A Security Domain Model for Implementing Trusted Subject Behaviors

Alan Shaffer, Mikhail Auguston, Cynthia Irvine, Timothy Levin

Naval Postgraduate School, Computer Science Department, 1411 Cunningham Rd.,
Monterey, California, USA 93943
{abshaffe, maugusto, irvine, levin}@nps.edu

Abstract. Within a multilevel secure (MLS) system, trusted subjects are granted privileges to perform operations that are not possible by ordinary subjects controlled by mandatory access control (MAC) policy enforcement mechanisms. These subjects are trusted not to conduct malicious activity or degrade system security. We present a formal definition for trusted subject behaviors, which depends upon a representation of information flow and control dependencies generated during a program execution. We describe a security Domain Model (DM) designed in the Alloy specification language for conducting static analysis of programs to identify illicit information flows, access control flaws and covert channel vulnerabilities. The DM is compiled from a representation of a target program, written in an intermediate 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 detect potential security policy violations in the target program. In particular, since the operating system upon which the trusted subject runs has limited ability to control its actions, static analysis of trusted subject operations can contribute to the security of the system.

Keywords: Security domain model, trusted subjects, static analysis, automated program verification, specification language.

1 Introduction

Within a multilevel secure (MLS) system, *trusted subjects* may be granted privileges to perform operations, in some cases within prescribed limits [22], not normally allowed for ordinary subjects controlled by mandatory access control (MAC) policy enforcement mechanisms. Granting of such privileges is predicated on the idea that trusted subjects will not conduct malicious activity or degrade the system's overall security. This paper presents a formal definition for trusted subject behaviors in certain program implementations. These behaviors depend upon a representation of information flow and control dependencies generated during a target program execution, thus extending classic work in this area [8][19][33]. We describe a security domain model to formally represent trusted subject behaviors, information flow tracing through program execution, various types of covert channels, and a means for conducting static analysis of certain program implementations.

Widely accepted evaluation standards [5][6][18] 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. Formal security models are often based on concepts of program secure state and state transitions. Our approach analyzes programs for preservation of security properties through state transitions, and advances the concepts of secure information flow in classic work by Denning and others [8][33] by describing automated techniques for information flow static analysis. Our previous work has demonstrated the ability to detect illicit information flow security violations [24], and covert channel and overt access control flaws based on control dependency analysis [25].

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. The *target program* is an original high-level language program from which we extract a *base program*, the IML abstraction that provides a basis for analysis of the target program 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][9] specification. The DM provides a framework for specifying program state and state transitions, as well as security-related concepts such as security policy, information flow, access control, and covert channel vulnerabilities. The 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.

Our approach uses the Alloy Analyzer tool [1] to perform static analysis of base programs to identify execution paths that might violate the security policy rules. The Alloy Analyzer performs symbolic execution of all base program execution paths within a defined *scope* (the upper limit of the size of models considered); the scope is generated heuristically, based on the total number of statements in the base program. It is assumed that the Alloy *small scope hypothesis*, which states that most flaws in models can be revealed on small instances [9], holds for information flow tracing. A description of the DM structure, and examples of refinement of a base program to Alloy can be found in [23][25].

Both model checking [4] and Alloy analysis are examples of heuristics for static analysis. Model checkers build models using finite numbers of states and transitions to represent system behavior, and temporal logic to represent assertions about those models. Model checkers are limited by the way they construct models, such that their subsequent modification is rendered extremely difficult. Alloy, which uses the full power of sets and first-order predicate calculus, does not have such limitations. The analysis of a complex model in Alloy may be adjusted by simply modifying the *scope* of analysis (size of analyzed instances), a much easier process than redesigning the entire model. For our approach, Alloy and its Analyzer provide a well-suited tool for creating and analyzing target program abstractions.

Section 2 of this paper provides background discussion on trusted subjects and processes. Section 3 presents an overview of the Security DM methodology for modeling programs and security policies. Sections 4 and 5 describe analyses of several example base programs using the DM, and summarize our test results with these examples. Sections 6 and 7 discuss related work, and planned future work in this research.

2 Trusted Subjects

Users in a multilevel secure (MLS) environment are assigned a clearance level based on the relative level of trust placed in them by security administrators. A user is allowed to log into a system at any level that is at or below (*dominated by*) his assigned clearance, and a *session* at that level is created. Subjects that act on behalf of a given user are labeled with an access class that is at the same level as the user's session. A subject is allowed to read information (*objects*) whose sensitivity level is up to the subject's sensitivity level (*access class*), and write to objects at or above its access class.

In contrast to this, a *trusted subject* is one that is allowed to read and write within a *range* of access classes [16]. Depending upon the implementation, this can limit the authority of the trusted subject to "read-up" and "write-down" [3]. MLS systems with trusted subjects defined this way do not require a separate access control lattice or special rules specifically for their actions [16]. As a result, a trusted subject does not need to be given a privilege to bypass or violate the security rules.

Since trusted subjects are allowed to interact with (read and write) information across access classes, they must be trusted not to abuse these special privileges. The existence of trusted subjects is generally required for certain services provided in MLS systems, such as login, information downgrading, and data backup utilities across multiple access levels. MLS system administration may also require a trusted subject to interact with and manage regular user accounts and information across multiple access levels [31]. Such actions represent a good target for trusted subject implementation, however the design principle that trusted subjects should be small and minimized within an MLS system [12] is not always observed.

With respect to security policies, a trusted subject should not move data between sensitivity levels, other than in constrained, explicitly defined ways [30]. The specification of a trusted subject must explicitly define how it can do this. Levin et al. [15] point out that trusted subjects do not violate a general policy in place, but their behavior must be a defined part of a policy. Such a policy for trusted subjects, referred to as a "relaxed MLS policy," must be integrated with the general MLS policy, such that the resultant union of the two can allow trusted subjects to operate effectively, while ensuring that non-trusted subjects cannot cause unwanted information flows. In a "downgrader" role, for example, a trusted subject may essentially change the label of information from *high* to *low* by reading the information from a *high* object and moving it into another *low* object.

Trusted subjects can be defined by their behaviors in an MLS system. Although some [30] would have trusted subjects relabel objects, we maintain the tranquility of

object labels [3], and abstract the idea of downgrading information by changing variable labels from the viewpoint of, i.e., internal to, the trusted subject. Allowing movement of information within a range of access classes represents the trusted subject actions we model in our DM approach.

3 Security Domain Model Methodology

Our approach to program security verification using the Security Domain Model has been described in [24][25], and an overview is provided below.

A *base program* represents an abstraction of a target program implementation, and is written in Implementation Modeling Language (IML) notation. The IML defines a simple imperative language that captures the basic capabilities and constructs, with respect to security, of high-level programming languages, such as Java or C++. The IML was motivated by a requirement to represent the information flow properties in target program implementations. A complete IML and DM reference manual is available online at [23].

The IML enables trusted subject behavior by providing a special *trusted assignment* statement. This statement allows trusted subjects to modify the labels of internal variables ("regrading"), while respecting the tranquility of external object labels. The trusted assignment allows filtering of variable values based on existing content or label. This filtering is analogous to a "dirty word search" of a document prior to downgrading its classification level, which ensures that certain sensitive words are first filtered from the document.

The trusted assignment statement allows a trusted subject to assign a value to a *destination* variable, with an explicitly defined security label. When an IML base program is translated, only a trusted subject may perform trusted assignment. The trusted assignment syntax follows:

`Assign` *destination* `from` *source1* `as` *source2*`;`

In this operation, the *destination* variable takes on the data value of the *source1* variable, however it does not automatically take its label, as would normally be the case for an assignment statement in IML. Instead, the *destination* is explicitly assigned a label, as determined by a filter function that is automatically invoked with each trusted assignment. The trusted assignment *source1* can be either a variable or constant, and the *source2* can be either a variable (in which case the access label currently assigned to the value stored in this variable is used) or an explicitly defined access label. The new content and access label of the *destination* variable are defined in the DM Invariant Model by an Alloy function TS_filter (further discussed in Section 4.1), as follows:

(*destination_value'*, *destination_label'*) =
  TS_filter((*destination_value*, *destination_label*),
    (*source1_value*, *source1_label*), (*source2_value*, *source2_label*))

This function specifies the behavior of trusted subjects in our model, and examples are described in detail in section 4.

The DM *Invariant Model* defines security rules that have the Bell and LaPadula security model [3] as their basis, i.e., flows from higher to lower secrecy levels are not allowed by either writing down or reading up. The general DM security policy defines a lattice with flows allowed from lower to higher (or equal) secrecy levels, represented by access labels, for instance:

```
one sig Policy {
  ord: AccessLabel -> AccessLabel  }
{  ord = ^( (SysLow -> SysMid)
          + (SysMid -> SysHigh) )
          + (iden & (AccessLabel -> AccessLabel) )  }
```

In Alloy notation this defines a recursive closure of the access label relations (*SysLow -> SysMid*) and (*SysMid -> SysHigh*). The "basic" security policy is defined in the DM Invariant Model by reads and writes of external I/O devices that conform to this policy lattice. The trusted policy is defined such that trusted subjects are allowed to change labels and data within the constraints of the TS_filter.

4 Testing and Analysis of Trusted Subject Behaviors in the DM

This section presents examples of program security vulnerabilities that illustrate how trusted subjects are constrained by both the basic security policy and the trusted policy (as implemented in the TS_filter). In these examples, the security rules for discovering information flow errors, overt access control flaws and covert channels, are described using Alloy notation, and a base program written in IML is presented to illustrate the particular security violation. The complete Alloy models for these and other examples can be found online at [23].

4.1 Information Flow Violation Caused by a Trusted Subject Operation

The first example illustrates a trusted subject regrade operation that, based on allowed trusted subject behavior, leads to an information flow violation. In the example, an attempt is made by a trusted subject to downgrade a destination variable label from *SysHigh* to *SysLow*. Here, trusted subjects are allowed to perform downgrading of information from *SysHigh* to *SysMid*. To support the policy, a TS_filter function is defined (below in Alloy notation) to ensure that "downward" info flows are allowed only from *SysHigh* to *SysMid*. The function takes as input parameters three `Values` and three `AccessLabels`, specifically, the data values and labels of the *destination*, *source1* and *source2* variables in the Trusted Assignment (see Section 3 for trusted assignment IML syntax), and returns an instance of `FTuple` (i.e., a filtered `Value` and `AccessLabel`). In essence, the policy for trusted subject behaviors is captured in the semantics of this filter function.

For example purposes here, this `TS_filter` function returns the greater of constant 0 and the *source1* `Value` (`s1v`), and the higher of *SysMid* and the *source2* `AccessLabel` (`s2a`). As shown in this example `TS_filter`, it is not necessary to use all of the parameters passed into the function to generate a resulting `FTuple`.

Note that a different DM Invariant Model could define a `TS_filter` function that would return different results based on the specific input parameters, and thus define a different security policy for trusted subject behaviors.

```
sig FTuple {
  val:   Value,
  label: AccessLabel
}

fun TS_filter[dv, s1v, s2v: Value,
              da, s1a, s2a: AccessLabel]: FTuple {
{ result: FTuple | {
    result.val = (((s1v->const0) in LT.lt)
        => const0 else s1v)
    result.label = (((da->s2a) in Policy.ord)
        => s2a else
      (((s2a->SysMid) in Policy.ord)
        => SysMid else s2a)) }

} }
```

The base program example below demonstrates a security violation based on the trusted subject filter and security policy. Initially, values are read into two variables with security labels *SysHigh* and *SysMid*, respectively (s1-s2). A trusted assignment operation is then performed (s3), in which the data value stored in `x2` is copied into variable `x1`, and `x1` is assigned a *SysLow* label. During this statement operation, the TS_filter function is applied to the parameters of the trusted assignment, "filtering" the label assignment to *SysMid*, which results in `x1` being assigned a higher label than was intended by the trusted assignment operation (s3).

```
(s1) Read_dev (SysHigh, x1);
(s2) Read_dev (SysMid, x2);
(s3) Assign x1 from x2 as SysLow;
(s4) Write_dev (SysLow, x1);
(s5) Stop;
```

When the next statement (s4) attempts to write the value held in `x1` to a *SysLow* external device, an illicit flow results since `x1` is labeled as *SysMid*. The Alloy Analyzer detects this situation as a violation of the Alloy security predicate below, and correctly reports an illicit information flow, tracing execution through statements (s1)(s2)(s3)(s4). The same base program, under a DM Invariant Model with a different policy and filter function, would not necessarily result in this flow violation.

```
pred consistent_with_FlowPolicy [current: State] {
  let stm = current.stmt | {
    ( stm.type in (Write_dev + PutDirectFile) &&
      stm.source in Variable )
    => (current.access_label[stm.source] ->
        stm.subject_label) in Policy.ord
} }
```

4.2 Trusted Subject Flow Violation & Control Dependency Flaw

The second example base program illustrates two different security violations that may result from a trusted subject operation. In the program, a successful trusted subject regrade creates an overt control dependency flaw, however when the trusted subject regrade fails to occur, illegal information flow results. For purposes of this example, the security policy and TS_filter function described above apply.

In the base program, values are initially read into three variables, with assigned security labels *SysHigh*, *SysMid* and *SysLow*, respectively (s1-s3). Depending on the value stored in `x1` (s4), a trusted assignment statement is performed (s5) in which the value of `x1` is modified to that of `x2`, and the label of `x1` is downgraded to that of `x3`, *SysMid* in this case. Since a regrade from *SysHigh* to *SysMid* is allowed by the security policy (as reflected in the TS_filter function), `x1` is assigned the *SysMid* label.

```
(s1) Read_dev (SysHigh, x1);
(s2) Read_dev (SysLow, x2);
(s3) Read_dev (SysMid, x3);
(s4) if x1 < 0 then {
(s5)   Assign x1 from x2 as x3;
(s6)   Write_dev (SysMid, x1); }
     else
(s7)   Write_dev (SysMid, x1);
(s8) Stop;
```

The next statement (s6) attempts to write the value of `x1` to a *SysMid* external device, a seemingly legal flow. However, since this operation occurs within the if-block, it creates a control dependency from *SysHigh* (`x1` label when it was examined in s4) to *SysMid*, representing an overt access control flaw (i.e., in the *SysHigh* context, a write to *SysMid* violates the security policy). Based on the Alloy security rule predicate below, the Alloy Analyzer properly detects this violation, tracing execution through statements (s1)(s2)(s3)(s4)(s5)(s6).

```
pred dependency_flaw_found [current: State] {
  let stm = current.stmt,
      pre = current.influenced_by[stm.source] |
  {
    stm.type = Write_dev  &&
    stm.source in Variable &&
    not ((pre.access_label[pre.stmt.source] ->
      stm.subject_label) in Policy.ord)
} }
```

An additional violation occurs when the conditional check (s4) evaluates to false, and the else-branch is executed. In this case, an attempt is made to write the value stored in `x1` (still assigned its original *SysHigh* label) to a *SysMid* external device (s7). Since this represents an overt illegal flow from *SysHigh* to *SysMid*, the Alloy Analyzer properly identifies and reports the error, tracing execution through statements (s1)(s2)(s3)(s4)(s7).

4.3 Covert Channel Resulting from a Trusted Subject Operation

The third scenario describes execution of a trusted assignment that could produce a covert storage channel [14]. Our earlier paper [25] describes in detail how the DM formalizes the notion of covert channels, and defines a security rule to identify a class of covert storage channel vulnerabilities in a base program execution.

In the base program below, we assume a direct file with a maximum capacity of two records, initially empty. To begin, *SysLow* values are read into variables `x1` and `x2` (s1-s2). A trusted assignment is performed (s3) in which `x1` is assigned the value of `x2`, and upgraded to a *SysHigh* label. Next, the value of `x1` is examined to verify whether it is non-negative (s4). Since the TS_filter function returns only values of 0 or greater, `x1` holding a non-negative value is an indication that the trusted assignment resulted in the assignment of source data to the destination variable. When this evaluates to true, the values of `x1` and `x2` are stored into the direct file by the *SysHigh* sender, resulting in the internal *full* direct file flag being set.

```
(s1)  Read_dev (SysLow, x1);
(s2)  Read_dev (SysLow, x2);
(s3)  Assign x1 from x2 as SysHigh;
(s4)  if x1 > const_minus_1  then {
(s5)    PutDirectFile (SysHigh, 1, x1);
(s6)    PutDirectFile (SysHigh, 2, x2); }
```

The next sequence of program statements represent execution by a *SysLow* covert channel receiver. When the *SysLow* subject attempts to store a value into the direct file using a new key 3 (s7), the system issues a failure indication since the direct file is *full* (note that in the translation to an base program, the internal system flag translates to an explicit flag, accessible in IML as in statement (s8)). Depending on the success or failure of the direct file store (s8), *SysLow* writes a constant '1' or a '0' to an external device (s9 & s10) to exploit the storage channel.

```
(s7)  PutDirectFile (SysLow, 3, 1);
(s8)  if full = True then
(s9)    Write_dev (SysLow, 1);
(s10) else Write_dev (SysLow, 0);
(s11) Stop;
```

Because a higher-labeled subject caused the direct file to become full, the Alloy Analyzer detects and reports this violation of the below Alloy security predicate, tracing the flow of execution through statements (s1)(s2)(s3)(s4)(s5)(s6)(s7). The actions of two regular subjects at *SysHigh* and *SysLow*, acting in collusion to exploit the direct file, could bring about the same security violation (i.e., a storage channel).

```
pred storage_channel_found [current: State] {
  let stm = current.stmt | {
    stm.type = PutDirectFile &&
    current.direct_file.full = const1 &&
    not (current.direct_file.last_written ->
         stm.subject_label) in Policy.ord
} }
```

5 Testing Results

The base program examples presented above were evaluated using Alloy Analyzer 4.1.7, running under Mac OS X™ 10.5.4 on a 2.5 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 described above.

Test results are summarized in Table 1 below. The Analysis Size defines the size of Alloy model instances considered *(scope)* during Alloy Analysis; Analysis Time represents *total time (ms),* broken down into *(time to generate model, time to find a counterexample)*:

Security Violation Description	**Analysis Size (*scope*)**	**Analysis Time (*ms*)**
Information flow violation, resulting from trusted subject operation	7	1516 (1277, 239)
Overt control dependency flaw, resulting from trusted subject operation	9	3335 (2290, 1045)
Information flow violation, resulting from trusted subject operation	9	2692 (2236, 456)
Storage covert channel, resulting from trusted subject operation	12	48631 (9852, 38779)

Table 1. Results of Alloy Analysis Testing

6 Related Work

Previous work in trusted subject implementation [35] developed a framework for running a trusted multi-level database management system (DBMS), referred to as a "trusted subject," to be hosted on any trusted operating system. This work established a layered policy, with a general policy for the *trusted computing base* (TCB) layer of the operating system, and a separate policy for the DBMS TCB layer. Their premise was that, for a DBMS hosted on a known secure operating system, only the DBMS TCB layer must be subjected to security analysis to ensure that it meets all access control requirements. This work did not appear to outline a concrete policy for trusted subjects, and allowed modification of object labels as a valid action for trusted subjects, whereas our model preserves tranquility of object labels.

Previous 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 secrecy*) information to ciphertext (*low secrecy*) information must be addressed in light of noninterference rules that would seem to prevent such interaction [11][28]; *probabilistic noninterference*, where probability distributions are used to determine a likelihood of interference from *high* to *low* variables, primarily for multi-threaded processes where scheduling is nondeterministic [32][20]; and *type inference*, in which the flow of information is automatically determined based on semantic analysis [26][7]. Eventually, Smith & Thober [29] enhanced the linguistic type

system model of secure information flow such that sensitivity labels need to 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.

In contrast, 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 (the *base program*) to the analysis of *target programs* in order to detect potential problems, and otherwise demonstrate security of the abstraction with respect to select security properties. Our approach performs exhaustive information flow tracing of all execution paths in a program, to a predetermined length (defined by the Alloy model *scope*). 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 [34]. 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.

Landauer et al. [10] introduced a formal model for managing trusted processes, by defining a state machine whose state space can be locked, or isolated, in order to allow privileged actions to overlap. The authors described a trusted process as possessing special privileges to alter operating system kernel access control decisions, or other security critical operations. This paper provided an in-depth mathematical analysis of the security policy derived from trusted process principles, and is a useful source regarding security policy issues for trusted subjects.

Steffan and Clow [30] defined a set of trusted process classes, to identify their relative privileged status. These classes correspond to combinations of override privileges in the areas of Tranquility (labels), MAC (content) and DAC (privileges). As the class numbers increase, so do the privileges granted, and the risk associated with using a trusted process in that class. In contrast to this paper, our work characterizes trusted subjects without violating tranquility of object labels.

Levin et al, [13] discussed trusted subject actions within a security kernel architecture. With respect to the principle of least privilege [21], they described how a trusted subject in a downgrader role must be constrained to perform only the minimum required operations, namely, downgrading of labels in this case. Other operations such as "dirty word search" (DWS) of a document for specific words or phrases prior to downgrade, must be handled by other trustworthy system processes to prevent unintended or malicious consequences. They defined a framework for performing filtering and downgrade of information, separating tasks between users and processes, both untrusted and trusted. We believe our model is in line with this thinking, when one considers that if our trusted subject acts as a downgrader, the Invariant Model filter function can reflect a separate untrusted process in the target program that performs DWS. We generalize this concept by allowing the trusted subject to downgrade based on content or label information. In our model, the DWS might represent examination of a highly classified document for specific references to some classified topic, with subsequent removal of these references prior to downgrading the document. Alternately, the DWS could represent filtering of a

document by its creation date, where downgrading of the document will occur only if this information is older than some predetermined date.

7 Discussion and Future Work

This paper has provided a survey of ongoing research to develop a formal security domain model that formalizes security policies for both regular and trusted subjects. The model formalizes trusted subject behaviors, using the specialized imperative language. Our approach defines a formal security Domain Model (DM) that facilitates specification of security vulnerabilities and trusted subject behaviors, independent of program implementation.

By Alloy's *small scope hypothesis* [9] it is assumed that most program errors may be revealed by relatively small counterexamples. Using the Analyzer to perform static analysis of the DM provides assurance that, within a specified search scope, a counterexample will be found when one exists, and that false negatives and false positives are eliminated within the defined scope. This assumption necessitates our implementation of a relatively small trusted subject, which is aligned with the Reference Monitor Concept principle that a reference validation mechanism "must be small enough to be subject to analysis and tests" [2] to ensure its correctness.

Future work will expand the DM to enable dynamic security policies [14]. This concept would allow the DM to support a sequence of polices during program execution, and support the ability of a system to adapt to a dynamically changing security environment by using different policies [17]. We could extend this by adding functionality for multiple trusted subjects. By defining multiple filter functions within a DM Invariant Model, and modifying the IML syntax to support this, the model could represent separate trusted subjects, each governed by a different policy as defined by its own filter function.

Additionally, extension of this research will focus on tailoring our approach toward the model-driven software design process. We understand that automation of the software development cycle, such that resulting software systems fully conform to the Common Criteria Development requirements, is not a trivial effort. We have focused specifically on the Implementation Representation and Security Objectives stages of development [5], devising an automated way to verify that the former abides by the latter. A framework to automate the actual *production* of the implementation representation, based on functional requirements and security objectives, is an ideal goal.

8 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, http://alloy.mit.edu/.
2. Anderson, J. (1972). *Computer Security Technology Planning Study*. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA.
3. Bell, D., & LaPadula, L. (1973). Secure Computer Systems: Mathematical Foundations and Model, *MITRE Report*. The MITRE Corp.
4. Clarke, E., Emerson, E., & Sistla, A. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 244-263.
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*, 543-548.
8. Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 504-512. ACM Press.
9. Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA, and London, England: MIT Press.
10. Landauer, J., Redmond, T., & Benzel, T. (1989). Formal policies for trusted processes. *Proceedings of the Computer Security Foundations Workshop II*, 31-40.
11. Laud, P. (2003). Handling encryption in analyses for secure information flow. *Proceedings 12th European Symposium on Programming, (ESOP)*, 159-173.
12. Levin, T., Irvine, C., Benzel, T., Bhaskara, G., Clark, P., & Nguyen, T. (2007). *Design principles and guidelines for security*. Technical Report NPS-CS-07-014, Naval Postgraduate School, Monterey, California.
13. Levin, T., Irvine, C., and Nguyen, T. (2006). Least Privilege in Separation Kernels. *Proceedings of the 2006 International Conference on Security and Cryptography*, 355-362.
14. Levin, T., Irvine, C., & Spyropoulou, E. (2006). *Quality of security service: Adaptive security*. Handbook of Information Security (H. Bidgoli, ed.), vol. 3, 1016–1025. Hoboken, NJ: John Wiley and Sons.
15. Levin, T., Irvine, C., Weissman, C., & Nguyen, T. (2007). Analysis of three multilevel security architectures. *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, 37-46. ACM Press, New York, NY.
16. Lunt, T., Denning, D., Schell, R., Heckman, M., & Shockley, W. (1990). The seaview security model. *IEEE Transactions on Software Engineering*, 16(6), 593-607.
17. National Security Agency IA Directorate. (2004). *Global Information Grid Information Assurance Reference Capability/Technology Roadmap*, Version 1.0.
18. National Security Agency. (2007). *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Version 1.03.
19. Sabelfeld, A., & Myers. A. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 5-19. IEEE Press.
20. Sabelfeld, A., & Sands, D. (2000). Probabilistic noninterference for multi-threaded programs. *Proceedings of the IEEE Computer Security Foundations Workshop*, 200-214.
21. Saltzer, J. & Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278-1308.
22. Schell, R., Tao, T., & Heckman, M. (1985). Designing the GEMSOS Security Kernel for Security and Performance. *Proceedings of the 8th National Computer Security Conference*, 108 - 119.

23. Security Domain Model Project website, http://cisr.nps.edu/projects/sdm.html.
24. 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*, 160-171. Montreal, Canada.
25. Shaffer, A., Auguston, M., Irvine, C. and Levin, T. (2008). A Security Domain Model to Assess Software for Exploitable Covert Channels. *Proceedings of the ACM SIGPLAN Third Workshop on Programming Languages and Analysis for Security (PLAS'08)*, 45-56. Tucson, Arizona. ACM Press.
26. 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, 283-302. Beijing, China: Springer-Verlag.
27. Smith, G. (2006). Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security* 14(6), 591-623.
28. Smith, G., & Alpizar, R. (2006). Secure information flow with random assignment and encryption. *Proceedings of the 4th ACM Workshop on Formal Methods in Security*, 33-44. ACM Press.
29. 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.
30. Steffan, W., & Clow, J. (1996). Trusted process classes. *Proceedings of the 19th National Information Systems Security Conference*.
31. Thomas, R., & Sandhu, R. (1996). A trusted subject architecture for multilevel secure object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 16-31.
32. Volpano, D., & Smith, G. (1999). Probabilistic noninterference in a concurrent language. *Journal of Computer Security* 7(2,3), 231–253.
33. Volpano, D., Smith, G., & Irvine, C. (1996). A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 167-187.
34. von Oheimb, D. (2004). Information flow control revisited: Noninfluence = noninterference + nonleakage. *Proceedings of the 9th European Symposium on Research Computer Security,* 225-243. Sophia Antipolis, France.
35. Wilson, J. (1989). A security policy for an A1 DBMS (a trusted subject). *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 116-125.