

# Doc, Wyatt, and Virgil: Prototyping Storage Jamming Defenses

J. McDermott, R. Gelinas, and S. Ornstein  
Center for High-Assurance Computer Systems<sup>1</sup>  
Naval Research Laboratory, Washington, DC 20375, USA

## Abstract

This paper describes progress to date on three prototype tools for detecting storage jamming attacks. One prototype uses a replay defense; another uses logical replication, and the third can be used to determine the source and pattern of a detected attack. Three prototype jammers are used to test the effectiveness of the defenses. Initial experiments have shown that access control, encryption, audit, and virus detection do not prevent or detect storage jamming. The prototype tools have been effective in detecting the same attacks. Object-oriented data storage may require the use of application-specific techniques for applying checksums.

## 1 Introduction

Storage jamming [3] (called information warfare by Ammann, Jajodia, et al. [1]) is a particularly troublesome kind of data integrity attack. Storage jamming is malicious but surreptitious modification of stored data, to reduce its quality. The person initiating the storage jamming does not receive any direct benefit. Instead, their goal is more indirect, such as deteriorating the position of a competitor. We assume that a Trojan horse does the storage jamming, since the Trojan horse may run with privileges the attacker does not have. Manual storage jamming is possible, but in general much less effective.

We call values that should be stored *authentic* values. We call values stored by a jammer *bogus* values. We call the action of storing a bogus value a jam. A storage jamming attack diverges the state of the stored data from the authentic state. The attacker expects that the victim will not detect the damage but will continue to use the damaged data. The *lifetime* of a storage jammer (how long it is able to jam and remain undetected) is a function of the rate and extent of its jamming, the specific user population, and the seriousness of its impact on the real world.

The most promising targets are systems with complex

stored data, the authenticity of which cannot be determined by inspection. This includes legacy systems, distribution and inventory systems, distributed interactive simulations, data warehouses, and command and control systems. The most promising hosts for the attacking Trojan horses are data-creating application programs that do not have high assurance. All the attacker has to do is occasionally write a wrong, but plausible answer. For this reason, more conventional security techniques such as access control, cryptography, and intrusion detection are not generally useful. Since the current trend is toward rapidly developed special purpose applications based on low-cost shrink-wrapped general purpose software, there are many potential hosts.

McDermott and Goldschalg identified detection as the best defense and shown how it can be implemented [4]. Amman, Jajodia, et al. have developed an algorithm for repairing damaged data and for partial operation during an attack [1]. We are currently building prototypes of replication and replay defenses. We use our prototypes to investigate the basic effectiveness (or lack thereof) of these proposed defenses. This paper describes our progress to date. We have two prototypes, one for a replay defense and one for a replication defense.

## 2 Target System

We chose Windows NT (running on IBM PC compatible hardware) as the underlying operating system for several reasons. Windows NT is a modern operating system that has a rich set of security features, and good built-in audit tools. The relatively low assurance of these features is of no consequence, because our example storage jamming attacks do not challenge access control, audit, or authentication features. Windows NT running on PC hardware is relatively low cost, which is good for prototypes. Finally, some of our sponsors are interested in using Windows NT, so the specific results are of interest as well.

Because of its access control features, we limited our

---

1. This work was supported by ONR. Any opinions, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views, policies, or decisions of the Office of Naval Research or the Department of Defense.

target system to the NTFS file system, there are no DOS (i.e. FAT) or OS/2 (i.e. HPFS) partitions. We enabled most of the C2 security features [2], omitting those that have no bearing on storage jamming attacks (e.g. the boot time-out value was not set to zero). Permissions on the *%systemroot%* directory were set so that no one but administrators could change the directory or its contents. Specific files in *%systemroot%* that applications needed to write were then reset to allow *write* or *change* access, to those files only.

We tried a range of audit settings, all the way up to maximum auditing on the target files. Every file and subfolder of the target folder was set for audit of *read*, *write*, *execute*, *delete*, *change* permissions, and *take ownership*. The log manager was configured not to overwrite security log entries but to halt the system if the security log became full.

We also tried two commercial off-the-shelf virus detection packages, running in their maximum search settings (e.g. cryptographic checksums based on all bytes of a file). We do not identify the packages in this paper because their failure to detect jamming does not diminish their value as security tools, when used as intended.

## 2.1 Target Applications

We chose two applications for jamming attacks: Microsoft Access and Microsoft SQL Server.

Our first target application was Microsoft Access. This is an inexpensive database system that has both access control and the capability to encrypt its stored data. This allowed us to demonstrate that storage encryption per se is not a general defense against storage jamming attacks. The discretionary access control in this database system is equivalent to access control lists, i.e. specific permissions can be set for individual named objects, on a per user basis.

The specific application is an in-flight refueling database inspired by Ammann, Jajodia, et al. [1]. It is a toy application with 13 tables, 4 pre-defined queries, and one form. Figure 1 shows the base table *Tankers* from the database. *Tankers* records information about the assignment of tankers to refueling missions. The average amount of data in the database is a few thousand small records. This makes it easy to inspect the integrity of the stored data.

Mission	Slot	Planned Fuel	First	Last name	Type	YR	Serial
Pepsi	1	10000	F	Morton	K21	4	7
Pepsi	1	10000	F	Morton	K21	4	8
Pepsi	1	10000	F	Morton	K19	99	9
Pepsi	1	10000	F	Morton	K19	99	10
Pepsi	1	10000	W	Kim	K21	4	7
Pepsi	1	10000	W	Kim	K21	4	8
Pepsi	1	10000	W	Kim	K19	99	9
Pepsi	1	10000	W	Kim	K19	99	10
Coke	1	10000	F	Morton	K21	4	7
Coke	1	10000	F	Morton	K21	4	8
Coke	1	10000	F	Morton	K19	99	9
Coke	1	10000	F	Morton	K19	99	10
Coke	1	10000	W	Kim	K21	4	7
Coke	1	10000	W	Kim	K21	4	8
Coke	1	10000	W	Kim	K19	99	9
Coke	1	10000	W	Kim	K19	99	10

Figure 1. Base table *Tankers* from the target database

We experimented with a range of security policies for our database. Passwords were assigned to individual users and permissions were set for specific named objects. In the most extreme policy, user *mcdermott* was only given *read* permission for the database form *Refueling* and all other users were given no access. (This strongest policy would probably not be usable in a practical system but was useful in testing the limits of jamming and access control). In all policies, the database was encrypted using the built-in security wizard.

Our second target application is Microsoft SQL Server, a full-featured database system. It uses discretionary access control on objects as fine as individual attributes, on a per-user or per-group basis. SQL Server also utilizes the underlying access control and audit features of the Windows NT operating system. We performed the same series of experiments against SQL Server, with the exception of those related to encryption.

### 3 The Defensive Prototypes

Our prototyping included three defenses: *Doc*, *Wyatt*, and *Virgil*. Each prototype exists in several versions, depending on the features we are exploring.

#### 3.1 Doc

The Doc prototype is an architecture-independent defense that replays command sequences in order to detect surreptitious attacks originating from a single application. Doc records sequences of user keystrokes and later replays them, checking that the results of the replay are the same as the results for the original input sequence. The recorded sequences are stored in scripts that include steps where the state of the stored data is checked against the correct state. The script results are based on an initial state that must be recorded with the script. Our initial Doc prototype uses two kinds of scripts: *statistical scripts* and *syntactic scripts*. Statistical scripts are based on typical usage of the protected application. Statistical scripts contain sequences of commands that appear most frequently, with the most commonly used parameters. Syntactic scripts are based upon language generators that are specified by a grammar. The grammar specifies a legal set of commands or input values that we wish to deny the jammer.

The state of the stored data is recorded as a cumulative CRC-32 check value. We chose a simple, non-cryptographic check value because the check value is not stored with the corresponding detection objects but in an encrypted file. We also chose a simple, non-cryptographic check because the CRC-32 check value calculation is fast. We also want to minimize the storage required for check values.

The defensive principle used in Doc is *trigger denial*. The problem of detecting all jammers by replay is similar to the problem of exhaustive testing. Our alternative is to deny potential jammers the use of the most frequently occurring trigger conditions, the principle of trigger denial. So instead of saying that the Doc prototype stops all jammers, we say that there is no jammer present that uses the trigger conditions exercised by Doc, with further restrictions discussed below. We call a set of triggers that cannot be used (because of replay) a *denial set*. We can design specific denial sets by choosing appropriate replay scripts. In our initial experiments, we use regular sets to analyze the denial set of a script.

An example will clarify. Suppose we wish to protect the relation *Tankers* from attacks that are triggered by values of the attribute *Tankers.PlannedFuel*. (By this we do not mean that a value triggers the attack, but that the

decision to jam is conditioned on values of *Tankers.PlannedFuel*.) Suppose we know, from statistical analysis our application, that the values of *Tankers.PlannedFuel* range between 5000 and 30000. We use Doc to record sequences of commands from frequently occurring tasks (obtained from statistical analysis of the application), so that the most frequent events. For this example, suppose we include four events: *OpenWindow*, *ResizeWindow*, *GetFocus*, and *CloseWindow*. This initial script is a statistical script. Then we modify the script by adding a syntactic generator that causes the script to insert all values between 5000 and 30000 into *Tankers.PlannedFuel*. We now have a script that will detect any attack that is triggered by the events *OpenWindow*, *ResizeWindow*, *GetFocus*, *CloseWindow* or update to *Tankers.PlannedFuel*, and conditioned on a value for *Tankers.PlannedFuel* between 5000 and 30000. Jammers that wish to remain undetected will have to use a trigger that does not use the frequently occurring events and is not conditioned on the value of *Tankers.PlannedFuel*. This latter kind of restriction can be troublesome for the jammer, since it runs the risk of inserting implausible values if it does not check the current data.

The Doc prototype checks all NTFS permanent storage objects (i.e. files and folders) that are accessible to the application being checked. This will detect jammers that change files or folders not currently opened by the application. For example, a jammer hosted in Microsoft Access may target *xls* (i.e. spreadsheet) files in the current user's folder, so a check of the currently open *mdb* (i.e. database) file is not enough. On the other hand, for performance and security reasons, we do not plan to use Doc in a mode that allows us to detect bogus changes to any data stored on a system. So we only detect jammers that do not break access control or encryption.

The scripts, initial state, and check values of the Doc prototype can be stored on removable media (e.g. Iomega JAZ drive). This allows us to increase the size and variety of the scripts that can be used. The removable nature of these drives and their built-in read/write protection has the benefit of complicating matters for Trojan horses that may try to search for or tamper with the scripts. The final version of Doc will be designed to run from the removable media, so no image of the Doc system will be on fixed storage. We are also experimenting with encryption for the scripts and code, to guard against tampering.

Doc has a graphical user interface, but also works with command lines. The graphical user interface allows the user to name a script, specify the target directory that defines the environment for a replay, or to chose a script

to replay.

Doc's output includes a visible alarm and the recording of an event in the security log of the target machine.

Doc is designed to allow the creation of scripts based on recorded command sequences. This enhances the indistinguishability [4] of the detection replays and is much easier to use. Doc includes a script editor that allows a Doc user to insert commands into previously recorded sequence of user interface commands. The inserted commands can build a script that inserts a range of data values into a table.

Doc's script language allows full manipulation of keyboard and mouse commands. The keyboard commands duplicate keystrokes in the order that they are to be replayed. Pressing the key and releasing are considered to be two separate actions. The number of milliseconds to wait before the next action can also be specified (if not specified, a default value will be used).

Mouse movements specify the row and column on which the mouse cursor is to be placed, as well as the time delay before the next action. Right, middle, and left mouse buttons can be double-clicked, pressed and released.

The following shows a sample of a Doc script:

```
LBUTTONDOWN 96 CB DELAY=94
LBUTTONUP 96 CB DELAY=0
STRING TEST
KEYDOWN <RIGHTSHIFT> DELAY=656
KEYDOWN 3 DELAY=203
KEYUP 3 DELAY=109
KEYUP <RIGHTSHIFT> DELAY=172
FIELD ###
KEYDOWN <RETURN> DELAY=562
KEYUP <RETURN> DELAY=63
MOUSEMOVE 96 C9 DELAY=5187
MOUSEMOVE 99 CA DELAY=16
MOUSEMOVE 9E D0 DELAY=31
MOUSEMOVE AC D6 DELAY=16
MOUSEMOVE BC E4 DELAY=15
LBUTTONDOWNBC E4 DELAY=297
LBUTTONUP BC E4 DELAY=125
```

There are several areas of note. The *STRING* command breaks the text output into *KEYDOWN* and *KEYUP* sequences. Thus:

```
STRING TEST
```

Breaks out into:

```
KEYDOWNT DELAY=63
KEYUP T DELAY=63
KEYDOWNE DELAY=63
```

```
KEYUP E DELAY=63
KEYDOWNS DELAY=63
KEYUP S DELAY=63
KEYDOWNT DELAY=63
KEYUP T DELAY=63
```

where 63 milliseconds is the default delay time.

The *FIELD* command allows for strings of text that change with each replay of the script. This gives the ability to test for certain trigger values, as well as make it more difficult for the virus to detect our scanning.

Like the *STRING* command, the *FIELD* command is disseminated into the following:

```
FIELDCHAR # 1
FIELDCHAR # 1
FIELDCHAR # 1
FIELDACTIVATE 1
```

*FIELDCHAR* contains part of a sequential chain of values that ultimately are recreated into the field template (The # character specifies a digit from 0-9). *FIELDACTIVATE* is used increment the field and output the appropriate *KEYUP* and *KEYDOWN* commands.

Using the preceding script and replaying it 3 times, the output would be:

```
TEST:0
TEST:1
TEST:2
```

By using three # characters, we tell Doc to output sequentially, incrementing digits from 0 to 999. If more than 1000 iterations are used, we roll back to 0.

### 3.2 Wyatt

The Wyatt prototype is a replication-based defensive tool. Wyatt is designed to use logical replication as a defense. Replication in general is problematic. Under many approaches, bogus data can be replicated automatically and precisely to many locations. However replication works as a defense if we use logical replication over distinct application systems. Many replication algorithms copy data values from the source data item to its replicas. However, logical replication copies the command that caused the source data item to change. The command is executed at each replica's site and, because of one-copy serializability, results in the same new value for the replica. If we assume a distinct provenance for the application system at each site, then a would-be attacker must install distinct (i.e. site-specific) cooperating Trojan horses at each site. The prototype can achieve higher assurance by increasing the distinction in provenance. The most distinction would

be a completely heterogeneous set of sites, with software from different vendors, purchased via blind buys. Assuming the would-be attacker can insert set of distinct Trojan horses is not enough; the Trojan horses must be implemented to insert bogus data in one-copy serializable fashion. In an architecture that prevents unauthorized communication between sites, this will require the attacker to use stateless jammers. Stateless jammers are less effective because the rate and extent of their jamming is less predictable, because it is entirely dependent on user input. For the same reason, it will be easy for a victim to reproduce the effects of a stateless jammer and locate it.

The Wyatt prototype can detect stateless cooperating Trojan horses at  $n-1$  sites, if there are  $n$  replication sites. In cases where the Trojan horses are not stateless, it may even detect the presence of  $n$  cooperating Trojan horses at  $n$  sites, because their serializability may fail. In either case, replication is a more effective defense than the more general replay approach. There are two reasons why a replicated defense is more effective. First, a replication defense over  $n$  sites can deny all triggers at  $n-1$  sites. Second, the replicas can be used for damage assessment and for continued operation during and after a storage jamming attack.

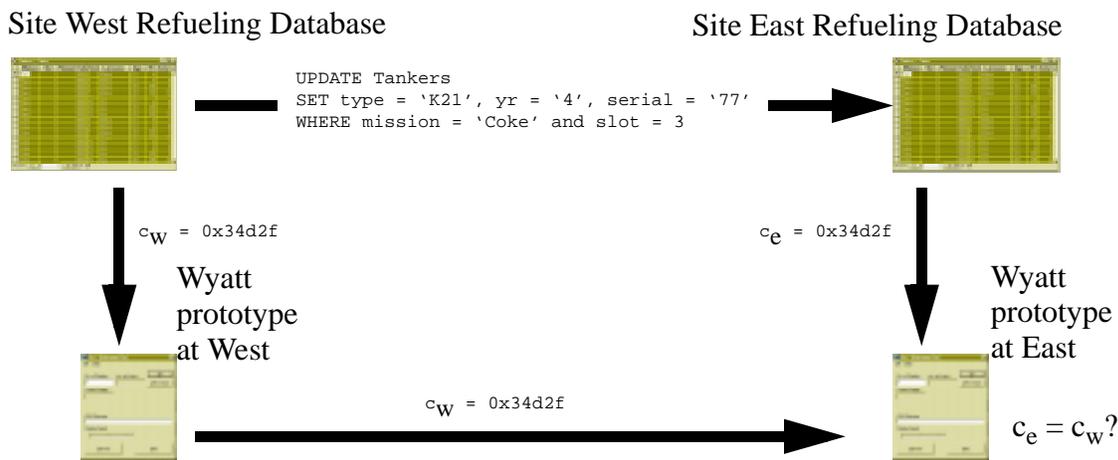
Detection is simple in the Wyatt prototype. There is a

detection process at each source or replica site. Following changes to protected data, the process at the source site computes a checksum and sends it to each replica site, along with the identification of the change. After the logical update is performed at a replica site, the detection process at the replica site computes its own checksum and compares it to the checksum transmitted by the source site detection process. If there is disagreement, there is a problem.

An example will clarify. Suppose we have replicated our target refueling database at two sites *East* and *West*, with the primary copy at site *West* (see Figure 2). Authentic changes to the *Tankers* relation are replicated to site *East* by logical replication. So if *Tankers* is changed at site *West* by the SQL command

```
UPDATE Tankers SET type = 'K21',
  yr = '4', serial = '77' WHERE
  mission = 'Coke' AND slot = 3,
```

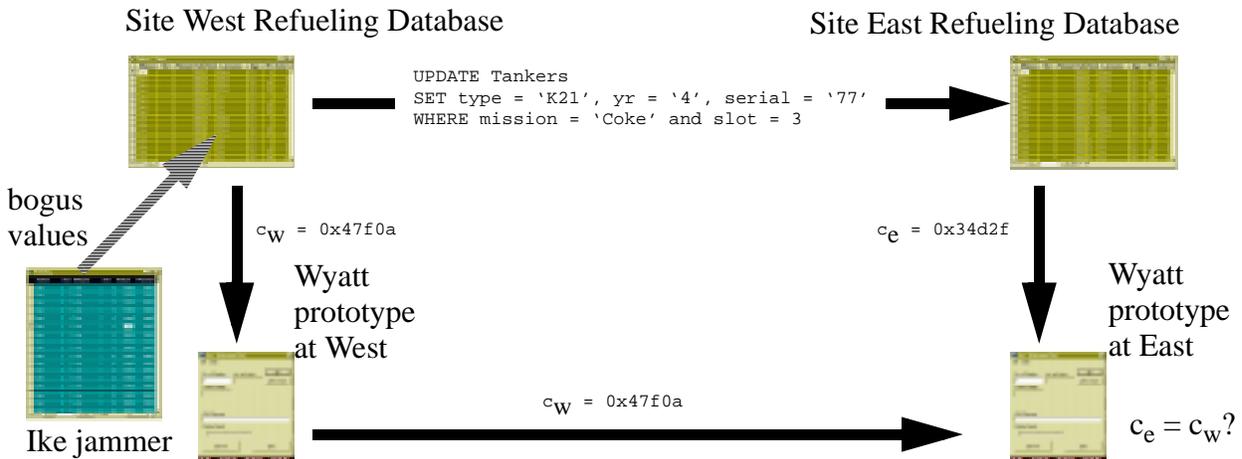
then this command is sent to site *East* and run against the copy of the *Tankers* relation at *East*. Wyatt will compute a checksum  $c_w$  at *West* and send it to site *East*; at *East*, the local copy of Wyatt will compute its own checksum  $c_e$  and compare the two. In this case, they will agree because of the one-copy serializability of the replication mechanism.



**Figure 2. The Wyatt Prototype**

Now suppose we have a jammer running at site *West* that reduces *Tankers.PlannedFuel* by 10% for a single tuple, each time a read-only form named *Refueling* is opened at site *West*. If the form is opened, then the *Tankers* relation is changed at *West*, but not at *East*. On

the next authentic update, the checksum  $c_w$  sent to *East* by Wyatt will not agree with local checksum  $c_e$  computed by Wyatt at *East*. See Figure 3 below.



**Figure 3. Wyatt Detecting the Ike Jammer**

The Wyatt prototype can also protect applications that do not have logical replication built in. It protects applications that cannot replicate appropriately by recording the inputs and replaying them at the replica sites. If the context for the recorded commands is replicated at each site, then replaying the commands will update or create new data that is logically identical at each site. We have already done some testing of this for Microsoft Word. In the case of database systems, this form of replication may be preferable to any built-in replication [5]. This is because database replication by itself may not catch bogus values introduced by Trojan horses in programs such as middleware or window managers.

### 3.3 Virgil

The Virgil prototype is a data file checker. Virgil is a tool that can pinpoint an attack detected by Doc. Virgil uses complex check value structures that can identify the specific file or folder that was corrupted, and the time of the unauthorized change. These more precise check values require a significant amount of storage and slow down the checking, so we do not use them in the initial detection process.

The Virgil prototype can also be used to watch portions of a file system. Unlike virus protection software, the Virgil prototype checks arbitrary file types. Virgil also uses logical checksums (see below), so it does not detect changes that do not matter. It also runs entirely from removable media, with its executable code and check values stored outside the normal file system. This makes it more tamper-proof than a system based on a fixed drive.

## 4 The Prototype Jammers

Our prototype jammers included Ike, Curly Bill, and Ringo. We consider them prototypes because they were not designed for “use” as actual Trojan horses. We expended minimal effort on making them either robust or hard to find. Instead, they were designed to be easy to modify for experimentation. We assume that it is possible to build robust, hard to find Trojan horses. However, proving this is outside the scope of our work.

We omit details on the construction of our prototype jammers for obvious reasons.

### 4.1 Ike

Ike is a stored-procedure Trojan horse. The first Ike prototype was coded in Visual Basic for Access and attached to various events or properties within likely Access objects. Ike was designed to apply SQL to select targets for attack, for example:

```
SELECT * FROM Tankers WHERE
PlannedFuel BETWEEN 10000 AND
20000
```

If the query returns enough data, the target is considered large enough to jam. Ike then jams random targets chosen from a fixed fraction of the tuples. Bogus values are generated by simple arithmetic operations applied to the valid data. This makes it easy for us to detect the effects of various experiments.

We were pleased to find that the internal access controls of the Access database system apparently prevent naïve jamming across their boundaries. Attempts to insert bogus values into objects the current user could not modify were prevented. For example, if we restricted user mcdermott to just read access on the table Tankers,

then Ike was unable to modify it when run by user mcdermott.

However, the main point of this is that access control is not generally effective. As discussed below, we were able to jam any data that our user had privileges to modify.

#### 4.2 Curly Bill

Curly Bill is a Trojan horse Java front end program that attacks database files. Curly Bill is not used to experiment with Java security per se, but to investigate jamming via front end software, where the source of the jamming is outside the application being jammed. The approach used in Curly Bill is similar to Ike. First, a potential target is queried to see if its contents are large enough to make detection unlikely. Then random targets are chosen from a fixed fraction of the tuples and bogus values are generated by simple arithmetic operations applied to the valid data.

#### 4.3 Ringo

Ringo is a sophisticated low-level Trojan horse attached to Access. Its main use is to investigate the effectiveness of our defensive prototypes against low-level attacks. Ringo has a separate “user” interface and is designed to display and record extensive debugging information about its own actions. Ringo uses a naive jamming strategy of randomly swapping values of a randomly selected attribute.

### 5 Initial Experiments

Our first round of experiments tested the Doc prototype against the first Ike jammer. The access controls were able to prevent Ike from jamming data, when the user running Ike did not have *write* access to the data (henceforth called a read-only user). We did not experiment with bypassing the controls, other than to run Ike as a

read-only user. We did this to confirm that the access controls did appear to work. We did notice that Microsoft Access did not alert the read-only user to failed attempts to modify a protected object. This lack of notification makes Trojan horse attacks simpler, since an attack does not have to catch and mask exceptions from the user.

The use of database encryption made no difference to the first Ike jammer. This was not surprising to us, since Ike operates from within the database and has access to the plaintext data.

When the user did have *write* access to stored data, we were able to jam it, even when the jammer was associated with a read-only object (e.g. a form defined on a multiple table view). In the case of a stored procedure triggered from within a read-only object, it was possible to jam writeable objects stored in the same database. Once again, Access failed to notify the user that records were being modified, so there was no need to catch these notifications and discard them before the user could see them. This would be a case of jamming from within the GUI of an application.

The audit records generated by these initial experiments were not useful in detecting the attacks. As Figure 4 shows, the audit records are too coarse in granularity. In the case of SQL server, the necessary information must be available for recovery and for replication, but it is not available through the tools that are provided with SQL Server. Even if tuple-by-tuple information was available, it would be very difficult to locate a few bogus changes among the hundreds of authentic modifications. Furthermore, garbage collection of the logs might remove the needed records before they could be examined. SQL Server does provide users a notification that records were modified, so an attacker must include logic to mask out these notifications.

```
4/23/972:24:48 PM Security Success Audit Object Access 560*****WAL-
LACE
Object Open:
Object Server:Security
Object Type:File
Object Name: C:\users\mcdermott\spoofing\testing\ikeS.mdb
New Handle ID:192
Operation ID:{0,7495737}
Process ID:2152207776
Primary User Name:*****
Primary Domain:WALLACE
Primary Logon ID:*****
Client User Name:-
Client Domain:-
Client Logon ID:-
Accesses:
  READ_CONTROL
  SYNCHRONIZE
  ReadData (or ListDirectory)
  WriteData (or AddFile)
  AppendData (or AddSubdirectory or CreatePipeInstance)
  ReadEA
  WriteEA
  ReadAttributes
  WriteAttributes
Privileges:-
```

**Figure 4. An Audit Record Generated By a Storage Jamming Attack**

The Doc prototype did successfully detect every jamming attempt made by the Ike and Curly Bill jammers. We detected direct attacks, where the Ike prototype made bogus changes to data underlying its host object and jamming attacks from read-only objects, where the bogus changes were made to tables not currently open. In its present form, the Doc prototype cannot pinpoint the unauthorized changes more finely than individual tables. Even table-level damage location requires a complex checksum calculation that creates tension between performance and precision. As checksum complexity increases, we can locate damage more precisely, but the time and space needed to perform a check increases too. At present the best approach appears to be to use two scripts, a search script with simple checksums and a location script with complex checksums. The location script is run when a previous search indicates possible storage jamming.

Our initial tests for the Wyatt prototype have detected jamming in Microsoft Word, via the playback scheme we described earlier. Further tests and development of the Wyatt prototype for Microsoft SQL Sever are currently ongoing.

### **5.1 Checksumming the Representation of Objects**

We encountered an interesting factor when we began to develop the Doc prototype. The kind of checksums we needed were not the same as we would need to prevent tampering with a valid original. We want a bit-wise checksum to show that an original is unchanged. Jamming works by creating originals that contain small amounts of incorrect data, so what we need is a checksum that shows the equivalence of two instances of the same data.

Modern object-oriented software, i.e. software that uses objects to store data, will change the representation of an object when the abstract (logical) stored values have not changed at all. There are at least two reasons why this can happen. First, object-oriented applications are usually designed to save state information about their presentation (e.g. window position) and about their last context (e.g. files open for editing). Any changes to this are recorded, even though the user's stored data has not changed, just its display and context. Second, objects that store large amounts of data usually organize the

representation of the data according to complex internal structures such as B trees. Optimization of these structures may cause two equal (but not identical) instances of an object class to have different internal data structures, and hence different checksums for the same abstract values.

If the application to be protected has this characteristic, then we must use abstract checksums to compare the state of the stored data. Abstract checksums are computed over just the values stored by an object, instead of the bits that currently represent the object. For example, in checking our database prototypes, we computed checksums over the tuples stored in each table. If  $crc()$  denotes our checksum function then we compute  $crc(\text{SELECT } * \text{ FROM Tankers})$  instead of  $crc(\text{Tankers.mdb})$ . Readers familiar with relational database systems will notice that even this command may not be enough, we may need  $crc(\text{SELECT DISTINCT mission, fuel, type, serial FROM Tankers order by mission, fuel, type, serial})$ . Abstract checksums are often easy to compute, but they are application specific. For this reason, we may have to develop an abstract checksum class or package for each application we want to protect. This problem is not unique to database systems. We also encountered this in our first experiments with Microsoft Word.

## 6 Conclusions

Our prototypes have demonstrated the feasibility of detecting storage jamming by either replay or by replication. Defense via replication is preferable, because replication can deny a much larger set of triggers. Replication can also be used for continued operation and damage assessment. However, it is not necessary to replicate data to be able to detect storage jamming. The replay defense will work in any architecture, and can be used to provide a reasonable measure of protection.

Design and programming of these prototypes was complicated by the fact that the target systems are not open. Generation, transmission, and handling of low-level events by these black boxes is not uniform. Most of the licenses supplied with the software prohibit disassembly, which further complicates the process of

understanding the interactions between the application and the operating system. Care must be taken to ensure that the scripts used in the replay defense are indistinguishable from real user input.

Our experience so far is that both replication and replay defense tools must be application specific. Checksums used to compare stored values must be logical checksums. Efficient computation of these is not only application specific, but suffers from the problems of closed design mentioned above. In many cases, we expect that either replay or replication will require detailed logic to deal with application specific commands. We conjecture that this difficulty is fundamental to the nature of storage jamming, because data storage is application specific (if we include general purpose database systems in our definition of applications).

Our future plans are to continue prototyping to investigate two more issues we have not looked at: 1) replay and replication of network input, and 2) the impact of application specifics in an operational system.

## References

1. AMMANN, P., JAJODIA, S. McCOLLUM, C. and BLAUSTINE, B. Surviving information warfare attacks on databases. *Proc. of the IEEE Symposium on Research in Security and Privacy*, , Oakland, CA, USA, May, 1997.
2. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD-5200.28-STD, December, 1985.
3. McDERMOTT, J. and GOLDSCHLAG, D. Storage jamming. In *Database Security IX: Status and Prospects* (D. Spooner, S. Demurjian, and J. Dobson, eds.), 365-381. Chapman and Hall, 1996.
4. McDERMOTT, J. and GOLDSCHLAG, D. Towards a model of storage jamming. In *Proc. of the IEEE Computer Security Foundations Workshop*, 176-185. Kenmare, Ireland, June, 1996.
5. McDERMOTT, J. Replication does survive information warfare attacks. In *Proc. of IFIP WG 11.3 11th Annual Working Conference on Database Security*, Lake Tahoe, CA, USA, August, 1997.