Cryptographic Hashing for Virus Localization

Giovanni Di Crescenzo
Telcordia Technologies
Piscataway, NJ
giovanni@research.telcordia.com

Faramak Vakil
Telcordia Technologies
Piscataway, NJ
farm@research.telcordia.com

ABSTRACT

Virus detection is an important problem in the area of computer security. Modern techniques attempting to solve this problem fall into the general paradigms of signature detection and integrity checking. In this paper we focus on the latter principle, which proposes to label an executable or source file with a tag computed using a cryptographic hash function, which later allows to detect if any changes have been performed to the file. We suggest to extend this principle so that not only changes to the file are detected, but also these changes are localized within the file; this is especially useful in the virus diagnostics which can then focus on the localized area in the file rather than the entire file. This implicitly defines an apparently new problem, which we call "*virus localization*". We design techniques to solve the virus localization problem based on repeated efficient applications of cryptographic hashing to carefully chosen subsets of the set of file blocks, for each of the most important and known virus infection techniques, such as rewriting techniques, appending and prepending techniques, and insertion techniques.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability, Validation*; F.2 [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous; G.4 [**Mathematical Software**]: Algorithm Design and Analysis—*Efficiency*

General Terms

Security, Algorithms, Theory

Keywords

Virus Detection, Virus Tolerance, Virus Localization, Integrity Verification, Cryptographic Hashing

WORM'06, November 3, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

Virus detection is a problem of surprisingly wide impact to any PC user, as she or he is routinely asked to take preventing measures against viruses, such as buying and running antivirus software. In the light of early impossibility results [1,2] for general virus detection, having theoretical value but small practical impact, research in this area has moved towards various heuristic approaches targeting, sometimes with success, specific classes of viruses. Some of the most successful modern techniques attempting to solve this problem fall into the general paradigms of signature detection and integrity checking (see, e.g. [3,4]). The former requires discovering pieces of infected code (called signatures) for known viruses, storing them and developing software that scans the computer memory to search for such signatures. The latter, which we focus on in this paper, requires using cryptographic hash functions that detect unauthorized changes to a file, possibly detecting the presence of unknown viruses. An important example of the success of the latter technique is Tripwire [5], a widely available and recommended integrity checking program for the UNIX environment. It might also be instructive to look at both techniques with an analogy with the two main intrusion detection principles of signature and anomaly detection (see, e.g. [6]): the signature virus detection paradigm is similar to the signature detection principle in the intrusion detection area; the integrity checking paradigm, instead, is more similar to the anomaly detection principle in the intrusion detection area.

Our results. We suggest to extend the integrity checking principle so that not only changes to the file are detected, but also these changes are localized within the file; this is especially useful in the virus diagnostics which can then focus on the localized area in the file rather than the entire file (we stress that this phase is usually both very resource-expensive and failure-prone in the absence of information about the virus). This implicitly defines an apparently new problem in the area of software security, which we call "*virus localization*". We start the study of this problem by characterizing virus infection techniques, such as rewriting techniques, appending and prepending techniques, and insertion techniques, and by posing the problem of designing techniques that achieve virus localization for all of them. Our techniques are based on repeated efficient applications of cryptographic hashing to various subsets of the set of file blocks, chosen according to codes that are specific to the infection technique. In all cases, our techniques achieve satisfactory localization (an area at most twice as large as the

virus is identified in the corrupted file), and efficiency (the hashes returned by our localizing hash functions are longer that those returned by a conventional cryptographic hash function by at most a logarithmic factor). We stress that this output efficiency is a crucial property of a localizing hash function, as there is a trivial construction (which we later discuss in detail) that returns much longer hashes (i.e., by a factor linear in the number of atomic blocks of the input document).

Previous results. To the best of our knowledge, the problem of virus localization has never been rigorously investigated or posed before. Cryptographic hashing is a well-known paradigm for integrity verification, and is fundamental for programs that verify the integrity of file systems, like Tripwire [5]. Cryptographic hashing of all atomic blocks of a file is also a known paradigm, and is important for programs that remotely update files over high latency, low bandwidth link, like rsync [7], or address write-once archival data storage purposes, like the Venti system [8]. We note that none of these programs gets any closer than the mentioned trivial construction to solve our virus localization problem.

Organization of the paper. In what follows, we first review in Section 2 some well-known examples of PC viruses and describe a classification of mechanisms followed by known viruses (this will be useful before designing our localization techniques). Then, in Section 3 we review known techniques for virus detection and describe our novel techniques for virus localization through cryptographic hashing. Finally, in Section 4 we discuss properties and compare performances of our techniques.

2. DEFINITIONS

We recall the known notion and formal definition of (cryptographic) collision-resistant hashing (see, e.g. [9]), and describe a classification of virus behaviors that is useful in the rest of the paper.

2.1 Collision-resistant hashing

Mathematical tools often used in Cryptography, called "collision-intractable" hash functions (a.k.a. "collision-free", "collision-resistant", and sometimes just called "cryptographic"), are very often used for several applications, including those requiring integrity verification of files, messages, etc. These functions are defined as follows: they take as input a binary string of arbitrary length (representing the file to be compressed), and return a fixed-size (e.g., 128- or 160-bit) binary string as output (representing the hash or fingerprint of the original file). Needless to say, the input to the hash function being typically much longer than 160 bit implies that the number of preimages of any single output of the hash function is very large. Yet, the amazing property expected by these functions is that it seems computationally intractable for any efficient algorithm to find even just two preimages that map to the same output, in the sense that any algorithm would have to take infeasible time (e.g., a few centuries or more) to succeed in doing that.

We now recall their formal definition.

DEFINITION 2.1. Let k be a security parameter and let $\mathcal{H} = \{H_w\}_{k \in \mathcal{N}}$ be a family of functions $H_w : \{0,1\}^k \times \{0,1\}^a \rightarrow \{0,1\}^b$, where $a > b$ and w is a function index satisfying $|w| = k$. We say that $\mathcal{H}$ is a *collision-resistant hash function family* if: (1) there exists a probabilistic polynomial time (in k) algorithm computing H_w for each w; (2) for any probabilistic polynomial-time algorithm A, there exists a negligible function ϵ such that, when w is uniformly chosen, the probability that $A(w) = (x_1, x_2)$ such that $H_w(x_1) = H_w(x_2)$ is at most $\epsilon(k)$.

Several constructions of collision-intractable hash functions have been given in the literature under the hardness of specific number-theoretic problems; in practice, for efficiency purposes, heuristic constructions, such as SHA1 or more recent proposals [10], can be deployed.

In designing our techniques, we will use the following property of collision-resistant hash functions (that directly follows from their definition): except with negligible probability, $H_w(x_1) \neq H_w(x_2)$ when $x_1 \neq x_2$, for $|x_1| \geq k$ and $|x_2| \geq k$. We will then define constructions of (composed) collision-resistant hash functions that make repeated applications of an (atomic) collision-resistant hash function over their input, so that the resulting output of the composed function on any two inputs reveals information about the inputs' similarity/difference.

2.2 Viruses

We will consider viruses that perform certain modifications to a *target file*, resulting in a *corrupted file*, where the type of modifications may vary according to specific *infection techniques*.

We first discuss well-known examples of viruses and then present a characterization of PC viruses' behaviors, defining the infection techniques that we consider in the rest of the paper.

Examples of PC Viruses. Let us briefly examine a few examples of well-known virus attacks, e.g., Michelangelo [11], Melissa [12], and Nimda [13], on the Internet to understand how they infected the PCs and disrupted the Internet, its users, and resources.

The well-known Michelangelo virus infected the boot sector of PCs running MS-DOS versions 2.xx and up. It moved the contents of the original master boot record (MBR) file to another location on the disk and placed itself into the MBR. Upon next startups, the basic input-output system would execute the Michelangelo's code that would load the virus into memory, and then would pass the control back to the copy of the original MBR to continue the boot process unless it was March 6. On that day Michelangelo would completely wipe out the hard drive of the PC.

The Melissa virus/worm infected the MS Office applications and spread via Outlook e-mail. For instance, when Melissa virus infected in a MSWord document, its code was located in the DocumentOpen() subroutine that automatically executed when a user tried to open a document.

The Nimda worm would exploit multiple vulnerabilities of the Internet to infect the PCs and servers and spread across the Internet rapidly. This worm would modify web documents (e.g., .htm, .html, and .asp files) and certain executable files found on the systems it infected, and would create numerous copies of itself (using file names with .eml and .nws extensions) in all writable directories. It would propagate through email arriving as a MIME "multipart/alternative" message consisting of two parts. The first part was defined as MIME type "text/html", though it contained no text, so the email appears to have no content. The second

part was a MIME type "audio/x-wav" containing a base64-encoded attachment that was a binary executable named "readme.exe".

Characterization of known PC viruses' behaviors. As illustrated in the above virus examples and discussed in some computer virus literature (see, e.g., [3,4]), most Internet viruses can be characterized to follow one of these infection techniques: Overwriting, Prepending, Appending and Embedding infection techniques, which we now describe in detail.

Overwriting infection technique. The virus infects an executable file by replacing portion of the host's code. One way that this can be achieved is for the virus to simply open the target file for writing as it would open a regular data file, and then save a copy of itself to the file. When the executable file is later launched by its owner, the operating system will execute the code instead of the expected program. This infection technique typically damages the host file to making it not usable.

Prepending infection technique. The virus inserts its code at the beginning of the executable file that it desires to infect. This technique is generally more elaborated than the overwriting technique and was used, for instance, by the virus Nimda [13]. When the executable file is later launched by its owner, the operating system will execute first the virus code since it is located at the beginning of the executable, and then the intended program. This infection technique not only may not damage the host file but may have effects that are not easily detectable from the code's owner. In fact, the presence itself of such a virus may not be easy to detect.

Appending infection technique. The virus inserts its code at the end of the executable file that it desires to infect, and, typically, a jump to the beginning of the virus code at the beginning of the infected file. When the executable file is later launched by its owner, the operating system will jump to execute the virus code first and then return control to the intended program. Similarly as with the prepending technique, the presence of a virus using this infection technique may be not easy to detect. This technique was used, for instance, by the virus Michelangelo [11].

Embedding infection technique. The virus inserts a piece of code or a command pointing to code in a separate file anywhere into a shell script, a Perl script, a Java class file or source code of files that will eventually be compiled into regular executables. Although less examples have been seen of such techniques, they are very useful in reminding that every program containing executable instructions is a potential target for a virus.

Detection and Analysis of PC viruses. Available antivirus software typically uses three main techniques for detecting viruses: signatures, heuristics, and integrity verification, which we briefly explain. The signature technique is similar to the signature detection approach in intrusion detection systems (or, perhaps, a variant of it): first, known viruses are studied and signatures of them are stored; then occurrences of these signatures are looked for into candidate executable files. Although this is the most popular approach for virus detection, it relies on quick update of the signature database by vendors and of their signature files by users, and it is easily defeated by polymorphic and metamorphic virus techniques. The remaining two techniques (heuristics and integrity verification), are more similar to the anomaly detection approach in intrusion detection systems. Heuristic techniques may be somewhat sophisticated in that they attempt to identify viruses based on some behaviors that they are likely to exhibit, such as attempts to write on executable files, to access boot sectors, to delete hard drive contents, etc. Integrity verification techniques try to detect unexpected modifications to files and after the infection has happened, but potentially before the execution of the infected file happens, thus still making the infection harmless. While both heuristics and integrity verification techniques have the potential of catching 'more intelligent' viruses such as those equipped with polymorphism and metamorphism capabilities, they are at most able to raise an alert on a particular file, which later has to be carefully emulated and analyzed in the virus diagnosis phase under a controlled environment where a conclusion about the location, nature and consequences of the potential virus need to be derived. Due to the difficulty of the realization of an accurate controlled environment for emulation, the accuracy of the derived consequences may be not entirely or not at all trusted and it is naturally desirable to strengthen the mentioned techniques. Moreover, in many cases the modification carried by the virus to the original file is very minimal (e.g., a subroutine call to a program located somewhere else in memory), and therefore it would be very helpful to have additional information about the virus itself. In what follows, we describe integrity verification techniques that additionally provide useful virus localization properties.

3. OUR DETECTION APPROACH

As mentioned, the integrity verification techniques attempt to detect unexpected modification to a file, before executing it. In our approach, we would like to extend these techniques so that, in case an unexpected modification is detected, some additional information is obtained about the location of the modification, within the file. We will refer to each of the previously discussed virus infection techniques (characterized as: rewriting, prepending, appending, and ebbedding techniques) and we will design virus localization technique for each of them. To that purpose, it is useful to discuss in slightly greater detail the original integrity verification technique.

3.1 Integrity verification based on cryptographic hashing

Given any collision-resistant hash function family $\mathcal{H} = \{H_w\}_{k \in \mathcal{N}}$ (for simplicity of notation, we will drop the index w from H_w), and a file f, the 160-bit value $hash = H(f)$ gives evidence that any efficient algorithm (possibly creating f itself) would not be able to obtain in feasible computation time or to already have a g different from f meeting the integrity verification test $H(g) = H(f)$. Integrity verification of executable or source files is then done as follows. In a preliminary stage, when the file f is believed to be not infected, or as soon as the file is downloaded from its original source, a tag $hash = H(f)$ is attached to the file. Later, at any time a legal modification on the file is performed, resulting in file f', a new value $hash' = H(f')$ replaces the previous hash value. Later, at any time the file needs to be executed, or compiled for execution, the integrity verification test is performed on the current version f'' of the file; that is, it is checked whether $hash' = H(f'')$. If the match is satisfied, then we have evidence that only legal modifications

have been performed to the file; otherwise, some unexpected modification has been performed. As this modification can be caused by an infection, the file is then returned to an analyzer that tries various diagnosis techniques, including emulating the execution of the file in a safe environment, to derive useful information about the virus and possibly restore the file f'' to its previous, uninfected, state. In what follows, we show techniques that help the emulation process to focus on a certain area of the file that more accurately localizes the potential virus.

We note that the original integrity verification technique does not specifically deal with viruses changing the stored hash value. Presumably, this technique may be modified to deal with this case using any of the following ways:

(a) It assumes the existence of a secure read-only memory, and stores all hash values in this memory (this can actually be implemented more efficiently, by associating all the hash values to leaves of a Merkle tree whose root is the only value stored in the read-only memory);

(b) It assumes the existence of a memory that cannot be accessed by an attacker, and stores a symmetric key k into this memory; then, a keyed hash function H_k is used analogously as H was used in the above description, and a public-key encryption of key k is sent to the user downloading the file so that the user can run H_k. Here, a different key k should be used for each file, and, as before, instead of storing all keys in the secure memory, one can define a Merkle tree over them and only store the root value.

(c) It assumes the existence of a memory that cannot be accessed by an attacker, and stores a signature secret key sk into this memory; then, the computation of a hash value $hash'$ using H as in the above description is followed by a signature sig of $hash'$, and only the value sig' is returned for verification.

In the rest of the paper we also do not deal with viruses changing any stored hash values, and carry on any of these assumptions to our techniques.

3.2 Virus localization based on cryptographic hashing

Towards our goal, the basic property of the above integrity verification technique can be rephrased as follows. Given files M_1 and M_2, and their hashes $hash(1)$, and $hash(2)$, there exists an efficient algorithm DIFF that decides the function $p(M_1, M_2)$, defined as equal to 1 if $M_1 = M_2$ and 0 otherwise. Algorithm DIFF, on input $p, hash(1)$, and $hash(2)$, simply checks whether $hash(1) = hash(2)$ and returns 1 if yes or 0 otherwise. We would like to generalize this approach to more general functions, in particular, to functions detecting information about differences between M_1 and M_2, that are caused by each one among the previously discussed infection techniques (rewriting, appending, prepending and embedding infection techniques). In general, we will talk about *p-revealing* collision-resistant hash functions, that are informally defined as collision-resistant hash functions such that for any efficient adversary returning two files M_1, M_2, there exists an efficient algorithm Det that, on input the two hashes of M_1 and M_2, is able with very high probability to compute the function $p(M_1, M_2)$, where typically p is a function returning some information about differences between the two files. (The formal definition requires a somewhat elaborated probability experiment and is deferred to the full paper.) While an arbitrary collision-intractable hash function does not seem to help in revealing useful information about the nature of the differences between M_1 and M_2 we will consider more involved collision-intractable hash functions (starting from conventional ones) that do achieve that.

An immediate (but inefficient) solution to construct a p-revealing collision-resistant hash function is as follows. First consider that in all our techniques we will view each file M as composed of n atomic blocks $M(1), M(2), \ldots, M(n)$, where a block size can be appropriately chosen; for instance, it could be determined as the smallest semantic unit that a file can be decomposed into (e.g., a line in a source file). Then a very simple hash function is obtained by returning an atomic hash of each block of the input file and then an atomic hash of all the hashes previously computed, where an atomic hash is computed using any collision-intractable hash function. Note that with this technique one can use the above algorithm DIFF on each block of the file and therefore derive any interesting information that is required by function p, for *any* p. The obvious drawback however is that the returned hash is too long, as it is equal to n times the size of a conventional hash, which can be already impractical for files of quite small sizes, such as a few kilobytes. (Still, it is worth noting that this technique is used as a crucial component of the rsync algorithm [7], which addresses the task of remotely updating files over a high latency, low bandwidth link). This motivates our definition of two main metrics associated with any construction of a p-revealing ollision-resistant hash functions.

Two metrics of interest. As a consequence of the previous discussion, the main metric we pay special attention to is the size of the returned hash, which we will refer to as the *expansion factor* of a construction, defined as follows: let H be a collision-intractable hash function, and cH be a construction of a hash function which makes calls to H; then the expansion factor of cH is the size of the output returned by cH on an n-bit input divided by the size of the output returned by H on an n-bit input. Naturally, we aim at designing constructions with the shortest possible expansion factor (e.g., logarithmic in n). This implies very small computational and storage overhead resulting from the added verifications and generation of hashes, since all our constructions can be described as a number of applications of hash function equal to the expansion factor. Our general approach to achieve this performance, as instantiated on each of the infection techniques, is that of computing multiple applications of a basic collision-intractable hash function to carefully chosen subsets of the message bits; here, the choice of the subsets is performed according to an appropriate "encoding schemes" that later allows to "decode" the desired information about the differences between the two messages from the multiple hashes alone, where both the encoding/decoding schemes and the desired information depend on the infection technique.

Another metric of interest will be that of minimizing the localized area in which the virus is detected to belong to. In particular, we aim at returning an interval of blocks containing a number of blocks that is at most $\max(\rho \cdot b(V), 1)$, for some *localization factor* ρ, where $b(V)$ is the number of blocks that virus V is made of.

We note that all our constructions achieve localization factor $\rho = 2$, expansion factor logarithmic in n, and only

require unfrequent, off-line computations of the hashes (the computation overhead is thus not a significant factor towards the efficiency of the overall system).

3.3 Localization of rewriting infections

Recall that in rewriting infection techniques, a virus is a portion of, say, text that rewrites some blocks of the file. We will assume, for simplicity and since this seems to be the case in most virus instances, that the file keeps the same size even after rewriting and that the rewritten blocks are adjacently located. We will design a p-revealing collision-resistant hash function, where p is defined as follows. On input n-block files M_1 and M_2, function $p(M_1, M_2)$ is equal to

- 0 if $M_1 = M_2$;
- (a, b) if $b - a < n/4$ and $M_1(a), M_1(a+1), \ldots, M_1(b)$ and $M_2(a), M_2(a+1), \ldots, M_2(b)$ are the only blocks where M_1 and M_2 differ;
- $\perp$ otherwise.

In the above definition we restrict the analysis to viruses rewriting at most $n/4$ blocks. We consider this to be satisfactory enough as when a larger number of blocks is rewritten, the localization problem loses relevance, in that a localizing hash function is not significantly more useful than a (conventional) hash function.

Our construction. Informally, our construction works as follows. At each step, it uses the atomic collision-resistant hash function H to compute hashes of some blocks of the input file. Given any two files M_1, M_2 that differ due to a rewriting virus that modified M_1 into M_2, the hashes in each step should help in finding the location of the virus by eliminating half of the candidate blocks in M_2. The blocks eliminated are one among these four: the first half or the second half of the yet not eliminated blocks in the file, or, the first half or the second half of an appropriate cyclical shift of yet not eliminated blocks in the file. This is enough to achieve localization factor 2, and, since at each step 4 hashes are computed, output expansion at most $4 \log n$.

Formal description. Let H denote a collision-resistant hash function. Our p-revealing collision-resistant hash function, which we denote as pH, takes as input an n-block message $M = M(0), \ldots, M(n-1)$, and applies $4 \log n$ times the original function H to subsets of the ℓ blocks of M, as follows (we assume that n is a power of 2 for simplicity and logarithms are in base 2):

Step 1: In this first step, the blocks in M are grouped into 4 segments, each obtained by concatenating $n/2$ blocks, defined as

$S_{1,1} = M(0)|\cdots|M(n/2-1)$,
$S_{1,2} = M(n/2)|\cdots|M(n-1)$,
$S_{1,3} = M(n/4)|\cdots|M(3n/4-1)$, and
$S_{1,4} = M(0)|\cdots|M(n/4-1)|M(3n/4)|\cdots|M(n-1))$.

(Note that the indices of the blocks in $S_{1,3}$ and $S_{1,4}$ are obtained by cyclically shifting the indices of the blocks in $S_{1,1}$ and $S_{1,2}$, respectively, by $n/4$.) The 4 hashes in this step are computed as $h_{1,i} = H(S_{1,i})$ for $i = 1, 2, 3, 4$.

Step j, for $j = 2, \ldots, \log n - 1$: Extending step 1, the blocks in M are grouped in $2 \cdot 2^j$ segments, each obtained by concatenating $n/2^j$ blocks, defined as $S_{j,i} = M((i-1) * n/2^j)|\cdots|M(i*n/2^j - 1)$ and $S_{j,i+2^j} = M(((i-1)*n/2^j + n/2^{j+1}) \mod n)|\cdots|M((i*n/2^j + n/2^{j+1} - 1) \mod n)$, for $i = 1, \ldots, 2^j$, (note that the indices of the blocks in $S_{j,i+2^j}$ are obtained by cyclically shifting those in $S_{j,i}$ by $n/2^{j+1}$). The 4 hashes in this step are computed as follows: when $i = 1, 2$,

$$h_{j,i} = H(S_{j,i}|S_{j,i+2}|S_{j,i+4}|\cdots|S_{j,i+2^j-2});$$

and, when $i = 3, 4$,

$$h_{j,i} = H(S_{j,i-2+2^j}|S_{j,i+2^j}|S_{j,i+2^j+2}|\cdots|S_{j,i+2^j+2^j-4}).$$

Output: $h = (\,(h_{1,i}, h_{2,i}, \ldots, h_{\log n-1,i})\,)_{i=1,2,3,4}$.

Properties of our construction. The overall number of atomic hashes of pH is only $4 \log n - 4$, which is essentially optimal (up to a small constant) in a model where each hash reveals one bit of information about the location of the virus interval.

To get convinced that pH is collision-resistant, observe that an algorithm finding collisions in the output of pH finds two different inputs that are mapped by H to the same $h(j|i)$ for at least one of the values $j = 0, 1, \ldots, \log n$, and $i = 1, 2, 3, 4$, which implies an algorithm for finding a collision for H.

We now prove that pH is p-revealing. First of all, we note that by definition of p, it is enough to only consider the case $b - a < n/4$. Then the claim follows by the following lemma: There exists an efficient algorithm that, for any virus interval (a', b') of size at most $n/(2 \cdot 2^j)$, on input M' (a version of M subject to rewriting infection with the virus interval (a', b')) and the hashes obtained until step j of the construction pH on input the original file M, returns an interval (a, b) such that $a \le a' < b' \le b$ and, if $n/2^j \ge (b - a + 1)$, $(b - a + 1) \le n/2^j$. To see that the lemma implies the fact that pH is p-revealing and has localization factor 2, consider any virus interval (a', b') of size $b' - a' < n/4$, and take j such that $n/2^{j+1} \le (b' - a' + 1) < n/2^j$; by applying the lemma for this value of j, we obtain that $a \le a' < b' \le b$, which implies that pH is p-revealing, and $(b - a + 1) \le n/2^j \le 2(b' - a' + 1)$, which implies that it has localization factor 2. We now prove the lemma by induction over j. We show an efficient algorithm *Det* that first computes the output $h' = (\,(h'_{1,i}, h'_{2,i}, \ldots, h'_{\log n-1,i})\,)_{i=1,2,3,4}$, of pH on input M', and then computes $t_{j,i} = 1$ if $h_{j,i} \ne h'_{j,i}$ or $t_{j,i} = 0$ if $h_{j,i} = h'_{j,i}$, for all indices (j, i), $j \in \{1, \ldots, \log n - 1\}$, $i \in \{1, 2, 3, 4\}$, and uses values $t_{j,i}$ to return an interval (a, b).

Base case ($j = 1$). We note that any virus interval (a', b') of size at most $n/4$ is strictly contained in exactly one among $S_{1,i}$, for $i = 1, 2, 3, 4$, and therefore exactly one of values $t_{1,i}$ will be equal to 1. Given values $t_{1,i}$, for $i = 1, 2, 3, 4$, such that $t_{1,i^*} = 1$, algorithm *Det* can return (a, b) equal to $(0, n/2-1), (n/2, n-1), (n/4, 3n/4-1)$ or $(3n/4, (5n/4-1) \mod n)$, if $i^* = 1, 2, 3$ or 4, respectively.

Inductive step. We consider any virus interval (a', b') of size at most $n/(2 \cdot 2^j)$; since $n/(2 \cdot 2^j) \le n/(2 \cdot 2^{j-1})$, by the induction hypothesis, we obtain that there exists an algorithm that by only using the hashes obtained until step $j - 1$ of the construction pH, returns an interval (a'', b'') that contains (a', b') and satisfies $(b'' - a'') < n/2^{j-1}$. We would like to extend the algorithm so that it returns an interval (a, b) that contains (a', b') and satisfies $(b - a) < n/2^j$.

We assume $(b'' - a'') \geq n/2^j$ (or otherwise there is nothing else to prove), that $(a'', b'') = S_{j-1,i}$ for some i (note that $|S_{j-1,i}| = n/2^{j-1}$), and consider the hashes obtained at step j of the construction pH. These hashes contain hashes of 4 intervals $S_{j,i1}, S_{j,i2}, S_{j,i3}, S_{j,i4}$ such that the following holds:

(a) the interval (a'', b'') can be partitioned into the 2 equal-size intervals $T_1 = (a'', b'') \cap S_{j-1,i1}$ and $T_2 = (a'', b'') \cap S_{j-1,i2}$ and into the 2 equal-size intervals $T_3 = (a'', b'') \cap S_{j-1,i3}$ and $T_4 = (a'', b'') \cap S_{j-1,i4}$;

(b) $|T_1| = |T_2| = |T_3| = |T_4|$; and

(c) the indices of the blocks in T_3 and T_4 are obtained by cyclically shifting those in T_1 and T_2, respectively, by $n/2^j$.

Due to (a), (b), and (c), we can apply the same reasoning done in the base case to intervals T_1, T_2, T_3, T_4, and thus obtain that any virus interval (a', b') of size at most $n/(2 \cdot 2^j)$ is strictly contained in exactly one among T_i, for $i = 1, 2, 3, 4$, and therefore exactly one of the values $t_{j,i}$ obtained from the hashes at step j will be equal to 1. Given values $t_{j,i}$, for $i = 1, 2, 3, 4$, such that $t_{j,i^*} = 1$, the algorithm can return (a, b) equal to T_1, T_2, T_3 or T_4, if $i^* = 1, 2, 3$ or 4, respectively.

3.4 Localization of appending infections

We will only deal with appending infection techniques as prepending infection techniques can be dealt with in a symmetric way. Recall that in appending infection techniques, a virus is a portion of, say, text that is appended at the end of the target file, and, as a consequence, the file size increases and after the virus is appended the previous file size may be lost. We will design a p-revealing collision-resistant hash function, where p is defined as follows. On input M_1 and M_2 (of not necessarily equal length), function $p(M_1, M_2)$ is equal to

- 0 if $M_1 = M_2$;
- x if $|M_2| \leq 2 \cdot |M_1|$ and $M_1(i) = M_2(i)$ for $i = 1, \ldots, x-1$ and $M_1(i) \neq M_2(i)$ for $i = x$.
- $\perp$ otherwise.

Similarly as in the case of overwriting infections, here we can restrict the analysis to viruses appending at most n blocks (that is, we assume that the number of blocks in M_2 is at most $2n$, where n is the number of blocks in M_1).

Our construction. Informally, our construction works as follows. At each step, it uses the atomic collision-resistant hash function H to computes 2 hashes of some blocks of the input file. Given any two files M_1, M_2 that differ due to an appending virus that modified M_1 into M_2, the 2 hashes in each step should help in finding the location of the virus by eliminating about half of the candidate blocks in M_2. One main difficulty with appending viruses is that the length of file M_2 may be different from the length of target file M_1, and, therefore, we cannot use "length-dependent" techniques such as those for the rewriting viruses. Instead, we start eliminating candidate blocks from the beginning of the file, by carefully computing hashes on subsets independent from the length of the target file.

Formal description. Let H denote a collision-resistant hash function. Our p-revealing collision-resistant hash function, which we denote as pH, takes as input an ℓ-block message $M = M(1), \ldots, M(\ell)$, and applies at most $2 \log \ell$ times the original function H to subsets of the n blocks of M, as follows:

1. set $j \leftarrow 1$ and $M_j = M$;
2. compute the largest positive integer k_j such that $2^{k_j} \leq \ell/2$

   obtain, from M_j, the following two segments:

   $S_{j,1} = M_j(1), \ldots, M_j(2^{k_j})$,

   $S_{j,2} = M_j(2^{k_j} + 1), \ldots, M_j(2^{k_j+1})$;

   compute hash $h_{1,i} = H(S_{j,i})$ for $i = 1, 2$;
3. repeat step 2 recursively over $M_{j+1} = M_j(2^{k_j+1} + 1)| \cdots |M_j(\ell)$ if $\ell - 2^{k_j+1} + 1 \geq 1$ (that is, unless the new input M_{j+1} has 0 or 1 blocks, when the recursion ends), and set $j = j + 1$;
4. return all computed hashes $(h_{1,i}, h_{2,i}, \ldots, h_{\log \ell, i})_{i=1,2}$.

Properties of our construction. We denote by n the number of blocks of the target file M and by n' the number of blocks of the corrupted file M'. Then we note that there are at most $\log \ell$ recursion steps and therefore the overall number of hashes returned by pH is $\leq 2 \log n' \leq 2 \log n + 1$, since we assume that $n' \leq 2n$. This expansion factor is essentially optimal (up to a small constant) in a model where each hash reveals one bit of information about the location of the virus interval.

To get convinced that pH is collision-resistant, observe that an algorithm finding collisions in the output of pH finds two different inputs that are mapped by H to the same $h(j|i)$ for at least one of the values $j = 1, \ldots, \log n$, and $i = 1, 2$, which implies an algorithm for finding a collision for H.

We now sketch the proof that pH is p-revealing. We start by noting that the computation of pH is divided into recursive steps, where the j-th step takes as input $M_j(s_j)| \cdots |M(\ell)$, and $(s_1, \ldots, s_{\log \ell})$ in an increasing sequence of integers in $\{1, \ldots, \ell\}$ such that $s_j = 2^{k_{j-1}+1} + 1$, for $j \geq 2$, and $s_j = 1$ when $j = 1$. We also note that by definition of p, it is enough to only consider the case $s \geq \ell/2$. Then the claim follows directly by the following lemma: there exists an efficient algorithm that, for any virus interval (s, ℓ) with at least $\ell - s_j + 1$ blocks, on input an ℓ-block corrupted file M', equal to the target file M with interval (s, ℓ) appended to it, and the hashes obtained in the first $j - 1$ steps of the recursion in the construction pH, returns a value x such that $x \leq s$ (that is, the algorithm localizes the virus) and $x \geq 2s - n - 1$ (that is, the localization factor of pH is at most 2).

We now prove the lemma by induction over j. A first observation is that there exists an efficient algorithm Det that, as a first step, runs a slightly modified version of function pH on input M', the modification being as follows.

(a) Step 2′ computes values k'_j and segments $S'_{j,i}$ using the number of blocks n' of M' while Step 2 computes values k_j and segments $S_{j,i}$ using the number of blocks n of M (this implies that, in general, $k_j, S_{j,i}$ may be different from $k'_j, S'_{j,i}$, respectively);

(b) in step 2′, 3 segments (rather than 2 as in step 2) are obtained from M_j: $S_{j,1}$ and $S_{j,2}$, as defined in step 2, and also $S_{j,0}$, defined as $= M_j(1), \ldots, M_j(2^{k'_j - 1})$; at the end of step 2′, the 3 hashes $h_{1,i} = H(S_j(i))$, for $i = 0, 1, 2$, are computed.

Thus, algorithm *Det* can compute an 'augmented' hash output $h' = (\,(h'_{1,i}, h'_{2,i}, \ldots, h'_{\log n',i})\,)_{i=0,1,2}$.

Base case ($j = 1$). Recall that k_1 is defined as the largest positive integer such that $2^{k_1} \leq n/2$, and k'_1 is defined as the largest positive integer such that $2^{k'_1} \leq n'/2$. Since $n' \neq n$, it may hold that $k_1 \neq k'_1$; however, since $n' \leq 2n$, it holds that $k'_1 \in \{k_1, k_1 + 1\}$. We assume wlog that the algorithm *Det* can check whether $k'_1 = k_1$ or $k'_1 = k_1 + 1$ (this can be achieved in many ways; for instance, by augmenting each of the $2 \log \ell$ atomic hashes in the output returned by pH with the number of blocks in its input). First, consider the case $k'_1 = k_1$; then, algorithm *Det*, after computing h', returns $x = 2^{k_1} + 1$ if $h'_{1,1} = h_{1,1}$ and $h'_{1,2} \neq h_{1,2}$; or returns $x = 1$ if $h'_{1,i} \neq h_{1,i}$ for $i = 1, 2$. Now, consider the case $k'_1 = k_1 + 1$; then, algorithm *Det*, after computing h', returns $x = 2^{k_1} + 1$ if $h'_{1,0} = h_{1,1}$ and $h'_{1,1} \neq h_{1,2}$; or returns $x = 1$ if $h'_{1,i-1} \neq h_{1,i}$ for $i = 1, 2$. To see that this output returns a valid localization, it is enough to observe that the following holds: either $|S'_{j,1}| = |S_{j,1}|$, in correspondence to $k'_1 = k_1$, or $|S'_{j,0}| = |S_{j,1}|$, in correspondence to $k'_1 = k_1 + 1$.

Inductive step. The proof of the inductive step is an adaptation of the reasoning in the base case.

3.5 Localization of embedding infections

Recall that in embedding infection techniques, a virus is a portion of, say, text that is inserted somewhere in the middle of the target file (and not at the beginning or the end), thus again increasing the size of the file. We will design a p-revealing collision-resistant hash function, where p is defined as follows.

On input M_1 and M_2 (of not necessarily equal length), function $p(M_1, M_2)$ is equal to

- 0 if $M_1 = M_2$;
- (a, b) if $b - a < n/2$ and $M_1(i) = M_2(i)$ for $i = 1, \ldots, a-1$, and $M_1(i_1) = M_2(i_2)$ for $i_1 = b+1, \ldots, n$, and $i_2 = i_1 + b - 1 + 1$.
- $\perp$ otherwise.

Here we restrict the analysis to viruses embedding at most n blocks (that is, we assume that the number of blocks in M_2 is at most $2n$, where n is the number of blocks in the target file M_1).

Ideas behind our construction. This construction is somewhat less interesting as it is obtained as a direct combination of ideas from the previous two ones.

First, we use a variant of the construction for appending viruses to obtain a 'length-independent' choice of subsets for the localization of the beginning of the embedded virus interval, and we apply the same technique to the reversed input file (i.e., the same file, reordered from the last block to the first) for the localization of the end of the embedded virus interval.

This is then combined with a variant of the construction for rewriting viruses to eliminate, on each step of the construction, half of the candidate blocks in the corrupted file; the combination consists of employing this technique using 'length-independent' subsets of file blocks.

The resulting construction has localization factor 2 and expansion factor $16 \log \ell$, where ℓ is the size of the input file.

Details are deferred to the full paper.

Construction name	Infection technique	Virus length x restriction
detection	any	none
trivial	any	none
pH_1	rewriting	$x < n/4$
pH_2	appending	$x \leq n$
pH_3	embedding	$x \leq n$

Figure 1: Applicability of our techniques.

Construction name	Localizing factor	Expansion factor	Time blowup
detection	n	1	1
trivial	1	n	1
pH_1	2	$4 \log n$	$4 \log n$
pH_2	2	$2 \log n$	$2 \log n$
pH_3	2	$32 \log n$	$32 \log n$

Figure 2: Performance of our techniques.

4. PERFORMANCE ANALYSIS

We now summarize and comment the applicability and performance of our techniques, when compared with two known techniques mentioned in the paper.

The summary is depicted in Figures 1 and 2. (Here, recall that by "detection construction" we denote the construction that only computes a single hash of its entire input file; by "trivial construction" we denote the construction that computes a single hash of each block of the input file; and our constructions in Section 3.3, 3.4, and 3.5, are denoted as pH_1, pH_2, pH_3, respectively.)

We now discuss the entries in the above tables.

While the detection and the trivial constructions are applicable to any infection technique in the characterization given in Section 2, each of the three hash function constructions pH_1, pH_2, pH_3 was tailored for a specific infection technique. Additionally, we note that one should consider the restrictions on the virus length in these constructions to be essentially without loss of generality, as the virus localization problem loses significance whenever such restrictions are not satisfied.

With respect to the localizing factor, we note that the detection construction practically does no localization, while the trivial construction has the best possible localization (as it localizes all infected blocks), and the constructions pH_1, pH_2, pH_3 localize an area only at most twice as large as the infected area.

With respect to the expansion factor, we note that the trivial construction has a too large expansion, while the detection construction has practically no expansion and the constructions pH_1, pH_2, pH_3 only have a logarithmic (in n) expansion of the hash.

Finally, with respect to the time performance, we note that the trivial and detection construction have essentially no time blowup (with respect to a single hash computation), and constructions pH_1, pH_2, pH_3 only compute a logarithmic (in n) number of hashes.

Acknowledgement.

We thank the WORM 2006 committee for very helpful comments.

5. REFERENCES

[1] F. Cohen. Computer Viruses - Theory and Experiments. Computers and Security, vol. 6, 1987.

[2] F. Cohen. On the Implications of Computer Viruses and Methods of Defense. Computers and Security, vol. 7, 1988.

[3] E. Skoudis. MALWARE: Fighting Malicious Code. Prentice Hall, 2004.

[4] P. Szor. The Art of Computer Virus Research and Defense. Addison Wesley, 2005.

[5] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. Proc. of 2nd ACM Conference on Computer and Communications Security (ACM CCS), 1994.

[6] G. Di Crescenzo, A. Ghosh, and R. Talpade. The design and implementation of tripwire: a file system integrity checker. Computer Security - ESORICS 2005, Proc. of 10th European Symposium on Research in Computer Security, vol. 3679 of Lecture Notes in Computer Science, Springer-Verlag, 2005.

[7] A. Tridgell. Efficient Algorithms for Sorting and Synchronization. http://samba.org/ tridge/phd_thesis.pdf.

[8] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. Proc. of USENIX Conference on File and Storage Technologies (FAST), 2002.

[9] B. Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, 1993.

[10] 1st NIST Cryptographic Hash Functions Workshop. http://www.csrc.nist.gov/pki/HashWorkshop/2005/program.htm, 2005.

[11] CERT Advisory CA-1992-02 Michelangelo PC Virus Warning, http://www.cert.org/advisories/CA-1992-02.html, 1997.

[12] CERT Advisory CA-1999-04 Melissa Macro Virus, http://www.cert.org/advisories/CA-1999-04.html, 1999.

[13] CERT Advisory CA-2001-26 Nimda Worm, http://www.cert.org/advisories/CA-2001-26.html, 2001.