A Security Domain Model for Static Analysis and Verification of Software Programs

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

Abstract- *Unauthorized information flows can result from malicious software exploiting covert channels and overt flaws in access control design. To address this problem, we present a precise, formal definition for information flow that relies on control flow dependency tracing through program execution, and extends Dennings' and follow-on classic work in secure information flow [7][19][27]. We describe a formal security Domain Model (DM) for conducting static analysis of programs to identify illicit information flows, access control flaws 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 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.*

I. INTRODUCTION

Identification of exploitable covert channel vulnerabilities is vital in the development of systems intended to enforce mandatory access control policies, and in fact is required for the successful evaluation of such systems at the highest levels of assurance [3][18]. 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 [7][19][27]. 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 [3][4][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. 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 [3].

Formal security models are often based on concepts of program secure state and state transitions. High assurance evaluation standards [3][4] require a formal verification that the state transitions resulting from program execution preserve the security properties defined by a policy. 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 [7][27], by describing automated techniques for information flow static analysis. Previous work in developing our approach has demonstrated the ability to detect illicit information flow security violations [22], and covert channel and overt flaw vulnerabilities based on control flow dependency analysis [23].

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][11] 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. The DM is verified using the Alloy Analyzer, which

[1] In this context, *static analysis* refers to analysis of program code without actual program execution.

[2] EAL-7 is the highest Common Criteria evaluation assurance level.

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 legal information flows based on the relationship between the subject label, and that of the I/O source/destination variable, 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 provides background discussion on covert channels, control flow dependencies, and dynamic program slicing. Section 3 presents an overview of the DM methodology for modeling programs and security policies. Section 4 summarizes our test results with several program examples. Sections 5 and 6 discuss related work, and planned future work in this research.

II. BACKGROUND

We discuss several computer security concepts relevant to this research.

A. 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 [12][14]. Such channels allow processes to transfer information in a manner that violates a security policy [8].

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 [20]. For both storage and timing channels the sender and receiver (typically subjects) must have [12]:

1. Indirect access to an attribute of a shared resource, which the sender can modify, and the receiver can view.
2. A means to initiate and synchronize their actions.

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.
4. Timing – the receiver views changes to the relative timing of "legal" events.

The attribute in question forms a *point of interference* [9] 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) [26].

B. 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* [12][14][15]. 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 [19].

C. Dynamic Slicing

Slicing algorithms are used as a means of tracing data or control dependencies between variables and statements processed during program execution, traditionally for program debugging purposes [13]. 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 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).

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. The access labels of variables can be used to determine potential security violations, based on the dependencies between these variables. 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;
```

It is 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.

III. SECURITY DOMAIN MODEL METHODOLOGY

An overview of the Domain Model (DM) approach to program security verification is depicted in Fig. 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 abide by. The DM is composed of an invariant and a variable section, derived from the security rules and a target implementation, respectively.

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 completely. As Jackson [11] 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 [11].

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 a unique, 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 using Implementation Modeling Language (IML) notation [23]. The IML defines a simple domain-specific language that captures the basic capabilities and constructs, with respect to

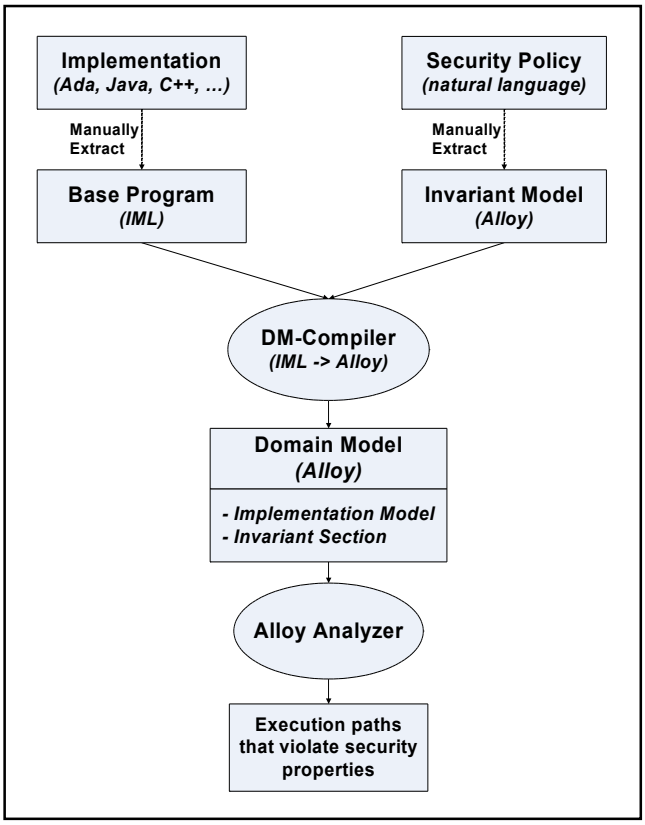

Fig. 1. Domain model approach to system security verification.

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 the most essential security information flow properties in target program implementations. This was our goal in describing IML syntax and constructs.

By analyzing a model of the program, rather than actual program code, security verification can focus on elements of information flow analysis, e.g., I/O, access labels, direct file access, and timing (system 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 overt and covert flow violations 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.

The Invariant Model includes the definition of security rules, written as Alloy assertions, which must be enforced by the DM 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 associated with 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. A detailed description of the DM structure can be found at [23].

When analyzing a base program, the Alloy Analyzer performs an exhaustive search of all paths 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 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 [11], holds for information flow tracing in our approach.

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. From the base program, the DM-Compiler generates Value and Variable signatures, representing the number and value of unique constants explicitly present in the base program, and the variables used in the base program, respectively. The DM-Compiler defines an Alloy signature that establishes a less-than relationship between the constant values, enabling comparison of values for equality and inequality in the base program.

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. From these statement signatures, it generates a predicate representing the state transition trace for the base program execution. This 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. A detailed example of this refinement from base program to Alloy signatures and transition predicate is provided at [23].

IV. TESTING AND ANALYSIS OF THE DM

We tested the DM approach using base program examples with illicit information flows, and overt flaw and covert channel vulnerabilities. In each case, a rule for discovering the illicit flow or covert channel is defined as an Alloy assertion, and an example base program is presented to illustrate the error or violation. Each example represents the transmission of one bit of information; more complex examples would involve such concepts as looping, synchronization, etc., to provide the covert channels with a stream of bits.

Our base program examples were evaluated using Alloy Analyzer 4.0. 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. The complete Alloy models for these examples can be found at [21].

The "*IllicitFlow*" example [21] demonstrates an illicit information flow based on violation of the BLP simple security policy, i.e., a flow from a *High* object to a *Low* device. The Alloy assertion below defines a security rule for such a policy that examines each execution state, and evaluates to true whenever the state (`s`) is the result of a Write_dev operation to a *Low* device, from a variable whose access label is *Low*. The DM searches for execution paths for which this assertion is not true, i.e., those with a flow that violates the security rule.

```
assert correct_access1{
   all s: State | Property1[s] }

pred Property1 [s: State]{
   let stm = s.stmt | {
      (stm.type = Write_dev and
         stm.subject_label = Low and
         stm.source in Variable)
      => s.access[stm.source] = Low }
}
```

The base program below is an example of a violation of this security assertion. The program first reads a value into variable `x1` at a *High* access level, and then checks the variable's value against a constant. Based on the result of this conditional check, the value in `x1` is either written to a *High* or a *Low* external device.

```
(s1) Read_dev (High, x1);
(s2) if (x1 > 3) then
(s3)   Write_dev (High, x1);
(s4) else Write_dev (Low, x1);
(s5) Stop;
```

The violation occurs when the conditional (s2) evaluates to false, thus the value of `x1` is written to the *Low* device (s4), creating a flow from *High* to *Low*. The Alloy Analyzer detects this situation, and reports a violation of the security assertion through statements (s1)(s2)(s4).

Further examples include "*OvertFlaw*" [21], which 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 approach uses dynamic slicing techniques to discover these flow violations.

The "*StorageChannel*" example [21] describes a classic covert storage channel [16] resulting from access to the direct

file by a *Low* subject (who uses a PutDirectFile operation), after a *High* subject has caused it to be full. The Alloy security assertion defines logic to capture this vulnerability by checking for states where the label of the direct file key slot (keyLabel) is higher than that of the subject (subject_label). The nexus of this covert channel is that *High* can write to the internal resource full (indirectly), and *Low* can observe it.

Our "*TimingChannel*" example [21] describes a covert timing channel that occurs when a *Low* subject twice checks the system clock, between which a *High* subject prevents the *Low* subject from executing through execution of a Read_dev/Write_dev or direct file operation. 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 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.

V. 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 [6][7] 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 have viewed no difference 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 [9] described that security policies can be defined in terms of only noninterference assertions, rather than by the combination of access control and covert channel restrictions. Their ideas were further expanded in [10].

Volpano et al [27] 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 [19].

Other work in using sound type systems for secure information flow has focused on type inference, in which the flow of information is automatically determined based on semantic analysis [5][24]. Eventually, Smith & Thober [25] 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 [28]. 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.

VI. Discussion and Future Work

This paper has provided a survey of ongoing research to develop a formal security domain model for analyzing programs for information flow vulnerabilities, including exploitable covert channels and overt access control flaws. The approach defines a formal security Domain Model (DM) that facilitates specification of security vulnerabilities, independent of program implementation.

Although encoding and checking program semantics and properties is not in itself revolutionary, we feel that 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, within a finite scope.

The Alloy Analyzer guarantees, by the small scope hypothesis [11], that most program errors should be revealed in 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. This means that false negatives and false positives are eliminated within the defined scope.

Future work will focus on formally proving the DM, and on extending its capabilities. In the former case, formal semantic analysis of the IML and DM-Compiler is needed 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 [19], 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 the code itself. In addition to semantic analysis of these DM components, the results of the Alloy Analyzer acting on a compiled DM must be formally proven to be both sound and complete, i.e., that they produce neither false positives nor false negatives, respectively.

Work has begun to implement the notion of a trusted subject into the DM. This class of subject is trusted to circumvent certain access control policy rules, to allow such actions as regrading of objects, e.g., downgrading a *High* labeled object to a *Low* level. This requires defining a separate trusted subject policy within the DM, and the ability for the model to administer multiple policies, i.e., for regular and trusted subjects.

Other planned work includes expansion of the DM to enable support for dynamic security policies [16]. 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 [17].

REFERENCES

[1] *The Alloy Analyzer*. (2000). Retrieved March 3, 2008, from the Alloy Analyzer website: http://alloy.mit.edu/.

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[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] 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.

[21] *Security Domain Model Project*. (2008). Retrieved March 5, 2008, from Naval Postgraduate School (NPS) Center for Information Systems Security Studies and Research (CISR) Projects website: http://cisr.nps.edu/projects/sdm.html.

[22] 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.

[23] Shaffer, A., Auguston, M., Irvine, C., and Levin, T. (2008). *A security domain model to assess software for exploitable covert channels*. Manuscript submitted for publication.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.