Security by virtualization

A novel antivirus for personal computers

Balduzzi Marco, 2nd of March 2007

Master Thesis in Computer Engineering, University of Bergamo

This page has been left intentionally blank.

Content

1. Introduction

A sort of virtualization appeared four decades ago to perform multi-programming and simple time-sharing tasks inside a single mainframe. Virtualization became quickly the solution to limit cost and save money by server consolidation: aggregate the workload of several under-utilized servers to fewer machines reduces hardware and management resources. Nowadays virtualization is a “hot topic”, being habitually adopted in develop environment for testing and debugging purposes.

A novel paradigm to secure personal computers is presented. Virtualization creates inside a second operating system *(security shell)* an isolated and encapsulated environment, in which the user system is moved and protected. The security layer is decoupled in this inaccessible system that, through special services, ensures the user environment’s tamper resistance. The security policies defined by administration are deployed from a central site to the single computer and locally enforced.

While conventional *personal antivirus* can be switched off, manipulated, or avoided by sophisticated malign codes and technically experienced users, the antivirus designed in the security shell enforces a continuous protection of the user environment, against viruses and malware.

Three complementary services are run: the virtual disk is scanned on demand or together with computer start-up and backup policies; the network connections established over embedded and removable devices, as well as encrypted protocols, are inconspicuously inspected before reaching the user environment; and the file-system accesses to disk and removable mass-storage are scanned for real-time viruses.

The latter component *(on-access scan)* analyzes the content of files as they are accessed by the virtual environment. A special driver installed into virtual machine intercepts the disk operations and builds a logical representation of files' structure. With the read access of an application or a system service, a reference to involved blocks is inserted or looked up in a cache. The antivirus uses the cache's information to address the data for scanning. This cache should be efficient and fast enough to not waste memory or "bottleneck" the user system. At the same time, a robust cache avoids hangs and problems of synchronization and inconsistency.

Further existing security technologies benefit of virtualization, e.g. *disk encryption*. An embedded layer encrypts inconspicuously the user system, requiring no configuration or encryption support. A pre-boot authentication secures the access to whole personal computer.

1.1. Motivations

The goals of information security are described by the *CIA paradigm*, in terms of confidentiality, integrity, and availability. Confidentiality is defined as "ensuring that information is accessible only to those authorized to have access" and is traditionally represented as a relation between user and system. Cryptography is usually used to protect the confidentiality of stored and transmitted data.

Integrity is the system's ability to protect information from unauthorized modifications. In particular, this attribute assures that transmitted data are not altered and that the sender is who it is supposed to be. The encryption's digital signature and hashing are traditional solutions to provide integrity.

Availability is the system's capacity to offer to user a sure and immediate access to own resources: data and services have to be immediately available when required. Redundant network architecture, high-availability protocols, and hardware with more failure points ensure this attribute.

The CIA paradigm, despite of its linear and simple definitions, represents an "ideal world"; in real applications, complex and obfuscated issues arise and security is an "a priori" lost challenge. No solution establishes a complete protection against threats crashing against the computer infrastructure: unqualified users, newbie, evil users, viruses, and worms try out constantly the computer's CIA triad. Many scenarios can be proved.

When a laptop is lost or stolen, any information is easy to be extracted by passing any form of authentication: the hard disk can be installed into a second computer for being read, or the system can be started with a "live-CD" and data accessed. Disk encryption is considered a good solution to protect the confidentiality of personal computers, where the operating system offers this feature. However, quite few of them enable complete disk encryption[1], as metadata resources, temporary files and swap partition, and commercial products are often too expensive solutions to be adopted.

[1] E.g. Windows 2000, 2003 and XP support just a simple "file-system encryption" enabling specific folders and file to be encrypted, while the new Windows Vista embeds a "disk encryption" features termed BitLocker.

Encryption is used in network communications too, where special tools (e.g. *sniffers*) permit to an evil user to intercept the transmitted data and analyze it for confidential information as account, email, and credit card number. VPNs and cryptographic protocols as TLS/SSL are widely used for secure transmitted information's confidentiality and integrity. On the other hand, encrypted streams (especially those "handled" at application-layer) are not protected against viruses and malware because their content cannot be analyzed by the antivirus located in the network sub-layer.

Virus protections are threatened by user acts too: personal antivirus can be switched off, manipulated and avoided by intelligent malign codes and technically skilled users, so that the system stays unprotected and exposed to easy threats and attacks.

System confidentiality could be compromised just by a single "click" that involuntarily connects the user to "untrusted" networks: usually, computer users are not technicians, so they could be easily cheated for using special crafted hotspots configured as traps, and sending confidential information to them. User data is stolen and confidentiality is violated.

Similarly, *personal firewall* could be badly tailored by unqualified users that react inadequately to the firewall's "questions". In fact, the firewall filters the network connections using an application access-list interactively populated via user interaction: once a specific application is user-permitted, all traffic "related" to it is accepted. Nimbly, a negligent and distracted user's behaviour exposes the system to risks as remote exploiting and denial-of-services.

Protecting network usages is a primary need as well as enforcing a strict control of devices. The ability of users to add new hot-plug hardware, such as USB-sticks, does

not only make computers harder to maintain when users use them to install unsupported hardware, but they can pose threats to data security. A malicious user can potentially use a removable storage device to steal confidential information. An attacker also could "autorun" custom scripts stored on the device, installing malicious software such as spyware and Trojan-horses.

Virtualization is the answer to these security issues. Decoupling the security layer from the user system into an isolated and hardened *shell* prevents the exploiting of security functions and the subverting of security policies. Attacks to the CIA paradigm are inhibited: evil users, viruses, and malware acting with "administrator privileges" are inoffensive. By virtualization, the security services are not directly attackable, unless the security shell is compromised. In this way, the user environment is constantly monitored and secured from a set of external services.

This thesis is organized as follows. In chapter 2, the conventional application of virtualization technology and its new paradigm to secure personal computer are presented. In chapter 3, the novel antivirus approach is illustrated. The next three chapters deepen design and implementation of this virus protection: *image scan*, *on-access scan*, and *network scan* components are described. Chapter 7 proposes *disk encryption* as an evident benefit of coupling virtualization with existing security technologies. The last chapter summarises main challenges and open issues.

2. Security by virtualization

2.1. Introduction to computer virtualization

Computer virtualization allows running multiple operating system instances concurrently on a unique *host* computer, decoupling the hardware requirements from a single system and making it available to the whole "community" of operating systems. The virtual OS is managed by a special *Virtual Machine Monitor (VMM)* application, located between hardware and guests that provides a layer of abstraction for the computer hardware. Virtual machines create virtual devices as CPU, memory, network and storage to assign to the guest OS. By virtualization it is possible to control the access to hardware devices, enabling specific sets of devices for each guest OS. For example, it is possible to assign network devices to a system operating via network and hide them from a local one. At the same time, the second guest would require access to USB devices and multiple processors for heavy computational application. Virtual machines can provide the illusion of hardware, or hardware configuration that is not installed (such as SCSI devices) and be used to simulate special network infrastructures and scenario, for distributed applications.

Virtualization allows limiting and controlling the resources assigned to each guest. This distribution eliminates the danger of a single runaway process, consuming all available memory or CPU. On the other hand, it permits to assign more resource to greedy systems for specific applications. Since the guest is not bound to the hardware, it becomes possible to dynamically move an operating system from one physical machine to another. As a particular guest OS begins to consume more resources, during a peak

period, the offending guest can be moved to another server with less demand. With virtual deployments, it is possible to relocate an operating system to receive the resources needed at that time.

Virtual machines have been historically used for *server consolidation* purpose, aggregating the workloads of several under-utilized servers to fewer machines, perhaps a single machine. Nearly all mainframes have the ability to host multiple operating systems and thereby operate not as a single computer but as a number of virtual machines. In this role, a single mainframe can replace dozens or even hundreds of smaller servers. Sometime, the need of running legacy applications that does not run on newer hardware/OS and require very few resources is served well by virtualization technology. Benefits are related to saving hardware resources, reducing installation, management, and administration costs, while providing greatly improved scalability and reliability.

Virtualization provides independent and isolated environment, for running operating system with different levels of trust or critical applications. When a guest is virtualized, the hardware devices and resources are confined and protected; no interaction among different guests or host computer is possible. In the newer system, as Microsoft Vista, the virtual machine is loaded on the fly, creating such an execution environment dynamically, for encapsulate components considered vulnerable, as internet browsers and relative downloads.

Virtualization is also important to developers. Operating system's kernel occupies a single address space, which means that a failure of any driver results in the entire operating system crash. Using a virtual machine is possible to debug a kernel code similar to a standard application, restarting the execution from the host system. It is also possible to intercept device accesses, debugging their usage.

History and types

A sort of virtualization has been used since four decades ago, when Compatible Time Sharing System developed by the Massachusetts Institute of Technology was adapted for the IBM 704 mainframe. This system, splitting the CPU time in quantum (atomic entities assigned to single work), permitted the concurrent execution of multiple task, as independently user shells. IBM understood the importance of concurrency, offering much more services while keeping the same hardware and succeeding in cost reduction optimization. The virtualization as it is known to us was implemented in the Model 67, where all the hardware interfaces where virtualized through a *Virtual Machine Monitor*. In the following years, quite many approaches to virtualization were proposed, and nowadays the same results are achieved in several ways through different levels of abstraction. Among all, at least the following four types are remarkable:

- *hardware emulation* consists in emulating the complete set of CPU instructions for a desired architecture. With this approach it is possible to run unmodified operating systems intended for a different platform, for example ARM on an x86 processor host. Hardware emulation is used for firmware development: rather than wait until the real hardware is available, hardware VM supports the validation of many aspects

of the actual code in simulation. The trade-off is a significant performance hit: in fact, modelling a fairly complete architecture in software is extremely slow. *Bosch*[1] is such a kind of emulator;

- *full (native) virtualization*, known briefly as *virtualization*, uses a virtual machine monitor that mediates between the guest operating systems and the native hardware. Certain protected instructions must be trapped and handled within the VMM because the underlying hardware is not owned by an operating system but is instead shared by it through the VMM. Unmodified operating systems are run; the underlying hardware must be natively supported. The full virtualization is a good trade-off between performance and capabilities; currently it is the most used type of virtualization (*VMWare*[2]);
- *para-virtualization* is the most recent approach, currently supported by many vendors as *XEN*[3], and consists in integrating the virtualization-aware code into the operating system itself, increasing the performance nearly to that of a native system. The disadvantage is that para-virtualization requires the guest operating systems to be modified for the VMM. Intel and AMD recently support this technology in hardware[4], permitting extremely fast execution of unmodified guests. Para-virtualization it is expected to be the solution for future;

---

[1] Bosch IA-32 emulator is freeing downloadable from http://bochs.sourceforge.net/. It's Free Software

[2] VMWare virtualization, http://www.vmware.com/

[3] XEN software homepage is http://www.xensource.com/

[4] Intel VT (http://developer.intel.com/technology/virtualization/index.htm) and AMD Pacifica (http://enterprise.amd.com/us-en/Solutions/Consolidation/virtualization.aspx)

- *APIs virtualization:* since applications generally run in user sub-layer and communicate with the OS via a set of APIs, this strategy consists in intercepting and emulating the behaviour of this APIs using facilities in the existing OS. A pleasant side effect is that application binaries can be run natively. On the negative side, this approach works for running a single application or operating system, for which a special implementation has been done. *Wine*[1] is a well-known project making use of APIs virtualization for running Windows applications.

Performance test[2]

This section presents the result of a performance test run on five different virtual machines[3], in which the Innotek VBOX 1.1.6 product has been used as reference and a standard personal computer[4] configuration has been adopted.

---

[1] Wine project homepage is http://www.winehq.com/. It's Free Software

[2] Pass Mark Software Pty Ltd, Performance Test 6.0, http://www.passmark.com/

[3] The evaluated VMM are:

- Innotek VBOX, Version 1.1.6 (18 April 2006) and 1.1.10 (28 July 2006)
- VMWare Workstation 5.5.1 (build-19175)
- QEMU 0.8.2 (22 July 2006) with and without accelerator (KQEMU)

[4] The personal computer is configured with:

- Processor: Intel(R) Pentium(R) 4 CPU 2.80GHz (5630.34 bogomips)
- Memory: 1Gb RAM (Windows has been virtualized with 256Mb RAM)
- Hard disk: Maxtor 4D040H2, 40Gb, 2Mb Cache, UDMA-100
- VGA card: Elsa Erazor III LT equipped with a RIVA TNT2 Model 64

All the virtual machine suites have accomplished good results regarding the CPU and memory management, while the 2D marks are in average worse. However, the bad graphic result could be acceptable for standard computer applications, as office and internet, where the system is presumed to run.

Globally the VMWare suite and newer version of VBOX ran quite well, losing respectively 22% and 33% of performance in comparison with a native Windows. QEMU is much more slowly, especially when the accelerator module is not enabled. Anyway, this product has represented a good software asset for research and test purposes: the free license has permitted to hack the code, integrating QEMU in antivirus service. In particular, QEMU has been modified to handle antivirus cache's structures. Completed the research phase, commercial solutions will be adopted, integrating a faster and more reliable virtual machine.

Figure 1 - VMM performance test

2.2. Virtualization approach to security

Information that once has been stored exclusively on mainframes is now distributed on the network among several personal computers. Thin clients that were used to access central managed resources are now equipped with a robust workspace: local applications and data. The high costs that have compelled for asset aggregation are no longer a constraint to many single installations. Virtualization that has traditionally been used for server consolidation purposes is here involved in securing personal computers.

The security of the whole computer infrastructure is achieved by protecting each single host through a local virtualization.

Virtualization makes it possible decoupling security services and user environment: security is deployed in a separate second operating system *(security shell),* over which the user system runs virtualized. Virtualization creates an isolated and encapsulated environment that protects the user system by various services of the security shell. This detached shell is invisible and inaccessible from the top, and the user environment behaves just as a standard personal computer with a hidden security sub-layer.

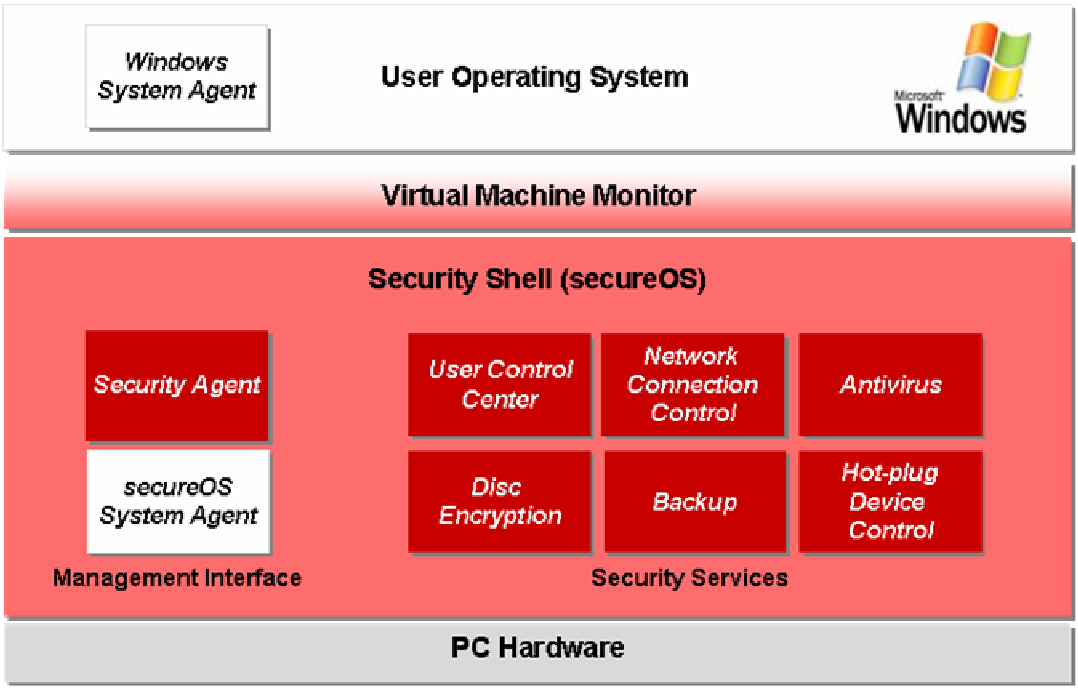


Figure 2 – The new virtualization paradigm

Securing the user system from an isolated sub-layer ensures its tamper resistance: not even a privileged user can exploit the system's security functionalities placed into a

hidden and decoupled environment. In fact the interaction with the security shell is handled via especially protected channels: the *user control center* as interface for the user and the *security agent* to communicate with the central management system.

The security policies define the valid behaviour of the user system and are distributed from a central site to each single computer, where the local agent is responsible to enforce them through various *security services* embedded into the shell. Numerous aspects of security can be hardened by administrators. A *network connection* component enforces a real-time firewall, limiting the access to trusted network resources. The *antivirus*, as will exhaustively presented in the next chapters, performs a continuous virus protection of the user system for viruses and malware coming over disparate channels as networks and removable mass-storage devices. The image scanning should be performed offline to avoid inconsistency problems. For this reason, it is usually coupled with the boot and backup processes. The *disk encryption* feature secures computer against theft and loss, while protecting data confidentiality from unauthorized access. Encryption is realized automatically and inconspicuously to user-system in which no configuration or encryption support is required. Encryption's keys could be managed remotely, enabling a secure recovering procedure. The virtual machine is extended with a *device control* mechanism that wraps the removable device usage and avoids a malign use, e.g. installing illegitimate software or stealing confidential information.

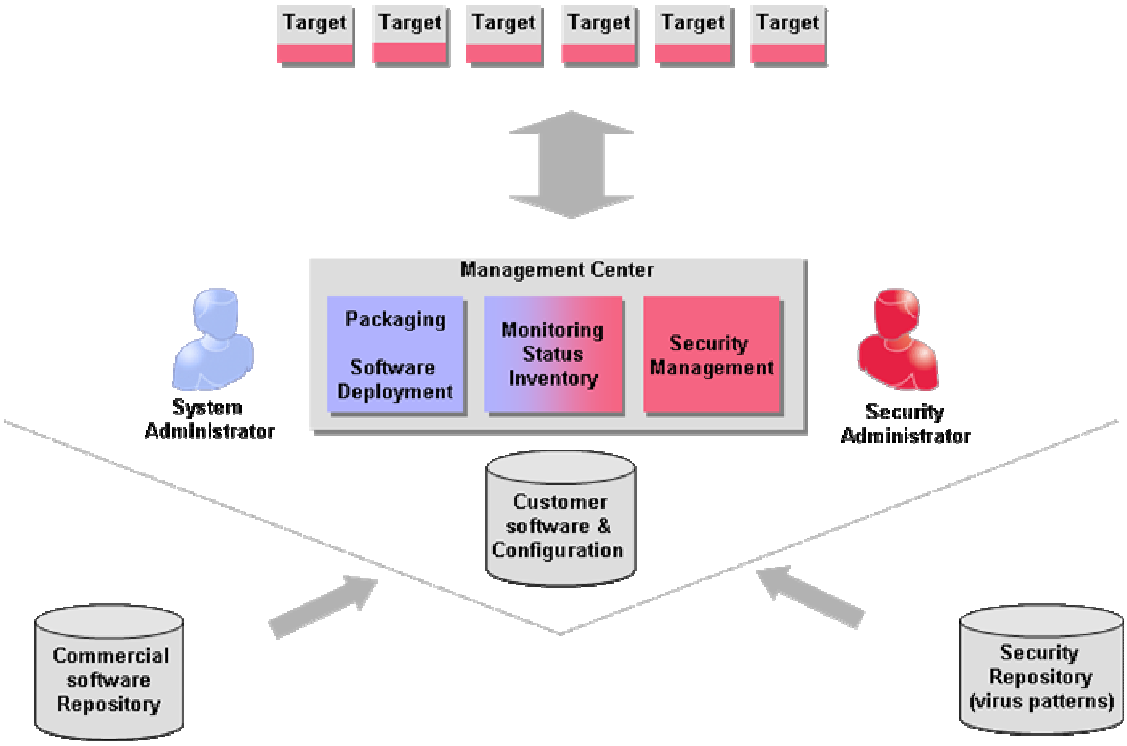


Figure 3 - Security and system management

Virtualization enables the homogeneous management of disparate personal computers by hiding the differences among user systems with a standard interface: Windows, GNU/Linux, or Solaris run over the security shell for being together managed. The management of system- and security- components can be tightly integrated into the same interface. Different policies, regarding security and system, could be aggregated and deployed at the same time.

3. Virus protection

Personal computers, in particular those configured with the Windows OS, are increasingly threatened by viruses and malware, spreading over email, http download, and external mass-storage devices, such as USB-sticks and floppy disks. The worldwide spread of Microsoft applications and standard Internet suites has encouraged virus coders to consider them as preferable targets, obtaining more visibility and damages. For many years, the answer of antivirus software houses has been *personal antivirus*, tailored for the user environment and installed as system service.

Since malign code's classification is nowadays quite confused due to the rapid evolution of threat techniques, here is given a personal and simple taxonomy, used as reference of the rest of work. A virus is a computer program that distributes copies of itself, installing itself in special locations executed at computer power-on, as the hard disk boot sector and the init scripts of operating system, or infecting clean files as binaries and word documents (macro virus); the virus should be executed to become dangerous. A malware represents a larger class of evil software, mostly spread over network channels and infect remote computer exploiting known vulnerabilities. To this category belong worms, as portable codes using computer networks and security flaws to create copy of them self, spywares and diallers that are unwanted advertising-supported software, activated automatically to present advertises or dialup premium-rate numbers, and Trojan-horses, "back-doors" that consent hidden connections to infected computer.

In conventional virus protection, an *antivirus driver* lies into the operating system as kernel service, integrated between system libraries and file-system layer. The personal antivirus console is used for its management from the user environment. This driver intercepts operations on files, coming from the user environment over the *WINDLL* on the file-system. At each reading access, application's code execution is temporarily arrested and the involved data blocks are virus scanned. After it, the scanner releases those blocks.

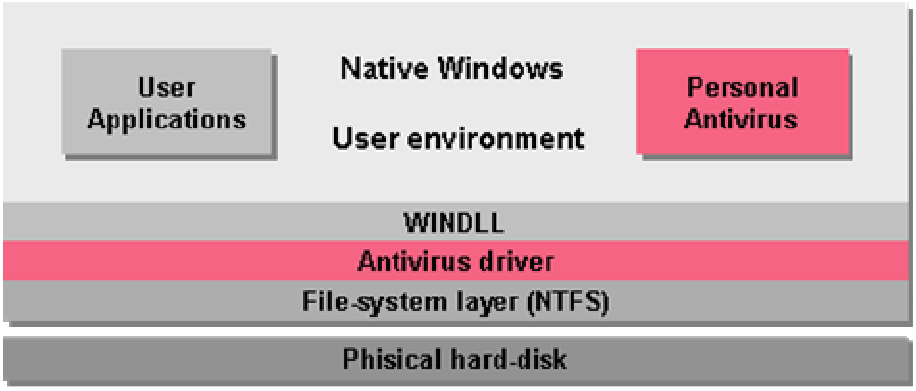


Figure 4 - Personal antivirus approach

In today's threat landscape, personal antivirus alone in the user environment is an insufficient protection. Since it can be switched off, manipulated, or avoided by intelligent malign codes and technically skilled users, the system becomes unprotected and exposed to easy threats and attacks, compromising both user and network security.

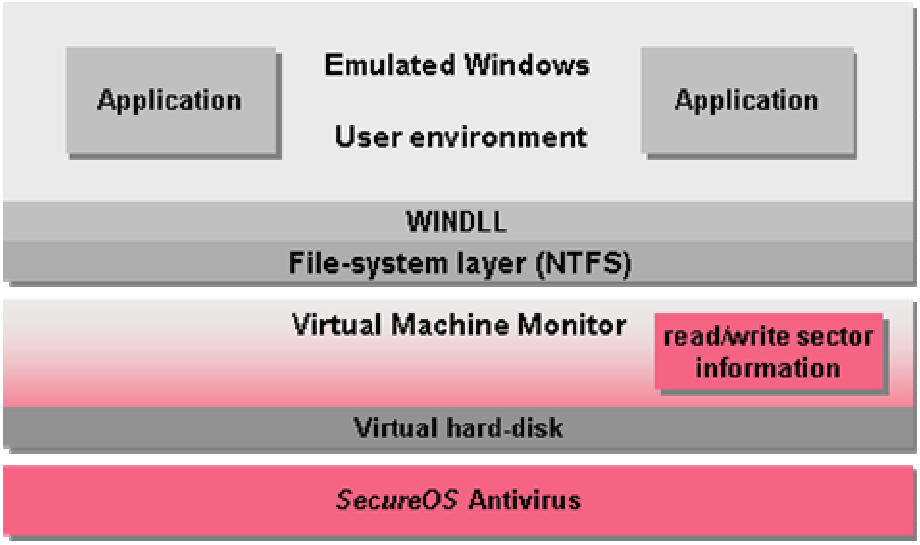


Figure 5 – Novel antivirus approach

In contrast to personal antivirus, the solution discussed here uncouples the virus protection from the user OS into the separate, invisible and inaccessible *security shell (secureOS)*. Virtualization is used for the separation of the two systems. The antivirus components are administrable by the user, through the secure communication channel of the *user control center*. For this reason, the security shell should be hardened against attacks and unauthorized accesses.

This virus scanner is therefore not directly attackable from the top by evil users and malicious codes: voluntary shutdown or protection-avoiding are no longer conceivable. The user system is continuously inspected for viruses and malware coming potentially from different sources, as network connections and mass-storage devices, warranting a steadily high security protection.

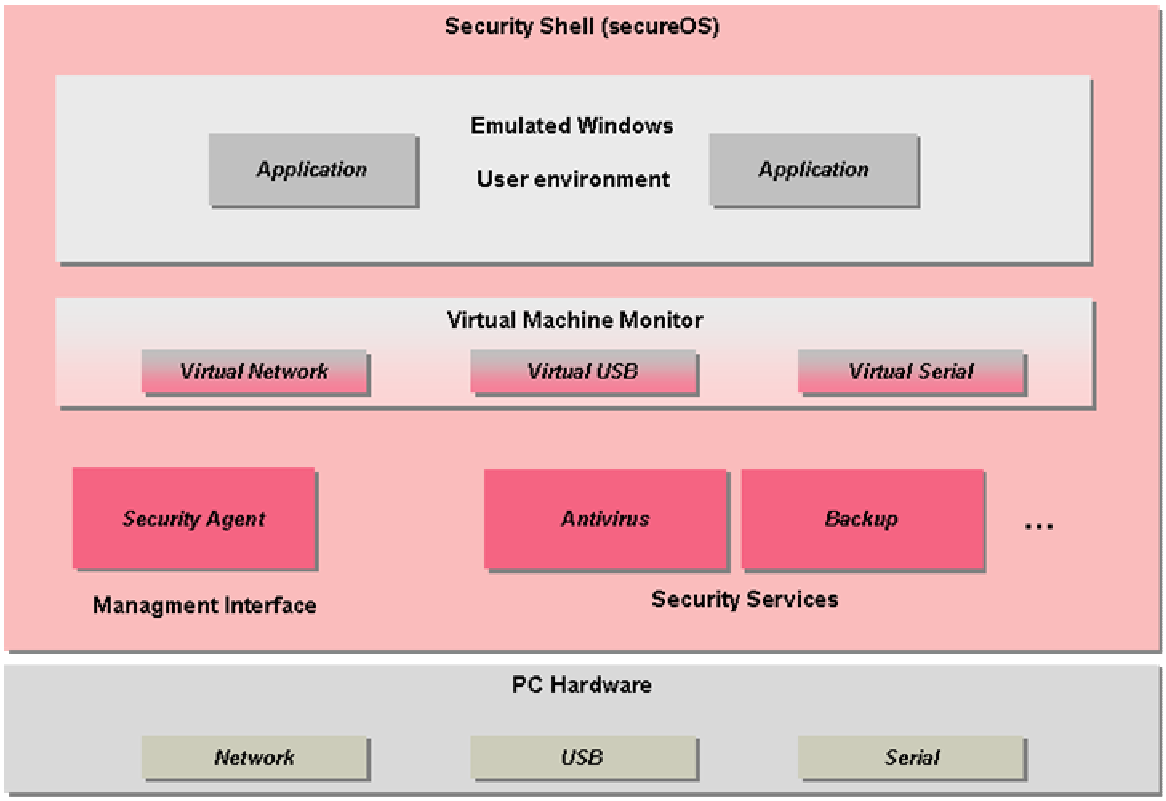


Figure 6 - Novel antivirus design

A *security agent* oversees the *antivirus service*, deploying the antivirus policy defined at central side, and managing the communication to the user. The user system is protected against threats coming over network channels and mass-storage devices. Moreover, the system is regularly scanned and virus-free backups are guaranteed. Three classes of antivirus services are achieved:

- *image scan*: the virtualized system, including Windows core components, applications and user documents, is fully scanned at user demand, computer boot (shutdown), or in conjunction with backup policies;
- *on-access scan*: this innovative feature realizes a virus protection of files at access time. With the reading access of an application and a system service on the file-

system, the involved data blocks are examined by the security shell. A customized driver, installed into the virtual machine, intercepts the virtual disk's operation and builds a logical representation of the file's structure. This permits to scan its content as it is accessed by the user;

- *network scan*: the security shell embeds a sort of transparent proxy that inspects any TCP/IP connection's data, such as email and http download, before it reaches the user environment. Encrypted streams like VPNs and SSL-enabled protocols are scanned too. *Network scan* protects both the local computer and connected LAN segments from viruses and malware.

4. Image scan

Image scan allows to virus-scan a completely virtualized operating system by the security shell, including the user's applications, configurations, and personal data. Image scan permits to secure backup copies, cleaned from virus infections, in order to restore the entire user environment if compromised; it could be seen as a security feature applied to the snapshot technology. Moreover, image scan can be integrated in the computer boot to assure system cleanness, and in the shutdown to avoid time wasting. In fact, a mandatory shutdown prevents inconsistent states. During the scan activity, the disk image could become easily inconsistent with the real content, due to the possible modifications carried out by the user environment. On the contrary, the antivirus "reparations" to the disk's content could damage the state of the user system.

The user system resides on a virtual disk, represented within the *secureOS* as a unique file, whose size is the same of the emulated disk drive. The first partition *(C:)* begins at the 63rd sector, while the preceding contain the MBR and the partition table, as in a normal hard disk structure; afterwards, the disk image could contain additional partitions. This type of disk image is known as *raw-format*, and it is the simplest one: data are written serially and the file is directly managed from the kernel driver as a normal hard disk. For its simplicity, the raw-format is immediately accessible from any GNU/Linux system.

The *fdisk* command shows the structure of the VM image, which contains a single NTFS Windows image of 3GB (3140896 blocks):

```
# fdisk -l -u marco.img

Disk marco.img: 0 MB, 0 bytes

128 heads, 63 sectors/track, 0 cylinders, total 0 sectors

Units = sectors of 1 * 512 = 512 bytes


            Boot    Start        End      Blocks   Id  System

marco.img1   *        63     6281855     3140896+   7  HPFS/NTFS
```

Other advanced formats have better performance, but are more complex to handle and to debug. The QEMU software community has developed an advanced image format called QCOW[1], which provides a smaller file size, even on file-systems that do not support holes (e.g. sparse files), optional zlib[2] based compression and AES[3] encryption. However, neither compression nor encryption are required, since personal computers normally run a single guest operating system and the entire secureOS (in which the VM image is stored) is already encrypted. For these reasons, the raw-format has been used, as confirmed by the *qemu-img* command:

```
# qemu-img info /secunet/marco.img

image: /secunet/marco.img

file format: raw

virtual size: 2Gb - disk size: 2Gb
```

[1] The QCOW Image Format, http://www.gnome.org/~markmc/qcow-image-format.html

[2] Zlib compression library, free-software, http://www.zlib.net/

[3] Advanced Encryption Standard (AES) block cipher

Linux comes with FAT read-write and NTFS read-only file-systems support, permitting to mount Windows partitions in a secure way. By kernel the device loop-back mechanism[1] and the device driver capabilities, the Windows image is associated to the *loop0* device (the first partition starts after 63*512 bytes), and subsequently mounted under /mnt/winOS as *read-only*:

```
# losetup -o $((63*512)) /dev/loop/0 /secunet/marco.img
# mount -o ro /dev/loop0 /mnt/winOS/
```

Once Windows image is mounted, the entire user system can be access from the security shell as a local file-system to be virus scanned.

Clamavis[2] antivirus has been used for research and test purposes. This toolkit is distributed as open source and can be modified due to its Free license. The engine framework (the *libclamAV* library) has been adapted and integrated in the *on-access* scan. The *clamscan* virus scanner has been used in the *image scan* research.

Clamavis have run on a standard personal system, configured with Microsoft Windows XP and Office, for a size of 2G, and has taken about a quarter of hour. The test environment is virus-free.

---

1 Device node that represents a regular file, http://en.wikipedia.org/wiki/Loop_device

2 ClamAV project homepage is http://www.clamav.net/

```
# clamscan --verbose --recursive --log FULL-SCAN.log /mnt/winOS/
/mnt/winOS/AUTOEXEC.BAT: Empty file
Scanning /mnt/winOS/boot.ini
/mnt/winOS/boot.ini: OK
Scanning /mnt/winOS/bootfont.bin
/mnt/winOS/bootfont.bin: OK
/mnt/winOS/CONFIG.SYS: Empty file
/mnt/winOS/MSDOS.SYS: Empty file
Scanning /mnt/winOS/NTDETECT.COM
/mnt/winOS/NTDETECT.COM: OK
[...]
----------- SCAN SUMMARY -----------
Known viruses: 76052
Engine version: 0.88.4
Scanned directories: 593
Scanned files: 7325
Infected files: 0
Data scanned: 2214.39 MB
Time: 980.635 sec (16 m 20 s)
```

From a careful analysis, a big problem arises and forces Windows to temporary hibernation[1], during the full image.

[1] Hibernate is a feature seen in many operating systems where the contents of RAM is written to non-volatile storage, such as the hard disk before powering off the system. Later the system can be restored to the state it was in when hibernating, so that programs can continue executing as if nothing happened. From http://en.wikipedia.org/wiki/Hibernate_(OS_feature).

When the Windows image is mounted with the *mount* command, the Linux's VFS Layer[1] accesses the loop-back device (/dev/loop0 in example) and builds an appropriate cache for data currently available on the image. Therefore, through a secure and sturdy mechanism, the user system is available as a local file-system to the secureOS environment for reading/writing operations. Processes can safety access the image's data from this location, by VFS layer.

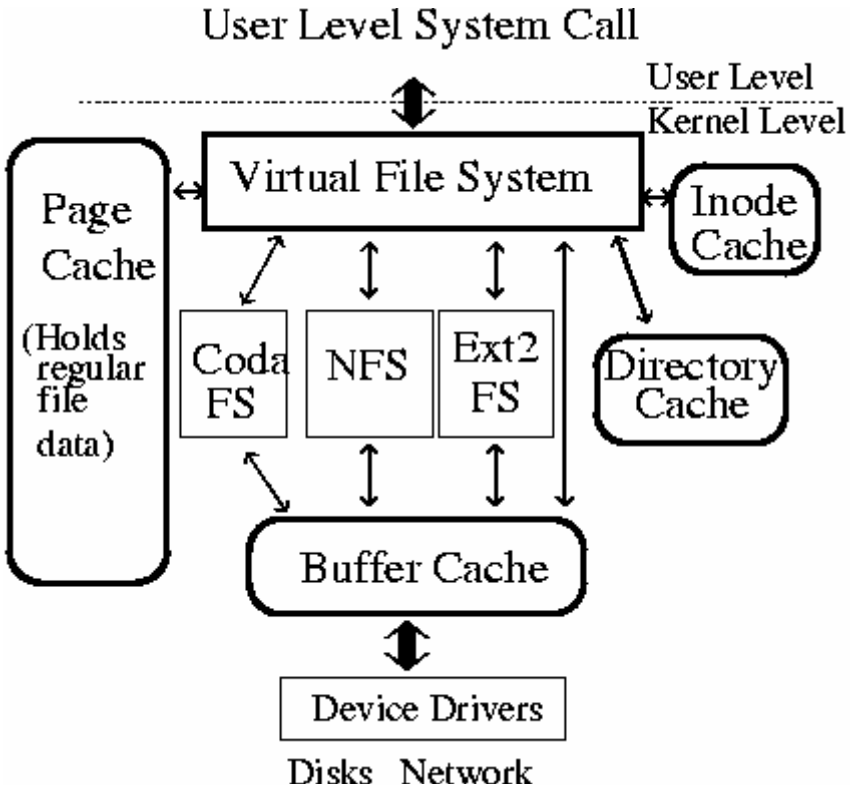


Figure 7 - Linux Virtual File-system (VFS)

[1] Virtual file-system is a kernel software that handles all system calls related to a standard Unix file-system. Its main strength is providing a common interface to several kinds of file-systems.

However, whatever an external process modifies the image, the antivirus cache becomes immediately inconsistent. For example, when a file is created/modified by the user system, the *virtual machine* updates the image's content and the Windows file-system layer is correctly updated to address the new data location. The antivirus cache, instead, cannot be updated: the writing has been done on the disk, skipping the Linux virtual file-system. Therefore, new files are not addresses from the security shell: they physically exist on the image file, but are completely invisible to a *list* command.

The following test shows how a *bar.txt* document is accessible only after remounting user image:

```
# ls -l /mnt/winOS/Dokumente\ und\ Einstellungen/embyte/Desktop/
-r-------- 2 root root 0 2006-09-26 14:02 foo.txt
# mount -o remount /dev/loop0
# v /mnt/winOS/Dokumente\ und\ Einstellungen/embyte/Desktop/
-r-------- 2 root root 11 2006-11-17 14:40 bar.txt
-r-------- 2 root root 0 2006-09-26 14:02 foo.txt
```

Any file-system operation realized from Windows, as file creation, modification, movement, deletion, is not captured from the security shell.

Due to the large time required by the image-scan, this condition of inconsistency is not acceptable, because it easily generates read failures. New viruses, malware, and infected files are not identified, reducing antivirus protection and effectiveness.

Moreover, when a virus is found and the antivirus tries to repair / delete the infected data, possible file-system corruptions (loss of files) could occur, compromising user

documents and the entire Windows system! If a writing occurs on the image while Windows is running, the NTFS cache becomes inconsistence and serious data losses occur.

To prevent unwanted problems, Windows system must be hibernated before an image-scan and it must be restarted as soon as the AV have finished. The image scan must run *offline*. A graphical interface should be prompted to the user, with information regarding the scan process (at least a progress bar and some status lines).

Alternatively, a scan could be accomplished during the Windows start-up (and/or shutdown) to assure the system's cleanness; it is sufficient to modify the boot (halt) sequence, introducing the security check of Antivirus before (after) the VMM start (stop).

In addition, the integration with the backup service is hypothetical. Image scan permits to secure backup copies from virus infections, in order to restore the entire user environment if compromised; it could be seen as a security feature applied to the snapshot technology.

4.1. File repair

Up to here, it has been discussed how to scan the entire user system, transparently and efficiently from the security shell in order to discover the existence of viruses and malware. The image file is simply mapped to a loop-back device and mounted somewhere in the secureOS file-system. To prevent unwanted error conditions, such as synchronization problems and data losses, the image must be scanned offline, prompting the user with an informative GUI and integrating the image scan with backup or boot processes.

However, what happens if a virus is found or a file is marked as infected? The antivirus normally offers the possibility to delete and to repair infected files. Otherwise, when viruses compromise a healthy file, the virus code is added to the original file somewhere not known before. Some bytes can also be overwritten and substituted with malign ones. Often, infected file appears more as garbage than meaningful information, making the file repair process much harder. Virus-files instead are completely malign code and should be immediately removed by antivirus.

Because of the difficulties to repair infected files (clean malign code and rebuild a healthy file), Clam antivirus does not provide this *file repair* feature. Instead, it offers the possibility to remove infected files and viruses (by the command line *-remove* option). This should be carefully used, because it implies the possibility to delete

healthy files when a false positive occurs. The same problem occurs in Network IPS[1], where network connections are dropped down when an exploit signature is matched. This issue is widely studied in both Antivirus and Intrusion Detection System research areas.

Clamscan has been tested with "EICAR Standard Anti-Virus Test File"[2] to verify its ability to identify and delete viruses and malware. EICAR is a legitimate DOS program that produces a sensible result when run (it prints the message "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!"). It is also short and simple. In fact, it consists entirely of printable ASCII characters, so that it can easily be created with a regular text editor. Any anti-virus product that supports the EICAR test file should detect it in any file providing that the file starts with the following 68 characters, and is exactly 68 bytes long:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Clamscan correctly identify and remove EICAR test file:

```
# clamscan --verbose --remove eicar-clone.com
Scanning eicar-clone.com
eicar-clone.com: Eicar-Test-Signature FOUND
eicar-clone.com: Removed
```

[1] A Network IDS is a system that tries to detect malicious activity such as denial of service attacks, port-scans or even attempts to crack into computers by monitoring network traffic and compare it to a well-known database of attack signatures. A Network IPS is an extension that "reacts" to positive results, e.g. dropping the connections.

[2] The Anti-Virus or Anti-Malware test file, http://www.eicar.org/anti_virus_test_file.htm

```
----------- SCAN SUMMARY -----------
Known viruses: 77136
Engine version: 0.88.4
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.00 MB
Time: 1.589 sec (0 m 1 s)
```

Due to the extreme difficulties of repair infected files, many antivirus go through the easier and simpler solution to remove them, immediately. The deletion a file requires the file-system's write support and sufficient account privileges. Therefore, Windows image must be mounted from secureOS antivirus agent with read/write support: when AV *file repair* is enabled, Windows file-system (FAT or NTFS) write support is required!

The contribution of Free Software community

As already said in previous paragraphs, Linux integrates special drivers, which permit fast, reliable, and secure access to Windows file-systems through the *Virtual File-System* architecture. In particular, the current VFAT/FAT32 driver has been rewrite from Gordon Chaffee[1], researcher in the Berkeley Multimedia Research Centre. The

[1] Linux VFAT/FAT32, http://bmrc.berkeley.edu/people/chaffee/fat32.html

driver is completely compatible with the Windows native one (full support) and allows either read and write operations, as file creation, modification, renaming, deletion.

On the other hand, the NTFS driver included in Vanilla Kernel-tree[1], originally established in 1995 by Martin von Loewis and now maintained by the Linux-NTFS-Project[2], features a good read support but partial and experimental write capabilities. It allows reading of files and rewriting existing files, but does not support creation of new files or deletion of existing files.

In 2002, S. Szakacsits joined the project, and worked on many areas, among others, he engineered *ntfsresize* that was the first open source NTFS software capable of heavy NTFS metadata modifications safely. While A. Altaparmakov rewrote driver and user space utilities from scratch, to support the new NTFS versions (Windows 2000 and Windows XP), Szakacsits kept working on the open source code base. On July in 2006 was released NTFS-3G[3], a full read-write NTFS driver that represented major functional and quality improvement to ntfsmount. NTFS-3G provides safe and fast handling of the new Windows file-systems operations, as file/directory creation, deletion, renaming, movement, etc...

---

[1] Vanilla is the official Linux Kernel, http://www.kernel.org/

[2] From project homepage http://www.linux-ntfs.org/: "The goals of the Linux-NTFS project are to develop reliable and full feature access to NTFS by the Linux kernel driver, and by a user space file-system (ntfsmount), and to provide a wide collection of NTFS utilities (ntfsprogs) and a developer's library (libntfs) for other GPLed programs."

[3] The 3rd generation of NTFS driver, http://www.ntfs-3g.org/

At the time of writing this thesis, the 3rd generation of NTFS driver (NTFS-3G) is in BETA status and it has not yet been included in Vanilla Kernel-tree. However, no driver crashes or data loss was experienced during the last month's heavy quality testing, so it is reasonable to use NTFS-3G write support for remove viruses and infected files during antivirus image scan.

The installation of NTFS-3G is quite easy. The driver makes use of *File-system userspace library* (FUSE[1]), which implements a fully functional file-system as a standard program. FUSE is implemented in Linux Kernel as a File-system wrapper to the VFS Layer. When a program makes use of the FUSE framework, each file-system operation (like the stat() method shown in figure) is redirect from VFS to user-space library *libfuse* that operate as "abstract class" for the user file-system.

[1] File-system in Userspace, http://fuse.sourceforge.net/

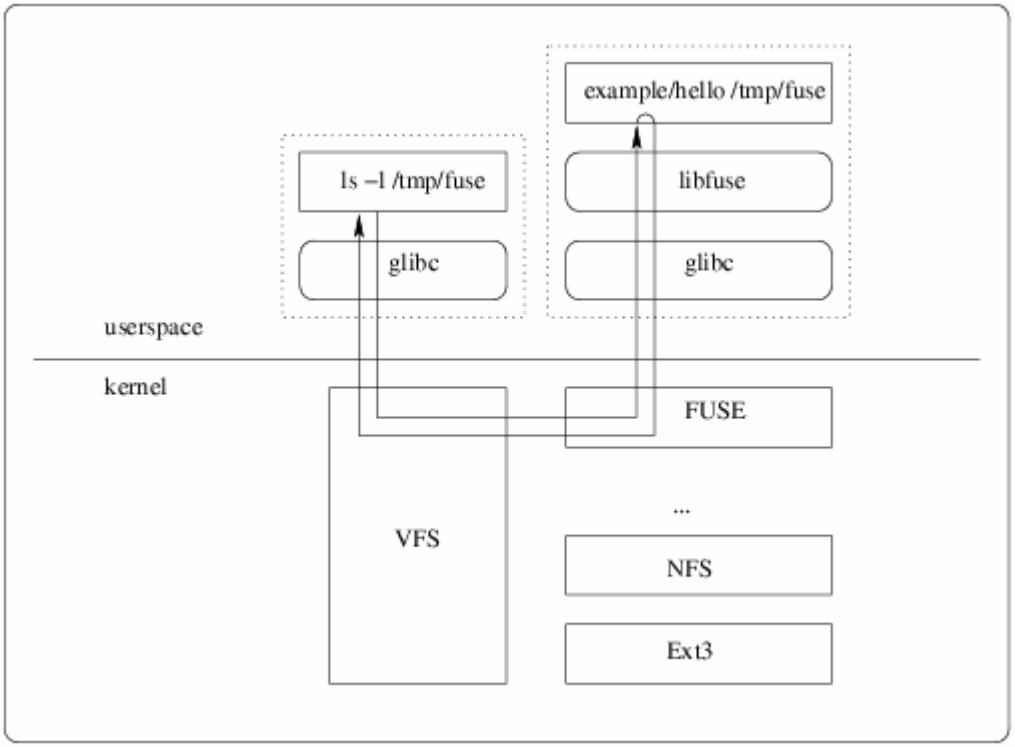


Figure 8 - FUSE Architecture - The stat() file-system call

FUSE was originally developed to support AVFS and it has become a separate project, featuring a simple API, a secure and stable implementation and an easy installation. FUSE driver is included in Linux Kernel from 2.6.15 version, and is compiled (fuse.ko module) enabling *File-system in Userspace support* option. NTFS-3G has been recently merged in Debian Unstable distribution and it is easily installable via this apt-get command:

```
# apt-get install ntfs-3g
```

The 3rd generation of NTFS drivers and the excellent native FAT support permit to mount Windows partitions, reading and writing safely; file repair is used to remove viruses and repair infected files.

This chapter has presented the *image scan* feature, used to virus-scan the entire Windows user system from the lower layer of security shell. Mapping the virtual image as a loopback device and accessing by kernel file-system drivers, Linux permits to browse and to scan the user file-system. The new NTFS driver has safe writing capabilities, enabling the antivirus to remove and to "fix" infected files. To prevent inconsistent states, the image scan should run offline, e.g. at computer boot to assure system cleanness, or together with a backup process to secure backup copies from viruses. In fact, while the system is running, the disk image could become easily inconsistent with the real content, due to the possible modifications carried out by the user environment. On the other hand, antivirus "reparations" to the disk's content could damage the state of the user system.

5. On-access scan

On-access scan is an innovative idea in the antivirus research field, which aims at protecting the user system, while preventing the antivirus to be manipulated. In fact, traditional antivirus products could be deactivated, faked, or avoided by intelligent malign codes and technically skilled users. On-access scan instead is tamper-proof and realizes a continuous virus protection of the virtual OS, real-time scanning the user file-system operations.

In contrast to the conventional *personal antivirus* where the protection is realized within the user environment, here a virtual machine decouples the security functionalities of the Windows system into a separate, invisible, and inaccessible system known as *security shell,* where the antivirus components are moved. The on-access antivirus operates from below the virtual disk, supported by a special *driver* installed into the VMM that deploys information regarding file-system operations. This software component intercepts each file-system access and builds a logical representation of the file's structure, in order to scan its content. In fact, with the reading of a file, from an application or a system service, the file's data blocks are examined by the security shell.

Encapsulating the on-access scan in this hidden, hardened layer, a constant protection against viruses and malware is established, because the security functions cannot be manipulated by unauthorized and evil users. "Untrusted" removable devices as UBS-

stick and standard floppy disks, internet download and temporary/cached files are systematically scanned as they are accessed.

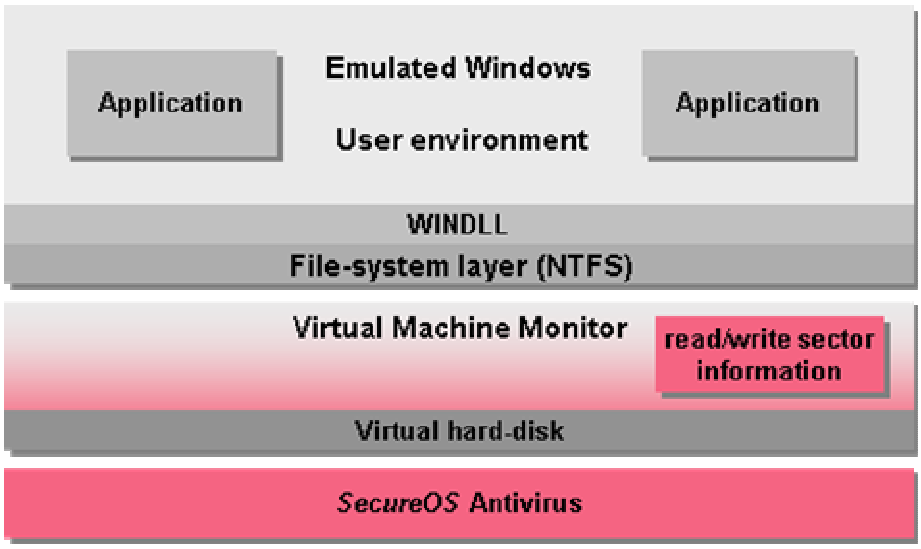


Figure 9 - On-access scan - idea

From the view of the security shell, the user system appears as an image file of the virtual machine, merely as an unstructured list of data blocks. The antivirus component interprets this amount of unorganized data and develops a high-level logical representation in the context of the security shell. Instead, conventional file-systems translate the file representation, known by applications, into low-level information directly applicable to the disk. In fact, a file is a logical description for a set of data, physically stored as spare blocks on the hard disk and grouped together as linked list of blocks (in the following referred to as file structure).

The on-access scan acts oppositely of how a standard file-system works. The Windows file-system initially *resolves* each file operation (reading and writing) into low-level

information about the sector's position and the block's size (in sectors). Then the antivirus driver of the virtual machine intercepts this physical disk access, in terms of a single data block, and *reverse-resolves* it. Since an evil pattern could be positioned anywhere in a file, a virus is perceivable only if the whole content is analyzed, that is, all the file's data blocks should be scanned. For this reason, when an I/O operation occurs, the interpreter reverse-resolves these blocks into the file, which they belong to, and builds the whole list of blocks.

Briefly, the algorithm has been designed according to this model: when a file is read, its physical structure is built and cached as a linked list of blocks. The next time the same file is accessed, if it has not been moved or deleted, the previous cached structure is ready to be used. The antivirus engine retrieves the needed sector information from the cache, addresses the file's data from the virtual disk, and processes it.

When a file is opened in writing mode, its structure is built similarly and compared with the cache's content. With each writing of the Windows system, the cached structures must be modified likewise or entirely replaced to avoid synchronization problems.

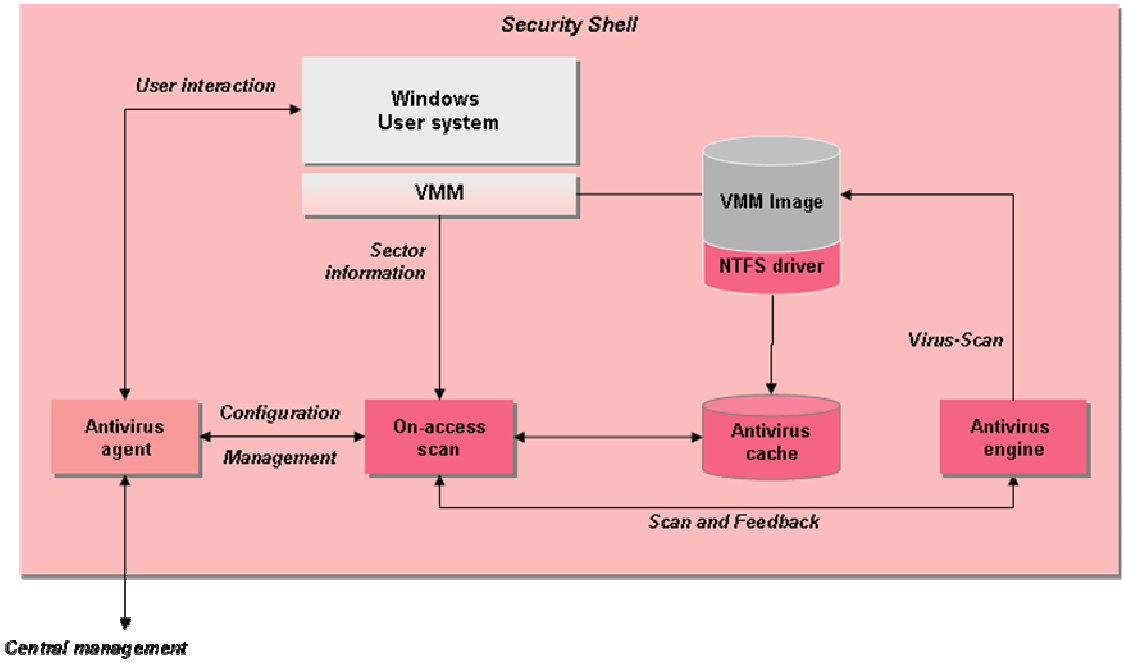


Figure 10 - On-access scan – global architecture

The challenge is synchronization and performance: different caching strategies affect considerably the antivirus' reliability and speed. Since the on-access scan is an "on-line component", it can generate inconsistent states when the image's content is modified. An infected file shall not be deleted, but replaced with a NULL content and a "read-error" shall be returned to the user system. That file can be safely removed only off-line. Moreover, if the files' structures are not correctly updated into cache, infinite loops can easily turn up, slowing down the Windows execution.

The performance is improved designing an optimized antivirus cache for handling file structures. A self-balancing BST is adopted for its capacity to reorganize the cache's information, to be quickly addressable. Standard operations are much faster then conventional trees, when handling non-random data. In fact, in real-world scenarios as file-system caches sectors are accessed often sequentially and repeatedly.

The caching algorithm speeds up of about two times the on-access scan, by marking newly written and freshly modified files with a *dirty flag*. When a new file is created, or at least a single file's block is modified (added or deleted), the file gets marked and considered dirty by the antivirus. When the same file is read afterwards, two alternatives occur: if it has been marked the antivirus scans it, otherwise it is simply skipped. This simple and sharp approach increases the time performance without losing the antivirus efficiency.

The algorithm

Here a simplified version of the *on-access algorithm* is briefly presented through two block diagrams, showing respectively what happens when a read and write is generated by the user system for the virtual disk. In practise, the real implementation uses a slightly more complex algorithm, due to the NTFS architecture, still now partially obfuscated; windows file-system is closed code and sparsely documented. Moreover, the difficulty of debugging strange file-system behaviours entails obvious slowdowns in code implementation. For example, efficient caching solutions are ruined when an unknown write occurs in the *file index table* (or MFT[1]).

In both read and write operations, the antivirus interrupts the user-system execution and "reverse-resolves" the accessed block into its correspondent file and block list.

[1] Reference to next paragraph for a file-system overview, focuses on Windows NTFS file-system.

At reading time, the cache is looked against the current read. If the answer is positive, the file structure is returned and the dirty flag verified. When the file has not been altered in the meantime, the system execution continues; otherwise, the file's content is accessed and scanned.

If a virus is found, the user is informed with an alarm. The file could also be considered damaged and marked for a future deletion. What is done is to replace file's content with NULL content and return a "read error code" to the NTFS file-system. To prevent a situation of inconsistency and a guaranteed crash, that file must be deleted only offline; the modification of the virtual image from the security shell, when the user operating system is running, creates a synchronization problem between the Windows file-system and the disk's content. The well-known Windows's "blue-screen" indicates the kernel panic. After the virus scanning, the file's dirty flag is cleaned.

When the entry is not cached, the block address is verified against the *file index table*. This area (MTF) contains metadata information, as the file name, its access times, permissions, and owner. If the address belongs to MFT, the block is simply read and released to the application. To speed-up file access time, NTFS places file's content directly in the MFT, if less of 1500bytes. The challenge is to understand what particular metadata has been accessed and decide if scan it, as should be done for file's content.

On the other hand, when an ordinary file's content is accessed, its structure is build and cached with the dirty flag unmarked.

The procedure terminates returning the data block or the error code, if a virus has been found. The user-system execution resumes.

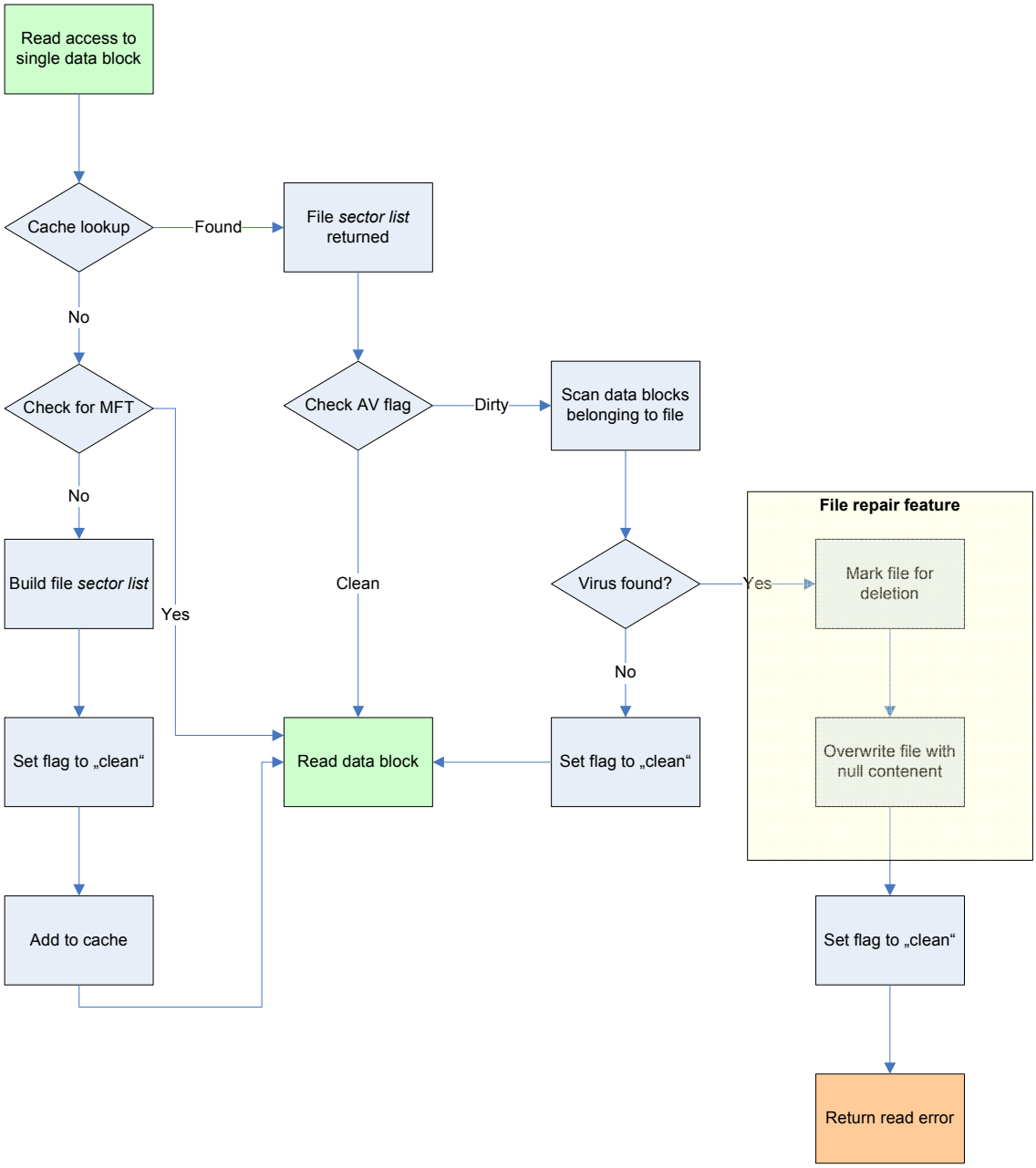


Figure 11 – On-access scan – read algorithm

For writing operations, the algorithm is shorter: if the involved block is already cached, file is considered modified and its dirty flag is marked; otherwise, the file structure is build. For new file creations, the file structure is inserted in the cache as new element; for modifications and deletions, the cache is upgraded with the new file's block list. In both case the file is marked as dirty.

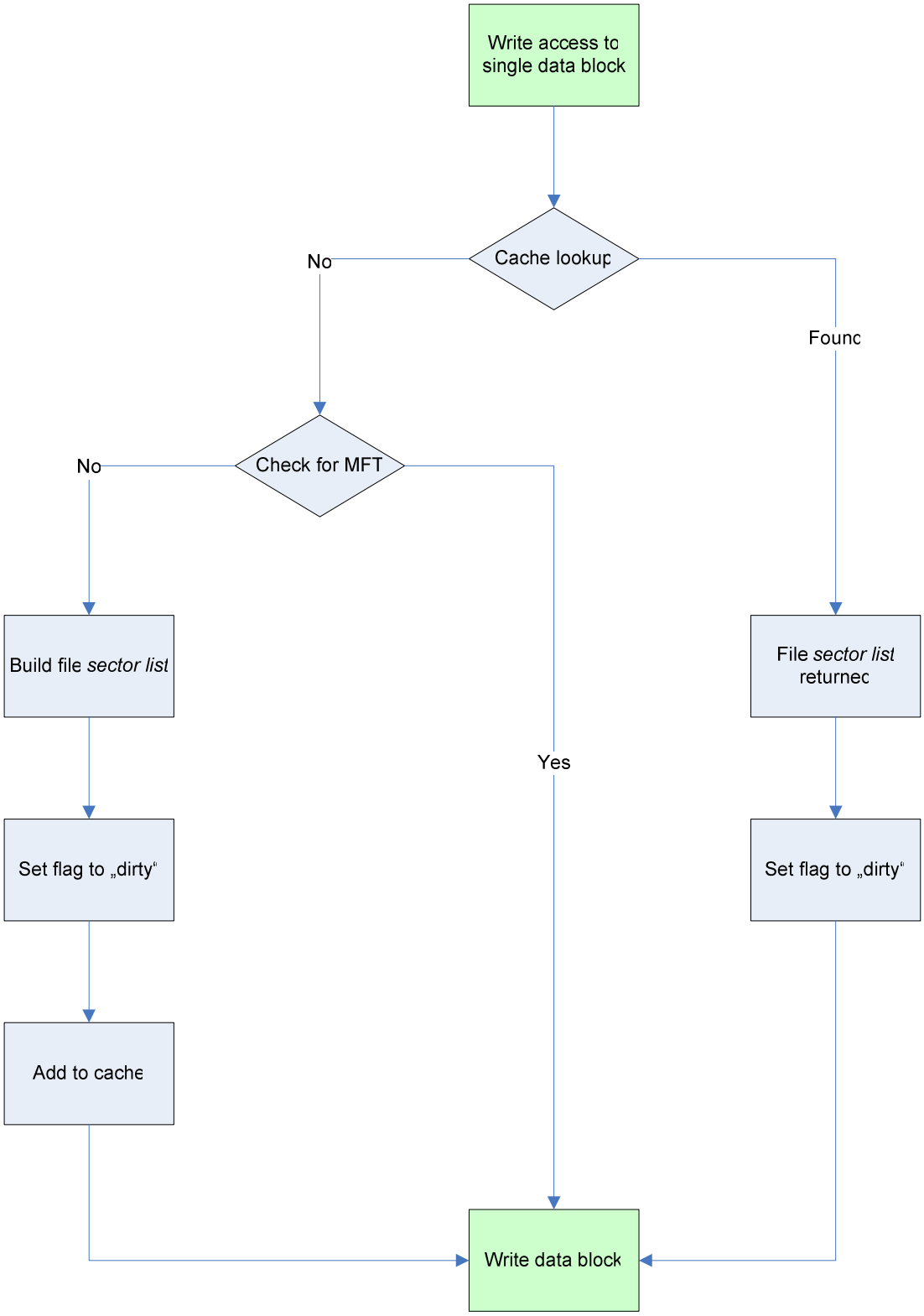


Figure 12 – On-access scan – write algorithm

A challenge in cache design is time waste and memory usage: the cache should be at the same time fast and requiring low memory for the file's structure representation.

The cache is queried each time a block is read and written, looking up if the current file is already present and extracting the file's structure for the antivirus engine. Data blocks have been therefore organized in a self-balancing Binary Search Tree (AVL), using the first sector of each data block as internal index. Self-balancing structures are considered faster compared with normal trees for being able to keep a low tree height. Furthermore, AVL subtype is faster in standard operations when dealing with no uniform statistics, like when data sets are not random, as sequential or repetitive ones.

To avoid useless memory consumption, the caching algorithm should be optimized. When a file gets modified, enlarged with new content, the antivirus driver intercepts the novel blocks and inserts them in the correct position inside cached file structure. It does not rebuild a new file's block list, adding it as double copy to the cache. In similar manner, when a portion of file is deleted, the cache is upgraded with the shorted file structure, or is completely erased when the file is not present anymore in file-system.

5.1. File organization

The smallest unit of space on a disk that any software can access is the *sector*, which normally contains 512 bytes. It is possible to have an allocation system for the disk where each file is assigned as many individual sectors as it needs (e.g., a 1MB file would require approximately 2048 individual sectors to store its data). However, for several performance reasons, in NTFS and other most file-systems, individual sectors are not used. It can get cumbersome to manage the disk when files are broken into 512-

byte individual pieces (e.g., a 20GB disk volume would contain over 40 million sectors). To keep track of these many pieces of information is resource consuming and disk's fragmentation[1] is much more of a problem.

Instead, the usual solution is to group sectors into larger blocks that are called *blocks,* or *clusters*. The block size is determined primarily by the size of the disk volume: larger volumes use larger cluster sizes. This dimension has an important impact on the system performance and the disk utilization: larger cluster sizes result in more wasted space because files are less likely to fill up an integral number of clusters.

Each file is stored as a linked list of blocks (*file sector chain*) and its data content can be located anywhere on the disk. The file-system's main task is to keep track of which blocks are assigned to each file, providing its entire content to the operating system (and hence to any software applications). The file's information is recorded and managed through a sort of index[2] that is usually located at the begin of the partition. Every cluster is chained to the next one using a number and it is not necessary to store the whole file

---

[1] Hard disk fragmentation is the degree to which each file is spread around the disk. In the ideal case, every file would in fact be contiguous: each cluster it uses would be located one after the other on the disk. File-system starts out with all or most of its file contiguous, and becomes more and more fragmented as a result of the creation and deletion of files over a period of time. Utilities Have been developed that can optimize the disk by rearranging the files so that they are contiguous (this process is called *defragmentation).*

[2] In most file-systems, a file allocation table is used to keep track of which clusters are assigned to each file. The operating system determines where a file's data is located by using the directory entry for the file and the file allocation table entries.

in a single continuous block. In fact, the file's blocks can be located anywhere on the disk, and can even be moved after the file creation. The operating system automatically "follows" these *file sector chains* so that to the user each file appears to be in one continuous chunk of disk space.

In the NTFS file-system, every structure is virtually a file, including the structures used to maintain the volume's content and the partition itself. This control information is stored in a special *metadata file* and is initialized when the partition is firstly created. It includes items such as the lists of files on the partition, the volume information, and the cluster allocations. The *Master File Table (MFT)* is actually one of these metadata files, but in some cases, it also contains descriptions of the other metadata files. The MFT contains a record describing every file and directory in the NTFS volume. If a file is small enough, its actual content may be stored in the MFT too. Since the metadata files are just "files" to NTFS, they too have records in the MFT. In fact, the first 16 records of the MFT are reserved for metadata files, as it is shown in the next table:

System file	File Name	MFT Record	Purpose of the File
Master file table	$Mft	0	It contains one base file record for each file and folder on an NTFS volume. If the allocation information for a file or folder is too large to fit within a single record, other file records are allocated as well.
Master file table mirror	$MftMirr	1	Guarantees access to the MFT in case of a single-sector failure. It is a duplicate image of the first four records of the MFT
Log file	$LogFile	2	Contains information used by NTFS for faster recoverability. The log file is used by Windows Server 2003 to restore metadata consistency to NTFS after a system failure.

			The size of the log file depends on the size of the volume, but you can increase the size of the log file by using the Chkdsk command
Volume	$Volume	3	It contains information about the volume, such as the volume label and the volume version
Attribute definitions	$AttrDef	4	Lists attribute names, numbers, and descriptions
Root file name index	.	5	The “root” folder
Cluster bitmap	$Bitmap	6	It represents the volume by showing free and unused clusters
Boot sector	$Boot	7	It includes the BPB used to mount the volume and additional bootstrap loader code used if the volume is bootable
Bad cluster file	$BadClus	8	It contains bad clusters for a volume.
Security file	$Secure	9	It contains unique security descriptors for all files within a volume
Upcase table	$Upcase	10	Converts lowercase characters to matching Unicode uppercase characters
NTFS extension file	$Extend	11	Used for various optional extensions such as quotas, reparse point data, and object identifiers
		12–15	They are reserved for future use

Table 1 - Metadata files ftored in the MFT

The elegance of the metadata system is that by storing internal information in files, it is possible to expand on the capabilities of the file-system. In addition, these files do not need to be stored in a specific location on the disk, so if a specific area becomes damaged, they can be moved.

The following example helps to understand how a file is organized on disk. The DOS-interpreter *cmd.exe* is distributed on the disk volume in four pieces (blocks for simplicity) in ranges:

- from sector 200 to 220 (20 sectors);

- from sector 500 to 800 (300 sectors);
- from sector 1000 to 1100 (100 sectors);
- from sector 11000 to 11650 (650 sectors)

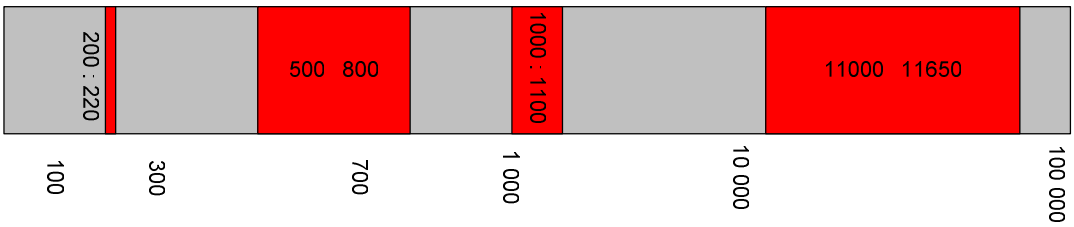


Figure 13 – File organization on disk

The file is therefore 1170-sectors large and occupies 858KB (assuming a sector size of 512 bytes).

A file is represented within the antivirus cache in the same way as it is physical organized. The *file sector chain* is implemented as a linked-list and contains three pieces of information: the file's name, the file's number of blocks and a special flag by the antivirus engine to scan the appropriate files. Each list's node holds the first and the last sector of each block.

Using the example, the *cmd.exe* interpreter is represented as a *sector_list*, in which the *filename* is cmd.exe and the *dirty_flag* is 0 (assuming that the file has been just be scanned). The list contains four *sector_list_entry* nodes that are initialized with the addresses of the file's blocks. In particular, because of the fragmentation problem, the *cmd.exe* could be physically written as an unordered sequence.

So, when a new list is allocated for the DOS-interpreter, its elements are attached to the list in the same order in which the file's blocks are physically stored on the volume. In the same way, when a file becomes larger (smaller), the new blocks are inserted (deleted) into the list in the correct order. It is very important to manage the order correctly in order to prevent that the file's content becomes garbage (the two sentences "these files contain the virus *don't*" and "these file *don't* contain the virus" differs only for the position of the "*don't"* word, but have two opposite meanings!).

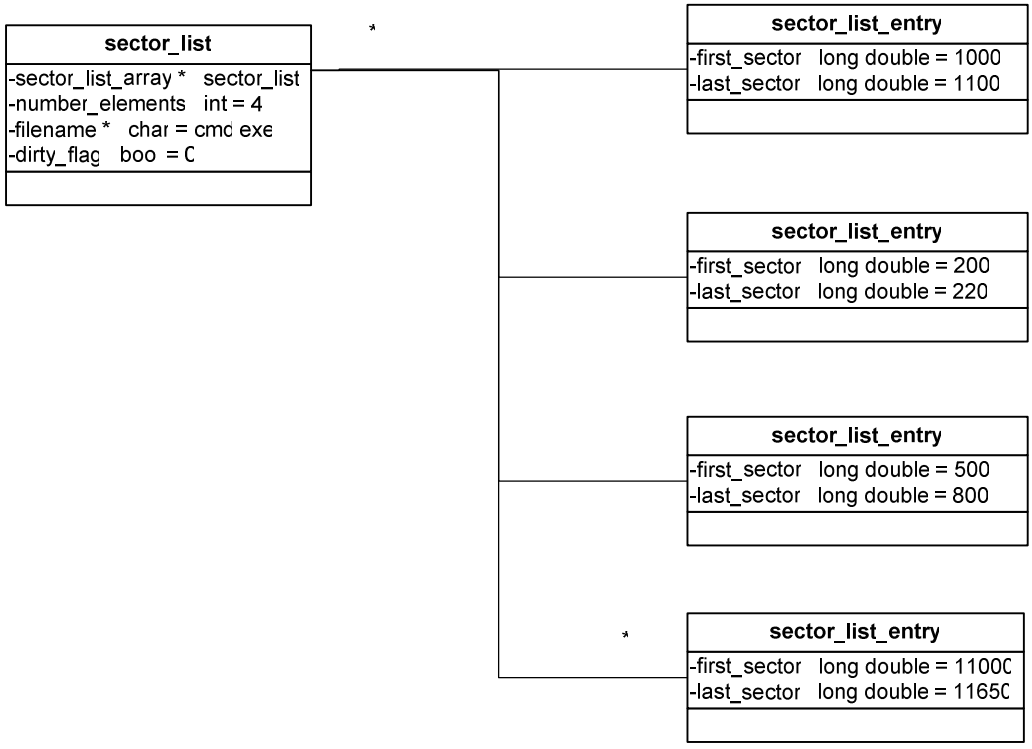


Figure 14 - On-access scan - file architecture[1]

[1] The file's blocks has been written with this order: 1) block 1000:1100; 2) block 200:220; 3) block 500:800 and 4) block 11000:11650

5.2. Cache design

Two efficient and common data structures are used to represent data in memory: *hash tables* and *self-balancing binary search trees*.

Hash tables, sometimes also known as associative arrays, or scatter tables, are data structures that offer fast lookup, insertion, and deletion of (key, value) pairs. In a well-designed implementation of hash tables, all of these operations have a time complexity of *O(1)*, rather than the *O(n)* that linked lists, association lists, and vectors would require for some of them.

Hashing was proposed and first studied and implemented on computers in the early 1950s, and is based on a simple idea: transform the key to a number (the *hash* process), and then use that number to index a table. In order to keep the table size manageable, reduce the computed number modulo the table size.

Notice, however, that hash functions do not preserve key order: even if keys are entered in, say, alphabetical order, they will in general be stored randomly in the table. Thus, the extra work of a sorting step may be needed, if key order is to be restored when the table is traversed. Moreover, the hash function itself is time-consuming.

When the keys of more than one item map to the same position, we have a called a collision. Collision handling is the price to be paid for the bonus of *O(1)* complexity, and if the mechanism for doing so is poorly designed, performance may well deteriorate to *O(n)*, or worse. There are many collision resolution schemes, but they may be divided into open addressing, chaining (shown in figure), and keeping one special overflow area. Perfect hashing avoids collisions, but may be time-consuming to create.

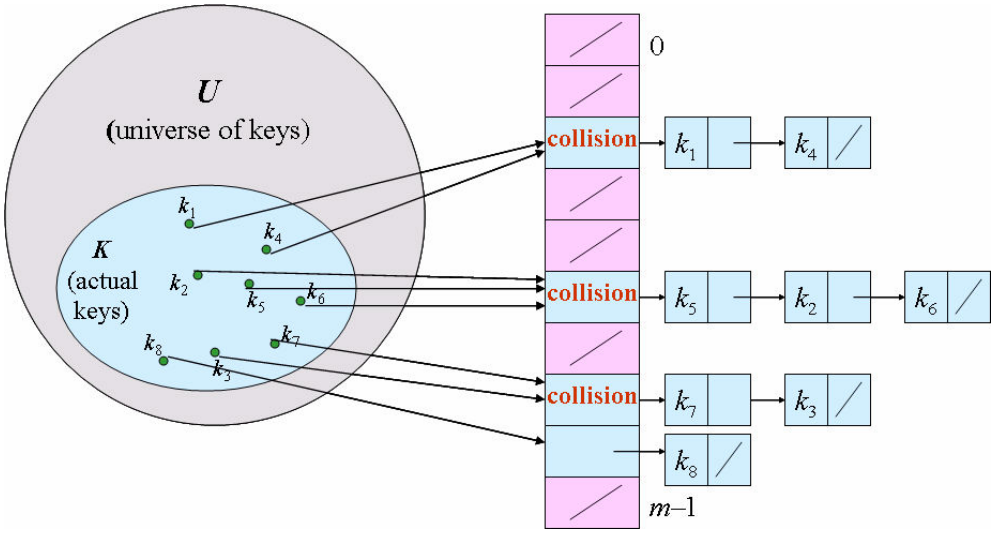


Figure 15 - Hash table showing collision problem

A *Tree* is defined as a not empty finite set of labelled nodes such that there is only one node called the root of the tree, and the remaining nodes are partitioned into sub-trees. If the tree is either empty or each of its nodes has not more than two sub-trees, it is called a *Binary Tree*. Hence, each node in a binary tree has either no children, one left child, one right child, or a left child and a right child, each child being the root of a binary tree called a sub-tree.

Every node (object) in a binary tree contains two pieces of information. The first one is proper to the structure of the tree, that is, it contains a key field (the part of information used to order the elements), a parent field, a left child field, and a right child field. The second part is the object data itself that can reside inside the tree or outside, referenced by the node. The root node of the tree has its parent field set to null. Whenever a node does not have a right child or a left child, then the corresponding field is set to null.

Trees support a set of basic operations, common to any family of trees:

- searching for an item (lookup);
- adding a new item at a certain position on the tree (insertion);
- enumerating all the items (enumeration);
- deleting an item (deletion);
- removing a whole section of a tree (pruning);
- adding a whole section to a tree (grafting);
- finding the root for any node.

A *Binary Search Tree (BST)* is a binary tree with more constraints. If x is a node with key value key[x] and it is not the root of the tree, then the node can have a left child (denoted by left[x]), a right child (right[x]) and a parent (p[x]). A BSD is a tree with the following *Binary Search Tree property:*

1. All nodes y in left sub-tree of x have key[y] < key[x]
2. All nodes y in right sub-tree of x have key[y] > key[x]

The major advantage of binary search trees is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

The basic operations on a binary search tree of *n* nodes (lookup, insertion, removal and sort) take time proportional to the height of the tree, ranging from *n* in the worse case, when the unbalanced tree resembles a linked list, to *log(n)* in the average case, when a complete balanced BST occurs.

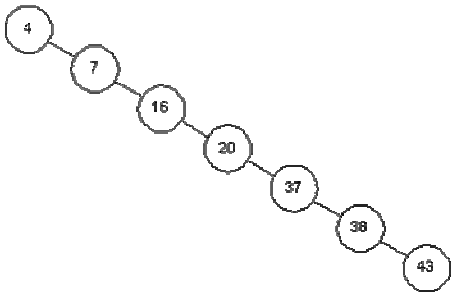


Figure 16 - Tree degrading to linked list

Self-Balancing Binary Search Trees

For this reason, *self-balancing BST* have been studied and proposed as a solution to keep tree *height*, or the number of levels of nodes beneath the root, as small as possible at all times, automatically. The height must always be at least the ceiling of log n, since there are at most 2k nodes on the k^{th} level; a complete or full binary tree has exactly this many levels. Self-balancing BS Trees are one of the most efficient ways of implementing associative arrays, sets, and other data structures.

Ordinary binary search trees have the primary disadvantage that they can attain very large heights in rather ordinary situations, such as when the keys are inserted in order. Self-balancing binary trees solve this problem by performing transformations on the tree (such as tree rotations) at key times, in order to reduce the height. Although a certain overhead is involved, it is justified in long runs where the time of later operations drastically decreases.

The following figures show the same tree, firstly un-balanced and later after a balancing algorithm has been applied; although the tree is the same, operations taken on the balanced one require less time.

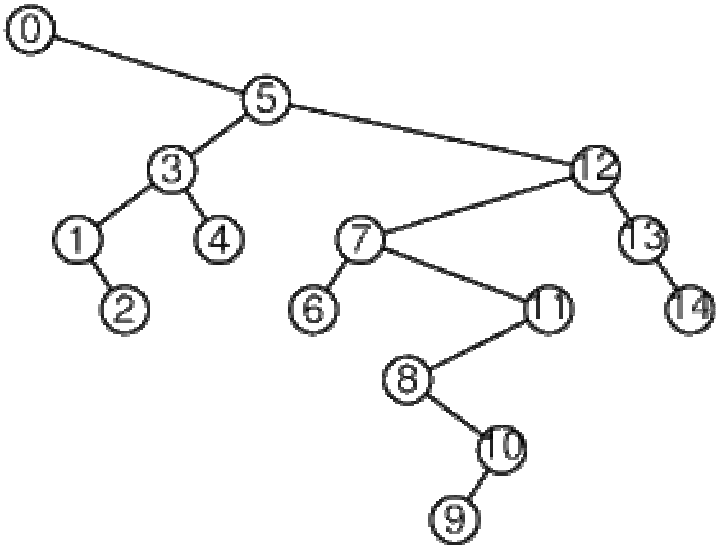


Figure 17 - Example of un-balanced BST

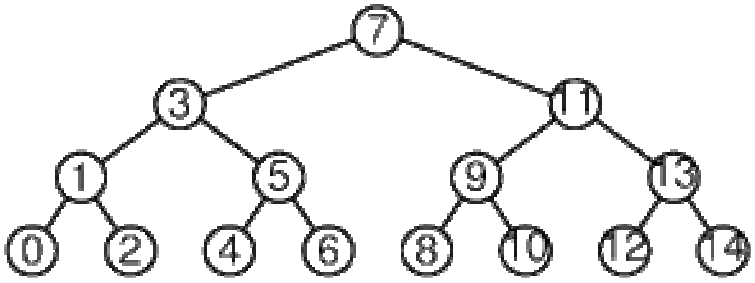


Figure 18 – The same tree after balancing

Reducing the tree height, using the balancing techniques implemented in the self-balancing trees, the computational complexity gets reduced of a n/log(n) factor in its basic operations.

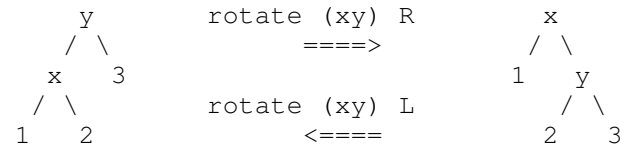
BST operation	Completely unbalanced (linked list)	Balanced (Self-balancing BST)
Height	n	log b
Lookup	O(n)	O(log n)
Insertion	O(n)	O(log n)
Removal	O(n)	O(log n)
Sort	O(n)	O(n log n)
Enumeration (In-order iteration)	O(n)	O(n)

Table 2 - Computational complexity of BST operations

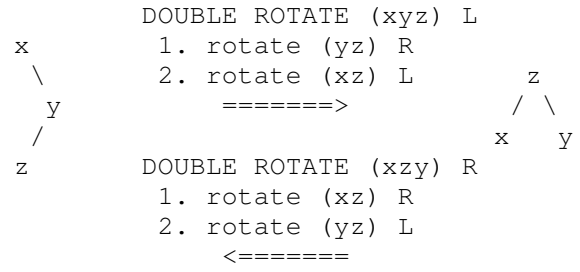
Hundreds of self-balancing BST have been proposed to keep automatically tree height near log(n). Some well-known implementations are **AVL**, Red-black and **Splay trees**.

In *AVL-tree,* each node has a balanced factor set to the difference of height of its right and left sub-trees. A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced (marked as “illegal”) and requires rebalancing the tree, by sequenced rotations. Let X be the deepest node whose balance factor has become "illegal". In this scenario X's balance factor has become too large, +2 (the scenario that X's balance factor has become too small, -2, can be handled in a symmetric manner.) This illegal imbalance happened because X previously had a balance factor of +1, and the insertion caused X's right sub-tree (headed by Y) to increase in height. This height increase implies that Y had a balance factor of 0,

because otherwise either Y would not gain height or Y would become illegally imbalanced. This scenario deploys in 2 cases: the insertion is made into Y's right sub-tree or into Y's left sub-tree. Insertion into Y's right sub-tree requires a single rotation to rebalance the tree:



On the other hand, when the insertion is done into Y's left sub-tree, a double rotation is done:



Sleator and Tarjan suggested the *Splay-tree* as an efficient alternative implementation of self-balancing BST that takes advantage of locality in the incoming lookup requests to increase data lookup. Locality in this context is a tendency to look for the same element multiple times. A stream of requests exhibits no locality if every element is equally likely to be accessed at each point (uniform access). For many applications, there is locality, and elements tend to be accessed repeatedly. This is truer for caches, where groups of data are accessed more frequently than the rest.

Whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.

A splay tree has no explicit balance condition and a special operation ("splay") is done after each search or insertion operation. Splaying at node x causes node x to become the root of the binary search tree through a specific series of rotations as follows. Three cases occur:

1. X has no grandparent (zig)
    * If X is left child of root Y, then rotate (XY)R.
    * Else if X is right child of root Y, then rotate (YX)L.
2. X is LL or RR grandchild (zig-zig)
    * If X is left child of Y, and Y is left child of Z,
      then rotate at grandfather (YZ)R and then rotate at father (XY)R.
    * Else if X is right child of Y, and Y is right child of Z,
      then rotate at grandfather (YZ)L and then rotate at father (XY)L.

   If X has not become the root, then continue splaying at X.
3. X is LR or RL grandchild (zig-zag)
    * If X is right child of Y, and Y is left child of Z,
      then rotate at father (YX)L and then rotate at grandfather (XZ)R.
    * Else if X is left child of Y, and Y is right child of Z,
      then rotate at father (YX)R and then rotate at grandfather (XZ)L.

If X has not become the root, then continue splaying at X.

Performance analysis

Choosing the right kind of tree affect the performance significantly. For this reason, Ben Pfaff has done an empirical study of the relationship between the algorithms used to manage BST-based data structures and performance characteristics in real systems. He compared four variants of the BST's data structure: unbalanced-, AVL-, Red-black-, and Splay- trees. For each BST data structure variant, he compared five different node representations: plain, with parent pointers, threaded, right-threaded, and with an in-order linked list.

At the end, he ran three experiments in real-world scenarios and one in random data-set scenario on 20 BST variants. The next table shows results of running a Virtual Memory Area activity (*mmap()* and *munmap()* calls) during the execution of three programs and one simulated.

The results of the "Squid" test are the most relevance for the design of the antivirus cache, because of the many similarities between the caching of web pages and disk sectors.

Test	*Node represent.*	*Time (seconds)*				*Performance improvement against BST*		
		BST	*AVL*	*RB*	*SPLAY*	***AVL***	*RB*	***SPLAY***
Mozilla	Plain	4.49	4.81	5.32	2.71	-7.12%	-18.48%	39.64%
	Parents	15.67	3.65	3.78	2.63	76.70%	75.87%	83.21%
	Threads	16.77	3.93	3.95	2.67	76.56%	76.44%	84.07%
	R. threads	16.91	4.07	4.20	2.68	75.93%	75.16%	84.15%
	Linked list	16.31	3.64	4.35	2.74	77.68%	73.32%	83.20%
Vmware	Plain	208.00	8.72	10.59	3.77	95.80%	94.90%	98.18%
	Parents	447.40	6.31	7.32	3.62	98.59%	98.36%	99.19%
	Threads	445.80	6.91	8.51	3.64	98.45%	98.09%	99.18%
	R. threads	446.40	6.88	8.59	3.51	98.45%	98.07%	99.21%
	Linked list	472.00	7.35	8.60	3.45	98.44%	98.17%	99.26%
Squid	Plain	7.34	4.41	4.67	2.84	39.91%	36.37%	61.30%
	Parents	12.52	3.69	3.80	2.64	70.52%	69.64%	78.91%
	Threads	13.44	3.92	4.18	2.70	70.83%	68.89%	79.91%
	R. threads	14.46	4.17	4.27	2.86	71.16%	70.47%	80.22%
	Linked list	13.13	4.02	4.19	2.65	69.38%	68.08%	79.81%
Random	Plain	2.83	2.81	2.86	3.43	0.70%	-1.06%	-21.20%
	Parents	1.63	1.67	1.64	1.94	-2.45%	-0.61%	-19.01%
	Threads	1.64	1.74	1.68	2.02	-6.09%	-2.43%	-23.17%
	R. threads	1.92	1.96	1.93	2.22	-2.08%	-0.52%	-15.62%
	Linked list	1.46	1.54	1.51	1.74	-5.47%	-3.42%	-19.17%

Table 3 - BST evaluation

For the *random data set*, unbalanced BSTs are best because they do not insert extra work due to tree rebalancing[1]. There is no need for complex rebalancing algorithms, because ordinary BSTs produce acceptably balanced trees with high likelihood. Thus, for random insertions, data structures with the least extra overhead (above ordinary BSTs) yield the best performance.

[1] For the random data set, ordinary BSTs make no rotations, red-black trees make 209, AVL trees make 267, and splay trees make 2,678

The red-black balancing rule is more permissive than the AVL balancing rule, resulting in less superfluous rebalancing, so red-black trees perform better for random insertions. Splay trees consistently performed worst, and required the most comparisons, within each node representation category in the random test. This reflects the work required to splay each node accessed to the root of the tree, in the expectation that it would soon be accessed again. This effort is wasted for the random set because of its lack of locality.

On the other hand, when data are not uniformly accessed, ordinary BSTs become much slower than self-balancing trees. This condition of not randomness is usually respected for real-world scenarios, where data are inserted and accessed sequentially and repeatedly. If data are sequentially inserted into the cache, unbalanced BSTs generate a tree shaped as linked list, degrading completely tree performance. As previously discussed, in this situation node operations on ordinary BSTs reach a complexity O(n), while self-balancing BSTs as AVL and splay-trees, to the capacity of reorganizing the tree, reduce time to O(log(n)).

Looking at the *Squid* scenario**,** both AVL and splay trees beat unbalanced BSTs by a wide margin, ranging from 40% of AVL with plain node-representation to 80% of splay tree when right threads node representation is used. In particular, splay trees are better, due to splay tree's ability to keep frequently used node near the top of the tree.

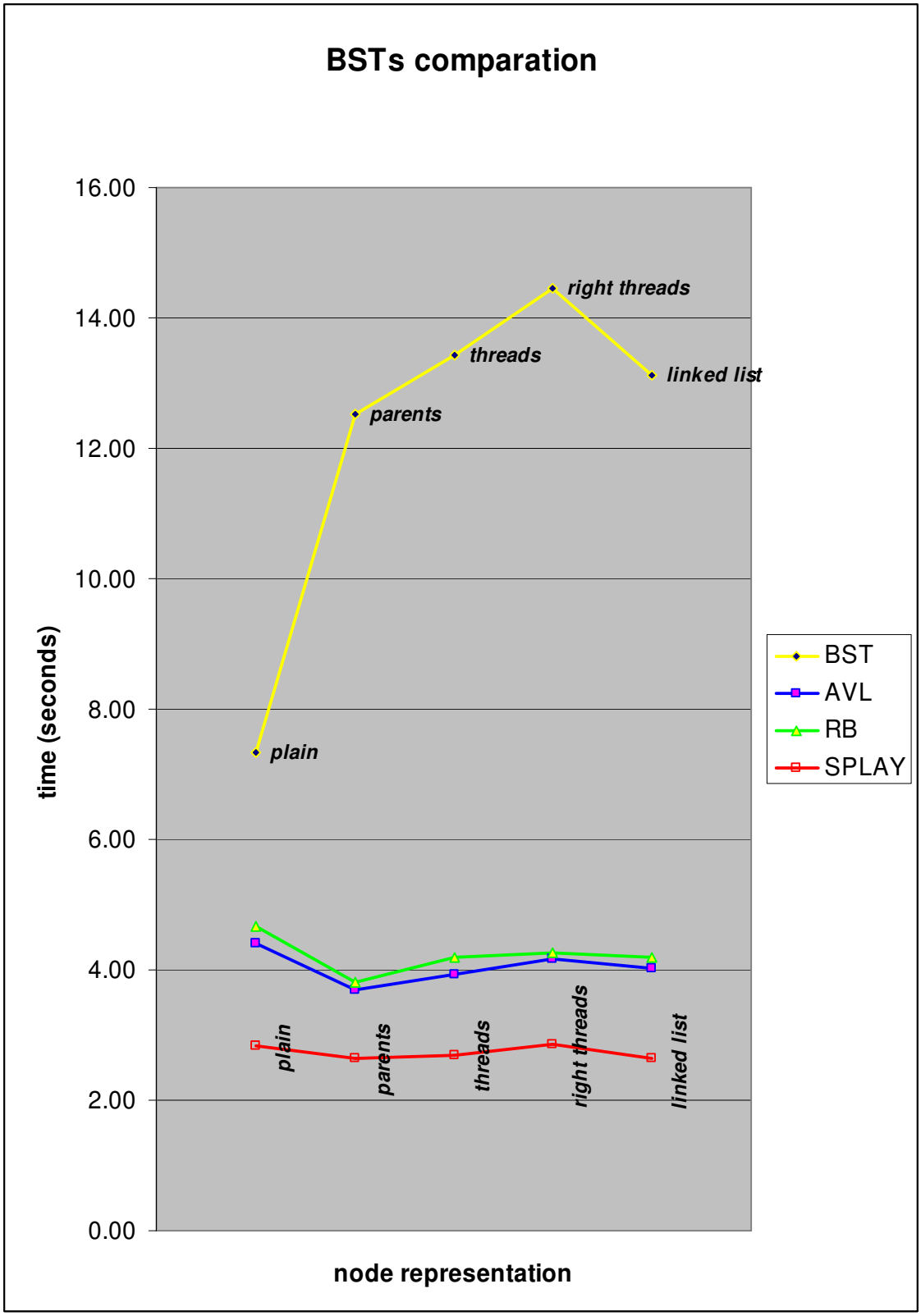


Figure 19 – BST evaluation - squid scenario

Conclusions

In a real-world scenario as in a file-system cache, sectors are accessed mostly sequentially and often repeatedly. Unbalanced BSTs have been demonstrated to be too slowly for practical applications, and more sophisticated self-balancing tree has been proposed in order to reduce computation complexity and improve the global performances (40-80%). Both AVL and splay trees represents valid solutions: first ones for ability to rebalance tree only when necessary, reducing "extra work" times when operating with random data set; second ones for offering frequently used nodes fastest.

The antivirus cache is designed as AVL-tree, where the leaf reference data-blocks of the virtual disk. Each addressed block is represented as an object linked by the leaf and a cached file consists of a set of leaf. As already discussed one block is a sequence of continuous sectors, delimited in a range by the first and the last sector.

The entire tree is indexed using the first sector of each cluster. When a new data block has to be added to the cache, a new leaf is created to reference the cluster and the correct position is looked up using the block's first sector. Later on, possible tree reorganization could occur.

In details, each leaf references one *sector_cache_data* element that belongs to a *sector_list* list, used to index blocks belonging to the same file. In this way, when a file has to be accessed, its blocks are addressed using the *sector_list* list and not querying the tree for each single cluster. This mechanism improves at the same time cache performance and efficiency.

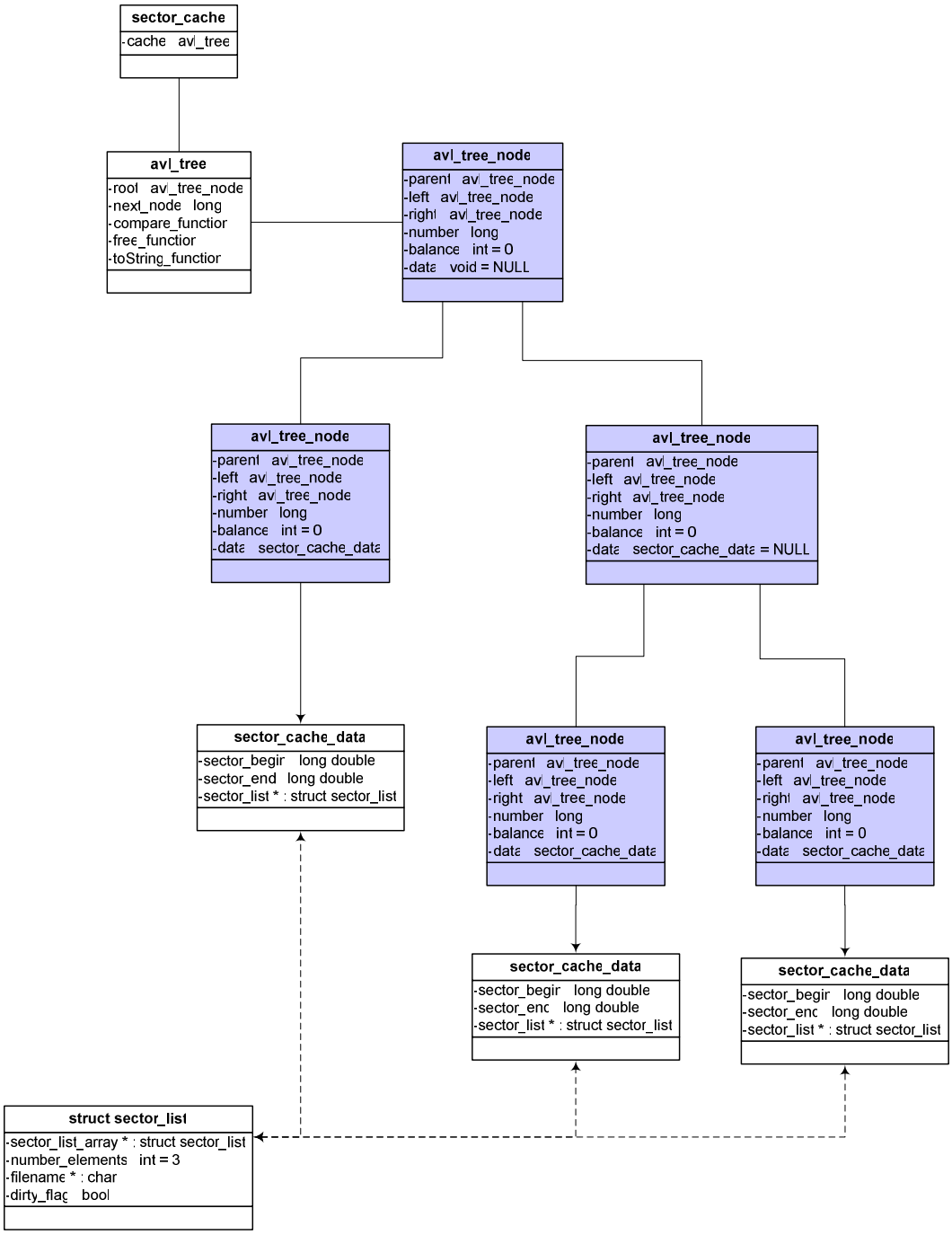


Figure 20 - On-access scan – cache architecture

5.3. Antivirus integration

The *libclamAV* open source library *(version 0.88)* has been adapted and reused in the test implementation, integrated as antivirus engine for providing virus scan facilities. The library has been slightly modified, to be able to operate with the antivirus cache; antivirus engines are suitable for file scanning, while *secureOS* antivirus should work at lower level, directly on hard disk sector and block data structures. The goal has been to create a generic antivirus interface, making the antivirus layer extremely portable and reusable in different implementations; in fact just few "lines of code" of the library have been modified, while much of the code has been included into the *secureOS* service. Technically, the routine for virus-scanning a file has been hacked to handle data coming from a list of sectors instead of a file descriptor (once opened, a file is read iteratively as a sequential set of data).

Here is the hack of the `matcher.c` source file[1]:

```
< int cli_scandesc (int desc,
<                   const char **virname,
<                   unsigned long int *scanned,
<                   const struct cl_node *root,
<                   short otfrec, unsigned short ftype)
[…]
<   while ((bytes = cli_readn(desc, buff, SCANBUFF)) > 0)
```

```
> int _td_scanlist (struct sector_list *pSectorList, int BDRVfd,
>                   const char **virname,
>                   unsigned long int *scanned,
>                   const struct cl_node *root,
>                   short otfrec, unsigned short ftype)
>
> int desc = -1; // private descriptor set to dummy value of 1
[…]
>   while ((bytes = _td_readn(pSectorList, BDRVfd,
                              buff, SCANBUFF)) > 0)
```

[1] Red/minor (<) is original code and blue/major (>) is new code

The `_td_readn() function`[1] is responsible for parsing the *file sector chain* (sectors belonging to the file to scan) in sequential fixed-size data sets, reading at each iteration the file's content from the virtual disk. This routine works similarly to the `libc's read()`, where an internal mark is moved forward at each function call, labelling the next "element" to be read.

The algorithm is particularly interesting and here is presented as *pseudo-code* [2]:

```
while (still to read > 0 AND number of sectors != 0)
  {
   current byte = element.first_sector * 512;
   number of sector to read = (element.last_sector -
                               element.first_sector) * 512;

// seek to the current position
   lseek (device, current byte + offset, SEEK_SET);

   if (number of sectors - offset > still to read)
// the current element is bigger of the read one
   {
     ret = read (device, buffer + already read, still to read);

// update the indexes
        still to read -= ret;
        already read += ret;
        offset + = ret;
  }

// the left sectors of the i-element are not enough
  else
  {

     ret = read (device, buff + already read,
                 number of sectors - offset);
```

[1] Please see Appendix A for the entire_`td_readn()` source code.

[2] Appendix A contains the C implementation.

```
// update the indexes
        still to read -= ret;
        already read += ret;

// set the status pointer to next element
       i++;

// go to the next element
       pStatus.element++;
       pStatus.offset = 0;
  }
}
```

Moreover, this function calculates each file size and discards those bigger than 10MByte (20.000 sectors). This trade-off between performance and security has been temporarily used in the research environment.

When the antivirus service starts, the `td_init_avscan() routine`[1] load the Clamavis engine the virus pattern database.

The *td_avscan() function* is the real interface to the antivirus engine. In agreement with the on-access scan algorithm, when reading on a dirty file has been intercepted, the file's whole content is virus-scanned; the antivirus interface is queried with the *file sector chain* and the scan results, together with a possible virus name, are returned:

```
int td_avscan (struct sector_list *pSectorList, int BDRVfd,
               char **virname)
```

As it can be seen, this antivirus interface is very generic, to be high reusable and portable on different antivirus engines.

[1] Please see Appendix A for `td_init_avscan()` source code.

5.4. Testing

Testing represents the last phase of *on-access scan* research, proving that this antivirus agent, implemented into the test prototype, works. Future commercial applications, as all-in-one security solutions, can be developed. Testing shows that the on-access component correctly intercepts the reading and writing operations of virtual disk's sectors, builds *file sector chains* from single sector accesses and keeps its internal cache synchronized with Windows system. The on-access algorithm is right enforced into the agent, which keeps a footprint on which files has been lately created and modified. This permits to virus-scan just critical file-system resources, realizing a performance improved solution in terms of time overhead of the antivirus service.

Here it is given the evidence of the obtained results.

Low-level I/O activities are wrapped by the modified version of the VMM for antivirus agent. Each Windows file-system operation to the virtual disk image is captured as block reading or writing. Blocks are usually determined by a size of 2, 8, and 16 sectors:

```
hda: read()  ; [ 3191953 – 3191968 ] ; (16 sectors)
hda: read()  ; [ 3192072 – 3192087 ] ; (16 sectors)
hda: write() ; [ 2108207 – 2108214 ] ; (8 sectors)
hda: read()  ; [ 3192088 – 3192089 ] ; (2 sectors)
Hda: read()  ; [ 3273873 – 3273880 ] ; (8 sectors)
hda: write() ; [ 2108367 – 2108374 ] ; (8 sectors)
hda: write() ; [ 2112623 – 2112638 ] ; (16 sectors)
```

Windows NTFS (version 3.1) has been counted as “de facto standard” for the user system’s file-system and the development process has been supported through NTFS Linux driver, distributed in the Linux-NTFS toolkit. Therefore, current prototype tested here assumes an NTFS file-system and does not work with FAT.

When a sector is written, the agent identifies the file owning that sector. If a cache miss occurs, the antivirus builds a new *file sector chain* to add into the cache; otherwise, it upgrades the cache’s content with the current status of the disk. In the same way, at read access, cache is scanned for the list of blocks for that operation.

In both use cases, the agent has given evidence of being able, given an individual sector, to build the whole list of blocks forming that file and to lookup for the file name.

For example, the *catsrvut.dll* library has been “resolved” into a list of five blocks, while the *change.log* log seems of bigger size.

```
*** sector_cache_add_sector_list ***
Adding catsrvut.dll, m_sector_list_size = 5
    #0: 1264592 to 1264663
    #1: 774992 to 775295
    #2: 394224 to 394535
    #3: 710000 to 710327
    #4: 1205040 to 1205255
```

```
*** sector_cache_add_sector_list ***
Adding change.log, m_sector_list_size = 15
    #0: 909872 to 909903
    #1: 1483592 to 1483687
```

```
    #2: 2966360 to 2966391
    #3: 150960 to 150991
    #4: 2752456 to 2752487
    #5: 107448 to 107479
    #6: 723448 to 723479
    #7: 896272 to 896303
    #8: 71336 to 71367
    #9: 91856 to 91887
    #10: 123680 to 123711
    #11: 989600 to 989631
    #12: 985600 to 985631
    #13: 895632 to 895663
    #14: 1474056 to 1474087
```

The caching mechanism works pretty well, in both reading and writing.

When Windows starts, the agent shows a read access of the *volume boot sector*, indexes as *sector 0* in the file-system and named *$Boot* (a dollar is set above metadata files); the access is predictable, since boot sector contains the bootstrapping code used to run Windows. When accessed, $Boot is cached and the *dirty flag* is cleaned; later on, it is simply shipped:

```
hda: read() ; [ 0 - 0 ] (1 sectors)
rM: $Boot - flag set to CLEAN
hda: read() ; [ 0 - 0 ] (1 sectors)
rH: $Boot
```

The same behaviour has been noticed for the *plug&play* driver archive (*driver.cab)*:

```
hda: read() ; [ 63 - 63 ] (1 sectors)
rM: driver.cab - flag set to CLEAN
hda: read() ; [ 63 - 63 ] (1 sectors)
rH: driver.cab
```

When a write occurs, a comparable procedure is run; anyway, now the file is considered as dirty:

```
hda: write() ; [ 65593 - 65593 ] (1 sectors)
wM: change.log
ntfs_reverse_lookup_and_add_to_cache() - dirty flag set on
hda: write() ; [ 65593 - 65593 ] (1 sectors)
wH: change.log
write() - dirty flag set on change.log
```

The antivirus engine seems to work quite well, being activated when a read occurs on a cached file that appears new or modified (dirty flag set).

This first log shows two skipped files, a third one scanned and the last one reported infected:

```
1.  raw_read() - cache hit
file ipxwan.dll has clean flag - scan not required
raw_read() - cache hit
file SiteGeneric.css has clean flag - scan not required
```

```
2.  raw_read() - cache hit
file wdma_rip.PNF has dirty flag
going to scan it
File clean
flag reset to CLEAN
raw_read() - cache hit
file test2.txt has dirty flag
going to scan it
VIRUS FOUND: Eicar-Test-Signature
flag reset to CLEAN
```

This second log is higher-level and reports which files are scanned. It's interesting to note that both *plug&play* driver archive and the swap file are skipped because they are bugger than 10Mbyte. The "Eicar-Test-Signature" is recognized in the *text2.txt* document.

```
* skipping driver.cab (76 MB)
td_avscan(): ntldr - virus not found
td_avscan(): CollectedData_15.xml - virus not found
td_avscan(): GTL_SiteGeneric[1].css - virus not found
* skipping pagefile.sys (3422 MB)
td_avscan(): noise.chs - virus not found
td_avscan(): noise.cht - virus not found
td_avscan(): comexp.chm - virus not found
* td_avscan(): test2.txt - VIRUS FOUND: Eicar-Test-Signature
```

Finally a hued screenshot, displaying Windows XP “on the top” of the *security shell*, proves that the on-access antivirus works! As soon as the EICAR-test file has been opened within user environment, the agent console reports the VIRUS FOUND error, along with virus filename. The two real-time logs print each virus-scanned file and the caching activity.

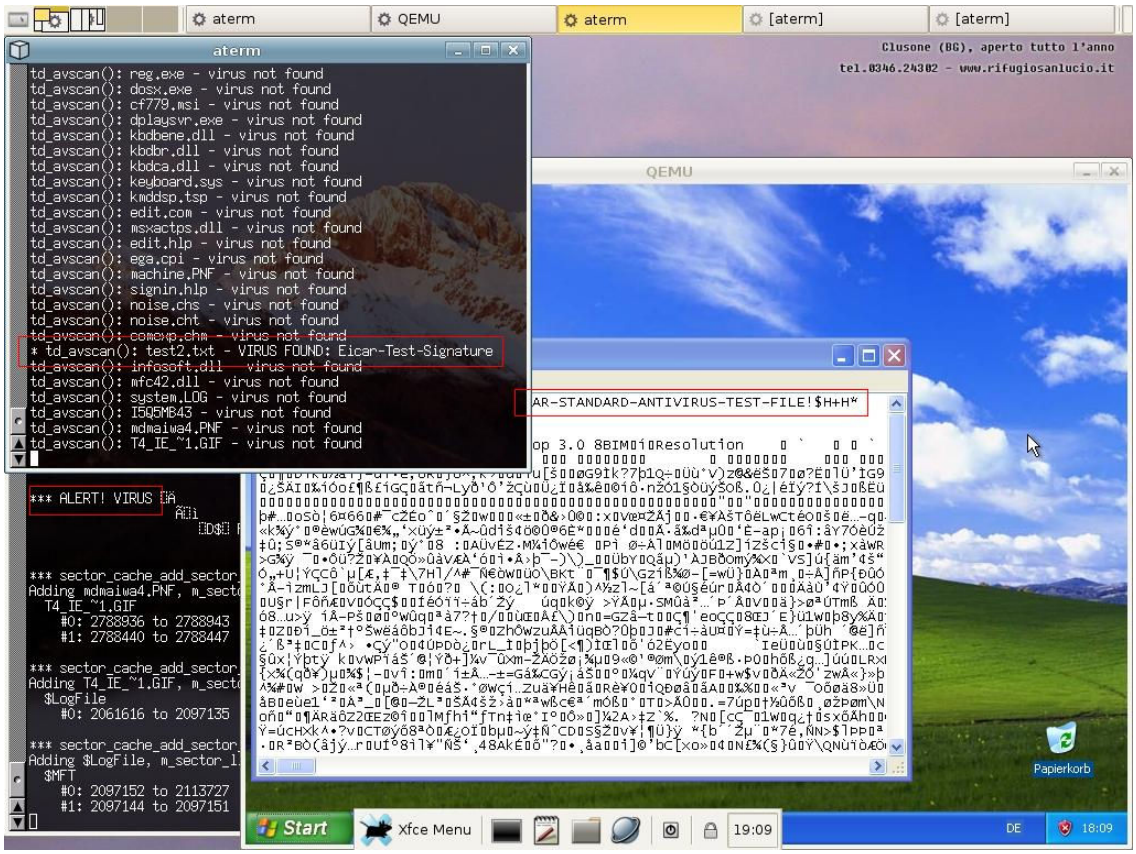


Figure 21 - On-access scan – screen-shoot: virus found

6. Network scan

Network scan deals with the problem of scanning TCP/IP data incoming and outgoing the user system for known viruses and malware. At run-time, the scan engine identifies and deletes threats spreading over "untrusted" networks before reaching and infecting the user environment. At the same time, in case of infection, neither viruses nor malware are able to propagate outside the secured computer: the antivirus analyzes any data before they are transmitted to the physical interface. Cleaning the user's connection in both directions, the personal computer is protected from external threats and vice versa the neighbourhood (e.g. the LAN) is secured from possible virus spreads.

To accomplish this goal, the following software architecture has been designed and implemented. Virtualization provides the guest OS with a virtual Ethernet device, and its associate network, known as *VM-NET*, used to communicate with external networks. Therefore, the user system detects just one interface, using it for its network connections, independently from the type or the number of physical devices. All network connections established by Windows are carried out on this virtual network and automatically routed by the *secureOS*, from the *VM-NET* to the appropriate physical interface. This routing mechanism is encapsulated into the *security shell*, in order to be completely hidden from user prospective.

Since all network connections related to the Windows system come over the virtual network, it is quite easy to capture any data from the sub layer system. An *AV stream interceptor* is installed in the secureOS, on the VM-NET segment connecting the *virtual*

environment with the kernel routing service. Then, the captured data are analyzed by the antivirus engine for viruses and malware.

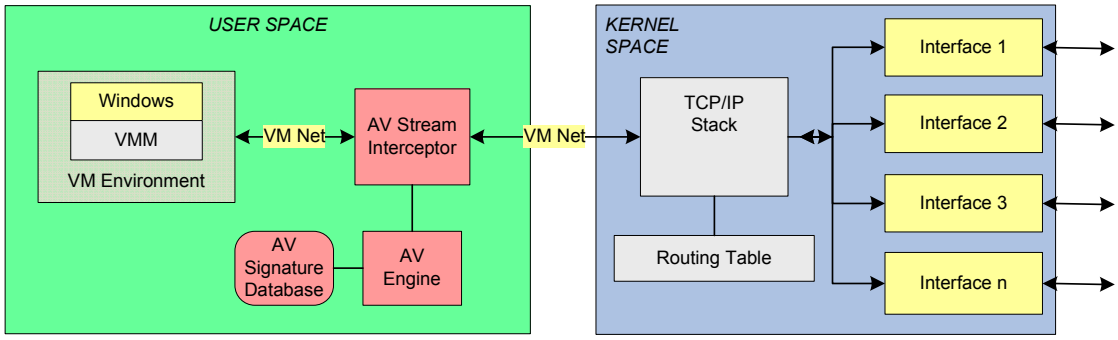


Figure 22 - Network scan - basic architecture

The great benefit of the proposed solution is the capacity of virus-scanning any TCP/IP stream, independently from the physical interface and inconspicuously to the user and Windows system.

Free Software solutions

Any antivirus can be used to implement the *AV Engine* and the *Signature Database* components. Since Clamavis does not natively support network scanning, it should be coupled with an external *Interceptor* component, which would capture the data packets and would forward them to the antivirus core. Because of the Free Software license adopted by ClamAV antivirus (GPL), at least three Free extensions are indexed on the 3rd party software page of the project[1]:

[1] ClamAV 3rd party software, http://www.clamav.net/3rdparty.html

- *kclamav*: light and streaming version of Clamavis, built as a Linux 2.6 kernel module and hooked via the *Netfilter* API. The ClamAV virus database is loaded into kernel memory. Kclamav still in PRE-ALPHA status (version 0.0.1, released in January 2006), and its installation is quite tricky: it requires Linux Kernel and Netfilter API patching, and some rules of *Iptables* to be compiled;
- *snort-ClamAV*: Snort pre-processor that scans data in network packets for viruses. Snort is a Free Software network intrusion prevention and detection system utilizing a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods.
- *snort_inline ClamAV pre-processor*: snort-inline[1] is shipped with a ClamAV pre-processor that scan network traffic for viruses. It can choose which protocols must be monitored. When a virus is detected, snort-inline can send a reset and drop the relative packets.

6.1. VPNs

A general issue in the antivirus research field is the scanning of encrypted communication channels. VPNs[2] are frequently used by companies to interconnect

[1] From snort_inline homepage, http://snort-inline.sourceforge.net/: snort_inline is basically a modified version of Snort that accepts packets from iptables, via libipq, instead of libpcap. It then uses new rule types (drop, sdrop, reject) to tell iptables whether the packet should be dropped, rejected, modified, or allowed to pass based on a snort rule set. We can think of this as an Intrusion Prevention System (IPS) that uses existing Intrusion Detection System (IDS) signatures to make decisions on packets that traverse snort_inline.

[2] VPN is acronyms for Virtual Private Network.

geographically distributed offices and departments, with a secure communication channel over a publicly "untrusted" accessible network (normally theInternet). The access to the company's Intranet is usually permitted from outside only via VPN. VPNs use cryptographic tunnelling protocols to provide the intended confidentiality, sender authentication, and message integrity. Cryptography can be applied to the transport layer[1] or more securely to the bottom network layer[2]; in both cases the applicative payload containing the data is completely encrypted.

VPN consequences on virus protection

VPNs build encrypted network channels to ensure the confidentiality of data and to prevent malicious users from sniffing and stealing confidential information. However, the encrypted nature of VPN-protocols prevents trusted applications to access information transferred on VPN channels. This native limitation is reflected on the *AV stream interceptor* previously described, which cannot scan VPNs established by the Windows system. Viruses and malware could spread through the VPN channel from the Internet to the user-system, and later on to every department connected via VPN.

[1] OpenVPN encrypts at transport layer, http://openvpn.net/

[2] IPSEC running in tunneling mode encrypt at network layer, http://www.ietf.org/rfc/rfc2401.txt

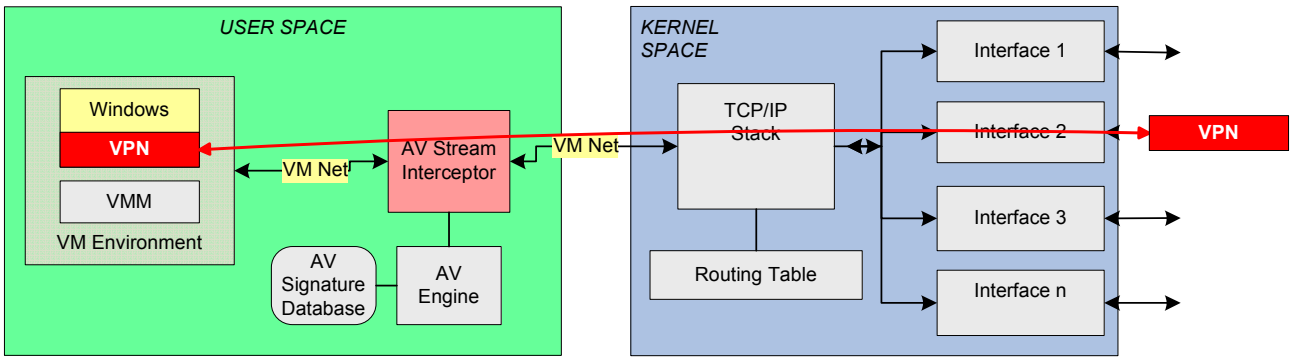


Figure 23 - VPN scan problem

An innovative approach has been suggested as a solution to the VPN virus scanning issue, ensuring that all data transmitted on encrypted channels, independently from the layer (transport or network) or the encryption algorithm used, get scanned for viruses and malware.

The answer

The idea is rather simple: VPNs are established from the *secureOS* and their content is extracted, virus scanned and bridged to Windows (and vice versa). The end user manages the encrypted channels from a control panel (the *User Control Center (UCC)),* where preconfigured VPNs are deployed by the security administrator from a central site to each personal computer, as *Network Connection Control*[1] policies (basically an XML file containing the configuration). Normally the user cannot create new VPNs but should consider using the preconfigured ones. The NCC module offers a visual interface

[1] NCC as acronym of Network Connection Control

to list preconfigured VPN, activate and shutdown single encrypted channels, show information regarding the status.

When a VPN has been selected by the user, its configuration is sent to the appropriate agent and an internal mechanism is run. The VPN agent makes use of a database to establish encrypted tunnel and to bring the encrypted network channel up. This includes information regarding available VPN drivers and a compatibility list of clients and servers. In this way, the agent loads the appropriate driver and uses the client for the VPN-server specified in the received configuration.

When the VPN is correctly established and it is operative into secureOS, the agent configures the routing-table to bridge data extracted from VPNs to Windows, by the virtual network.

As it can be seen in the figure, VPNs (red colour) terminate inside the secureOS; their network data are “extracted” as plain text (blue colour) and forwarded through the virtual network to the VM environment and to Windows. The antivirus is therefore able to capture and virus-analyze VPN data (in blue), now running in plain text on the VM-Net, just like of not-VPN traffic.

The bridging mechanism continues in both directions until the user closes voluntarily the secure tunnel. At that point, VPNs established from the *security shell* are shutdown.

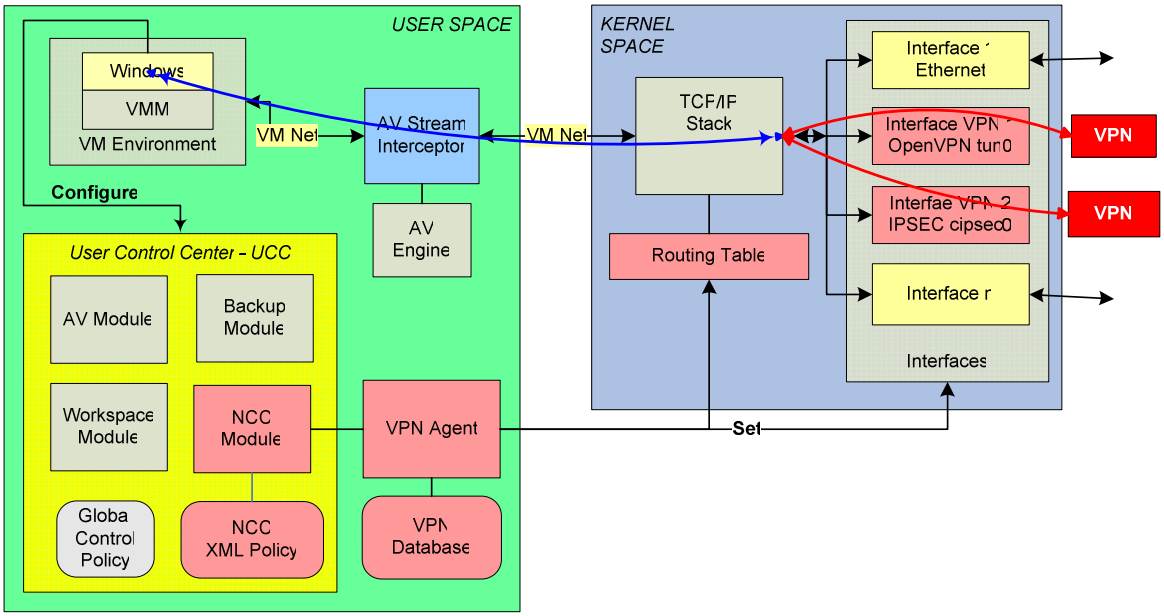


Figure 24 – Network scan – architecture for VPNs

Scan of encrypted channels guarantees higher protection against viruses and malware, especially when VPNs are the only communication channels allowed, either for the single computer than its neighbourhood. A network infrastructure built entirely with VPNs, could be compromised globally by viruses, spreading without control inside encrypted channels.

Unfortunately, the described architecture has not been practically implemented because of the lack of commercial Linux-compatible clients. Since many vendors supply only Windows VPN clients and a strict point is portability, to be intended as migration from an existing system, it becomes mandatory to maintain current VPN infrastructures and establish Virtual Private Network within Windows. As already discussed, this configuration prevents antivirus installed into *secureOS* to scan VPN data, with negative security implications on the user environment.

6.2. TLS/SSL and SSH protocols

In the previous chapter, the problem of scanning TCP/IP network streams encrypted at Transport- or Network layers has been analysed, as it has been done in the Virtual Private Networks. The idea of managing VPNs from the *security shell*, managing new encrypted tunnels at user demand, enables antivirus to intercept any data for a real-time virus scan, before being bridged to the user system.

However when a network stream is encrypted at application layer, the encryption is directly established between applications and data cannot be accessed by the antivirus of secureOS, that operate at lower layers. Viruses and malware spreading across these communication channels cannot be intercepted by antivirus, because information is encrypted.

This is the case of TLS/SSL and SSH protocols, used to access remote information is a secure wavy, through application layer encryption. Both TSL/SSL and SSH are distributed as suite of set of network protocols that allows establishing of secure channels between two end-points, by the use of public-key cryptography features, and providing end-points authentication and communications privacy. TLS for example works involving three basic phases:

1. Peer negotiation for algorithm support
2. Public key encryption-based key exchange or certificate-based authentication
3. Symmetric cipher -based traffic encryption

SSH is typically used to log into a remote machine and execute commands, but it also supports tunnelling, forwarding arbitrary TCP ports and X11 connections; it can transfer files using the associated SFTP or SCP protocols. From an evil prospective SSH protocols suite could be adopted by an attacker to transfer evil code (as backdoors and DoS worms) on the target machine, avoiding antivirus controls and run unauthorized software. At the same time, SSH could be used to instantiate connections from the target machine to outside world, avoiding possible content-filtering controls and stealing sensible information from the company network.

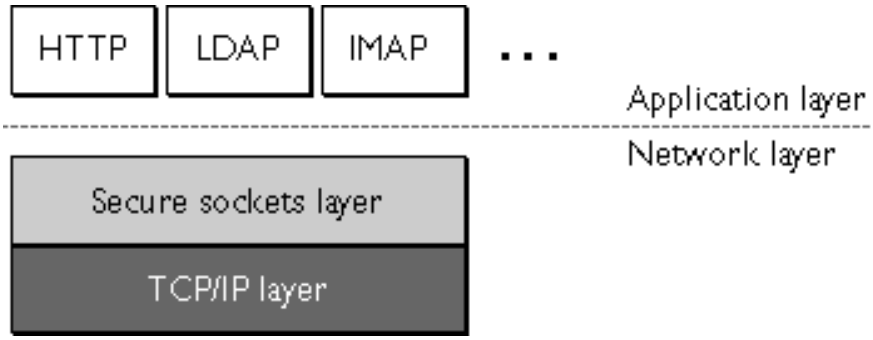


Figure 25 – SSL layer design

Technically TLS/SSL runs on layers beneath application protocols such as HTTP, FTP, SMTP, and NNTP, and it can add encryption security to any protocol that uses reliable connections (such as TCP). However it's most commonly adopted to implement HTTPS, an extension to standard HTTP protocol, used to secure World Wide Web pages for applications such as electronic commerce, in which sensitive information such as credit-card numbers are transmitted over Internet. SSL is also used in conjunction with mail protocols, like SMTP and POP3, to grant data confidentiality and personal

privacy. When emails are sent (received), their content is directly encrypted at application layer by the mail-client (server) and transmitted in a secure form over the network.

At the same time, the advantages offered by the TLS/SSL protocol represent a limit to antivirus efficiency, since web pages transmitted over HTTPS channels and emails received and sent via POP3S and SMTPS cannot be scanned by the antivirus engine installed in a security shell.

Nowadays a lot of viruses and malware spreads across World Wide Web using security flaws discovered in common browsers, as Internet Explorer, and http servers, as Microsoft IIS. Core-Red[1] for example is a well-known worm that exploits a buffer overflow in Microsoft IIS index-service library and installs itself on the victim target, permitting execution of arbitrary code in the Local System security context. This level of privilege effectively gives an attacker complete control of the victim system. Moreover, in the earlier variant of the worm, victim hosts with a default language of English experienced the following defacement on all pages requested from the server:

HELLO! Welcome to http://www.worm.com! Hacked By Chinese!

In addition to possible web site defacement, infected systems may experience performance degradation as a result of the scan activity of this worm. This degradation can become quite severe since it is possible for a worm to infect a machine multiple times simultaneously.

[1] CERT Advisory n.2001-19 on Code Red worm:http://www.cert.org/advisories/CA-2001-19.html

Malware propagates easily also as email attachment, taking advantage of security flaws of email-clients or browser-shared libraries. This is the case of the *Nimda*[1] worm, which exploits the "Automatic Execution of Embedded MIME Types" vulnerability discovered in Microsoft Internet Explorer 5.5 SP1, to propagate through email arriving as MIME "multipart/alternative" and execute arbitrary commands on the victim host. The impact of the Nimda worm is similar to the Core-Red one.

While a standard antivirus installation is not able to extract and scan data-streams encrypted at application layer, the idea described here permits to run the antivirus service also on connections encrypted at application layer, just for a restricted set of applications, as SSL/TLS over HTTP (HTTPS) and SSH suite (STFP and SCP). Due to the high number of malware spreading over Web through HTTP(s) protocols, it is of primary importance to support HTTPS antivirus filtering.

The idea

In order to virus-scan Web pages served through HTTPS, the idea is to implement a sort of transparent proxy agent that acts as *Man-In-The-Middle (MITM) attacker*. The agent creates a couple of private/public key for each new HTTPS connection, and intercepts TSL/SSL handshaking to realize a MITM attack. It decrypts HTTPS traffic and sends

[1] CERT Advisory n.2001-26 on Nimda worm: http://www.cert.org/advisories/CA-2001-26.html

Web content to the antivirus engine. If a virus is not found the Web request is re-encrypted for the user browser, otherwise an alarm is raised.

The Man-In-The-Middle attack could be explained as “story” with three “actors”: Alice as end-user (the victim), Bob as Web-server (www.server.com) and Mallory as attacker (the transparent proxy www.attacker.org). Suppose Alice wishes to communicate with Bob via HTTPS, and that Mallory wishes to eavesdrop on the conversation, or possibly deliver a false message to Bob. To get started, Alice asks Bob for his public key. Bob sends his public key to Alice, but Mallory intercepts it. Mallory simply sends Alice a public key for which she has the private, matching, key. Alice, believing this public key to be Bob's, then encrypts her message with Mallory's key and sends the encrypted message back to Bob. Mallory again intercepts, decrypts the message, keeps a copy, and re-encrypts it (after alteration if desired) using the public key Bob originally sent to Alice. When Bob receives the newly encrypted message, he will believe it came from Alice.

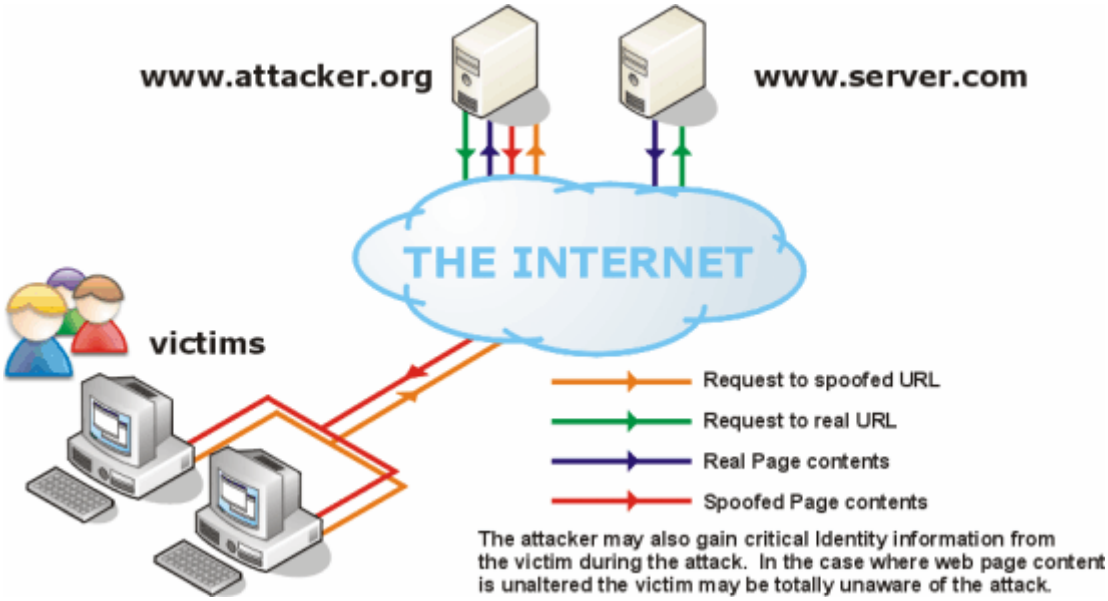


Figure 26 - Man-In-The-Middle attacks against HTTPS

A PoC[1] has been developed by Simon Newton to show MITM attack for HTTP over TLS/SSL.

Since MITM is to be considered more as an attack technique than a SSL feature, it should not to be considered a reliable solution, but more as a hack that permits to intercept and scan Web encrypted pages with strong limitations, for a possible product (and not for the research environment).

For example, MITM mechanism implemented in the proxy can't work when digital certificates are used. A certificate is a combination of public key and its digital signature, made by an external authority (the Certification Authority) that grants key

[1] Ssl_proxy PoC: http://www.nomis52.net/data/ssl_proxy.pl

belong to an individual. In this occasion MITM attack is immediately detected by both communication sides, blocking SSL authentication.

Two approaches can be applied to **virus-scan SSH (and SFTP/SCP)** network connections, one which is exactly the same used for HTTPS and that consists on implementing a sort of agent that acts as hidden proxy running MITM attacks, the other similar to idea presented for VPN scan. The second one in particular makes use of the SSH embedded proxy capabilities that permit to instantiate a fictitious session from the secureOS to target, and bridge content data to the user-system as plain-text (interposing the antivirus interceptor to capture and send the network stream to analyze to the antivirus).

6.3. Removable network devices

In last couple of years, removable network devices as *LAN*, *WLAN* and *UMTS* have become more widespread in IT market, distributed either in *PCMCIA* and *USB* format. Each computer is equipped with at least two USB ports and PCMCIA slots (for laptops). Almost any operating system, as *Windows XP*, *Linux* and *MacOS*, support hot-plug interfaces and numerous devices, as printers, scanners, cameras and mass-storage, through periodic drivers upgrades.

Removable network devices are used for many purposes. LAN PCMCIA cards allow physical connection of old laptops to network cabled infrastructures. Ten-year-old laptops are usually not equipped with RJ-45 connectors or 100mbit compliant Ethernet chipset, used in newer LANs. On the other hand, Wireless devices are used to access network infrastructures easily and comfortably, without using wires. At the same time wireless technology guarantees worldwide Web access. By UTMS technology, telephony companies offer special contracts for worldwide Internet access. In highway petrol station, airports, train stations, and also bars, Internet access is offered to customers, for free or cheaply, by WLAN technology (technically known as IEEE 802.11 or WiFi standard).

An interesting Spanish project, called FON[1], aims to create a WiFi community to permit free world-wide Web access. FON homepage is self-explaining: "*FON is the largest WiFi community in the world. Our members share their wireless Internet access at home and, in return, enjoy free WiFi wherever they find another Fonero's Access Point. It all started as a simple idea. Why should you pay for Internet access on the go when you have already paid for it at home? Exactly, you shouldn't. So we decided to help create a community of people who get more out of their connection through sharing.*"

Security implications

Wireless infrastructures are for their nature insecure, since data are not wired-embedded, but are transmitted as radio signals in the air; information are not physically protected as in standard wired LAN. Wireless communications are much easier to be intercepted and analyzed by evil users, looking for sensitive data. Many wireless antennas are available at cheap price on market.

So, wireless networks should be used with attention and parsimony, only when strictly necessary (LANs must be preferable when available), and strong encryption

[1] Movimento FON, http://en.fon.com/

mechanisms must be implemented: for example, WPA2[1] in conjunction with a RADIUS[2] server or VPN.

Moreover, fake public WiFi hotspots[3] could be configured as *traps*, in order to force unaware users to associate with them and to send personal information, like accounts and credit-card numbers, to the attacker.

At the same time, it is to be considered that generic user is normally not a computer expert, so could involuntarily associate his laptop to untrusted WiFi networks. When a wireless card is found, Windows XP supplies the user with a user-friendly GUI where it is possible double-click for the desired network. Easily and frequently, the user makes a wrong decision and associates his personal laptop with an evil hotspot.

At the end, but not less important, it is the problem of virus scan many hot-plug network interfaces. When a removable device is configured within Windows, data transmitted through it are not intercepted by processes running outside the operating system, as it could be the antivirus layer of secureOS. Thus, the idea previously described to intercept network streams from the bottom layer matches the problem, but it is not able

---

[1] WiFi Protected Access is a class of systems to secure wireless networks, created in response to several serious weaknesses found in the previous WEP system. http://en.wikipedia.org/wiki/Wi-Fi_Protected_Access

[2] Remote Authentication Dial In User Service (RADIUS) is an AAA (authentication, authorization, and accounting) protocol for applications such as network access or IP mobility. It is intended to work in both local and roaming situations. FreeRADIUS is a valid Free Software alternative to commercial ones, http://wiki.freeradius.org/Main_Page

[3] Hotspots are venues that offer WiFi access. The public can use their laptop, PDA, or Dual-mode phone to access the Internet, http://en.wikipedia.org/wiki/Hotspot_%28Wi-Fi%29

to solve it. Therefore, an improved extension, which it is able to handle hot-plug network devices, has been researched and it is presented here.

Massive distribution of removable network devices and the insecure nature of the wireless technology make important to secure their use, in order to grant confidentiality and integrity of personal computer that uses these devices. The proposal solution gives an answer to these open issues.

Security policies for hot-plug devices are centrally configured by the security administrator, remotely deployed to each personal computer, and locally enforced, to be strictly respected by individual users. A device control mechanism has been implemented to wrap removable devices "offered" to user-system and network connections are managed from the *security shell*.

In this way, local users cannot connect to untrusted and unconfigured Internet hotspots. At the same time, information data carried out over hot-plug network channels are secured and virus-scanned from the secureOS antivirus layer.

The idea

The innovative idea is to move the *hot-plug layer* from user-system to security shell. Virtualization permits to manage devices "offered" to the guest OS through a sort of access-list. Normally when a removable interface is installed (e.g. plugging in a wireless device into the laptop's PCMCIA slot), Windows installs the drivers for the new peripheral and automatically configures it, providing immediately functionality to end user.

The removable devices are managed from the secureOS and Windows is denied from installing and configuring unwanted hardware. Two big benefits are achieved: first, new device installation is performed only after being accepted by security policy, enabling an optimal policy enforcement mechanism; second, the hot-plug network interfaces (both wireless and LAN) are automatically and inconspicuously scanned for viruses and malware by the existing TCP/IP network layer. Virtualization is essential to create this security layer that wraps removable devices and manages them safely.

To user is offered the possibility to manage Hot-plug devices graphically by the *User Control Center*: a module called *Network Connection Control* is designed to prompt the user for the list of preconfigured connections, previously centrally managed by security officers.

When a removable network device is plugged in, the user is informed about the available connections: for example, an UTMS connection to company provider or a wireless association with trusted hotspots. Administrators could enable the user to setup his own wireless connections or to reconfigure an existing one, after verification by security policies.

At that point, the appropriate agent setups the desiderate network interface, querying an *hot-plug device database* for the appropriate kernel driver to load. The database contains two tables: the device type/name and the appropriate Linux driver (driver table); the application used in configuration and their syntax (configuration table).

The configuration is translated from XML to the specific format of network utilities like *ifconfig* and *iwconfig*. When the interface is up, the routing table is appropriately reconfigured in order to bridge network data to user-system. Network packets are routed from Windows to hot-plug interfaces (and vice-versa), enabling the user to send and receive data transparently over the plugged card. With this architecture, antivirus scan is also performed over removable network interfaces, automatically and without any extra configuration.

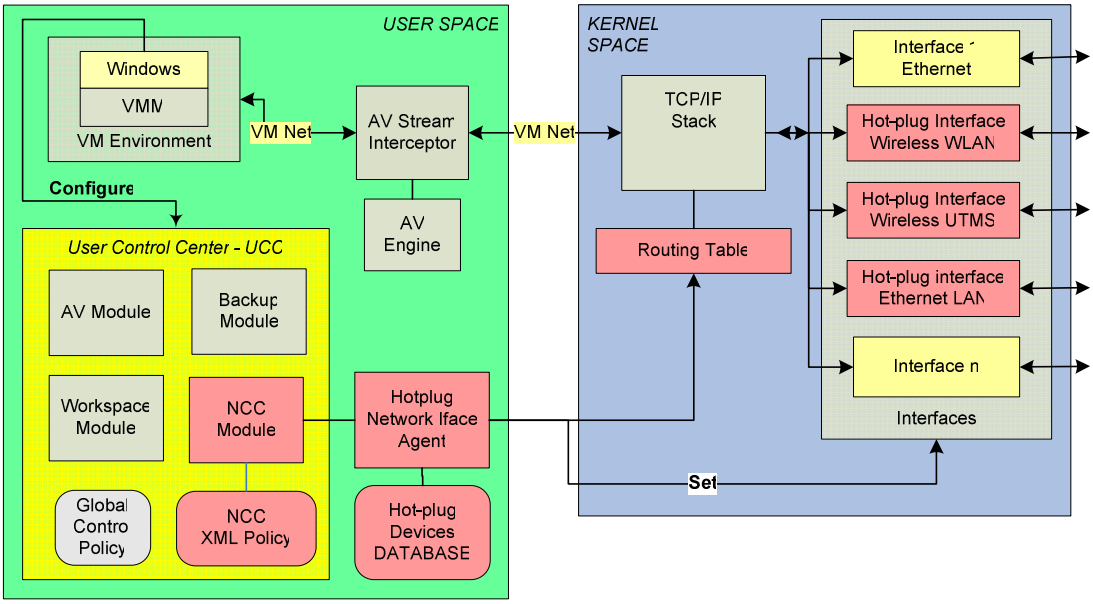


Figure 27 – Network scan – architecture for removable network devices

Moving "hot-plug device management" should not limit user availability. This means to be able to support network Windows features after migration process. Therefore, it is not enough be able to set-up removable network interfaces and bridge them to user environment, but some other extra-mechanisms should be "cloned" from Windows.

For example, what happens when the user travels with the laptop? He would like to use a new wireless connection, not already pre-configured. Here the agent scans the neighbouring area for existing hotspots (e.g., with the Kismet[1] program), parses the results and presents them as a multiple-choice list to the user via the User Control Center.

What has been presented here is a novel approach to secure hot-plug network devices, as LAN, WLAN, and an UTMS cards, available either in USB and PCMCIA format. The greater security comes from two separate concepts: policy enforcement and antivirus scan.

Network policies for hot-plug devices are centrally configured by the security administrator, remotely deployed to each personal computer, and locally enforced. A device control mechanism has been implemented to wrap removable devices "offered" to user-system, and network connections are managed from the *security shell*. Antivirus scan is inconspicuously achieved on data transmitted over removable devices, using the existing AV layer.

[1] From project homepage, http://www.kismetwireless.net: "Kismet is an 802.11 layer2 wireless network detector, sniffer, and intrusion detection system. Kismet will work with any wireless card which supports raw monitoring (rfmon) mode, and can sniff 802.11b, 802.11a, and 802.11g traffic."

7. Further security with disk encryption

The use of virtualization to encapsulate the user environment in a protected "security system" is compatible with the principles of *disk encryption*. It makes sense to link these two technologies, embedding a hidden sub-layer that protects both, security shell and virtual system from confidentiality attacks. The user system can be secured automatically and transparently by an "external" encryption, and no configuration or encryption support is required in the user system. A pre-boot authentication secures the access to the personal computer.

Mobile computing and data portability have changed the paradigm for security management. Information that was once stored exclusively on mainframes and storage servers are now frequently stored outside the corporate perimeter as hard disk data on laptop computers, which are highly susceptible to theft or loss. In fact, recent studies reveal that more than 80% of U.S. companies have lost laptops with sensitive data in the last year, and a single laptop theft can cost to organizations, especially when private consumer information is exposed.

The solution, accepted by research community and software market, is *data encryption*, which results in good protection of confidential information. In computer security, the most common way to protect confidentiality and authenticity of transmitted and stored data is encryption, which allows users to obscure personal information to make them

unreadable without a special "knowledge", as a password, a software key, or a physical token.

A typical approach to protect data confidentiality and authenticity is to *encrypt single files*, using PGP[1] or other similar tools. Explicit decryption is required before a file can be modified and after plaintext file is no longer needed; it must be re-encrypted and securely deleted (wiped). File encryption is widely used to archive secure data transmission, such as in secure email deployment. However, other more inconspicuous methods have been devised for storage confidentiality. These can be categorized into file-system and disk encryption.

In *file-system encryption*, often called folder encryption, individual directories can be encrypted. The main problem is that temporary files (such as swap devices / page files, various caches, "temp" directories) and metadata (such as the directory structure, file names, size and timestamp) are often stored unencrypted. This exposes sensitive information as clear text. Notable file-systems that support this kind of encryption include Microsoft's EFS[2] and UNIX's CFS[3] and TCFS[4].

---

[1] Pretty Good Privacy (PGP), originally written by Phil Zimmermann in 1991, it's a public key encryption software, nowadays becomes the "de facto standard" in single file encryption as in email cryptography.

[2] Microsoft Windows 2000 and later O.S. offer the Encryption File System (EFS) layer to encrypt transparently single files and folders on NTFS volumes.

[3] M. Blaze. *A Cryptographic File System from Unix.* Proc. First ACM Conference on Computer and Communication Security, Fairfax, VA, 1993.

[4] G. Cattaneo, L. Catuogo, A. Del Sorbo, and P. Persiano. *The Design and Implementation of a Transparent Crypthographic File System for UNIX.* USENIX Annual Technical Conference 2001.

A complementary approach known as *disk encryption* encrypts the whole hard disk from the bottom (below the file-system layer), so as all data are encrypted inconspicuously. Independently from installed operating system, each hard disk sector get encrypted and decrypted individually, once it is access from the file-system. The operating system, together with drivers, internal registries, and system configurations gets encrypted with a unique key. At the same time, each user's profile and personal document is secure stored in the same way.

Disk encryption is considered a more secure approach, having several benefits compared to conventional approaches. Major encryption coverage is obtained, since file-system metadata, temporary files and swap-space are encrypted too; no temporary data remains unencrypted, exposing sensible information to an attacker. Since disk encryption is completely transparent, no user interaction is required for specific file and folder encryption. In disk encryption, the whole hard disk is protected with a global key, unlocked at boot-time, so as to make all data decryptable for the operating system when computer is powered on. The credentials are inputted via a *pre-boot authentication*, which usually prompts for username and password, manages network authentication or hardware tokens. Special pre-boot authentication procedures can be implemented to archive fine-grained user management and better user policy enforcement.

Disk encryption is sometimes used in conjunction with *file-system encryption*, resulting in a more secure implementation. Since disk encryption uses the same key for encrypting the whole volume, all data are decryptable when the system runs. If an attacker gains access to the computer at run-time, he has access to all files. Conventional file and folder encryption instead allows different keys for different

portion of disk, and thus an attacker cannot extract information from still-encrypted files and folders.

Cryptography notions

For a cryptographic point-of-view, disk encryption is traditionally implemented as *block cipher encryption* on individual sectors, the basic unit of hard disk storage, consisting of 512 continuous bytes (4096 bits). Individual encryption is required to perform quick single sector random access reads and writes. A block cipher is a symmetric encryption algorithm that operates on fixed-length groups of n bits, termed block[1], as 128bit for the Advanced Encryption Standard (AES).

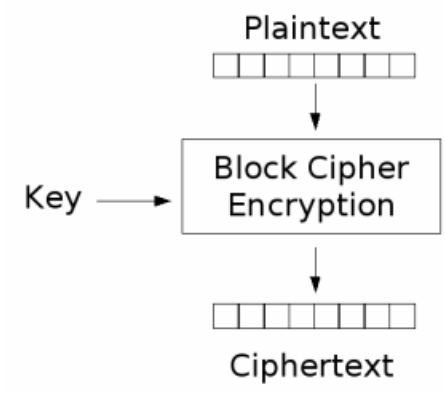


Figure 28 – Block cipher encryption

When encrypting, the cipher function E takes an n-bit input block P_n and a k-bit key, yielding an n-bit output block E_k. The decryption function D is the inverse of encryption, so that:

$C_n = E_k (P_n)$

$P_n = D_k (C_n)$

[1] To not be confused with *hard disk block* term, denoted a grouped set of sectors, as 16 sectors. Here the term refers to the crunch of data on which the cipher works, as 128bit for AES.

A block cipher operates on a single block, and thus encrypting all k blocks in the sector requires some "mode of operation" to concatenate different outputs so as to provide confidentiality for the entire sector. The simplest of the encryption modes is the *electronic codebook (ECB) mode*, in which the message is divided into blocks and each block is encrypted separately. The disadvantage of this method is that identical plaintext blocks are encrypted into identical cipher-text[1] blocks; thus, it reveals data pattern and does not provide good quality message confidentiality.

For this reason, a recursive algorithm has been devised and become the "de facto standard" for disk encryption. Here is the importance from spending some pages on this argument.

In the **cipher-block chaining** (CBC) mode each block is XORed with the previous cipher before being encrypted; each cipher-text is dependent on all plain text blocks processed up to that point. To make each message unique, an *initialization vector (IV)* is used as input in the first block, resulting in the following formulas:

$$C_0 = IV$$

$$C_n = E_k (P_n \, XOR \, C_{n-1})$$

$$P_n = D_k (C_n) \, XOR \, C_{n-1}$$

Since CBC encryption is a recursive algorithm, the encryption of the n^{th} block requires the encryption of all proceeding blocks, 0 until n-1. This is an undesired property; therefore, the CBC chaining is cut every sector and restarted with a new initialisation

[1] An encrypted block (the output) of a cipher function is termed cipher-text.

vector, so that sectors are encrypted individually. The choice of the sector as smallest unit matches with the smallest unit of hard disks, where a sector is also atomic in terms of access.

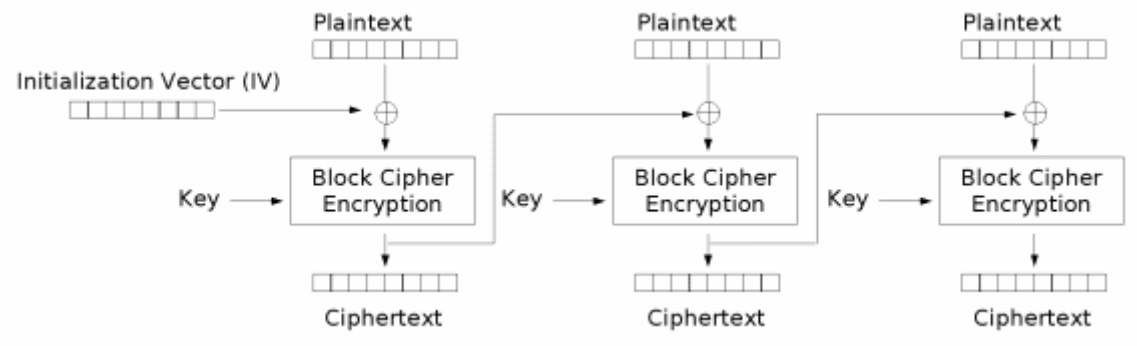


Figure 29 - Cipher Block Chaining (CBC) mode encryption

The IV for sector *n* is usually set to the 32-bit version of the number *n* encoded in little-endian padded with zeros to the block-size of the cipher used, if necessary. This is the most simple IV mode, but at the same the most vulnerable.

Security analysis

Because of the "de facto standard" of CBC in disk encryption and its adoption in the implementation, a security analysis on risks concerning the use follows.

In some condition, it is conceivable that disks encrypted in CBC-mode are subjected to repeated scans and active manipulation attacks based on such scans. Some well-known attacks, pointed out in 2002 by J. Etienne are:

- corruption: corruption of chosen data blocks is difficult to detect. As CBC decryption has little error propagation, modifying a cipher-text block within sector only points

out the corresponding plain-text block and causes the chosen bit to change only the block immediately following it;

- translation: CBC decryption depends on two variables, C_{n-1} and C_n. Both can be modified at free will. To make meaningful modifications, an attacker has to replace the pair C_{n-1} and C_n with other cipher-text pair from disk. The first block C_{n-1} will decrypt to garbage, but the second block C_n will yield a copy of the plain text of the copied cipher block. This attack is also known *as copy&paste attack*;
- reverting: it is relatively easy to revert chosen sectors to their previous values without detection.

Next to these attacks, an embedded vulnerability of the CBC-mode permits to reveal data patterns and facilitates a cryptanalyst to extract information as plain text, without knowing the encryption key. This opens a big toolbox of *watermark attacks.*

The consequence is exactly the same reported for the ECB-mode, where identical plain-text blocks are encrypted into identical cipher-texts, suggesting data structure and content. However, it has been proved that the risk deriving from this vulnerability is minimal; in fact, the attack requires about 146 million TB of storage to find at least one pair of identical cipher blocks, and much more to rebuild a meaningful data pattern[1].

---

[1] Recently it has been discovered by O. Saarinen how to raise this probability, opening the possibility for new critical attacks. With public IV and the predicable difference introduced in the first blocks of a sequence of plain text, data can be watermarked, which means, the watermarked data is detectable even when the key has not been recovered. As the IV progresses with a foreseeable pattern and is guaranteed to change the least significant bit ever step, it can be build identical pair of cipher text by writing three

The attack is based on the property that in CBC decryption the preceding block's influence is simple, that is, it's XORed into the plain text. If two identical cipher blocks C_n and C_m are found, the attacker knows that both have been formed according to:

$$C_m=E(P_m \text{ XOR } C_{m-1})$$

$$C_n=E(P_n \text{ XOR } C_{n-1})$$

Since he founds that $C_m = C_n$, it holds:

$$P_m \text{ XOR } C_{m-1} = P_n \text{ XOR } C_{n-1}$$

which rewritten is

$$C_{m-1} \text{ XOR } C_{n-1} = P_n \text{ XOR } P_m$$

Since the left part is known to attacker, the difference between the two plain texts is derivable (if one of the blocks happens to be zero, the difference corresponds to the original content of the other related block).

If C_m or C_n is the first block, the IV must be examined (this is possible cause in standard CBC-mode the initialization vector is public formed from sector number). For this reason, the new ESSIV IV-mode has been proposed:

---

consecutive sectors each with a flipped LSB relative to the previous (the reason for three instead of two is that the second least significant bit might change as well). This "public-IV"-driven CBC encryption will output exactly the same cipher text for two consecutive sectors. An attacker can search the disk for identical consecutive blocks to find the watermark. This can be done in a single pass, and is much more feasible than finding two identical blocks, that are scattered on the disk, as in the previous attack. A few bits of information can be encoded into the watermarks, which might serve as tag to prove the existence searched sensitive material.

E(Sector|Salt) IV, short ESSIV, derives the IV from key material via encryption of the sector number with a hashed version of the key material, the salt. ESSIV does not specify a particular hash algorithm, but the digest size of the hash must be an accepted key size for the block cipher in use.

Since ESSIV initialization vector depends on a confidential piece of information, the sequence of IV is not known, and *watermark attacks*, based on public-IV, can't be launched.

New chaining mode as CMC[1] and EME[2] are considered secure, because they solve these open issues, but are heavy in terms of computational performance. The novel LRW[3] seems the correct answer; in fact, it has been designed to be robust against known vulnerabilities and to provide good performance. CMC- and EME-modes are already implemented in some commercial encryption software, while LRW is still in an earlier phase.

---

[1] Shai Halevi and Phillip Rogaway, A Tweakable Enciphering Mode, CMC Cryptology ePrint Archive: Report 2003/148, http://eprint.iacr.org/2003/148

[2]IEEE P1619 EME-32-AES Teakable Wide-block Encryption draft, http://grouper.ieee.org/groups/1619/email/pdf00011.pdf

[3] LRW draft, http://grouper.ieee.org/groups/1619/email/bin00014.bin

7.1. Design

The goal has been to disk-encrypt the user system from "the outside", without requiring no encryption support or configuration from it, and realizing a hidden and automatic security protection. The whole personal computer has been encrypted at once: the *security shell*, with its operating system (data and swap), security layer (agents and configurations), security policy database and VMM, and each virtualized user-system. A unique key has been used, but up to eight keys associated with eight distinct users or local administrators could be adopted. A pre-boot authentication secures the access to the personal computer.

A scenario has been considered for the test implementation and its extension has been suggested. The whole disk labelled as *root partition* is encrypted at once, and a *boot partition* used to start the computer and run a pre-boot authentication. The boot partition is accessed at an early stage, so it cannot be encrypted. If computer gets lost or stolen, no data can be extracted from the hard disk, which is encrypted with strong algorithms and protected by a secure authentication mechanism (even opening the computer case, extracting and installing the hard disk on a second machine does not give access to confidential information).

The computer disk can be decrypted and accessed exclusively by a *normal user*, authenticated via username and password, and a *local administrator* via hardware token as USB-stick, achieving better security and physically protecting administrator login. In detail, this stick is entirely encrypted and is unlocked with a password prompted at pre-

boot time. It contains the owner's username and a sequence of computer-ID and disk password pairs, enabling the access to multiple encrypted systems with a unique authentication token. It can be considered as a master key, which contains different passwords to many associated computers. This feature is quite useful and fits this scenario, in which a local administrator is designed for manage a set of computers, independently and differently configured, but would like to remember just the single password of his personal token. For a secure point-of-view, token- and disk- passwords are decoupled, and local administrator does not need to know the second ones, which are stored into the token from central administration.

The boot partition contains a special *admin token* used as "test" for administrator authentication. In the current implementation, when the root partition password is retrieved, the admin token is tried to be unlocked; if decryption fails the user does not own administrator rights (disk and admin-token are obviously encrypted with the same key). Alternatively, the token, now unlocked, communicates to the system that the administrator has been correctly authenticated during the pre-boot phase.

This admin token gets such a high importance, because it represents a secure communication channel between pre-boot- and system-boot- phases for the group the user belongs to (generic or administrator); no sensitive information as the key's passwords are stored.

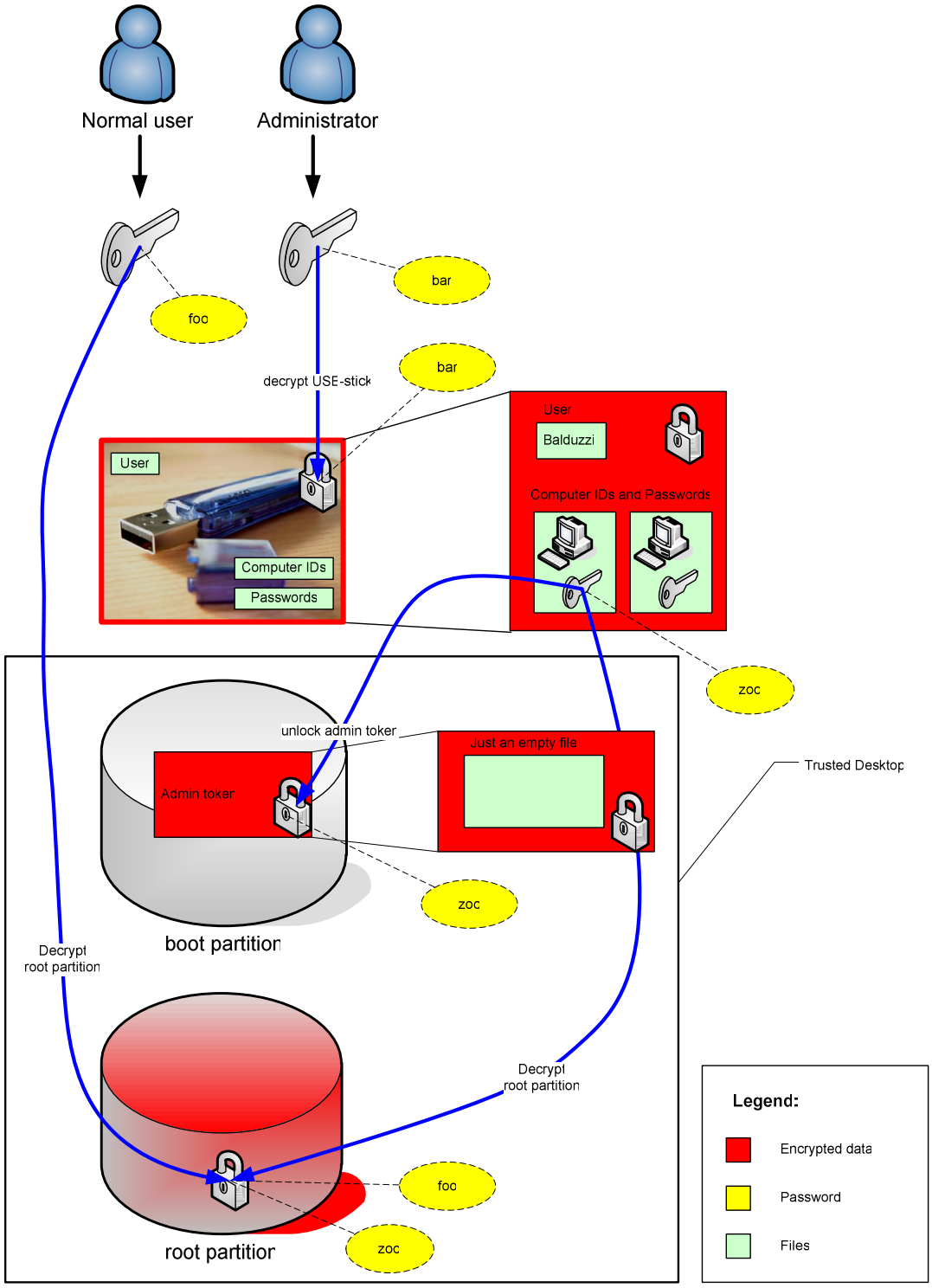


Figure 30 - Disk encryption - design

When the computer powers on, the boot manager loads kernel and ram disk[1] from the boot partition; this is why both are stored unencrypted. The *initrd* contains kernel drivers[2] to handle disk-encryption and the pre-boot authentication procedure that is run[3]. A password is prompted for disk decryption and user authentication. Later on, the system tries to detect connected USB-sticks, querying the USB kernel framework and presenting two different use cases.

If no USB is found, the standard *password authentication* is followed. The admin token is queried with the password to understand if user is privileged. In that case, the system hangs and an error is returned (remember that administrator must login only via USB-token for security reasons). Otherwise, the disk is decrypted and system is loaded in *user environment*.

When USB-stick is connected, the *token authentication* is used. The same password is used to decrypt the stick and when no error occurs, the computer-ID is looked up and verified against the real installation. The associated password is read and used for disk decryption and admin token unlocking, in the same way as in the *password authentication*. However here, only administrator login is granted and an error is raised when the stick belongs to an unprivileged user.

---

[1] The ram disk is a file that, when loaded, behaves as virtual disk, located in the active computer memory (RAM). Linux uses a initial ram disk, termed *initrd*, to preload a set of kernel modules and execute special scripts at pre-boot time.

[2] Technically drivers stand for modules.

[3] Appendix A contains an example used in the test implementation.

Once the administrator is authenticated, the system offers the downgrading of privileges in order to simulate the standard user boot. This feature is very important for debugging. When the system starts in admin environment, any graphical interface is replaced with debugging text and, instead of running the Windows system, a *root* shell is opened.

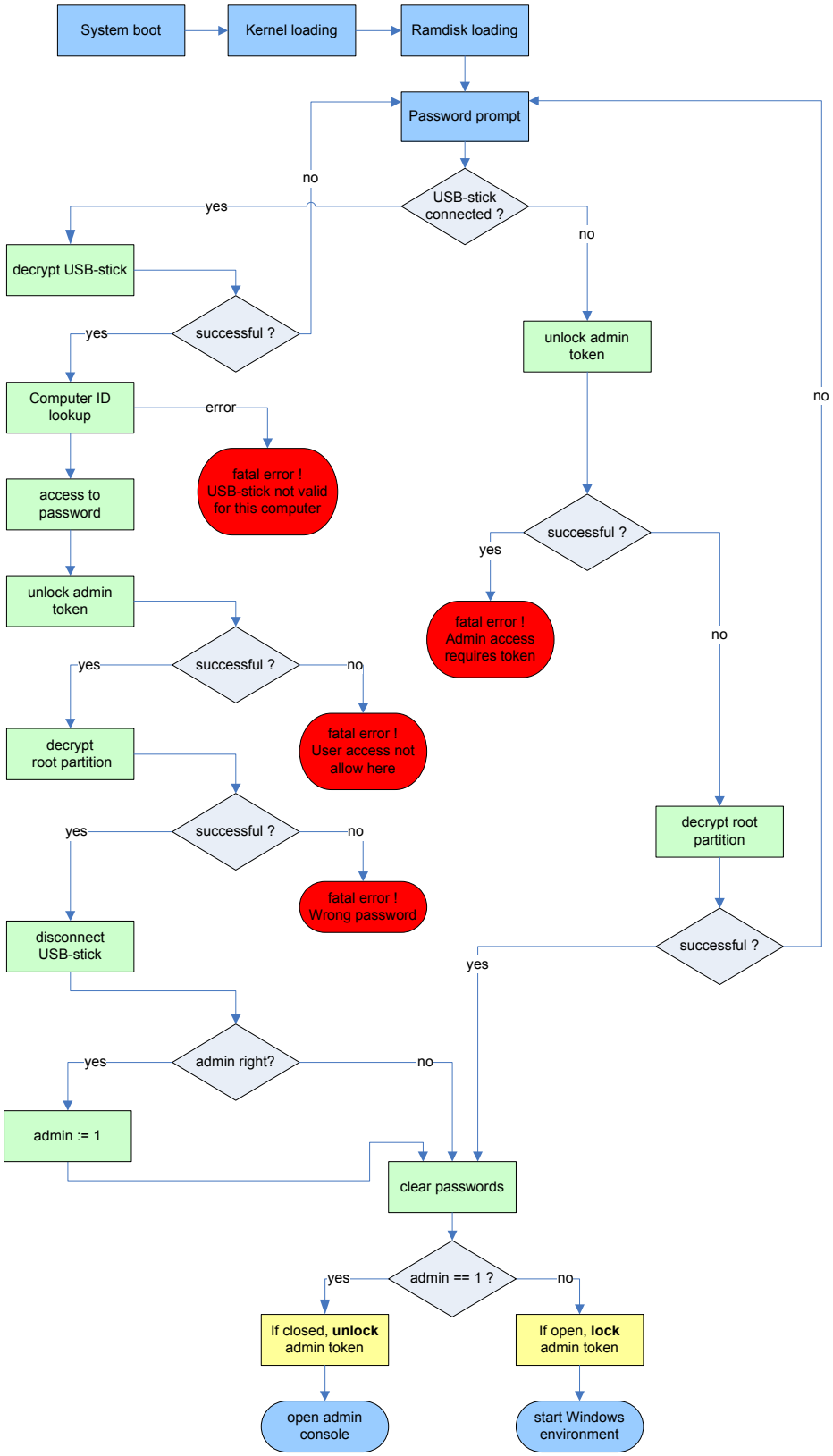


Figure 31 - Disk encryption - architecture

7.2. Implementation

The test implementation has been done with LUKS[1], being the upcoming GNU/Linux standard in disk encryption and providing good documentation, facilitating initial installation and future system maintenance. At the same time, LUKS provides secure encryption ciphers and a trusted management of multiple key passwords. Up to eight keys can be configured, as well as different users.

AES in CBC-ESSIV mode has been adopted as block cipher, being approved by the US government's National Institute of Standards and Technology and considered highly secure by the cryptography community. AES is the successor of DES, it is effective standard since 2002, and it is now the most popular algorithm used in symmetric key cryptography and particularly appreciated for disk encryption applications. The ESSIV mode protects against watermark attacks as previously discussed.

Disk encryption is enabled formatting the entire disk with the command:

```
echo <password> |
cryptsetup --cipher aes-cbc-essiv:sha256
--verify-passphrase luksFormat --batch-mode <disc>
```

The *aes-cbc-essiv:sha256* flag specifies AES in CBC-ESSIV mode as cipher algorithm, in conjunction with SHA256 for digest functionalities.

LUKS is natively supported by Linux kernel, when compiled with these options:

[1] Linux Unified Key Setup, http://luks.endorphin.org/

```
Code maturity level options --->
      [*] Prompt for development code/drivers
Device Drivers -->
  Multi-device support (RAID and LVM)  --->
      [*] Multiple devices driver support (RAID and LVM)
      <*> Device mapper support
      <*> Crypt target support
  Cryptographic Options -->
      <*> AES cipher algorithms (i586)
      <*> SHA256 digest algorithm
```

The last step is to instruct the ram-disk with the pre-boot authentication process. In this way, the computer starts presenting a login mask, where to insert the credentials for decrypting the hard disk.

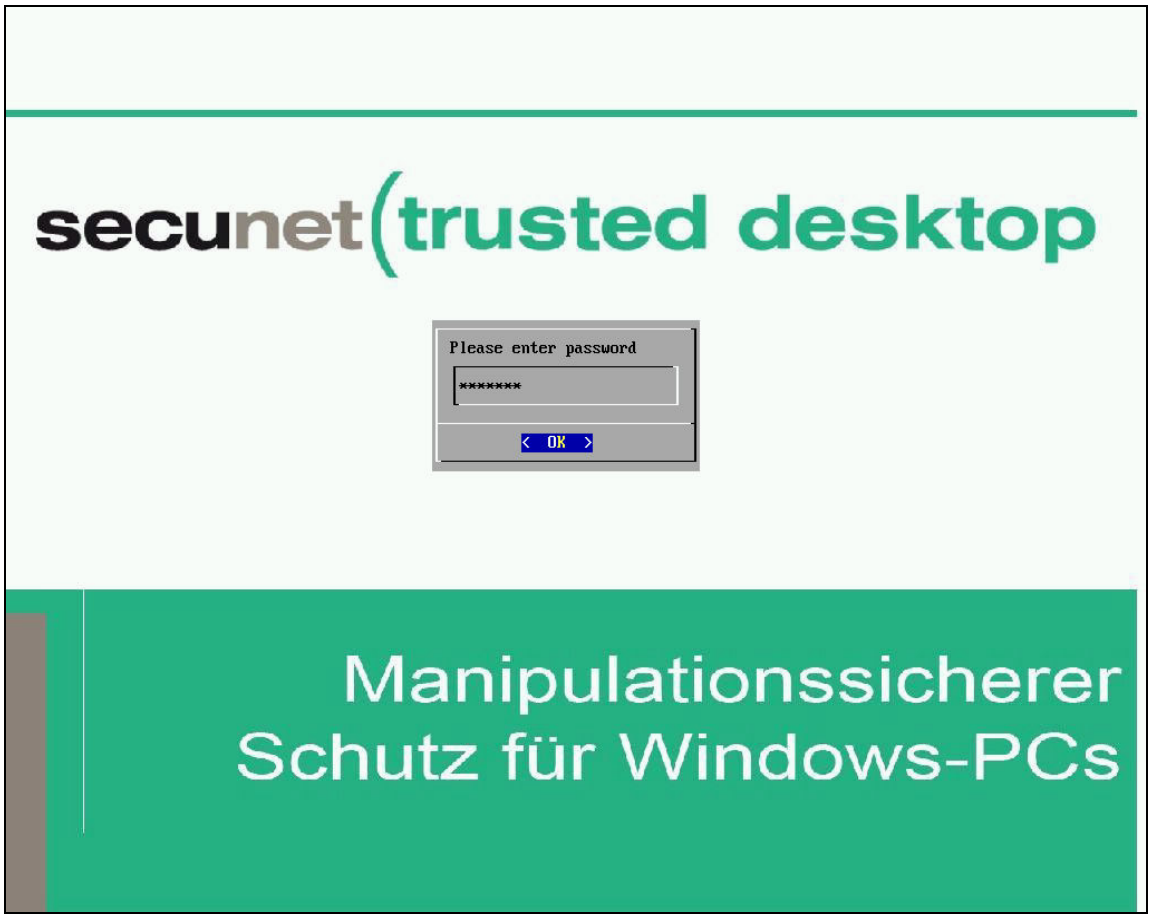


Figure 32 - Disk encryption – screen-shot

The interface is completely loocked and secured, so that the interaction is with the input box. Standard GNU/Linux shortcuts are disabled via boot loader and appropriate kernel patching. There is no way to interrupt or modify the correct boot process.

Debugging information, generated by the encryption layer and the init scripts, is hidden graphically through a "boot splash" software. This solution shows a set of animated icons that are dynamically activated during the boot process, as a progress bar, when decryption is done via USB-stick for example. If the pre-boot authentication validates the user as local administrator, this graphical interface is immediately substituted with a debugging terminal and the guest OS in not loaded.

Figure 33 - User-friendly boot process

A generalization to the problem

Instead of using the *admin token* residing on boot partition as "test" for administrator authentication, a central database is queried to retrieve the group the user belongs to. This database could reside remotely and be queried via some sort of directory access protocols like LDAP. This extension requires both password and username inputted (except when username is read from the stick), but it is more flexible and generic.

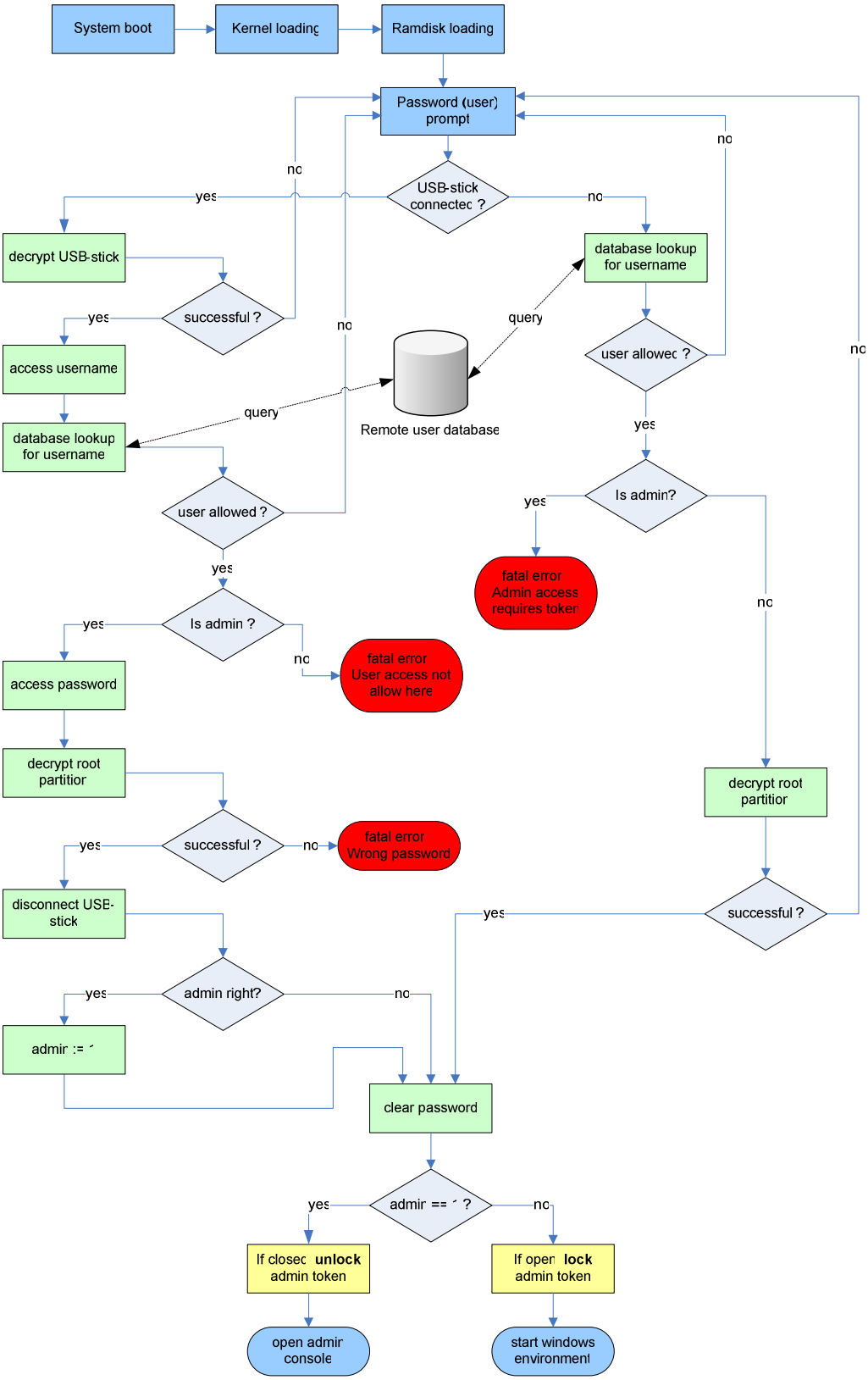


Figure 34 - Disk encryption – future architecture

8. Conclusions

A new approach to virtualization for secure personal computers has been presented. Virtualization, which traditionally was used for server consolidation purposes, has been distributed to each single computer. Thin clients that were used to access the mainframe's resources now have been equipped with two operating systems, separated by a virtualization layer. By virtualization, the security of disparate systems is homogeneously managed.

The security services have been detached from the user system in order to be encapsulated in a hidden *security shell*. Protecting these services from the user environment has shown many advantages. First, the security services cannot be manipulated and the system's tamper resistance is ensured. Second, the security policies defined on the central site can be locally enforced.

A novel antivirus has been designed to exploit this new application of virtualization. While conventional personal antivirus could be switched off and avoided by evil codes and skilled users, now the virus protection has been embedded into the security system, realizing a threat resistant protection for the user environment.

The whole user system, handled by the antivirus as a single virtual-image file, can be efficiently virus scanned. The 3rd generation of the NTFS GNU/Linux drivers offers secure writing operations on NTFS file-systems. The antivirus could be integrated in the computer boot to assure system cleanness, and in the shutdown to avoid time wasting. In fact, a mandatory shutdown of the virtual machine prevents inconsistent states.

During the scan activity, which takes on a standard-configured workspace about ten minutes, the disk image could become easily inconsistent with the real content, due to the possible modifications carried out by the user environment. On the contrary, the antivirus "reparations" to the disk's content could damage the state of the user system.

It has been proved how the relevant *on-access scan* component analyzes the content of the user's files as soon as they are addressed by the virtual machine. A special driver integrated into the virtual machine has been designed to intercept the user's file-system operations and to build a logical representation of the file structure. With the read access from the user system to the disk's file-system, the involved data blocks are looked up or inserted in a cache and examined by the antivirus engine.

The big challenge has been to develop an efficient, fast, and reliable cache that does not behave as a "bottleneck" for the user-system and does not kill its usability.

The file structure has been accurately studied in order to design a fast and a no memory-waste model of cache. A self-balancing BST (AVL-tree) has been adopted for its capacity to reorganize the cache's information, to be quickly addressable. Standard operations are about 70% quicker than conventional trees, when handling non-random data. Real-world scenarios as file-system caches realize this non-uniform statistic because sectors are accessed often sequentially and repeatedly.

The caching algorithm speeds up the antivirus of about two times, scanning only those files that have been previously modified, hypothetically by a virus. By marking the file with a "flag", it is possible to understand which data could be infected. Each file is

marked when being created or modified and unmarked after being scanned. Therefore, only marked files are scanned at read access, while unmarked files and writing operation are discarded.

A filter could be inserted to restrict the scan to those particular classes of files that are targets for viruses like binaries and word macros, skipping metadata and large files (e.g. *plug&play* archive). Since the on-access scan is a synchronous process and interrupts the user system while executing, a large amount of data to scan could freeze the computer for several seconds. It can be reasonable to use the "dirty flag" to mark a set of blocks instead of the whole file, so that just a part gets scanned (e.g. the swap file with size around the gigabyte)

Solutions as enabling the on-access scan only for removable mass-storage devices, by means of a tight collaboration with the VMM's device control, are too risky. Viruses could spread to the user system in alternative forms and channels, avoiding both, the network protection and the curbed on-access scan (e.g. stenographic exploits and encrypted viruses coming over the network).

The antivirus impact on performance is remarkable when a large amount of data is read. When the user system starts, for example, the antivirus slows down the computer of several times due to cache building. An interesting idea has been to preload the cache content with structures of files read at boot time, such as operating system's drivers and libraries, empirically chosen.

Finally, the NTFS architecture is still partially obfuscated: neither the source codes nor adequate technical information is provided. The difficulty of debugging strange file-

system behaviours has generated obvious problems in the code implementation. The performance is quite bad, because they are damaged by periodical cache rebuilds. These are needed to prevent inconsistent states between disk- and cache- contents, whenever an unknown writing operation occurs on the file index table.

A sort of transparent proxy has been designed to inspect viruses and malware coming over TCP/IP connections, such as emails and web downloads, before they reach the user environment. This solution has shown the benefits of scanning inconspicuously any user's network stream from the hidden sub-layer. This is realized independently from the physical interface used by the stream to carry out such information.
This is valid also for encrypted tunnels (VPNs) and SSL-enabled protocols. VPNs could be established by the *security shell* and their content bridged as plain text to the user system, after being inspected (and the other way round); while the man-in-the-middle attack could be exploited to intercept the SSL-protocols handshaking and to real-time decrypt the connection's content. For completeness, the removable network devices get virus protected too, by an appropriate agent that controls their use. Special policies define which devices are accepted. Their configuration is realized by the security shell.

Further existing security technologies have found great benefits in this virtualization paradigm. The thesis has considered the *disk encryption* issue as one of the technologies worth of investigating. A hidden encryption layer has been embedded to protect at the same times both security shell and virtual system from confidentiality attacks. The user environment is therefore encrypted "from the outside".

By coupling encryption and virtualization technologies, two evident benefits arise: encryption is realized automatically and not visible from the user-system in which no configuration or encryption support is required; the disk's keys are managed homogeneously at the central site, independently from the virtual system installed on the single computers.

Appendix A. Sample code

Interface between the Clamavis antivirus engine *(libclamAV)* and the on-access scan service

```
// Initialize the antivirus engine
int td_init_avscan(void)
{
   int ret = -1;

   // init globals
   virusdb = NULL;
   bzero (&claminfo, sizeof (struct s_info));
   strcpy (dbdir, "/var/lib/clamav");

   // load virus database
   if((ret = cl_loaddbdir(dbdir, &virusdb, &claminfo.sigs)))
     {
            printf("@%s\n", cl_strerror(ret));
            free(dbdir);
            return ret;
     }

   if(!virusdb)
     {
            printf("@Can't initialize the virus database\n");
            return ret;
     }

   if((ret = cl_build(virusdb)) != 0)
     {
            printf("@Initialization error: %s\n",  cl_strerror(ret));
            return ret;
     }

   // open virus scan
   char buf[4096];
   snprintf(buf, 4096, "/secunet/AVSCAN.log");
   AVfd = open(buf, O_CREAT | O_RDWR | O_TRUNC, 0644);

   return 0; // ok
}

// This is the real interface to the antivirus engine
int td_avscan (struct sector_list *pSectorList, int BDRVfd,
               char **virname)
{
   // reverse engineering on eicar.com
   short otfrec = 1;
   unsigned short ftype = 500;

   //const char **virname;
```

```
   int rc;

   // reset status pointer
   pStatus.element=0;
   pStatus.offset=0;

   // scan
   rc = _td_scanlist (pSectorList, BDRVfd, virname,
                      &claminfo.blocks, virusdb, otfrec, ftype);

   // check output
   if (rc == CL_VIRUS)
     {
           dprintf (AVfd, "* td_avscan(): %s - VIRUS FOUND: %s\n",
                    pSectorList->m_p_filename, *virname);
     }
   else
     {
           dprintf (AVfd, "td_avscan(): %s - virus not found\n",
                    pSectorList->m_p_filename);
     }

   return rc;
}

// It retrieves the file's content for the antivirus engine
int _td_readn (struct sector_list *pSectorList, int BDRVfd,
               void *buff, unsigned int count)
{
   unsigned int i        = 0;
   unsigned int todo     = count;
   unsigned int done     = 0;
   unsigned int ret;
   unsigned long long base_sector_i, nb_sectors_i;

   /* count file size */
   unsigned int c;
   unsigned int tsize = 0;

   // exclude file bigger of 10Mbyte (20.000 sectors)
   for (c=0 ; c<pSectorList->m_sector_list_size; c++)
      tsize+=
      pSectorList->m_p_sector_list_entries[c].m_last_sector-
      pSectorList->m_p_sector_list_entries[c].m_first_sector;

   if (tsize>20000)
      {
           dprintf (AVfd, "* skipping %s (%u MB)\n",
                    pSectorList->m_p_filename, tsize*512/1000000);
           return 0;
      }

   while (todo>0 && (pStatus.element<pSectorList
                    ->m_sector_list_size))
      {
       base_sector_i=
```

```
    pSectorList->m_p_sector_list_entries[pStatus.element].
    m_first_sector * 512;
nb_sectors_i = (
    pSectorList->m_p_sector_list_entries[pStatus.element].
    m_last_sector
    -
    pSectorList->m_p_sector_list_entries[pStatus.element].
    m_first_sector
               ) * 512;

// seek to current position
lseek (BDRVfd, base_sector_i+pStatus.offset, SEEK_SET);

if (nb_sectors_i-pStatus.offset > todo)
// current element is bigger of what we have to read
  {
     if ((ret = read (BDRVfd, buff+done, todo))<0)
       {
          dprintf (2, "fatal error while reading sector
          content %llu for AV\n", base_sector_i+pStatus.offset);
          return -1;
       }

     // update I have to still read and I have read so far
     todo-=ret;
     done+=ret;

     // increment offset pointer
     pStatus.offset+=ret;
  }

//left sectors on i-element are not enough
else
  {

     if ((ret = read (BDRVfd, buff+done,
                      nb_sectors_i-pStatus.offset))<0)
     {
          dprintf (2, "fatal error while reading sector
          content %llu for AV\n", base_sector_i+pStatus.offset);
          return -1;
     }

     // update I have to still read and I have read so far
     todo-=ret;
     done+=ret;

     // set status pointer to next element
     i++;

    // go to next element
     pStatus.element++;
     pStatus.offset=0;
  }
}
```

```
   return done;
}
```

The pre-boot authentication used in the test implementation for disk encryption

```
#!/bin/sh
[cut]

/bin/loadkeys /etc/console/boottime.kmap.gz
ADMIN=0                 # Admin boolean FLAG

while [ 1 ]
do
      PASS=`/bin/dialog --insecure --nocancel --passwordbox
           "Please enter password" 0 0 3>&1 1>&2 2>&3`
      /bin/clear

      # check if we have a usb-pen.
      # yes -> decrypt it
      # else -> decrypt directly hard-disk
      USB=`/bin/dmesg|/bin/egrep -e
      "(sd.*removable|removable.*sd)"|/bin/rev|/bin/cut -f1 -d "
      "|/bin/rev|/bin/tail -1`

      if [ "$USB" = "" ]; then
      echo "USB Device NOT found. Going through HD decryption."

      # We don't want administrator to log-in WITHOUT USB
      /bin/mount -t ext3 /dev/sda7 /mnt # mount boot-partition

      # assign loop-device to admincontainer
      /sbin/losetup /dev/loop0 /mnt/admincontainer

      echo $PASS|
      /sbin/cryptsetup luksOpen /dev/loop0 admincontainer -T 1

      if [ $? = 0 ]; then     # admin
            echo "DEBUG (TO DELETE AND GIVE GENERIC ERROR):
                  Admin access is permitted only via USB"

            /sbin/cryptsetup luksClose admincontainer
            /sbin/losetup -d /dev/loop0
            /bin/umount /mnt
            continue

      else            # user
        # unlock /
            echo $PASS|/sbin/cryptsetup luksOpen /dev/sda3 cryptoroot
            if [ $? = 0 ]; then
                  break      # password correct -> we have finished
            else
                  continue   # password wrong -> ask another passwd
            fi
      fi
```

```
else
      USB=$USB"1"       # concat 1 to hd? /sd?
      echo "USB device FOUND: $USB"

      # unlock the usbstick
      echo $PASS|/sbin/cryptsetup luksOpen /dev/$USB cryptoUsb

      if [ $? != 0 ]; then     # the USB password is WRONG
                  continue    # ask another passwd
      fi

      # mount usbstick
      /bin/mount -t ext3 -o ro /dev/mapper/cryptoUsb /mnt

      USER=`/bin/cat /mnt/username`
      if [ $USER = "admin" ]; then
            echo "Hello Administrator :-)"

      # now we have only one system (easier test procedure)
      PASS=`/bin/cat /mnt/passwd|/bin/cut -d : -f 2`

      # unlock /
      echo $PASS|/sbin/cryptsetup luksOpen /dev/sda3 cryptoroot

      if [ $? != 0 ]; then     # wrong
            echo
      echo "ERROR: system password mismatch or user not allowed."
            echo "--- SHUTTING DOWN ---"
            echo
            /bin/sleep 5
            /sbin/halt -f -p

      else    # passwd correct
            # close the usb key
            /bin/umount -f /mnt
            /sbin/cryptsetup luksClose cryptoUsb

            # ask for downgrading to normal user
            dialog --defaultno --no-shadow --yesno
                  "Hello Admin!  Put usb in a secure place.
                  Satisfied with User-Rights?"
                  0 0 3>&1 1>&2 2>&3

            if [ $? = 0 ]; then     # yes
                  /bin/clear
                  echo "Ok! Your rights will be downgraded to
                        User-Rights for this session."

            else                  # Admin wants to be Admin
                  /bin/mount -t ext3 /dev/sda7 /mnt
                  /sbin/losetup /dev/loop0 /mnt/admincontainer
                  # assign loop-device to admincontainer

                  echo $PASS|/sbin/cryptsetup
                  luksOpen /dev/loop0 admincontainer -T 1
```

```
                        if [ $? != 0 ]; then   # just for debugging
                              echo
                        echo "ERROR: can't unlock admincontainer!"
                              echo "--- SHUTTING DOWN ---"
                              echo
                              /bin/sleep 5
                              /sbin/halt -f -p
                        fi

                        ADMIN=1
                  fi

                  # we have finished
                  Break
            fi
      fi
fi

done

PASS= # overwrite the password string

if [ $ADMIN -eq 1 ]; then
      echo 0 > /proc/splash
      /bin/clear
      echo -e "\033[01;32m*** Booting in Admin mode ***\033[01;37m"
fi
```

Appendix B. Index of figures and tables

Figures

Tables

Appendix C. References

1. D. Chu, Senior Director of VMware, *What is Virtualization?,* `http://news.zdnet.com/2036-2_22-6058678.html`
2. S. Potter, J. Nieh, D. Subhraveti, *Secure Isolation and Migration of Untrusted Legacy Applications,* Columbia University Technical Report CUCS-005-04, January 2004
3. N. kiyanclar, *A Survey of Virtualization Techniques. Focusing on Secure On-Demand Cluster Computing,* ACM CORR. Technical Report cs.OS/0511010, May 17 2006
4. P. Ferrie, *Attacks on Virtual Machine Emulators*, Symantec Advanced Threat Research
5. O. Saarinen, *Encrypted Watermarks and Linux Laptop Security,* Proc. The 5th International Workshop on Information Security Applications (WISA2004), Jeju Island, Republic of Korea, August 23-25, 2004
6. C. Fruhwirth, *New Methods in Hard Disk Encryption*, Institute for Computer Languages Theory and Logic Group. Vienna University of Technology
7. J. Etienne, *Vulnerability in encrypted loop device for Linux*, 2002
8. C. Kent, *Draft Proposal for Tweakable Narrow-block Encryption*, IEEE P1619 draft, October 19, 2004
9. N. Weaver, V. Paxon, S. Staniford, R. Cunningham, *A Taxonomy of Computer Worms,* Proceedings of the ACM CCS First Workshop on Rapid Malcode (WORM), October 2003
10. T. Kojm, *libclamAV*, `http://www.clamav.net/doc/`

11. S. Gordon, D. Chess, *Attitude Adjustment: Trojans and Malware on the Internet: An Update,* 1999
12. M. Bishop, *An Overview of Computer Viruses in a Research Environment***,** 1992
13. Microsoft TechNet, *How NTFS Works*, March 2003
14. N. Rajeev, *Windows NT File System Internals: A Developer's Guide (1st ed.)*, O'Reilly, ISBN 1-56592-249-2, (1997).
15. D. P. Bovet, M. Cesati, *Understanding the Linux Kernel (3rd ed.),* O'Reilly, ISBN 0-59600-565-2, (2005)
16. R. Russon, Y. Fledel, *Linux-NTFS Technical Development Documentation (V. 0.5.8)*, `http://www.linux-ntfs.org/content/view/104/43/`
17. D.D. Sleator and R.E. Tarjan. *Self-Adjusting Binary Search Trees*. Journal of the ACM 32:3, pages 652-686, 1985.
18. B. Pfaff, *Performance Analysis of BSTs in System Software.* SIGMETRICS/Performance poster, June 2004.
19. J.-L. Baer and B. Schwab, *A comparison of tree-balancing algorithms*. Communications of the ACM, vol. 20, no. 5, pp. 322–330, 1977.
20. W. E. Wright, *An empirical evaluation of algorithms for dynamically maintaining binary search trees* in Proceedings of the ACM 1980 annual conference, pp. 505–515, 1980.
21. E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*. Computer Science Press, 1995. ISBN 0-7167-8292-8
22. B. Pfaff, *GNU libavl*, `http://www.stanford.edu/~blp/avl/libavl.html/`

23. A. Ornaghi, M. Valleri, *Man-in-the-Middle: cos'è, come ottenerlo, come prevenirlo, come sfruttarlo,* Italian Black Hats Association, September 2002