TALES FROM CLOUD NINE

Mihai Chiriac
BitDefender, 24 Preciziei Blvd, West Gate Park, Building H2, Bucharest, Romania

Email mchiriac@bitdefender.com

ABSTRACT

At last year's VB conference we promised to answer a set of questions concerning the performance of cloud-based anti-virus software. The feedback was overwhelming, both from fellow researchers and large corporations, particularly ISPs. No wonder, since the number of viruses grows at an exponential rate. Being able to provide instant protection and enhanced detection rates at a (possibly) lower bandwidth cost proved to be a winning combination.

In the first part of this paper we will describe, in detail, our cloud-based anti-virus engine, including a set of statistics, optimization opportunities that were revealed only after performing a few hundred thousand scans, comparisons with current technologies, etc. We will talk about the benefits and drawbacks of keeping at least part of the virus signature database and scanning logic on our servers and, more interestingly, about the instances when cloud-based scanning is clearly more efficient than traditional approaches.

The second part of the presentation will cover a new client-server technology, called 'IMD' (Intelligent Malware Detection). The client side of IMD runs on the client and is responsible for gathering 'IMD flags', while the server side is responsible for collecting the flags, applying rules and ultimately deciding whether a file is suspicious or not. We will also describe some cases when the server has enough information to blacklist files automatically, thus reaching the holy grail: instant detection.

INTRODUCTION

Ten to fifteen years ago it wasn't uncommon to evaluate the quality of an anti-virus program based on its reported number of signatures – but things have changed. Virus numbers grew at a steady pace until around 2005, when most products passed the 100,000 mark. From that point on, however, virus numbers started growing at an exponential rate (see Figure 1), mostly because both the profile and the motivation of the malware writer had changed dramatically. From computer programmers who wrote tiny gems of assembly code, mainly for fun and recognition in the underground world, we now deal with professional networks of hackers that are purely financially driven. These hackers continuously generate new malware versions in order to evade detection and have started employing techniques like server-side polymorphism (same malware, modified on the server so that each user gets a slightly modified version) to evade signature-based detection.

Obviously, the best way to fight this type of malware is generic detection, either by behaviour (studied in a virtual environment or on the real machine, using host-based intrusion prevention software) or by a dedicated detection routine based on the characteristics of that specific malware family. But signatures are still very important, because they have many advantages. First, signature-based detection is usually faster than other types of detection; second, exact variant identification is very important for support and remediation purposes; third, to reduce to a minimum the time frame between a new piece of malware first appearing and the development of a generic detection routine.

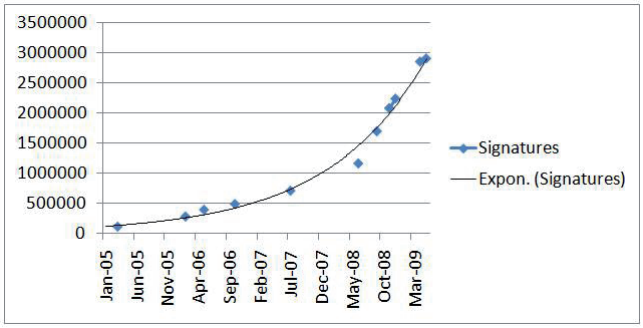

Figure 1: Virus signatures, BitDefender Antivirus, 2005–2009.

Depending on the relationship between signature detection and non-signature detection, most anti-virus companies display a number of signatures of between two and three million – but products displaying more than five million signatures are not uncommon (April 2009). This large number of signatures poses many problems for anti-virus companies, mainly because it is becoming increasingly difficult to deliver all the virus updates to the customers. Recent estimates are that more than 10,000 new malware samples appear each day. Indeed, *BitDefender*'s anti-virus labs are adding, on average, 4,700 new signatures per day (April 2009), trusting the remaining samples to generic detection routines and other proactive detection technologies. This leads to considerable resource consumption, both on the client (where memory usage of 50–80 MB is not uncommon) and also on the bandwidth required to deliver definition updates to the customers.

CLOUD SCANNING

Historically, anti-virus programs kept the entire set of scanning engines and signature databases on the client's computer. In-the-cloud scanning is a technology that moves at least parts of the malware signatures and scanning logic onto servers, thus attempting to relieve some of the burden from the client. Three major approaches have appeared in the past years. The CloudAV academic project [1] involved uploading entire files for server-side scanning. Interestingly, the server side uses more than one detection engine, an approach with advantages (better detection rates) but also disadvantages (larger operating cost, higher risk of false positives, etc.). Since entire files have to be uploaded for server-side scanning, this approach has huge problems regarding bandwidth consumption, but it may be successfully implemented in local area networks. CloudAV only uploads for scanning executable files (although it may be configured to scan files of any type) and uses complex local and remote caching systems.

Advantages:

- Very thin client, this means very low resource consumption on the client and fewer vulnerabilities.
- Possibly better detection rates, due to the use of more than one anti-virus engine.

Disadvantages:

- Bandwidth problems render the product unusable for

general purposes anywhere in the near future; however, the product can successfully be used in local networks.

- Privacy issues, especially when scanning non-executable file types like documents and scripts; however, omitting these files from scanning poses a serious security breach.
- Local caching may pose problems with virus updates; moreover, a secure caching mechanism involves performing a secure hashing algorithm over the entire file, which is usually slower than actually scanning the file locally.
- Using more than one anti-virus engine means more problems with false positives, although the product may be configured to issue a warning only when a specified number of AV engines report an infection.

Another approach comes from the industry and it involves computing a static signature for every malicious file and reporting the static signature (a cryptographic hash over the file) to a server. The server's responsibility is to manage this huge 'blacklist' of hashes. In this case the client can be reduced to a simple program that computes a cryptographic hash over a file and queries a server (in some implementations, a DNS query).

The technique is very effective, because the vast majority of common malware is static and can be uniquely identified with checksums over fixed areas (fixed offsets/sizes). However, this approach has problems when dealing with non-static malware, because in some cases it creates static signatures for non-static malware, thus leading to an unjustified increase of the blacklist. More importantly, for example with file infectors, some of the products create static signatures for the samples 'seen' by the scanner (either scanned by the full version of the scanner, or detected using heuristic methods), leading to a false sense of security. (The samples from virus collections are the first to receive static signatures; these collections are exchanged by anti-virus researchers, but also by anti-virus testers and reviewers.) There is the possibility that a 'new' file (identified by a new cryptographic hash) infected with an older, non-static piece of malware (for example, an older file infector), will evade detection. We believe that in order for a product to achieve reliable detection of non-static malware, it needs either to integrate a local detection component or to allow uploading of at least parts of the scanned file to the server.

The third approach uses the cloud's ability to gather a large quantity of information (intelligence) and process this information for product enhancement: faster reaction to virus outbreaks, better detection rates and faster mitigation of problems like false positives.

A MIDDLE-OF-THE-ROAD SOLUTION

It is possible to combine the cloud's capability to store and process multiple gigabytes of data while also keeping any CPU-intensive task on the client. A modern anti-virus engine contains not only malware detection engines, but also decompressors (RAR, ZIP, etc.), parsers for different file formats used by email clients, etc. It is obvious that any compressed file should be uncompressed on the client's machine, since decompressing is a CPU-intensive task. In *BitDefender*'s case, 98.9% (size of code + data) are malware detection engines, so it's very efficient to keep all the non-detection engines on the client.

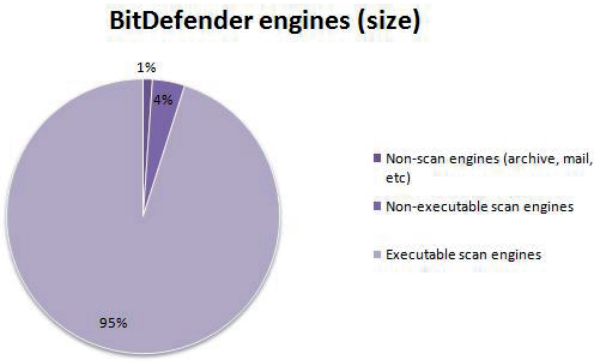

Figure 2: 95% of all BitDefender's engines' code and data detect executable malware.

Even detection engines can be divided into many categories: detection of executable malware, detection of macro viruses, detection of script viruses (JavaScript, Visual Basic Script, Batch files but also Perl, Python, etc.) and so on. Judging by the file sizes (detection engine plus virus database), around 95% are used for detecting executable malware (see Figure 2). Judging by the number of signatures, 97% are signatures for executable malware and only 3% of signatures are for non-executable malware. Since privacy is one of the most important factors in cloud scanning, we believe that it is safer (and more efficient) to keep the detection engines for non-executable malware on the client.

There are many techniques used for the detection of executable malware. Historically, viruses were identified with 'scan strings' that were searched in various areas of the file. Even if the technique is deprecated, there are still some cases where it is used (mainly to detect some specific cases of corrupted viruses, etc.). Even today, some anti-virus engines use this technique, usually an implementation of the Aho-Corasick algorithm [2]. However, this method is CPU-intensive and works better on the client. For viruses encrypted with a rather trivial encryption algorithm the

Engine type	Size (%)	Comments
String search	0.1%	Used rarely, for corrupted viruses, strange/uncommon file formats. Should run on the client.
X-ray	1.6%	CPU-intensive; should run on the client.
Algorithmic detection	1.8%	Generic detection routines; should run on the client.
Heuristics (sandbox)	3.8%	CPU-intensive; should run on the client.
Flow scan	1.8%	Scans emulator pages as they are accessed/modified; should run on the client.
Static detection	90.9%	The vast majority of signatures; can work well in a client-server architecture.
	100%	

Figure 3: BitDefender, executable detection by engine types, March 2009.

anti-virus engine applies cryptanalysis (also known as x-raying) which is technically a known-plaintext attack over the encrypted virus body. This technique is CPU-intensive as well, and works much better on the client. Code flow signatures are normally scanned for when the code emulator discovers new code areas or when a specific decryption/decompression routine has successfully been executed. This technique, as well as all generic detection techniques (sandbox, etc.) works much better on the client.

We see (Figure 3) that static signatures represent the vast majority of virus samples. Static signatures involve checksumming an area or a set of areas of a file, and looking up the result in a list of 'known bad checksums'. It is trivial to modify an existing anti-virus engine to use all the complex and CPU-intensive detection methods on the client, while keeping all the bulky static signatures on servers.

Usually, the procedure for scanning for static signatures is as follows:

- Parse the file format; choose *N* important zones, depending on the format.
- For each of these *N* important zones compute a checksum (C1…Cn).
- Look up these checksums in a table of checksums. If any of the checksums is present in the table then we obtain, from the table, more information about the pieces of malware that match that checksum (usually a structure containing a second, typically larger area to be checksummed, information about the virus name, disinfection information, etc.).
- The engine computes the checksum for the second area and checks for matches.

It is very important to mention that a checksum hit (e.g. the specific checksum found in a table) triggers a series of I/O requests (reading and checksumming the second block). Anti-virus programs have in place a series of automated tests that check for checksum hits on common files. Should a checksum for a malicious file match the checksum of a commonly used clean file, the signature is simply changed to include a different file area.

Example: Let's assume that an implementation of static detection uses signatures containing values such as: checksum, signature/block type, disinfection information, malware name information, offset of the second block, size of the second block, checksum of the second block. Since all values are 32-bit, this means that a signature has a size of at least 28 bytes.

It is very possible to keep the table of checksums on the client. Possible checksum values range from 0 to 2^32–1, but, in the case of *BitDefender* (March 2009), there are only 1,498,252 unique checksums. Assuming an even distribution of the checksum function there is a 0.0348% chance for a checksum to be present in the table. For *N*=12, there is a 1–(1–0.0348)^12, or 0.34% chance that at least one of the checksums is present in the table. In practice, on a normal computer with an average of 200,000 files, an initial checksum of 12 file areas will only trigger additional I/O on an average of 800 files (i.e. at least one of the checksums was found in the table) (see Figure 4).

In this particular case we reduce memory consumption to only 14.2% (1/7) and the cases when we find a checksum on a normal computer are extremely rare. A possible implementation would only perform database updates on the table of checksums (thus theoretically reducing the database updates to only 14.2% of the original). In the case of a checksum match the client computer has to query the server for all the signatures matching that checksum. The problem of disconnected operation can be solved easily, since the engine may store, locally, the signatures for all the checksums that matched a signature on the user's computer and for all in-the-wild pieces of malware. In this case we will only encounter problems when a checksum of a newly arrived file (probably via a USB stick or CD-ROM) is found in the checksum table. Since this rarely happens, the product can easily offer a list of alternatives to the user.

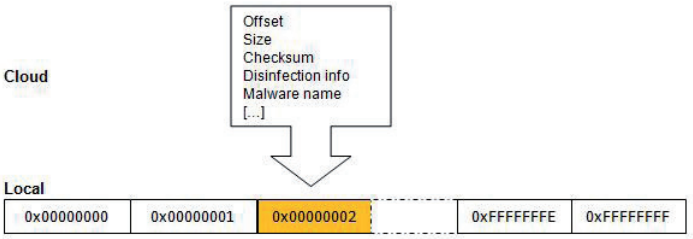

Figure 4: Model of a possible partial cloud-scanning technique.

One could argue that if virus numbers keep growing the table will become increasingly populated, leading to matches that require file and network I/O and, in our case, cause network latency. However, it's very easy to add more information to the local component. If we add a further eight bits we reduce the hit rate by a factor of 256, while only increasing the local component from 14.2% to 17.85% of the original size.

Keeping the bulk of the static signatures on the server has many advantages: lower memory consumption for the user, lower bandwidth requirements for downloading updates, etc. A very interesting advantage comes from the question/answer model on which the scan is based: the malware database becomes a dynamic entity, directly connected to the anti-virus labs. It is possible for the anti-virus researcher to optimize the database (remove possible 'busy' checksums), determine malware prevalence, etc. However, a major disadvantage of this approach is that since the scan itself is interactive, anti-virus companies cannot use standard content delivery services. Thus content delivery, content mirroring, redundancy etc. has to be handled by mechanisms developed in-house.

A REAL-WORLD IMPLEMENTATION

A possible implementation of the above technology is a very quick scan for active malware. Nowadays the vast majority of malicious software tries to find a way of remaining memory resident and surviving across reboots. That's why if we only scan the sensitive areas of a computer we will detect the vast majority of executable malware. Our implementation currently scans:

- All executable modules loaded inside all processes (on-disk and in-memory contents).
- All kernel modules.
- All system services (regardless of their execution status).
- All system entry points (files referenced by autorun registry keys, files in key folders like 'Startup', etc.).

- All Browser Helper Objects or browser add-ons (even if the browser is not loaded).
- All registry entries related to 'interesting' extensions (e.g. exefile\shell\open\command) and generally sensitive registry keys.
- All modules registered as Winsock Layered Service Providers, Winlogon Notify DLLs, etc.

We collected a series of statistics on the first two months of the beta period. We have observed that the average number of unique 'important' files is 675 for *Windows XP*, while the average number of important files for *Windows Vista* is 755. The average number of important files for all scanned machines was 685. It is very important to mention that we omitted from the statistics all the x64 machines, because in the beta period the product didn't have x64 support, so it only had access to 32-bit processes, leading to a far smaller number of scanned files.

The first step is to make a list of important files to be scanned and create a cryptographic hash to uniquely identify the file. Probably the largest advantage of using cloud scanning is that a server has the ability to keep huge databases of known-clean and known-malicious files. A medium-sized whitelist of around 100 terabytes will help to weed out a large majority of known-clean executable files.

The scanning process works like this:

1. Create the list of cryptographic hashes for all important files.
2. Send the list of cryptographic hashes to the server.
3. The server's reply may be, for each file:
   a) Clean – present in our whitelist of 100% trusted files.
   b) Not detected – the file has previously been seen and scanned, was found to be clean, but it is continuously re-scanned on the server.
   c) Infected – the file has previously been seen by one of the clients and the scanning engine determined that the file is infected.
   d) Unknown – the file is unknown and needs additional processing.
4. Scan using static signatures.
5. For every unknown file, depending on the user's acceptance, the client uploads the file (or specific chunks of the file) onto the server, where it is scanned by the full range of detection engines.

In the beginning of the beta period, each user, when first executing our product, had to upload an average of eight files. This number has decreased continuously, and as this paper was written the average was three files for first-time users (May 2009). We expect this average to decrease further over time (see Figures 5 and 6).

The average file size amongst the first million uploaded files was 220 KB, meaning that, on average, a first-time user uploaded 660 KB of data. It would have taken around 4 MB (more than five times the size) to download the non-static detection engines and the corresponding signature databases. However, it's important to mention that even if uploading unknown files is more efficient in terms if bandwidth, some users might have a much lower upload rate compared to their download rate. On the other hand, the following scans won't

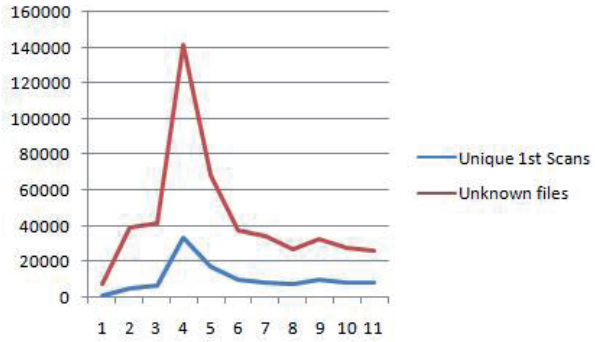

Figure 5: Analysis of unknown files and unique scans, beta period. The peak is April 1st (Conficker activation day).

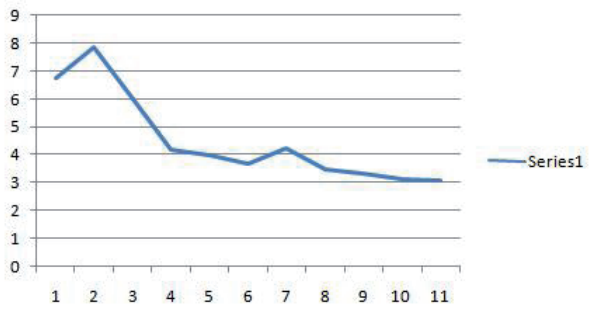

Figure 6: Number of unknown files decreasing over time.

have to upload the files again, as they're already known to the server. This also helps all other users that might have those files on their systems.

We carried out a second set of tests over the first million uploaded files, with a specially tweaked version of the *BitDefender* engines that logged every file chunk read; we discovered that, on average, we only need 18 KB of a file to be uploaded to properly determine whether the file is infected or not. Uploading only the required file chunks (technically this means forwarding file I/O calls to the client) uses less than 8% of the bandwidth needed to upload entire files (an average of 54 KB for a first-time user).

We've decided that for the beta period, we should upload the entire file, mainly because:

- A lot of new detection technologies have been added to the product; these technologies are constantly tweaked, so it's possible that additional data may be needed from the file.
- The re-scans need the entire file because it may be possible that new detection routines would need new file chunks.

It may be possible for a user to run a start-up script. In a worst-case scenario the user might be infected with a script-based malware that runs on startup (e.g. Js.Kak.A@mm). However, it is possible for a user to have a network login script (with username and password) that runs on startup. In terms of privacy it's a big concern to upload these files for server-side scanning. At this point our technology doesn't scan non-executable files, but we believe the best way to scan these files would be to download the specific detection engines and virus signatures on the client computer.

During the beta period we noticed that 20% of our users had at least one piece of malware running on their computer. However, it is possible that the users started the scan

specifically because they felt their computer was infected. One of the most interesting statistics is that 10% of the detected malware was non-static, and their corresponding unique file was seen only once – this means that a static-only scan would not have detected the piece of malware.

We believe our technique is extremely effective for detecting active or potentially active malware. The average first scan time was only 59 seconds, and the average second scan took just 29 seconds, which means the technique may be very useful in circumstances where a very quick system check is needed.

A big disadvantage of our technique is that we need to upload files, or at least chunks of files, in order to achieve the same detection rate as the full product, including for non-static malware. A possible solution to this problem is a heuristic detection engine for dynamic malware based on the executable structure alone. We can assume that the vast majority of executables found on users' computers obey PE standards; a file whose structure looks like the structure of a file produced by a linker, for example, if it were malware, would *probably* have been detected using a static signature. If we only upload for server-side scanning the files that are somewhat non-standard from the point of view of executable structure and/or executable file entropy we would decrease the number of uploaded files while also keeping a fairly high rate of detection. However, we feel that the number of uploaded files is already very low, so at least for now we prefer to upload any unknown file for scanning with the full range of detection engines.

The server side runs the whitelist manager, the blacklist for already-seen pieces of malware and the detection engines for non-static malware. It also runs the databases that collect information about executables (typically distribution, spreading and geography) and the routines that continuously re-scan unknown files. There is a possible privacy issue when the user uploads executable files to the server, so the user may choose not to upload some of the files – thus slightly reducing the detection rate. A possible solution for enterprises would be to keep a specially configured slave server that performs privacy-sensitive tasks and exchanges non-privacy-sensitive data with a master server; this solution would also minimize the external bandwidth used.

INTELLIGENT MALWARE DETECTION (IMD)

The cloud has the ability to collect and manage vast amounts of information. Various whitelisting companies have used the cloud for storing huge databases of known-clean programs. Anti-virus companies have used the cloud for many years, specifically for real-time virus reporting and detection of virus outbreaks. In this scenario, the anti-virus engine monitors the execution of unknown files and reports the event to the server. The server then decides whether the unknown file represents a virus outbreak by analysing details such as file spreading, distribution and a set of geographic details.

IPS companies have used the cloud to report the behaviour of executables, thus extending the detection capabilities of host-based intrusion prevention systems.

The simplest implementation of the client module, which proved to be remarkably effective, consisted of performing a live system analysis for compromise detection. Instead of reporting behaviour events, our engine performs a snapshot of the live system and gathers information. This snapshot lasts no more than a few seconds for a normal system.

Our engine supports three information types for every module:

1. *File information* analyses the files on disk. Information such as PE executable structure abnormalities, entropy, whether or not the file is digitally signed with a valid digital signature, imported functions, etc. are all helpful in determining whether a file is suspicious or not.
2. *Memory information* analyses the in-memory image of modules. Since the modules are already executing, it is safe to assume that, at this stage, most modules are decrypted/decompressed and we have access to their unencrypted memory image. Of course, some pieces of malware decrypt some of their internal strings only on demand; in this case the amount of information is limited. Among the information retrieved we mention:
   - Exploits and shellcode.
   - Embedded executables (particularly device drivers!).
   - Strings used by various protocols, interesting registry keys, etc.
   - Whether the in-memory code section exactly matches the on-disk code section (of course, after we apply relocation information).
3. *System information* analyses the way the module interfaces with the system, and possibly other systems.
   - A hidden process, or a hidden module within a process is a warning sign.
   - A process that waits on a specific port, or is connected to a server on a specific port may be a warning sign, depending on the port, server address and other flags.
   - A process with multiple valid and visible windows may be considered less suspicious than a process with no windows, or with windows outside the viewing area of the screen.
   - API hooking, although used in legitimate software as well, is mostly used by malware, typically by injecting unconditional branches to the new handler function. If we analyse the important exported function of a set of system DLLs and follow the branch we will arrive within the memory area of the 'hook' module. Being the target of an API hook may be a very important heuristic flag, particularly if combined with other flags.
   - The *Windows* operating system offers functions like SetWindowsHookEx, used by applications to register themselves to receive messages related to keyboard, mouse and various other *Windows* events. This is also a common technique of injecting a DLL into all processes with a window loop. A DLL loaded via the SetWindowsHookEx family of functions may filter keyboard messages (thus the suspicion of being a keylogger), or other messages.
   - A presence in a 'hot' area of the file system (the Windows or System32 directories, Startup, Temporary Folder, etc.) or presence of an executable in a file's list of streams may represent a warning sign, depending on other factors.

- The way a process is started may reveal interesting information. A process automatically started via an autorun registry key may receive a different score compared to a process that was manually started by the user (we can check the parent process and process execution times).
- For a kernel driver we can gather many pieces of information. A kernel driver's presence in the keyboard, network or file system stack is reported. Some pieces of malware use different tricks to install drivers (system calls like ZwLoadDriver or ZwSetSystemInformation are widely used), or use a legal way of loading the driver but then delete the corresponding registry key. An 'orphan' driver may be a sign of infection.
- Different ways of loading a DLL into the system are important flags in determining whether a file is suspicious or not. Modules registered as 'Winsock Layered Service Providers' (LSP), Winlogon Notification DLLs, modules injected into various system components, or loaded via registry keys like 'AppInit_DLLs' may be of more interest than other modules.

We have to take into account the fact that we are running on a live, possibly infected, system. The way we actually gather information is of huge importance. For example, in order to determine whether a process is hidden or not, we need to have a powerful technique to detect all processes. A vast majority of rootkits hook into the registry and file system functions to hide their presence, so our tool employs specific anti-rootkit functionality. To list the current network connections we currently use the widely documented *Windows* API 'GetExtendedTcpTable'. We are aware that the results of this API can be altered by malware, and we are currently researching techniques to retrieve the list of current TCP/UDP connections as reliably as possible. A possible way of double-checking would be to use remote analysis of the open ports, a technique that is currently being used in the industry.

The server-side part of the engine is responsible for gathering all this information, storing and organizing it depending on a series of rules. When this paper was written the rules were added manually, but we expect to develop automatic weighting of flags in the near future. We currently use the rules for sample prioritization, but we expect to perform automatic blacklisting at least for certain sets of rules. We also expect to integrate new flags gathered from our HIPS component and our sandbox.

We consider this an interesting proactive detection technique; although we never use signatures, the set of IMD flags itself may become a pseudo-signature. It is possible to have completely different files (judging by their binary data) and identical IMD flags, which is an interesting way of categorizing files into families by their behaviour alone.

Unfortunately the reverse is also true: it may be possible to have the same file with different sets of IMD flags. This happens especially when legitimate executables are used for non-legitimate purposes. For example, an IRC script trojan that uses the legitimate mirc32.exe executable, but hides its process and all its windows might be considered suspicious by the server-side decision engine. In this case information such as the file's reputation is very valuable.

Even if the set of IMD flags doesn't contain any user-identifiable information, we acknowledge the fact that some users may consider sending this information a privacy concern. Thus the information is sent to the server only with the user's specific approval.

CONCLUSION

The cloud has been presented by some as the ultimate solution against malware. We have demonstrated that this is not the case just yet: some implementations have bandwidth and privacy issues, while others do not offer the same detection rates as their client-only version. We have shown that there are specific cases when cloud scanning is clearly more efficient than local scanning, but local scanning is still generally the best option. In the future, should virus numbers continue to grow at exponential rates, it would be better to employ a best-of-both-worlds solution – that is to use the server only for storing and manipulating bulky multi-gigabyte data while still keeping most of the scanning logic on the client.

REFERENCES

[1] Oberheide, J.; Cooke, E.; Jahanian, F. CloudAV: N-Version Antivirus in the Network Cloud. Proc. 17th USENIX Security Symposium, July 2008.

[2] Kojm, T. Introduction to ClamAV. http://www.clamav.net/doc/webinars/Webinar-TK-2008-06-11.pdf. Accessed 2 May 2009.