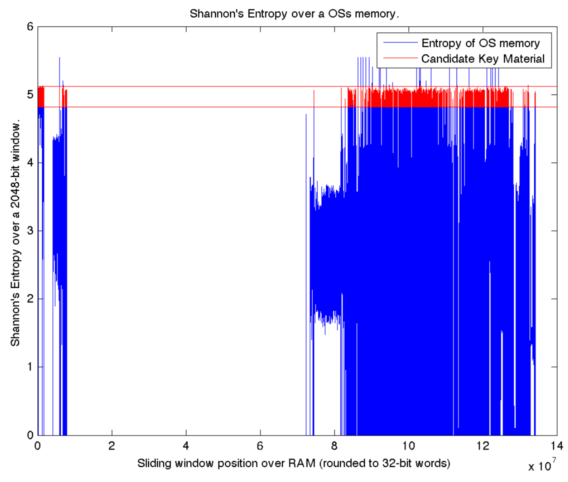


Fig. 2: Entropy of a 512MB Ubuntu 12.04-4 Memory Dump.

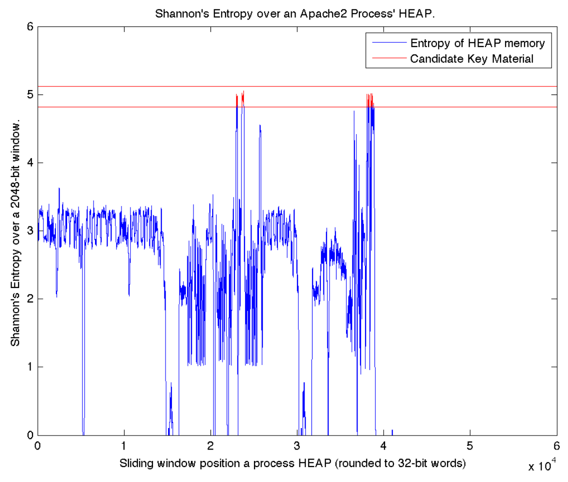


Fig. 3: Entropy of an Apache2 Process' HEAP.

becomes a core requirement. Each read from the guest VM, and each tick of a CPU used takes processing power away from the customer VMs the virtualised host is providing.

For continuous analysis, in the case of a mobile FVM, performance is paramount. This is due to all communication between the guest and the introspector being sent via the controller domain, the entity managing resources within the virtual environment. With this additional requirement we concentrate our research on finding RSA keys within memory using a method that could be plugged into an FVM with performance in mind [6], [7].

Our work concentrates on two well known libraries, OpenSSL [23] and GnuTLS [26], used by applications to provide TLS transport in order to find a more targeted search algorithm to collate keys from memory. These were chosen due to their prominence within applications that use secure communication. The internal mechanisms used for holding RSA key material are big number libraries. These libraries provide methods for representing and manipulating numbers larger than 64-bits. Specifically OpenSSL implements its own library, BN, whilst GnuTLS uses another alternative, the GNU Multiple Precision Arithmetic Library (GMP) [27]. Both, BN and GMP, have a similar internal structure: a contiguous array of integers and some flags, i.e. is the number negative, etc. As these numbers have to be able to expand and contract, they are kept on the heap, within user-space of the process. Although it is more natural to order the array with the least significant bytes at the higher end of the array, big numbers reverse this as to easily allow them to expand. These techniques are common within libraries that are similar and we believe our method can be adjusted, with minor modifications, to work with others. For instance libtommath only uses 28-bits of each integer word, rather than entire 32-bits, so it can handle overflows better when bit shifting[1].

The use of big number libraries in RSA allowed us to narrow the search space of memory quite substantially. As shown in Figure 2, there is a lot of space where keys can be. The HEAP however is a lot simpler, as shown in Figure 3. This illustrates the HEAP of an Apache2 process, in this you'll see two small, distinct, areas for exploration. The latter block, at approximately $3.8 * 10^4$ bytes, is actually where the RSA key resides. A sliding window over this particular instance still requires $6 * 10^4$ calculations. We will demonstrate that our method allows us pin-point the key avoiding boundary issues generally apparent when using a sliding windows.

The heap is a sequential list of elements within the `brk` (the data segment, where the heap resides) such that one can traverse down using its simple format without the aid of the memory manager. This header gives more information regarding the entry, like size. The libraries in question are optimised to only use as much memory as required, since we know the length of the key and its components we can filter the HEAP into segments we deem *interesting*.

With the basic knowledge of the HEAP and its structure, we can analyse the building blocks of RSA. These components have certain properties that can be validated quickly to improve the detection of key material and save computation whilst doing so. Specifically, we start by looking at the length of the components to group possible heap entries, then take advantage of the way the modulus is generated to find tuples of n, p and q. The latter is possible as *ALL* key material is kept within the certificate to quicken the calculations that use the private key. To further narrow this we conduct a small, efficient probabilistic primality check of p and q and only then do we check the entropy of n.

Completing these tasks reduces the required number of reads of the guest VM's internal state, thus reducing its impact on the virtualised environment. We could argue, however, that our structured approach revokes some generality as we need to know the internal structure of the OS itself as well as its memory allocation techniques. For VMI to be used in more applications: APIs that wrap VMI for accessing OS specifics

[1]https://github.com/libtom/libtommath/blob/master/tommath.h#L81

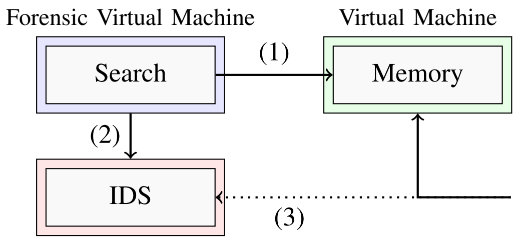


Fig. 4: Using FVMs for finding keys for an IDS.

must be in place, otherwise this places a domain knowledge burden onto the developer.

V. Finding Key Material

Figure 4 describes the use of FVMs to find keys that can be passed onto an IDS to allow it to decrypt traffic for analysis. As described in subsection II-C, each FVM has a search module which includes the code for conducting a search of memory. We implement the method suggested in this section in the FVM. Then, as depicted in Figure 4, the FVM searches the memory for keys (1). After finding the keys, they are passed onto the IDS (2) which consequently decrypts and analyses traffic for anomalous or harmful communication (3). This section discusses the properties of RSA encryption such that we can use VMI to find potential RSA keys used within a process, whilst keeping the total number of false positives down to a minimum. We do this by describing the tools that are used for the deployment and implementation of cryptographic schemes, particularly RSA. We explain where these structures are kept and how we can analyse these containers to find what is a key and what is not.

A. Big Number Libraries

Many popular cryptographic schemes, like RSA [28], Diffie-Hellman [29] and the Digital Signature Algorithm [30] use long integer modular exponentiation. A key difference with RSA is that the modulus is the product of two primes, this allows us to use the Chinese Remainder Theorem (CRT) to increase the speed of private key operations [31]. Since these schemes use numbers greater than a 64-bit unsigned integer can handle, 2^{64}, they rely on *big number*, or *bignum*, libraries that provide methods to complete mathematical operations on an array of unsigned integers. We use our knowledge of the internal representation of big numbers to generate optimised checks, i.e. with as fewer reads as possible, for RSA. OpenSSL [23] and GnuTLS [26] are key libraries that provide an encrypted channel over a network, including RSA. OpenSSL has its own, built in, API for big number arithmetic, the `BN_*` functions, while GnuTLS uses the GNU Multiple Precision Arithmetic Library (GMP) [27]. Both of these consist of a contiguous array of unsigned 32-bit integers as explained in section IV.

B. Where are the keys?

The array of integers used, as we have mentioned, can expand thus they cannot be kept on the stack of a process; consequently the search space decreases substantially as we need only look at the heap of a process. On a Linux distribution, this means finding and interpreting the `mm_struct` structure for each process. This structure informs the user where the boundaries are within a process's virtual address space. The heap is located in between the two attributes: `start_brk` and `brk`.

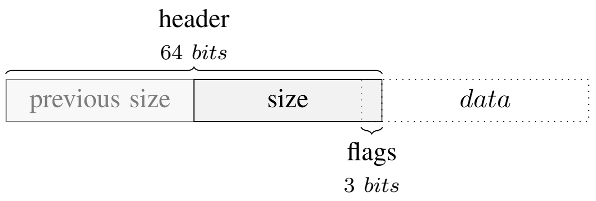


Fig. 5: An allocated heap entry in Linux.

The heap itself is managed by the process's memory management. There are other allocations libraries like jemalloc [32], etc.; however we are aiming to target applications that use the standard allocation provided by glibc. This mechanism involves a simple doubly linked list, whereby the header contains the previous entry's size, the current entry size and some flags. This is illustrated within Figure 5. The header itself is only two 32-bit integers in length. Once at the start, in our case `start_brk`, we can easily traverse the HEAP without access to the internal structuring of the memory manager. We can move to the next element by adding the size to our current position in memory, allowing fast traversal for analysis.

C. A Structured Walk to Find Key Material

The HEAP Primitive: As the key material is being kept within the heap, we can iterate through each element and find *interesting* features that could be construed as key material. In order to show this we provide a formalism. Suppose that $H = \{h_1, h_2, \ldots, h_n\}$ denotes the heap and its elements. The RSA cipher is generally used with three key lengths, specifically 1024, 2048 and 4096-bits. We denote this by L, where $L \in \{1024, 2048, 4096\}$

The Modulus: all elements within H are of a particular length, such that they can hold r bytes of data. The value of r can be found within the header of the entry. Assume that the length of a heap entry, h_i, is denoted by $len(h_i)$. When dealing with RSA, for any given key length L, the candidate modulus values reside in the following set of heap elements with the length L.

$$N_L = \{h \mid h \in H, len(h) = L\} \tag{1}$$

The Primes: The modulus itself is the product of two primes, p and q. These primes are often stored within the private key as they can be used to speed up signing and decryption. Signing and decrypting are computationally expensive operations and when used on public services any optimisations will allow a greater number of clients to use them concurrently. The length of a prime is half of the key length, as they both form the modulus. They are of equal length to make trial division as difficult as possible, as factoring is only as strong as its smallest factor [33]; therefore we need to find each heap element h_i that is half of the key length, i.e. $len(h_i) = \frac{L}{2}$. This can be used to reduce the search space within the heap. Particularly for larger keys as these sizes are less common in practice: BUFSIZ, valued at 1024 bytes, is a common buffer length for use with reading from the network and files. Primes can be found in the following set:

$$P_L = \{h \mid h \in H, len(h) = \frac{L}{2}\} \tag{2}$$

The elements in the above sets are in the form of 32-bit words. We denote the least significant word, of a potential big number, as $lsw(h_i)$ where h_i is the entry in the heap H. As the modulus n is the product of p and q, i.e. $n = p \times q$, the product of the least significant words of p and q are equal to the least significant word of n. For heap elements h_i, h_j and h_k we create a checking function f as follows:

$$f(h_i, h_j, h_k) = \begin{cases} \text{true} & \text{if } lsw(h_i) == lsw(h_j) \times lsw(h_k) \\ \text{false} & \text{otherwise} \end{cases} \tag{3}$$

Consequently $lsw(n) = lsw(p) \times lsw(q)$ then $f(n, p, q) = true$.

Using Prime Numbers 2, 3, ..., 23 reduces the search space by 83.61%. Consider the prime numbers 2, 3 ..., 23. If we multiply them we end up with the largest multiplication of all consecutive prime numbers which fit within a 32-bit word, i.e. $\lambda = 2 \times 3 \times \ldots, \times 23 < 2^{32}$. For any candidate prime number of the heap, if we calculate the remainder of h by λ, i.e. $h = \lambda \times q + r$ and if r has a factor of 2, 3, ..., 23 then clearly h is not a prime number. This simple test has two advantages. Firstly dividing a *big number* by a single 32-bit number is very fast. Secondly using this method will filter out 86.61% of possible numbers. To see this assume that for $i \in I = \{2, 3, \ldots, 23\}$, A_i denotes the event of h being a multiple of i. Then the probability of h being a multiple of 2, 3, ...23 is the same as $\mathcal{P}(A_2 \cup A_3 \ldots \cup A_{29})$ which using the inclusion-exclusion principle is:

$$\sum_{i \in I} \mathcal{P}(A_i) - \sum_{i<j \in I} \mathcal{P}(A_i \cap A_j) - \sum_{i<j<k} \mathcal{P}(A_i \cap A_j \cap A_k) \ldots$$

For A_2 half of the numbers are multiples of 2, so $\mathcal{P}(A_2) = \frac{1}{2}$, similarly $\mathcal{P}(A_3) = \frac{1}{3}$, the following probabilities can be computed similarly. The above sum can be calculated as 83.6%. A parity check as well as the above function is denoted as $is_p(h_i)$ where h_i is the heap element to be checked.

Entropy: A final characteristic of n is that it has comparatively high entropy. This is due to how the primes p and q are generated. They are formed by gathering random data and populating the big number array, followed by incrementing or decrementing until a safe prime is found [34]. This can narrow substantially the search space as quite often pieces of memory will store textual information, or at least data with a low entropy. For each key length we have a maximum and minimum entropy value, denoted as E_L^{min} and E_L^{max}, which can be pre-calculated based on test certificates. For our implementation we used Shannon's entropy as it is commonly used in this context. If X is a stream of characters produced from characters $\{x_1, x_2, \ldots x_n\}$. Then Shannon's entropy is calculated by $E(X) = -\sum_{i=1}^{n} p(x_i) log_b p(x_i)$ where $p(x_i)$ is the probability of x_i appearing in X. In our case there are only 256 possible values for a byte; this allows us to calculate $p(x_i)$ as we need only count how many times a particular byte appears within the window.

The entropy of a heap element h is calculated by the method $E(h)$. With this we can now have a set of potential (n, p, q)-tuples, as show in (4) found in Figure 6, we denote this set by G_L. The only remaining pieces of information is with regards to the private exponent d, and the public exponent e. e is used along with the two primes in order to find the multiplicative inverse that derives d. This value however is usually made small enough to reduce the speed of encryption, and large enough to prevent some feasible attacks [35]. It is also made constant in a few libraries that implement RSA, and can be part of standards [36].

$$\begin{aligned} G_L = \{(n, \{\{p, q\} \mid \{p, q\} \in [P_L]^2, is_p(p), is_p(q)\}) \mid \\ n \in N_L, E_L^{min} \leq E_L(n) \leq E_L^{max}, f(n, p, q) = \text{true})\} \end{aligned} \tag{4}$$

Fig. 6: Partial key generation material.

The Public Exponent This provides us with the basic RSA key material used when finding the multiplicative inverse to derive a private exponent, d. The public exponent, e is often 65537 and has been omitted from the formalism. In order to maintain the security and keep the speed of decryption and verification, implementations of RSA generally use the value 65537 for their public exponent, e. It is common in OpenSSL, GnuTLS and others to use this value by default, the user actively has to specify a different value. 65537 is used as it is a Fermat prime, specifically $F_{[4]}$, and is chosen as it makes the modular exponentiation faster, whilst being large enough to decrease the vulnerabilities caused by lower primes. This gives us a final set of candidate private keys from memory. With this assumption, the private exponent can be calculated externally.

D. Candidate Private Exponents

Depending on resources one could return the values within G_L and have the command and control, an external server, to handle the overhead of generating the actual key. This would involve the assumption of e being 65537. One could also return all values of candidate d for verification; however there are few characteristics for this value: $D_L = \{d \mid d \in N_L, E_L^{min} \leq E_L(d) \leq E_L^{max}\}$ To narrow this set we can take advantage of the two primes from the key generation. The private exponent is computed as the modular inverse of e such that: $d \times e = 1 + k \times (p-1) \times (q-1)$. However, we can say that each prime minus one has a number of trailing zero bits: $p - 1 = p' \times 2^i, q - 1 = q' \times 2^j$, such that, for both primes, p' and q' are odd and i and j are greater than 0. $2^{i|j}$ acts as a bit shift to the left, which means there will be a minimum of $i+j$ trailing zero bits. k can also be noted in the same manner; however

```
Input: A set $H$ that contains elements of a process' heap and a
       number $L$, which represents the key length.
Output: A set of three tuples containing RSA key material
        (public modulus, and two primes).
1  $N_L \leftarrow \emptyset$
2  $P_L \leftarrow \emptyset$
3  $G_L \leftarrow \emptyset$
4  begin
       /* filter for moduli and primes */
5      for $h \in H$ do
6          if len($h$) $= L$ then
7              $N_L \leftarrow N_L \cup \{h\}$
8          else if len($h$) $= \frac{L}{2}$ then
9              $P_L \leftarrow P_L \cup \{h\}$
10     end
       /* combinations of key material */
11     foreach $\{(n, \{\{p, q\} \mid \{p, q\} \in [P_L]^2\}) \mid n \in N_L\}$ do
           /* keep those tuples where n = p . q
              could match */
12         if lsw($p$) $\times$ lsw($q$) $=$ lsw($n$) then
13             $G_L \leftarrow G_L \cup \{(n, p, q)\}$
14         end
15     end
16     foreach $\{(n, p, q) \mid (n, p, q) \in G_L\}$ do
17         if not is$_p$($p$) then
18             $G_L \leftarrow G_L \setminus \{(n, p, q)\}$
19         else
20             if not is$_p$($p$) then
21                 $G_L \leftarrow G_L \setminus \{(n, p, q)\}$
22             else
                   /* is entropy in range */
23                 if $E_L^{min} >$ E$_L$($n$) or E$_L$($n$) $> E_L^{max}$ then
24                     $G_L \leftarrow G_L \setminus \{(n, p, q)\}$
25     end
26     return $G_L$
27 end
```

Algorithm 1: Finding Key Material

the shift is unknown and can provide no information to our calculation, thus we can only determine the minimum number of trailing zero bits. We can now rewrite this as: $d \times e - 1 = k \times p' \times q' \times 2^{i+j}$. If we were to calculate the LHS, once again assuming e was equal to 65537, using each of our candidate d values, our answer must have at least $i + j$ trailing zeros. As a result, we can drop any candidate private exponents that do not have these trailing zeros.

VI. Sketch of the Implementation

We define our method in Algorithm 1. Here you can see the steps defined in our method above. Firstly, between lines 5-10, we store elements that are of the correct length for key material. The modulus values are the length of the key length and the primes are half of that. Following this we cross multiply each candidate prime's least significant word, this will be equal to the least significant word of the modulus (lines 11-15). We then narrow this list by removing elements where the primes are not considered prime (on lines 17 and 20, by our method discussed in subsection V-C), and followed by an entropy check at line 23.

VII. Evaluation

In order to evaluate our methods we wrote an application that could be used to form the search module of an FVM. The application completed the tasks as per subsection V-C.

A. Environment

For these tests we used two machines:

1) A MacBook Pro (Late 2013) with a quad-core Intel i7 and 16GB of memory. This machine provided our platform for virtualisation using VMWare Fusion[2].
2) A HP EliteBook with a dual-core i7 and 4GB of memory. This machine was used to create traffic using siege [37]. This traffic allows the services, on the virtual hosts, expand as the load increased and thus made more use of a process' heap.

Rather than managing a hypervisor like Xen [18] and using the live features of libvmi [15], we chose to find keys within snapshots of a VM. Libvmi works on memory snapshot files in the same manner as a live guest machine. The hypervisor used was VMWare Fusion, we used this as we could automate the creation and restoration of snapshots, and also automate the upload of key material using their command line tools.

The VM was an Ubuntu 12.04-4 Server with 512MB of RAM and two of the i7's cores. 512MB was used as we needed to snapshot each iteration of a test. Any larger and we would have come across storage issues. The VM was then updated such that all, packaged, security and performance upgrades had been applied. Initial tests are mainly aimed at the use of Apache HTTPD [38], a common web server used today, and two different SSL/TLS libraries (OpenSSL and GnuTLS). These were also installed and configured, particularly their Apache HTTPD modules (`mod_ssl` and `mod_gnutls`).

B. Upper and Lower Bounds for Key Entropy

As we mentioned in subsection V-C, RSA keys have high entropy. This is due to the way the primes, p and q, are formed. The acquisition of random data from a source populates the buffers and then are incremented/decremented to find a probable prime. To determine this level of entropy that characterises RSA key material, we generated 2000 RSA keys in parallel for all key lengths (L). These were created using GNU Parallel [39] and the OpenSSL command line tools. We parallelised the creation as each command would provide randomisation for the subsequent and concurrent commands. The default value for the public exponent, e, i.e. 65537, was used as it is a common choice without specifying otherwise.

The resulting keys were then parsed to determine the Shannon entropy of the private exponent and modulus of the key with length L. The entropy is displayed in Table I and define E_L^{min} and E_L^{max} used within checks for the RSA modulus, n. These keys gave us the entropy range that we believe characterised the private exponent and the modulus of RSA. In order to test this we generated a new set of 10000 keys, in the same manner as before, and verified this hypothesis.

[2]Available: http://www.vmware.com/products/fusion

key length	entropy	
	E_L^{min}	E_L^{max}
1024	4.36853729	4.70040432
2048	4.81762365	5.11726980
4096	5.16982622	5.34433150

TABLE I: The Shannon entropy for keys of different length.

Our minimum and maximum values gave a 0.09% error in the case of 1024-bit, 0.01% in the case of 2096-bit and 0.04% for 4096-bits. The tests themselves, i.e. the ability to find keys within a virtual hosts memory used another unknown set of 2000 keys for each key length.

C. A Single RSA Key

A small business may only serve one website from their web servers so we designed this test to emulate this characteristic. Our instance of the Ubuntu VM contained a single Apache HTTPD virtual host, serving clients using a 2048-bit key. We ran 200 tests for each of the two libraries that we were analysing, GnuTLS and OpenSSL. Each test had a different RSA key pair, randomly selected from our library of RSA keys.

Apache HTTPD spawns new processes to handle greater loads on the system be able to spread over more of the servers resources. This meant we found many duplicate keys as each process contained the same configuration obtained from the parent process. Removing these duplicates our program managed to find all the keys, using both libraries, within memory with no false positives. We were able to use our derived keys to decrypt data encrypted by their original counterparts.

D. Multiple RSA Keys of the Same Length

Our second test was aimed at emulating a web host. A web host may allow its users to upload their own keys which means their servers will contain references to multiple keys within the system. In order to see if there would be any conflicts or false positives we needed to find multiple keys in a single process. This may happen as there is more material with similar characteristics that could combine and seem valid using our checks. This test required four keys of 2048-bit length and therefore four virtual hosts (using different ports). Siege [37] was amended to take these additional URLs into account when bombarding the services with traffic. This time we looked for 500 of our 2048-bit length keys, randomly selected from our library, making 125 iterations.

Once again, Apache HTTPD expanded in order to maintain performance for its clients whilst maximising the use of each processes' resources to handle each request. Our tool found all keys for both libraries with no false positives.

E. Multiple RSA Keys of Different Lengths

The application we have written, in the form of an FVM's search code, can only look for one key length at a time. This allows a search to concentrate on looking for a single length of key, using standard and statically compiled buffers: in turn optimising the program for that key length. All key lengths have the same process placed upon them for detection, there are few variables that change, particularly the lengths of p, q and n and therefore the minimum and maximum entropy of n.

Vendors may allow clients to upload their own SSL/TLS certificates, these need not be of the same length, as per the previous test. To show this we repeated our previous test to allow 500 randomly selected keys, 125 iterations, of different key lengths (1024-bit, 2048-bit and 4096-bit). For each test, we ran each version of our program on the resultant snapshot to find the keys within. Removing duplicates, we found all keys of all lengths and found no other potential keys.

F. Other Applications

Our tests have focused upon a well used web server, as this may be seen as the main entry point for attack via SQL injections as they are generally public facing. Which in turn, makes them more likely to have rules assigned to their traffic within an IDS. To test the validity of our approach we tested it against other applications. Particularly of note we found the RSA key from the SSH daemon running on the Ubuntu instance in all tests.

Other, more cursory, checks were made on applications that use SSL/TLS via these two libraries using the RSA asymmetric cipher. Our program worked upon services that used nginx [40] and node[3]. This was due to them directly using the OpenSSL library.

We also believe that the Apache Tomcat server, a Java web server, would also be open to this analysis when using Tomcat's Native Library, as it uses OpenSSL via the Java Native Interface (JNI). However Java does not use the standard UNIX *brk* for heap allocation and instead uses its own allocation mechanism via the JVM.

G. Performance

To illustrate performance we show the number of reads, from the VM, that is necessary to gather key material in Table II. There are three processes, two Apache2 processes and an SSH daemon. The first Apache2 process contains four RSA keys on different virtual hosts, the others only have one. In order to find signatures using Klein's approach [11] or indeed the sliding window of entropy one must read the total heap size, minus the expected length of the certificate or the length of the key. Iterating through heap elements reduces the number quite substantially.

VIII. Discussion and Related Work

A. Cross Compatibility

There are of course multiple platforms that need to be supported so questions into the feasibility of handling them all is up for discussion. For VMI to succeed, efforts need to be in place to create wrapper APIs to navigate these common features, like the process table, raw heap elements and file descriptors. This is a reverse engineering task for some OSs,

[3]Available: http://nodejs.org

Process Name	Heap Size (in bytes)	Number of HEAP elements	Candidate Moduli	Candidate Primes	P is Prime?	Q is Prime?	Entropy Checks	Total # of Reads (in bytes)
(1) Apache2	937984	6776	28	37	7	7	7	30948
(2) Apache2	897024	6110	5	7	1	1	1	25000
(3) SSHd	21000	703	5	10	1	1	1	3384

(1): An Apache process with four RSA Certificates (three duplicates found)
(2): An Apache process with one RSA Certificates
(3): An SSHd service running with an RSA Certificate

TABLE II: The number of reads necessary to find keys using Algorithm 1.

but much work has been completed on the structure of some Windows instances for use with Volatility [14]. Libvmi starts by finding the OS's core kernel structure allowing applications to traverse down and find information; it would be unwise to repeat this work.

Our approach allows for traversal of a Linux distribution, using glibc's native heap management. The default allocator in most cases for applications. There are others including Lockless[4], Google's tcmalloc[5] and jemalloc [32] for use within applications, as well as the Windows HEAP allocator. All of which will have different structures to navigate through, but when VMI is used within the Cloud we need optimised searches to prevent starving the guest VM of resources, at least for continuous monitoring which the FVM attempts to do [6], [7].

B. Controversy

This technology is open to abuse as it can be used by anyone who has access to the raw memory of a VM. Ensuring correct usage of this technology by Cloud providers is an area for experts in privacy and law. Further work is required to create a framework for the legitimate and legal use of this technology. This is an area which is out of scope of this paper and remains open for future research.

C. A Different Public Exponent

It is unlikely that the RSA key uses a different value for e, however this is still possible. Particularly for those running older versions of cryptographic libraries or even if the certificate creator specified a value. In the former case, historically e could have been any of the previous Fermat numbers, $\mathbb{F}_{[i]}$ where $0 \leq i < 4$, i.e. 3, 5, 17, 257. The latter gives the creator the ability to use any number where $1 < e < \phi(n)$ and $gcd(e, \phi(n)) = 1$ (coprimality) applies. This means e has no real distinguishing features that can be ascertained without a big number library built into the FVM, which would increase the footprint of the sensor. We cannot look for a specific heap length as this may differ. We do however have the candidate private keys. This means if we report both G_L and D_L, we can brute force the historical case.

In either case our tool may give an indicator that some RSA encryption is going on within the processes, virtual address space. G_L on its own is not enough for encryption or decryption, we need to know the values of the private exponent in order to continue; therefore we need to know e as it is used to derive it.

IX. Conclusion

We present an algorithm for a structured walk through a VM's memory to find RSA keys. The use of VMI can be quite expensive, particularly since there is a large semantic gap between the raw representation that VMI provides and that of native OS APIs. Our method addresses this by carrying out a structured walk over key elements within memory reducing the interaction between the application that is looking for keys and the guest VM that is being introspected. Iterating the raw heap and implementing small checks allow for a steep reduction in *potential* key material, thus data can be ignored.

References

[1] M. Tanase. (2001). The future of IDS, [Online]. Available: http://www.symantec.com/connect/articles/future-ids (visited on 06/29/2014).

[2] S. Kumar. (2007). Survey of Current Network Intrusion Detection Techniques, [Online]. Available: http://www.cse.wustl.edu/~jain/cse571-07/ftp/ids.pdf (visited on 06/29/2014).

[3] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, 2003.

[4] 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*, 2005.

[5] M. Keshavarzi, "Traditional host based intrusion detection systems' challenges in cloud computing," *Advances in Computer Science: an International Journal*, 2014.

[6] K Harrison, B Bordbar, S. T. T. Ali, C. I. Dalton, and A Norman, "A Framework for Detecting Malware in Cloud by Identifying Symptoms," in *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, 2012.

[4] Available: http://locklessinc.com/
[5] Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[7] A. L. Shaw, B. Bordbar, J. T. Saxon, K. Harrison, and C. I. Dalton, "Forensic virtual machines: dynamic defence in the cloud via introspection," in *IEEE International Conference on Cloud Engineering*, 2014.

[8] Symantec. (). Trojan.Zbot.B, [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2010-100908-2548-99 (visited on 01/16/2014).

[9] H. w. Baek, A. Srivastava, and J. V. d. Merwe, "Cloud-vmi: virtual machine introspection as a cloud service," in *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, 2014.

[10] 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*, 2009.

[11] T. Klein. (2006). All your private keys are belong to us, [Online]. Available: http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf (visited on 07/10/2014).

[12] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: issues, security threats, and solutions," *ACM Comput. Surv.*, 2013.

[13] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)," *NIST special publication*, 2011.

[14] V. Systems. (). Volatility, [Online]. Available: https://www.volatilesystems.com/ (visited on 01/16/2014).

[15] B. Payne. (). Virtual machine introspection tools, [Online]. Available: https://code.google.com/p/vmitools/ (visited on 01/16/2014).

[16] 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 *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012.

[17] F. Westphal, S. Axelsson, C. Neuhaus, and A. Polze, "Vmi-pl: a monitoring language for virtual platforms using virtual machine introspection," *Digital Investigation*, 2014.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, 2003.

[19] S. Popuri. (). A tour of the Mini-OS kernel, [Online]. Available: http://www.cs.uic.edu/~spopuri/minios.html (visited on 08/03/2014).

[20] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2.," RFC Editor, Tech. Rep., 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5246.txt.

[21] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology CRYPTO 85 Proceedings*, Springer Berlin Heidelberg, 1986.

[22] J.-J. Quisquater and C. Couvreur, "Fast decipherment algorithm for rsa public-key cryptosystem," *Electronics letters*, 1982.

[23] OpenSSL Project. (2014). OpenSSL: Documents, rand(3), [Online]. Available: https://www.openssl.org/docs/crypto/rand.html (visited on 07/12/2014).

[24] T. Pettersson, "Cryptographic key recovery from linux memory dumps," Chaos Communication Camp 2007, 2007, [Online]. Available: http://events.ccc.de/camp/2007/Fahrplan/events/2002.en.html.

[25] A. Shamir and N. van Someren, "Playing hide and seek with stored keys," in *Financial Cryptography*, Springer Berlin Heidelberg, 1999.

[26] GnuTLS Project. (2014). GnuTLS, [Online]. Available: http://gnutls.org (visited on 06/06/2014).

[27] ——, (2014). The GNU Multiple Precision Arithmetic Library, [Online]. Available: http://gmplib.org (visited on 06/06/2014).

[28] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, 1978.

[29] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, 1976.

[30] D. W. Kravitz, "Digital signature algorithm," U.S. Patent US5231668 A, 1993.

[31] J. Groβchädl, "The chinese remainder theorem and its application in a high-speed rsa crypto chip," *Computer Security Applications Conference, Annual*, 2000.

[32] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[33] A. K. Lenstra, "Key lengths," Wiley, Tech. Rep., 2006.

[34] OpenSSL Project. (2014). OpenSSL, [Online]. Available: http://www.openssl.org/ (visited on 06/06/2014).

[35] P.-A. Fouque, S. Kunz-Jacques, G. Martinet, F. Muller, and F. Valette, "Power attack on small rsa public exponent," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, Springer Berlin Heidelberg, 2006.

[36] D. Crocker and K. M. Hansen T., "DomainKeys Identified Mail (DKIM) Signatures," RFC Editor, Tech. Rep., 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6376.txt.

[37] J. Fulmer. (2014). Seige, [Online]. Available: http://www.joedog.org/siege-home/ (visited on 08/03/2014).

[38] Apache Software Foundation. (). The Apache HTTP Server Project, [Online]. Available: http://httpd.apache.org/ (visited on 08/03/2014).

[39] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, 2011.

[40] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux J.*, 2008.