An Information Flow Tool for Gypsy
An Extended Abstract Revisited

John M$^{\text{c}}$Hugh

CERT/CC®, Software Engineering Institute, Carnegie Mellon University
E-mail: `jmchugh@cert.org`

Abstract

In 1985, we outlined an information flow tool for the Gypsy language that could be used to support covert channel analysis. The proposed tool offered substantial advantages over existing tools in flexibility and promised to be useful for a variety of analyses. Two versions of the tool were subsequently built and used for a variety of MLS and other projects. This paper draws on the original, adding motivational and explanatory material and adds descriptions of the subsequent tools. It also illustrates a novel use of one of the tools in developing an architecture for a MLS windowing system.

1. Introduction

As one component of my dissertation[18] work at the University of Texas in the early 1980's, I implemented a dependency analyzer for the executable portion of the Gypsy[11] programming and specification language. The purpose of the analyzer was to identify apparently executable code that had no effect on the functioning of the program. This was a problem in Gypsy because intrinsic functions that represented information about the state of a computation, e.g. buffer histories containing copies of all messages sent to a given buffer, were often "computed" and passed as arguments to other routines where they were used only in (non-executable) specifications. The dependency analyzer marked the internal representations of such code as not affecting the program results so that the code generator could omit it from the execution.

In the early to mid 1980's, Gypsy was one of three specification systems[1] that were supported by the National Computer Security Center for use in the development of systems seeking an A1 evaluation under the TCSEC[7]. As the introduction to the original extended abstract[25] stated:

> The Gypsy [11] language is seeing increasing use as a tool for designing, specifying, and sometimes implementing computer systems intended for certification at the A1 level by the Department of Defense Computer Security Center. One of the criteria for A1 certification [7] is a formal proof that the information flows within the design conform to a policy defined by a formal security model. Despite the fact that it is possible to state such models in Gypsy and to prove some properties of programs with respect to a model[30], a flow analysis tool within the Gypsy environment would appear to be useful. The GVE (Gypsy Verification Environment) contains the basis for such a tool in the form of a flow analyzer used to detect unused variables during optimization[18]. In the discussion below, we will describe a simple information flow analyzer based upon this analysis.

At the time the extended abstract was written, several large multi-level secure systems targeting A1 were either under development or in the early planning stages. Gypsy had been used to specify some aspects of the Honeywell SCOMP (Secure COmmunications Processor) and the lack of a Covert Channel Analysis tool was seen as a barrier to the further use in future systems. The extended abstract represented our thinking on the problem of information flow characterization, and was, to some extent, an early attempt to obtain funding for the development of a Gypsy covert channel analysis tool.

Before proceeding, we briefly discuss covert channels and MLS systems as these may not be familiar to newcomers to the field. The remainder of the paper contains substantial quotations from the original, interspersed with commentary and exposition. This is followed by an account of subsequent events, i.e. the development of two generations of information flow based covert channel analysis tools for Gypsy and their subsequent use on systems such as LOCK[28].

[0]This work was sponsored by the Department of Defense.

[1]HDM from SRI and Ina-Jo from SDC were the other two.

2. Multi Level Secure systems and Covert Channels

At the time the abstract was written, MLS systems and the covert channel problem and its relationship to information flow were well known, if only partially understood. Thus, it is only slightly surprising that the application of the analyzer to covert channel analysis is nowhere mentioned. MLS systems are not explicitly mentioned, but A1 systems are inherently MLS.

Within DoD and similar organizations, there is a need to handle both classified and unclassified information, often at the same time. In the print world, classified information is be kept under appropriate physical security and can often be used only in specified locations. Classified documents are clearly marked with their classifications and, at least under some circumstances, can be accessed concurrently with unclassified information. As computers began to be used to store classified information, the print model broke down. Cleared individuals are trusted not to incorporate classified information into unclassified documents that they may be creating. In the early 1970s, there was no basis for extending the human trust model to software nor was there an appropriate labeling mechanism. This led to the development of single level or "system high" systems in which all material, regardless of its intrinsic classification, is effectively classified at the level of the most sensitive material in the system. Because highly sensitive material is typically a small percentage of the total, the resulting over classification creates substantial costs and operational difficulties. Against this background, mechanisms[2, 3] were proposed to allow the safe manipulation of material of differing classifications within a single computer system. These systems were characterized by a mandatory security policy in the style of Bell and LaPadula[4], in which access decisions are based on labels associated with data containing objects and with (abstractly) data free subjects that move information between objects. At the higher levels of assurance (B3, A1), the policy is enforced by a security kernel which comprises an unbypassable reference monitor through which all access requests must pass.

Unfortunately, direct access to objects followed by an explicit, direct transfer of some or all of the object's contents to another object is not the only way in which information flows in a computer system. In most systems, it is possible for one process to perceive some aspects of the operations of other processes through conflicts in attempts to access resources. The locking of a record, file, or device to permit modification without fear of interference is one example. The variation in response times due to processes competing for CPU cycles or file access on a shared disk is another. Such mechanisms can be manipulated deliberately to signal between processes and it is from such mechanisms that covert channels are created.

Covert channels can arise outside of a MLS context but they were most commonly considered in the context of systems that have enforce a mandatory access control policy. In general, a covert channel is a mechanism that can signal information across a protection boundary in violation of the mandatory policy. Original work in this area was done by Lampson and Lipner[14, 16] in the mid 1970s. Because signaling rather than direct data transfer is involved, the information being compromised may be coded and a given covert signaling scenario may pass a half a bit (or less) per invocation. By convention, covert channels are classified as storage channels in which the sender affects the value of some storage entity that can be observed by the recipient or timing channels in which the sender affects the recipients perception of the time required to carry out some action. Kemmerer[13] provides definitions of these as follows:

> In order to have a storage channel, the following minimum criteria must be satisfied:
>
> 1. The sending and receiving processes must have access to the same attribute of a shared resource.
> 2. There must be some means by which the sending process can force the shared attribute to change.
> 3. There must be some means by which the receiving process can detect the attribute change.
> 4. There must be some mechanism for initiating the communication between the sending and receiving processes and for sequencing the events correctly. This mechanism could be another channel with a smaller bandwidth.
>
> If criteria (1)-(3) are satisfied, one must find a scenario that satisfies criterion (4). If such a scenario can be found, a storage channel exists. This last step requires imagination and insight into the system being analyzed. However, by using the shared resource matrix approach, attributes of shared resources that do not satisfy criteria (1)-(3) can readily be identified and discarded.
>
> Timing channels are discovered in a similar manner, but different criteria are used. The minimum criteria necessary in order for a timing channel to exist are as follows:
>
> 1. The sending and receiving processes must have access to the same attribute of a shared resource.

2. The sending and receiving processes must have access to a time reference such as a real-time clock.
3. The sender must be capable of modulating the receiver's response time for detecting a change in the shared attribute.
4. There must be some mechanism for initiating the processes and for sequencing the events.

Any time a processor is shared there is a shared attribute, i.e., the response time of the CPU. A change in response time is detected by the receiving process by means of monitoring the clock.

3. The Information Flow Problem

At the time the flow tool was proposed, I was thinking primarily in terms of analyzing the flows in running code, and the exposition given in the abstract follows that model.

> Briefly stated, the information flow problem[26, 6, 15] is: Given a program and its sets of input and output variables, determine for each output variable, the subset of the input variable set about which it might possibly contain information after execution of the program. In strongly typed languages such as Gypsy or Ada, the type mechanism effectively prevents us from describing information flows among incompatible types. This is because information flow as an abstraction is independent of the type mechanism or value sets of the program variables. For example, it is not possible to specify the nature of the information transferred from the sequence S to the integer I in the Gypsy statement
>
> `I := length(S)`
>
> using Gypsy specifications; yet it is clear that an information flow has taken place. We can overcome the restrictions in a number of ways. The work referred to above [30] uses an incomplete program representation involving pending or not yet defined types to provide a framework for the proofs. Unfortunately, this technique cannot be applied to fully defined or executable programs, as it is usually necessary to instantiate the types on which the proof depends in a variety of incompatible ways. The SRI MLS tool [9] contains implicit models of both information flow and a flow policy. The tool described in this paper supports information flow proofs by creating an abstraction of the program in which all objects are of the same type and information flow is the only operation performed by the abstraction. The user of the tool can created an information flow theory or policy in terms of the abstract information type used in the process and prove properties of both the program and the policy in a manner similar to that currently used for proofs of correctness in Gypsy.

The proposed tool relied heavily on an information flow analyzer for Gypsy code which had been developed to identify code fragments that had no observable effect in the running program. Although the term was unknown at the time, proponents of static code analysis will recognize the following description as being a description of program slicing[31].

> The Gypsy optimizer contains code to identify variables, which have no effect on the outputs of the program. Such variables, known as "ghosts" are often introduced into programs to facilitate proofs. Elimination of these variables and any otherwise executable code, which references them, can effect a substantial improvement in program size and performance. In order to identify ghosts, the optimizer conducts a detailed flow analysis, identifying all objects in the program, which contribute information to its output parameters. Any objects, which do not contribute to the output, are ghosts and can be eliminated.
>
> The flow analysis defines information flow semantics for each Gypsy construct. The program is transformed into an information flow analogue using a procedure similar to the one followed during verification condition generation. A path set through each routine is generated, and the contributors to each output parameter of each routine are determined. The semantics used account for flows resulting from control constructs and buffer operations blockage as well as from normal assignment operations. Procedure calls are treated as a set of assignment statements exchanging information among the actual parameters of the routine. Explicit exception paths are handled, and provision is made for treating implicit exception paths in a number of ways, depending on the proof status of the routine and the degree of caution required.
>
> Gypsy routines can communicate with the outside world only through their parameters; these routines do not retain information from one invocation to another. Information can flow to

a VAR or variable parameter from other parameters, global constants, and literals. Given a method of assigning information labels to these items, it is necessary to show that the flows identified conform to the stated policy. Only if we fail to show this are we interested in the details of the flow paths.

It is clear from the description that the analysis used in the optimizer to identify ghosts can be applied to detect the kind of resource sharing or information flow dependencies that allow the construction of covert channels.

4. The Gypsy Information Flow Abstraction

Information flows of the kind we are concerned with violate the moderately strong typing that the Gypsy semantics imposed. In order to make a tool that could operate within the semantic framework imposed by the language, it is necessary to play some tricks. The abstract proposed the following mechanism.

> In order to develop information flow theories in the Gypsy context, we have developed an abstraction, which allows us to capture the information flow of a Gypsy program. The abstraction consists of a Gypsy scope, which defines an abstract type called `INFORMATION` and a set of functions, and procedures, which implement operations on the type. Two properties of an object of this type are of interest for proving properties about information flows. These are the information label and the information content of the object. The label is an element from the associated information policy scope. The only restriction placed on the label is that the policy must be stated in terms of relationships among members of the value set of the label type. The content of the object is a set of values from the label type. The label of an object is fixed at the time the object is instantiated. The content is a function of the operations, which have been performed on the object. The abstraction provides two operations to support information flow modeling. The first is an information merging function, which returns and object whose contents is the union of the contents of its arguments. The second is an assignment procedure, which replaces the contents of its output parameter with the contents of its input parameter. These two routines are sufficient to model information flows obtained from the flow analysis.
>
> The tool uses this abstraction and the results of the flow analysis to create an information flow analogue of the program being analyzed. A set of routines comprising the interface for which flow proofs are required is identified. For each routine in the set, a flow analogue consisting of flow merge and assignment operations capturing the flows identified in the analysis stage is created. All parameters, constants and literals of interest from a flow standpoint are replaced with objects of the information type. Constant functions may be used to allow assertions to be made about the labels and contents of constants and literals.

The intent of this abstraction was to use the internal mechanisms of the GVE, i.e. the verification condition generator and the theorem prover to generate and prove a set of formulae that would show a given Gypsy program to be secure in a MLS sense. Such formulae require that the information flows as well as the sensitivity of the information containing objects be represented.

5. Policies and a Simple Tool

As noted above, earlier tools contained a “builtin” MLS security policy model that we found somewhat constraining. We were starting to consider covert channel analysis as a mechanism for looking at other flow related security issues such as “red/black” separation in cryptographic systems and we wanted the flexibility to introduce any appropriate policy formulation. The abstract proposed the following mechanisms:

> An information flow policy is represented by a scope, which contains type, constant, and routine declarations, which define the policy. At a minimum, a policy scope defines a type whose value set is used to provide labels and contents sets for information objects and a function, which is used to determine if the relationship between the label and contents of an object is in conformance with the policy. The policy may contain much more. In particular, lemmas and functions may be required to show that the label type exhibits desired properties. It may be necessary to define a comparison function for the label type and to provide support to prove that this function has the properties required of it for policy implementation. This approach has the advantage that the policy statement is expressed in the same metaphor as the program and can be subjected to the same sorts of analysis, proofs, and review as the programs to which it applies. The policy and any proofs of its properties, which are independent of its application, can be reused in many programs.

Given that some of the proof work is extremely tedious, the ability to reuse components such as a security policy model is highly desirable. Because the security label type and comparison function are user defined, the user has an obligation to ensure that, for example, the label and function define a lattice, if a Bell and LaPadula style of policy is being used. This requires that the existence of label values representing Least Upper and Greatest Lower Bounds be demonstrated for all values of the label type and that the comparison function be shown to be reflexive, transitive, and anti-symmetric.

Because Gypsy was intended as a general purpose programming language as well as a specification language, it had no inherent TCB or reference monitor model. Typically, TCBs are modeled as finite state automata that respond to an input by (possibly) modifying a global state, and (possibly) returning some value that is a function of the state and the input. Although one can model such a system in Gypsy, the model is not built in and the lack of global variables in the language forces the state to be passed about as a var parameter to all the interface routines. This complicates the tool as represented in the abstract.

> As noted above, information flows out of a Gypsy program only through its VAR parameters. In Particular, we can analyze the information flow behavior of an entire Gypsy program, no matter how large or complex, by performing the flow analysis discussed above and then proving that the flows among the parameters to the main routine of the program conform to our policy. Unfortunately, this model is often too simple for realistic applications. In many of the systems for which this type of analysis is required, the code to be analyzed serves as part of a larger system, and the interface of the secure portion, the trusted computing base or TCB, is more complex. In order to capture this behavior in Gypsy, we define the notion of a TCB interface set. The TCB is more complex. In order to capture this behavior in Gypsy, we define the notion of a TCB interface set. The TCB interface set contains exactly those routines of the TCB, which are intended to be called by routines outside the TCB. The TCB consists of all routines reachable from the TCB interface set. Having analyzed the flows within the TCB, it remains to show that the flows among the parameters of the interface set routines satisfy the flow Policy. This is done by creating the flow analogue routines, specifying and proving them. Use of the information flow tool consists of the following Steps:
>
> 1. Define or select a previously defined information flow policy expressed as a Gypsy theory, i.e. a set of Gypsy functions, constants, lemmas, and data types.
> 2. Identify the TCB interface set.
> 3. Perform information flow analysis for the TCB.
> 4. Construct information flow analogues for each routine in the TCB interface set using the information flows from the previous set.
> 5. Provide information policy specifications for the routines of the TCB interface set and for literals or constants appearing in the flows to the parameters of the interface set.
> 6. Generate verification conditions for the flow abstraction routines
> 7. Prove the VCs
>
> If the program satisfies the policy, the proofs should be straightforward. Unfortunately, many programs do not satisfy their flow policies. Reasons for this and an improved tool are discussed below.

After some discussion as to the reasons why the proposed tool would be superior to other approaches (elided here), the abstract concluded:

> Many programs for which information flow proofs are desired will not conform to the stated policy for a variety of reasons. Contrary information flows may exist through blockage or control variable information transfer. Overt channels [5] may be deliberately established to allow a trusted individual, such as the System Security Officer, to perform activities such as downgrading which, though necessary, violate the security policy. In many cases, the program is acceptable in spite of the contrary flows, but showing this requires identification and evaluation of the paths involved. The primary disadvantage of the simple tool discussed here is the fact that it provides little or no help in locating the precise source and nature of flows which are in violation of the policy under consideration. This is because the abstraction used collapses all paths for a routine to a single set of information assignments. In many cases it is possible to determine the source of violations with a simple manual inspection of the code involved; however, this may be difficult or impossible for large programs with numerous paths, especially if some of the paths involve implicit exceptions raised by Gypsy operators. We feel that

a tool, which overcomes most of these disadvantages, can be built should there be sufficient interest in the security community. Such a tool would retain the path structure of the flow analysis and allow identification of the offending paths in the corresponding Gypsy version of the program.

We also feel that techniques similar to those used in the Gypsy tools could be applied to Ada. A prerequisite would be the availability of an Ada based verification system.

The abstract was, in effect, a "fishing expedition." We knew that a Covert Channel Analysis facility was essential if Gypsy was to be used for specifying high assurance TCBs. At the time the abstract was written, we had not identified a funding source. The initial development of the GVE had, in fact, been "bootlegged" under funding that was intended to produce security proofs, *not* a verification system. Several months after the abstract appeared, funding materialized, I left RTI, and joined Computational Logic to develop the first of two information flow based covert channel analysis tools.

6. The SRM Tool

The first of the Gypsy covert channel analysis tools used the dependency analyzer to produce a Shared Resource Matrix (SRM) in the style of Kemmerer[13]. The work was funded by TRW in support of an unnamed customer. At the time the tool was started, there was some concern that the quantity of formulae produced by the SRI tool and the difficulties involved in either proving them or explaining their unprovability were excessive and that a simpler approach might be preferred. The SRM is an intuitive and elegant way of presenting information flow information in a way that helps a skilled analyst understand the system and identify potential covert channels. Flow information extracted from the optimized had been used to construct an SRM of the Secure Ada Target (SAT)[2] abstract model[12] and we were familiar with the form and its use.

6.1. The Shared Resource Matrix Methodology

The SRM methodology [13] was developed as a tool for finding both storage and timing channels in a variety of system description paradigms. The reference cited above applies the technique to a specification characterized in English, in the formal specification language Ina Jo, and as Pascal code. The SRM is an abstraction of the system behavior that reduces the system description to a simple matrix. The horizontal axis of the matrix lists the operations of the system. The vertical axis lists the system resources that are shared among users and manipulated by the operations. Each entry in the matrix may have one of four values:

1. R-indicating that the operation references the resource in some way.
2. M-indicating that the operation modifies the resource.
3. R, M-indicating that the operation both references and modifies the resource.
4. Blank-indicating that the operation does neither.

As originally formulated, the SRM does not distinguish between conditional and unconditional references and/or modifications, but merely indicates the potential for such actions.

6.2. Extensions to the SRM

The Gypsy Shared Resource Matrix tool extends the SRM methodology in several ways. These extensions serve to refine the technique by eliminating apparent flows that are artifacts of a compressed presentation format, and by explicitly identifying flows to and from the user. They are the result of experience with manual application of the technique as well as discussions with other researchers in the field.

6.2.1 User Flows

Whenever the user invokes a TCB interface routine that alters the internal state of the system, an implicit or explicit transfer of information from the user to the state occurs. On the other hand, a TCB call may or may not transfer information from the state to the caller. In Gypsy it is possible to specify TCB interface routines in a way such that it is not possible for the caller to distinguish between a successful or unsuccessful completion. In these cases no information is returned to the user.

To record transfers to and from users, the basic SRM is extended by adding two rows at the bottom. These are labeled "user in" and "user out". The "user in" row always contains Rs, indicating the unconditional transfer of information from the user to the state. The "user out" row contains an M if the operation returns information to the user.

6.2.2 Operation Splitting

The resolution of the SRM can be improved in several ways. Consider the following fragment of code taken from a routine that we shall call `OP1`.

[2]Later known as LOCK.

```
A := B;
C := if D
        then E
        else F ;
```

Under the original formulation of Kemmerer in which all information flows associated with a given operation are recorded in one column, this would lead to SRM entries

	Operation
Resource Attribute	*Op1*
A	M
B	R
C	M
D	R
E	R
F	R

The natural interpretation of this SRM is that flows from B, D, E, and F to both A and C are present. Inspection of the code shows that this is not true and that a more appropriate SRM would be

	Operation	
Resource Attribute	*Op1*	*Op1*
A	M	
B	R	
C		M
D		R
E		R
F		R

6.2.3 Guard Expansion

Even this version can stand refinement since it shows C as always containing information about both E and F, when in fact it contains information about one or the other. To reflect this refinement, the following expansion of the SRM is used:

	Operation/Guard			
Resource Attribute	*Op1 G1*	*Op1 G2*	*Op1 G3*	*Op1 G1*
A	M			
B	R			
C		M	M	M
D				R
E		R		
F			R	

where: G1=true
G2=D
G3=¬ D

The Gypsy SRM tool displays a basic view of the complete SRM in which the multiple flows of complex routines are composed into a single column, but it calculates the expanded form. The flow explanations, which are done on an operation by operation basis, display expanded columns as well as guard information.

6.3. The GIFT

The Gypsy SRM tool was successful, but the manual analysis required, even with the explanatory material provided made its application to large or complex systems problematic and the Gypsy Information Flow Tool (GIFT[3]) was developed to help mechanize the analysis by creating information flow formulae.

Several covert channel tools have been built based on the generation of information flow theorems. The best known of these are the old and new SPECIAL tools of [10] and [27]. These tools require a restricted form of a state machine specification for the system being analyzed. A security level is assigned (often arbitrarily) to each state component and to the invokers of the system operations. For each transfer of information within the system, a putative theorem is generated that, if true, guarantees that the information transfer proceeds according to the system's security policy.

This approach is overly conservative for several reasons:

1. It considers operations in isolation. It is often the case that there is no way to exploit the apparent insecurity of a given routine.
2. The required labeling of every state component with a classification results in inappropriate labels for some components. In many cases, the number of failed theorems is a function of the labeling strategy used, and substantial effort is required to choose a suitable labeling.

If it is possible to find a labeling that results in no failed theorems, then the model is free of covert storage channels. Failed theorems do not necessarily indicate exploitable channels, but substantial effort is required to ensure that this is the case. Most real systems exhibit a fair number of failed theorems when this approach is applied.

6.4 A Simplified Example

The fragment of code used in connection with the above discussion of the SRM approach does not satisfy our conventions for Gypsy covert channel analysis. Nevertheless, it can be used to demonstrate the nature of the formulas that

[3]The acronym was coined by Craig Singer, one of the developers. Since some of the development was done in the hope that a contract to support LOCK with the tool would materialize, the name is a potential pun.

will be produced by the GIFT. If we present the fragment as an equivalent set of guarded assignment statements, it becomes

```
    if  true then A := B;
    if   D   then C := E;
    if not D then C := F ;
```

In addition to the guarded assignments that characterize the system in terms of conditional information flows, it is necessary to know the security level associated with each of the system entities, A, B, C, D, E, and F. Without a loss of generality, we will define a unique level function for each entity, leaving the arguments required unspecified. We will name these functions `Level_of_A(...)` through `Level_of_F(...)`. As can be seen from the SRM and the guarded assignments, information flows into A from B unconditionally, into C from E when `D=true`, into C from F when `D=false`, and into C from D unconditionally. These flows are secure if and only if the targets of the flows have security levels equal to or above those of the sources when the flow condition is satisfied. This gives rise to the following information flow formulas:

```
Level_of_A(...) ge Level_of_B(...)
Level_of_C(...) ge Level_of_D(...)
D => Level_of_C(...) ge Level_of_E(...)
not D => Level_of_C(...) ge Level_of_F(...)
```

If each of these formulas can be proven, then the fragment is secure.

The GIFT provided facilities for defining a TCB in terms of a type that represented the internal state and a set of procedures or functions that operated on the state and/or returned information about it. It also provided mechanisms for defining security labels, for associating labels with state components (as seen above), and for defining a level comparison function. In addition to lattice policies, policies based on total orderings, partial orderings, and isolation among equivalence classes.

In most cases, Gypsy was used solely to specify the TCB, but not to implement it. The original dependency analyzer was designed to operate on code rather than specifications and had to be modified to work on specifications. This leads to fairly draconian requirements on specification style, but the requirements are necessary to ensure that the specifications explicitly reflect the effects of TCB operations on the TCB state. The GIFT manual[21][4] goes into considerable detail on the restrictions and conventions and provides an example.

[4]Like many of the CLI reports associated with Gypsy and the GVE, this report is out of print. Copies are available in Postscript form from the author, jmchugh@cert.org.

7. An Interesting Application

The SRM tool supported analyses other than for covert channels. In the late 1980s, CLI worked with TRW and TIS on the development of a process model for the construction of high performance trusted systems[17]. For the second phase of this project, we applied the model to the construction of a MLS windowing system[8]. Formal analysis was one of the primary risk mitigation strategies used in the process and the SRM tool played a significant role in the design of the windowing architecture. We were debating whether to attempt to secure the X-window protocol by imposing a TCB on the reference implementation of X windows or to use a more radical approach. Compartmented Mode Workstations (CMWs) had taken the former path, but we were uncertain that the requirements for our target, the B3 level of the TCSEC could be satisfied with this approach. Cut and paste operations were particularly problematic. We created a model of the ICCCM cut and paste protocol in Gypsy and used the SRM tool to analyze its information flow characteristics. In ICCCM cut and paste, the cutting window asserts ownership of an X resource known as the “selection” and associates the object to cut with it. The pasting window determines the owner of the selection, inquires as to the formats in which the selection can be made available, chooses one (or more) and asks for the selection to be converted to that (those) format(s). The cutting window performs the conversion. The pasting window then asserts ownership of the selection and copies the converted object. It is clear that no MLS policy would permit cutting from a high level window followed by pasting to a low level one, the reverse seems desirable.

We modeled this in some detail in Gypsy, producing the SRM shown in figure 1. Analysis of the SRM confirmed our intuitions, i.e. the bidirectional messages required by cut and paste could not be supported in a MLS context. The CMW effort swept this under the rug (where it remains to this day) by declaring that these were covert channels and, since the B1 criteria do not require covert channel analysis, the behaviors were acceptable. At first, we were tempted by the interpretation and performed analyses to attempt to determine the information capacity of these communications mechanisms. One problem that arose is the fact that the design of the X protocol goes to great length to make it difficult to determine whether events result from human or program activity. Thus, a mechanism that might be restricted to a few bits per second when driven by a human clicking a mouse could be driven by program commands at much higher rates. Ultimately, we concluded that communications mechanisms *required by the X protocol* could not be considered as covert channels under any circumstances (even at B1) and that the X protocol was inherently insecure and unsecurable in an MLS sense. This led us to an archi-

Resource	SETSEL #1	GETSEL #2	CONVRT #3	CHANGE #4	SENDEV #5	SELECT #6	SELECT #7	SELECT #8
SIZE (STATE.ATOM_LIST)	R	R	R	R				
DOMAIN (STATE.ATOM_LIST)	R	R	R	R				
RANGE (STATE.ATOM_LIST)								
SIZE (STATE.ATOM_LIST[IND#1])								
STATE.ATOM_LIST[IND#1][IND#2]								
SIZE (STATE.CLIENT_LIST)								
DOMAIN (STATE.CLIENT_LIST)								
RANGE (STATE.CLIENT_LIST)								
STATE.CLIENT_LIST[IND#3].P								
STATE.CLIENT_LIST[IND#3].SL	R		R	R	R	R	R	R
STATE.CURRENTTIME	R			R				
SIZE (STATE.PENDING_EVENTS)	M R		M R	M R	M R	M R	M R	M R
STATE.PENDING_EVENTS[IND#4].BODY	M R		M R	M R	M R	M R	M R	M R
STATE.PENDING_EVENTS[IND#4].EV_NAME	M R		M R	M R	M R	M R	M R	M R
STATE.PENDING_EVENTS[IND#4].ORIGINATOR	M R		M R	M R	M R	M R	M R	M R
STATE.PENDING_EVENTS[IND#4].RECIPIENT	M R		M R	M R	M R	M R	M R	M R
STATE.REQUESTOR	R		R	R	R	R	R	R
STATE.REQUEST_P				R		R	R	R
SIZE (STATE.SELECTION_LIST)	R	R	R					
DOMAIN (STATE.SELECTION_LIST)	R	R	R					
RANGE (STATE.SELECTION_LIST)	M R							
STATE.SELECTION_LIST[IND#5].OWNER	M R	R	R					
STATE.SELECTION_LIST[IND#5].TIME	M R							
SIZE (STATE.WIN_LIST)	R		R	R	R			
DOMAIN (STATE.WIN_LIST)	R		R	R	R			
RANGE (STATE.WIN_LIST)				M R				
STATE.WIN_LIST[IND#6].OWNER	R		R	R	R			
SIZE (STATE.WIN_LIST[IND#6].PROPS)				R				
DOMAIN (STATE.WIN_LIST[IND#6].PROPS)				R				
RANGE (STATE.WIN_LIST[IND#6].PROPS)				M R				
SIZE(STATE.WIN_LIST[IND#6].PROPS[IND#7].DATA)				M R				
STATE.WIN_LIST[IND#6].PROPS[IND#7].DATA[IND#8]				M R				
STATE.WIN_LIST[IND#6].PROPS[IND#7].FORMAT				M R				
STATE.WIN_LIST[IND#6].PROPS[IND#7].TYPE_VAL				M R				
USER-IN	R	R	R	R	R	R	R	R
USER-OUT	M	M	M	M	M	M	M	M

Figure 1. The Shared Resource Matrix for ICCCM Cut and Paste

tecture based on independent replicas of the X server (one per active level) and a restrictive version of cut and paste that could be shown to adhere to MLS restrictions.

8. In Retrospect

The original paper was an unabashed attempt to get funds to build the tool that was described. Note that the abstract is ambiguous about the state of the analyzer. On the other hand, it worked. Perhaps, this is the real significance of the paper, a classic sales job. In retrospect, the line of work that began with the paper had a substantial impact on several aspects of the MLS world. It was the first MLS tool to give the system designer a large degree of flexibility in the way both systems and policies were described. Its reporting mechanisms allowed both summary and detailed reports to be produced. The conservative approximation mechanisms introduced in the GIFT tool allowed the analyst to greatly reduce the number of detailed flows to be considered by assuming that information flowed universally from all inputs to all outputs of certain functions. If the system was secure under these assumptions, it was clearly secure under more detailed analysis. The nature of the information flow formulae produced led to minor improvements in the strategy of the Gypsy theorem prover, reducing proof efforts substantially.

Both the SRM tool and the GIFT were used on real MLS systems. MLS systems are out of fashion these days, the common wisdom being that the marketplace passed them by, but my recent experience with attempts to deal with web sites that are classified in their entirety because some page may be classified indicates that the need still exists. With a few exceptions, mostly dealing with small, very sensitive, information objects (such as long lived encryption keys), small information leakages are not of much concern today; after all, most systems are so vulnerable that it is far easier to take ownership of the system via a simple exploit than it is to attempt to signal information through the protection state.

Nonetheless, much of the work that went into the GIFT and the GVE are relevant today. I suspect that covert channel mechanisms will again be considered important in the contexts of restricted execution environments for mobile agents. Proof carrying code may require proofs that covert signaling mechanisms are not present as well as the higher level proofs currently proposed. There has been some recent interest in a GVE style verification system for Java and this is certainly feasible. Moore's law has already solved many of the performance problems for which the GVE of a decade ago was rightly criticized. Dependency analysis is gaining increasing importance as a mechanism for addressing the buffer overflow problem. Another part of the Gypsy optimizer addressed this problem explicitly by generating formulae that, if proven, justified the removal of run time array bounds checks and it is clear that this is equivalent to static analysis of a program slice focused of the potentially offending operation with the objective of showing that the potential overflow cannot occur.

More importantly, the effort that began with the 1985 abstract contributed substantially to our understanding of the subtleties of information flows and their role in developing secure systems. In addition to a number of CLI reports and notes[21, 1, 23, 24, 20, 22], the effort produced a Masters Thesis[29], and several conference and journal papers[12, 19]. The GIFT was by no means a solo effort. Major contributions were made by a number of people in Durham and in Austin. Larry Akers, Bret Hartman, Craig Singer, and Tad Taylor were all major contributors to the effort and without their work, it could not have come to fruition.

References

[1] R. L. Akers. Information flow into structured objects. Internal Note 20, Computational Logic Inc., 1717 West 6th. Street, Austin, Texas, November 1987.

[2] J. P. Anderson. Computer security technology planning study, Volume I. Technical Report ESC–TR–73–51, Vol I, ESD/AFSC, Hanscom AFB, Bedford, MA 01731, Oct 1972. [NTIS AD-758-206] - Available online at `http://seclab.cs.ucdavis.edu/projects/history/CD/ande72.pdf`.

[3] J. P. Anderson. Computer security technology planning study, Volume II. Technical Report ESC–TR–73–51, Vol II, ESD/AFSC, Hanscom AFB, Bedford, MA 01731, Oct 1972. [NTIS AD-758-206] - Available online at `http://seclab.cs.ucdavis.edu/projects/history/CD/ande72.pdf`.

[4] D. E. Bell and L. L. Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD–TR–75–306, ESD/AFSC, Hanscom AFB, Bedford, MA 01731, 1975. [DTIC AD-A023588] - Available online at `http://seclab.cs.ucdavis.edu/projects/history/CD/bell76.pdf`.

[5] R. Boebert. Private communication, November 1984.

[6] D. Denning and P. Denning. Certification of programs for secure information flow. *CACM*, 20(7):504–513, July 1977.

[7] Department of defense trusted computer system evaluation criteria. CSC-STD-001-83, Department of Defense Computer Security Center, August 1983.

[8] J. Epstein, J. McHugh H. Orman, et al. A high assurance window system prototype. *Journal of Computer Security*, 2:159–190, 1993.

[9] R. Feiertag. Technique for proving specifications are multilevel secure. Technical Report CSL-109, SRI International Computer Science Laboratory, January 1980.

[10] R. Feiertag, K. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proc. 6th Symp. on Operating System Principles*, pages 57–65. ACM, November 1977.

[11] D. Good, R. Cohen, C. Hoch, L. Hunter, and D. Hare. Report on the language gypsy: Version 2.0. Technical Report ICSCA-10, University of Texas at Austin Institute for Computing Science and Computer Applications, September 1978.

[12] J. T. Haigh, R. A. Kemmerer, J. McHugh, and W. D. Young. An experience using two covert channel analysis techniques on a real system design. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 14–24. IEEE, April 1986.

[13] R. A. Kemmerer. Shared resource matrix methodology: A practical approach to identifying covert channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[14] B. W. Lampson. A note on the confinement problem. *CACM*, 16(10):613–615, Oct 1973. Currently on line at `http://research.microsoft.com/lampson/11-Confinement/WebPage.html`.

[15] C. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–248, September 1981.

[16] S. B. Lipner. A comment on the confinement problem. In *Proceedings of the Fifth Symposium on Operating System Principles*, pages 192–196, 1975. Appeared as ACM Operating System Review 9(5).

[17] A. Marmor-Squires, J. McHugh M. Branstad, B. Danner, L. Nagy, P. Rougeau, and D. Sterne. A risk driven process model for the development of trusted systems. In *Proceedings of the 1989 Computer Security Applications Conference*, Tucson, AZ, December 1989.

[18] J. McHugh. Towards the generation of efficient code from verified programs. ICSC Report 40, Institute for Computing Science University of Texas at Austin, March 1984.

[19] J. McHugh. A formal definition for information flow in the gypsy expression language. In *Proceedings of The Computer Security Foundations Workshop*, pages 147–165, June 1988.

[20] J. McHugh and R. L. Akers. A formal justification for the gypsy information flow tool. Technical report, Computational Logic, Inc., Austin, Texas, 1987.

[21] J. McHugh and R. L. Akers. Gve users manual: The gypsy information flow tool, a covert channel analysis tool. Technical Report CLI-12, Computational Logic, Inc., November 1987. Draft.

[22] J. McHugh and R. L. Akers. Specification and rationale for the implementation of an analyzer for dependencies in gypsy specifications. Technical report, Computational Logic, Inc., Austin, Texas, 1987.

[23] J. McHugh, R. L. Akers, and C. D. Singer. Gve user's manual: Shared resource matrix covert channel analysis tool. Technical report, Computational Logic, Inc., Austin, Texas, 1987.

[24] J. McHugh, R. L. Akers, and C. D. Singer. Gypsy information flow tool software requirements specification. Technical report, Computational Logic, Inc., Austin, Texas, 1987.

[25] J. McHugh and D. I. Good. An information flow tool for gypsy: Extended abstract. In *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, pages 46–48, 1985.

[26] R. Reitman and G. Andrews. Certifying information flow properties of programs: an axiomatic approach organization. In *Conference Record of the Sixth annual ACM Symposium on the Principles of Programming Languages*, pages 283–290. ACM-SIGPLAN, January 1979.

[27] J. M. Rushby. Mathematical foundations of the mls tool for revised special. Technical report, Computer Science Laboratory, SRI International, March 1984. Draft.

[28] O. S. Saydjari, J. M. Beckman, and J. R. Leaman. Lock trek: Navigating uncharted space. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 167–175, May 1989.

[29] C. D. Singer. An extension of the gypsy information flow semantics for dynamic and indexed types. Master's thesis, Duke University, Dept. of Computer Science, 1988.

[30] M. Smith. Model and design proof in gypsy: An example using bell and lapadula. ICSC Internal Note 122, Institute for Computing Science University of Texas at Austin, 1984.

[31] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(7):352–357, July 1984.