Specifying and Checking Unix Security Constraints

Allan Heydon[1]
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
(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 called Miró 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, so it is easy to extend or modify a policy simply by augmenting or changing the specification for the current policy. Finally, our system is *expressive* enough to describe many relations on file system configurations; however, it is not expressive enough to describe more subtle security holes like Trojan horses or weaknesses in the passwords chosen by the system's users.

This paper is a case study of the Miró languages and tools. We show how to represent various Unix security constraints — including those described in a well-known paper on Unix security [5] — using our graphical 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 [2]. 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 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 *expressive*. 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 existing policies. A system administrator should be able to easily specify a new security hole to check for, without having to write a special-purpose program to perform the check. Finally,

[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. Additional support was received from Motorola, TRW, IBM, and the United States Postal Service.

the system should be expressive enough to describe any security hole we might want to detect.

To meet these goals, a group at Carnegie Mellon developed a security specification and checking system called Miró [10, 7]. The Miró system consists of two languages and a collection of software tools. One specification language is for protection configurations and the other is for security policies.

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, so 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) [16]. Finally, the system is expressive enough to describe many relations on the configuration of the file system; however, it is not expressive enough to describe more subtle security holes like Trojan horses or weaknesses in the passwords chosen by the system's users, since such security threats cannot be thwarted by access control restrictions.

This paper is a case study of the Miró languages and software tools. 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. Hence, one aspect of the case study is to test how well our tools performed on real Unix security policies. Some of the security policies 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 [5]. Hence, the second aspect of the case study is to demonstrate the utility and expressive power of the Miró policy specification languages as applied to a set of well-known Unix security holes.

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 [4] or U-Kuang [1], the power of the Miró system derives from the ease by which it allows users to express and check new security constraints.

Since textual specifications are often plagued by errors [3, 12], we have attempted to develop specification languages that are more intuitive to use so that errors will be less likely. Our languages are primarily graphical, but they mix graphical and textual notations where each is appropriate. For example, we use nested boxes to represent group membership and directory containment, but we use text to represent Boolean formulas. Our graphical notation borrows heavily from the higraphs proposed by David Harel [6]. We believe the use of a graphical notation makes our specifications simpler and more natural to use than equivalent textual specifications.

We describe the tools comprising our system in Section 3. Our system is implemented on Unix, and one of its components is built using the Garnet user interface management system [15, 14], which runs on X windows.[2] Throughout the design and implementation of our tools, we have stressed algorithmic efficiency, so the system runs quickly and is effective at catching policy violations.

In summary, the novel contributions of this work are:

- tools for processing graphical descriptions,
- a library of predefined security constraints for Unix,

[2]Our system is available via anonymous ftp. For details, contact the authors or send mail to `miro@cs.cmu.edu`.

- a real and efficient system for checking file systems against security constraints, and
- security configuration and policy specification languages that are more natural to use.

In Section 4, we present some security constraints for Unix, and in Section 5, we evaluate our tools by measuring their performance on these constraints.

2 The Miró Languages

We address two different aspects of the security specification domain.

First, we use the *instance language* to specify security *configurations*. By a security configuration we mean a set of access relationships between subjects and objects on a file system. In particular, the Miró semantics of a configuration specification is a Lampson access matrix [13], which specifies for every subject and object whether access for that subject on that object is granted or denied for each access permission.[3] Since these specifications can be both read and written, they give users the ability to determine the access rights granted on their directories and files, and to modify those rights.

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 c is simply a (possibly infinite) *set* of configurations C. A policy is specified by a set of constraints, $c_1, \ldots, c_n$. If these constraints represent the sets 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 $\bigcap_i C_i$; that is, if it is in each of the configuration sets represented by the constraints comprising the policy.

The constraint language has two orthogonal uses. First, we can use the constraint language to guarantee that a configuration is *realizable* by the underlying operating system. Since the security mechanisms of any operating system can permit only certain access matrices, some configurations must be disallowed. Second, we can use the constraint language to guarantee that a configuration is *acceptable* according to some security policy. The constraint language's domain of applicability in this second sense is quite large. We imagine that some policies will be written by system administrators and enforced across entire sites, while others will be written by individual users and enforced only on their own files.

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. The detailed syntax and semantics of both languages are described elsewhere [11].

2.1 The Instance Language

The vocabulary of the instance language consists of rectangular boxes and of arrows labeled with access permissions. A typical picture drawn in the instance language — henceforth called an *instance picture* — has its boxes arranged in two separate collections, as shown in Figure 1. The boxes on the left are subjects (users) or sets of subjects, the boxes on the right are objects (files) or sets of objects, and the arrows represent relations between them. We use box grouping to indicate sets, and the language permits us to focus on particular levels of hierarchy in the picture. For example, we do not detail all of Bob's files; we simply use a single box to denote the entire set. A positive arrow indicates that access is permitted, and an arrow with an "X" through it indicates that access is denied.

[3] The particular set of access permissions embodied by an access matrix depends on the security architecture of the underlying file system.

Figure 1 shows a simple instance 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, 4) all users in the `staff` group can `write` all of Bob's files (including Bob), and 5) all users can `read` all of Bob's files.

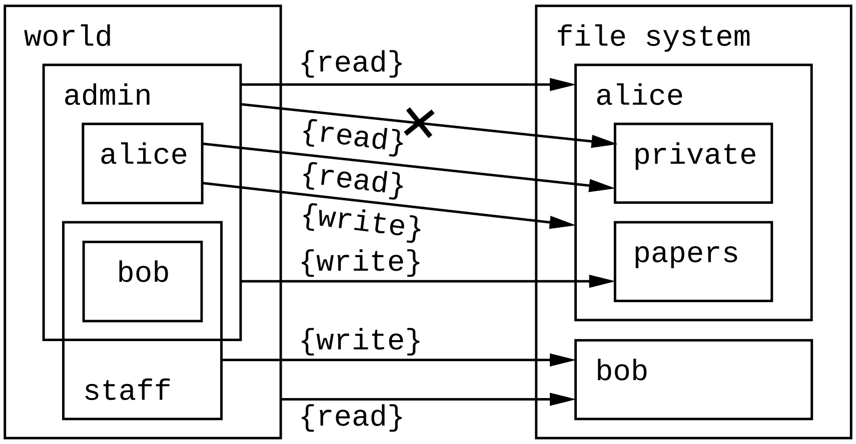

Figure 1: 10A Simple Instance Language Picture.

Often, security specifications are stated in general rules with explicit exceptions, as exemplified by the top three arrows in Figure 1. This pattern of exceptions is what makes security specification difficult — the patterns are complicated, but the exceptions are very important. Users want to know the specific exceptions to general access rules. In Miró, exceptions are indicated by arrows connecting more tightly nested boxes. For example, because the box for Alice's `private` files is nested in the box `alice` for her home directory, the negative arrow from `admin` to `private` takes precedence over the positive arrow from `admin` to `alice`. Negative arrows are not strictly necessary; we can represent any access configuration by sufficiently many positive arrows alone. However, including negative arrows allows us to represent exceptions explicitly and more concisely.

Unfortunately, the addition of negative arrows to the language makes it possible to draw ambiguous pictures. For example, suppose Figure 1 also contained a negative `write` arrow from the `admin` group to Bob's home directory. This negative `write` arrow denies access to members of the `admin` group, but the positive `write` arrow grants access to members of the `staff` group. Since Bob is in both these groups, it is not clear whether he should be allowed `write` access to the files in his home directory or not. We say such pictures are *ambiguous*. The instance language semantics precisely defines for each subject/object/permission triple in an instance picture whether the access relation for that subject on that object for the named permission is positive, negative, or ambiguous.

Two other orthogonal aspects of the instance language are worth mentioning:

Box Layout and Box Aliases: It is not possible to represent all grouping relationships using box containment. Moreover, symbolic links may also make it difficult to lay out boxes corresponding to a Unix file system. The syntax and semantics of the instance language can be extended easily to include the notion of box *aliases*, which would address both of these problems. However, for reasons described in Section 3, we have not yet found it necessary to add box aliases to the instance language.

Correspondence to the File System: Users are not required to draw instance pictures that have a one-to-one correspondence with the file system. For example, if a subset the entries in a particular directory share the same protections, a user can group them by drawing an additional box around them, and then specify the protections for all of the entries in the group by drawing arrows to the grouping box only. This grouping box need not correspond to a directory on the file system; its only function is to group the relevant boxes in the instance picture so as to make the picture more concise and to visually reflect that the grouped boxes share certain security properties.

One important property of each box in an instance picture is its *type*. A Miró user can define an arbitrary type tree to suit a particular system and security policy. Figure 2 shows a sample type tree for Unix. The three types outlined in bold at the top of the tree are built in to the Miró system; users extend the tree by defining new subtypes of either the `subject` or the `object` built-in types.

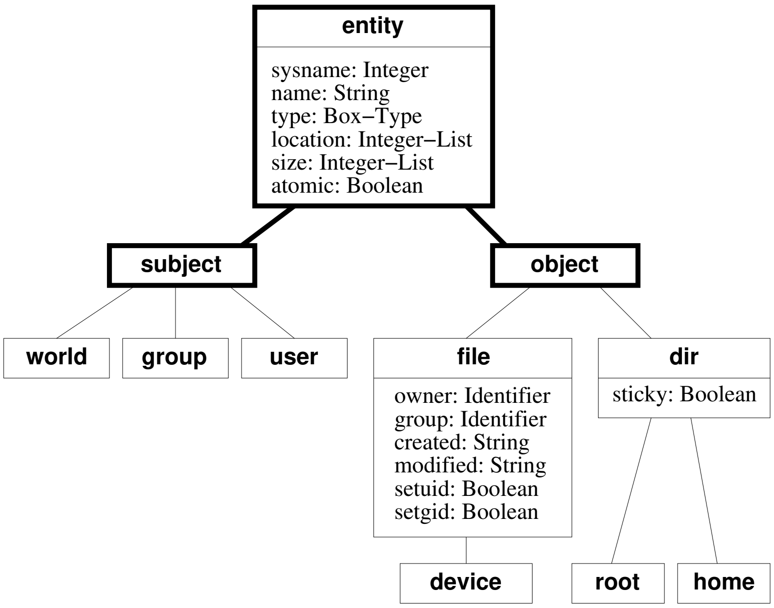

Figure 2: 10A Box Type Tree for Unix

Each type specification includes a set of typed attributes associated with that type, and any box in an instance picture with that type has values for those attributes. 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 type inherits the `atomic` attribute, so every box in an instance picture has a Boolean value indicating whether or not it contains any other boxes. We can also associate attributes with user-defined types. For example, to accommodate Unix, we've specified 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 constraint specifications.

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 using examples.

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 or distinct. For example, we can specify that the `name` of some user differs from the `owner` of some file by writing "`name = $A`" in the box predicate of one box pattern and "`owner # $A`" in the box predicate of the other.

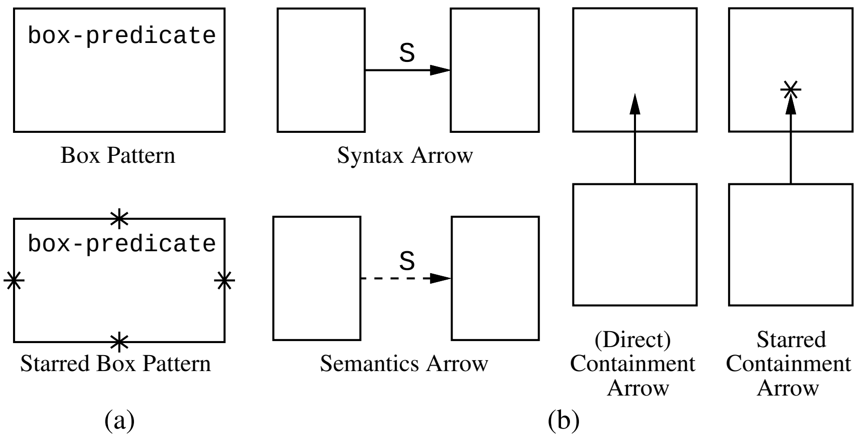

Figure 3: 10Renderings of Box Patterns (a) and Constraint Arrows (b)

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. Two instance boxes b_1 and b_2 match box patterns connected by a positive (negative) starred containment arrow if b_1 is (is not) contained in b_2.

The constraint language shares the instance language's powerful visual representation for box containment. In the constraint language, however, drawing one box directly inside another is simply a shorthand for connecting disjoint versions of the boxes with a containment arrow. As shown in Figure 4a, drawing one box pattern inside another is a shorthand for drawing the same two box patterns connected by a (direct) containment arrow. If the inner box pattern is starred, as shown in Figure 4(b), the original constraint is equivalent to one in which a starred containment arrow connects the two boxes.

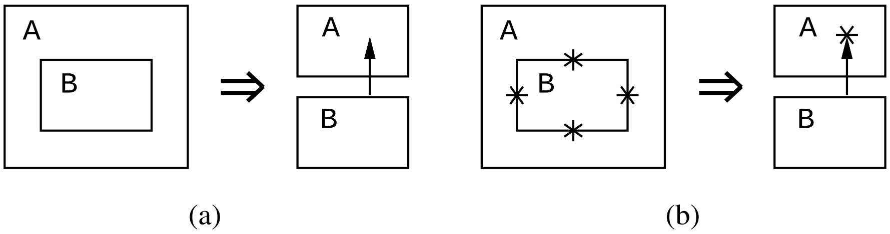

Figure 4: 10The Direct (a) and Starred (b) Box Containment Shorthands

The constraint language presented so far allows us to require only the existence of certain 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. Each box and arrow is drawn in either thick or thin style. Those 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 5 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 (the `read` arrow) then requires (as the consequent of the implication) that the user also has `read` permission on the file. Thus, Figure 5 expresses the constraint that "`write` permission implies `read` permission".

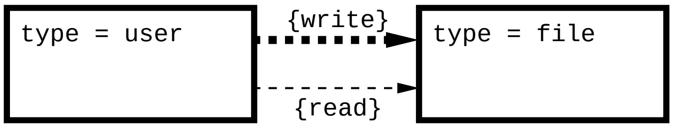

Figure 5: 10The Constraint WRITE-READ

3 The Miró Software System

Figure 6 shows the software tools comprising the Miró system and their inter-relations [8]. We classify the tools (and languages) as either *front end* or *back end* components. The front end components are designed to work independently of any operating system, while the back end components depend on the particular details of the file system with which they interact. To check file systems other than Unix file systems, we would have to reimplement only the back end tools. So far, we have implemented these tools only for Unix, but we do not expect that they would be much more difficult to implement for other kinds of file systems.[4]

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

[4] In fact, it is quite easy to implement a prober that produces an instance picture that does not exploit the instance language's grouping mechanism; the challenge is to build a prober that produces relatively concise output pictures.

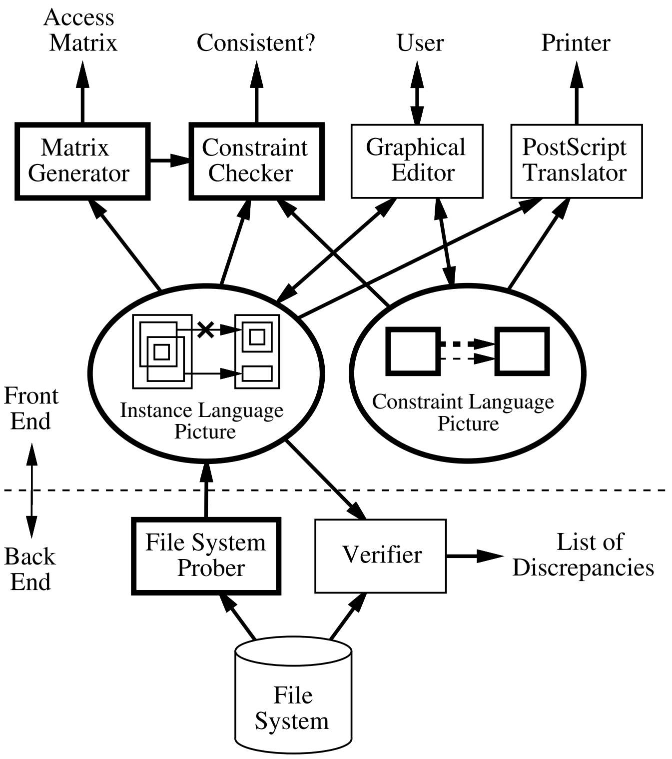

Figure 6: 10The Software Tools and Languages Comprising the Miró System.
The tools and languages relevant to this paper are shown with a thick outline.

printer. To write this paper, we used the graphical editor to draw the constraint pictures only; the instance picture we used was produced automatically by the prober tool described below.

The *access matrix generator* verifies that an instance picture is well-formed, and then generates a compressed representation of the access matrix corresponding to the instance picture. This access matrix and the instance itself can then be fed to the *constraint checker* along with some constraint picture to check whether 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 [11].

The tools described so far work independently of any file system. To interact with a Unix file system, we provide the back end *verifier* and *prober* tools. The verifier compares a given instance picture to a file system and produces a list of discrepancies between the two. 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 Unix prober produces a picture that represents the `read`, `write`, and `exec` access relations between users and files, and the `ins-del` (insert-delete) and `list` access relations between users and directories. The prober arranges for the access matrix corresponding to its output to reflect group memberships and the "execute" directory bits of each directory along the path down to each directory (for `ins-del` and `list` permissions) or file (for `read`, `write`, and `exec` permissions).

The prober does not explicitly lay out the boxes in the result it produces, since we did not need to visually inspect each result. Instead, it represents the abstract containment relation on boxes in its output. Hence, the prober need not determine a geometrically correct rendering of the containment relation. Also, the prober ignores symbolic links, since a symbolic link cannot be used to gain access permissions that are not otherwise granted through hard links.

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 access matrix generator to generate its corresponding access matrix. Finally, we fed the access matrix, the instance picture, and each of our 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 that 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 [9, 10]. The others were suggested in the Grampp-Morris article referred to earlier; we have simply translated their most promising written security suggestions into constraint pictures.

4.1 Miró Security Constraints

The constraints suggested by previous Miró papers are designed to fulfill a variety of needs. We have chosen a few representative samples. The first is a general security constraint for Unix, and the rest enforce various containment requirements relative to the box type system.

4.1.1 PRIVATE-MAIL

The PRIVATE-MAIL constraint is shown in Figure 7. This constraint assumes that the mail system organizes 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.

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

The constraints shown in Figure 8 place realizability restrictions on the nesting of `group` and `world` boxes in Unix — namely, that every `group` box is contained in the unique `world` box and no other.

Constraints (a) and (c) introduce a new aspect to the constraint language syntax and semantics for restricting the cardinality of the number of thin matchings in a constraint. This is done by associating an integer-valued interval with the 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: 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 *zero* thin extensions to each thick matching.

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

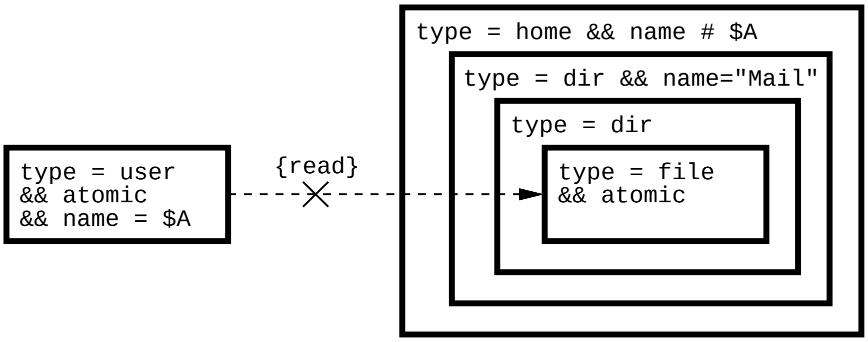

Figure 7: 10The Constraint PRIVATE-MAIL

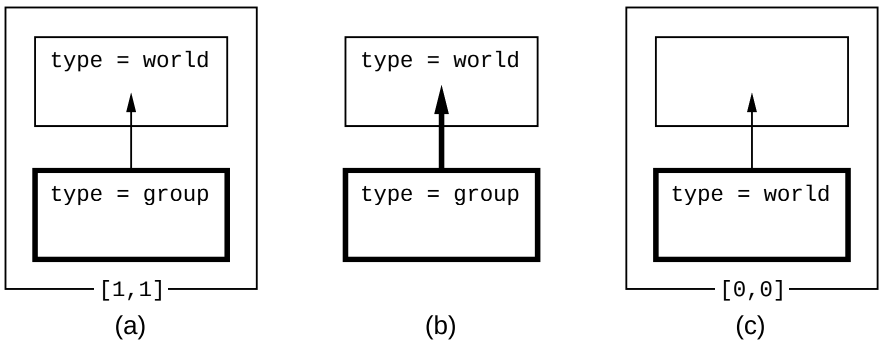

Figure 8: 10The Constraints GRP-IN-1-W, GRP-IN-W-ONLY, and W-IS-ROOT

and Morris go on to describe security holes that may occur on a Unix system. We have transcribed 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`. Obviously, no one besides its owner (the super user) should have permission to change this file. The constraint PASSWD-SAFE shown in Figure 9 requires that no user can `write` the password file except its owner.

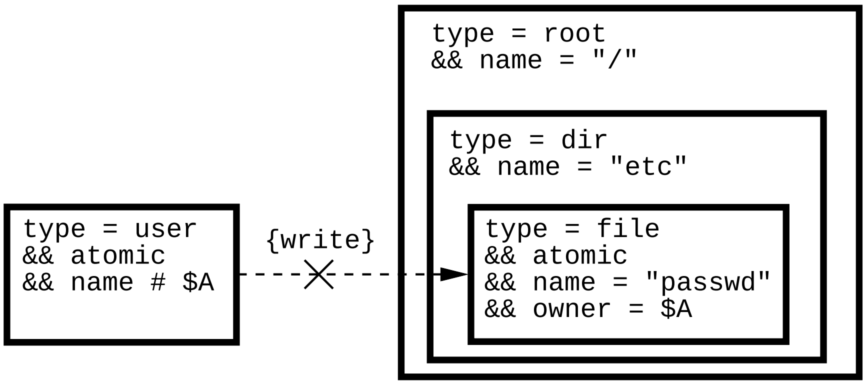

Figure 9: 10The Constraint PASSWD-SAFE

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 arbitrarily.

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 the file, but also 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: Any non-owner of a directory who has write permission on that directory must also have write permission on all files in that directory. This constraint is shown in Figure 10. There are several points we should make about this constraint.

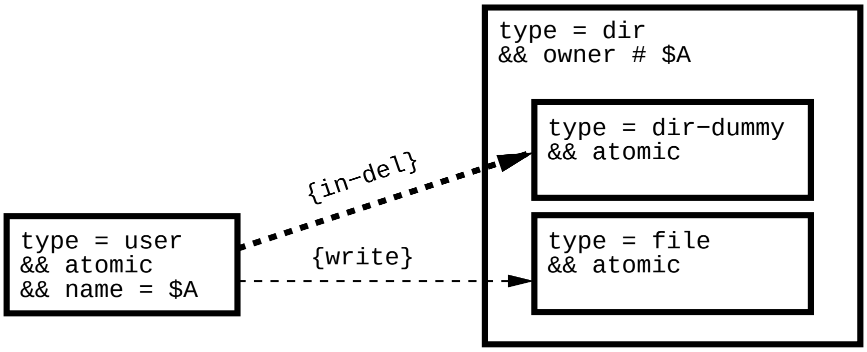

Figure 10: 10The Constraint WRITABLE-DIR

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 is 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 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.

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 copied shell executable 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. The thin negative `write` arrow then requires that that user does not have write access on the ".login" file.

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 prober required approximately 206 seconds of CPU time and 314 seconds of real time to produce this instance on a MicroVAX II. The access matrix produced by our access matrix generator on this instance picture contains a total of 2,490,033 entries; the access matrix file is approximately 348 Kbytes in size. The access matrix generator took 78 seconds of CPU time and 87 seconds of real time to run on a DECstation 5000. These results are summarized in Table 1.

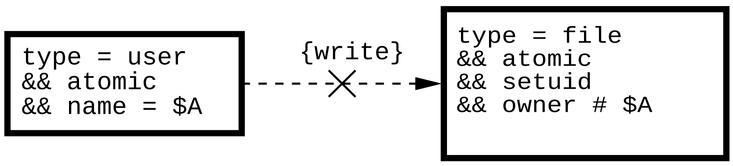

Figure 11: 10The Constraint SETUID-SAFE

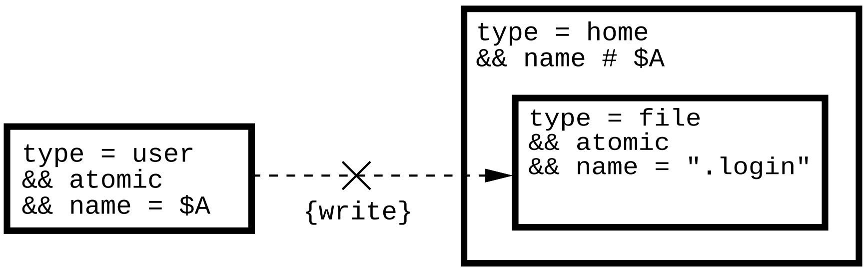

Figure 12: 10The Constraint LOGIN-SAFE

Tool	Machine	CPU Time	Real Time	Output File "Size"
Prober	DS-5000	206	314	23,679 total items
Generator	uVAX-II	78	87	2,490,033 entries

Table 1: 10Running Times (in Seconds) of Preliminary Tools

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 2 shows the times required for these two phases on each of the constraints. These experiments were performed on a DECstation 5000 with 128 MB of main memory 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. Even so, it would still be quite practical to run these tests automatically each night to check for new violations.

In general, we have shown that the constraint checking problem is Π_2^p-hard [11], which means that the problem is at least as hard as the hardest problems in the class NP. How well the performance indicated by Table 2 scales up to larger file systems is largely dependent on the constraints being checked. The constraint with the worst performance — WRITABLE-DIR — has a running time only on the order of quadratic in the number of boxes in the input.

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.

5.1 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 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

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	$8+1=9$	⟨ 44.7 / 45.2 ⟩
PASSWD-SAFE	$6+1=7$	[31.8 / 0.0]
WRITABLE-DIR	$7+1=8$	⟨ 44.1 / 1,200 ⟩
SETUID-SAFE	$2+1=3$	[24.8 / 0.0]
LOGIN-SAFE	$4+2=6$	⟨ 43.3 / 13.7 ⟩

Table 2: 10Constraint Checker Running Times. 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). Entries in [square brackets] have trivial query times, while those in ⟨ angle brackets ⟩ indicate that the corresponding instance is *inconsistent* with the corresponding constraint. Inconsistencies indicate potential security holes.

also found to be writable by someone other than their respective owners.

5.2 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, the default group for accounts on this machine. 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 by 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 read 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.

5.3 LOGIN-SAFE — 24 Violations

All 24 violations were produced because a single user's ".login" file was writable by the `theory` group. As mentioned previously, the theory group on this machine has 25 members, so the 24 people other than the file's owner have the ability to maliciously alter his ".login" file.

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 to solve this problem. The primary advantages it offers over existing tools are its generality, flexibility, and expressive power. This system can specify real world security policies and detect violations of those policies. Moreover, the constraint language gives users the power to formally specify abstract security models that closely match the way users think about their security policies. 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 Acknowledgements

We would like to thank Cynthia Hibbard, Jeannette Wing, and Amy Moormann Zaremsky for their comments on drafts of this paper. We would also like to thank Karen Kietzke for her help and patience in testing our access matrix generator and constraint checker.

References

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

[2] 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.

[3] T. Benzel. Analysis of a Kernel Verification. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 125–131, Oakland, CA, May 1984.

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

[5] 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.

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

[7] Allan Heydon, Karen Kietzke, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. The Miró Video. Published by the Industrial Affiliates Office, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1991. This videotape summarizes the results of the Miró project, and shows each of the Miró software tools in use. The video is approximately 20 minutes long. Requests for the video can be sent to the address above or via e-mail to `miro@cs.cmu.edu`.

[8] 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.

[9] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Constraining Pictures with Pictures. In *Proceedings of the IFIP 11^{th} World Computer Congress*, pages 157–162, San Francisco, CA, August 1989.

[10] 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.

[11] 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.

[12] Richard A. Kemmerer. *Formal Verification of an Operating System Security Kernel.* Computer Science: Systems Programming, No. 2. UMI Research Press, Ann Arbor, Michigan, 1982.

[13] 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.

[14] Brad A. Myers. The Garnet User Interface Development Environment: A Proposal. Technical Report CMU-CS-88-153, Carnegie Mellon University, School of Computer Science, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, September 1988.

[15] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Technical Report CMU-CS-90-117, Carnegie Mellon University, School of Computer Science, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, March 1990.

[16] 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.