University of Amsterdam

Faculty of Physics, Mathematics and Informatics

MSc System and Network Engineering

Reliable Library Identification Using VMI Techniques

Nick de Bruijn, Leandro Velasco

Abstract

For cloud providers it is important to offer services that safeguard their users from existing vulnerabilities. Research has shown that is not uncommon for libraries to contain vulnerabilities that can have serious security implications. Traditional host based vulnerability scanners can be used to identify such vulnerable libraries. However, these scanners require the user to install and maintain the software. Our research explores the feasibility of implementing a reliable library identification scanner based on virtual machine introspection (VMI) techniques provided by LibVMI, which would not require such user intervention. We start by creating a program that combines the VMI techniques to extract a running library from a virtual machine's memory with an implementation of a library identification method based on all the printable strings contained in the library's binary. We then test the accuracy of our program and evaluate its performance by doing measurements of several indicators under different system loads. Our experiments show that our method can extract and accurately identify libraries within a few milliseconds.

April 15, 2017

Contents

1 Introduction

Cloud services offer platforms for running applications and entire infrastructures. In the context of infrastructure as a service (IaaS) this is typically done by allowing users to create Virtual Machines (VMs) which can be configured to accommodate business needs.

The VMs might be connected to the internet which exposes them to several security risks. In order to minimize these risks, it is important to take preventive measures so as to detect vulnerabilities.

Research has shown that it is common to find vulnerabilities in libraries installed in the system. The OpenSSL heart bleed [7], OpenSSL Null pointer assignment [2] and the glibc getaddrinfo() stack-based buffer overflow [1] are examples of such vulnerabilities. Therefore, it becomes relevant for cloud providers to protect their users against these kind of threats.

A solution to detect vulnerable libraries is using vulnerability scanners. These scanners identify running libraries on the user's VM and check if the library version is known to have vulnerabilities. Traditionally, these scanners have to be installed on the user's VM meaning that every VM on the cloud provider should have this piece of software running. Furthermore, the user is responsible for installing and maintaining the host vulnerability scanner.

Another alternative could be that cloud providers would offer vulnerability scanning as a service. In order to do so, a scalable and transparent solution capable of scanning VMs should be implemented.

A method that could be used to enable scanning of VMs is known as virtual machine introspection (VMI) [5]. VMI provides a mechanism to analyze VMs from the hypervisor. During the last years researchers have been studying how Virtual Machine Introspection (VMI) methods could be used to design security solutions for cloud environments. Examples of its possible applications are intrusion detection and malware analysis systems.

Krishna and Ricci implemented a vulnerability scanner using VMI techniques [9] which allows detection of running libraries by accessing the VM's memory. Their proposed method is based on identifying libraries by extracting their version numbers from the name they present in memory. However, this approach does not take into account that library names might not contain version information or that version numbers could be tampered with.

In this report, we propose a method based on VMI techniques, to reliably identify a running library in a VM where the library names can not be trusted. The idea is to extract the library binary from memory and use all the available printable strings to identify the library.

To measure the efficiency and effectiveness of our proposed method we perform performance tests that measure the library binary extraction times, identification time, and the identification accuracy rate under different system loads.

1.1 Research Question

Our main research question is: *To which extent library versions can be reliably identified using the VMI techniques provided by LibVMI?*. In order to guide this research we have identified the following sub-questions.

- What approach needs to be taken to extract library information from the VM's memory by using LibVMI?
- Which approaches can one use to identify the version of a running library that does not rely only on version numbers in library names?
- What are the performance implications of running such a library scan?

2 Related Work

Over the last decade VMI has been extensively researched and used for different security applications. In this section we will mention the work that is more relevant to our research project.

In 2003 Tal Garfinkel et al. presented the method known as virtual machine introspection as a technique to analyze VMs from the hypervisor [5]. This paper shows an architecture that retains the visibility of a host-based Intrusion Detection System (IDS), but pulls the IDS outside of the host for greater attack resistance. The proposed method shows that VMI is a difficult process and hard to address.

In 2012 Bryan D Payne introduced LibVMI which is an open source software project that aims at simplifying VMI [11]. Later, Xiong et al. presented a paper [18] that shows how LibVMI can use the hypervisor to translate and interpret the binary representation of the virtual memory used by the OS of a VM.

The most relevant work is the one presented by Krishna and Ricci. These researchers implemented a vulnerability scanner using stackdb[1] as a VMI library [9]. Their proposed method identifies libraries by extracting their version numbers from the library name or from the library executable present in the virtual memory. This means that instead of using the complete library executable code to identify a library, they only use a small part of the library executable for the identification. Moreover, this approach does not take into account that library names might not contain any version information, and that version numbers can be tampered with. Therefore, the method is not capable of reliably identifying libraries under the aforementioned circumstances. Furthermore, this research points out that their method needs to pause the VM for at least three seconds during version scanning, which has a direct impact on the performance.

In 2017 Thomas Rinsma researched about automatic library version identification [13]. In his research he explores different ways of identifying library versions in Linux and tests their effectiveness, accuracy and speed. He concludes that the method based on printable strings is the most suitable candidate for identification.

In our research we try to overcome the limitations of the work done by Krishna and Ricci, by combining VMI techniques with the library identification method based on printable strings, which is not based on the availability of version numbers.

[1] https://www.flux.utah.edu/paper/johnson-vee14

3 Background

3.1 System Virtualization and Instrospection

System virtualization allows one to run multiple isolated systems on the same physical hardware. These systems are known as virtual machines (VMs) [6]. For the physical hardware to host VMs an additional software layer is required which is known as the hypervisor or virtual machine monitor (VMM) [3]. This component is implemented in between the physical hardware and the hosted VM and its purpose is to provide and allocate resources to the VMs.

3.1.1 Virtual Machine Introspection

VMI is a method to monitor and analyze VMs from the hypervisor [5]. The hypervisor is responsible for allocating physical resources to the VM and aiding the Operating System (OS) execution. It has access to the internal state of the VM, including the binary representation of the virtual memory used by the OS running inside the virtual machine [14].

Nonetheless, the hypervisor has no knowledge about the semantics in this binary representation because the OS running inside the virtual machine is in charge of the usage and management of this memory. The lack of knowledge from the hypervisor's perspective is called the semantic gap [18]. As a consequence, the semantic gap needs to be bridged before the hypervisor can be used to introspect VMs.

3.1.2 LibVMI

LibVMI[2] is a VMI library developed to simplify virtual machine introspection, based on XenAccess [12]. The library provides an application programming interface (API) for reading and writing to a virtual machine's memory. However, before the API can interpret the VM's virtual memory, the semantic gap should be overcome. In order to so, LibVMI requires the guest's OS type, the location of symbolic information, and the offsets that will be used to access data within the virtual machine. This information needs to be provided for each VM that one wants to introspect as the data can differ from VM to VM.

The location of the symbolic information is a path to the system.map[3] file of the VM. The system.map file is used by LibVMI to translate kernel symbols to kernel addresses and vice versa. An illustration of LibVMI's inner workings is shown in Figure 1.

[2] http://libvmi.com/

[3] http://www.dirac.org/linux/system.map/

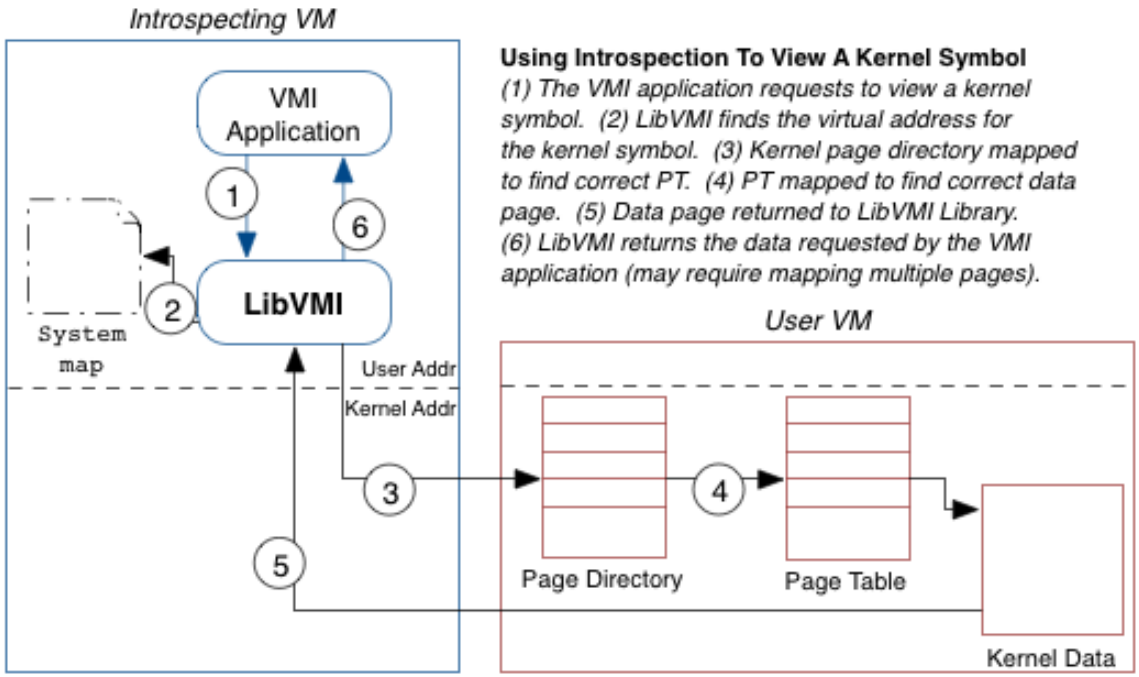

Figure 1: LibVMI memory mapping[4]

3.2 Library Identification

By combining known library identification techniques and VMI techniques, efficient and powerful library identification tools can be created for cloud environments. In this section we take a closer look at different library identification techniques. We discuss three approaches: version number extracting, behaviour based identification and signature identification.

3.2.1 Version Number Extracting Identification

Version numbers can appear in library names and virtual memory page contents. To extract the version number from library names, one can list all shared library names linked to a particular process in a file. Library names usually follow a strict structure which includes the major and minor versions. An example is libc-s2.23.so where 2 is the major version and 23 is the minor one.

For libraries that do not contain any version number in the filename, a dump from memory can be made to extract the binary representation from the library. Usually, the version number is stored as a string in the executable file. These version numbers can then be parsed from the dumped memory and used to identify the exact library version.

However, this method has a limitation as it completely relies on library names for identification. This approach does not take into consideration that version names can be tampered with. In addition, some libraries, such as libz, and rpcbind, do not include version numbers in their names or binary, which would render this method ineffective [9].

[4]http://libvmi.com/docs/gcode-intro.html

3.2.2 Behaviour Based Identification

Behavior based identification consists of looking at how a library interacts with the OS. The interaction between a library and the OS can be determined by observing which system calls are invoked. However, most libraries do not invoke system calls directly, but they use wrapper functions provided by standard system libraries instead. E. Jacobson, N. Rosenblum and B, Miller propose a method for identifying libraries based on wrapper functions [8]. The idea is to capture the high-level semantics of wrapper functions from the binary representation of the library. These high-level semantics are the system call names and arguments invoked by the wrapper function. Semantic descriptors are used to turn these high-level semantics into fingerprints. To match their semantic descriptor fingerprints, they use a reference set of fingerprints of different library versions.

3.2.3 Fingerprint Identification

This approach proposes the generation of fingerprints on libraries, which can then be used as an identification method. A straightforward solution to accomplish this, is to extract the complete binary representation of the library and use a hashing algorithm to create a digest. The hashed values can be used to compare libraries. However, there are two challenges that need to be overcome. The first challenge is that the binary of a library is affected by the way it is compiled. This means that several different binaries can be obtained from the same library. The second challenge is that building the binary in different environments, for example under different operating systems, also affects the binary representation. As a result, for every possible variation a signature must be created. This is the reason why more efficient approaches exist to create fingerprints based on digests.

Wildcards can be used to overcome some of the limitations that a signature based approach imposes. Such an approach is presented by M. Van Emmerik in [17]. He proposes to extract the functions used in the library's executable and create signatures for every derived function. In addition, to minimize the influence of the compiler or environment, fixed wildcards are used. These wildcards replace the variable values. To keep the signatures small, only the first n bytes of the function are hashed. The hashes of the functions can be linked together to create a graph which represents a library. This method overcomes the problem of changing variables that would lead to different signatures. However, it has its limitations. On one hand, only the first n bytes of the functions are used which could mean that libraries using the same functions are not distinguished from each other. On the other hand, placing wildcards in the right place is cumbersome. These situations might lead to false-positives.

Thomas Rinsma worked on another method to identify library versions based on matching control flow graphs (CFG). CFGs are originally introduced in [4] to classify malware, but can also be used to identify library versions. The method creates control-flow graphs of executable objects. The control flow is a representation of the execution path of a program. This method requires one to disassemble the objects of the executable to create CFCs for all the functions within the object. Knowledge of the instruction set and calling conventions are used in order to build a directed graph of the control flow [13]. Automated tools like IDA Pro[5] and Unstrip[6]

[5]https://www.hex-rays.com/products/ida/

[6]http://www.paradyn.org/html/tools/unstrip.html

are available for this task. The paper presented by Emily R Jacobson et al. [8] shows that both tools can be used to accurately identify library versions. However, this technique faces scalability issues. These graphs need to be created for each reference file and each extracted library executable, and possibly for each compilation of the same version by different compilers.

The last method we will discuss is identification based on printable strings. Thomas Rinsma shows in his research that this method can be used to accurately identify running libraries [13]. It is based on the techniques presented in [15] to identify malware families. It looks at the printable strings that can be extracted from an executable and compares them against a reference string set. Examples of strings are error messages and copyright or usage information.

Linux provides a standard tool named **strings**[7] that is capable of extracting these printable strings. An example snippet of the output of this command can be seen in Figure 2.

```
setrpcent
__progname
mbrtoc32
_IO_free_backup_area
creat
setnetent
wcschr
__strxfrm_l
posix_spawn_file_actions_addclose
argp_err_exit_status
getgrgid_r
__vfwprintf_chk
unshare
_seterr_reply
__recv_chk
_IO_getline_info
__fwriting
__finitel
_itoa_lower_digits
inet6_opt_finish
```

Figure 2: Printable String Output

Whenever extracting the printable strings from both a sample library and the reference library, the similarity of these two printable string sets can be calculated. To do this calculation the so called Jaccard Similarity Coefficient can be used. This indicator is obtained by calculating the overlap ratio between the two sets as depicted in Figure 3.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

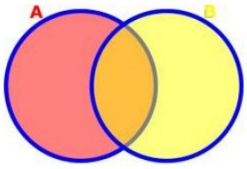

Figure 3: Jaccard Similarity Coefficient[8]

[7] https://linux.die.net/man/1/strings
[8] https://axialcorps.wordpress.com/category/datascience

4 Methodology

The literary analysis phase has resulted in the election of a mechanism for extracting information from a VM and a library identification method. By combining the two, we designed a program that identifies a library running on a guest VM by using the printable strings it contains.

The second phase of this research project consists of the implementation of such a program, and the analysis of its effectiveness and efficiency. In order to measure these aspects, several experiments were designed and executed.

4.1 Experimental Environment

The experimental environment shown in Figure 4 consisted of a privileged system (Domain0) responsible for running the introspection program and a guest VM (DomainU) that was introspected. These two systems were running on a Dell PowerEdge server with Xen Hypervisor installed. It is important to mention that the Dell servers used during this project have a four core hyper-threaded Intel E3-1240L CPU that delivers a total of eight vCPUs. Two of these were allocated to the guest VM. Please refer to Table 1 for the specification details.

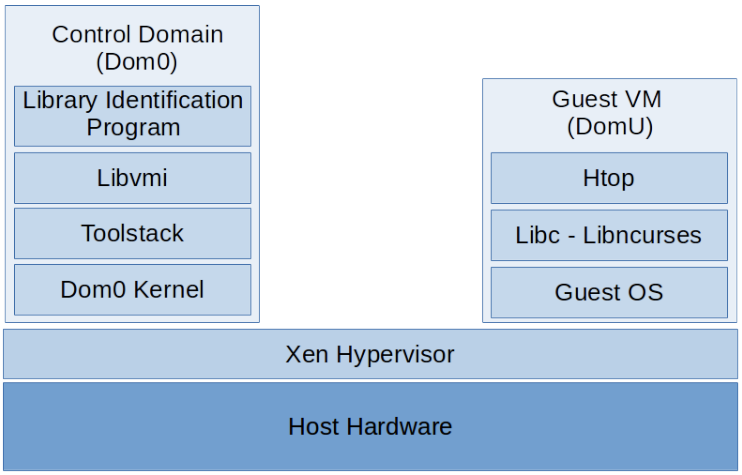

Figure 4: Experimental Environment

System	**Hardware Specification**	**Software Specification**
Server - Domain0	Intel E3-1240L 2.10GHZ CPU, 16GB RAM	Ubuntu 16.04, Linux Kernel 4.4.0-59-generic, Xen Hypervior 4.6, libvmi 1.15
Guest VM - DomainU	2 VCPU, 1024MB RAM	Para-virtualized Ubuntu 16.04

Table 1: Detailed System Specification

4.2 Program Implementation

The program was divided in two modules, the **library extractor** and the **library identifier**. The library extractor makes use of the LibVMI API to handle the guest VM memory and extract the pages where the selected library binary was loaded. Once the memory is dumped into a file, the library identifier takes control of the execution. First, it generates a fingerprint from the printable strings contained in the dump and then it compares the fingerprint against all the records of a reference database which is described in detail in Section 4.2.3.

It is important to note that the program was implemented as a proof of concept. This means that during the design and implementation some compromises were made. Examples of such are the program performance and the way that the reference database was built.

4.2.1 Library Extractor

Before explaining the inner workings of the library extractor, its is necessary to understand the way that the kernel data structures shown in Figure 5, relate to each other. Each process running on a Linux system is represented by a *task_struct* structure. This structure holds vital information about the process such as its PID number, parent PID number and a pointer to the memory descriptor. The memory descriptor represents the memory address space of a process which includes the virtual memory areas assigned to it. Each virtual memory area is represented by the *vm_area_struct* structure and contains properties like permissions, first memory address, last memory address and a set of associated operations. It is important to mention that all the virtual memory areas of a given process form a double linked list. There is one value in the *vm_area_struct* structure that has significant relevance for our project, and that is the pointer to the *file struct* structure. This pointer is used by the library extractor to identify the virtual memory area where the selected library was mapped [10].

When a shared library is linked to a process the runtime linker actually links three different segments. The data segment contains initialized static variables, the bss or Block Started by Symbol segment stores statically-allocated variables and the code segment has executable instructions [16]. To generate the fingerprint, we have focused on the code segment as it is the one that contains the executable instructions of the selected library. To be able to distinguish between these three segments we made use of the *vm_flags* value contained in the *vm_area_struct* structure. The segment with the execution flag enabled is the one with the library binary representation and therefore, the one that the library extractor needs to dump.

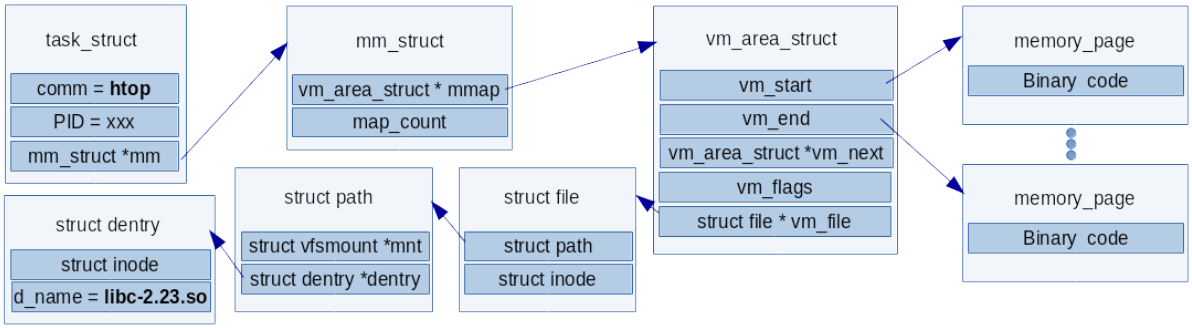

Figure 5: Kernel Data Structures

As mentioned before, the library extractor is in charge of identifying and dumping the guest

VM memory pages in which the selected library was loaded. To do so, it first pauses the VM; this is done to access its memory in a consistent way. Then it walks through the aforementioned kernel data structures until the right virtual memory area is found. This is, the virtual memory area where the selected library was mapped. Once the right segment is found, the program writes the contents of the memory pages into a file. As soon as it finishes writing to the file, it resumes the VM's execution and exits.

For more details about the implementation, please refer to Figure 13 in the Appendix, which contains the pseudo code of the library extractor. It is important to note that the specific offsets used by the program were obtained by running a customized kernel module on the guest VM system. This module calculates the offsets by using the memory addresses of the kernel data structures. In a production scenario this kernel module could be executed in a VM that has the same kernel version and compilation flags as the VM that is intended to introspect. By doing so we avoid the requirement of executing software on the customer VMs. In addition, this kernel module should be executed only when there is a change in the kernel that affects the data structures shown in Figure 5, such as a kernel upgrade.

To simplify the execution of the experiments described in Sections 4.3 and 4.4, we have used the name of the libraries as presented by the OS to indicate which library the library extractor needs to dump. However, this module was designed to be called for each library linked to a given process PID. In other words the name of the library is used as a mean of selection rather than identification.

4.2.2 Library Identifier

This module implements the chosen library identification method. First of all, the library identifier generates the fingerprint from the memory dump extracted by the library extractor. It does so by calling the Unix **strings** command. The output of this command is a file containing all the printable strings of a given input. In this context, the output of the strings command is what we call the fingerprint of the library. Once the fingerprint is obtained, the library identifier calculates the "match score" for each fingerprint in the reference database. The match score is calculated as shown in Figure 3. Finally, this module sorts the results of the calculation and then returns the top reference fingerprints along with their corresponding match score.

4.2.3 Reference Database

The reference database consist of 151 fingerprints generated from multiple versions of different libraries. To create this database we have downloaded the source code of the libraries listed in Table 2 and built them by only making use of the *"prefix=<Install Directory>"* argument. After installing the libraries in a custom directory we obtained 151 different shared objects. The reason for obtaining more shared objects than library packages downloaded is that glibc also installed auxiliary libraries such as *librt*, *libresolv* and *libutil*. Then for each shared object obtained in the previous step, we have generated the fingerprint as explained in the previous section.

It is important to mention that the libraries shown in Table 2 were chosen for their simplicity and availability. In addition, the method to obtain the shared objects was selected because, in the context of our project, it resulted to be the simplest and fastest to implement. However, as discussed in Section 5.1, it does not yield the most accurate results.

Libraries	Versions
glibc	2.19, 2.20, 2.21, 2.22, 2.23 and 2.24
ncurses	5.7, 5.8, 5.9, 6.0
jpeg	9b
libmpeg	2-0.5.1
libpng	1.6.28
libzip	1.1.3

Table 2: Libraries that where downloaded and manually built

4.3 Performance Experiments

In order to evaluate our proposed implementation we have tested the introspection program by running it to identify two libraries, *libc-2.23* and *libncurses-5.9*, both dynamically linked to the *Htop* program running on the guest VM. These experiments were executed under different load configurations to analyze both the effectiveness and efficiency of the program. In addition we are interested in measuring the performance impact that the program has on the guest VM and the hypervisor.

We defined three load states: Low, Mid, and High. These represent either the guest VM's CPUs or the hypervisor's CPUs stressed to 0%, 50% and 100% respectively. Furthemore, a load configuration represents a combination of the VM load state and the hypervisor load state. This leads to nine possible load configurations. Examples of this load configurations are, VM_Low-Hypervisor_Low, VM_Low-Hypervisor_Mid and VM_High-Hypervisor_Low. The tool used to cause the different load levels was stress-ng[9].

First of all, we measured the time that the library extractor pauses the guest VM and the time that the library identifier takes to finish with the identification. On one hand, the Pause Time indicates the impact that the program has over the introspected VM. On the other hand, it is used to calculate the total execution time of the library identification program by adding it to the Identification Time.

Secondly, we collected the memory and CPU usage that the whole program exhibits during its execution in order to assess the performance impact imposed on the hypervisor under different load configurations.

Finally, the highest match score obtained in each experiment was registered. This indicator is used to determine the effectiveness of the library identification program under the aforementioned load levels. As the program relies on the content of the library memory pages on the guest VM, we expect that the memory utilization levels of the guest VM will have a direct impact on the match scores calculated. This is due to the fact that when the memory of a system is stressed, pages are swapped out of the RAM. This behavior causes that the amount of strings obtained from the extracted memory pages is a subset of the total. As a consequence, the fingerprint generated from the memory dump differs more from the reference fingerprint so the match score decreases. To measure this effect we defined two extra load states in which we stressed the memory of the hypervisor and the guest VM to 100% .

[9]https://github.com/ColinIanKing/stress-ng

We have run each test one hundred times and then calculated the mean and the sample standard deviation for each value.

4.4 Version Strings Relevance Experiments

To evaluate the relevance of the strings containing version information of a library in the identification process, we have performed two experiments. The idea of these experiments is to quantify the influence that these strings have in the obtained match scores. An example of such strings can be observed in Figure 6.

The first experiment consisted in tampering with the fingerprint obtained from the library libc-2.23. We modified it to include the strings containing version information of a newer library (libc-2.24). The idea behind this experiment was to emulate what an attacker could do to hide a vulnerable library version.

For the second experiment we removed from the sample fingerprint and the reference fingerprints all the strings containing version information. Then, we executed the implemented program under this modified scenario and compared the obtained results with a "normal" execution of the program in which the strings were part of the fingerprint. The objective was to test if the identification method is independent from these strings.

```
GLIBC_2.22
GLIBC_2.23
GLIBC_2.24
glibc 2.24
NPTL 2.24
GNU C Library (Ubuntu GLIBC 2.24-0ubuntu5)
    stable release version 2.24, by Roland
    McGrath et al.
```

Figure 6: Example of strings containing version information of libc-2.24

5 Results

In this section we start by presenting the outcome of our library identification program and then, we show and analyze the results of the conducted experiments.

It is important to note that all the the experiments have been performed for the *libc-2.23* and *libncurses-5.9* libraries. These are dynamically linked to the process *Htop*[10] that was running on the guest VM during the execution of the experiments.

To avoid redundancy, only the results for *libc-2.23* are shown as part of the Results section. The Appendix contains detailed tables with the outcome of the experiments for both libraries.

As for the experiments results, we first present the pause and identification times. Secondly, the memory and CPU usage are shown and explained. Thirdly, we present the results regarding the effectiveness of the program under different load configurations. Finally, the result of the experiments performed to evaluate the relevance of the version strings at the moment of identification are shown and analyzed.

5.1 Program Output Results

The library identification program returns the top reference fingerprints along with their corresponding match scores. This match score represents the overlap ratio between the sample fingerprint and a given reference fingerprint. As it can be seen from the example shown in Table 3, the program returned the top 8 reference fingerprints. The result with the highest match score returned by the program is the one corresponding to the sample version number. This means that the identification program succeeds in identifying the correct library version when the database contains the exact reference for that library. Furthermore, we can observe that for related libraries the match score slightly differs (e.g. libc-2.23.so vs. libc-2.22.so) whereas it significantly drops when comparing the sample library against an unrelated library (e.g. libc-2.23.so vs. libjpeg.so.9.2.0).

Match Score	**Fingerprint in the DB**
20.59%	libc-2.23.so.strings
19.73%	libc-2.22.so.strings
19.71%	libc-2.24.so.strings
19.34%	libc-2.21.so.strings
18.78%	libc-2.20.so.strings
18.25%	libc-2.19.so.strings
3.56%	libjpeg.so.9.2.0.strings
2.91%	libncurses.so.5.9.strings

Table 3: Output for libc-2.23

In addition, it can be observed that our program returns a result which is not greater than 21%. However, the expected value under low memory usage conditions should be close to

[10]https://github.com/hishamhm/htop

100%. This is because the amount of memory pages swapped out should be minimal, meaning that, the library extracted should be almost exactly like the corresponding shared object file. The reason for the difference between the expected match scores and our results, has to do with the way that the reference database was built. When we built the libraries, we did not configure and install them in the same way a Linux package manager does it. The way that we have built the libraries introduces external strings, such as the path where the libraries were compiled and as a result the match scores are drastically affected.

To verify that the strings added in the reference fingerprints affect the match scores, we compared the fingerprint obtained from the shared object file with the fingerprint extracted from the same shared object loaded in the VM's memory. Then, we calculated the similarity coefficient between these two, and obtained a match score of 97.06%. The match score only reaches 100% when the complete binary representation of the library is in memory, in other words, no memory page corresponding to the library was swapped out.

Another result that needs to be discussed, is related to the outcome of our library identification program, when the sample library is not included in the reference database. In this context, the program could exhibit two different results.

For the first one, the result obtained from the program consists of all match scores below 9%. We consider such a result a mismatch. This means that the selected library could not be identified. This result indicates that the program behaves as expected, as it is not possible to identify the exact version of a library if there is no reference for it in the database.

On the other hand, when the result obtained contains values above 9% but from a different library than the one being identified, we consider the result to be a false positive. It is important to mention that the "9%" threshold was obtained from the experimental results on the paper [13].

5.2 Paused Time and Identification Time

Figures 7 and 8 show the time that the VM was paused by the library extractor module and the time that the library identifier module required to identify the *libc-2.23* library. These two figures depict these metrics under different load configurations.

As can be observed, the Paused Time and Identification Time are significantly affected by the load of the hyvervisor's CPU. This was an expected outcome as the library identification program was running on this privileged system.

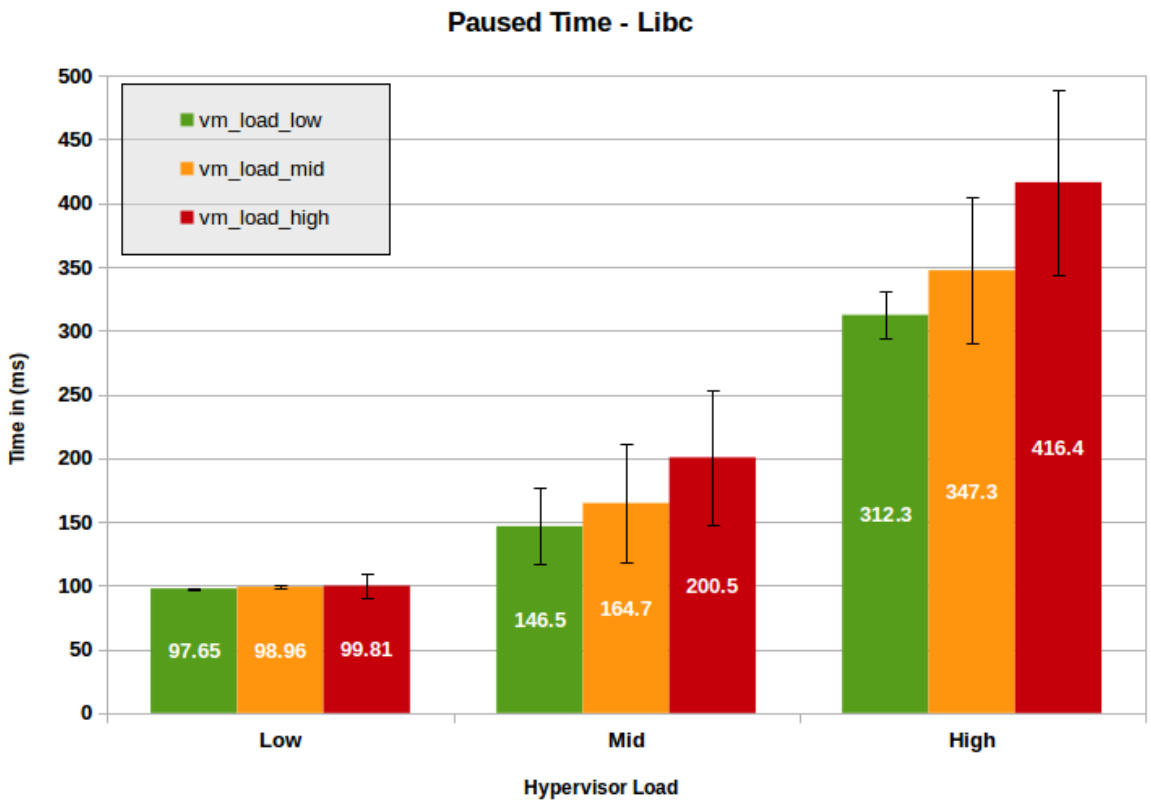

Figure 7: Paused Time under the different load configurations

The fact that the VM load starts to affect the Paused and Identification Times whenever the hypervisor's vCPUs are under stress has to do with how the server's CPU is being used. As explained in Section 4.1, our experimental environment consisted of an eight vCPUs hypervisor where two of those were assigned to the VM. Whenever this VM is stressed under a low hypervisor load, only the two vCPUs allocated to the VM are affected. This means that one of the six unused vCPUs can be used to execute the library identification program. This results in a minimum loss of performance.

When the hypervisor CPU load starts increasing, every vCPU is evenly stressed. This means that in the situations of a Mid and High loaded CPU, all eight vCPUs are stressed to 50% and 100% respectively. When the VM is also stressed, the overall load of the hypervisor increases as well. As a result the Paused and Identification times are affected. In the worst case scenario the Paused Time and Identification Time are approximately 400 ms and 1000 ms

respectively.

It is important to recall from the explanation given in Section 4.2.1, that the library identification module starts running after the execution of the guest VM is resumed. In other words, the identification is done offline. This design decision was made to reduce the performance impact on the running VMs so that they are only affected by the Paused Time.

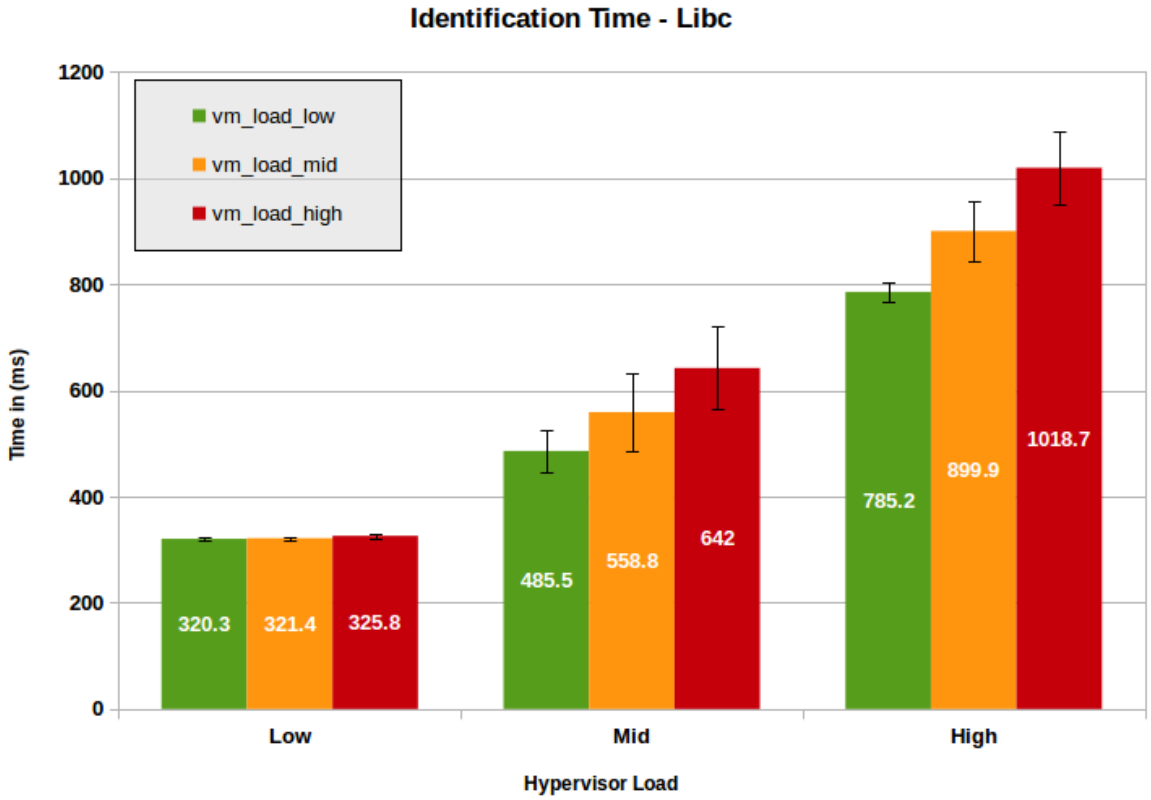

Figure 8: Identification Time under the different load configurations

5.3 Memory and CPU Usage

Figures 9 and 10 depict the memory and CPU usage of the complete library identification program under the different load configurations. As the idea of the program presented in this report was to provide a proof of concept, during the implementation of it we did not focus on optimizing the performance. However, these metrics were included for completeness as they provide an indication of the resource utilization of the program.

From Figure 9 we can observe that our program is not greatly affected by the hypervisor CPU load, the VM CPU load, or the combination of both. We believe that the variations in the memory usage are caused by the differences in the amount of strings that were extracted by the library extractor. In Section 5.4 we analyze in more detail the possible reasons for this behaviour.

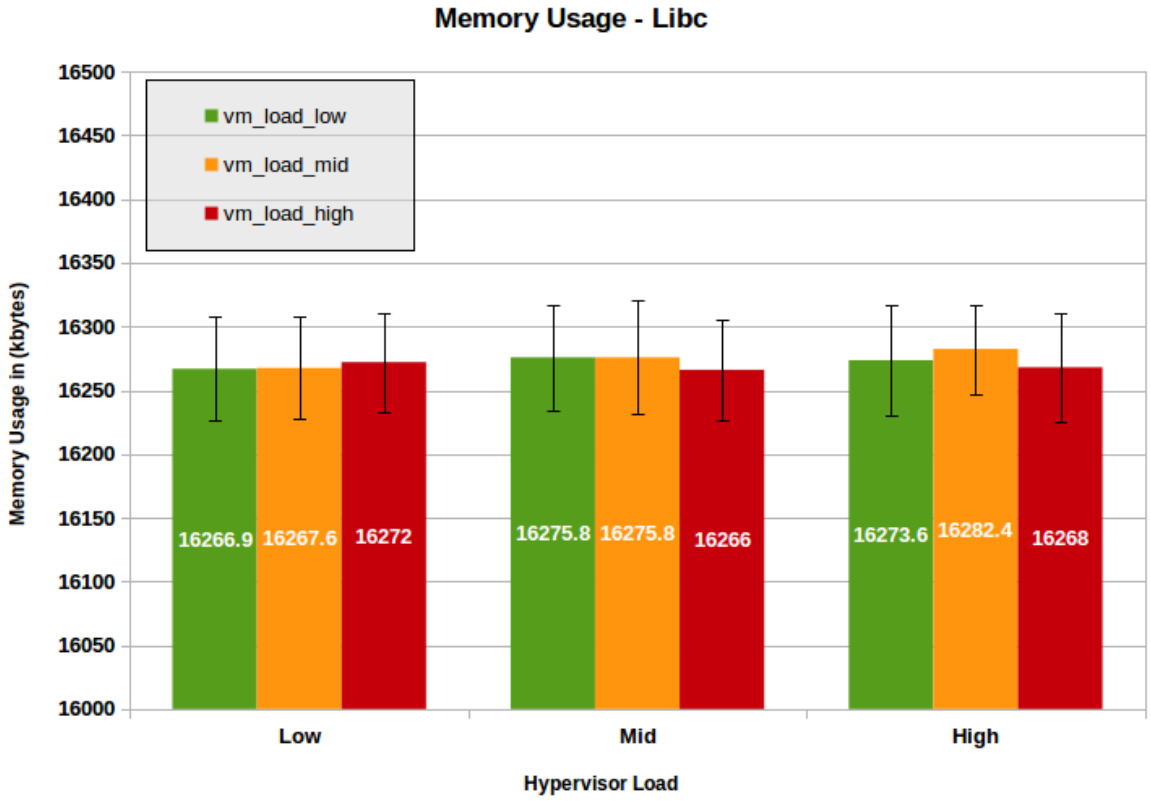

Figure 9: Memory Usage under the different load configurations

Figure 10 shows how the CPU time given to our program decreases whenever the CPU load of the hypervisor increases. The reason for this, has to do with the way that the OS scheduler works and the way that the CPU time is calculated.

The CPU time, or CPU percentage is calculated by adding the user and system times and then dividing it by the total running time. When the program is waiting in the process queue only the total running time is increased but the user and system times are not affected, which causes the CPU usage to drop.

When the CPU load is low the OS can assign more often our program to the CPU which reduces the waiting time, and makes the CPU usage increase. On the other hand, when the load is mid or high the CPU is shared among more processes and the OS scheduler has more processes to choose from. This impacts the total execution time of our program because it needs to wait in the process queue to resume its execution, meaning that the program accumulates more waiting time thus decreasing its CPU usage.

The standard deviation values shown in Figure 10, give an indication that there are fluctuations among different runs of the program. This can be caused by several factors, such as processes priorities and the stress tool used, which has a non uniform way of generating system load.

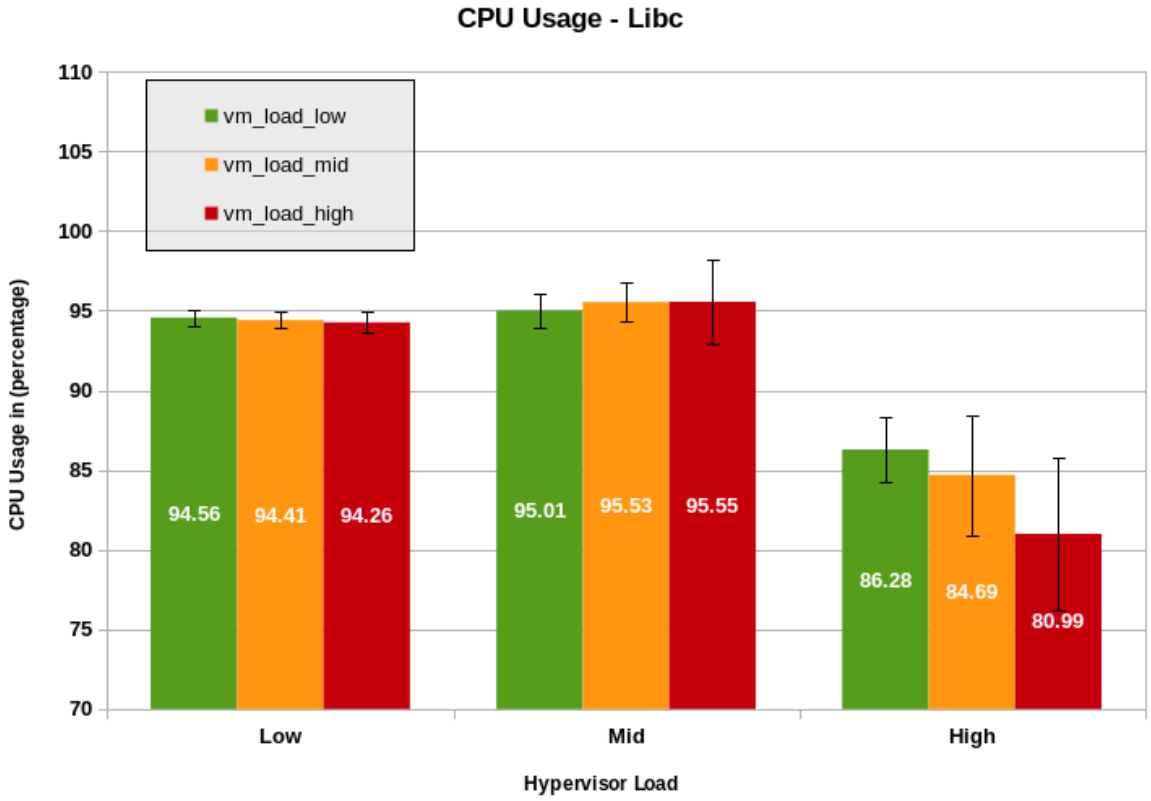

Figure 10: CPU Usage under the different load configurations

5.4 Match Score

The following charts show how the match score calculated by the library identification program is affected by the different load configurations.

If we analyze the data provided by Figure 11 we can observe that the match score is not significantly dependent on the CPU load. This was an expected result as there is no correlation between the way that the match score is calculated and program CPU usage.

However, as shown in Figure 12 the match score is affected by the memory load of the hypervisor and the VM. This is because memory pages corresponding to the selected library are getting swapped out, causing the fingerprint generated from the extracted memory to differ more from the reference fingerprint. The reason for the swapping is that the tool used to stress the systems, as explained in Section 4.3, causes a race condition for memory pages between the `Htop` process (that makes use of the library that is being identified) and the `stress-ng` threads. So, even though the library is being used, not all the memory pages are residing in RAM at the time the memory is extracted.

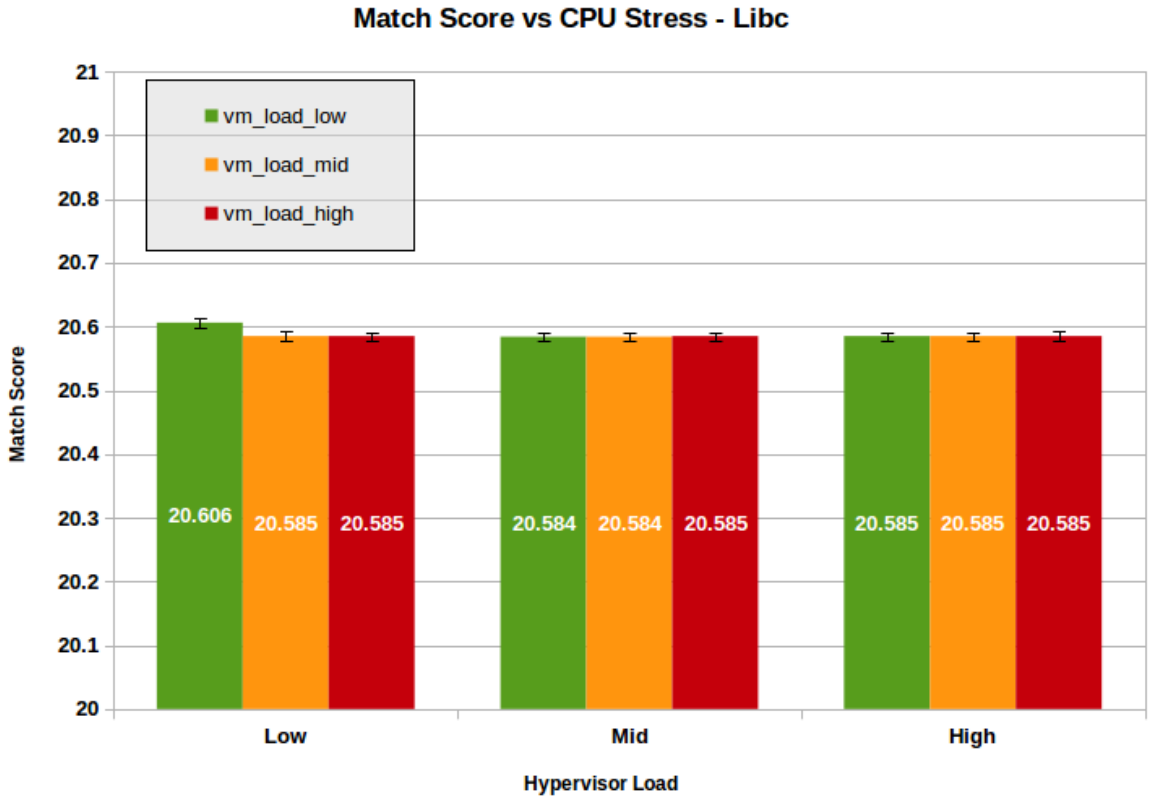

Figure 11: Match Score obtained under the different CPU load configurations

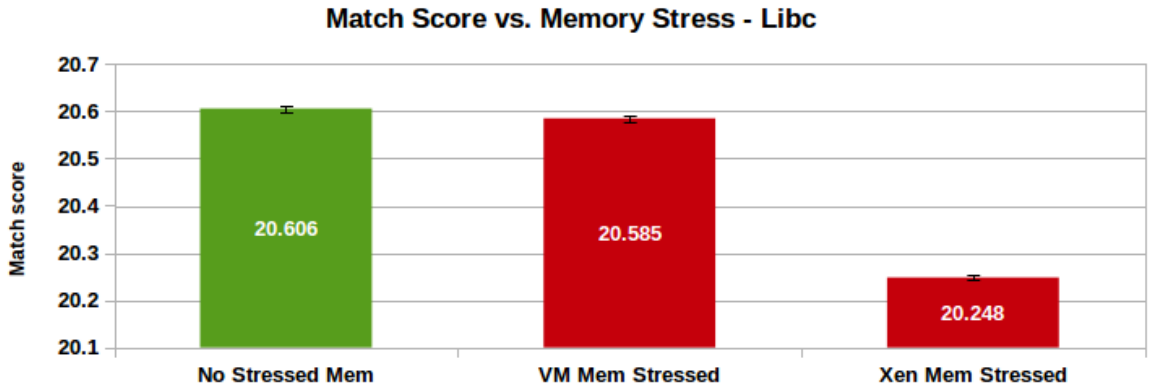

Figure 12: Match Score obtained under the different memory load configurations

5.5 Version String Relevance Results

Table 4 shows the results of manually adjusting the fingerprint obtained from the libc-2.23 running on the VM. If we recall from the experiment explanation, this fingerprint was modified to include the strings containing version information of a newer library (libc-2.24). The table

contains the match scores obtained when the original sample fingerprint was compared against the libc-2.23 and libc-2.24 reference fingerprints. The same values can be found but for the tampered fingerprint. From this result, we can observe that the impact of manually changing the sample fingerprint by altering the version numbers, only affected the match score by 0.01%. This value gives us an indication that the relevance of strings containing version information is not significant.

Sample Fingerprint	**Libc-2.23 Ref. Fingerprint**	**Libc-2.24 Ref. Fingerprint**
libc-2.23 original	20.60%	19.82%
libc-2.23 tampered	20.59%	19.83%

Table 4: Manually Tampered Scenario

The following tables show the result of removing all the strings that contain version information from the sample fingerprint and the listed reference fingerprints. Table 5 shows the results of a normal execution while Table 6 shows the results of the scenario without version strings. The differences between these tables show that the match score was affected by only a 0.03% when the strings containing library version information were removed. This value supports the idea that the chosen library identification method is independent of the aforementioned strings.

Match	**Fingerprint in the DB**
20.59%	libc-2.23.so.strings
19.73%	libc-2.22.so.strings
19.71%	libc-2.24.so.strings
19.34%	libc-2.21.so.strings
18.78%	libc-2.20.so.strings
18.25%	libc-2.19.so.strings

Table 5: Normal Scenario

Match	**Fingerprint in the DB**
20.54%	libc-2.23.so.stripped
19.70%	libc-2.22.so.stripped
19.68%	libc-2.24.so.stripped
19.31%	libc-2.21.so.stripped
18.74%	libc-2.20.so.stripped
18.22%	libc-2.19.so.stripped

Table 6: Stripped Scenario

6 Discussion

In our project we have shown that by combining VMI techniques and the printable strings of libraries it is possible to identify a library that is being used on a selected VM. Although we were successful in implementing the library identification program, the experiments done to test this proof of concept were executed in an ideal environment. As a consequence the following limitations should be addressed before experimenting with our program in a production environment.

First of all, our solution is based on the Ubuntu Linux distribution and it is not directly portable to a different OS. However, the mechanisms and methodology used to build our program are OS independent. This means that it should be possible to change the configurations to support different OS's including other Linux distributions and Windows.

Secondly, the experiments done during this project were executed on a paravirtualized VM. Nonetheless, there is no indication that our program needs adjustment to support Full virtualization as LibVMI does not impose restrictions on the type of virtualization when it comes to access VMs' memory.

Furthermore, the library identification program we developed focuses on identifying dynamically linked libraries and was tested only with *libc-2.23* and *libncurses-5.9*. However, this is not a strong limitation because previous research has shown [13] that printable strings identification works with a rich variety of libraries as well as with statically linked ones.

Additionally, as a result of analyzing the implementation of the library identifier module, it can be inferred that the size of the database has a high impact on the identification time. Solutions like program parallelization and optimized database indexing can be used to overcome this limitation.

One of the most important limitations is imposed by LibVMI, which requires details of the guest kernel to bridge the semantic gap, and be able to understand the memory layout of the guest VM. In our solution, this was partially mitigated by executing a custom kernel module in the guest VM, that extracted the required information such as memory offsets. However, as explained in Section 4.2.1, one could run the kernel module in a VM that has the same kernel version and compilation flags as the VM that is intended to introspected.

Moreover, page swapping of the library contents may result in lower matching scores. This has to do with the fact that when memory pages of the library are swapped to disk, our tool can not extract the complete library from memory. This results in a less accurate fingerprint being used to compare against the reference fingerprints in the database.

Finally, it is important to note that whenever a library that is not included in the reference database goes through the identification process, a false-positive might occur. This happens with our data set when the output of the library identification program shows match scores higher than 9%.

7 Conclusion

Our project researched the use of the VMI techniques provided by LibVMI to create a reliable library identification program. We have shown that the library LibVMI can be used to effectively and efficiently extract libraries from the VM's memory. To identify the library version from this extracted memory, we used a method based on the printable strings present in the library's executable. This method shows to be an accurate way to identifying libraries, even when the version strings are removed from the executable or tampered with.

To evaluate our program we performed different experiments that measure the program's effectiveness and efficiency under different system loads. The results of these experiments show that the time that the VM is paused and the library identification time is primarily dependent on the CPU load of the hypervisor. Despite the fact that the program is affected by the hypervisor's load, we show that our implementation is able to perform the task at hand in approximately a second in the worst case scenario. By optimizing the program, better performance can be achieved.

When analyzing the memory stress results, we can see that memory page swapping increases under higher loads. In this context, there is a chance that library memory pages get swapped out whenever the memory of the VM or hypervisor is highly loaded. The result of this situation is that it is not possible to extract the complete library executable from memory, which ends up affecting the amount of printable strings that can be extracted thus decreasing the match score.

8 Future Work

In this project, we show that it is possible to implement a method for reliable library identification based on LibVMI. However, there are several aspects that can be researched to enhance the solution.

As discussed, the way in which the database is built highly influences the program's match scores. It would be beneficial to work on a method for building the database, so the reference libraries are compiled in a similar way as the running ones. As a result the fingerprints extracted from memory would present a higher match score when compared against the reference ones.

We have implemented a library identification program based on printable strings. Although this method succeeded in identifying different library versions, it would be interesting to implement a second identification method to support decision making. This one could be based for example on behavioral analysis.

Finally, an interesting point would be to integrate our solution with a database containing known vulnerable library versions so as to support automated vulnerability scanning.

9 Acknowledgments

We would like to thank our supervisors Ralph Koning and Ben de Graaff for their constant feedback, guidance and input throughout the research project. In addition, we would like to thank Thomas Rinsma who provided us with a valuable draft of his research done with Riscure.

References

[1] glibc getaddrinfo() stack-based buffer overflow. *URL: https://sourceware.org/ml/libc-alpha/2016-02/msg00416.html*.

[2] Openssl dos vulnerability, new bagel variants. *URL: https://isc.sans.edu/forums/diary/Updated +1345+318+GMT+OpenSSL+DoS+Vulnerability +New+Bagel+Variants/140/.*

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[4] Silvio Cesare and Yang Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 181–189. IEEE, 2011.

[5] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

[6] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[7] Siddharth Gujrathi. Heartbleed bug: An openssl heartbeat vulnerability. *International Journal of Computer Science and Engine ter Science and Engineering*, 2(5):61–64, 2014.

[8] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM, 2011.

[9] Anil Kumar Konasale Krishna and Robert Ricci. Vmicvs: Cloud vulnerability scanner. *https://pdfs.semanticscholar.org/1415/02fa0f8f01738481730040 4d2c11a23aca9c.pdf*, 2016.

[10] Robert Love. *Linux kernel development second edition*. Sams Publishing, 2005.

[11] Bryan D Payne. Simplifying virtual machine introspection using libvmi. *Sandia report*, pages 43–44, 2012.

[12] Bryan D Payne, DP de A Martim, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397. IEEE, 2007.

[13] Thomas Rinsma. Automatic library version identification, and exploration of techniques. Riscure and Faculty of Science, Radboud University, Under submission, 2017.

[14] Christian A Schneider. *Full Virtual Machine State Reconstruction for Security Applications*. PhD thesis, München, Technische Universität München, Diss., 2013, 2013.

[15] Ronghua Tian, Lynn Batten, Rafiqul Islam, and Steve Versteeg. An automated classification system based on the strings of trojan and virus families. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 23–30. IEEE, 2009.

[16] Peter Van der Linden. *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.

[17] Mike Van Emmerik. Signatures for library functions in executable files. *Citeseer*, 1993.

[18] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. Libvmi: a library for bridging the semantic gap between guest os and vmm. In *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*, pages 549–556. IEEE, 2012.

10 Appendices

A Library extractor pseudo code

```
libVMI_PauseVM (VM_Name)

addressOfProcTaskStruct = libVMI_TranslateKernelSymbol("init_task")

//Walk through the task_struct linked list
while (TaskStructPID != Selected_PID ) {

    addressOf_ProcTaskStruct = libVMI_ReadVMAddress(
        addressOfProcTaskStruct + Next_ProcTaskStruct_offset)
}

//Get memory info of the selected process
adressOf_mmStruct = libVMI_ReadVMAddress(addressOfProcTaskStruct +
    mmStruct_offset)

//Walk through the VM_area_struct linked list
adressOf_VMAreaStruct = libVMI_ReadVMAddress(adressOf_mmStruct +
    VMAreaStruct_offset)

while [ ( VMArea_flags != "executable" ) AND ( FileNameMapped !=
    Selected_LibraryName) ] {

    adressOf_VMAreaStruct = libVMI_ReadVMAddress(adressOf_VMAreaStruct
        + Next_VMAreaStruct_offset)

}

//Get memory pages covered by the VM_area
adressOf_firstMemoryPage = libVMI_ReadVMAddress(adressOf_VMAreaStruct
    + firstMemoryPage_offset)

adressOf_lastMemoryPage = libVMI_ReadVMAddress(adressOf_VMAreaStruct +
     lastMemoryPage_offset)

adressOf_currentMemoryPage = adressOf_firstMemoryPage

while ( adressOf_currentMemoryPage < adressOf_lastMemoryPage ) {

    rawData = libVMI_ReadVMAddress( adressOf_currentMemoryPage )
    write_to_file (rawData)
    adressOf_currentMemoryPage += MemoryPage_Size

}

libVMI_ResumeVM (VM_Name)

exit
```

Figure 13: Pseudo code of the library extractor (simplified to improve readability)

B Program Output libncurses

Match S.	**Fingerprint in the DB**
15.50%	libncurses.so.5.9.strings
15.47%	libncurses.so.5.8.strings
15.20%	libncurses.so.5.7.strings
14.00%	libncurses.so.6.0.strings
4.89%	libjpeg.so.9.2.0.strings
4.65%	libmenu.so.6.0.strings
4.48%	libresolv-2.23.so.strings
4.41%	libresolv-2.24.so.strings

Table 7: Program output for libncurses-5.9

C Detailed Results

C.1 Pause Time Tables

Xen Load \ VM Load	Low	Mid	High
Low	97.65ms ±0.67	98.96ms ±1.46	99.81ms ±9.32
Mid	146.50ms ±30.02	164.70ms ±46.28	200.50ms ±52.51
High	312.30ms ±18.95	347.30ms ±57.16	416.40ms ±72.25

Table 8: Paused Time - libc

Xen Load \ VM Load	Low	Mid	High
Low	80.12ms ±0.57	81.80ms ±3.86	83.90ms ±4.90
Mid	138.50ms ±26.83	159.10ms ±44.40	185.40ms ±61.51
High	304.00ms ±14.49	341.00ms ±45.24	416.10ms ±72.47

Table 9: Paused Time - libncurses

C.2 Identification Time Tables

Xen Load \ VM Load	Low	Mid	High
Low	320.30ms ±1.71	321.40ms ±3.48	325.80ms ±4.96
Mid	485.50ms ±41.00	558.80ms ±73.68	642.00ms ±78.49
High	785.20ms ±19.20	899.90ms ±57.19	1018.70ms ±70.13

Table 10: Identification Time - libc

Xen Load \ VM Load	Low	Mid	High
Low	260.30ms ±1.71	259.40ms ±3.97	259.20ms ±6.61
Mid	393.10ms ±42.27	454.80ms ±62.83	517.10ms ±78.79
High	643.30ms ±19.01	748.0ms ±57.80	840.40ms ±71.88

Table 11: Identification Time - libncurses

C.3 Memory Usage

Xen Load \ VM Load	Low	Mid	High
Low	16266.900 kbytes ±40.565	16267.600 kbytes ±39.904	16272.000 kbytes ±38.908
Mid	16275.800 kbytes ±41.677	16275.800 kbytes ±44.465	16266.500 kbytes ±39.195
High	16273.600 kbytes ±42.746	16282.400 kbytes ±35.004	16268.200 kbytes ±42.430

Table 12: Memory Usage - libc

Xen Load \ VM Load	Low	Mid	High
Low	14943.700 kbytes ±37.373	14947.100 kbytes ±39.632	14943.700 kbytes ±39.066
Mid	14942.400 kbytes ±37.277	14949.100 kbytes ±34.8112	14942.100 kbytes ±37.808
High	14946.000 kbytes ±35.982	14937.900 kbytes ±41.129	14941.000 kbytes ±40.178

Table 13: Memory Usage - libncurses

C.4 CPU Usage

Xen Load \ VM Load	Low	Mid	High
Low	94.56% ±0.49	94.41% ±0.53	94.26% ±0.66
Mid	95.01% ±1.08	95.53% ±1.23	95.55% ±2.63
High	86.28% ±2.05	84.69% ±3.76	80.99% ±4.79

Table 14: CPU Usage - libc

Xen Load \ VM Load	Low	Mid	High
Low	94.49% ±0.50	94.29% ±0.74	93.90% ±0.71
Mid	94.43% ±1.08	94.88% ±1.44	95.16% ±2.69
High	84.09% ±1.81	82.08% ±3.90	77.84% ±5.49

Table 15: CPU Usage - libncurses

C.5 Match Score

Xen Load \ VM Load	Low	Mid	High
Low	20.606% ±0.007	20.585% ±0.007	20.585% ±0.006
Mid	20.584% ±0.006	20.584% ±0.007	20.585% ±0.006
High	20.585% ±0.006	20.585% ±0.006	20.585% ±0.007

Table 16: Effectiveness - libc

Xen Load \ VM Load	Low	Mid	High
Low	15.500% ±0.000	15.500% ±0.000	15.500% ±0.000
Mid	15.500% ±0.000	15.500% ±0.000	15.500% ±0.000
High	15.500% ±0.000	15.500% ±0.000	15.500% ±0.000

Table 17: Effectiveness - libncurses