In Proceedings of the 3rd USENIX Security Symposium, September 1992, pp. 211-226

Specifying and Checking Unix Security Constraints

Allan Heydon[1]
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94313
(415) 853-2142
`heydon@src.dec.com`

J. D. Tygar[1]
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-6340
`tygar@cs.cmu.edu`

Abstract

We describe a system we have built for specifying and checking security constraints. Our system is *general* because it is not tied to any particular operating system. It is *flexible* because users express security policies in a formal specification language, and it is easy to extend or modify a policy simply by augmenting or changing the specification for the current policy. Finally, our system is *powerful* enough to check for many relations on the current configuration of a file system; however, it is not powerful enough to check for more subtle security holes like Trojan horses or weaknesses in the passwords chosen by the system's users.

We show how to represent various Unix security constraints — including those described in a well known paper on Unix security [GM84] — using our specification language. We then describe the results we obtained from running our tools to check an actual Unix file system against these constraints.

1 Introduction

An important security task faced by any system administrator is that of formulating and enforcing a security policy. One example of a security policy was proposed by Bell and LaPadula [BL73]. In their model, each user and file is assigned a linear security level (e.g., top secret, secret, not secret); roughly speaking, it is only acceptable for users to write files at their security level or higher and to read files at their level or lower. If we could specify such a policy and then run a program to check a file system against it, then we could easily detect security holes on that file system.

We are immediately faced with two problems: that of developing a language in which to formulate such security policies, and that of developing algorithms to automatically check that some specified policy is not violated. Ideally, we would like to provide a policy specification and checking system that is *general*, *flexible*, and *powerful*. First, the system should be general enough to allow and understand policy specifications for different operating systems. Second, it should be flexible enough to allow extensions and modifications to an existing policy. If a system administrator finds a new security hole for which she would like to check, she should be able to do so easily, without having to write a special-purpose program to check for that hole. Finally, the system should be powerful enough to detect any security hole we might want to specify.

[1]This research was sponsored in part by the National Science Foundation under a Presidential Young Investigator Award, Contract CCR-8858087. It was also supported in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The members of the Miró project at Carnegie Mellon have developed a security specification system that meets these goals [HMT$^+$90]. The Miró system is general because it is not tied to any particular operating system. It is flexible because users express security policies in a formal specification language, and it is easy to extend or modify a policy simply by augmenting or changing the specification for that policy. In addition, our policy specification language might be used to configure existing security tools such as the Integrated Toolkit for Operating System Security (ITOSS) [RT87]. Finally, the system is powerful enough to check for many relations on the configuration of the file system; however, it is not powerful enough to check for more subtle security holes like Trojan horses or weaknesses in the passwords chosen by the system's users.

Our system has been implemented on Unix, and one of its components is built using the Garnet user interface management system [Mye89], which runs on X windows. Not only is our system real, but it is also practical. Throughout the design and construction of our system, we have stressed algorithmic efficiency, so the system runs quickly and is effective at catching policy violations. We describe the tools comprising our system in Section 3.

This paper is a case study. It shows how the Miró security checking tools can be used to specify a Unix security policy and to check that policy against an existing Unix file system. Some of the security constraints we examine are taken from previous Miró papers. However, most have been simply "transcribed" from textual descriptions found in a well-known paper on Unix security by Grampp and Morris [GM84]. Hence, one important aspect of this paper is that it demonstrates the utility and expressive power of our policy specification language.

Although the security constraints described here are written for the Unix operating system, we want to stress that the Miró specification languages described in Section 2 can be applied to operating systems other than Unix. Also, as opposed to security systems like COPS [FS91] or U-Kuang [Bal88], the power of the Miró system derives from the ease by which it allows users to express and check new security constraints.

2 The Miró Languages

We focus on two different aspects of the security specification domain. First, we use the *instance language* to specify security *configurations*, that is, fixed access relationships between users and files on a file system. Since these specifications can be both read and written, they give users the ability to determine the access rights granted on their files and to modify those rights. The semantics of a configuration specification is a Lampson access matrix [Lam71], which specifies for every user and file whether access for that user on that file is granted or denied for each access permission.

Second, we use the *constraint language* to specify security *policies*. The constraint language is a meta-language of the instance language, since the semantics of a security constraint is simply a *set* of configurations. A policy is specified by a set of constraints, $c_1, \ldots, c_n$. If each constraint represents the set of configurations $C_1, \ldots, C_n$, respectively, then we say a particular configuration is *consistent* with the policy if it is a member of the set $\bigcap_i C_i$, i.e., if it is in each of the configuration sets represented by the constraints comprising the policy.

In this section, we describe the instance and constraint languages by example, so that the constraints described in Section 4 will make sense to the reader. Both languages use a visual notation that borrows heavily from the higraphs proposed by David Harel [Har88]. The detailed syntax and semantics of both languages are described elsewhere [Hey92].

2.1 The Instance Language

The vocabulary of the instance language consists of rectangular boxes and of arrows labeled with access permissions. Boxes represent users and files; they also group users and files to form a hierarchy. Arrows are either positive or negative; they denote the granting or the denial of access rights, respectively. An "X" through an arrow *denies* the access rights denoted by the arrow.

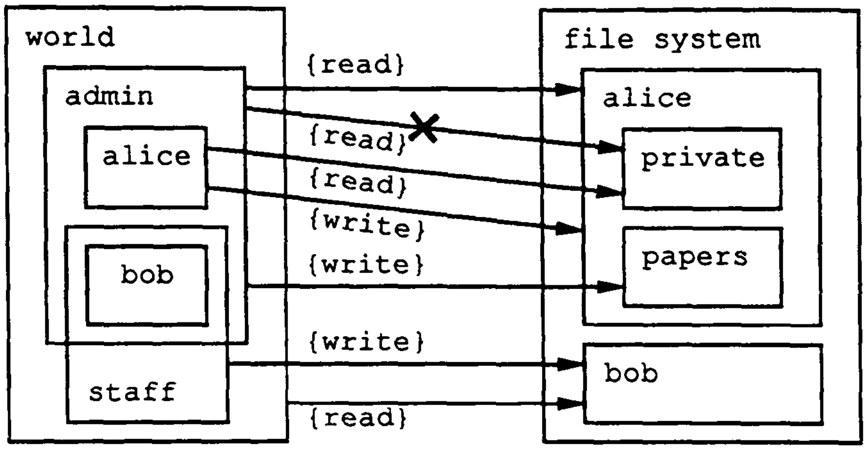

Figure 1: A Simple Instance Language Picture.

Consider Figure 1, which shows a simple instance language picture. Reading the arrows from top to bottom, this picture specifies that: 1) every user in the `admin` group can `read` all of Alice's files, except for those in her `private` directory (which she alone can `read`), 2) Alice can `write` all of her files, 3) all users in the `admin` group can `write` the files in Alice's `papers` directory, 5) all users in the `staff` group can `write` all of Bob's files (including Bob), and 4) all users can `read` all of Bob's files.

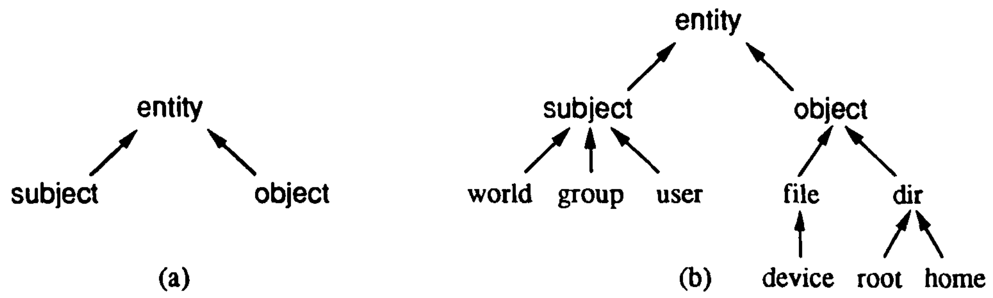

Figure 2: The Built-In Box Types (a) and a Full Box Type-Tree for Unix (b)

One important property of each box in an instance picture is its *type*. A Miró user can define an arbitrary type tree to suit her particular system and security policy. Figure 2(a) shows the three types built into the Miró system, and Figure 2(b) shows how this tree might be extended to accommodate Unix. Each type specification includes a set of attributes associated with that type. For example, the `entity` type includes a Boolean valued attribute named `atomic`; this attribute is true for a box iff that box contains no other boxes. Since the `entity` type is at the root of the type tree, every box inherits the `atomic` attribute. We can also specify attributes to be associated with types we have added to the type tree. For example, to accommodate Unix, we can specify that a Boolean valued setuid attribute is associated with the file type. As we shall see in the next section, types are used primarily in the specification of constraints.

2.2 The Constraint Language

A picture drawn in the constraint language — henceforth called a *constraint picture* or simply a *constraint* — specifies a (possibly infinite) set of instance pictures. Each constraint picture can be thought of as a *pattern* for instance pictures, just as a regular expression is a pattern for character strings. We now briefly describe the syntax and semantics of the constraint language; we hope to illustrate the semantics primarily by example.

The building blocks of the constraint language are *box patterns*. A box pattern is denoted by a rectangle like an instance box, but it contains a Boolean predicate written in a simple *box predicate language* instead of a simple name (see Figure 3(a)). An instance box b matches a box pattern with predicate α if the values of b's attributes, when substituted for the corresponding attribute names in α, make α true. The box predicate language also provides a mechanism to require relationships between instance boxes matching two different box patterns. Box predicate *variables* (denoted by identifiers preceded by "$") allow users to require that some attributes of instance boxes matching two or more box patterns are identical. For example, we can specify that the name of some user matches the owner of some file.

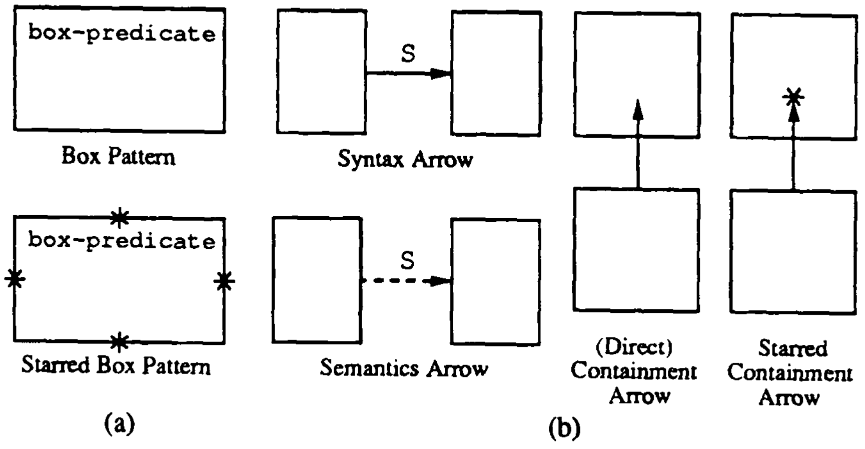

Figure 3: Renderings of Box Patterns (a) and Constraint Arrows (b)

The constraint language includes three kinds of arrows, each of which may be negated as in the instance language (see Figure 3(b)). The two arrows we use most in constraints are the *semantics arrow* and the *containment arrow*. The former are labeled with access permissions just like instance arrows. Two instance boxes b_1 and b_2 match box patterns connected by a positive (negative) semantics arrow labeled with permission p if b_1 has (does not have) access permission p on b_2. Boxes b_1 and b_2 match box patterns connected by a positive (negative) containment arrow if b_1 is (is not) directly contained in b_2. As shown in Figure 3, there are starred variants to both box patterns and containment arrows; these allow us to express containment at any level.

The constraint language presented so far only allows us to require the existence of certain

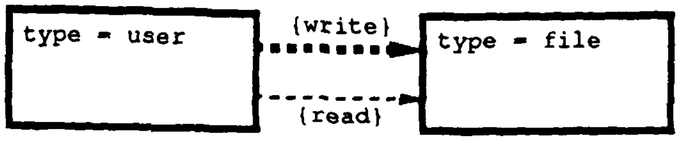

Figure 4: The Constraint WRITE-READ. This constraint requires that a user can read a file whenever he/she can write that file.

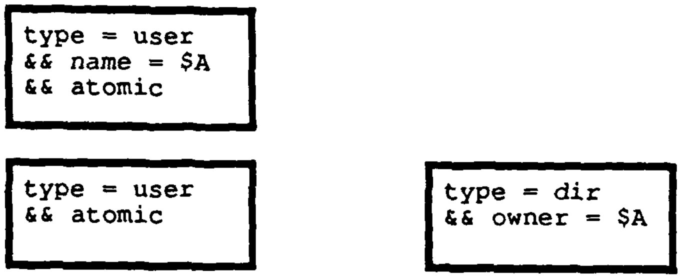

Figure 5: Exploiting Injective Mappings.

entities and relationships between those entities. However, typical security requirements are often conditioned on the existence of some situation. The constraint language provides the power to express such conditional constraints: constraint elements drawn with thick lines represent the antecedent of an implication, while those drawn in thin lines represent its consequent. For example, the constraint named WRITE-READ shown in Figure 4 is interpreted as follows. The thick part of the constraint matches *any* user/file pair such that the user has `write` permission on the file. The thin part of the constraint then requires (as the consequent of the implication) that the user also has `read` permission on the file. In summary, this constraint states that "`write` permission implies `read` permission".

In any given matching between instance boxes and constraint box patterns, no two box patterns may be matched with the same instance box. Thus, if we have two box patterns matching `user` boxes, and the first pattern requires that the `user` box matching it has a particular `name`, as shown in Figure 5, then we can conclude that any `user` box matching the second pattern does *not* have the same `name` as the box matching the first pattern. Hence, by the variable "$A", the `owner` of any box matching the directory box pattern on the right will always differ from the `name` of any box matching the user box pattern on the lower left[2]. Several of the constraints we present in Section 4 exploit this one-to-one property of the matching semantics. We use this example involving two "user" box patterns because it is the only one appearing in our constraints. It would be more straightforward to write such constraint using constructions like "`name` $\neq$ `$A`", but our box predicate language currently does not allow "$\neq$".

3 The Miró Software System

Figure 6 shows the software tools comprising the Miró system and their inter-relations [HMT+89]. We classify the tools (and languages) as either *front end* or *back end* components. The front end components are designed to work independent of any operating system, while the back end components depend on the particular details of the file system

[2]We are assuming here that no two `user` boxes have the same `name`.

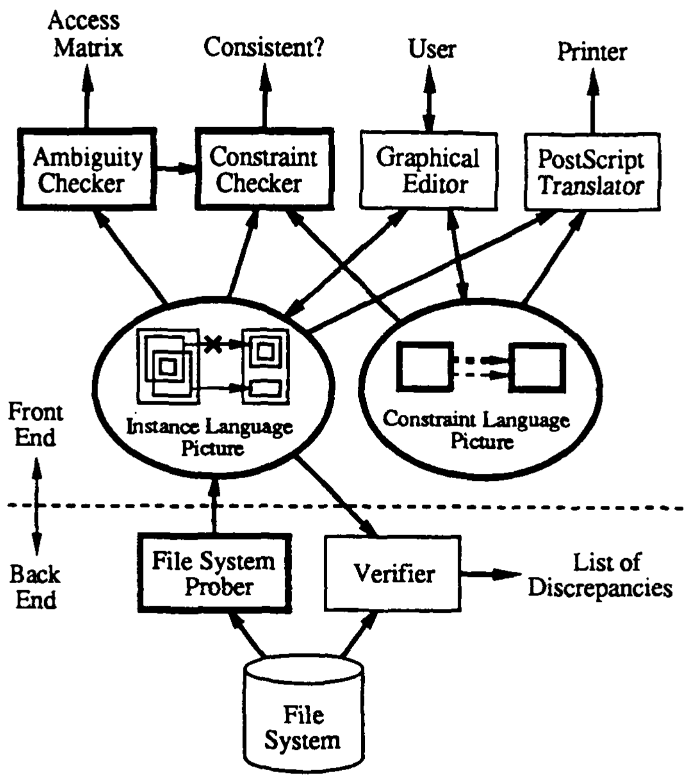

Figure 6: The Software Tools and Languages Comprising the Miró System.
The tools and languages relevant to this paper are shown with a thick outline.

with which they interact. To check file systems other than Unix file systems, we would only have to reimplement the back end tools.

The *graphical editor* allows users to draw and edit instance and constraint pictures, and the *PostScript translator* produces PostScript programs to render these pictures on a printer. Once a user has drawn an instance picture, she can feed it to the *ambiguity checker* to check that it is well-formed, and to generate its corresponding access matrix. This access matrix and the instance itself can then be fed to the *constraint checker* along with some constraint picture to check if the instance is consistent with the constraint. The constraint checker works by modeling the instance picture as a special kind of database and compiling the constraint picture into a program that queries that database [Hey92].

The tools described so far work independent of any file system. To interact with a Unix file system, we provide the back end *prober* and *verifier* tools. The prober searches some subtree of a Unix file system and produces an instance picture with the same structure and security relationships as that file system. The verifier compares a given instance picture to a file system and produces a list of discrepancies between the two.

To perform the experiments described in this paper, we used both front- and back-end tools. We first drew our constraint pictures using the graphical editor. We then used the file system prober to produce an instance picture corresponding to the /usr0 directory of one of the file systems at CMU. Next, we fed that instance picture through the ambiguity checker to generate its corresponding semantics (an access matrix). Finally, we fed the access matrix, the instance picture, and each of out constraint pictures to the constraint checker to determine if the file system was consistent with each constraint.

4 Unix Constraints

In this section, we describe the constraints we use in Section 5 to evaluate the performance of the constraint checker. We have adapted some of these constraints from original constraints suggested in previous Miró papers [HMT+90, HMT+90]. The others were suggested in the Grampp-Morris article referred to earlier; we have simply translated their written security suggestions into constraint pictures.

4.1 Miró Security Constraints

The constraints suggested by previous Miró papers are designed fulfill a variety of needs. Some guarantee "well-formedness" properties of any instance picture, some enforce various containment relationships relative to the box type system, and others are general security constraints for Unix. Here, we present some representative constraints from a previous Miró paper [HMT+90].

4.1.1 GRP-IN-1-W, GRP-IN-W-ONLY, W-IS-ROOT

The constraints shown in Figure 7 place restrictions on the nesting of group and world boxes. Constraints (a) and (c) introduce a new aspect to the constraint language syntax and semantics we did not describe earlier. We may associate an integer-valued interval with each constraint. For each matching to the thick part of the constraint, we *count* the number of consistent extensions to the thin part of the constraint, and that number must fall in the specified interval. If no interval is specified, the default is "[1, ∞]"; this interval corresponds to the original semantics we described in which for each thick matching, there must exist (i.e., be at least 1) consistent thin matching.

The constraint named GRP-IN-1-W (a) requires that every group is contained in exactly one world. However, it is still possible that a group could be contained in a box other than a world. The GRP-IN-W-ONLY constraint (b) therefore requires that every box containing a group must be a world. Finally, the W-IS-ROOT constraint (c) requires that a world is contained in no other box. The negative nature of this constraint stems from its [0,0] integer range: an instance is consistent with the constraint only if there are *no* thin extensions to each thick matching.

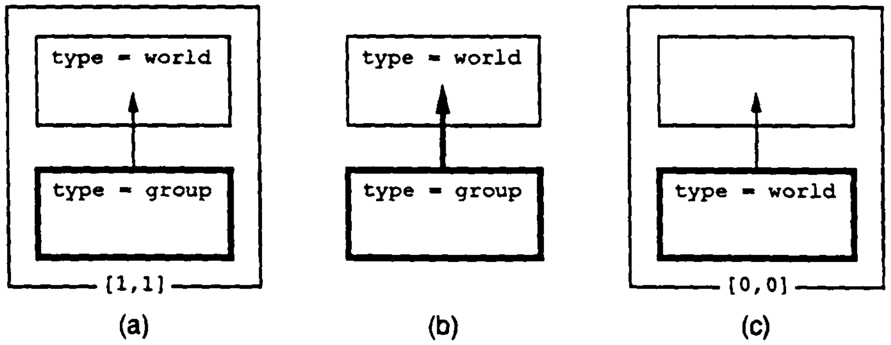

Figure 7: The Constraints GRP-IN-1-W, GRP-IN-W-ONLY, and W-IS-ROOT. These constraints require that each group box is contained in exactly one world box, that each group box is contained in world boxes only, and that world boxes are contained in no other boxes, respectively.

4.1.2 PRIVATE-MAIL

The PRIVATE-MAIL constraint is shown in Figure 8. This constraint assumes that mail systems organize each user's mail files in a certain way. Each user's mail is stored in a subdirectory of their home directory called "Mail". That directory contains subdirectories that partition the mail into categories, and the actual mail files themselves (one file for each mail message) reside in those subdirectories. For each user whose mail is organized in this manner, the PRIVATE-MAIL constraint checks that no one besides the owner of the mail files can actually read them.

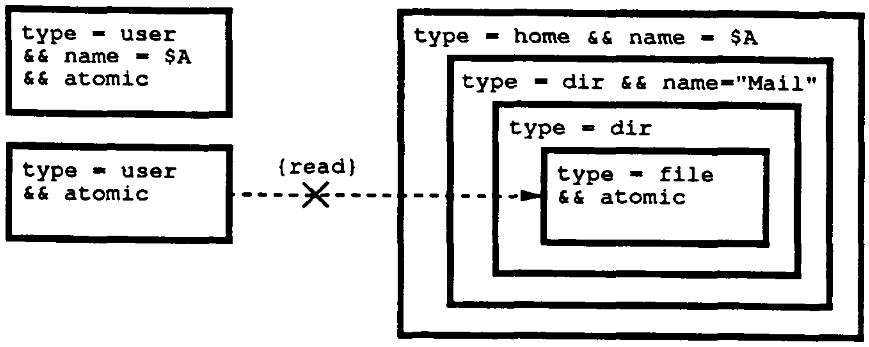

Figure 8: The Constraint PRIVATE-MAIL. This constraint requires that a user's mail files cannot be **read** by anyone except them. It assumes that mail files are stored in subdirectories of a directory named "Mail" in each user's home directory.

This is the first constraint we have encountered that exploits the one-to-one nature of the mapping function described in Section 2.2. In Figure 8, the **user** boxes matching the two "user" box patterns on the left must be distinct for each thick matching. This guarantees that the **user** matching the bottom "user" box pattern will not have the same **name** as the **home** box matching the outermost "home directory" box pattern on the right.

4.2 Grampp-Morris Security Constraints for Unix

Grampp and Morris have described several possible attacks on the security of a Unix system. They point out that most security attacks can be thwarted by educating users and by ensuring the "existence of administrative procedures aimed at increased security". In regards to their first point, users need to be taught the importance of choosing good passwords, and they need to be educated fully as to the security mechanisms they are using so that: 1) they can use those mechanisms to protect their files as they see fit, and 2) they do not inadvertently leave any files unprotected. As to their second point on administrative procedures, it is precisely this sort of capability that systems like the constraint checker provide.

Even if these points have been addressed, there can still be security lapses. Grampp and Morris go on to describe security holes that may crop up on a Unix system. We have "transcribed" some of their descriptions into the following constraint pictures.

4.2.1 PASSWD-SAFE

Unix uses passwords as its only barrier to unauthorized access; if a user's password is compromised, then an intruder can act as that user with impunity. We must therefore guarantee that passwords are adequately safeguarded.

Unix stores encrypted passwords in a world-readable file called **/etc/passwd**. The reasoning behind making the password file world-readable is that "concealment is not security": if we were to instead make the password file unreadable, we would be relying on the security of that one file to protect our entire system. Instead, we rely on the one-way nature of the encryption function; since it is an abstract entity, it is less vulnerable to attack.

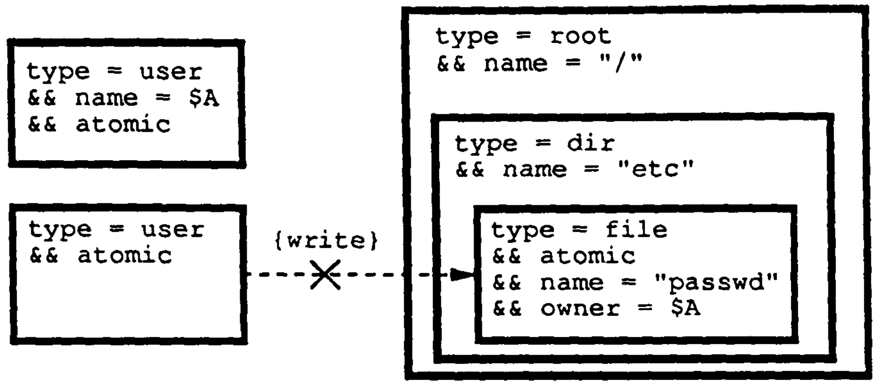

Figure 9: The Constraint PASSWD-SAFE. This constraint guarantees that only the **owner** of the password file **/etc/passwd** can change that file.

However, the password file certainly should not be writable by anyone except its owner (the super user), or else someone could install the encrypted version of a known word under some other user's entry in the file. The constraint PASSWD-SAFE shown in Figure 9 requires that no user can **write** to the password file except its owner. Like the PRIVATE-MAIL constraint of Figure 8, this constraint uses two separate "user" box patterns to ensure that only non-owners of the **file** in question are matched to the bottom "user" box pattern.

The unfortunate truth of the matter is that, even if the password file is protected according to the PASSWD-SAFE constraint, Unix passwords are easily compromised in practice. The simple reason for this weakness is that users tend to choose their passwords poorly. As Grampp and Morris point out, it is not difficult to write a *password cracker* program that guesses possible passwords for each user, and then encrypts each guess for a possible match against the encrypted password stored in the **/etc/passwd** file.

4.2.2 WRITABLE-DIR

On Unix, every file and directory has an associated set of protection bits that specify who may access that file or directory for each relevant access type. Grampp and Morris point out that on Unix, "underlying directory permissions can adversely affect the safety of seemingly protected files". In particular, a user u may have the ability to change a file f, even if f's protection bits specify that u is denied write access on f. How is this possible? Suppose that f resides in a directory d, and that d's protection bits grant write permission to u. That means that u can create and delete files in d. So u can change f by deleting the original f and then creating a new version of f in d. In this way, u can change f's contents to be anything she pleases.

Naive users are especially likely to be unaware of this Unix protection feature. The fact that none of f's write protection bits is on would seem to imply that the file cannot be changed. But the writable directory in which f resides gives u the power not only to change

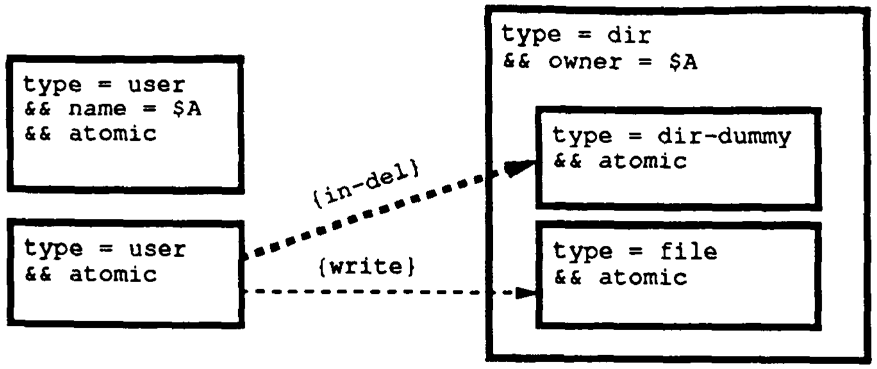

Figure 10: The Constraint WRITABLE-DIR. This constraint specifies that whenever a user has write (i.e., `in-del`) permission on a directory she does not own, she should also have explicit `write` permission on every file in that directory.

the file, but to delete it entirely! We clearly need a constraint to detect occurrences of this situation. However, if we translate this constraint directly, it may be overly sensitive. It is not uncommon for users to give themselves write permission on a directory they own, but to also explicitly deny themselves write permission on some of the files in that directory (to prevent them from being changed accidentally). For example, this situation arises in the use of the RCS version control system, which automatically turns off write permission on files that have not been explicitly "checked out" for modification.

Thus, the constraint we wish to express is that: "For any directory on which a user has write permission and which they do not own, they must have explicit `write` permission on all files contained in that directory". This constraint is shown in Figure 10. There are several points we should make about this constraint.

First, it is the first constraint we have seen so far that involves permission on a directory. Even though Unix overloads the protection bits on files and directories, our Unix prober distinguishes `write` and `read` permissions on files from those on directories. On directories, the prober instead generates the permissions `in-del` (insert-delete) and `list`, respectively.

Second, since permissions granted on a directory are completely unrelated to those granted on the directory's parent, it would be difficult for the prober to represent access relations on directories directly. The prober therefore takes the following simple approach. For each directory box, it installs a special atomic "dummy" box inside that directory box, and it draws arrows to the dummy box so that the access relations on the directory in the file system are represented in the instance by the relations between users and the unique dummy box inside that directory. The prober also gives the dummy box a `type` of `dir-dummy`; this new type becomes a child of the `object` type in the type-tree.

4.2.3 SETUID-SAFE

Many Unix security flaws arise from the *set-userid*, or "setuid", facility. This feature of the Unix protection semantics is a powerful tool, and it allows people to create systems that would be difficult to create otherwise. But as Grampp and Morris point out, "the feature is by no means tame". They suggest that setuid programs should only be used as a means of last resort, since each setuid program introduced onto the system is a potential security hole.

Grampp and Morris also state that "setuid programs that are writable by anyone should

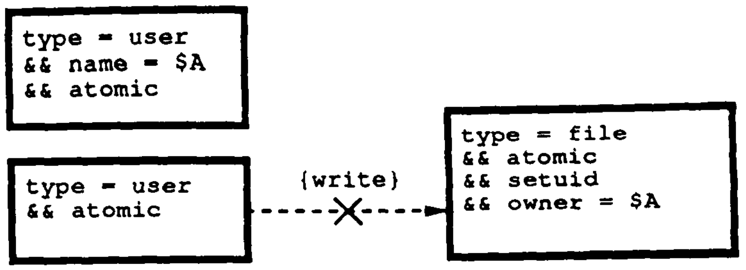

Figure 11: The Constraint SETUID-SAFE. This constraint guarantees that no **setuid** program is writable by someone other than its **owner**.

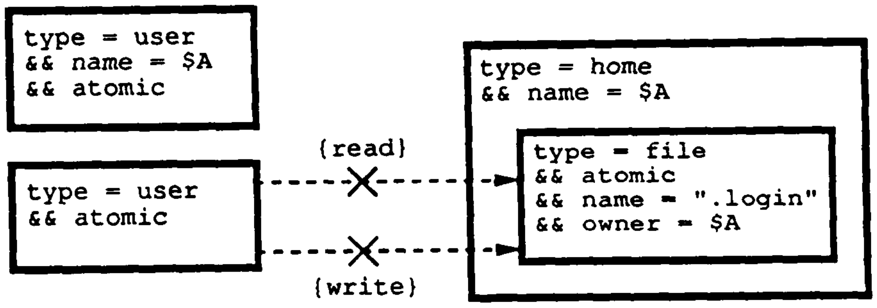

Figure 12: The Constraint LOGIN-SAFE. This constraint guarantees that each user's ".login" file cannot be read or written by anyone but them.

be considered threatening". The reason is that any user can write a copy of the shell, for example, onto the setuid program. That user can then run the newly copied shell; since it is a setuid program, it will be running as its owner. This bit of subterfuge thus gives a malicious party the power to impersonate the owner of the writable setuid program.

The SETUID-SAFE constraint shown in Figure 11 reports any setuid program writable by someone other than its owner. The prober makes the **setuid** attribute true of any instance box corresponding to a file on the file system whose setuid bit is turned on. This constraint again exploits the one-to-one requirement on the mapping to guarantee that the name of any **user** matching the bottom "user" box pattern is not the same as the **owner** of the setuid program matching the "file" box pattern.

4.2.4 LOGIN-SAFE

When a user logs in and/or starts a new shell, Unix automatically executes certain shell "scripts" in that user's home directory. For example, at login, the system executes the file named ".login". Suppose user u's ".login" file is writable by some other user u'. Then u' is free to edit the ".login" file at will. With this power, u' can edit the script to make a copy of the shell (in some directory private to u'), turn on the setuid bit of that copy, and make it world-executable. Since these commands will be executed when u logs in, the shell copy is owned by u. Thus, once u logs in and unwittingly creates a copy of the shell owned by him, u' can execute that copy and impersonate u.

This example clearly illustrates that scripts such as ".login" should never be writable by anyone but their owners. The LOGIN-SAFE constraint shown in Figure 12 tests for this condition. The thick part of the constraint matches two distinct users and every file named ".login" contained in a home directory. As before, we exploit the semantics of the constraint language to guarantee that the instance box matching the bottom "user" box pattern does not have the same name as the instance box matching the "home directory" box pattern. The thin negative `write` arrow then requires that that user does not have write access on the ".login" file.

Grampp and Morris also recommend that such files should not be readable by anyone but their owner either. Their reasoning is that the ability to read these files allows an intruder to determine the user's search path, thereby giving him clues as to candidate directories for trojan horses. The LOGIN-SAFE constraint therefore also includes a thin negative `read` arrow to enforce this requirement. However, as Grampp and Morris point out, the enforcement of this aspect of the constraint "offers little additional protection, as vulnerable [search path] components can be deduced in other ways".

5 Constraint Checking Results

The constraint checker's payoff is its ability to find security holes. Even our simple experiments uncovered some problems. If we had performed more comprehensive experiments, we may very well have found more. Instead, our focus was on gathering measurements of the constraint checker's performance.

To perform our tests, we applied our Unix prober to the `/usr0` subtree of a mainframe VAX at CMU. The instance picture produced by the prober contains 46 groups, 147 users, 677 directories, 5,195 files, and a total of 17,614 arrows. The access matrix produced by our ambiguity checker tool on this instance picture contains a total of 2,490,033 entries.

We then checked this instance with respect to each of the constraint pictures described in Section 4. The constraint checker models the instance picture (and its access matrix) as a database. It first compiles the constraint picture into a query program over this database; this compilation usually takes less than a second. To check the constraint, the checker executes the query program. We divide the time required to execute the query program into two parts: the time to load the instance database for that query, and the time required to perform the query itself.

Table 1 shows the times required for these two phases on each of the constraints. These experiments were performed on a DECstation 3100 running the Mach operating system. We have also run these constraints against other subtrees of the same file system, such as `/etc`, `/dev`, and `/sys0`. From this table, we see that most constraint checks required 1 or 2 minutes of CPU time. The notable exceptions were the WRITE-READ and WRITABLE-DIR constraints. These constraints took longer simply because there were more ways to match the thick part of the constraint to the instance, so there were more cases to check. In general, we have shown that the constraint checking problem is Π_2^p-hard [Hey92].

Our experiments uncovered some constraint violations, which we summarize here. The number of violations we report in each case is the number of thick matchings of the constraint to the instance such that no thin matchings existed. The checker has the tendency to produce voluminous output. We could easily implement simple filters to help solve this problem, but we suspect there may be more sophisticated and flexible solutions.

5.0.1 PRIVATE-MAIL — 438 Violations

The diagnostic output for this constraint/instance pair illustrates the problem with the constraint checker's verbosity. The 438 violations reported in this case amount to a total

Constraint Name	# Elts.	CPU Seconds
GRP-IN-1-W	$1+2=3$	30.8 / 0.1
GRP-IN-W-ONLY	$2+1=3$	19.4 / 0.1
W-IS-ROOT	$1+2=3$	20.8 / 0.0
WRITE-READ	$3+1=4$	39.8 / 194
PRIVATE-MAIL	$9+1=10$	⟨ 41.6 / 51.4 ⟩
PASSWD-SAFE	$7+1=8$	[31.2 / 0.0]
WRITABLE-DIR	$8+1=9$	⟨ 41.9 / 1,300 ⟩
SETUID-SAFE	$3+1=4$	[23.3 / 0.0]
LOGIN-SAFE	$5+2=7$	⟨ 58.6 / 71.3 ⟩

Table 1: Constraint Checker Running Times (in seconds). Values in the "# Elts." column indicate the number of thick, thin and total elements, respectively, in the constraint for that row (including implicit containment arrows). An entry of the form x/y in the row labeled with constraint C means that in the process of checking constraint C, it took x CPU seconds to load the instance database, and y CPU seconds to check the constraint. Entries in square brackets indicate that the query time y is trivial, while those in angle brackets indicate that the corresponding instance is *inconsistent* with the corresponding constraint. Inconsistencies indicate potential security holes.

of only *three* world readable mail files. Since there are 147 users on the system, 146 users were found to be able to **read** each of these three files when they should not have.

The three files found by the constraint checker are not mail messages *per se*. One of the mail systems at CMU keeps an index of messages in each mail directory. The index summarizes the mail messages in that directory, including who sent the message, when it was sent, and the subject line of the message. Even this summary information may be considered sensitive by some users. Upon closer inspection, two of the three index files were also found to be writable by someone other than their respective owners!

5.0.2 WRITABLE-DIR — 26,435 Violations

Obviously, there were too many violations in this case to enumerate them all. However, we can summarize some of the major security holes we found. One user alone accounts for many hundreds of the violations. This user has left many of his directories writable to members of the **theory** group. Since this group includes 25 of the machine's users, giving such a large number of people the ability to delete and overwrite files at will seems dangerous. We surmise from the names of the vulnerable directories that some of them probably contain sensitive files. In many cases, these same files were also readable by all members of the **theory** group. Perhaps most surprising is that one of this user's mail directories is writable to the same group of people.

Perhaps the most serious security hole detected by this run of the constraint checker is the protection on a directory containing bulletin board files. The protection bits on this directory designate it to be *world* writable. Thus, *any* user on the system can overwrite or delete any of the bulletin board files in the directory. These bulletin board files are read by a large number of users, so this is a serious threat. Moreover, since any user can also

;ad these files, a malicious user could easily make subtle changes to any of the files, and it would be impossible to track down the culprit. It is worth noting that since this directory contains over 100 files, and since there are approximately 150 users on the system, this one security hole is responsible for approximately 15,000 of the reported violations.

5.0.3 LOGIN-SAFE — 2,190 Violations

Almost all of the reported violations were due to the fact that nearly every user on this system has a world-readable ".login" file (thus, the number of violations reported is approximately the square of the number of users). However, when we removed the negative `read` semantics arrow from the constraint, we found only 24 violations. Without the negative `read` arrow, this constraint only detects ".login" files that are writable by people other than their owners. Violations of this more restricted constraint are much more serious. On this particular system, one user has given `write` permission to the members of the `theory` group on his ".login" file. These 24 people have the ability to maliciously alter his ".login" file and to install code to impersonate him at will.

6 Conclusions

Specifying and manipulating security specifications is not a toy problem. It is a real problem faced by anyone sharing a computer system with other users. Our constraint language and its associated compiler and run time system provide a mechanism unlike any other of its type to solve this problem. The primary advantages it offers over existing tools are its operating system independence and the means by which it allows users to easily tailor a security policy to their needs. Moreover, the constraint language also gives users the power to specify abstract security models. It can thus be used to drive or configure other security modeling tools.

This work can be extended to solve other problems. For example, as it is implemented now, the Miró system we have described is a static security checker. However, our techniques could be used to implement an *automatic* file system security checker that continuously monitors the file system for security holes. To make such a system practical, we would have to modify our algorithms to interpret incremental changes to the file system or to the security policy.

The work we have described in this paper is a specific application of a general approach. Our approach has been to design formal specification languages for a particular domain in the area of computer systems, and then to build efficient algorithms to process those specifications. Our success in the application of this technique to the domain of file system security leads us to believe that it holds promise for other domains, such as network security, parallel algorithm design, and computer systems management.

7 Where to Get More Information on Miró

If you have any questions about the Miró project, you can send e-mail to the address `miro@cs.cmu.edu`. There is also a video tape available that summarizes the results of the Miró project, and shows each of the Miró software tools in use. The video is approximately 20 minutes long, and is available for $15 within the U.S. and $17 internationally (these prices include first class shipping and handling). Requests for the video should be addressed to: Industrial Affiliates Office, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213. Checks should be made payable to "Carnegie Mellon University".

The first author's Ph.D. thesis [Hey92], as well as the sources for the Miró tools, are available for anonymous FTP from the site named "`ftp.cs.cmu.edu`" (the IP address is [128.2.206.173]). When you FTP to this host, use the login name "anonymous" and give your full e-mail address as the password. Then, type "`cd /afs/cs/project/miro/ftp`" (security restrictions in the FTP server will not allow you to "cd" into any intermediate directory along the path, so you must type this command as shown). Once in this directory, you can type "`get ftp-instructions`"; this will copy an ASCII text file to your machine that explains how to get the thesis and/or software.

8 Acknowledgements

We would like to thank Amy Moormann Zaremsky and Jeannette Wing for their comments on portions of this draft. We would also like to thank Karen Kietzke for her help and patience in testing the ambiguity and constraint checkers.

References

[Bal88] Robert W. Baldwin. *Rule Based Analysis of Computer Security.* PhD thesis, MIT, Cambridge, MA 02139, March 1988. Tech Report MIT/LCS/TR-401.

[BL73] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations (3 Volumes). Technical Report AD-770 768, AD-771 543, AD-780 528, The MITRE Corporation, Box 208, Bedford, MA 01731, November 1973.

[FS91] Daniel Farmer and Eugene H. Spafford. The COPS Security Checker System. Technical Report CSD-TR-993, Purdue University, West Lafayette, IN 47907-2004, 1991.

[GM84] F. T. Grampp and R. H. Morris. Unix Operating System Security. *AT&T Bell Laboratories Technical Journal*, 63(8):1649–1672, October 1984. Part 2.

[Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[Hey92] C. Allan Heydon. *Processing Visual Specifications of File System Security.* PhD thesis, Carnegie Mellon University, School of Computer Science, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, January 1992.

[HMT+89] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró Tools. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 86–91, Los Alamitos, CA, October 1989. IEEE Computer Society Press.

[HMT+90] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró: Visual Specification of Security. *IEEE Transactions on Software Engineering*, 16(10):1185–1197, October 1990.

[Lam71] B. W. Lampson. Protection. In *Proceedings Fifth Annual Princeton Conference on Information Science Systems*, pages 437–443, 1971. Reprinted in *ACM Operating Systems Review*, Volume 8, Number 1, (January 1974), pages 18–24.

[Mye89] Brad A. Myers et. al. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Technical Report CMU-CS-89-196, Carnegie Mellon University, School of Computer Science, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, November 1989.

[RT87] M. Rabin and J. D. Tygar. An Integrated Toolkit for Operating System Security. Technical Report TR-05-87, Aiken Computation Laboratory, Harvard University, May 1987.