Toward a Security Domain Model for Static Analysis and Verification of Information Systems

Alan Shaffer, Mikhail Auguston, Cynthia Irvine, Tim Levin

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
{abshaffe, maugusto, irvine, televin}@nps.edu

Abstract. Evaluation of high assurance secure computer systems requires that they be designed, developed, verified and tested using rigorous processes and formal methods. The evaluation process must include correspondence between security policy objectives, security specifications, and program implementation. This research presents an approach to the verification of programs represented in a specialized Implementation Modeling Language (IML) using a formal security Domain Model (DM). The DM is comprised of an invariant part, which defines the generic concepts of program state, information flow, and other security properties; and a variable part, specifying the behavior of the target program. The DM is written using the Alloy formal specification language, and its verification is accomplished using the Alloy Analyzer tool. It was found that, by separating the structural framework of the security policy from the semantics of the target program, efficiency of the Alloy Analyzer in detecting execution paths that violate the security properties specified in the DM is significantly improved.

***Keywords**:* Security domain model, static analysis, automated program verification, specification languages.

1. INTRODUCTION

Domain modeling techniques allow system developers to define the key concepts and properties for some area of interest, often for specific business fields such as medicine or engineering [13]. In the area of high assurance computer systems, these concepts can be applied to verify that a program meets some security policy, and more specifically, security properties that reflect that policy. This research is currently engaged in creating a framework for formally representing a security policy through software properties. To verify that the software implementation abides by the policy, the framework supports static analysis of the implementation. In this context, static analysis refers to analysis of program code without actual program execution [5]. Static analysis tools such as the Alloy Analyzer provide the ability to examine program execution paths for potential security property violations.

While it has many definitions, the objective of computer security is generally recognized to be the confidentiality, integrity, and availability of information, as specified by a policy [3]. This research concentrates on the first of these, confidentiality, by focusing on information flow policies that govern access control. Flow policies such as described by Denning [6] and Bell & LaPadula [2] ensure that the confidentiality of high sensitivity objects is maintained by ensuring that access to such objects is limited to only entities that possess the proper rights. As described in [3], such policies are concerned with preventing execution states in which the illicit flow of information or rights could occur.

The first novel concept introduced in this work is a security Domain Model (DM) represented as an Alloy specification [1][10]. The DM provides a means for specifying program state and state transitions, as well as security-related concepts such as subject, information flow, information access, and covert channel vulnerabilities. The model supports formalization of a security policy by providing a framework in which to specify the underlying security properties that represent that policy. The DM is composed of a generic representation of the security policy, and a variable representation of a specific program. The model can be verified, using the model checking capabilities of the Alloy Analyzer, to ensure that the security properties hold for all program executions. Where the properties do not hold, Alloy will identify those specific execution paths. Using a two-part DM, which separates the structural framework of the security policy from the semantics of the target program, significantly improves the efficiency of static analysis with the Alloy Analyzer.

The Implementation Modeling Language (IML) is the second novel concept presented in this work. The IML supports basic information processing in terms of assignment statements, conditional and loop statements, read/write statements, direct file access, and a clock. Models of programs represented in IML notation are called *base programs*, and a DM-Compiler was developed to convert a base program into an Alloy model.

This paper represents work in progress. The examples provide the initial steps toward verification of two types of security properties, which together comprise the possible information flows of a system: those related to access control, and those related to covert channel vulnerabilities. Static analysis of two simple, representative base programs against a Domain Model, shows that it is possible to identify potential security property violations in an implementation. The model is successful in identifying security vulnerabilities in certain program implementations.

Section 2 describes the motivation behind this research, and Section 3 discusses related work in this field. Section 4 presents an overview of the Security Domain Model, the approach to modeling a security policy and program implementation, and Section 5 provides details of the methodology, along with example implementations. Section 6 concludes with a discussion of preliminary results, and planned future work.

2. MOTIVATION

Evaluation standards [6][7] for high assurance secure systems require that they be designed, developed, verified and tested using rigorous processes and formal methods. This evaluation process must include correspondence between system representations at various levels of abstraction, security policy objectives, security specifications, and program implementation. Both functional and security requirements must be considered in this process.

Formal security models provide a precise, high-level representation of a security policy, such as those for confidentiality, availability, or integrity, and provide a basis for proving the validity of the model with respect to the policy [12]. Models are often based on an expression of properties such as secure state and secure transitions for the system, and it may be proven by induction that the transitions preserve the security properties.

This research uses the concept of a *domain model*, defined as “the domain within which an enterprise conducts its business” [13]. A domain model will often include all attributes and operations for classes within an area or field. Whereas security models usually capture information flow between subjects and objects, the DM suggested here does not explicitly define an object, but represents this concept through a variable access table. The table records access levels for program variables across state transitions.

The DM information flow model captures the concept of flows between variables with respect to a system subject, which can be defined as the executor of all program statements. The subject can send and receive information, through Read and Write statements, and has security attributes such as a sensitivity level. The access table is used to determine what information flows may be allowed or disallowed based on the relative access level of the subject and the variables, and the security properties defined by a policy.

3. RELATED WORK

Previous research in the area of modeling secure information flow and access control has incorporated both lattice theory and type theory. Little work that we are aware of has focused on developing both a specification language for formalizing an implementation, as well as a separate abstract framework for expressing the security policy.

Denning's seminal work on secure information flows provides a foundation for this research [6][9]. Her notions of partial ordering of security classes based on the dominance relationship, and the idea of defining state variables to regulate such flows, are integral to the approach described here.

More recent work has extended type systems for information flow analysis [14][15][16]. That work presents the view that secure information flow can be represented as a program property, enabling security classes to be type-checked through static analysis of a program. That approach differs from the present research in that it does not incorporate tools for automated verification that program implementations abide by a security policy, however, the concept of secure flow type systems can potentially be used for representing security information in the security DM.

Alloy has previously been used to model security requirements for secure communications [4]. This work specified predicates 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, which differs significantly from the present research that examines system security.

4. THE SECURITY DOMAIN MODEL APPROACH

The Domain Model approach to program security verification is depicted in Figure 1. Generally, the security DM includes the definition of program state and secure transitions between states, as well as the security properties to be enforced. The DM in this approach specifies security properties as Alloy assertions, to represent the generic properties by which a base program must abide.

As discussed, the DM is composed of an invariant and a variable section, which are initially derived from a required security policy, and, a high-level language program implementation, respectively. A base program is extracted from the implementation, using the IML which provides a common pseudocode-style notation for all base programs, focused on security properties and functionality. By analyzing an implementation model, rather than actual implementation code or the base program, security verification can focus on the programming constructs that bear upon specific security properties, such as I/O, subject access level, direct file access, and timing (clock), while ignoring unnecessary details. This technique provides an example of how static analysis can be performed on imperative programs.

In the current prototype for this research, extraction of a base program from the implementation is not supported by tools, and thus is a manual step. Developing a compiler that will translate any potential implementation from even a modest number of different high-level languages is viewed as a difficult task.

Security properties, written as Alloy assertions, are derived from the required security policy. Because such policies are often written using natural language, extraction of security properties is currently a manual step requiring great diligence to ensure that the objectives of the policy are met. For this research, commonly known security policy models have been used as prototypes.

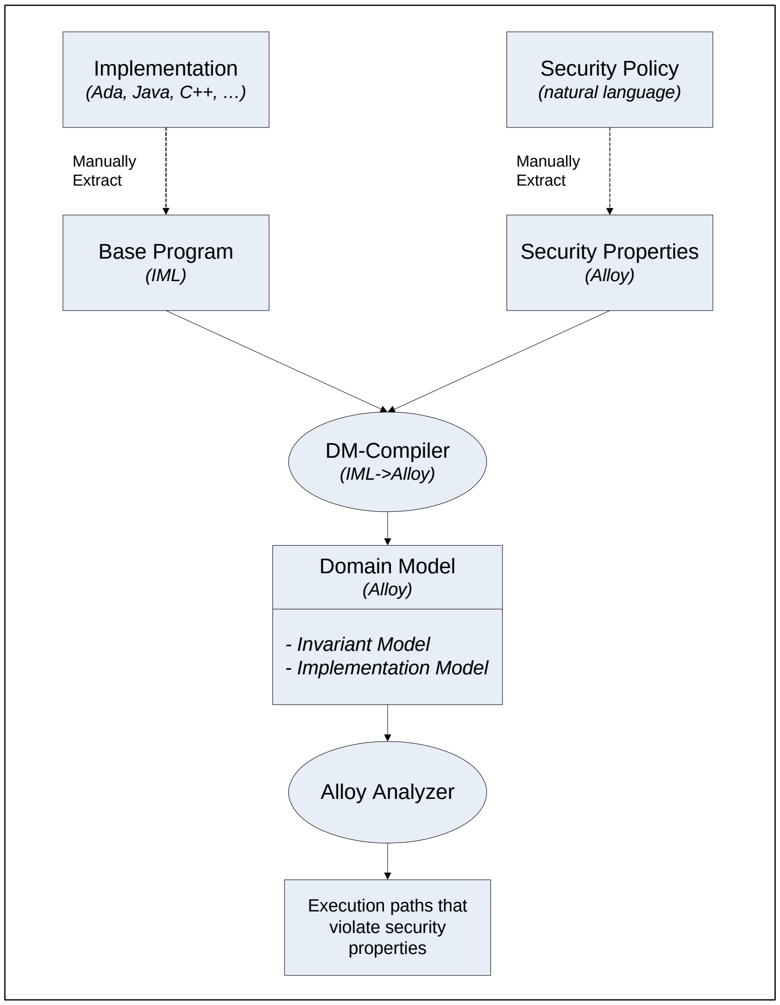

Figure 1. Domain Model Approach to System Security Verification.

Once the base program and security property models are defined, the DM-Compiler translates the base program from IML into state transition predicates, written in Alloy notation, creating the Implementation Model. It combines this derived model with the Invariant Model (security properties, state definitions, and language constructs) to create a complete DM. The approach uses the Alloy Analyzer tool [1] for automated verification of security properties to find execution paths within the DM that might violate the security properties. In essence, it creates a virtual machine for the specific base program, acting as an interpreter of the implementation model.

5. METHODOLOGY

The Implementation Modeling Language (IML) and Domain Model (DM) are introduced in this section.

5.1 Implementation Modeling Language (IML)

The Implementation Modeling Language (IML) enables the abstract specification of a program that is written in some common imperative programming language, e.g., Ada, Java, or C++. A base program written in IML is comprised of a number of basic constructs, as described below.

5.1.1 Lexical concepts

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

Constants are represented by integer constants: -1, 0, 1, etc. The only assumption about values stored in the variables is that they can be compared for equality (= operator) and inequality (`<`, `>`, `<=`, `>=` operators).

Variables can also hold a `time` value obtained by executing the statement `GetClock(variable)`. Statements can be separated by (optional) semicolons.

5.1.2 Assignment statements

Assignment statements propagate access labels attached to the values. Constants have the Low access label by default.

```
variable := variable
variable := constant
```

5.1.3 Read/Write statements

Read and Write statements in IML abstract the input from and output to, respectively, some (single) external device by a subject. The access label, either *Low* or *High* as encoded in the statement name, indicates the access level of the external device being read from or written to in the statement. The `variable` in the statement is assigned the label of the device that is accessed, regardless of the subject's label. For example, the statement `ReadLow(variable)` represents a subject reading a value from the *Low* device into `variable`; conversely, `WriteHigh(source)` represents assignment by a subject of the data from `source` into the *High* device. Note that `source` may be either a variable or a constant.

```
ReadLow(variable)
WriteLow(source)
ReadHigh(variable)
WriteHigh(source)
```

5.1.4 Direct File Access statements

The IML abstracts the concept of random access, where (*key*, *value*) pairs are used to store and retrieve information to a finite sized repository; the *key* and *value* can be either variables or constants. This DirectFile concept can be thought of as analogous to database or memory access. A single instance of a DirectFile in the base program can be accessed by both *Low* and *High* subjects, where a *Low* subject uses `GetLow` and `PutLow`, and a *High* subject uses `GetHigh` and `PutHigh`.

In a `PutLow/PutHigh` operation, the entry *value* is stored at the location indicated by *key*, and the level of the entry is set to the level indicated in the statement. Successful access to the DirectFile, using a valid *key* value, sets the global *Success* flag for the DirectFile; otherwise the *Failure* flag is set. These flags can be used, for example, in condition checks for `if` and `while` statements. The DirectFile has a finite, fixed capacity and becomes full once the final unfilled location is assigned a value; at this point, a global *Full* flag is set. For modeling purposes the capacity of the DirectFile is set to a small number.

The following statements are provided for storing and retrieving values to/from the DirectFile. As an example, `PutLow(key, source)` stores the `source` value into the DirectFile, at `key` location, and assigns the value a *Low* access level.

```
GetLow(key, variable)
PutLow(key, source)
GetHigh(key, variable)
PutHigh(key, source)
```

5.1.5 Clock statement

This statement stores the current clock value in `variable`. Only Read/Write and DirectFile access statements can advance the clock value (each by one clock unit). The clock values can be compared by predicates, `(var1 Before var2)` and `(var1 LongBefore var2)`, within the conditions of `if` and `while` statements.

```
GetClock(variable)
```

5.1.6 Control statements

The `condition` is constructed from variables, constants, flags, predicates =, >, <, >=, <=, `Before`, `LongBefore`, parentheses, and Boolean operations (`not, and, or`). The `statement` may be any statement or block of statements (a sequence of statements enclosed in curly braces). The `else` part is optional.

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

The loop statement repeats its body `while` the condition holds true.

```
while condition do statement
```

A `Stop` statement terminates execution of a program.

5.2 The Domain Model (DM)

The Domain Model (DM) is designed as an Alloy specification whose purpose is to provide a framework for describing security properties for a specific security policy. The DM is composed of two parts: an *invariant model*, and a compiler-generated *implementation model*. The following discussion provides an outline of the main signatures and predicates of the DM, in Alloy notation [1][10].

5.2.1 Invariant Model

The Invariant Model of the DM specifies the concept of state for a base program, and provides the basic concepts generic to the DM. These include statement type and structure, error messages, direct file structure, and clock signature.

The signature (`sig`) in Alloy, to a great degree, corresponds to the class declaration in OO languages. The example below describes a `State` signature, initial state definition, and various structures for variables and values, which are extended in the DM-Compiler generated Implementation Model. For simplicity of the model, only `Low` and `High` access levels are defined.

```
abstract sig Variable{}
abstract sig Value{}
abstract sig Level{}
one sig High, Low extends Level{}

sig State{
    stmt: Statement, -- next statement to execute
    vars: Variable -> (Value + Time), -- establishes variable table
    access: Variable -> Level, -- defines security labels for variable values
    direct_file: DirectFile, -- current snapshot
    current_clock: Time,
    pred_state: lone State, --predecessor state
    err_msg: lone Error
}
```

The `one` qualifier restricts a value to exactly one instance, and the `lone` qualifier restricts a value to either zero or one instance; pairs `(type->type)` represent binary relations, and '+' is the set union operator. Signatures with the `abstract` qualifier do not have their own instances, and are used only to derive other signatures.

An InitialState Alloy signature extends the State signature, and defines initialization values for each of its elements. The current clock in the InitialState is initialized to the first element of Time ordering, i.e., an ordered sequence of Time values (TO/first[]).

```
one sig InitialState extends State{}
{
    -- start with the first statement
    stmt   = s1
    -- all variables initialized with val0 and are Low
    vars   = (Variable -> const0)
    access = (Variable -> Low)
    -- direct_file is empty
    direct_file.full               = NotFull
    direct_file.content.Value      = none
    direct_file.last_result        = Success
    direct_file.last_written_by    = Low
    -- set the clock
    current_clock = TO/first[]
    pred_state = none
    err_msg = none
}
```

The Statement signature describes the general statement structure. Its elements include statement type, destination, source, key and subject access level attributes for Read/Write and Get/Put statements, and source and destination attributes for the assignment statement.

```
abstract sig Statement{
    type:   Stmt_type,
    destination: lone Variable,
    source, key: lone (Variable + Value),
    access:  lone Level, -- for Read/Write only
}
```

The Stmt_type signature is extended to include all types of statements that can be used in a base program.

```
abstract sig Stmt_type {}
one sig Assign, Condition,
    ReadHigh, ReadLow,
    WriteHigh, WriteLow,
    GetHigh, GetLow,
    PutHigh, PutLow,
    GetClock, Stop
extends Stmt_type{}
```

The signature for direct file access captures whether a file is full, its content value, the access level of the most recent source, and whether or not the last access (Get/Put) was successful. The second pair of braces contains an additional constraint on the DirectFile content size, limiting it to 2 in this example.

```
sig DirectFile{
    full:   Full + NotFull,
    content:   Value lone -> Value,
    last_written_by:  Level,
    last_result:  Success + Failure
}
{ -- size is limited
    #content =< 2
    #content = 2 => full = Full else full = NotFull
}
one sig Full, NotFull{}
one sig Success, Failure{}
```

The Clock signature provides an abstract representation of program execution time. The signature defines the concepts of an event that occurs immediately before another event, or one that happens at some time long_before another event.

```
sig Time{}
one sig Clock{
    before:  Time->Time,
    long_before: Time->Time,
}
{   long_before in before   &&
    all disj t1, t2: Time |
          ((t1->t2) in before <=> t2 in TO/nexts[t1]) &&
          ((t1->t2) in long_before <=> some t3: Time |
             (t3 in before[t1] && t3 in before.t2))
}
```

Lastly, the Invariant Model includes the definition of security properties to be enforced by the DM policy. These properties are specified as Alloy assertions, and will be described in further detail in Section 5.2.3.

5.2.2 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 state transitions. A decision was made for the DM-Compiler to generate a state transition predicate for each instance of a base program, as opposed to building the Implementation Model as an interpreter of the base program. This decision significantly reduces the resultant search space for the Alloy Analyzer, thus improving its efficiency, and represents an advantage in compiling over interpretation of the base program.

A simple base program below is used as an example. 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 check, the value in `x1` is either written to a *High* destination, or to a *Low* destination. The labels s1, s2, … are used as signature names in the model below.

```
(s1)  ReadHigh ( x1 );       -- x1 now has High data stored
(s2)  if ( x1 > 3 )  then
(s3)      WriteHigh ( x1 );  -- High data is written to a High device
      else
(s4)      WriteLow ( x1 );   -- High data is written to a Low device
(s5)  Stop;
```

From this program and the DM Invariant Model, the DM-Compiler generates specific variable and value signatures. The number and value of constants defined in the signature depend on the constants explicitly present in the base program (the constant 0 will be always added by default for initial variable values), and on the number of variables. To represent the state space, additional constants may be needed to fill the intervals between explicit constants. The DM-Compiler defines an Alloy signature that establishes a simple *less than* relationship between the required constant values, thus allowing the base program to check for numerical equality and inequality.

```
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)
    ) }
```

Each statement in the base program is represented by a separate Alloy signature. Below are the statement signatures the DM-Compiler generates for statements s1, s2 and s3 in the example base program.

```
one sig s1 extends Statement {}   -- for statement s1
{
    type = ReadHigh
    destination = x1
    source = none
    key = none
    access = High
}

one sig s2 extends Statement {}   -- for statement s2
{
    type = Condition
    source = none
    destination = none
    key = none
}

one sig s3 extends Statement {}   -- for statement s3
{
    type = WriteHigh
    source = x1
    destination = none
    key = none
    access = High
}
```

The DM-Compiler generates a state transition predicate for the base program. This predicate captures the semantics of the base program and the flow of statement operations within it. The code below shows the portion of the state transition predicate for statements s1, s2 and s3 in the example base program.

```
fact trans {
    all st1: State - InitialState | some st: State |
    ( st.stmt = s1 &&
          --  ReadHigh
          ( st1.access = st.access ++ ( x1 -> High ) &&
          some n: Value |
           st1.vars = st.vars ++ ( x1 -> n) &&
          st1.stmt = s2 &&
          st1.direct_file = st.direct_file &&
          st1.current_clock = TO/next[st.current_clock]
          )
    && st1.pred_state = st
    ) or

    ( st.stmt = s2 &&
          --  if
          ( st1.access = st.access  &&
          st1.vars = st.vars  &&
          st1.current_clock = st.current_clock  &&
          st1.direct_file = st.direct_file  &&
          (
           (( const3-> st.vars[ x1 ] ) in LT.lt)
            => st1.stmt = s3
          else st1.stmt = s4)
          )
    && st1.pred_state = st
    ) or

    ( st.stmt = s3 &&
          --  WriteHigh
          ( st1.access = st.access &&
          st1.stmt = s5 &&
          st1.direct_file = st.direct_file &&
          st1.current_clock = TO/next[st.current_clock]
          )
    && st1.pred_state = st
    )
)}
```

The implementation model produced by the DM-Compiler requires consideration of a number of states on the order of:

$$(c + c * v + v)^{v} * s \qquad (1)$$

where c is the number of constants in the base program, v is the number of variables in the base program, and s is the number of statements in the base program. Demonstration of the feasibility of this model is adequate with the Alloy Analyzer, in a reasonable time, using small base program examples (e.g., ones with less than a dozen statements, and only a few variables and constants).

For the next prototype we plan to improve the implementation model by merging execution of linear sequences of non-Read statements into a single transition and shifting a significant part of the state construction to compile time, thus reducing the amount of work performed by the Alloy Analyzer at verification time. This step should lower the upper limit on the state search space to approximately:

$$(c + c * v + v)^{v} * r \qquad (2)$$

where `r` is the number of Read statements in the base program.

5.2.3 Security Properties

Security properties are formalized within the Invariant Model of the DM, and written as Alloy assertions. The assertions are included in the Domain Model generated for any base program. Below are two example security properties.

The first property ensures that a `WriteLow` statement does not write a value from a *high* source, which would result in an illicit information flow from *high* access level to a *low* access device. This assertion is consistent with Bell & LaPadula's *confinement,* or **-property* [2]. The example base program presented in Section 5.2.2 illustrates a potential violation of this property, whereby *high* data is written to a *low* device.

```
assert correct_access1{
    all s: State |  Property1[s] }
pred Property1 [s: State]{
    let stm = s.stmt | {
      (stm.type = WriteLow and stm.source in Variable)
         => s.access[stm.source] = Low }
}
```

The second property ensures that a *low* source does not attempt to write to a *full* Direct File after *high* has written to it. Were this allowed to occur, the result would be that the *low* source, upon receiving e.g. a "file full" error message, could acquire some information from *high*, thus resulting in an illicit information flow from high access level to low access level. This situation represents a type of covert channel known as a *storage channel* [11], and the assertion below defines a security property to check for such a vulnerability.

```
assert correct_access2{
    all s: State |  Property2[s] }
pred Property2 [s: State]{
    let stm = s.stmt | {
           not( stm.type = PutLow &&
             s.direct_file.full = Full &&
             s.direct_file.last_written_by = High) }
}
```

The following example base program illustrates a violation of the storage covert channel property. For this example program, we assume the DirectFile has a capacity of two and has no values stored, as defined in the initialization state. Initially, a *high* subject reads a value from a *high* device into variable `x1`, and a *low* subject reads a value from a *low* device into variable `x2` (statements s1 and s2). The *low* subject then stores the value of `x2` into the DirectFile at *key* location 1 (statement s3). Depending on the value of variable `x1`, the *high* subject then stores that value into the DirectFile at *key* location 2, resulting in the *Full* flag being set (statements s4 and s5).

At this point, the low subject will attempt to check the DirectFile's full state, thus exploiting the covert channel. The full state can be observed by any subject when it attempts to store an element to the DirectFile, so there is no loss of generality, for this purpose, to have the *low* subject read the *Full* attribute directly. To exploit the storage channel, the *low* subject first attempts to store the (*low*) value of `x2` to the DirectFile at *key* location 2 (statement s6). If this attempt fails, i.e., the DirectFile is *Full*, in response to this, the *low* subject writes a '0' to some *low* external device (statement s8); otherwise, he writes a '1' (statement s9).

```
(s1)   ReadHigh (x1);
(s2)   ReadLow (x2);
(s3)   PutLow (1, x2);
(s4)   if (x1 < 0) then
(s5)      PutHigh (2, x1);
(s6)   PutLow (2, x2);
(s7)   if Full=1 then
(s8)      WriteLow (0);
       else
(s9)      WriteLow (1);
(s10)  Stop;
```

By using this method, a high subject and low subject are able to cooperatively exploit a shared resource (the DirectFile) as a means of passing information from a high level down to a low level. With the DM security property assertion above, however, the Alloy Analyzer is able to successfully identify execution paths that can be exploited for such a storage channel.

5.3 Preliminary Results

The suggested approach takes a base program and security properties, compiles them into a DM, and then verifies the generated DM using the Alloy Analyzer. Since the Alloy Analyzer has a very limited capability to handle integers, IML does not support arithmetic operations. The main problem in this approach lies in the limitation on the size of the search space. For the two examples of base programs presented above, the assertion checking time runs from a fraction of a second to approximately 15 seconds using Alloy Analyzer 4.0.

6. CONCLUSIONS AND FUTURE WORK

This paper has presented ongoing research to develop a domain specific model for verifying program implementations against a security policy. The Security Property Model is defined to capture a particular security policy, and is independent of a specific program implementation. It can be used to specify security properties in a generic way. The Implementation Modeling Language (IML) allows for specification of the implementation model. A base program in IML is compiled into an implementation model for the efficiency of model verification (see Figure 1).

By separating the invariant part of the DM from the variable part, which depends on the base program, the efficiency of analysis with the Alloy Analyzer improves significantly. The scalability of the approach, however, is still an issue and will require further work. A potential solution under investigation is to collapse execution of groups of statements into single transitions, thus reducing the total number of states and transitions, and the resultant Alloy Analyzer search space. This might only work for certain models, however, since some security properties may require all statements to be explicitly and individually stated.

There is room to extend the DM and IML to include more security-related constructs. These could include more refined concepts regarding covert channel vulnerabilities, information flow, timing (for detection of covert timing channels), and trusted subjects and processes, to name several. Correspondingly, the list of generic security properties also can be expanded based on well-known security policies.

Finally, the DM might be expanded to parameterize the notion of security policy. This will provide the ability to verify a base program as consistent with a variety of specific security policies, and to use a single security property model for verification of a variety of programs.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by the US Office of Naval Research. 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 US Office of Naval Research.

REFERENCES

[1] 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] Bishop, M. (2002). *Computer Security: Art and Science*. Boston, MA, USA: Addison-Wesley Professional.

[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] Chess, B., West, J. (2007). *Secure Programming with Static Analysis*. Boston, MA, USA: Addison-Wesley Professional.

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

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

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

[9] Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM, 20*(7), 504-513.

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

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

[12] McLean, J. (1994). Security Models. Excerpt from Encyclopedia of Software Engineering (ed. John Marciniak), Wiley Press.

[13] Oldfield, P. (2002). Domain modelling. *Appropriate Process Group*. Retrieved Aug 2, 2007, from http://www.aptprocess.com/whitepapers/index.htm.

[14] Volpano, D., Smith, G., & Irvine, C. (1996). A sound type system for secure flow analysis. *Journal of Computer Security, 4*(2-3), 167-187.

[15] Volpano, D., & Irvine, C. (1997). Secure flow typing. *Computers and Security, 16*(2), 137-144.

[16] Walker, D. (2000). A type system for expressive security policies. *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 254-267). Boston, MA, USA: ACM Press.