CHAPTER 10

Protecting File Systems Against Corruption Using Checksums[1]

Daniel Barbará, Rajni Goel, and Sushil Jajodia

Abstract: We consider the problem of malicious attacks that lead to corruption of files in a file system. A typical method to detect such corruption is to compute signatures of all the files and store these signatures in a secure place. A malicious modification of a file can be detected by verifying the signature. This method, however, has the weakness that it cannot detect an attacker who has access to some of the files and the signatures (but not the signing transformation) and who replaces some of the files by their old versions and the corresponding signatures by the signatures of the old versions.

In this paper, we present a technique called Check2 that also relies on signatures for detecting corruption of files. The novel feature of our approach is that we compute additional levels of signatures to guarantee that any change of a file and the corresponding signature will require an attacker to perform a very lengthy chain of precise changes to successfully complete the corruption in an undetected manner. If an attacker fails to complete all the required changes, check2 can be used to pinpoint which files have been corrupted. Two alternative ways of implementing Check2 are offered, the first using a deterministic way of combining signatures and the second using a randomized scheme. Our results show that the overhead added to the system is minimal.

Key words: Security, data corruption, integrity

[1] This work was supported by the Air Force Research Laboratory/Rome under the contract F30602-98-C-0264.

1. INTRODUCTION

A file system may be a target of an attack that aims to disrupt its correct functioning so that the files containing user data, applications, or system executables become degraded and dysfunctional. To date, information security systems focus on preventing attacks, though experience has indicated that prevention is not completely successful. Our aim is to formulate a defence mechanism that is effective against an attacker who penetrates the prevention system, or an insider attacker. A legitimate user misusing the system within his or her access domain, or attackers having sniffed passwords thus appearing as authorized users could pose the insider threat. Access controls or encryption technologies do not stop this kind of file data corruption. In this paper we describe a technique called *Check2* that is designed to detect malicious changes to a file system. *Check2* provides a protection layer by creating a chain of obstacles that protect a file system from unauthorized malicious modifications, deletions or additions.

Existing integrity and intrusion detection tools assist the users and administrators in monitoring and detecting changes in individual files. To date, integrity-checking tools identify and notify security officials about any changed, deleted or added files, after the occurrence. Established approaches identify corruption using statistical or pattern matching techniques, or by comparing replicated copies of files' contents. Some [3] use securely stored signatures (usually a message digest) to identify an alteration to a single file. The functionality of integrity checking tools, such as Tripwire, relies on the tamper proof security of these signatures. But, this intrusion security system is susceptible to penetration by an attacker who has managed to gain access to the signatures. This well-equipped intruder *directly alters* the stored file signature to coincide with his or her malicious changes to a file. To detect such attacks, the latest release of Tripwire signs the database of signatures with a single signature [8]. In Tripwire, if an intruder gains write access to stored file signatures, he or she can then be successful in corrupting the system by completing one extra signature change. We propose to further complicate the situation for an attacker who gains write access to the checksums, which can be of concern in particular when the attacker tries to restore an old version of a file for which the correct signature had been observed.

Our proposed technique, *Check2*, operates on the file system as one entity; each file alteration produces a set of reactions throughout the file system. These reactions appear as modifications in some pre-calculated values involving the contents of the files. The algorithm utilizes unique combinations (subsets) of the set of file signatures to compute a second and a third level signature. Each level is recomputed upon any change to a file.

To significantly increase the workload of the intruder, each file change triggers numerous changes in these higher-level signatures. When damaging a single file, the intruder is accountable of updating each of these other signatures and/or files for the system validation. Accumulating a *stock*, a databank of old versions of valid files and matching signatures and identifying the string of signatures to replace is part of the intruder's battle. We aim to make this *stock* too large to collect and the chain of reactions nearly impossible to calculate and unrepeatable. The availability of high-speed processors results in low maintenance costs for *Check2* (the extra cost is primarily computation time of extra signatures).

Check2 has one additional very desirable property: If the intruder fails to complete all this work, the system is able to identify the files that have been corrupted, so that the security officer can take proper action (presumably by restoring them to the proper versions, safely stored elsewhere).

We present two strategies for *Check2*. Both use techniques that have been utilized previously to solve other problems (viz., file comparison and mutual exclusion) [2]. From the set of file signatures, the *Check2* algorithm utilizes either a randomized or deterministic procedure to produce a set of subsets. In the randomized algorithm, each file signature appears exactly once in one of the subsets. The deterministic strategy derives the subsets where each file signature appears in more than one subset, which abides by the parities non-null intersection property. In this strategy, the system further calculates a third level of signatures using the same algorithm except using the set of second level signatures in place of the file signatures. Furthermore, the assumption is that authentic changes to the file system occur occasionally. Although *Check2* produces extra system expenses of calculating signature values beyond the existing file signatures upon each update in the system, our empirical results, using the MDS [6] signature algorithm, show that this cost is extremely low. Moreover, this cost is incurred only occasionally, and it is well justified by the chain of copying and the enormous data bank the intruder must accumulate in order to complete a single file corruption.

We begin in Section 2 by presenting a motivating example. In Section 3, we present our deterministic and randomized techniques. Finally in section 4, we offer some conclusions and future directions.

2. MOTIVATING EXAMPLE

To illustrate, Figure 1 displays a simplified application file system with seven files C1, C2, ..., to C7. Each column *i* represents a file name, its content, and its signature value, Ck^1_i.

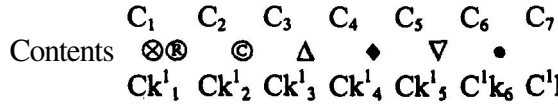

Figure 1. An example of a signature protected file system. The symbols indicate different contents, and these in turn determine the signatures

The system utilizes a trusted and secret one way hashing function f (in reality the function is not the "trusted" part, rather the function will be driven by a trusted, presumably inviolable key that is likely to be stored in a safe place), to compute a checksum value [6,7] involving the contents of each file: $f(\#) = Ck^1_i$. Here "#" represents the contents of the file i (i.e., one of the symbols in {⊗,®,©,Δ,♦,∇,•}). In this paper we use the term signature in a general manner: a fixed-size "fingerprint" generated by a one-way hashing function on any set of data. These checksums are stored in a highly secure database, where the value automatically is recalculated when a file contents changes. When an unauthorized user corrupts a file item, the stored signature will no longer be valid, thus indicating intrusive behavior.

Now, consider a scenario in which the software application new release is to be installed. Some files have been altered from a prior version. The process of installing these new releases of information consists of securing the validity of many separate files. Since each of these files is protected by one signature whose value is unlikely to be calculated by an intruder, a malicious altering appears improbable.

Unfortunately, one may circumvent this protection. A legitimate system user (or an intruder who has broken into the system) may have extensive knowledge of the layout of the file information storage and security system. This intruder tracks previous versions of the software and collects copies of the files and associated signatures. Having *accumulated* previous copies of numerous combinations of the valid file versions with the corresponding matching checksums associated to them, he or she can compromise the integrity of the file system as follows. He replaces a current new version of a file by *copying* over it an entire *old valid file*, *and* copy the matching signature into the database. If the system performs a checksum validation on this block, it will not discover any anomaly in the file data. In our example above, an intruder has copies of the files of the system at time t_0 as it is in Figure 1. After several updates and/or new releases take place, the state of the file system at time t_1 is as shown in Figure 2:

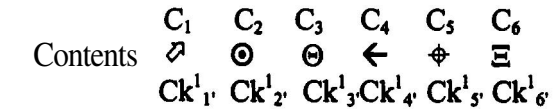

Figure 2. The file system of Figure 1 after a new release

To maliciously modify file C_3 back to the content at t_0, the attacker replaces the current column 3 in figure *2*, with a *copy* of the previous contents, column 3 in figure 1 (checksum changed in the database), resulting on the status shown in Figure 3. File corruption has been successfully completed.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7
Contents	⇗	⊙	©	←	⌖	Ξ	}
	$Ck^1_{1'}$	$Ck^1_{2'}$	Ck^1_3	$Ck^1_{4'}$	$Ck^1_{5'}$	$Ck^1_{6'}$	$Ck^1_{7'}$

Figure 3. The corrupted file system

To detect this attack, we propose a technique that employs a second set of checksums. The system determines subsets of the file signatures and applies a predetermined one way hashing function, *g,* on the contents of each subset, as in figure 4. These *2nd level signatures*, Ck^2_i, are also stored in a secure database.

$Ck^2_1 = g(Ck^1_{1'},Ck^1_{2'},Ck^1_{3'})$ $Ck^2_2 = g(Ck^1_{1'},Ck^1_{4'},Ck^1_{5'})$ $Ck^2_3 = g(Ck^1_{1'},Ck^1_{6'},Ck^1_{7'})$
$Ck^2_4 = g(Ck^1_{2'},Ck^1_{4'},Ck^1_{6'})$ $Ck^2_5 = g(Ck^1_{2'},Ck^1_{5'},Ck^1_{7'})$ $Ck^2_6 = g(Ck^1_{3'},Ck^1_{4'},Ck^1_{7'})$
$Ck^2_7 = g(Ck^1_{3'},Ck^1_{5'},Ck^1_{6'})$

Figure 4. Possible subsets of example system to link the seven file signatures of revised files with 2nd level signatures

With this layer of signatures, when the intruder copies signature Ck^1_3 to match the old copy of the file 3, all second level signatures involving Ck^1_3 in their calculations must also be replaced, namely Ck^2_1, Ck^2_2, *and* Ck^2_3. In order to successfully complete this, two conditions must have occurred. First, at some prior time, the system must have been in such a state where $Ck^2_1 = g(Ck^1_{1'},Ck^1_{2'},Ck^1_3)$, $Ck^2_6 = g(Ck^1_3,Ck^1_{4'},Ck^1_{7'})$, and $Ck^2_7 = g(Ck^1_3,Ck^1_{5'},Ck^1_{6'})$; implying that files 1,2,4,5,6, and 7 contained contents as in Figure 2, and file 3 has contents as in Figure 1. Second, the intruder copied the signatures at that instance into his *stock.* Now, not only must the attacker consistently track prior behavior of the system, store and make the 3 extra signature copies, but also track any other file modifications that occur concurrently in the system during the copying process.

Furthermore, if the intruder fails to change all these files properly and simply performs the change shown in Figure 3, the security administrator will be notified of the anomaly and will be able to track the exact file that was modified. Noting which three 2^{nd} level signatures changed and calculating the intersection of the subsets of the file signatures from which each was computed, only one file, precisely C_3, can be diagnosed as the corrupted file. Moreover, this design is flexible and scalable with the file system size.

3. OUR TECHNIQUE

We consider a file system S that is composed of N files: $F_1, F_2, \ldots, F_N$, denoted by $S = [F_1, F_2, \ldots, F_N]$. Since each file size may be quite large, we first compute a concise representation, called a *signature* or *check sum*, of each file F_i: $SI_i = Ck^1(F_i)$. The *signature* has a fixed length of b bits, for some integer b, and is computed by some one-way hash function. It is stored in a secure database elsewhere. Before a change to a file is saved, the system computes and validates the *signature* value for the updated file. An intruder copying an old version file, F'_i, in place of one current file, F_i, will also need to manually replace the signature of F'_i with the correct match, since during this type of intrusion, the system would not automatically compute values.

Generating signatures may require more computation, though little time is necessary for this and it requires less storage than storing a copy of the entire file. Also, two different pages may have the same signature. The probability of this happening is proportional to 2^{-b}. Moreover, it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target signature [6].

The system responds to any file alteration by computing the 2^{nd} *level signatures*, which utilizes a non-reversible, trusted and secret hash function, (again, the key to the function is unattainable, thus 'trusted'), on a combination of a selection of the file signatures, Ck_i^1's. Thus since it is unlikely that an intruder accesses the ability of compute the signatures, copying valid values is the avenue of corruption. *Check2* computes a maximum of N second level signatures; each is a function of a subset of the N file signatures: $Ck^2_j = Ck(Ck^1_1, Ck^1_2 \ldots Ck^1_i)$ where $1 \le i, j \le N$

Each checksum, Ck^1j, is integrated with $(i - 1)$ other file signatures to compute another signature. We propose two algorithms to determine the elements of these subsets. One is a deterministic approach, using methodology proposed by Albert and Sandler [1]. The other is a randomized strategy that has been previously used to compare file copies in [2].

3.1 Deterministic Signatures

Consider the N files (objects) in our system; from the contents of each, the system generates a set, *A*, containing N elements, each a file signature Ck (F_i), $1 \le i \le N$. A set of subsets of *A* will have at most 2^N elements. We shall not be concerned with every collection of subsets of the set *A*, but rather with those sets which satisfy the specific properties, defined later in this section.

The elements of each subset form the combination onto which the secret and trusted function is applied to calculate the set of 2^{nd} *level signatures.*

The goal is to produce subsets, S_i, which satisfy a pairwise non-null intersection property: $S_i \cap S_j \neq \emptyset$, for any combination *i* and *j* where $1 \leq i, j \leq N$. The fundamental idea of such sets stems from the Sperner sets theorem, which provides existence of sets that follow the properties stated. This provides assurance that when a change occurs in all the elements used in the calculation of one signature (Ck^2j), it directly relates to changes every other 2^{nd} level signature. Each file signature appears in more than one subset and there exists at least one common node between a pair of S_i and S_j. In essence, viewing all the signatures as nodes in a hyperplane, each file signature node will be connected to every other signature node in the system.

Following are properties that the second level subsets satisfy. They mimic the same structure as used by Maekawa [5] to create mutual exclusion in decentralized systems.

– At least one common element between a pair of subsets, S_i and S_j
– $|S_i| = K$ for any *i*
– Any *j* is contained in the D S_i's where $1 \leq i,j \leq N$. (i.e., D is the number of subsets in which each element appears, number of duplications of each distinct element)

Each member of $|S_i|$ can be contained in D-1 other subsets, and the maximum number of subsets that satisfy the intersection property is (D - 1)K+1, where K is the size of the subset. It is shown in [5] that K and D are such that K=D, and that N and K are related as follows

$$N = K(K-1) + 1 \quad (1)$$

As described in [1], such a system of subsets is perceived as a projective plane of N points with properties equating to the conditions stated above. There exists a finite projective plane of order *k* if *k* is a power p^m of a prime p. This finite projective plane has k(k+l) +1 unique points. Hence in our technique for creating subsets S_i's (considering K(K-1)+1 vs. k(k+l)+l) a set of S_i's for the N files exists if (K-l) is a power of a prime. For other values of *k,* the system can still create a set of S_i's by relaxing conditions that each subset having same number of elements K, as well as relaxing the property that each file signature must appear in K-l subsets. As Maekawa [5] also concludes, $K = \sqrt{N}$, with some fractional error.

Figure 5 illustrates an example for a file size of 13, where $S2_i$ represents the checksums using the subset S_i. The change of one file triggers K 2^{nd} level signature changes and if all K files of any subset are altered simultaneously, then all N second level signatures change.

N=13, K=4 $S_1 = \{1,2,3,4\} = S2_1$ $S_5 = \{1,5,6,7\} = S2_5$
$S_8 = \{1,8,9,10\} = S2_8$ $S_{11} = \{1,11,12,13\} = S2_{11}$
$S_2 = \{2,5,8,11\} = S2_2$ $S_6 = \{2,6,9,12\} = S2_6$
$S_7 = \{2,7,10,13\} = S2$ $S_{10} = \{3,5,10,12\} = S2_{10}$
$S_3 = \{3,6,8,13\} = S2_3$ $S_9 = \{3,7,9,11\} = S2_9$
$S_{13} = \{4,5,9,13\} = S2_{13}$ $S_4 = \{4,6,10,11\} = S2_4$
$S_{12} = \{4,7,8,12\} = S2_{12}$

Figure 5. 2nd level signature sets using combinations offile signatures 1-13. The intersection ofany two combinations is nonempty

When the values of N cannot be expressed as K(K -1) + 1, the subsets are built making one of the following corrections [5]:

– Pad the file signature set with extra duplicate values, i.e., increase the value of the N until N can be expressed as K(K-1) + 1.
– Create degenerated (nonequivalent number of elements) sets of S_i's by temporarily assuming enough elements exist so that N = K(K -1) + 1.

The same process is repeated to construct the elements of a *3rd level of signatures* for added protection. These third level signatures are each calculated using the set of second level signatures, $Ck^2 1$. The domain for the hash functions calculating third layer signatures are the elements of the set of Ck^2_1, from which the system generates the subsets, abiding by the above stated properties, for the 3rd level. The system stores the N, or possibly more if N can not be expressed as N = K(K-1)+1, third level signatures. With the inclusion of this level, one file change causes all 3rd level signatures to change. If only the contents of one file change, exactly (N + K) unique signatures automatically change.

3.1.1 Diagnosis of deterministic signatures.

The structured design of the second and third level checksum produces a set percentage of increase in system calculations during an authorized file update. But, it forces an internal intruder to track and copy a precise chain of signatures. With the predetermined subset algorithm, when *one* authorized file changes, *Check2* generally recomputed the K 2nd and N 3rd level signatures; this causes the system to do at most N + K calculations. If N ≠ K(K-1) + 1, then these quantities will be larger because of the increase in the value of K, as displayed earlier above in (2). For the intruder to incur this minimum copying cost, he/she must collect a *stock* of 1st, 2nd and 3rd level signature values, corresponding to the appropriate files. Because all file signature nodes are interleaved in the subsets, this *stock* must include every variation of any previously occurring state of file contents of the entire system. The general (assuming N can be expressed as K(K-1) +1), the combined cost includes collecting all the stock, making K signature copies at

the 2^{nd} level, and N signature copies at the 3^{rd} level, thus the cost of copying becomes **K + N**.

In a crude analysis, it is possible that just one state of the system is captured by the malicious user. For that one instance, this malicious user copies all file contents, their file signatures, 2^{nd} level and 3^{rd} level signatures of that one state into his/her *stock* and then copy over the entire system and signatures. But, with such drastic and noticeable changes, the security administrator would immediately flag such modifications.

Furthermore, the system is able to determine exactly which file is corrupted. This is achieved by calculating the intersection of the K second level subsets. These K subsets are those used in computing the 2^{nd} level signatures that were affected by a particular file modification. The system captures the file signature, Ck^1_i, which is common in these K subsets causing the 2^{nd} level signature to change. The file corresponding to this file signature is which has been modified.

3.2 Randomized Subsets

The randomized technique produces the subsets of the set of file signatures by using a pseudorandom generating algorithm. This strategy uses a notion of creating subsets whose elements are randomly determined each time the *Check2* is executed. This schema avoids malicious tampering, since the system is programmed to randomly change the sets frequently by using a new seed for the pseudorandom generator. Again, for each of the N files, we compute a checksum: $CK^1_i = Ck(F_i)$.

The system's security administrators decide on the number of subsets, *m* ($m<N$), $S_1, S_2, \ldots S_m$, where each set $S_i \subseteq \{CK^1_1, CK^1_2, \ldots CK^1_N\}$. That is, each subset is made up of a number of first-level checksums. The composition of these subsets is decided in a randomized way (there are actually many strategies to achieve this, and we will illustrate one later in the paper). Each subset *m* may not necessarily have the same number of first-level checksums. The signature for the *i*-th subset is constructed as the hash or Exclusive OR of the first-level checksums in S_i. This combined signature is the 2^{nd} *level Signature,* Ck^2_i. If the original signature has b bits, the combined signature will also have b bits. When a file, F_i, is altered, the system generates the 2^{nd} level signatures, compares them to the stored signatures (which were computed using the same random sets) and with authorization, automatically changes those signatures that utilize F_i in its computation.

This algorithm allows varying levels of security. The number of second level signatures is variable, unlike in the deterministic approach, where each 2^{nd} signature is a function of K signatures. The number, *m*, of 2^{nd} level

signatures is directly proportional to the level of security the administration wishes for the system. The more of 2^{nd} level signatures computed, and higher the security because the larger the *stock* necessary to make a valid copy. By frequently recreating new a set of randomized subsets from the set of N file signatures, there is an extremely low probability that the intruder has acquired an exact copy with which to corrupt.

Once *m* is established, at the initialization of the algorithm, using a pseudorandom number generator and a strategy to decide membership, we can assign first-level checksums into the *m* subsets. At any update or alteration of a file, the 2^{nd} level signatures are recomputed and stored and at times. Moreover, the system may randomly redistribute the N signatures by applying again the pseudorandom generator and the strategy to the set of files.

There are many strategies to randomly combine first-level signatures into the second level signatures. A few of them are described and analyzed in the context of the file comparison problem in [2].

To illustrate this, we include here one of their algorithms, called *Innocents.* In this strategy, each first-level checksum is included in a set *Si*, with probability *1/f*, where *f* is a parameter of the algorithm. In the original strategy, *f* meant the number of pages on the file that the algorithm was designed to diagnose as being different in the two copies; in our technique it represents the number of corrupted files that we are set to detect. Due to the randomness of the algorithm, there is a 2^{-m} probability of a page being left out of all second level signature subsets. To resolve this issue, after the random subsets have been chosen, the system checks whether every file has been included in a subset. If it is not the case, it adds an additional set that includes all files that have been left out.

Recall that the *stock* only contains old file copies and corresponding signatures. In this approach, two exact same file systems may have a different set of second level signatures. For a file F_1, the attacker's copy of a 2^{nd} level signature in his or her *stock* from time t_0 may not be the same signature that is generated at time t_1. In the randomized approach, the probability that the randomized distribution for the 2^{nd} level combined signatures corresponds to the copy in the intruder's *stock* is $1/(m^N)$. This does not reflect any modifications to system files from the time of the old copy. Thus, it is very unlikely the intruder process the correct signatures, which the system would validate.

3.2.1 Diagnosis of probabilistic techniques.

Figure 6 shows the algorithm that diagnoses corruption in a file system protected by the *Innocents* technique. This algorithm can be run periodically

in order to alert the security officer from possible corruption in one or more of the files. The algorithm proceeds as follows. First it computes all the subset's signatures and creates a syndrome matrix of elements (one per subset) whose values can be 0 or 1, according to whether the signature of the subset matches the stored signature or not. Once this matrix is built, the algorithm examines those subsets whose signature matched and puts the number of the files included in the subset in a set *T*. At the end, *T* will contain the "innocent" files, i.e., those presumably uncorrupted. The complement of that set is the set of files that might have been corrupted.

In diagnosing corrupted files by probabilistic means, one has to take into account the possibility of false diagnosis. There are two ways in which a false diagnose can happen: an uncorrupted file may be diagnosed as corrupted (false positive), or a corrupted file may fail to be diagnosed as corrupted (false negative). Fortunately, the probability for both of these events can be made arbitrarily low by setting the algorithm parameters (*m,b,f*) properly.

Create syndrome matrix with elements
α_i=*0 if the second-level signature of subset i matches the one stored.*
1 otherwise.
$T = \varnothing$
For i = 1 to m
If $\alpha_i = 0$
Then
For j = 1 to n *if* CK^1j *is in Si then*
$T = T \cup \{j\}$
$T = S - \{T\}$

Figure 6. Probablistic diagnosing algorithms for Innocents

In [2], the analysis of the probabilities for the false positive and false negative diagnoses is presented. We only repeat the results here. The probability of a false positive event can be made less than δ (which can be fixed by the security officer) by insuring that the inequality in Equation 3 is true.

$$b \geq \log(m) + \log(f) + \log(2/\delta) \qquad (3)$$

Equation 3 establishes a lower bound for the number of bits in the signature in order to guarantee that the probability of false positive diagnosis is less that δ.

On the other hand, the probability of a false negative will be bounded by δ if the number of subsets *m* is such that the inequality in Equation 4 is true.

$$m \geq 4f\,(\ln(n-f) + \ln(2/\delta)) \qquad (4)$$

For instance, selecting $m = 500$, $f = 10$, $b > 7$, and for a system with 100 files, we would achieve an upper bound for the false diagnosis of $\mathbf{2^{-10}}$ (or 0.00097).

4. CONCLUSIONS AND FUTURE DIRECTIONS

The file corruption addressed by our analysis involves copying of files by intruders who gain write access to a file system protected by signatures. We have presented *Check2,* a technique to protect a set of files against corruption by insider intruders (or individuals who impersonate authorized users). The system has two very desirable properties. First, it forces a potential intruder to track and perform a string of precise changes, if he or she wants to remain undetected. Secondly, if the attack is not performed properly, *Check2* is able to pinpoint the corrupted pages to the security officer.

The foundation of *Check2* is based on computing checksums for each file, as other techniques such as Tripwire currently use. However, our technique contributes the usage of two or more levels of signatures, combining file signatures with a deterministic or probabilistic schema, in order to increase the work of intruder. This includes calculating the exact lengthy chain of alterations and successfully implementing the changes in a dynamic real-time situation. In either strategy, an additional cost of copying the actual contents of extra *files*, other than the corrupted file can be integrated into each technique.

REFERENCES

[1] Albert, A.A and Sandler, R. An Introduction to Finite Projective Planes. Holt, Rinehart, and Winston, New York, 1968.

[2] Barbara, Daniel and Lipton, Richard, J. A class on randomized strategies for low-cost comparison of file copies. IEEE Trans. Parallel and Distributed System, Vol. 2, No. 2, April 1991, pages 160-170.

[3] Kim, Gene. H., Spafford, Eugene, H., The design and implementation of Tripwire: A file system integrity checker. Proc. 2nd ACM Conference on Computer and Communications Security, 1994.

[4] Kim, Gene. H., Spafford, Eugene, H., Experiences with Tripwire: Using integrity checkers for intrusion detection. Systems Administration, Networking and Security Conference III. Usenix 1994.

[5] Maekawa, Mamoru. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. ACM Transaction on Computer Systems, Vol. 3, No. 2, May 1985, Pages 145-159.

[6] MD5 Message - Digest Algorithm, MIT Laboratory for Computer Science and RSA Data Security, April 1992.

[7] Merkle, R. C., A Fast Software One-way Hash Function. Journal of Cryptology,3(1):43-58,1990.

[8] Website: www.tripwiresecurity.com/vs.html as seen in January 2000.