Support for the File System Security Requirements of Computational E-Mail Systems

Trent Jaeger and Atul Prakash
Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
E-mails: {jaegert|aprakash}@eecs.umich.edu

ABSTRACT

Computational e-mail systems, which allow mail messages to contain command scripts that automatically execute upon receipt, can be used as a basis for building a variety of collaborative applications. However, their use also presents a serious security problem because a command script from a sender may access/modify receiver's private files or execute applications on receiver's behalf. Existing solutions to the problem either severely restrict I/O capability of scripts, limiting the range of applications that can be supported over computational e-mail, or permit all I/O to scripts, potentially compromising the security of the receiver's files. Our model, called the *intersection model* of security, permits I/O for e-mail from trusted senders but without compromising the security of private files. We describe two implementations of our security model: an interpreter-level implementation and an operating systems-level implementation. We discuss the tradeoffs between the two implementations and suggest directions for future work.

KEYWORDS: File systems, security, computer-supported cooperative work, groupware, collaboration technology, computational e-mail, active e-mail.

1 INTRODUCTION

Electronic mail (e-mail) is a standard and popular mechanism for asynchronous communication, enabling users to send messages to one another. *Computational e-mail* [1, 2, 4], also called active or enabled e-mail, extends the power of standard e-mail, allowing a message to contain a command script. The command script in a message is executed automatically when the message is read, enabling a wide variety of actions to be encapsulated in messages. For example, a sender can provide a receiver with a nice interface for replying to a question in the message, for reviewing a document contained in the message, for voting via e-mail on an issue, for placing orders electronically, and for collaborative design.

Unfortunately, computational e-mail also presents a major security risk. When a receiver reads a message containing a command script, the command script is executed under the receiver's access rights. Malicious users can write command scripts that can potentially: (1) remove files from the receiver's file system; (2) fill the file system of the receiver's machine to capacity; and (3) start other applications under the guise of the receiver, such as to send annoying mail messages addressed from the receiver.

Besides computational e-mail, several other systems allow command scripts to be executed, and need to address the security problem. In Telescript[1] [13], a system meant for building electronic marketplaces, clients can send command scripts which execute at a server. In Mosaic, the popular browsing tool based on World-wide Web, an information server can be defined that enables a command script to be run at a client who accesses the server. Several groupware systems with replicated architectures maintain consistency by broadcasting users' commands that run at each site.

File security in the above systems is provided typically either by severely limiting the ways that I/O can be performed within a command script or by trusting users that will not send improper command scripts. Either solution is somewhat unsatisfactory. Severely limiting the I/O capability of scripts makes it difficult to build collaborative applications that must access shared files. On the other hand, permitting all I/O can cause private files to be accidentally or maliciously accessed or modified. We thus conclude that systems that utilize the paradigm of passing command scripts either provide security that is too restrictive or provide insufficient security.

In this paper, we present a security model that pro-

[1] Telesecript is a registered trademark of General Magic, Inc.

CCS '94- 11/94 Fairfax Va., USA
© 1994 ACM 0-89791-732-4/94/0011..$3.50

vides a level of security between the above two extremes. It enables users to collaborate using their shared files while protecting each user's private files. We investigate several ways of implementing the model and discuss the tradeoffs between the implementations. We also suggest ways to tailor the security model in order to deal with a variety of collaborative situations.

The structure of the paper is as follows. In Section 2, we examine how security is currently implemented in several systems that use the command script paradigm. In Section 3, we define our security model. In Section 4, we detail the implementation options for the security model. In Section 5, we discuss some variations of the security model that are useful for supporting various collaborative situations. In Section 6, we present conclusions and directions for future work.

2 PREVIOUS SOLUTIONS TO THE SECURITY PROBLEM

Below, we provide a brief description of systems that utilize command scripts and examine the security mechanisms used in each system. The systems are: (1) ATOMICMAIL; (2) Safe-Tcl; (3) Telescript; (4) Mosaic; and (5) collaborative systems based on replicated architectures.

Borenstein has proposed several implementations for computational e-mail, including ATOMICMAIL [1], in which command scripts are written in LISP, and its successor Safe-Tcl [2], in which scripts are written in another interpreted language, Tk/Tcl [7]. Borenstein has recommended the use of Safe-Tcl as the standard system for computational e-mail, partly because Tk/Tcl provides commands for building applications with graphical user interfaces and is available on a wide variety of platforms. Furthermore, to ensure that computational e-mail can be effectively used in a heterogeneous environment, e-mail messages of Safe-Tcl use a MIME-compatible [3] format, and they can be sent and read (executed) by Internet Mail systems such as mhn [12].

In ATOMICMAIL, file system security is provided by modifying I/O functions in the scripting language to prevent a script from accessing the file system, except for a single public directory. A potential problem with this solution is that since any computational e-mail message can read or write to the public directory, anyone can delete a file in that directory. So, the public directory is unlikely to be convenient to use as a group work directory. To use it as a work directory, users have to copy their work files to the public directory, keep track of any new files created during the collaboration, and then copy the files back to a safe directory.

Safe-Tcl provides two interpreters: a trusted interpreter and an untrusted interpreter. The trusted interpreter provides no security, so it is meant to be used for interaction with trusted sources. The untrusted interpreter provides tight security similar to ATOMICMAIL, by replacing all the I/O functions with safer I/O functions that disable all file I/O, except to a single public directory. This provides security for the file system, but it limits the ways in which computational e-mail can be used. Borenstein recognized that this approach can be too restrictive for many collaborative applications, so he has left open the possibility of "power-augmenting extensions" to the language.

Telescript [13] and Mosaic, the popular information server, allow client/server processing to be specified using command scripts. In Telescript, clients send scripts which execute at the server. File system security in Telescript is provided by eliminating the command script's ability to perform all I/O. Thus, Telescript's security is even more restrictive than ATOMICMAIL and Safe-Tcl's.

In Mosaic, an information server can be defined that enables a command script to be run at the client when the server is accessed. Suggestions for providing security for Mosaic command scripts include: (1) reviewing command scripts before execution and (2) allowing access to only pre-approved, "safe" scripts. The first approach assumes that a user is qualified to judge whether a script is safe or not. The second approach requires extra diligence on the part of the administrators to catalog safe scripts. Neither option provides a strong guarantee of the security of the file system.

Several distributed applications use a replicated architecture in which every command is executed at each site [6, 8]. For example, DistEdit [5, 6], a collaborative editor toolkit, replicates an editor process for each user in the collaboration. User commands are sent to each editor process, to ensure the consistency of the editors' buffers. Unfortunately, this also raises the possibility that, for example, if one user issues a command to save the editing buffer to a file, files with the same name of other users may get overwritten. DistEdit avoids the above problem by not broadcasting file I/O commands, but, in general, system designers must identify and manually close any security loopholes. This task could, in general, be arduous and error-prone.

None of these systems provide access to shared files and applications, which are necessary for collaboration, while protecting private data. The restrictive security provided by Safe-Tcl's untrusted interpreter and Telescript prevent access to shared data at its normal location and prevent the execution of applications. On the other hand, the Safe-Tcl's trusted interpreter and the proposals for security in Mosaic provide no assurance that private files will not be overwritten. DistEdit prevents the overwriting of private files but in an application-specific way.

3 THE INTERSECTION MODEL OF SECURITY

In this section, we illustrate the security problem using an example collaboration and define a security model sufficient for this type of collaboration. This example portrays a collaboration between two users using some shared applications and data. Collaboration requires that both users have access to the shared applications and data. Security is still needed to protect each user's private information, however, since this information is outside the scope of the collaboration.

In our example, two mechanical designers, who work for different companies, collaborate to design an artifact. An overview of their collaboration is shown in Figure 1. Designer A is sub-contracted to develop a part of a system that is being designed by designer B. The two designers collaborate on the design using a shared file space maintained at designer B's site. Since designer A does not work for B's company, B wants to ensure privacy of some files from designer A.

Suppose that A makes a design decision and wants B to approve it. The approval is based on some analysis information that is generated by a computer-aided design (CAD) tool available to both designers. To get B's approval, A sends a computational e-mail message to B that: (1) starts the CAD application; (2) has the CAD application load the design data files; and (3) puts the CAD application in a state that shows the analysis of the design decision.

Upon receipt of the message from A, designer B should be able to "read" the message (resulting in execution of the script contained in the message) and send back a response. This collaboration should be supported by enough security so that designer B does not have to worry about designer A stealing or accidentally modifying any private data. However, since any process designer B executes has, by default, the same access rights as B, B should be concerned at this point.

The security requirements for A (the sender) in this collaboration are listed below:

- **Prevent access to private files**: B should not be able to access A's private files. This implies that the computational e-mail message cannot be executed using the sender's user ID.
- **Allow access to shared files**: In order to execute the script, the computational e-mail message needs to be able to access both the CAD application as well as the design data files. Therefore, the security model needs to provide the ability to execute shared applications (i.e., other scripts or executable code) and to read/write shared files.

Now, we list designer B's (the receiver's) security requirements:

- **Prevent access to private files**: The computational e-mail script should not be able to access/modify B's private files. This implies that the computational e-mail message needs to run with more limited access rights than the receiver's normal access rights.
- **Permit access to shared and public files**: The computational e-mail script, and the CAD application invoked by it, should be able to access/modify files that are shared between A and B.
- **Not affect access rights of other processes:** The e-mail message must be executed under more limited access rights than B's, but this should not cause the access rights of B's other processes on the workstation to change. Otherwise, those processes may be prevented from accessing necessary files.
- **Make access requirements explicit:** B should know what access A requires, so as to be able to determine whether or not to take the risk of executing the message.

We define a security model, called the *intersection model*, that represents the security requirements presented above. In the intersection model, the access rights for each process in the collaboration are the access rights that are common to both users in the collaboration. The access rights available to a process using the intersection model are shown in Figure 2. For example, if designers A and B have read access to a CAD data file, then the computational e-mail process also has read access to this file. However, if designer A does not have write access to the file, then the computational e-mail process also does not have permission to write to that file.

Another requirement of the intersection model is that it permit processes that are active at the same time and belong to the same user to have different access rights. Using the intersection model, a user can undertake multiple collaborations that have different access rights. Consider Figure 3. Suppose that B, while executing A's computational e-mail message, is also executing a computational e-mail message from another user C. The access rights of the two execution processes must be different because the files shared between each set of collaborating users are different.

4 IMPLEMENTATION

In this section, we investigate the implementation of the intersection model via three different mechanisms and compare the mechanisms: (1) by the use of tokens and ACLs in the Andrew File System (AFS); (2) by

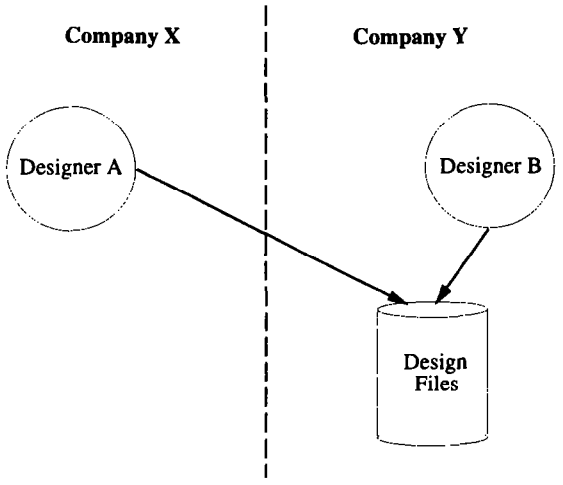

Figure 1: Design Collaboration Example

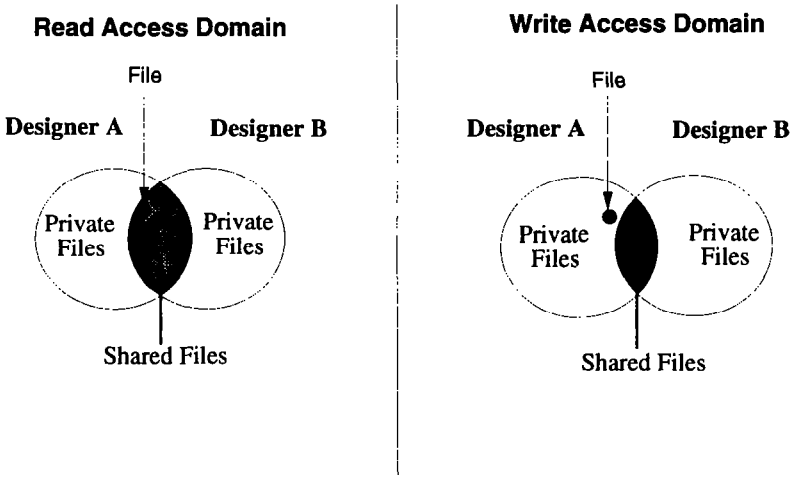

Figure 2: Intersection of Access Rights

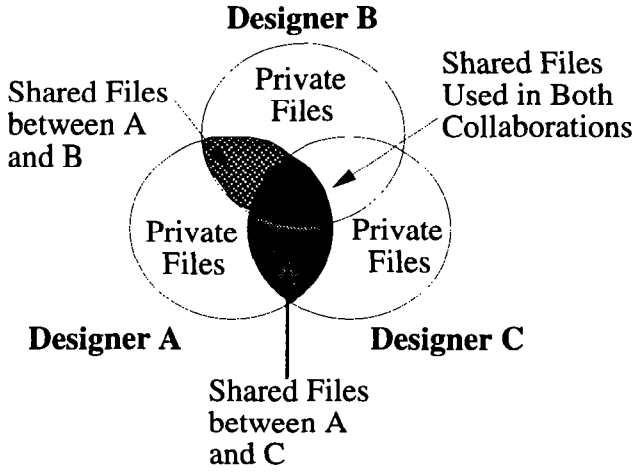

Figure 3: Multiple Processes with Different Access Rights

modifying the Safe-Tcl interpreter; and (3) by the use of user IDs, group IDs, and file permission mode bits in Unix[2]. All of the implementations described in this section require a mechanism to determine the identity of the sender. If the sender's identity cannot be determined, the implementation assumes the sender to be a dummy user, **nobody**.

To prevent access to the receiver's private data, the basic idea in the OS-based implementations of the intersection model, which use AFS ACL's and Unix mode bits, is to first remove the access rights of the process under which the computational e-mail executes. We then give back some of the rights so that the process has access rights to files accessible by both the sender and the receiver. As the following sections show, there does not seem to be an easy way to do the above using AFS file protection mechanisms, but we can devise a solution using Unix mode bits.

In the Safe-Tcl implementation of the intersection model, each I/O command is validated by the interpreter to determine if the command is legal with respect to the intersection model. As we show, this model works effectively as long as the Safe-Tcl command script does not execute any other non-Safe-Tcl applications. If such an application is allowed to be invoked, a security risk arises because the application may perform I/O without validating the I/O commands with respect to the intersection model.

4.1 AFS-BASED APPROACH

First, we examine the ability to limit a user's access rights using the security model provided by AFS [10]. AFS and Unix security models are often used together to provide file system security, but much of their functionality overlaps. In this section, we examine the use of only the AFS-specific functionality.

File security in AFS [9] is provided by: (1) tokens and (2) access control lists (ACLs). Tokens are granted by an authentication server called Kerberos [11]. The AFS authentication mechanism compares the token against each entry in the requested file's ACL. If the owner of the token matches a user on the ACL, or is a member of a group on the ACL, access privileges associated with the matching ACL entry are granted.

To limit the access rights of a mail reader executing a computational e-mail message on behalf of the receiver, one possible way is to have the process give up the receiver's token and then obtain a new token with fewer access rights. Unfortunately, AFS permits each user to carry only one token at a time on a particular machine. This implies that, under AFS, all the receiver's processes have the same token-based access rights. Thus, if the user's mail reader gives up the token, the user's other applications on the workstation also lose the token (and the ability to read/write private files). Clearly, such a situation is very undesirable!

[2]Unix is a registered trademark of the Unix Open Foundation, Inc.

```
/*
* Safe-Tcl function that validates that the sender
* and the receiver have the access rights necessary to open
* a file with specified rights. Returns TRUE if both
* the sender and the receiver have the necessary access rights.
*/

valid_rights(sender, receiver, file, rights)
{
   /* Determine if both users have the necessary rights */
   foreach user in {sender, receiver}
      foreach right in rights
         unless (has_right(user, right, file))
         return FALSE;
   return TRUE;
}
```

Figure 4: The valid_rights function

We also considered requiring a sender and a receiver who want to collaborate to create a new user-id that has access to files shared between the two but no access to their private files. Then the mail reader could execute under this new user-id. The problem with this approach is that it requires creation of these additional user-ids for every possible combination of sender-receivers who might collaborate. In most cases, establishing new user-ids involves going through system administrators. We feel that this problem will make this approach difficult to use, if not impractical.

4.2 SAFE-TCL

We next describe an implementation of our security model that uses the Safe-Tcl's untrusted interpreter as its basis. Two capabilities need to be added to the Safe-Tcl's interpreter to implement our security model: (1) Safe-Tcl's I/O functions must be modified to check the access rights of the sender and the receiver and (2) the Safe-Tcl functions that invoke external applications need to be modified to determine whether it is safe to execute an application from a command script.

Prior to performing a read or write operation, the Safe-Tcl functions that read or write to the file system should call our function `valid_rights` shown in Figure 4. The `valid_rights` function checks the access rights of the file (represented using ACLs and/or Unix mode bits), in order to determine whether the sender and the receiver have the necessary access rights to perform the operation.

It is more difficult to determine whether a non-Safe-Tcl application can be started safely from a Safe-Tcl script. This is because once such an application is started, it is outside the control of the Safe-Tcl interpreter. Therefore, the interpreter must be certain that

the application is safe to execute before the application is started. This is not easy to do because it is not feasible to determine what an executable application actually does by examining its object code. Also, seemingly safe applications may provide the ability to start other applications. From Emacs, for example, it is possible to start a shell process, from which other, potentially unsafe, applications can be executed.

To solve this problem, we require that application programmers and/or end users must identify applications that are safe and list ways in which they can be executed safely. For example, if one chooses to classify Emacs as a safe application then one needs to ensure that it is invoked in a manner that it will be actually safe. For example, one may choose to allow the sender to provide only a file name and a line number as input to Emacs. There are other loopholes that have to be closed as well before Emacs can be used safely, however. For example, we cannot let the computational e-mail script edit the `.emacs` file, which determines how Emacs is loaded. The sender may insert some unsafe actions that are run automatically when Emacs is started. Therefore, it is not necessarily trivial to ensure that an application is invoked in a safe manner.

Our implementation of `valid_exec`, a function to determine if an application should be allowed to be executed from a script, is shown in Figure 5. It makes three checks: (1) whether the sender and receiver have the necessary access rights to the application; (2) whether the application is on the list of safe applications; and (3) whether the argument list has a safe structure. If an application fails any of the three tests and it does not appear on a list of definitely unsafe applications, then the receiver is given the option of executing the application anyway. A list of known unsafe applications may be maintained by the system/users for this purpose. For instance, applications such as Unix shells, `rm`, etc., might be classified as being unsafe. Users are strongly dissuaded from running applications that fail the above validation checks, but if they consider an application to be safe, they may still run the script.

The addition of `valid_rights` and `valid_exec` changes the way that the untrusted interpreter is used. Since shared files can be accessed irrespective of where they are located, the need for a special public directory for I/O for this purpose no longer exists. Also, the modified interpreter allows access to applications that have been determined to be safe.

One advantage of using the Safe-Tcl implementation is that it is portable. This enables users to collaborate using Safe-Tcl in a heterogeneous environment. The other two implementations, based on AFS ACL's and on Unix mode bits, require that the appropriate operating system be present.

```
/*
* Safe-Tcl function that validates that an application,
* specified by its complete file name and with
* input arguments "args", can be invoked safely
* from a command script being executed by a receiver.
* Returns TRUE if the application and its args are safe
* or if the receiver permits the application to be run.
*
*/

valid_exec(sender, receiver, application, args)
{

   /* Determine if the application is accessible and safe */
   if (valid_rights(sender, receiver, application, execute) &&
       (safe_appl(application) &&
       (safe_args(application, args))
       execcmds(application, args);
   /* Give the user the option to OK the fn's exec */
   else if (!unsafe_appl(application) &&
           has_right(receiver, execute, application) &&
           (user_ok (sender, receiver, application, args))
       execcmd(application, args);
   else
       reporterr();
}
```

Figure 5: The `valid_exec` Function

4.3 UNIX MODE BITS

File system security in Unix is provided using mode bits. *Mode bits* represent the read, write, and execute privileges of three types of user levels: (1) `owner`; (2) `group`; and (3) `others`. The mode bits representation of access rights is global; that is, every request for access to a file sees the same mode bits value (unless the mode bits are changed, of course). Mode bits cannot be used to represent negative rights.

The access rights of a process are determined by the following values: (1) an *effective user ID* and (2) a set of *supplementary group IDs*. The effective user ID of a process indicates the `owner` access rights of the process; the process can access any file with the same user ID, with permissions specified by the owner mode bits of that file. The supplementary group IDs of a process specifies a set of groups for which the process has `group` access rights; the process can access any file whose group ID belongs to the set, with permissions specified by the group mode bits for that file. Unix provides super-user functions to set the user ID and the supplementary group IDs for any process.

The super-user function `seteuid` changes the effective user ID of a process. We use this capability to deny access to the receiver's files. The super-user function `setgroups` assigns a set of supplementary group IDs to a user process. We use this capability to add back access to files shared between the sender and the receiver.

An interesting characteristic of the Unix mode bits representation is that for any process, its effective user ID and supplementary group IDs are allowed to be inde-

pendent. For example, the super-user can set the supplementary group IDs of a process to include a group that does not contain the effective user ID of the process.

We utilize the flexibility allowed between effective the user ID and the supplementary group IDs to meet the security requirements of our example. We list the pseudo-code for a function called **intersect_rights** in Figure 6. This function, invoked by the mail reader running with **setuid** bit on, enables the mail reader process to set its effective user ID and supplementary group IDs so that it will execute a command script with access rights common to both the sender and the receiver.

In the **intersect_rights** function, first, the **owner** access rights are removed by changing the effective user ID of the mail reader process to a dummy user's ID (e.g., **nobody**). Then, access rights to files to which both the sender and the receiver have **group** access rights are added by resetting the supplementary group IDs of the mail reader process to only those group IDs to which both the sender and receiver belong.

The mapping between user IDs and supplementary group IDs is maintained in the file **/etc/group**. Each supplementary group ID is associated with a list of user IDs that have access rights to that group. Note that it is not necessary to add the dummy user ID to the **/etc/group** file entry for the supplementary group IDs of the process. This is because the super-user function **setgroups** does not require that the user ID of a process appear on the **/etc/group** entries for the supplementary group IDs of the process. Since no special access rights are added for the dummy user, each process can use the same dummy user. The dummy user, the sender, and the receiver all have **others** access rights, so these access rights are unaffected.

We have implemented this function and tested it by starting Emacs processes to edit specific files. The variables in the test cases are: (1) the sender's groups; (2) the files; (3) the file's group; and (4) whether the file has **others** access rights. The receiver is a member of the **scan** and **ssrlroot** groups but not of the **faculty** group. The following four cases were tested: (1) the sender belongs to a subset of the receiver's groups; (2) the sender belongs to a superset of the receiver's groups; (3) the sender and the receiver have no groups in common; and (4) the sender wants the receiver to access a file for which the receiver does not have access rights. In Table 1, we show the test cases variables and the results: (1) the intersection of groups between the sender and the receiver ($\cap$ *groups*) and (2) whether the files are accessed. As can be seen, the results satisfy the requirements of the intersection model.

The Unix implementation of our security model offers the following advantages: (1) it makes the intersection model available to a wide variety of Unix applications and (2) it does not require information about

```
/*
* Super-user function only!!!
* Determine groups that are common between
* sender and receiver.
* Set effective UID of process to nobody to limit
* user rights
* Set groups of process to groups to limit group
* rights to just the input group
* Run cmds using these limited rights
*/


#include<sys/types.h>    /* For setgid */
#include<grp.h>          /* For group fns */


intersect_rights(sender, receiver, cmds)
{
   /* Save old effective UID and groups of process */
   save_id();

   /* Get intersection of users' groups */
   groups = intersect_grps(sender,receiver);

   /* Count the number of groups in the intersection set */
   maxgrp = length(groups);

   /* Set the effective user ID and the group IDs */
   seteuid(nobody);          /* Set eff. uid to nobody */
   i = 0;
   foreach group in groups
      /* Get group entry for each group */
      grent = getgrnam(group);
      /* Get gid for the group from entry */
      group_gids[i++] = grent->gr_gid;
   /* Set groups of process to members of groups */
   setgroups(maxgrp, group_gids);

   execcmds(cmds);           /* Run script commands */

   /* Reset effective UID and groups of process */
   reset_id();
}
```

Figure 6: The **intersect_rights** Function

Sender's Groups	*Files*	*File Group*	`Others` *Rights?*	∩ *Groups*	*Accessed?*
scan	group.file	ssrlroot	No	scan	No
	group.tex	scan	No		Yes
	project-tasks	ssrlroot	Yes		Yes
ssrlroot	group.file	ssrlroot	No	ssrlroot	Yes
scan	group.tex	scan	No	scan	Yes
faculty	project-tasks	ssrlroot	Yes		Yes
faculty	profile.gwm	faculty	No		No
	project-tasks	ssrlroot	Yes		Yes
ssrlroot	profile.gwm	faculty	No	ssrlroot	No
scan				scan	
faculty					

Table 1: Test cases for the function `intersect_rights`. The receiver belongs to the `scan` and `ssrlroot` groups but not to the `faculty` group.

which applications are safe. The Unix function **exec** validates that each application file is accessible before it actually executes the application.

5 VARIATIONS OF THE INTERSECTION MODEL

The intersection model extends the range of applications that can be safely supported using computational e-mail. However, there are scenarios where even this model is not adequate. We discuss potential ways of addressing those situations.

- **Access to some private files**: In some scenarios, a sender may request that a receiver perform an action using a file that is private to the receiver. For example, one programmer may want another programmer to fix a function for which the second programmer is responsible. The first programmer does not have write access to the program code file, so the intersection model will not grant write access to the file.

  One way to provide such access in the Safe-Tcl interpreter is to modify the **valid_rights** function to ask the receiver to approve additional rights before permitting such access. Since the receiver owns the file, access can be added relatively easily using Safe-Tcl. A limitation of this approach is that if the Safe-Tcl interpreter invokes a process that is not written in Safe-Tcl, **valid_rights** and **valid_exec** have no control over the access rights of that process.

  Allowing such access in the implementation based on Unix mode bits appears difficult. The problem is that these permissions are checked on Unix system calls, and those system calls do not ask the user for approval of additional rights when private files are accessed. Unless a mechanism for asking users to approve additional rights upon file access can be designed, giving selective permission to the mail reader to access some private files, but not all, appears difficult.

- **No execute rights**: In this case, a receiver allows a sender's script to read and write data using the access rights of the intersection model, but does not want the computational mail to execute any external applications. This model is just the intersection model with all execute rights removed. The easiest way to do this appears to be to limit access to the **exec** function (and its variants) from the e-mail script.

6 CONCLUSIONS

We defined a security model that permits access to files shared between a sender and a receiver during the execution of a computational e-mail message but that does not compromise the security of private files. We called this model the intersection model. Two implementations are provided for this security model: (1) a Safe-Tcl implementation and (2) a Unix implementation. The AFS's security model was found to be too restrictive to implement the intersection model.

The Safe-Tcl and Unix implementations each have different advantages. The Safe-Tcl solution is more portable and somewhat easier to extend. However, it is difficult to ensure that an executable program started by a Safe-Tcl script is safe. The Unix solution enables both a script and executable programs invoked from it to be run safely. The Unix-based solution, however, is not easily extensible to permit variations of the intersection model. Therefore, we believe that parallel development of security models for collaboration at both the interpreter level and at the operating system level is

necessary. We plan to explore the interactions between the implementations at the two levels in the future.

ACKNOWLEDGEMENTS

We thank the referees for helpful comments. This work is supported in part by the National Science Foundation under the cooperative agreement IRI-9216848.

REFERENCES

[1] N. S. Borenstein. Computational mail as a network infrastructure for computer-supported cooperative work. In *CSCW 92 Proceedings*, pages 67–74, 1992.

[2] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, 1994. Available via anonymous ftp from ics.uci.edu in the file mrose/safe-tcl/safe-tcl.tar.Z.

[3] N. S. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions): Mechanisms for specifying and describing the format of internet message bodies. RFC 1521.

[4] Y. Goldberg, M. Safran, and E. Shapiro. Active Mail – a framework for implementing groupware. In *CSCW 92 Proceedings*, pages 75–83, 1992.

[5] M. Knister and A. Prakash. DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third ACM Conference on Computer-Supported Cooperative Work*, pages 343–355, October 1990.

[6] M. Knister and A. Prakash. Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems*, 6(2):135–166, 1993.

[7] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[8] A. Prakash and H. Shim. DistView: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the Fifth ACM Conference on Computer-Supported Cooperative Work*, October 1994.

[9] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[10] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–21, May 1990.

[11] J. G. Steiner, C. Neumann, and J. J. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Usenix Conference*, pages 191–202, 1988.

[12] J. Sweet. A multi-media e-mail tutorial with MH.

[13] J. E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper.