A Security Domain Model to Assess Software for Exploitable Covert Channels

Alan B. Shaffer
Naval Postgraduate School
Computer Science Dept.
Monterey, CA, USA
abshaffe@nps.edu

Mikhail Auguston
Naval Postgraduate School
Computer Science Dept.
Monterey, CA, USA
maugusto@nps.edu

Cynthia E. Irvine
Naval Postgraduate School
Computer Science Dept.
Monterey, CA, USA
irvine@nps.edu

Timothy E. Levin
Naval Postgraduate School
Computer Science Dept.
Monterey, CA, USA
levin@nps.edu

Abstract

Covert channels can result in unauthorized information flows when exploited by malicious software. To address this problem, we present a precise, formal definition for covert channels, which relies on control flow dependency tracing through program execution, and extends Dennings' and subsequent classic work in secure information flow [9][40][30]. A formal security Domain Model (DM) for conducting static analysis of programs to identify covert channel vulnerabilities is described. The DM is comprised of an Invariant Model, which defines the generic concepts of program state, information flow, and covert channel rules; and an Implementation Model, which specifies the behavior of a target program. The DM is compiled from a representation of the program, written in a domain-specific Implementation Modeling Language (IML), and a specification of the security policy written in Alloy. The Alloy Analyzer tool is used to perform static analysis of the DM to automatically detect potential covert channel vulnerabilities and security policy violations in the target program.

Categories and Subject Descriptors D.2.4 [**Software Engineering**]: Software/Program Verification – *assertion checkers*; D.3.1 [**Programming Languages**]: Formal definitions and theory – *semantics, syntax*; D.3.4 [**Programming Languages**]: Processors – *compilers*; D.4.6 [**Operating Systems**]: Security and Protection – *access controls, information flow controls.*

General Terms Design, Languages, Security, Verification.

Keywords Security domain model, static analysis, automated program verification, specification language, covert channel, dynamic slicing.

1. Introduction

Identification of exploitable covert channel vulnerabilities is vital in the development of systems intended to enforce mandatory access control policies, and is required for the successful evaluation of such systems at the highest levels of assurance [27][5]. This paper presents a precise, formal definition for various types of covert channels, which depends upon a representation of control flow dependencies, thus extending classic work in this area [9][40][30]. A security domain model is described for formally representing different types of covert channels, and for conducting static analysis[1] of certain program implementations. This model employs dynamic slicing techniques to analyze programs for the existence of access control flaws, where appropriate.

Widely accepted evaluation standards [6][27][5] require that high assurance secure systems be designed, developed, verified and tested using rigorous processes and formal methods. This evaluation process must include demonstration of correct correspondence between system representations at various levels of abstraction, e.g., security policy objectives, security specifications, and program implementation. The Common Criteria for Information Technology Security Evaluation requires that systems at EAL-5 or higher[2] undergo covert channel analysis to ensure that the system is capable of enforcing its security policy in terms of covert as well as overt interactions [5].

Formal security models are often based on concepts of program secure state and state transitions. High assurance evaluation standards [6][5] require a formal verification that the state transitions resulting from program execution preserve the security properties defined by a policy. The approach described here analyzes programs for preservation of security properties through state transitions, and specifically for the existence of covert channel vulnerabilities. This work advances the concepts of secure information flow in classic work by Denning and others [9][40], by describing automated techniques for covert channel static analysis.

Previous work in developing our approach has demonstrated the ability to detect overt information flow security violations [33]. The current work progresses toward verification of programs for the existence of covert channel security vulnerabilities, as well as *overt flaws* based on control flow dependencies. Covert channels are categorized here as storage channels and timing channels.

The *Implementation Modeling Language* (IML), the first novel element in this approach, is a language that supports basic information processing via assignment statements, conditional and loop statements, read/write statements, file random access, and access to a system clock. Program implementations represented in IML are called *base programs*, and they provide a standardized notation for conducting static analysis of target programs for adherence to a security policy.

The second novel element in this work is the definition of a security *Domain Model* (DM), represented as an Alloy [1][16] specification. The DM provides a framework for specifying

PLAS'08 June 8, 2008, Tucson, Arizona, USA.

[1] In this context, *static analysis* refers to analysis of program code without actual program execution.
[2] EAL-7 is the highest evaluation assurance level.

program state and state transitions, as well as security-related concepts such as security policy, information flow, access control, and covert channel vulnerabilities. Because of decidability issues associated with modeling arithmetic operations, Alloy by design supports only a limited representation of integers and basic arithmetic operations. Since the DM is implemented using Alloy, it is similarly limited.

Our Security DM is comprised of an *Invariant Model*, which defines the generic concepts of program state, information flow, and security policy; and an *Implementation Model*, which specifies the behavior of the base program. A specialized *DM-Compiler* was developed to translate a base program in IML into an Implementation Model, and to integrate it with the Invariant Model to form a complete DM specification; the DM-Compiler thus has visibility of the security policy, as implemented in the Invariant Model. The DM is verified using the Alloy Analyzer, which identifies execution paths where the security policy rules are violated.

Whereas many previous security models capture information flow between *objects* and *subjects*, the DM does not explicitly define an object, but implements this concept through variables. An access table records sensitivity labels for program variables, as a means of tracking information flow across state transitions. These labels indicate the sensitivity of data stored within a variable, and may change over time as data flows through the system.

The DM captures the concept of information flows with respect to a system subject for input to and output from an external device or random access file. The subject is essentially the executor of the statement, and has a defined access label. The policy rules define the legal information flows, based on the relationships between the subject label and the I/O source/destination variable label, e.g., in a Write_dev operation, a subject label must dominate a source variable label, in order for the variable to be successfully accessed for writing. This requirement might seem counter to the BLP *-property, however in our approach a Write_dev is modeled as a flow from a source variable to a target device, with the latter specified at the level of the subject label.

Section 2 of this paper describes the IML syntax. Section 3 provides background discussion on covert channels, control flow dependencies, and dynamic program slicing. Section 4 presents an overview of the DM methodology for modeling a security policy. Sections 5 and 6 demonstrate analysis of several example base programs using the DM, and summarize our test results with these examples. Sections 7 and 8 discuss related previous work, and planned future work in this research.

2. Implementation Modeling Language (IML)

The Implementation Modeling Language (IML) defines a simple domain-specific language that presents some of the basic capabilities and constructs, with respect to security, of high-level programming languages. Our intent is that IML enables the specification of relatively simple programs written in some common programming language, such as Ada, Java, or C++. While future iterations of IML might handle other more advanced language features, e.g., concurrency, inheritance, etc., this initial language description was motivated by a requirement to represent essential security information flow properties in target program implementations, balanced by the desire to limit complexity during experimentation.

2.1 Lexical Concepts

A variable name is an identifier distinct from IML keywords and Alloy keywords. No variable declarations are required.

The only assumption about values stored in variables is that they can be compared for equality and inequality (<, =, >, <=, >= operators) with other variables, or with constants. Variables can hold integer constants, but the value of a variable can be interpreted also as a time value (see `GetClock` below). Constants are represented by integers: `-1, 0, 1,` etc.

Statements provided in the IML include capabilities for assignment to a variable, reading to and writing from a variable, accessing an I/O device's flags and a system clock, and basic control structures. Semicolons separate statements in IML.

2.2 Assignment Statements

Assignment statements propagate access labels from the right-hand side to the left-hand side of the statement. For the current model, constants have a *Low* access label by default.

variable `:=` *variable*`;`
variable `:=` *constant*`;`

2.3 Device Input/Output Statements

`Read_dev` and `Write_dev` statements abstract the input from and output to an external device at a specific access level. We make the simplifying assumption that there are two external devices: one *High* and one *Low*; the label (*High* or *Low*) indicates which device. For a `Read_dev` statement, the variable is assigned the label of the device that is read from; for a `Write_dev` statement, source may be either a variable or a constant.

`Read_dev (`*label*`,` *variable*`);`
`Write_dev (`*label*`,` *source*`);`

2.4 File Random Access Statements

The IML abstracts the concept of random access to an indexed file, where *(key, value)* pairs are used to store and retrieve information in a finite-sized repository. This conceptual repository, referred to as a *direct file*, can be thought of as a database or memory file and (for this model) is represented as a single-level store (i.e., there is no distinction between persistent and volatile memory).

All subjects in the base program can access a single instance of the direct file, according to their access label. Initially, all direct file slots have a *Low* access label, and can be written to by any subject. Once a subject has stored a value into a keyed slot using the `PutDirectFile` statement, that slot retains the label of the subject. Subsequently, another (or the same) subject may read from this direct file slot using the `GetDirectFile` statement, only if the subject's label dominates that of the key slot. A given key slot can be overwritten an unlimited number of times by a subject with a higher- or lower-labeled value, so the label of a given slot may change over time.

The direct file has a limited number of keyed slots, all of which have empty keys and values at the start of program execution, and a given slot's key value is determined when it is first assigned a key/value pair. The direct file tracks the number of slots that have been assigned a key, zero at the start of execution and incremented by one whenever a key slot in the direct file is written to for the

first time. The direct file capacity equates to the number of key slots that can be allocated in the direct file.

When a `PutDirectFile` is executed for a given key for the first time, an available key slot is allocated, the data is stored in the direct file, and a global *Success* flag is set to 1; otherwise, if no key slot is available, the *Success* flag is set to 0, and no data is stored. When all available slots have been allocated, the direct file is considered filled, and a global *Full* flag is set to 1. The *Success* and *Full* flags are global state variables maintained by the execution environment, and are internal resources that would not be directly accessible in a high level language. Their values could be inferred, however, based on system errors seen by the user, and we abstract such system errors in the IML by allowing direct examination of the flags in a base program.

The following statements are provided in the IML for storing and retrieving values to/from the direct file. The label indicates the level of the subject performing the operation; the key and source may be either variables or constants:

```
GetDirectFile (label, key, variable);
PutDirectFile (label, key, source);
```

2.5 GetClock Statement

This statement stores the current clock value to a variable:

```
GetClock (variable);
```

We model only the time taken by file and external device accesses, i.e., during `Read/Write_dev` and `Get/PutDirectFile` operations. These statements may cause the CPU, or some other resource, to be busy such that some action visible to another subject is delayed with respect to a reference clock (for simplicity, we model one time source – the system clock).

The clock value can be compared with other constants and variables, using the `Before` operator:

```
(var1 Before var2)
```

2.6 Control Statements

A conditional expression is constructed from variables, constants, flags, and operators =, >, <, >=, <=, `Before`, `not`, `and`, `or`. A *statement* may be any statement or block of statements (a sequence of statements is enclosed by braces). Two forms of control statements are provided:

```
if condition then statement [else statement];

while condition do statement;
```

In the `if-then-else` statement, the `else` block is optional. The `while-do` control statement repeats its body as long as the *condition* holds true.

The following statement signifies termination of a base program:

```
Stop;
```

3. Background

We now discuss here several computer security concepts relevant to this work.

3.1 Covert Channels

Covert channels use entities other than data objects as a way to transfer information between system subjects, specifically entities not intended for information transfer [20][17]. Such channels allow processes to take advantage of communication channels to transfer information in a manner that violates a security policy [11].

An operating system may virtualize a shared physical resource so that each subject, or equivalence class of subjects, perceives that it has exclusive access to the resource. A covert channel can result from the incomplete virtualization of a resource such that some *attribute* of the resource remains shared, indirectly.

A common taxonomy of covert channels defines them as being either *storage* or *timing* channels [32]. For both storage and timing channels the sender and receiver (typically subjects) must have [17]:

1. Indirect access to an attribute of a shared resource, which the sender can modify, and the receiver can view. For example, the shared resource is the CPU, and the attribute is its "busy" state; or the shared resource is the disk, and the attribute is the location of the disk arm, or the attribute is the "full" state.
2. A means to initiate and synchronize their actions. The sender and receiver need to know when to modify and observe the attribute, the importance of which increases when they wish to transmit a stream of data.

In our analysis, we consider that the primary distinction between a covert storage channel and a covert timing channel is the means by which the receiver observes the change in the attribute:

3. Storage – the receiver views an error message, or other information placed in its address space by the system. E.g., if the disk is full, the receiver is provided an error message to that effect.
4. Timing – the receiver views changes to the relative timing of "legal" events. For example, if the sender's activity makes the CPU busy, the receiver's request to execute an operation on the CPU will complete (event 1) after the expected time of day occurs (event 2); or, turning to the disk arm attribute, depending on where the sender has left the arm (e.g., by reading a sector near the inner or outer edge of the disk), two disk sectors read by the receiver will occur in a different order (events 1 and 2).

The attribute in question forms a *point of interference* [14] between the subjects. To be the basis for an exploitable covert channel, the interference must also be contrary to the computer security policy – i.e., with a mandatory access control (MAC) policy, the sender's security level must be higher than the receiver's level (with respect to confidentiality) [38]. Thus, the determination of the potential covert channels in a system depends not only on the policy in place, but also on the implementation of that policy on a specific system [11], thus our approach here considers both the security policy and its implementation.

The criteria listed above enable one or more bits of information to be passed for each interference event (i.e., log2(n) bits, where n is the number of possible states that the observer can differentiate in the shared resource, such as different amounts of delay).

However, if the interference event can be repeated in a cycle, or loop, a stream of data can be transmitted through the channel, although additional synchronization between sender and receiver may be required to do so.

The point of interference of a covert channel is considered an *internal resource* of the system, as it is not directly accessible to subjects, as are *exported resources* [27]. Note that if a *Low* subject can directly view the value of an exported resource (e.g., a variable) that has been modified by a *High* subject then an overt flaw, rather than a covert channel, results.

3.2 Control Flow Dependency Flaws

Covert storage channels based on control flow dependencies often involve the indirect use of internal resources, such as buffers or non-exported files in a program control decision, to pass information from *High* to *Low* [20][17][1][10]. In addition to this, our approach is capable of detecting overt flaws based on control flow dependencies.

The approach here for discovering flaws based on control dependencies employs a dynamic slicing analysis. To determine the existence of such a dependency within the program, the chain of statements preceding a value assignment is examined with respect to the access labels of the variables in these statements. If the context of a previous statement includes variables that are higher than the destination, then there is an overt flaw.

The code snippet below would not be classified as having a covert channel since internal attributes are not referenced, however it provides an illustration of a control flow dependency that constitutes an overt flaw. In the example, a constant value is written out to a *Low* external device (s3), depending on the *High* value read into variable `v1` (s1).

```
(s1) Read_dev (High, v1);
(s2) if v1 > 0 then
(s3)    Write_dev (Low, 1);
```

The *Low* value assignment depends on a *High* source (`v1`) in the `if` block (s2), therefore an *implicit flow* from `v1` to the *Low* device exists [30].

3.3 Dynamic Slicing

Integral to certain covert channels is the notion of data or control dependency. *Slicing* algorithms are used as a means of tracing such dependencies between variables and statements processed during program execution, traditionally for program debugging purposes [18]. Slicing algorithms generate an executable subset of a program, creating a subprogram whose behavior is the same as the original with respect to some variable. They allow one to isolate the behavior of, and dependencies acting upon, that variable.

Slicing algorithms are categorized as either *dynamic* or *static*, depending on whether they take into account dependencies derived during one particular program execution path (dynamic), or for all possible execution paths (static). Dynamic slicing techniques generally analyze only the narrow portion of the code representing a single execution path.

Since slicing techniques have been shown to be useful in tracking data and control dependencies, they can also provide a means of detecting potential overt flaws based on dependencies. As an example, consider the following code snippet:

```
(s1) if v3 > 17 then
(s2)    v1 := 0;
(s3) else if v4 = 5 then
(s4)         v1 := 1;
(s5)      else v1 := -1;
(s6) v2 := v1;
```

In the example, it is fairly clear that `v2` depends on `v1` (s6). Static slicing can show that `v2` has a dependency on both `v3` (s1) and `v4` (s3), since there is a dependency from each of these to `v1`. With dynamic slicing, however, not all execution paths will result in the same control dependencies, e.g., when the conditional expression in (s1) evaluates to true, the final value of `v2` depends on `v3` but not on `v4`, since (s3) is never executed.

The access labels of variables can be used to determine potential security violations, based on the dependencies between these variables. For a finite number of paths within a given scope (see 4.1.1), our tool performs static analysis of the DM by using dynamic slicing to discard previous states that could not have contributed to an overt flaw, thus a complete result is obtained without having to maintain a history of all preceding states.

4. Security Domain Model Methodology

An overview of the Security Domain Model (DM) approach to program security verification is depicted in Figure 1. The DM includes the definition of program state and transitions between states, as well as security rules, specified as Alloy assertions, representing the generic policy a program must conform to. The DM is composed of an invariant and a variable section, derived from the security rules and a target implementation, respectively.

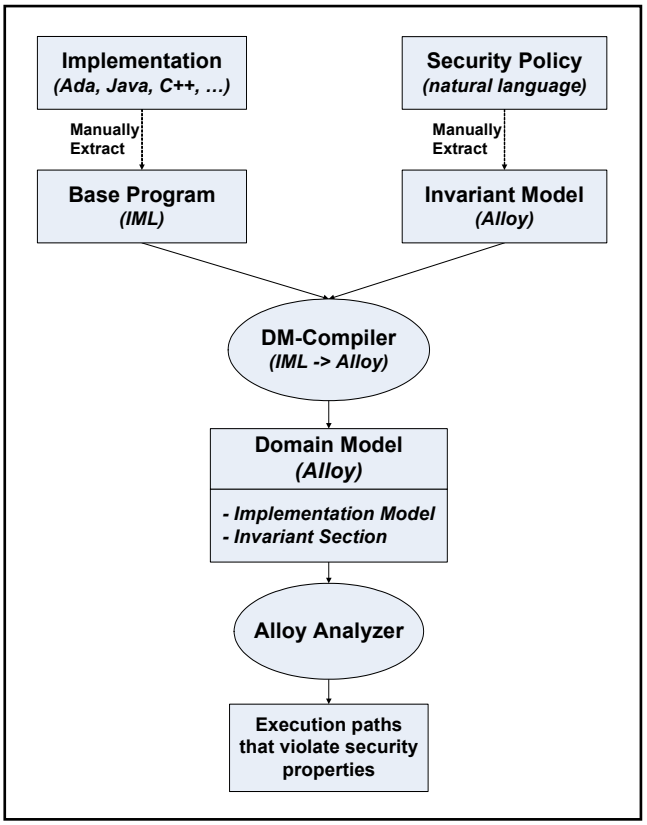

Figure 1. Domain model approach to system security verification.

While there are numerous model checker tools currently available, we chose to use the Alloy specification language primarily because of its ability to represent program language abstractions simply and explore their semantics with a well-integrated analysis tool. As Jackson [16] points out, referring to his approach as "lightweight formal methods," Alloy models can be easily created and initially tested early in the development process, and then incrementally expanded. He states that the goal of Alloy was to "obtain the benefits of traditional formal methods at lower cost, without requiring a big initial investment," presumably in time and effort [16].

As with traditional model checkers, Alloy deals with finite models, though it handles them very differently. Model checkers typically build Kripke structures to represent the states and transitions of a program execution. Such finite model structures have limits not easily adjusted by the user during analysis. The Alloy Analyzer tool, however, affords the ability to easily increase the depth of analysis for models as they are developed and expanded. For our approach, Alloy and its Analyzer provide an ideally suited tool for creating and analyzing target program abstractions.

In our approach, a *base program* is an abstraction of a target program implementation, and is written in IML notation. By analyzing a model of the program, rather than actual program code, security verification can focus on elements of information flow including covert channel analysis, e.g., I/O, access labels, direct file access, and timing (clock), while ignoring other program details not pertinent to such analysis.

In the current prototype, translation of the base program from an implementation is a manual step. Developing a separate compiler to translate a high-level language program to IML is a difficult task, beyond the scope of this work. The possibility must be considered that covert channels existing in the original program implementation may be lost in the IML representation, and for now we depend on the knowledge of the manual translator to avoid this problem.

Security rules, written as Alloy assertions, are derived from the security policy. Such policies are typically written in natural language, and extraction of security rules is a manual step in our approach. As currently implemented, the DM defines security rules, which have as their basis the Bell & LaPadula security model [2], i.e., flows from *High* to *Low* secrecy levels are not allowed.

After the base program and Invariant Model with security rules are defined, the DM-Compiler compiles the base program from IML into state transition predicates, written in Alloy notation, creating the DM Implementation Model. The DM-Compiler combines this with the Invariant Model to complete the DM. The approach uses the Alloy Analyzer tool [1] for automated verification of the security rules, defined in the DM as Alloy assertions, to find execution paths within the DM that might violate the security policy or create covert channels. In essence, it creates an interpreter for the specific base program, modeled by the DM.

4.1 Domain Model Structure

The following provides an outline of the Domain Model structure.

4.1.1 DM Invariant Model

The Invariant Model specifies the conceptual framework of the DM with the Alloy specification language. This section describes statement types and structure, program execution state, direct file structure, and clock signature.

In the Alloy language, all atomic structures are modeled as sets and relations. Sets are represented as unary relations; scalars are simply singleton sets. A set or relation declaration can be constrained using several keywords indicating multiplicity: `one` restricts sets to exactly one instance of a type; while `lone` restricts them to either zero or one instance; and `none` refers to the empty set. The `all` quantifier must hold for all instances of a type, and the `disj` quantifier specifies variables that are necessarily disjoint from one another.

Alloy provides standard logical operators, e.g., negation (`!`), conjunction (`&&`), disjunction (`||`), implication (`=>`), and bi-implication (`<=>`). Pairs `(type->type)` represent binary relations, and '+' is the set union operator. The override operator '++' examines two sets of pairs and overwrites the pair in the first set with the second whenever the first elements of the pairs match. The '^' operator represents transitive closure for binary relations.

The signature (`sig`) construct in Alloy, roughly synonymous with the class declaration in object-oriented programming languages, defines a set of atoms (elements), and any relations between them. Signatures with the `abstract` qualifier cannot have their own instances, and are used only to derive other signatures. For further details on the Alloy language, see [16].

The signatures below describe program `State`, the initial state, and structures for variables and values, which are extended in the DM-Compiler generated Implementation Model (discussed in Section 4.1.2). The `Dominates` signature defines a partial ordering between access labels. For simplicity, only *Low* and *High* access labels are defined here.

```
abstract sig Variable{}
abstract sig Value{}
abstract sig AccessLabel{}
one sig High, Low extends AccessLabel{}
one sig Dominates {
   ord: AccessLabel -> AccessLabel
}
{ ord = (High -> Low)

}
```

The `Statement` abstract signature captures a single instance of a given statement. For I/O (Read_dev/Write_dev) and direct file access statements, the signature defines statement `type`, `destination`, `source`, `key` (for direct file only) and `subject_label` attributes. The `subject_label` specifies the security label of the calling subject for a particular statement; this label represents the access label of the device, in the case of I/O statements. For assignment statements, only `source` and `destination` attributes are defined. For conditional statements, the `source` attribute defines the set of control variables used in an `if-then-else` or `while-do` statement. For GetClock statements, only the `destination` attribute is defined, while the Stop statement defines no attributes.

```
abstract sig Statement{
   type:           Stmt_type,
   destination:    lone Variable,
   source:         set Variable + Value,
   key:            lone (Variable + Value),
   subject_label:  lone AccessLabel }
```

The `Stmt_type` abstract signature is extended to include all statement types that can be used in a base program.

```
abstract sig Stmt_type {}
one sig Assign, Condition,
   Read_dev, Write_dev,
   PutDirectFile, GetDirectFile,
   GetClock, Stop
extends Stmt_type {}
```

The `DirectFile` signature defines key/value pairs (`keyContent`) for each of its storage slots, and the current access label (`keyLabel`) for each key slot value. The latter is used to track the label of the current value, to ensure valid flows during subsequent attempts to access that value, i.e., `GetDirectFile` statements. The element `last_written` stores the label of the last subject that wrote to the direct file, and is used when checking for potential covert storage channels (discussed in detail later). The signature also defines the direct file `max_slots` (set to 2 for modeling purposes); note that Alloy provides the predefined type `Int` to represent sets of integer atoms. Also, `full` and `success` are used as internal resource system flags, as previously described in Section 2.4.

```
sig DirectFile{
   keyContent:   Value -> lone Value,
   keyLabel:     Value -> lone AccessLabel,
   last_written: lone AccessLabel,
   max_slots:    Int,
   full:         (const0 + const1),
   success:      (const0 + const1)
}
{ max_slots = 2 }
```

The `Clock` signature provides an abstraction for program execution time. The signature defines the concept of some event occurring at some time before another event (`before` relation), which enables testing for the relative timing of events during base program analysis. In this implementation, '`TO`' is defined as a `Time` ordering instance, using the Alloy library utility for ordering. The `nexts` function returns a set of all next values in an ordering – in this case the next `Time` values after the one in question. For example, the code below checks whether `t2` is contained within the set of time values that occur after `t1`.

```
sig Time{}
one sig Clock{
   before:  Time->Time
}
{all disj t1, t2: Time |
  (t1->t2) in before <=> t2 in TO/nexts[t1]}
```

The `State` signature captures the current state of the system, and the next statement (`stmt`) to be executed.

```
sig State{
   stmt:              Statement,
   vars:      Variable-> one (Value + Time),
   access_label: Variable-> one AccessLabel,
   direct_file:       DirectFile,
   current_clock:     Time,
   prev_state:        lone State,
   last_cond_checked: set State,
   influenced_by:     Variable -> State }
```

The `State` sig includes the current type of statement being executed, the current table of variable values, the access label for each value stored in `vars` (for information flow tracing), and a snapshot of the current direct file; the flags `full` and `success` are contained within the `direct_file` attribute. This signature also includes the `current_clock` value, the previous state leading to the current state, and `last_cond_checked`, which identifies a set of conditionals within which the current statement may be nested, enabling dependencies from those conditionals to be propagated.

The `influenced_by` attribute is used for tracking control flow dependencies, and is at the heart of the dynamic slicing algorithm used in this approach. It stores, for each `source` variable in the current state, all of the previous states that have influenced that variable. This attribute enables the Alloy Analyzer to narrow its focus in examining previous states, thus reducing the search space necessary in determining control dependencies. By storing variable/state pairs, we can enable the Analyzer to examine all variable access labels from previous influencing states.

When analyzing a base program, the Alloy Analyzer performs an exhaustive search of all paths up to a defined length (the *scope,* specifying the size of the models considered). In fact, it performs symbolic execution of all base program paths with length up to the given scope limit. In our generated DM, the scope is generated heuristically, based on the total number of statements in the base program. This ensures that all execution paths of that length or less will be scrutinized. It is assumed that the Alloy *small scope hypothesis*, which states that most flaws in models can be revealed on small instances [16], holds for information flow tracing in our approach.

The Invariant Model includes the definition of security rules that must be enforced by the DM security policy. These rules are specified as Alloy assertions, and will be described further in Section 5.

4.1.2 DM Implementation Model.

The Implementation Model of the DM is automatically generated by the DM-Compiler from a base program, and specifies the base program's semantics in terms of statement signatures and state transitions. Example base programs, and their resultant compiled Alloy models, are presented in Section 5.

From the base program, the DM-Compiler generates `Variable` and `Value` signatures. The number and value of constants defined in the signature depend on the number and value of unique constants explicitly present in the base program (the constant 0 will always be added by default for initial variable values); similarly, the variable signatures reflect the variables used in the base program. To represent the state space, additional constants may be needed to fill the intervals between explicitly defined constants. The DM-Compiler defines an Alloy signature that establishes a simple *less than* relationship between the required constant values, thus enabling the base program to compare values for equality and inequality. Some examples are:

```
one sig x1
   extends Variable {}
one sig  const_minus_1, const0,
   const1, const2, const3
   extends Value {}

one sig LT {
   lt:  Value -> Value }
{ lt = ^(
      ( const_minus_1 -> const0)
    + (  const0       -> const1)
    + (  const1       -> const2)
    + (  const2       -> const3) ) }
```

The DM-Compiler compiles each base program statement into a separate Alloy signature, based on the type of statement and associated variables and constants used. Elements of the `Statement` signature not needed for a particular statement type are not initialized. The base program example below implies a signature sequence that would be compiled from an assignment statement (s2) nested within a conditional statement (s1):

-- *Base Program statements*

```
(s1) if ( x1 < 0 ) then
(s2)   x2 := x1;
(s3) Stop;
```

-- *DM-Compiler-generated Alloy signatures*

```
one sig s1 extends Statement {}
{ type = Condition
  source = x1
  destination = none
  key = none }

one sig s2 extends Statement {}
{ type = Assign
  source = x1
  destination = x2
  key = none }

one sig s3 extends Statement {}
{ type = Stop
  source = none
  destination = none
  key = none }
```

From these statement signatures, the DM-Compiler generates a *transition predicate* representing the state transition trace for the base program execution. The transition predicate captures the semantics of the base program by specifying all possible sequences of statement executions for the program. It also implements dependency tracking within the execution path. Although we refer generally to the transition "predicate," we actually represent this structure using an Alloy `fact` rather than a predicate (`pred`). Although both define Alloy constraints, a `pred` only holds when invoked, while a `fact` is assumed to always hold.

The remainder of this section shows a representation of the state transition predicate derived by the DM-Compiler for the base program above. Note that for each statement, `pre` represents a state before its statement (`stmt`) has been executed, and `post` represents the state after statement execution.

```
fact trans {
 all post: State - InitialState |
     some pre: State |
```

For the conditional statement (s1), since no variable value assignments are made, the variable table, access labels, direct file (including system flags), clock time value, and `influence_by` table remain the same after execution:

```
(pre.stmt = s1 &&
  (post.vars = pre.vars &&
   post.access_label = pre.access_label &&
   post.direct_file = pre.direct_file &&
   post.current_clock = pre.current_clock &&
   post.influenced_by = pre.influenced_by &&
```

The `last_cond_checked` attribute is calculated to include all previous states currently in `last_cond_checked` (excluding the current state, s1), plus the `pre` state itself, in order to set the context of statements within the conditional:

```
   post.last_cond_checked =
     {cond: pre.last_cond_checked |
         cond.stmt != s1 } + pre &&
```

Based on the outcome of the conditional check, the next statement to execute is set to either the "then" branch (s2), or the "else" branch (s3) statement:

```
   (((pre.vars[x1]-> const0) in LT.lt)
        => post.stmt = s2
      else post.stmt = s3)
   )
   && post.prev_state = pre
) ||
```

In the assignment statement (s2), the access label and value for the target variable (`x2`) are set to those of the source variable (`x1`):

```
(pre.stmt = s2 &&
  (post.vars = pre.vars
     ++ (x2 -> pre.vars[x1]) &&
  post.access_label = pre.access_label
     ++ (x2-> pre.access_label[x1]) &&
  post.stmt = s3 &&
```

The direct file (including system flags), clock value, and `last_cond_checked` attribute all remain the same after execution of an assignment:

```
   post.direct_file = pre.direct_file &&
   post.current_clock = pre.current_clock &&
   post.last_cond_checked =
     pre.last_cond_checked &&
```

The `influenced_by` attribute is calculated based on the source variable dependencies. Recall that `influenced_by` is declared within the `State` sig as the relation (*Variable->State*), which is a set of pairings from variables to states. Alloy treats sets and subsets the same when defining relations, thus the pairing of a variable to a set of states (Variable->{State}), shown below, denotes a set of pairings from that variable to each of the states ({Variable->State}). See Section 3.2.2 of [16] for a full discussion of Alloy's treatment of sets as relations.

In calculating `influenced_by`, first all previously recorded dependencies (other than those for `x2`, the destination variable) are included:

```
post.influenced_by =
 {v: Variable, s: State |
   (v -> s) in pre.influenced_by &&
    v != x2 }
```

Second, dependencies for `x1`, the current assignment statement source variable, are added as dependencies for `x2`:

```
+ (x2 -> pre.influenced_by [x1])
```

Next, from the current set of states defined in `last_cond_checked`, those whose scope this assignment falls within are included. This captures dependencies from any conditional within which the current statement may be nested; in this case base program statement (s1):

```
+ (x2 -> {cond: pre.last_cond_checked |
   cond.stmt = s1})
```

Finally, when an assignment statement is nested within a conditional statement, dependencies from the `source` variables participating in the conditional must be included:

```
+ (x2 -> State.
   {cond: pre.last_cond_checked,
    infl: cond.influenced_by
    [cond.stmt.source] | cond.stmt = s1}
  )
) && post.prev_state = pre
) ||
```

The transition predicate concludes with the Stop statement (s3). Since execution terminates when this point is reached there is no need to assign values for the resultant (i.e., `post`) state, other than setting the previous state for tracing continuity:

```
(pre.stmt = s3
    && post.prev_state = pre
)}
```

5. Examples of DM Analysis

This section presents three examples of overt flaw and covert channel vulnerabilities, discoverable using the DM approach. In each case, a rule for discovering the potential flaw or covert channel is defined as an Alloy assertion, and an example base program is presented to illustrate the problem. Each example shows the transmission of one bit of information; a more complex example would need to involve such concepts as looping, synchronization, etc., to provide the covert channels with a stream of bits.

The complete Alloy models for these and other examples can be found on our website at http://cisr.nps.edu/projects/sdm.html.

5.1 Overt Flaw (Example 1)

The first example illustrates an overt flaw based on a control flow dependency. This example shows an exploitation scenario that culminates with an IML `Write_dev` operation, where the variables written to the external device have been influenced by values at a higher level than that of the device itself.

The Alloy predicate below examines each execution state, and evaluates as true whenever the state (`current`) is the result of a `Write_dev` statement, and the value to be written out was `influenced_by` some previous state (`pre`) that had access to a variable with a higher `access_label` than that of `current` state, i.e., the flow from the previous State to the current State is from *High* to *Low*.

```
pred dependency_flaw_found [current: State]{
  let stm = current.stmt,
    pre = current.influenced_by[stm.source]|
  (stm.type = Write_dev  &&
  ((pre.access_label [pre.stmt.source])
     -> stm.subject_label) in Dominates.ord
) }
```

The following base program illustrates an example of this flaw. Initially, a *High* value is read into variable `x1` (s1). Based on the value of `x1` (s2), new variable `x2` is assigned either '0' (s3), or '1' (s4). Variable `x2` is then written to a *Low* device (s5).

The violation occurs when `x2` is written to a *Low* device, because its value has been potentially influenced by a *High* value, specifically `x1` when it was accessed in (s2).

```
(s1) Read_dev (High, x1);
(s2) if x1 = 0 then
(s3)    x2:= 0;
(s4) else x2:= 1;
(s5) Write_dev (Low, x2);
(s6) Stop;
```

The Alloy Analyzer detects this situation, and correctly reports an overt flaw by tracing the control flow through statements (s1)(s2)(s3)(s5).

5.2 Storage Covert Channel (Example 2)

The second scenario describes a classic covert storage channel [23] resulting from access to the direct file by a *Low* subject (using a `PutDirectFile` operation) after a *High* subject has caused it to become full (represented by a successful `PutDirectFile` that consumed the last open key slot). The security predicate below defines this vulnerability by checking for states where a `PutDirectFile` is being attempted into a full direct file by a subject whose security level (`subject_label`) is dominated by that of the most recent subject to store a value to the direct file (`last_written`):

```
pred storage_channel_found [current: State]{
  let stm = current.stmt | {
    stm.type = PutDirectFile &&
    current.direct_file.full = const1 &&
    ((current.direct_file.last_written)
      -> stm.subject_label) in Dominates.ord
} }
```

The following example base program illustrates a direct file storage channel. For the example, we assume a direct file with a capacity of two records, initially empty. The first part of the program shows events that occur representing a covert channel sender. A *High* value is first read into variable `x1` (s1). Based on a check of the value of `x1` (s2), constant value '2' with a *High* access label is stored into the direct file at key slot 1 (s3), and then constant '0' with a *High* access label is stored into the direct file at

key slot 2 (s4); these statements result in the internal *full* flag being set.

```
(s1) Read_dev (High, x1);
(s2) if x1 = 1 then {
(s3)   PutDirectFile (High, 1, 2);
(s4)   PutDirectFile (High, 2, 0); }
```

The latter part of the program represents statements executed by a *Low* covert channel receiver. When the *Low* subject attempts to store a value into direct file key slot 3 (s5), the system issues a failure indication since the direct file is *full*. The IML abstracts this system failure by allowing direct examination of the system *full* flag by the *Low* subject (s6). To exploit the storage channel, *Low* writes a constant '1' or a '0' to an external device (s7 & s8), depending on the result.

```
(s5) PutDirectFile (Low, 3, 1);
(s6) if full = 1 then
(s7)   Write_dev (Low, 1);
(s8) else Write_dev (Low, 0);
(s9) Stop;
```

The nexus of this covert channel is that *High* can write to the internal resource *full* (indirectly), and *Low* can observe it. Because a higher labeled subject caused the direct file to become full, the Alloy Analyzer detects and reports this violation, tracing the flow of execution through statements (s1)(s2)(s3)(s4)(s5).

5.3 Timing Covert Channel (Example 3)

The third scenario describes a covert timing channel that occurs when a *Low* subject executes two `GetClock` statements (`gc1` & `gc2`), and between them a *High* subject prevents the *Low* subject from executing, through execution of a Read_dev/Write_dev or direct file operation (`rw` state in the text). Thus, when the *Low* subject next runs, it can examine the clock to detect this interference with its access to the CPU; these channels are thus often called *CPU channels*. The Alloy assertion below will detect this potential covert timing channel, which utilizes the system `Clock`.

```
pred timing_channel_found [gc2: State]{
  some disj rw, gc1: State | {
   gc2 -> rw in State_order.st_after &&
   rw -> gc1 in State_order.st_after &&
   gc1.stmt.type = GetClock &&
   gc2.stmt.type = GetClock &&
   rw.stmt.type in
     (Read_dev + Write_dev +
      PutDirectFile + GetDirectFile) &&
   gc1.stmt.subject_label =
      gc2.stmt.subject_label &&
   (rw.stmt.subject_label
      -> gc2.stmt.subject_label)
      in Dominates.ord  }

}
```

The base program below illustrates this timing channel. A *High* value is initially read into variable `x1` (s1). A *Low* subject then stores the current clock value in `t1` (s2). Based on a check of `x1` (s3), its value is stored into the direct file at key slot 1 (s4). The *Low* subject again examines the clock, and stores its value into `t2` (s5).

```
(s1) Read_dev (High, x1);
(s2) GetClock (Low, t1);
(s3) if x1 < 0 then
(s4)   PutDirectFile (High, 1, x1);
(s5) GetClock(Low, t2);
```

At this point a timing channel has occurred, and the Alloy Analyzer detects the violation, tracing execution flow through statements (s1)(s2)(s3)(s4)(s5). The crux of this covert channel is that a *Low* subject, the covert channel receiver, has been allowed to observe (by examining the clock) a change in some internal resource (the CPU busy state), which was indirectly affected by the actions of a *High* subject, the covert channel sender. The remaining statements illustrate how the *Low* subject compares the two clock values (s6) to see whether the *High* subject has interfered with it through performance of some operation, and writes either a '1' or '0' accordingly (s7 & s8).

```
(s6) if t1 Before t2 then
(s7)   Write_dev (Low, 1);
(s8) else Write_dev (Low, 0);
(s9) Stop;
```

6. Testing Results

The base program examples presented above were evaluated using Alloy Analyzer 4.0 build RC17, running under Mac OS X™ 10.5.1 on a 2.4 GHz Intel Core 2 Duo processor, with 2 GB of memory. In test runs, the Alloy Analyzer successfully found valid counterexamples for violations of each security rule assertion, i.e., an existing overt flaw or covert channel was detected in each case. Test run times, in *ms*, were as follows; *total time (time to generate model, time to find counterexample)*:

- Example 1 (scope = 7): 688 (640, 48)
- Example 2 (scope = 10): 2284 (1953, 331)
- Example 3 (scope = 10): 5891 (2771, 3120)

7. Related Work

Previous research in modeling secure information flow and access control, and in covert channel analysis is described below. We have extended previous work by integrating a language for formally specifying an implementation with a framework for expressing security policies, particularly with respect to covert channel rules and control dependency flaws.

Classic work on secure information flow [8][9] provides a foundation for this research, including the notion of partial ordering of security classes based on the dominance relationship, the idea of labeling state variables to track such flows, as a way to certify a program.

Other approaches do not distinguish between classes of covert channels, or between covert and overt flows for that matter. These approaches rely on the concept of *noninterference*, which states that the actions of one subject can have no effect on the output of a lower subject in a system. Goguen & Meseguer [14] described a model wherein security policies are defined in terms of only noninterference assertions, rather than by the combination of access control and covert channel restrictions. Their ideas were further expanded by Haigh & Young [15]. Noninterference with respect to security properties, however, is considered to be limited by the *refinement paradox*, i.e., a system's abstract security properties for information flow cannot be guaranteed to be

preserved through refinement to concrete implementation [24][29][25].

Graham-Cumming & Sanders [12] used the *unwinding theorem* [13][15] to describe refinement of a system such that noninterference between users in an abstract specification of the system could be preserved through more concrete representations of the system, however the results of this work were limited to noninterference, and were not extensible to more general security policies or confidentiality rules, such as those for covert channels. Enforcement of a range of security properties is possible in our approach through the use of security assertions that are explicitly checked during Alloy program analysis. The semantics of information flow are represented in the DM by the definition of access label ordering, and through the compiler-generated transition predicate, unique to a particular base program.

Volpano et al [40] furthered the language-based flow analysis work by defining a linguistic type system for secure flow, and rigorously proving the soundness of the core language with respect to noninterference. Well-typed programs are then guaranteed to be noninterfering – and thus secure by this definition – which was the basis for much related research, summarized by Sabelfeld & Myers in their survey on language-based information flow systems [30].

Other work in using sound type systems for secure information flow has focused on areas such as: encryption and decryption of information, where flows from plaintext (*High*) information to ciphertext (*Low*) information must be addressed in light of noninterference rules that would seem to prevent such interaction [21][35]; *probabilistic noninterference*, where probability distributions are used to determine a likelihood of noninterference from *High* to *Low* variables, primarily for multi-threaded processes where scheduling is nondeterministic [39][31][36]; and *type inference*, in which the flow of information is automatically determined based on semantic analysis [34][7]. Eventually, Smith & Thober [37] enhanced the linguistic model of secure information flow such that sensitivity labels need be assigned only at I/O boundaries, while the labels of variables and constants, as well as data information flow through a program's execution, are automatically derived relative to the I/O (device) labels.

Our DM-Compiler similarly tracks the flow of data based on the input device label with no requirement to annotate the code in any other way. Our work differs from the linguistic type system approach in that, rather than constructing a type-safe language with which to write secure programs, we apply abstract interpretation to the analysis of programs in order to detect potential problems and otherwise demonstrate their security with respect to select security properties. Our approach is based on exhaustive information flow tracing of all execution paths in a program, to a certain length (determined by the model scope of Alloy). This tracing is applied for both overt and covert channel static analysis, using dynamic slicing techniques where appropriate such that read-up, as well as violations of noninterference, are detected [41]. Additionally, we provide a compiler to generate a formal specification of a program. Although it yet lacks a formal soundness proof, the DM-Compiler enables generation of formal logic that can be automatically analyzed (using the DM) for secure information flows.

Other covert channel research has also focused on information flow analysis, using the principles of Kemmerer's SRM [17]. For example, in one approach, covert channel analysis is based on observing the change in entropy, or uncertainty, of process variables that occurs when covert channels exist [10]. The concept of "network" covert channel analysis introduced detection methods based on in-depth IP packet analysis as a way to differentiate covert channels from legitimate network traffic [28][3]. Approaches such as these could potentially be incorporated into the DM security rule assertions, as methods for detecting covert channels in base programs within this domain.

Previous work in implementing dynamic slicing algorithms has included development of a tool for finding privacy violations in networked environments, targeting spyware in networked applications [19]. The approach uses dynamic slicing techniques to trace program execution, to search for data dependencies that might illuminate privacy violations. When such dependencies are found, they are specified using an event description language to capture event parameters, values, etc. Their goal is to use these parameterized events as abstract inputs for an event sequence language, as a means of generating a security policy. Based on the observed privacy violations, a policy is written that will prevent the specific events that caused the violations. Our suggested approach differs in its goal of analyzing a target program for adherence to a specific policy, as opposed to some generated policy.

The Alloy language has been used to model security *requirements* for secure communications [4], where predicates were specified for secure message confidentiality, integrity, authenticity and non-repudiation, as well as numerous "obstacles to security", e.g., eavesdropping or spoofing. The work was successful in designing a general, reusable model for communication security properties, and differs significantly from our research presented here, which examines program implementations for security.

8. Discussion and Future Work

This paper has presented ongoing research to develop a formal domain model for analyzing programs for exploitable covert channel vulnerabilities. The approach defines a formal Security Domain Model (DM) that facilitates detection of covert channels, independent of program implementation. This approach can also be defined to specify security policies in a generic way, beyond covert channels [33]. Although encoding and checking static program semantics and properties is not in itself revolutionary, this work is evolutionary in extending previous work in the area of information flow tracking based on a precise, formal definition for overt information flaws and covert channels. Our model provides a means of conducting automated static analysis of a program implementation within a finite scope of execution paths. Flow control dependencies and related overt flaws are analyzed using dynamic slicing techniques. This paper has shown the feasibility of this approach on a specific set of examples for which the Alloy Analyzer, within a finite scope, ensures that false negatives and false positives are eliminated.

Because the scope of analysis is directly related to the size of the base program being analyzed, a key problem with this approach lies in the limitation on the size of the possible search space due to the potential for state explosion in the analyzer tool. Extension of the analyzer tool could expand the scope, and enable analysis of larger programs.

Future work will focus on formally analyzing the semantics of the IML and DM-Compiler to ensure that the artifacts of each (e.g.,

the base program and DM Implementation Model) are accurate refinements of the original target implementation. As pointed out in [30], information flow analysis should take place "as close to the executed code as possible." Analysis of a compiled abstraction of the execution code creates a requirement for trustworthiness in the compiler, as well as in the process of deriving an abstraction of the target code itself.

The DM can be expanded to enable the notion of parameterized security policies, i.e., the security rules defined by a policy would become part of the DM Implementation Model (as opposed to the current Invariant Model). This would enable an evaluator to analyze a base program for consistency with a variety of security policies, the policies themselves being input parameters to the DM, as is done in a separation kernel [27]. Banking provides an example of this concept, where there is a need to maintain secrecy of data during execution of some applications (e.g. programs that manipulate customer personal data), while a need exists to maintain integrity in others (e.g. programs that execute money transactions). A DM with parameterized policies could analyze either of these categories of programs for adherence to the appropriate policy to be enforced. Additionally, the DM could be extended to provide generalization of security rule predicates, enabling detection of additional covert channels without the need for modification of the Invariant Model.

Another advance could be expansion of the DM to enable support for dynamic security policies [23]. This concept would allow the DM to support multiple polices in existence during program execution, with the ability of a system to adapt different policies based on a dynamically changing security environment [26].

9. Acknowledgements

The authors are grateful for support from the Office of Naval Research and the National Science Foundation under grant CNS-0430566. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ONR or the NSF.

References

[1] The Alloy Analyzer website. (2000). Retrieved January 11, 2008, from http://alloy.mit.edu/.

[2] Bell, D., & LaPadula, L. (1973). Secure Computer Systems: Mathematical Foundations and Model, *MITRE Report*. The MITRE Corp.

[3] Cabuk, S., Brodley, C., & Shields, C. (2004). IP covert timing channels: Design and detection. *Proceedings of the 11th ACM Conference on Computer and Communications Security* (178-187). Washington DC, USA: ACM Press.

[4] Chen, C., Grisham, P., Khurshid, S., & Perry, D. (2006). Design and validation of a general security model with the alloy analyzer. *Proceedings of the ACM SIGSOFT First Alloy Workshop* (pp. 38-47).

[5] *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model*, version 3.1. Document number CCMB-2006-09-001. September 2006.

[6] *Department of Defense Trusted Computer Security Evaluation Criteria*, DOD 5200.28-STD, National Computer Security Center, December 1985.

[7] Deng, Z., & Smith, G. (2006). Type inference and informative error reporting for secure information flow. *Proceedings of the 44th ACM Southeast Conference* (pp. 543-548). Melbourne, Florida.

[8] Denning, D. (1976). A lattice model of secure information flow. *Communications of the ACM*, 19(5), 236-242. ACM Press.

[9] Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 504-512. ACM Press.

[10] Gianvecchio, S., & Wang, H. (2007). Detecting covert timing channels: An entropy-based approach. *Proceedings of the 14th ACM Conference on Computer and Communications Security* (pp.307-316). Alexandria, VA, USA: ACM Press.

[11] Gligor, V. (1993). *A guide to understanding covert channel analysis of trusted systems*. Technical Rep. NCSC-TG-030, National Computer Security Center, Ft. Meade, MD, USA.

[12] Graham-Cumming, J., & Sanders, J.W. (1991). On the refinement of non-interference. *Proceedings of the Computer Security Foundations Workshop IV* (pp.35-42).

[13] Goguen, J., & Meseguer, J. (1984). Unwinding and inference control. *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 75-86). IEEE Computer Society Press.

[14] Goguen, J., & Meseguer, J. (1982). Security policies and security models. *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 11-20). IEEE Computer Society Press.

[15] Haigh, J.T., & Young, W.D. (1987). Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2), 141-150.

[16] Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA, and London, England: MIT Press.

[17] Kemmerer, R. (1983). Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3), August 1983. ACM Press.

[18] Korel, B., & Rilling, J. (1997). Dynamic program slicing in understanding of program execution. *Proceedings of the 5th International Workshop on Program Comprehension* (pp. 80-90). Dearborn, MI, USA: IEEE Computer Society.

[19] Kruger, L., Wang, H., & Jha, S. (2004). *Towards discovering and containing privacy violations in software* (Technical Report No. 1515). Madison, WI, USA: University of Wisconsin-Madison.

[20] Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM* 16(10), 613-615. ACM Press.

[21] Laud, P. (2003). Handling encryption in analyses for secure information flow. *Proceedings 12th European Symposium on Programming, ESOP* (pp. 159-173).

[22] Levin, T., & Clark, P. (2004). A note regarding covert channels. *Proceedings of the 6th Workshop on Education in Computer Security* (pp. 11-15). Monterey, CA, USA.

[23] Levin, T., Irvine, C., & Spyropoulou, E. (2006). *Quality of security service: Adaptive security*. Handbook of Information Security (H. Bidgoli, ed.), vol. 3, pp. 1016–1025, Hoboken, NJ: John Wiley and Sons.

[24] McLean, J. (1990). Security models and information flow. *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 180-189). IEEE Computer Society Press.

[25] McLean, J. (1996). A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1), 53-67. IEEE Press.

[26] National Security Agency IA Directorate. (2004). *Global Information Grid Information Assurance Reference Capability/Technology Roadmap*, Version 1.0.

[27] National Security Agency. (2007). *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Version 1.03.

[28] Padlipsky, M., Snow, D., & Karger, P. (1978). *Limitations of end-to-end encryption in secure computer networks*. MITRE Technical Report, MTR-3592, Vol. I, May 1978 (ESD TR 78-158, DTIC AD A059221).

[29] Roscoe, A. (1995). CSP and determinism in security modelling. *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 114-127). IEEE Computer Society Press.

[30] Sabelfeld, A., & Myers. A. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 5-19. IEEE Press.

[31] Sabelfeld, A., & Sands, D. (2000). Probabilistic noninterference for multi-threaded programs. *Proceedings of the IEEE Computer Security Foundations Workshop* (pp. 200-214).

[32] Schaefer, M., Gold, B., Linde, R., & Scheid, J. (1977). Program confinement in KVM/370. *Proceedings of the 1977 Annual ACM Conference* (pp. 404-410). ACM Press.

[33] Shaffer, A., Auguston, M., Irvine, C. and Levin, T. (2007). Toward a security domain model for static analysis and verification of information systems. *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling* (pp. 160-171). Montreal, Canada.

[34] Simonet, V. (2003). Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'03)*, vol 2895 (pp. 283-302). Beijing, China: Springer-Verlag.

[35] Smith, G., & Alpizar, R. (2006). Secure information flow with random assignment and encryption. *Proceedings of the 4th ACM Workshop on Formal Methods in Security* (pp. 33-44). ACM Press.

[36] Smith, G. (2006). Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security* 14(6), 591-623.

[37] Smith, S., & Thober, M. (2007). Improving usability of information flow security in java. *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security* (pp. 11-20). ACM Press, New York, NY.

[38] Tsai, C., Gligor, V., & Chandersekaran, C. (1990). On the identification of covert storage channels in secure systems. *IEEE Transactions on Software Engineering*, 16(6), 569-580. IEEE Press.

[39] Volpano, D., & Smith, G. (1999). Probabilistic noninterference in a concurrent language. *Journal of Computer Security* 7(2,3), 231–253.

[40] Volpano, D., Smith, G., & Irvine, C. (1996). A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 167-187.

[41] von Oheimb, D. (2004). Information flow control revisited: Noninfluence = noninterference + nonleakage. *Proceedings of the 9th European Symposium on Research Computer Security* (pp. 225-243). Sophia Antipolis, France.