preprinted from the Proceedings of the Ninth Computer Security Foundations Workshop
Kenmare, Ireland, June 1996, pp. 176-185

Towards a Model of Storage Jamming

John McDermott and David Goldschlag
Center for High Assurance Computer Systems
Naval Research Laboratory, Washington, DC 20375
{mcdermott,goldschlag}@itd.nrl.navy.mil

Abstract

Storage jamming can degrade real-world activities that share stored data. Storage jamming is not prevented by access controls or cryptographic techniques. Verification to rule out storage jamming logic is impractical for shrink-wrapped software or low-cost custom applications. Detection mechanisms do offer more promise. In this paper, we model storage jamming and a detection mechanism, using Unity logic. We find that Unity logic, in conjunction with some high-level operators, models storage jamming in a natural way and allows us to reason about susceptibility, rate of jamming, and impact on persistent values.

1. Introduction

Storage jamming [10] is malicious modification of stored data, for the purpose of degrading real-world operations that depend on the correctness of the data. The jammer's objective is to reduce the quality of stored data below a certain level, without being detected. We assume the person initiating the malicious modification (frequently via a Trojan horse) does not receive any direct benefit, financial or otherwise, but rather is motivated by more indirect goals such as improving the competitive position of his or her own organization. We also assume that the attacks are not made across access control boundaries. The target data need not be data stored by a database system, it can be any values stored for future reference. We call the values introduced into storage by the jammer *bogus values*. We call the values we mean to store *authentic* values. If a storage object contains a bogus value, we say that *the storage object has been jammed*.

In the past, the most likely motive for attacks that modify data would have been financial gain. The problem of fraud has been addressed by Clark and Wilson [3], by Sandhu and Jajodia [15], and by others [8, 12]. However, changes in technology have made many organizations dependent on information systems. It is now possible to disrupt or degrade their operations by interfering with their supporting information systems [4].

There are several security-oriented data integrity approaches that were designed for other purposes but also may hinder jamming attacks [10]. The Clark-Wilson model offers little protection because the jammer can construct bogus values that satisfy its constraints. The *assured pipeline* of Boebert and Kain [1] also appears promising, but really does not work for data that is not immediately sent to output. Sandhu's *transaction control expressions* [14] do better because a human is required to validate changes, and because updates can be forced to go through multiple integrity domains with separate checks. However, to be fully effective against storage jamming, transaction control expressions require partial correctness[1] of the majority of the software. Wiseman's *extended trusted* [18, 19] path does prevent storage jamming, but also requires partial correctness of every piece of software that accesses data, from keyboard to display. While possible in principle, these last two approaches are unworkable storage jamming defenses in a world of shrink-wrapped general-pur-

[1] We use the term in its broadest sense: that something much more rigorous and resource consuming than conventional software engineering is required. Because it is so easily detected, we consider nontermination to be an ineffective storage jamming technique.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **1996**	2. REPORT TYPE	3. DATES COVERED **00-00-1996 to 00-00-1996**

4. TITLE AND SUBTITLE **Towards a Model of Storage Jamming**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **10**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

pose products and rapidly-developed low-cost custom applications. Since jamming can occur upstream of any encryption process, direct application of cryptographic techniques does not seem to be workable either.

The most promising defense against jamming is detection [10]. Even then, the detection mechanisms must be suited to the problem; the various intrusion detection approaches will not work because the necessary audit and reduction would be computationally infeasible, if even decidable. Instead, we propose *detection objects* as a suitable defensive mechanism.

This paper presents our work to date on formulating a satisfactory model of storage jamming. First we provide an example of jamming, then we look at the criteria a good model of storage jamming should satisfy. We then present our current model and use it to describe an example. Then we show how to add a defense to our example. We conclude by discussing how our model meets our criteria.

At this point, an example will clarify our presentation. Suppose we have a simple word processor with three commands: *new*, to create a new document, *edit*, to enter or change text and figures, and *delete*, to remove a document and return its associated file handle to the file system. The jammer is attached to the word processor. Following certain *edit* operations, it overwrites the authentic file with an earlier version, thus destroying the results of any intervening sessions. (This kind of *repeat-back attack* cannot be detected with conventional or cryptographic sequence numbers because the malicious software is not a third party. Since the malicious software is part of the program that originates authentic files, the malicious software can obtain an arbitrary number of new, valid sequence numbers to associate with earlier versions of word processing documents.) The jammer is not intended to destroy all editing sessions, only a small fraction of them. The jammer does not attempt to write bogus values in any files outside its current access control domain.

2. Criteria for Modeling

We want to use our model to describe and predict three things: the strategies that might be employed by specific jammers, the susceptibility of target systems, and the effectiveness of protection mechanisms we might adopt.

2.1. Strategies for the Jammer

It is helpful to characterize storage jamming in terms of the possible strategies. There are many possible characteristics, we consider eight here that are applicable to models:

Persistence Of Bogus Values: The unauthorized changes can be persistent or the jammer can restore the changed values after an arbitrary length of time. One useful variation of this, shown in our example, is to save deleted objects or values and reintroduce them at a later time. Temporary bogus values would be harder to detect than persistent ones but may still be read by critical applications or system programs.

Security Attributes Of The Jamming Program: The jamming may be done by an authorized program or by an unauthorized program. If it is done by an authorized program it may be done as part of an authorized invocation, i.e., the program simply writes incorrect values, or the jammer may be able to cause an unauthorized invocation of a legitimate application.

Means Of Choosing Bogus Values: The jammer can adopt a number of basic algorithms for generating the data to write. For example, the bogus values can be chosen arbitrarily, randomly, by interpolation, by replay, or by permutation. Arbitrary choices may be easier to detect, but can be performed by small programs that may be easier to insert into a system.

Means of Choosing Target Storage Objects: The jammer can select targets randomly, via some selection criteria, or by simply piggybacking on an application program. The latter approach lets the application chose the target for the jammer. As we said in the introduction, the data can be application data, linkage data, metadata, or system data. Others important target selection characteristics are the level of abstraction and target granularity. For example, the units of target data (storage objects) could be data in a relational database or they could be disk blocks in the nodes of a B^+tree. The jammer could target entire sets or lists of data, or select components of a single storage object.

Rate Of Change In Target Data: If there are many updates to the data, then jamming may be easier. There will be more opportunities and more checks will be required to detect the jamming.

Rate Of Jamming: The rate at which bogus changes are made is significant. A jammer may be designed to jam as fast as possible without being detected, with the expectation that the jammer will only be triggered at a critical moment. Alternatively, the jammer may run continuously and make changes infrequently.

Extent Of Jamming: A slow jammer can still do much damage by using a cumulative strategy of jamming slowly but widely, i.e. ultimately change every value stored in a system. This type of jamming is called *barrage jamming*. On the other hand, a jammer can hope to escape detection but still disrupt operations by only modifying a critical subset of the stored data. This kind of jamming is called *spot jamming*.

Adaptability Of The Jammer: An enemy may hope to do more damage by having the jamming software change its strategy. This may be a simple adaptation, such as changing the constraints that are checked when generating bogus values. The adaptation may be more complex; for example the jammer might try adapt to detection mechanisms that might be present. On defense, we may have to prevent the jammer from reading the data or code of a detection mechanism, in order to frustrate this adaptability.

2.2. Vulnerability to Jamming

A system's vulnerability to electronic warfare is often characterized in terms of interceptibility, accessibility, and susceptibility [17]; this seems appropriate for storage jamming as well. *Interceptibility* is a measure of the ease with which an enemy can determine the existence, function, and location of a system. *Accessibility* is a measure of the ease with which an enemy can reach a system with an effective electronic warfare attack. *Susceptibility* is a measure of system properties that determines the effect of various attacks on the system. In our models we are primarily concerned with susceptibility.

Performance criteria for measuring susceptibility can include

1. *mission failure rate*: the rate at which activities supported by the system fail,
2. *query error rate*: the rate at which queries are not processed according to the system data model, database design, and the authentic history of the system,
3. *record error rate*: the rate at which erroneous records, object instances, etc., occur in storage,
4. *field error rate*: the rate at which erroneous fields of a record, attributes of an object, etc., occur in storage, and
5. *bit error rate*: the rate at which erroneous bits occur in the representation of data.

We want our model to be able to predict vulnerability criteria two through five. Since prediction of criterion one measurements requires subject matter expertise from the application domain and heuristics taken from the realm of artificial intelligence, mission failure rate is outside the scope of this paper.

2.3. Effectiveness of Protection Mechanisms

Our third criterion for a model is ability to describe and predict the effectiveness of defenses we might employ. The most promising approach is detection of jamming [10]. If the jamming is detected, we may often assume that it will cease to be effective. So a system that allows easy detection of jamming may not be very susceptible to it, even though the system has no way of preventing or tolerating the jamming that may occur before detection. Thus we want to be able to describe detection of jamming, both in possibilistic terms and in probabilistic terms (to construct statistics for rate and extent).

3. Modeling

The critical factor in storage jamming is the assignment of values to data objects: which storage objects are changed and are the changes authentic or bogus? For this reason, the operations and structure of our model are based on assignment to variables.

We adopt the Unity [2] model of computation as our starting point. Unity programs have assignments but no control flow. The control flow is replaced by unbounded nondeterministic iteration. Unity defines both a notation for writing concurrent programs, and a logic for reasoning about computations (executions of those programs). Besides its freedom from flow-control, Unity has two important characteristics:

> Unity provides predicates for specifications and proof rules to derive specifications directly from the program text. This type of proof strategy is often clearer and more succinct than an argument about a program's operational behavior.
>
> Unity separates the concerns of algorithm and architecture. It defines a general semantics for concurrent programs and encourages the refinement of architecture independent programs to architecture specific ones.

As an example of the Unity notation, consider the following program which sorts an array of n elements $X[1]\ldots X[n]$ into non-decreasing order. (This introduction to Unity is from Goldschlag [6].) The program consists of an **assign** section, lines 2-4. Unity programs are organized into several sections: **declare**, **initially**, **always**, **assign**. The **assign** section contains all of the assignments. We shall see examples of the other sections as they become necessary.

This program contains $n(n\text{-}1)/2$ statements, each of which swaps an out-of-order pair of array elements. Unity uses the interleaved model of concurrency, so the execution of this program is as follows: Some statement is chosen. The condition (guard) following the *if* is evaluated. If the condition is false, the statement's execution is equivalent to a *skip* statement. If the condition is true, the statement is executed, and the out-of-order pair is swapped. Another statement is then chosen, and the process is repeated. The only restriction on the scheduling of statements is a fairness restriction, which requires that every statement be scheduled infinitely often. Although execution never terminates, a *fixed point* may be reached when all statements are equivalent to *skip*'s.

To demonstrate the correctness of this program, we must first present the specification. We will do this somewhat informally. The specification is broken down into two parts. The first states that the final array is a permutation of the original array. This is a consequence of the following property: the bag of values that fills the array X is unchanged throughout the computation. This sort of property is an *invariant*, specified as: **invariant** $bag(X) = K$. This means, roughly, that if the bag of values in array X is equal to K before executing any statement in the program, then the bag of values in that array is unchanged subsequent to executing any statement in the program. Since every statement at most swaps array values and no values are ever lost, this invariant is true.

The second part of the specification states that the array will eventually become sorted. This is a liveness (or a progress) property. In Unity, this is stated by *true* **leads-to** *sorted(X)*. The value *true* simply states that there are no preconditions on this property. To prove this liveness property, we use the following measure: Imagine a lexicographic less than relation on n elements. If the array is not sorted, then every statement in the program either modifies the array to one with a smaller lexicographic order, or does not change the array at all. Furthermore, if the ar-

```
1 program prg
2   assign
3     < [] i, j : 0 < i < j ≤ n :: X[i], X[j] := X[j], X[i]       if X[j]<X[i]
4     >
5 end // prg //
```

ray is not sorted, some statement modifies the array. By fairness, the array's lexicographic order will eventually decrease. The fact that this lexicographic order decreases may be repeatedly applied by induction, to conclude that eventually the array becomes sorted.

The Unity logic differs from other temporal logic proof systems for reasoning about concurrency because it is really only a subset of temporal logic. Unity's specification predicates provide a simple and powerful vocabulary to specify and reason about the behavior of concurrent programs. They permit the specification of many temporal properties without introducing all of temporal logic. However, the specification predicates **invariant** and **leads-to** are operators that take predicates as arguments, and are not quantifiers like ∀, ∃, or temporal logic's *always* or *eventually*. This means that these operators may not be nested, and that Unity is less expressive than full first-order temporal logic. Unity provides proof rules for taking large formal proof steps, and is (relatively) complete, even though it contains fewer proof rules than other temporal logics. The soundness and completeness of Unity are discussed in [5,9,11].

To see how we use Unity to model storage jamming, let us return to our example jammer, now modeled in Unity logic as program *wj*. We begin by describing the file system as an array of files and the user input as a tuple that contains the intended word-processor operation and the name of its operand. The **define** section is a natural extension to Unity that allows us to define new data types to model complex storage structures.

Now that we have established the structure of the data, we can present the word processor itself, with the attached jammer.

We model the jammer and the word processor as a concurrent program with only one statement that covers lines 16 through 20. This one statement is composed of three assignment statements separated by the *concurrent operator* ||, which means that the three assignments are executed concurrently. The variables that are being assigned must appear only once, to avoid conflicts. The new values of these variables are computed by evaluating the expressions on the

```
1 program wj
2   define
3     command = tuple of  (
4       op : operation;     // the name of the word processor operation to be applied to //
5       file        : int        // the specified file //
6     )
7   declare
8     f = array[0 .. MAX] of filetype    // the part of the filesystem that is accessible to us //
9     t = array[0 .. MAX] of filetype    // simplistic temporary storage for the jammer //
10    in : command              // the command input by the user //
11  initially
12    t = f              // jammer copies the initial values of  the file system //
13    count = 0       // jammer initializes its counter //
14    in = EMPTY  // word processor waits for user input //
15  assign
16    f[in.file], count := in.op(f[in.file]), count+1if count < JAM ∧ in ≠ EMPTY
17                        ~ in.op(f[in.file]), 0           if count ≥ JAM ∧ in.op ≠ edit ∧ in ≠ EMPTY
18                        ~ t[in.file], 0                  if count ≥ JAM ∧ in.op = edit
19  || in := EMPTY                                          if in ≠ EMPTY
20  ||  t := f                                              if in ≠ EMPTY
21end // wj //
```

right hand sides of the assignment operator first, and then doing the assignments. The first assignment statement, beginning on line 16, has three sets of expressions controlled by guards separated by ~. The guard that is true defines which set of expressions is evaluated. At most one guard may be true at a time. If no guard is true, the statement is equivalent to a *skip* statement.

The third assignment statement, on line 20, copies the array *f* to array *t*, modeling the temporary storage of old values for rewrite attacks. For simplicity, we do not test individual files to see if they have changed. The second assignment statement, on line 19, makes the program work in lockstep with the user: it "waits" for a command, performs the request and then "waits" for another command. We say "waits" because, in our model of execution, program *wj* could be chosen to run an arbitrary number of times before any other program could set the value of *in* to some nonempty tuple. Then every execution of *wj* would be equivalent to a *skip*. In execution histories where *in* is set to something other than EMPTY, then *wj* operates on a file once for each time *in* is set, and then resets it to EMPTY.

We call this jammer a *counting jammer* because it uses a counter to control its operation, rather than, say, a random number generator. We designed this jammer not to act when changes to the file system are deletes or resets because we are assuming, on the part of the adversary, that restoring deleted or empty files would attract too much attention.

A proof that the jammer jams is simple. We want to show that, if the trigger condition is satisfied, line 18 of program *wj* is the one that sets the value of the file *f*[*in.file*], using a value from array *t*. The trigger condition requires that some operation whose index in the input sequence is divisible by *JAM*+1 be an (*edit*, *f*[*in.file*]) command. The proof itself is accomplished by superposing a variable *Xc* to count the number of times a bogus value is written. Initially *Xc* is zero and we show that it must be positive if the trigger condition is satisfied. We superpose *Xc* by adding a line ‖ Xc := Xc+1 **if** count ≥ JAM ∧ *in.op* = *edit*, to the one statement in the **assign** section. Since the value of *Xc* is not read by any part of the original program, this superposed variable does not change the behavior of the original program.

We also want to be able to reason about the rate and extent of the jamming by program *wj*. To do this we superpose variable *Yc* to count the number of times an edit operation is input just when *count* ≥ *JAM*. Then we show that $Yc=Xc$ is an invariant of program *wj*. Given this relationship, we can now define simple random variables *X* and *Y* to correspond to the program variables *Xc* and *Yc*. Then we can assign a distribution (for example, a binomial distribution) to *Y*, allowing us to make probability statements $P\{Y \le k\}=y$, for appropriate values of *k*. Since $Yc=Xc$ is an invariant, we can say that random variable *X* has the same distribution function as random variable *Y*.

This completes our discussion of modeling jammers and their targets. The next step is to expand our example to show how we model a defensive mechanism. We choose to model a *quarantine-subsystem detection-object defense* [10]. A detection object is an abstract mechanism that is intended to detect the actions of malicious software that jams storage. It overcomes the difficulty of checking the computation performed by a program, by always being in a precomputed (i.e., predictable) state. If the detection object is not in its proper, predicted state, then it was probably modified by a jammer. We call the storage objects that are intended to store legitimate data, i.e. not detection objects, *protected storage objects*. Protected storage objects and detection objects are defined in terms of jammers that might target them. Detection objects satisfy two properties

1. Detection objects are indistinguishable, to the jammer, from their corresponding protected storage objects, that is, they satisfy the jammer's target selection rules. We call this condition *indistinguishability*.
2. The only legitimate process that modifies detection objects is the jamming detection process. We call this condition *sensitivity*.

A quarantine subsystem detection object defense avoids the defensive problem of distinguishing detection objects from protected storage objects. In a quarantine subsystem the entire subsystem contains only detection objects, thus achieving complete sensitivity. An example would be a fake user implemented by a workstation. The workstation software would log on, create new directories and files, perform word processing edits, etc.

The program *detector* is our model of a quarantine subsystem. It runs interleaved with program *wj*, that is, *wj* [] *detector*. The program *wj* [] *detector* is equivalent to the program obtained by appending the corresponding sections of the two programs, with the interleaved operator [] between the statements of the **assign** sections.

```
program detector
  declare local
    op = array[0 .. LASTSTATE] of operation// script of operations to perform //
    file = array[0 .. LASTSTATE] of int        // script of filenames to be performed upon //
    ok = array[0.. LASTSTATE] of checksum// precomputed checksums for script states //
    var state : int
    var alarm : Boolean
    function check(var f : array of filetype) : checksum
  initially
    state = 0
    alarm = false
    <∀ i : 0 ≤ i ≤ MAX :: f[i] = NEW >
  assign
    in.op, in.file, state := op[state], file[state], state+1if state ≤ LASTSTATE ∧ in = EMPTY
                           ~ op[state], file[state], 0          if state >  LASTSTATE ∧ in = EMPTY
  || alarm := check(f) ≠ ok[state]                              if alarm = false ∧ in = EMPTY // latch alarm//
end //detector//
```

Program *detector* shares the i/o tuple *in* and the file system array *f* with program *wj*. *Detector* is designed to run in lock-step synchronization with the jammer, because of the way it uses the shared variable *in*; the detector only submits a new command when the previous one has been removed and carried out. *Detector* submits a sequence of commands from a script and checks the results by computing a checksum on the entire filesystem, using the function *check*. (We check the entire file system in case the jammer attacks a file other than the current operand.) As it repeats, the script must restore the file system to the same initial state, in order for the checksums to work. The success of this approach requires two assumptions: 1) that we had a good copy of the word processor when we generated the checksum values stored in array *ok*, and 2) that function *check* has the property that $check(f_1) \neq check(f_2)$ iff $f_1 \neq f_2$. It would be difficult to show that either of these assumptions were generally true, but it is reasonable to assume that we can approximate either of them well enough for practical application. In some cases that we do not have room to describe, we can use a Byzantine generals protocol on scripts and check values to ensure that we are using a good copy of the application program to generate the script.

The detector's script is stored in two arrays, one a list of operations *op* and the other a list of filenames *file*. Just like program *wj*, *detector* is a single concurrent assignment. The variable *state* represents the state of the detector and is used to index both the script and the checksums stored in the array *ok*. The detector cycles through the script, checking each result as it goes. We have attached the modifier **local** to the **declare** section to indicate that variables *op*, *file*, *ok*, *alarm*, *state*, and function *check*() are private to *detector* and cannot be accessed by the jammer.

To define our checksum array *ok*, we use the well-known notation for updates to data structures, where a data structure like an array is treated as a function. Thus ($f;\ i : u$) is an array whose ith element is u, with all other elements the same as the corresponding elements of f. By applying the script arrays *op* and *file*, k times to file system f, we get

$$a^{k} = ((\ldots((f;\ file[0] : op[0](file[0]));\ file[1] : op[1](file[1]));\ldots file[k] : op[k](file[k]))$$

This is the authentic value of our file system, after k steps of the detector's script. This allows us to say

$$ok[k] = check(a^{k})$$

So after each command to the word processor, we compare the check of our current file system f to the precomputed check value that f should have. Our program *detector* should set alarm to true if any check of the file system fails to return the precomputed value stored in array *ok*.

To guard against counterdetection of the fixed script cycle, we can interleave arbitrary don't-care operations that are either compensatable or undoable. For example, between script operation $op[i]$ and $op[i+1]$ we could have the detector submit *don't-care*[0] … *don't-care*[k] followed by *undo*[k] … *undo*[0], such that the value of f following *undo*[0] is the proper value for $op[i+1]$, that is a^{1}. The significance of the don't-care operations is that we don't check them and thus don't care what values they produce, as long as we can undo or compensate them away before running a checked operation and we can have different don't care operations each time we chose to interleave them.

We want to prove that our defense will detect the jammer. The the condition we want to prove is $f \neq a^{state} \wedge in$=EMPTY **leads to** *alarm*=*true*. The operator p **leads to** q means that once p becomes true q is or will become true. The proof is quite simple. The precondition of the **leads-to** makes program *wj* a *skip* statement. Then we can show that program *detector* makes the alarm true, as long as we can demonstrate that $f \neq a^{state}$ implies that $check(f) \neq ok[state]$. That implication follows from the definition of $ok[state]$. (For the reader familiar with Unity, proving **stable** *alarm* is also simple, and shows that the alarm latches.)

4. Conclusions

Storage jamming is a new security problem. Unlike confidentiality, information flow is not a central issue. Fraud may also be viewed as a problem of unauthorized flow of assets with the perpetrator being concerned with maintaining the integrity of the assets as they flow the wrong way. In the case of storage jamming, flows are of lesser importance because information is being destroyed at its source. Unlike denial of service, where there is little concern with avoiding detection, storage jamming generally only makes sense if it is not detected. So storage jamming lies between the boundaries of fraud, unauthorized leakage, and denial of service. It is a threat to complicated mission-critical systems where jammers can easily hide. We believe it is possible to provide reasonable protection against such attacks.

4.1. Meeting the Modeling Criteria

The Unity paradigm allows us to construct models of jamming in which the architectural features are limited to only data objects and data flows, the most critical aspect of storage jamming. The proof rules and the proof steps are based on data objects and data flows, so the reasoning is relatively close to our intuitive view of the jamming problem.

We have shown that reasoning about the actions of our example jammer and proposed defense is natural; assignment to variables lets us reason about jamming and detection when and where it happens in a system. We also showed how we can make simple probability statements about these examples. The probability statements are useful in modeling jammer rate and extent characteristics. Extending these probability statements to model susceptibility as bit, field, or record error rates is relatively easy. We expect to be able to extend our present model to reason about query error rates. Because we use assignments, it is easy to model persistence of values stored in specific data objects. Likewise, modeling selection of targets and generation of bogus values can be done in a natural way.

From our perspective, one of the nicest features of the Unity paradigm is the unbounded fair interleaving of the scheduler. This emphasizes any synchronization aspects of a jamming problem. This is particularly interesting for building simple high-level models of real systems, to analyze their vulnerability and the effectiveness of prospective defenses. Both the jammer and the defense will probably be limited in the protocols and interprocess communication primitives they use to synchronize their actions with other programs.

We have yet to investigate the impact of security attributes on our model. Many effective attacks do not need to cross access control boundaries; this can be devastating in production information systems where sharing and interoperation are mission critical requirements. However, we need to understand what kind of attacks might be able to cross access control boundaries.

Our proof sketches showed us that a quarantine subsystem detection object defense would benefit from a script that has two characteristics: 1) new values for the target data objects of each script operation are unique and 2) script operations applied to rewritten values do not result in an authentic file system state. These characteristics keep the detector from skipping over a specific jamming attack because the attack accidentally wrote a correct value or a script-generated update accidentally corrected a bogus value.

4.2. Future Extensions

We would like to be able to model the complex data structures and operations found in practical systems. To do this, we plan to extend the Unity model in two ways: by adding an explicit data model and by incorporating well-defined high-level functions on the right-hand sides of assignments.

Our first extension adds a set-and-tuple constructor data model for describing complex storage structures. We start with a set of *base data objects*; we plan to use the fundamental types *char*, *int*, and *float*. From the base objects we plan to construct *complex objects* from sets and tuples.

We have already used our second extension in the example jammer and in the detection program. Our second extension allows us to have abstract operations on the complex data structures introduced by the first extension. We plan to enrich the assignment semantics by allowing more powerful expressions to appear on the right. To model low-level operations that might be used for jamming, we include bitwise operations (e.g. left shift << , exclusive-or ⊕). The problem with allowing the high-level operators is that, in the unbounded nondeterministic iterative approach, we must assume termination of programs that compute the value of an expression that uses the high-level operators. This is not a serious problem because we are not interested in the possible effects of jamming in the presence of nonterminating operators.

References

1. BOEBERT, W.E. and KAIN, R.Y. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference* (Gaithersburg, Maryland, 1985). 18-28.
2. CHANDY, K.M. and MISRA, J. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
3. CLARK, D.D. and WILSON, D.R. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, April 1987). 184-194.
4. DEFENSE SCIENCE BOARD. *Report of the Summer Study Task Force on Information Architecture for the Battlefield*, December 20, 1994.
5. GERTH, R. and PNUELI, A. Rooting UNITY. In *Proceedings of Fifth International Workshop on Software Specification and Design, ACM SIGSOFT Engineering Notes*, 14, 3, 1989, 11-19.
6. GOLDSCHLAG, D. Mechanically verifying concurrent programs. Dissertation, University of Texas at Austin, 1992. Also available from Computational Logic, Inc. as TR 71.

7. JUNEMAN, R.R. Integrity controls for commercial and military applications, II. In *Report of the Invitational Workshop on Data Integrity* (RUTHBERG, Z.G. and POLK, W.T. editors), NIST, Special Publication 500-168 (September 1989).

8. KATZKE, S.W. and RUTHBERG, Z.G. (editors). *Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPICS)*, NIST, Special Publication 500-160, (January 1989).

9. KNAPP, E. *Soundness and Relative Completeness of Unity Logic*. Technical Report, University of Texas at Austin, Dept. of Computer Science, October, 1990.

10. MCDERMOTT, J. and GOLDSCHLAG, D. Storage jamming. In *Database Security IX: Status and Prospects* (D. SPOONER, S. DEMURJIAN, and J. DOBSON, editors). Chapman and Hall, 1996.

11. PACHL, J. *A Simple Proof of a Completeness Result for leads-to in UNITY Logic*, Technical Report RZ 2060 No. 72085, IBM Research Division, November, 1990.

12. RUTHBERG, Z.G. and POLK, W.T. (editors). *Report of the Invitational Workshop on Data Integrity*, NIST, Special Publication 500-168 (September 1989).

13. SANDHU, R.S. Terminology, criteria and system architectures for data integrity. In *Report of the Invitational Workshop on Data Integrity* (RUTHBERG, Z.G. and POLK, W.T. editors), NIST, Special Publication 500-168 (September 1989)

14. SANDHU, R.S. Separation of duties in computerized information systems. In *Database Security IV: Status and Prospects* (JAJODIA, S. and LANDWEHR. C.E., editors). North-Holland 1991, 179-189.

15. SANDHU, R.S. and JAJODIA, S. Integrity mechanisms in database management systems. In *Proceedings of the 13th NIST-NCSC National Computer Security Conference* (Washington, DC, October 1990), 526-540.

16. THOMSEN, D.J. and HAIGH, J.T. A comparison of type enforcement and Unix setuid implementation of well-formed transactions. In *Proceedings of Sixth Annual Computer Security Applications Conference* (Tucson, Arizona, December 1990), 304-312.

17. VAN BRUNT, L. *The Glossary of Electronic Warfare*, EW Engineering, Inc., 1984.

18. WISEMAN, S., TERRY, P., WOOD, A., and HARROLD, C. The trusted path between SMITE and the user. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, April 1988). 147-155.

19. WISEMAN, S. The control of integrity in databases. In *Database Security IV: Status and Prospects*, (JAJODIA, S. and LANDWEHR. C.E., editors).North-Holland 1991, 191-203.