DLP-Visor: A Hypervisor-based Data Leakage Prevention System

Guy Amit[2], Amir Yeshooroon[2], Michael Kiperberg[1] and Nezer J. Zaidenberg[2]
[1]*Software Engineering Department, Shamoon College of Engineering Beer-Sheva, Israel*
[2]*School of Computer Science, The College of Management, Academic Studies, Israel*

Keywords: Virtual Machine Monitors, Hypervisors, Trusted Computing Base, Data Leakage Prevention.

Abstract: Data theft by insiders is considered by many organisations to be one of the most serious threats. Data leakage prevention (DLP) systems attempt to prevent intentional or accidental disclosure of sensitive information by monitoring the content or the context in which the information is transferred, for example, in a file system, an email server, instant messengers. We present a context-sensitive DLP system, called DLP-Visor, which is implemented as a thin hypervisor capable of intercepting system calls in Windows operating systems equipped with Kernel Patch Protection. By intercepting system calls that govern the file system, inter-process communications, networking, system register and system clipboard, DLP-Visor guarantees that sensitive information can never leave a predefined set of directories. The performance overhead of DLP-Visor (7.2%) allows its deployment in real-world applications.

1 INTRODUCTION

One of the main goals of information security is to prevent unauthorised entities from accessing sensitive information. Many systems protect the sensitive information from external attacks by using encryption (Khati et al., 2017; Guo et al., 2016) or by implementing an access control (Li et al., 2019; Tourani et al., 2017). In this paper, we focus on insider attacks, in which an organisation employee steals sensitive information that they can legitimately access. An insider attack can also manifest in the form of a malicious program executed with the privileges of an employee who has access to sensitive information. In order to combat these attacks, researchers have proposed a new type of information security — Data Leakage Prevention (DLP) systems.

DLP systems can be divided into two types (Alneyadi et al., 2016): content-based and context-based. Both types prevent certain operations that can leak sensitive information. Context-based DLP systems determine the validity of an operation based on its metadata while content-based DLP systems consider the data itself. For example, a context-based DLP system can prevent sending of an email if the destination email is not in the organisation's domain. A content-based DLP system can prevent sending of an email if it contains the name of a classified project. Content-based DLP systems tend to have a greater performance impact due to the analysis they perform on the content, which is usually larger than the context. Content-based DLP systems use heuristics to determine whether the content matches a predefined rule. For example, Kantor (Kantor et al., 2012) proposed to compare hashes of individual paragraphs or sentences. Clearly, minor modifications of sentences will produce different hashes.

On the other hand, context-based DLP systems are less flexible. A context-based DLP system can be configured to either allow sending emails to certain domains or block them completely. The decision cannot be made based on the content of the email or an attachment.

We propose a context-based DLP system, implemented as a thin hypervisor (Shinagawa et al., 2009; Seshadri et al., 2007), capable of executing a single operating system. We call our system "DLP-Visor". Thin hypervisors have two benefits compared with full hypervisors: their code base is small enough to allow formal veirification and their performance is better due to the exclusion of time-consuming components such as memory management, hardware allocation and scheduling.

In contrast to system call interception mechanisms that are implemented in user-mode (Wüchner and Pretschner, 2012), DLP-Visor reliably intercepts op-

Amit, G., Yeshooroon, A., Kiperberg, M. and Zaidenberg, N.
DLP-Visor: A Hypervisor-based Data Leakage Prevention System.
DOI: 10.5220/0010221104160423
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 416-423
ISBN: 978-989-758-491-6

erating system events by intercepting kernel-mode functions. Moreover, DLP-Visor, having higher privileges than the operating system, can operate even in the presence of a patch detection mechanism like Microsoft Kernel Patch Protection (Field, 2006).

DLP-Visor allows the system administrator to select a set *S* of directories that contain *sensitive* files. A process that opens a sensitive file becomes *critical* and a file written by a critical process becomes sensitive. The system clipboard becomes sensitive when it receives data from a critical process. When a process receives data from a sensitive clipboard, it becomes critical. In other words, files and the system clipboard act as criticality mediators. In order to prevent information leakage through other channels provided by the operating system, DLP-Visor does not allow critical processes to write to the registry, use network sockets or write to memory of another process. DLP-Visor uses a system call interception mechanism to keep track of the critical processes and restrict their operations.

We make the following assumptions about the abilities of an attacker:

- The attacker does not have kernel-mode privileges.
- The attacker has full (administrator) user-mode rights.

Under these assumptions, we claim that DLP-Visor can provide the following guarantee: *information that resides in sensitive directories cannot be copied outside of them*. In particular, sensitive information cannot be copied to external media, sent in the body of an email or as an attachment, copied to a network folder or uploaded to a website.

The main contributions of the paper are:

- We describe the design of DLP-Visor, a hypervisor-based data leakage prevention system.
- We compare the performance of two system call interception mechanisms.
- We evaluate the performance and security of DLP-Visor.

Our results show that performance degradation due to DLP-Visor is insignificant (7.2 %), allowing its deployment even in high-performance workstations.

2 BACKGROUND

Virtualization extensions to CPUs were introduced in 2005-2006 by Intel and AMD. These extensions allow the CPU to execute multiple operating systems simultaneously in isolated environments called "Virtual Machines" (VMs). The software component that configures and monitors the execution of VMs is called a "Virtual Machine Monitor" (VMM) or a "hypervisor". In particular, the hypervisor can configure the interception of various events that occur in the VMs, e.g. execution of privileged instructions, interrupts, memory accesses, etc. When an event configured for interception occurs, the CPU transfers control from the VM to the hypervisor.

The hypervisor inspects the information that describes the occurred event and reacts accordingly. After completion of the event handling, the hypervisor transfers control back to the VM.

In 2007-2008, AMD and Intel introduced a Secondary-Level Address Translation (SLAT) mechanism to their virtualization extensions, allowing the hypervisor to handle memory virtualization more efficiently. With SLAT, the hypervisor can define a page table for each VM, which defines translation of the VMs' physical addresses to real physical addresses. Each entry of SLAT defines not only the mapping between addresses but also the access rights, thus effectively allowing the hypervisor to intercept accesses to physical pages.

A hypervisor can intercept a wide range of events. We will discuss only those that are relevant to this paper. The first type of event is access to MSRs, special registers used to report the features of a CPU and configure its state. They participate in the configuration of

- 64-bit environments via the `EFER` MSR;
- system call mechanism via the `STAR` and `SYSENTER` families of MSRs; and
- physical memory caching policies via the `MTRR` family of MSRs, etc.

Two special instructions allow the software to write to and read from MSRs, which are identified by a number. The hypervisor can intercept read and write MSR accesses separately for each MSR. Upon transition to the hypervisor, the number of the MSR and its new value, in the case of a write operation, are reported to the hypervisor.

Exceptions delivery is another type of event. By configuring the exception bitmap field, the hypervisor can intercept an exception before it is delivered to the operating system. Upon transition to the hypervisor, the exception number and error code are reported to the hypervisor.

The last type of event that has relevance to this paper is so-called "EPT-violations"; i.e. access to the VMs' physical addresses that cannot be translated to real physical addresses. In essence, EPT-violations are page-faults in the secondary-level address trans-

lation. EPT-violations can occur either due to absent entries in the secondary-level page table or due to inappropriate rights, e.g. write access to a read-only page. Upon transition to the hypervisor, the virtual and physical addresses are reported to the hypervisor. In addition, the hypervisor receives a value that resembles an error-code. This value can be used to determine the reason for the EPT-violation.

3 SYSTEM CALL INTERCEPTION

System call interception enables a software module to monitor the behaviour of an entire operating system or a specific application. This monitoring ability is useful in the implementation of protective, i.e. antivirus, and offensive, i.e. malware, software. System call interception approaches can be divided into two types: user-mode interception and kernel-mode interception. In user-mode interception, the code of the wrapper functions is replaced with a call to a monitor that eventually performs the actual system call. The monitor can inspect: the argument of the system call before its execution and the return values of the system call after its completion. The monitor can also respond to an attempt of system call execution by notifying the user, preventing the system call completely or modifying its arguments.

We note that user-mode interception is vulnerable to user-mode attacks and, therefore, is ineffective in our attack model. For example, a memory corruption attack can overwrite the monitoring system call wrapper with the code of the original system call wrapper, thus disabling the monitoring functionality. Ultimately, an attacker can execute the `sysenter` or the `syscall` instruction directly, without even calling the monitoring system call wrapper.

Kernel-mode system call interception can defeat the user-mode attacks by installing the interception mechanism in the kernel itself. Prior to introduction of Microsoft's Kernel Patch Protection (KPP) (Field, 2006), the interception was mainly performed by changing SSDT entries to point to monitoring functions, which then call the original functions. Because SSDT patching was frequently used for malicious purposes, Microsoft decided to prohibit its modification. Microsoft's KPP periodically computes a checksum of SSDT and produces a BSOD when a modification is detected. KPP can also detect modification of the model specific registers involved in system call handling.

Fortunately, in some cases, a hypervisor can hide modification of sensitive registers and data structures, thus allowing it to intercept system calls in kernel-mode even when KPP is active. In this section, we explain how DLP-Visor can intercept not only entries but also exits from system calls.

3.1 Interception of System Call Entries

In order to reduce the interception overhead of irrelevant events, we propose to use a targeted interception mechanism, which allows the hypervisor to intercept only those system calls that need to be monitored. Our approach is inspired by the *Stealth Breakpoints* introduced in (Deng et al., 2013; Lengyel et al., 2014). According to this approach, the hypervisor replaces the first instruction of the function it wishes to intercept with the `INT3` instruction. The hypervisor stores the original instruction in its internal data structures. Finally, the hypervisor configures interception of the breakpoint exception. When a breakpoint exception occurs the hypervisor emulates the first instruction, advances the instruction pointer to the next instruction, and continues the VM's execution. This method allows the hypervisor to intercept only the system calls that require monitoring.

The hypervisor can use officially distributed symbol files (Microsoft, 2020b) to find the addresses of the functions that need to be intercepted. Due to address-space layout randomisation (Cook, 2013), the actual addresses of the functions change on every boot, but relative offsets of the functions remain constant. The idea is to statically calculate the offsets of the function that require interception from the system call handler and use them later for calculation of actual addresses. When the operating system writes an address to the `IA32_SYSENTER_EIP` or the `IA32_LSTAR` MSRs, the hypervisor adds to this address the offsets of the functions than require interception and obtains the actual address in which a breakpoint is inserted.

Unfortunately, KPP triggers a BSOD when it detects modification of the kernel code, such as the breakpoint instruction that the hypervisor inserted. In order to prevent KPP from detecting the breakpoint, the hypervisor marks the page containing the breakpoint is not readable in the SLAT. The code in the page can run without intervention of the hypervisor (with the exception of the `INT3` instruction); however, any attempt to read the page is intercepted by the hypervisor. Normally, pages containing code are read only by the KPP mechanism. The hypervisor reacts to such read attempts by emulating the read instruction, advancing the instruction pointer and returning to the VM. During the emulation, the hypervisor re-

Original `NtCreateFile`:

```
                    NtCreateFile:
0xfffffffff6b2901c0:    sub rsp, 88h
0xfffffffff6b2901c7:    xor eax,eax
0xfffffffff6b2901c9:    ...
```

Instrumented `NtCreateFile`:

```
                    NtCreateFile:
0xfffffffff6b2901c0:    int3
0xfffffffff6b2901c1:    int3
0xfffffffff6b2901c2:    <unreachable fragment>
0xfffffffff6b2901c7:    xor eax,eax
0xfffffffff6b2901c9:    ...
```

Figure 1: NtCreateFile before and after instrumentation.

turns the original value if KPP attempts to access the `INT3` instruction.

3.2 Interception of System Call Returns

Interception of returns from system call handlers is a more challenging task. The idea is to change the return address of the call frame such that the instruction at the new address will trigger the hypervisor. The hypervisor reacts to this event by setting the instruction pointer to the original return address and returning control to the VM.

More precisely, the hypervisor installs two `INT3` instructions at the beginning of a function that needs to be intercepted. Figure 1 shows an example of `NtCreateFile` instrumentation. The purpose of each `INT3` instruction is to trigger the hypervisor. The first `INT3` instruction triggers the hypervisor on an entry to the intercepted function while the second `INT3` triggers the hypervisor on an exit from the intercepted function. The hypervisor reacts to the first `INT3` by setting the return address in the call frame to the address of the second `INT3`. Upon an exit from the intercepted function the second `INT3` is executed, triggering the hypervisor. The hypervisor reacts to the second `INT3` by setting the instruction pointer to the original return address. The hypervisor does not have to record the original return address before changing it in the call frame because this address is identical for all the system call handlers. Algorithm 1 summarises the operation of the hypervisor with regard to breakpoint handling.

Algorithm 1: Algorithm for breakpoint handling. In the case of `NtCreateFile`, InstructionLength is 7 and EmulateInstruction() is equivalent to `RSP := RSP - 0x88`.

```
 1: handled := FALSE
 2: for all intercepted functions do
 3:     if RIP = Address of 1st INT3 then
 4:         handled := TRUE
 5:         HandleEntry()
 6:         STACK[RSP - 8] := RIP + 1
 7:         RIP := RIP + InstructionLength
 8:         EmulateInstruction()
 9:     else if RIP = Address of 2nd INT3 then
10:         handled := TRUE
11:         HandleExit()
12:         RIP := OriginalReturnAddress
13:     end if
14: end for
15: if not handled then
16:     Inject the exception to the OS
17: end if
```

4 SYSTEM DESIGN

The main component of the DLP-Visor is the system call interception mechanism that was described in Section 3. During its initialization, DLP-Visor configures interception of writes to the `IA32_LSTAR` or `IA32_SYSENTER_EIP` MSRs, depending on the operating system architecture. When the operating system writes to the appropriate MSR, the hypervisor assumes that the kernel's initialiwation is complete and installw the breakpoints in the system call handlers that require interception. The hypervisor can obtain the addresses of the system call handlers from SSDT or from symbol files. Each entry in the SSDT contains the address of the system call that corresponds to this entry. The indexes of the entries can be hardcoded in the hypervisor or obtained synamically by disassembling the user-mode wrappers.

After completing the breakpoint installation process, DLP-Visor resumes the operating system. The operating system executes normally and DLP-Visor is notified about each entry to- and exit from the system calls that require interception. Table 1 summarises the intercepted system calls and the action performed by DLP-Visor on each system call.

The system calls intercepted by the DLP-Visor can be broadly divided into two categories: (a) informative and (b) operational. Informative system calls modify the internal data structures of the DLP-Visor and the DLP-Visor never blocks these system calls. Operational system calls are those that require filtering. For example, `NtCreateUserProcess` is an informative system call, whereas `NtWriteFile` is an operational system call.

DLP-Visor uses two sets in its internal data structures and an additional set for each critical process:

- S — the set of sensitive files and directories;
- C — the set of critical processes; and

Table 1: Intercepted System Calls. The current and the target (where applies) processes are denoted by p and q, respectively. The directory of the target file f is denoted by d.

System Call	When	Type	Action
NtCreateFile/NtOpenFile	Exit	Informative	If d is sensitive, mark p as critical.
NtCreateFile/NtOpenFile	Exit	Informative	Store the directory d of the target file.
NtCreateUserProcess	Exit	Informative	If p is critical, mark q as critical.
NtWriteFile	Entry	Operational	Block if p is critical and d is not sensitive.
NtDeviceIoControlFile	Entry	Operational	Block if p is critical.
NtSetInformationFile	Entry	Operational	Block if p is critical and f is sensitive.
NtReadVirtualMemory	Entry	Operational	Block if q is critical.
NtWriteVirtualMemory	Entry	Operational	Block if p is critical.
NtUserSetClipboardData	Exit	Informative	If p is critical, mark the clipboard as sensitive.
NtUserGetClipboardData	Exit	Informative	If the clipboard is sensitive, mark p as critical
NtSetValueKey	Entry	Operational	Block if p is critical.

- H_p — the set of handles to sensitive files in a critical process $p \in C$

Initially, set S contains the directories that were selected by the system administrator as sensitive and set C is empty. Set S may contain a special item $\diamond$ denoting the clipboard. An empty set H_p is allocated for each newly detected critical process $p \in C$. For simplicity, we write $f \in S$ if f belongs to S or if f resides in a directory that belongs to S.

DLP-Visor prevents the execution of system calls by corrupting their arguments. Specifically, `NtWriteFile` and `NtDeviceIoControlFile` receive a file handle as their first argument. By setting this argument to `INVALID_HANDLE_VALUE` (0), DLP-Visor causes these system calls to fail.

5 EVALUATION

This section presents the evaluation of DLP-Visor with respect to security guarantees and performance overhead. The system was tested in a virtualized environment. The exact configuration of the testing environment is presented in Table 2.

Table 2: Testing environment configuration.

Host CPU	Intel(R) Core(TM) i7-10610U
Host memory	16 GB
Host OS	Ubuntu 20.04.1 LTS
VMM	VMware Workstation 15.5.6
Guest CPU	Intel(R) Core(TM) i7-10610U
Guest memory	8 GB
Guest OS	Windows 10 (19041)

5.1 Security

DLP-Visor provides the following security guarantee: *information that resides in sensitive directories cannot be copied outside of them*. We assume that an attacker does not have kernel-mode privileges and, therefore, cannot observe the memory or the file system directly. The attacker is forced to use system calls to exfiltrate information from sensitive files. A process becomes critical on the first attempt to access sensitive files. Therefore, the guarantee of DLP-Visor can be formulated as follows: *Critical processes can output their data only to files that reside in sensitive directories*.

We identify the following channels through which critical processes can leak information:

- files — using the `NtWriteFile` or the `NtSetInformationFile` system call;
- Inter-Process Communication (IPC) — using the `NtWriteFile` system call;
- clipboard — using the `NtUserSetClipboardData` system call;
- drivers — using the `NtDeviceIoControlFile` system call;
- registry — using the `NtSetValueKey` system call; and
- memory — using the `NtWrite/ReadVirtualMemory` system call.

The current implementation of DLP-Visor prevents critical processes from writing to drivers (`NtDeviceIoControlFile`), the system registry (`NtSetValueKey`) and the memory of another process (`NtWriteVirtualMemory`). Likewise, DLP-Visor prohibits reading from the memory of a critical process (`NtReadVirtualMemory`). In additional, DLP-Visor prohibits moving (renaming) sensitive files using the `NtSetInformationFile` system call. Writing to file handles that correspond to: files that reside outside sensitive directories, mail slots, and pipes is prohibited as well. The sensitive data can pass freely

between critical processes using the system clipboard, but the critical processes will not be able to output this data outside the sensitive directories.

The current implementation of DLP-Visor is restrictive in terms of functionality available for critical processes. In order to assess the applicability of DLP-Visor to real-world applications, we selected a set of common desktop applications. All applications worked normally, until we pasted a sensitive text. The behaviour of the tested applications after becoming critical was as follows:

- *Microsoft Word* and *Microsoft PowerPoint* after pasting sensitive data, worked normally but failed to save a file in a non-sensitive directory. An error message appeared in response.
- *Microsoft Outlook* after pasting sensitive data, worked normally but it failed to send an email. An error message appeared in response.
- *Microsoft Edge* after pasting sensitive data, displayed an error message about loss of internet connection.
- *FileZilla* on an attempt to transfer a sensitive file to a remote FTP directory, an error message appeared.
- *PuTTY* on an attempt to transfer a sensitive file via SFTP, aborted the connection.
- *Firefox* and *Chrome* terminated after pasting sensitive data due to excessive use of inter-process communication.

In most test cases, the applications behaved in a predictable fashion: after becoming critical, the applications continue to function normally but with certain limitations. Notable exceptions are *Firefox* and *Chrome*, which terminated after becoming critical. This can be explained by the design of these applications. *Firefox* and *Chrome* consist of multiple processes that use inter-process communication mechanisms. Unfortunately, the current implementation of DLP-Visor blocks these mechanisms, resulting in hangs and termination of these applications. We plan to address this issue in future versions of DLP-Visor.

5.2 Performance

DLP-Visor intercepts several system calls. Some of these system calls, notably the `NtWriteFile` system calls, are frequently called during normal execution. In order to assess the performance degradation due to these interceptions, we ran a benchmarking tool called PCMark (Sibai, 2008) in four configurations: (a) without a hypervisor, (b) with a thin hypervisor without interceptions, (c) with DLP-Visor and (d) with Oracle VirtualBox (6.1.14) as an example of a full hypervisor, running the same version of Windows with 8GB of RAM. The exact configuration of our testing environment is summarised in Table 2. Table 3 presents the scores that were given by PCMark in each configuration for every category. As expected, the performance degradation of a thin hypervisor is negligible ($\approx 1.7\%$ on average), whereas the performance degradation of a full hypervisor is unacceptably high ($\approx 38.6\%$ on average). The average performance overhead of DLP-Visor is $\approx 7.2\%$. We believe that this performance degradation will not have a considerable negative impact on user experience.

Table 3: PCMark scores in four configurations.

Category	No Hypervisor	Thin Hypervisor	DLP-Visor	VirtualBox
App start-up	7277	7110	6358	4008
Video	2129	2307	1862	1197
Web browsing	2067	2012	2028	1390
Spreadsheet	4473	4420	4215	2688
Writing	5005	4928	4861	2870
Photo editing	850	852	817	524
Video editing	903	887	804	650

In some tests, the results seem counter-intuitive due to measurement errors. For example, in the "Video conferencing" test, the system performed better in the "Thin hypervisor" configuration than without a hypervisor. Another example is the "Web browsing" test, in which DLP-Visor performed better than a thin hypervisor.

6 RELATED WORK

DLP-Visor is based on two somewhat independent fields in software security: data leakage prevention and virtual machine introspection. In this section, we describe the similarities (and dissimilarities) among previous works in these fields and DLP-Visor.

Data leakage prevention systems can be divided into two categories (Alneyadi et al., 2016): context-sensitive and content-sensitive. Content-sensitive DLP systems use regular expressions, statistical methods or advanced hashing techniques to identify sensitive data exfiltration attempts (McAfee, 2020; Microsoft, 2020a; Google, 2020; Checkpoint, 2020; Shvartzshnaider et al., 2019). As such, these methods are not able to identify leakage of transformed or encrypted data.

Some context-sensitive DLP systems work according to coarse security policies, such as preventing users from using removable media (Halpert, 2004). DLP-Visor provides a more fine-grained configuration of sensitive and restricted locations.

The idea of sensitiveness that spreads between files touched by processes was first introduced in (Petkovic et al., 2012). The authors proposed a kernel module for the Linux operating system that monitors the operations performed on the filesystem. Specifically, after a process performs a read from a sensitive file, all its subsequent writes mark the target files as sensitive. DLP-Visor generalises this idea to the clipboard, memory and network channels, and adapts to the Windows operating system.

UC4Win (Wüchner and Pretschner, 2012) is probably closest to DLP-Visor's approach. UC4Win monitors system calls and matches them against a set of predefined rules. The rules determine whether the system call shall be allowed. Unlike DLP-Visor, UC4Win uses user-mode interception of system, which can be easily circumvented.

Virtual machine introspection allows the hypervisor to intercept various events occurring in the operating system. The first system to use virtualization extensions for introspection was Ether (Dinaburg et al., 2008), which used page-faults for system call tracing. We showed that interception of page-faults can severely degrade overall system performance.

Spider (Deng et al., 2013) is a stealthy breakpoint installation framework based on KVM, a full hypervisor. Spider is suitable for installing breakpoints in user-mode applications. The idea of stealthy breakpoints was later extended in Drakvuf (Lengyel et al., 2014), which is based on Xen, another full hypervisor. As shown by our experimental results, full hypervisors have a much higher performance overhead than thin hypervisors, like DLP-Visor. Recently, Drakvuf was ported to ARM (Proskurin et al., 2018), Because Drakvuf and DLP-Visor use similar underlying mechanisms, we believe that DLP-Visor can be ported as well.

7 CONCLUSIONS

In this paper, we presented DLP-Visor, a hypervisor-based context-sensitive data leakage prevention system. We showed that the performance overhead allows DLP-Visor to be deployed in practice. Despite the limitations imposed by current implementation of DLP-Visor, it can be applied to most real-world applications. We believe that future versions of DLP-Visor will address these limitations.

REFERENCES

Alneyadi, S., Sithirasenan, E., and Muthukkumarasamy, V. (2016). A survey on data leakage prevention systems. *Journal of Network and Computer Applications*, 62:137–152.

Checkpoint (2017 (accessed Sep 19, 2020)). *Data Loss Prevention Software Blade*. Checkpoint.

Cook, K. (2013). Kernel address space layout randomization. *Linux Security Summit*.

Deng, Z., Zhang, X., and Xu, D. (2013). Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 289–298.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62.

Field, S. (2006). An introduction to kernel patch protection.

Google (2016 (accessed Sep 19, 2020)). *Scan your email traffic using data loss prevention*.

Guo, C., Zhuang, R., Jie, Y., Ren, Y., Wu, T., and Choo, K.-K. R. (2016). Fine-grained database field search using attribute-based encryption for e-healthcare clouds. *Journal of medical systems*, 40(11):235.

Halpert, B. (2004). Mobile device security. In *Proceedings of the 1st annual conference on Information security curriculum development*, pages 99–101.

Kantor, A., Antebi, L., Kirsch, Y., and Bialik, U. (2012). Methods for document-to-template matching for data-leak prevention. US Patent 8,254,698.

Khati, L., Mouha, N., and Vergnaud, D. (2017). Full disk encryption: bridging theory and practice. In *Cryptographers' Track at the RSA Conference*, pages 241–257. Springer.

Lengyel, T. K., Maresca, S., Payne, B. D., Webster, G. D., Vogl, S., and Kiayias, A. (2014). Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 386–395.

Li, J., Chen, N., and Zhang, Y. (2019). Extended file hierarchy access control scheme with attribute based encryption in cloud computing. *IEEE Transactions on Emerging Topics in Computing*.

McAfee (2017 (accessed Sep 19, 2020)). *Total Protection for Data Loss Prevention (DLP)*.

Microsoft (2018 (accessed Sep 19, 2020)b). *Microsoft public symbol server*.

Microsoft (2020 (accessed Sep 19, 2020)a). *Data loss prevention in Exchange Server*.

Petkovic, M., Popovic, M., Basicevic, I., and Saric, D. (2012). A host based method for data leak protection by tracking sensitive data flow. In *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, pages 267–274. IEEE.

Proskurin, S., Lengyel, T., Momeu, M., Eckert, C., and Zarras, A. (2018). Hiding in the shadows: Empowering arm for stealthy virtual machine introspection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 407–417.

Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350.

Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., et al. (2009). Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130.

Shvartzshnaider, Y., Pavlinovic, Z., Balashankar, A., Wies, T., Subramanian, L., Nissenbaum, H., and Mittal, P. (2019). Vaccine: Using contextual integrity for data leakage detection. In *The World Wide Web Conference*, pages 1702–1712.

Sibai, F. N. (2008). Evaluating the performance of single and multiple core processors with pcmark® 05 and benchmark analysis. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):62–71.

Tourani, R., Misra, S., Mick, T., and Panwar, G. (2017). Security, privacy, and access control in information-centric networking: A survey. *IEEE communications surveys & tutorials*, 20(1):566–600.

Wüchner, T. and Pretschner, A. (2012). Data loss prevention based on data-driven usage control. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 151–160. IEEE.