Formal Verification and Visualization of Security Policies

Luay A. Wahsheh[1], Daniel Conte de Leon[2], and Jim Alves-Foss[1]*

[1]Center for Secure and Dependable Systems, University of Idaho,
P. O. Box 441008, Moscow, Idaho 83844-1008, USA
Email: {luay, jimaf}@uidaho.edu

[2]Division of Natural Sciences and Mathematics, Lewis-Clark State College,
500 8th Avenue, Lewiston, Idaho 83501-2691, USA
Email: dcontedeleon@acm.org

***Abstract*— Verified and validated security policies are essential components of high assurance computer systems. The design and implementation of security policies are fundamental processes in the development, deployment, and maintenance of such systems. In this paper, we introduce an expert system that helps with the design and implementation of security policies. We show how Prolog is used to verify system correctness with respect to policies using a theorem prover. Managing and visualizing information in high assurance computer systems are challenging tasks. To simplify these tasks, we show how a graph-based visualization tool is used to validate policies and provide system security managers with a process that enables policy reviews and visualizes interactions between the system's entities. The tool provides not only a representation of the formal model, but also its execution. The introduced executable model is a formal specification and knowledge representation method.**

***Index Terms*— Logic, security policy, validation, verification, visualization.**

I. Introduction

High assurance computer systems are those that require convincing evidence that the system adequately addresses critical properties such as security, safety, and survivability [1]. They are used in environments where failure can cause security, safety, or survivability failures. Examples include avionics, weapons control, intelligence gathering, and life-support systems. Before such systems can be deployed, there must exist convincing evidence that they support these critical properties.

Security in high assurance computer systems involves protecting systems' entities from unauthorized (malicious or accidental) access to information. In this context, we use the following terms: *entity* to refer to any source, destination, or intermediary through which information can flow (e.g., user, subject, object, file, and printer); *security enclave* (coalition) to refer to a logical boundary for a group of entities that have the same security level (e.g., CS faculty, ER physicians, and C-130 pilots); and *message* to refer to any data that has been encoded for transmission to or received from an entity (e.g., a method invocation, a response to a request, a program, passing a variable, and a network packet). The transmission mechanism can utilize shared memory, zero-copy message transport, kernel supported transport, TCP/IP, and so forth.

*Corresponding author.

In this paper, we use the term *policy* to refer to *security policy*. In the computer security literature, policy has been used in a variety of ways. Policies can be sets of rules to manage resources (actions based on certain events) or definite goals that help determine present and future decisions. In high assurance secure computer systems, compliance with security policies is mandatory. Policies are different from guidelines, which are optional and recommended actions. Since they are not consistent, guidelines often violate systems' security. We provided a detailed discussion of the meaning of policy in high assurance computer systems in our earlier work [2]. Broadly speaking, a security policy will address security issues: CIA (Confidentiality, Integrity, and Availability). *Confidentiality* is related to the disclosure of information, *integrity* is related to the modification of information, and *availability* is related to the denial of access to information. The security policies discussed in this paper are multi-level (e.g., based on security classification: Top Secret, Secret, Confidential, and Unclassified) and contain mandatory rules to guarantee that only authorized message transmission between entities can occur by imposing constraints on the actions (operations) of these entities. However, our work is not limited to military policies.

One fundamental key to successful implementation of secure high assurance computer systems is the design and implementation of these security policies. The policies must specify the authorized transactions of the system and actions for unauthorized transactions, all in a form that is implementable. Implementing the enforcement of a policy is difficult and becomes very challenging when the system must enforce multiple policies.

Policy verification and validation are essential in the life-cycle of high assurance computer systems because they ensure that the software being developed functions as required (i.e., complies with the policy). The terms verification and validation describe two different concepts that are often used interchangeably. *Verification* is the process of evaluating a system to determine whether the product of a given development phase satisfies the conditions imposed at the start of that phase [3]; that is, verification takes place at the end of each phase to determine whether a phase has been correctly carried out. *Validation* is the process of evaluating a system during or at the end of the development process to determine whether it satisfies specified requirements [3]; that is, validation takes place just before the product is delivered to the client to determine whether the product as a whole satisfies its specifications.

Our research focuses on the Multiple Independent Levels of Security (MILS) architecture, which is a high assurance computer system design for security and safety-critical multi-enclave systems. Although our research is not limited to MILS, it works well in this capacity. MILS is a joint research effort between academia, industry, and government led by the United States Air Force Research Laboratory with stakeholder input from many participants, including the Air Force, Army, Navy, National Security Agency, Boeing, Lockheed Martin, and the University of Idaho [4], [5], [6]. The MILS architecture is created to simplify the process of the specification, design, and analysis of high assurance computer systems [7]. This approach is based on the concept of separation, as introduced by Rushby [8].

The concept of separation is used, for example, in avionics systems and is a requirement of ARINC 653 [9] (a standard for partitioning of computer resources) compliant systems. Through separation, we can develop a hierarchy of security services where each level uses the security services of a lower level or peer entities to provide a new security functionality that can be used by higher levels. Effectively, the operating system and middleware become partners with application level entities to enforce application-specific security policies. Limiting the scope and complexity of the security mechanisms provides us with manageable, and more importantly, evaluatable implementations. A MILS system isolates processes into partitions that define a collection of data objects, code, and system resources. Partitions are defined by the kernel's configuration and can be evaluated separately. This divide-and-conquer approach reduces the proof effort for secure systems.

In this paper, we introduce a model for multi-level security policies and its corresponding Prolog-based implementation. This implementation is an expert system prototype that helps with the design and maintenance of security policies. For the rest of the paper, we use the term model to refer to this expert system prototype. Understanding how the system's entities interact is a key issue in the design and implementation of security policies. We use a graph-based tool that enables policy reviews and visualizes interactions between the system's entities.

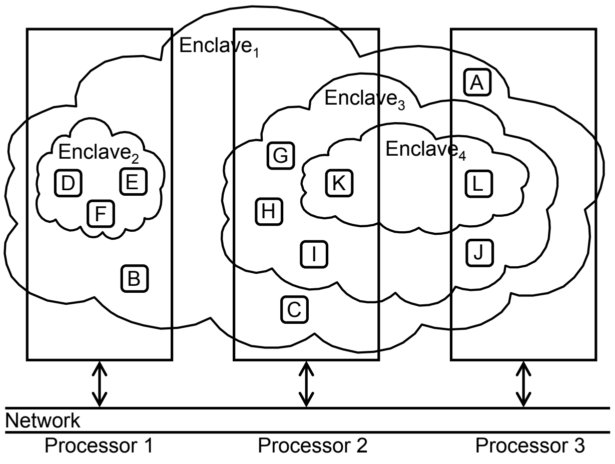

Figure 1. User view structure.

The remainder of this paper is organized as follows: the Inter-Enclave Multi-Policy (IEMP) framework is discussed in Section II. Verifying policies using Prolog is introduced in Section III. Enabling policy reviews through visualization is discussed in Section IV. Related research is outlined in Section V. Finally, we conclude the paper in Section VI.

II. Inter-Enclave Multi-Policy Framework

We introduced the IEMP framework in our earlier work [2]. In this paper, we introduce a formal verification and validation of IEMP security policies. We implement a Prolog-based model of IEMP and introduce not only visual representations of IEMP policies, but also the execution of the model. This section summarizes IEMP and introduces additional illustrations that help provide a visual description of entities' requests.

An enclave sets a logical boundary for a group of entities that can communicate with one another according to an individual security policy responsible for that enclave. Each enclave has its own individual security policy that controls communication between entities that belong to the enclave. While an individual policy controls message communication within its enclave, IEMP handles message communication between two or more enclaves. Enclaves can be arranged in a hierarchical structure and may exist across multiple processors. Figs. 1 and 2 show MILS enclaves distributed over separate processors in user and system views, respectively.

Any interaction between MILS entities is modeled as an entity e_1 accessing another entity e_2 through access operation op (e.g., *read* and *write*). $P(e_1, e_2, op)$ denotes the application of policy P to access (e_1, e_2, op), so $P(e_1, e_2, op)$ is of type *grant* or *reject*.

$Enclave_P$ is used to denote the domain belonging to P which consists of all entities that are submitted to P. For any access (e_1, e_2, op), a policy P will contain an access rule if and only if $e_1, e_2 \in Enclave_P$. $S_e = \{P|e \in$

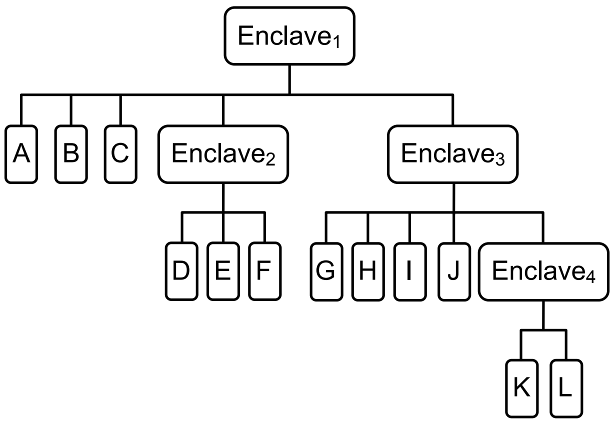

Figure 2. System view structure.

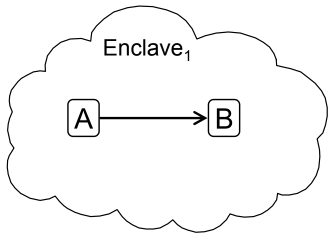

Figure 3. An example of Class 1 access.

$Enclave_P\}$ denotes the set of security policies that have entity e within their enclave and $\{P_i\}_{i\in I}$ denotes a set of security policies.

In a multi-policy enclave system with a set of security policies $\{P_i\}_{i\in I}$ and $e_1, e_2 \in \bigcup_{i\in I} Enclave_{P_i}$, any access (e_1, e_2, op) belongs to one of the following three disjoint classes:

- Class 1: $|S_{e_1}| = |S_{e_2}| = 1 \quad \wedge \quad S_{e_1} = S_{e_2}$
  Class 1 identifies the case in which interactions occur when both entity e_1 and entity e_2 belong to exactly one enclave (the same enclave). Fig. 3 shows an example of Class 1 access.
- Class 2: $|S_{e_1} \cap S_{e_2}| = 0$
  Class 2 identifies the case in which no security policy exists that has both entity e_1 and entity e_2 in its enclave. Two sub-classes exist:
  a. $|S_{e_1}| = 1 \quad \wedge \quad |S_{e_2}| = 1$
  Where each entity is a member of only one enclave. Fig. 4 shows an example of Class 2.a access.
  b. $\exists e \in \{e_1, e_2\} : |S_e| > 1$
  Where at least one of the entities is a member of more than one enclave. Fig. 5 shows an example of Class 2.b access.
- Class 3: $|S_{e_1} \cap S_{e_2}| \geq 1 \wedge \exists e \in \{e_1, e_2\} : |S_e| > 1$
  Class 3 identifies the case in which at least one policy provides an access rule for both entities and at least one of the involved entities is a member of more than one enclave. Two sub-classes exist:
  a. $|S_{e_1} \cap S_{e_2}| = 1$
  Where an entity is a member of more than one policy enclave. Fig. 6 shows an example of Class 3.a access.
  b. $|S_{e_1} \cap S_{e_2}| > 1$
  Where more than one policy exist for both entities that provide rules for the access. Fig. 7 shows an example of Class 3.b access.

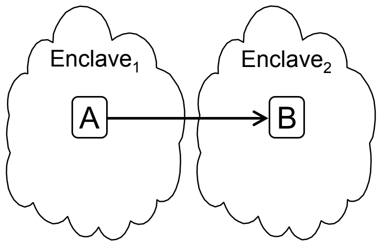

Figure 4. An example of Class 2.a access.

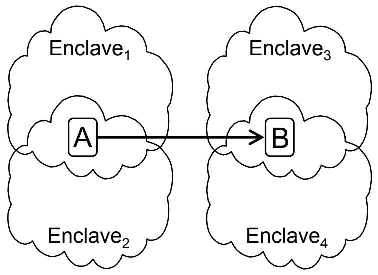

Figure 5. An example of Class 2.b access.

III. Formal Verification of Policies

Prolog is a first-order predicate logic programming language that uses Horn clauses to describe relationships based on mathematical logic. A Prolog program consists of clauses stating facts and rules. Facts and rules are explicitly defined and implicit knowledge can be extracted from Prolog's knowledge base. Queries are used to check whether relationships hold.

We use Prolog as a method to prove system correctness with respect to policies using Prolog's theorem prover that is based on a special strategy for resolution called SLD-resolution [10]. Unlike programs in many other procedural programming languages, a Prolog program is not a sequence of processing steps to be executed. A Prolog program is a set of formulas (axioms) from which other formulas expressing properties of this program may

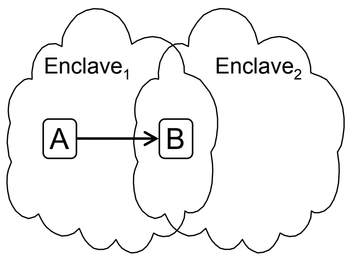

Figure 6. An example of Class 3.a access.

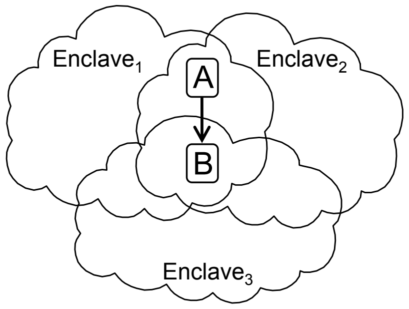

Figure 7. An example of Class 3.b access.

be deduced as theorems [11]. A theorem is proved using a proof procedure that is a sequence of *rules of inference* producing new expressions from old ones; the inference rules are repeatedly applied to a set of formulas and the new expressions, until the desired theorem appears [12]. In Prolog, the theorem to be proved is the starting goal (clause), while the inference rules and formulas are the program itself.

Because of its simple declarative nature, Prolog is an appropriate language for expressing and verifying security policies. To determine all possible answers to a query, Prolog supports backtracking. Backtracking is the process of determining all facts or rules with which unification (determining whether there is a substitution that makes two atoms the same) can succeed. Many authors in the literature indicated the use of Prolog as a policy specification language, including Lin [13] who argued that Prolog is a suitable language for specifying security policies due to multiple features: it is based on a subset of first-order logic with a solid mathematical foundation, it is a productive modeling language supporting incremental policy writing and refinement, it is able to reason from a set of rules, and it supports meta-level reasoning which makes policy conflict detection possible.

A. Implementation

Our model is menu-driven and test cases were run against it to check not only the program's correctness, but also its robustness; we provided invalid input data to determine whether the program was capable of dealing with bad data (e.g., invalid data types of arguments). We do not include the whole implementation model in this paper, we discuss an excerpt from it: only facts and rules (for Classes 1, 2, and 3) are discussed that demonstrate a knowledge base scenario. The expressed policy is based on the simple security property (no read-up: an entity cannot read from another entity at a higher security classification than itself) and star property (no write-down: an entity cannot write to another entity at a lower security classification than itself) as defined by Bell-LaPadula [14]. As demonstrated in the model, we use a *closed* security policy where only the allowed operations are specified; only authorized entities are allowed access. The operations to be denied are not explicitly specified because Prolog's *negation-by-failure* mechanism will enforce a default denial on messages other than those explicitly allowed by the knowledge base and inference rules.

B. Facts

We present a scenario that consists of entities that are users; however, in general, entities can be files, printers, or any source or destination through which information can flow. In the scenario, entities, enclaves, roles, and allowed operations between entities are defined as Prolog facts. The entity set is EntityId_EntityName = {(210,diala), (456,trudy), (698,evey), (331,raneem), (704,trudy), (555,penny), (369,dana), (491,diosa), (675,tina), (810,sam)}; the enclave set is Enclave = {1, 2, 3, 4}; the role set is Role = {Faculty, Staff, BS Student, MS Student, PhD Student}; and the operation set is Operation = {Read, Write}.

We define a security classification set Security_Classification = {1, 2, 3, 4}. We use the following values for security: 1 to refer to a security classification of Unclassified, 2 to refer to a security classification of Confidential, 3 to refer to a security classification of Secret, and 4 to refer to a security classification of Top Secret. A higher number indicates more authority.

We also define a priority set Priority = {1, 2, 3, 4}. A priority (importance) is a factor in determining whether an entity is given permission to access another entity for Class 3 accesses. An entity is assigned a priority number based on the highest security classification of the enclaves that it belongs to. A higher number indicates more authority.

Enclaves, roles, security classifications, and priority values, which are defined as Prolog facts, are assigned to entities. The format of the classification() predicate is:

classification(EntityId, EntityName, Enclave, Role,
SecurityClassification, Priority).

C. Rules

After Prolog facts have been defined, Prolog rules for Classes 1, 2, and 3 are stated. We briefly discuss Class 1 rules and include excerpts from Classes 2 and 3 rules.

1) Class 1 Rules: Class 1 rules process read and write operation requests for entities that are members of the same enclave. We define an allow() predicate that processes entity access operations. The format of the allow() predicate is:

allow(EntityAId/EntityAName/EnclaveA/RoleA,
    Operation,
    EntityBId/EntityBName/EnclaveB/RoleB,
    Response)

where Response is either to allow or deny entity access.

2) Classes 2 and 3 Rules: Class 2 identifies the case in which no security policy exists that has both entity A and entity B in its enclave, defined as follows:

```
%if entities belong to different enclaves,
%then set Class 2.a flag
```

```
not_equal(EncA,EncB) ->
  flag(FlagClass2a),

%check entities A and B for Class 2.b
%access
class_2_b(IdA/EntA/EncA/RoleA,
  IdB/EntB/EncB/RoleB,NAX,NBY),

%check whether entities A and B belong
%to more than one enclave
entity_belongs_more_than_one_enclave(
  NAX,NBY,FlagClass2b),

%in Class 2.a and Class 2.b access,
%check if the flag of the ss-property
%or *-property has been set and check
%whether entity A has a higher priority
%than that of B’s
((FlagClass2a == 1) ->
   (( not_equal(
        FlagPrintSSProperty,1);
      not_equal(
        FlagPrintStarProperty,1)
    ) ->
      (higher_or_equal_priority(
        PriorityA,PriorityB) ->
        !
      ; flag(FlagClass2aNotAllowed)
      )
    ; !
   )
 ; ((FlagClass2b == 1) ->
      (( not_equal(
           FlagPrintSSProperty,1);
         not_equal(
           FlagPrintStarProperty,1)
       ) ->
         (higher_or_equal_priority(
           PriorityA,PriorityB) ->
           !
         ; flag(
             FlagClass2bNotAllowed)
         )
       ; !
      )
   ; !
   )
).
```

Class 3 identifies the case in which at least one policy provides an access rule for both entity A and entity B and at least one of the involved entities is a member of more than one enclave. A built-in Prolog predicate called findall() is used to find all solutions in a list (data type): findall(S,C,L), where L is the list of all S that satisfy condition C. Prolog generates solutions one by one by using backtracking, but in order to collect all solutions in a list, Prolog uses the built-in predicates: bagof(), setof(), or findall().

We build a list of the current entities and check class type (3.a or 3.b) as follows:

```
%get a list of the source and destination
%entities that belong to Class 3
findall(IdA/EntA/EncAXX/RoleAXX,
  member(IdA/EntA/EncAXX/RoleAXX,
  IdEntEncRolListClass3),_),
findall(IdB/EntB/EncBYY/RoleBYY,
  member(IdB/EntB/EncBYY/RoleBYY,
  IdEntEncRolListClass3),_),

%get a list of the source and destination
%enclaves from the Class 3 list
findall(EncAXX,
  member(IdA/EntA/EncAXX/RoleA,
  IdEntEncRolListClass3),
  EntAClass3EnclaveListNew),
findall(EncBYY,
  member(IdB/EntB/EncBYY/RoleB,
  IdEntEncRolListClass3),
  EntBClass3EnclaveListNew),

%get the intersection of the source and
%destination lists
intersection(
  EntAClass3EnclaveListNew,
  EntBClass3EnclaveListNew,
  IntersectionList),

%check whether the access of the entities
%in the intersection list is Class 3.a or
%Class 3.b
class_3a_3b(IntersectionList,
  FlagClass3a,FlagClass3b),

%in Class 3.a and Class 3.b access,
%check if the flag of the ss-property
%or *-property has been set and check
%whether entity A has a higher priority
%than that of B’s
( (FlagClass3a == 1) ->
    %class 3.a
    (( not_equal(
         FlagPrintSSProperty,1);
       not_equal(
         FlagPrintStarProperty,1)
     ) ->
       (higher_or_equal_priority(
         PriorityA,PriorityB) ->
         !
       ; flag(FlagClass3aNotAllowed)
       )
     ; !
    )
  ; ((FlagClass3b == 1) ->
       %class 3.b
       (( not_equal(
            FlagPrintSSProperty,1);
          not_equal(
            FlagPrintStarProperty,1)
        ) ->
          (higher_or_equal_priority(
            PriorityA,PriorityB) ->
            !
          ; flag(
              FlagClass3bNotAllowed)
          )
        ; !
       )
     ; !
    )
).
```

We define a dominate() predicate that determines whether entities dominate one another (based on the simple security and star properties as defined by Bell-LaPadula [14]). The format of the dominate()

predicate is:

dominate(EntityAId/EntityAName/EnclaveA/RoleA,
EntityBId/EntityBName/EnclaveB/RoleB).

The rule is defined as follows:

```
%check whether entities dominate one
%another based on their security
%classifications
dominate(IdAX/EntAX/EncAX/RoleAX,
  IdBY/EntBY/EncBY/RoleBY):-
  classification(IdAX,EntAX,EncAX,
    RoleAX,CL1X,_),
  classification(IdBY,EntBY,EncBY,
    RoleBY,CL2Y,_),
  (CL1X >= CL2Y).
```

IV. Enabling Policy Reviews Through Visualization

SHriMP (Simple Hierarchical Multi-Perspective) is a graph visualization tool that is designed to enable the exploration of complex software programs and knowledge bases [15]. In our model, we use the stand-alone SHriMP which is a Java application that uses GXL data format. GXL is an XML-based standard exchange format for graph representation [16]. In order to present information flow graphical representations of the system, we developed a software interface that provides a communication mechanism between our model and the SHriMP tool. The interface maps Prolog's facts and rules into a standard graphical format that can be shared with other visualization tools.

A. Benefits of Policy Visualization

Being able to explore and navigate hierarchical representations of security policies provide significant benefits. Visual representations enable system security managers to perform policy reviews, such as walkthroughs, that help uncover errors and deficiencies during the policy development and maintenance processes. Because it can display graph hierarchical structures in Java (which is a common software, thus providing flexibility and portability), SHriMP can integrate GXL's standard format with other tools in different domains. Although in this paper we show visual representations of a relatively small policy scenario, SHriMP can scale to larger complex policy systems [17]. SHriMP provides different methods of visualization that can produce different views of data that are hidden in the model. Data can be filtered (modified) to customize how information is presented.

B. Policy Visualization

We use directed graphs to visually present information flow between entities. Nodes in a graph represent entities and arcs (between the nodes) represent allowed communication between entities. Since a group of entities that are members of an enclave can communicate with one another within that enclave without restrictions (as discussed in Section II), we do not show any read and write operations (arcs) in the graphs for Class 1 entity access; arcs are shown only for entities that communicate between different enclaves.

In order to display a message on the screen that describes allowed communication between entities, we hover the mouse pointer over a certain arc, and a message indicating properties of the arc will be displayed; moving the pointer away will cause the message to disappear. For example, the message "dana_369_faculty_4 —read—> tina_675_ms_stud_2" indicates that Dana, identification number of 369, role of faculty, and security classification of 4 is allowed to read from Tina, identification number of 675, role of MS student, and security classification of 2.

In order to provide views of the system's entities and characteristics of the policy model, we present some graphical representations. The following figures show policy visualization representations of the Prolog model. The representations that we discuss in this paper are subsets of what have been implemented; we show the following allowed accesses: all read and write operations, Class 2.a read operation, Class 2.b write operation, Class 3.a read operation, and Class 3.b write operation.

Fig. 8 shows read visual representations for all the system's entities; the arrows indicate the direction of one-way read operations. Fig. 9 shows write visual representations for all the system's entities; the arrows indicate the direction of one-way write operations. In both figures, the arrows, shown as bold lines, indicate the direction of one-way read and write operations for a specific entity (Dana, identification number of 369, role of faculty, and security classification of 4).

Fig. 10 shows a read operation for Class 2.a access, where Diala, identification number of 210, role of faculty, and security classification of 4 is allowed to read from Trudy, identification number of 704, role of staff, and security classification of 2 (shown as a bold, solid, red line).

Fig. 11 shows a write operation for Class 2.b access, where Diosa, identification number of 491, role of staff, and security classification of 1 is allowed to write to Dana, identification number of 369, role of faculty, and security classification of 3 (shown as a bold, dashed, blue line).

Fig. 12 shows a read operation for Class 3.a access, where Diala, identification number of 210, role of faculty, and security classification of 4 is allowed to read from Dana, identification number of 369, role of faculty, and security classification of 3 (shown as a bold, solid, red line).

Fig. 13 shows a write operation for Class 3.b access, where Sam, identification number of 810, role of BS student, and security classification of 2 is allowed to write to Dana, identification number of 369, role of faculty, and security classification of 3 (shown as a bold, dashed, blue line).

V. Related Work

Significant related work regarding security policies had been reported in the literature, including our earlier

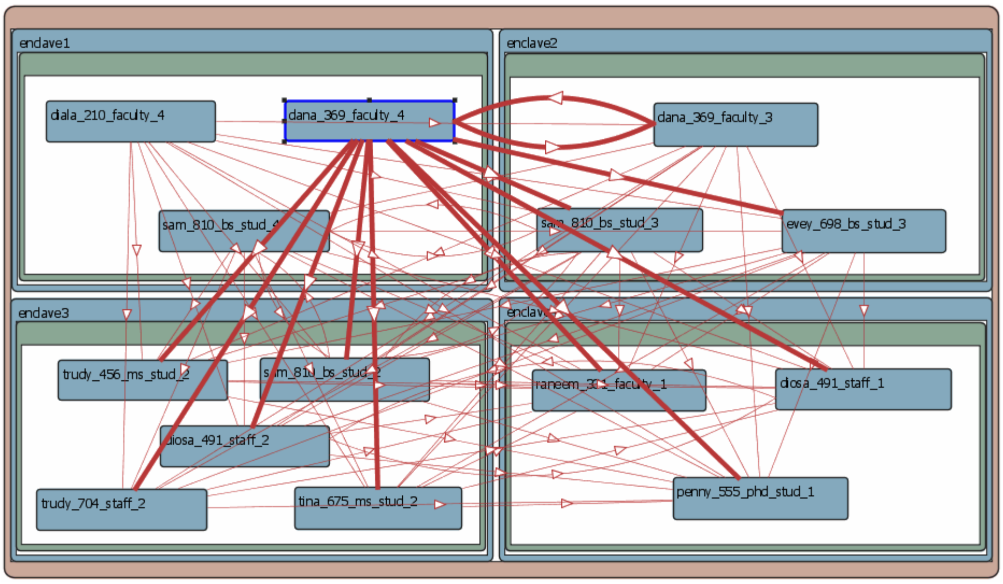

Figure 8. Read operations.

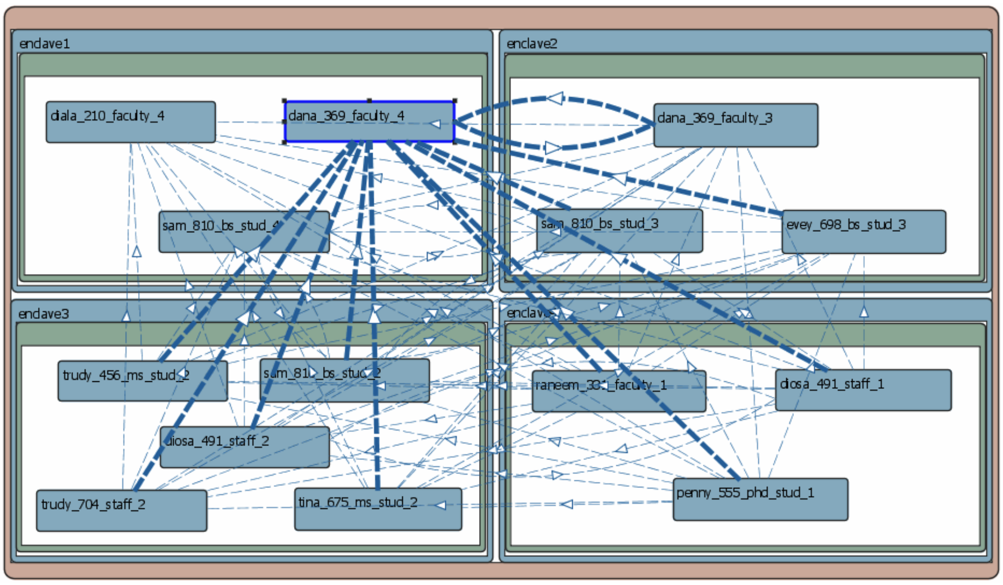

Figure 9. Write operations.

work [2], [18], [19]. In this section, we focus on related policy work regarding Prolog and visualization techniques.

Although various languages had been proposed for specifying policies for different purposes, a standard language does not yet exist for the policy community to use. Moffett and Sloman [20] provided an analysis of policy hierarchies by specifying policy hierarchy refinement relationships in Prolog. Lupu and Sloman [21] applied Prolog to meta-policies to identify several types of policy conflicts. DeTreville [22] presented Binder, a logic-based security language that provides low-level programming tools to implement security policies. Binder adopted Prolog's syntax and its programs can be translated into Prolog.

A discussion of multiple visualization techniques can be found in recent surveys [23], [24], [25]. One of the first visualization environments that were used by system developers was FIELD (Friendly Integrated Environment for Learning and Development) that provided an understanding of programs' behavior [26]. SmartMap is a security policy visualization tool that provides improved security by allowing security managers to validate the integrity of their security policy prior to deployment [27]. The G^{SEE} (Generic Software Exploration Environment) tool was applied to improve the understanding of different software products [28].

VI. Conclusions

A fundamental issue in high assurance computer systems is the protection of information against unauthorized access. This paper introduces a Prolog-based

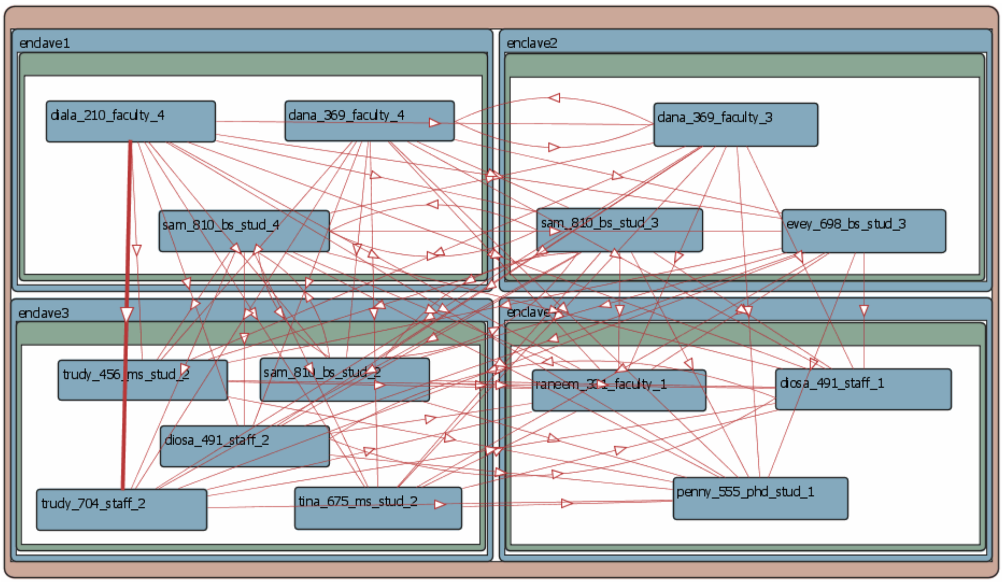

Figure 10. Class 2.a read operation.

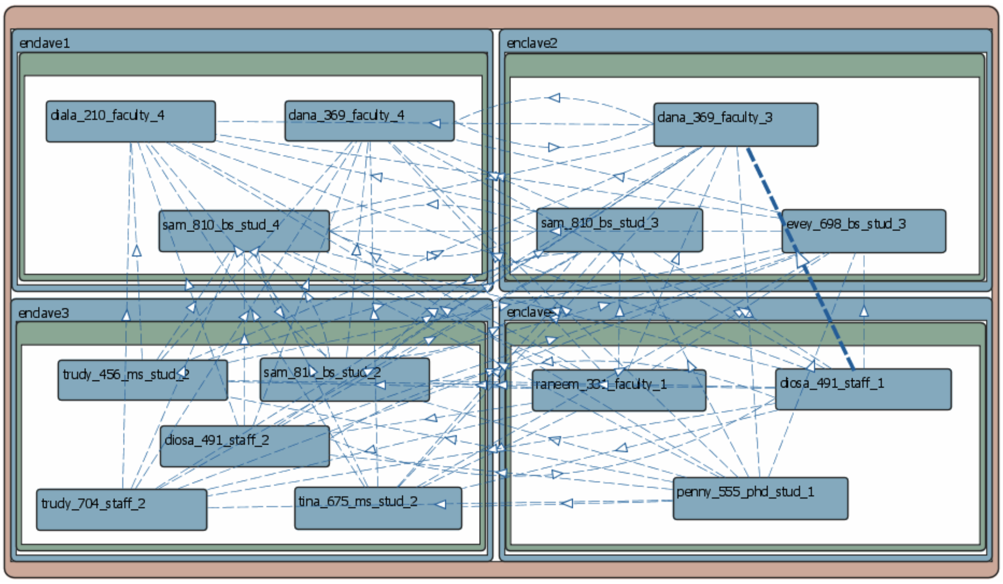

Figure 11. Class 2.b write operation.

model and shows how Prolog is formally used to verify the security policies of a multi-level secure high assurance computer system. In order to reduce policy life-cycle development time, we show how visual representations of security policies are used to provide system security managers with a tool that helps in the process of development and maintenance of security policies.

The approach proposed in this paper is an important step towards leveraging a set of methods and tools that help with the design of security policies. Future investigation is needed to better understand policy systems' behavior and further explore other benefits of policy visualization, including measuring security metrics (properties) and providing a tool for testing system usability.

Acknowledgements

We wish to acknowledge the United States Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) for their support. This material is based on research sponsored by AFRL and DARPA under agreement number F30602-02-1-0178. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, or the U.S. Government. We also wish to acknowledge the anonymous reviewers and journal editors for reviewing this paper.

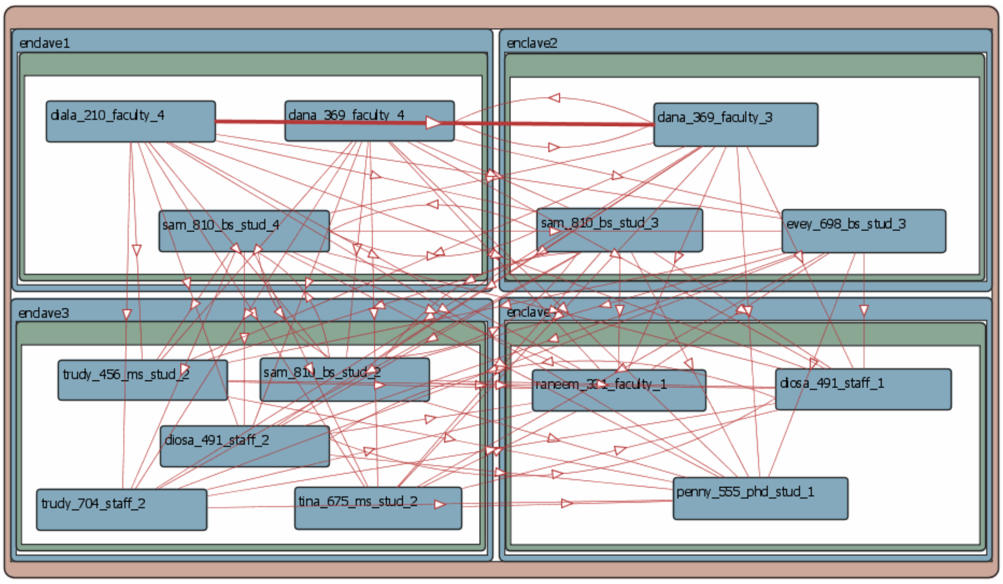

Figure 12. Class 3.a read operation.

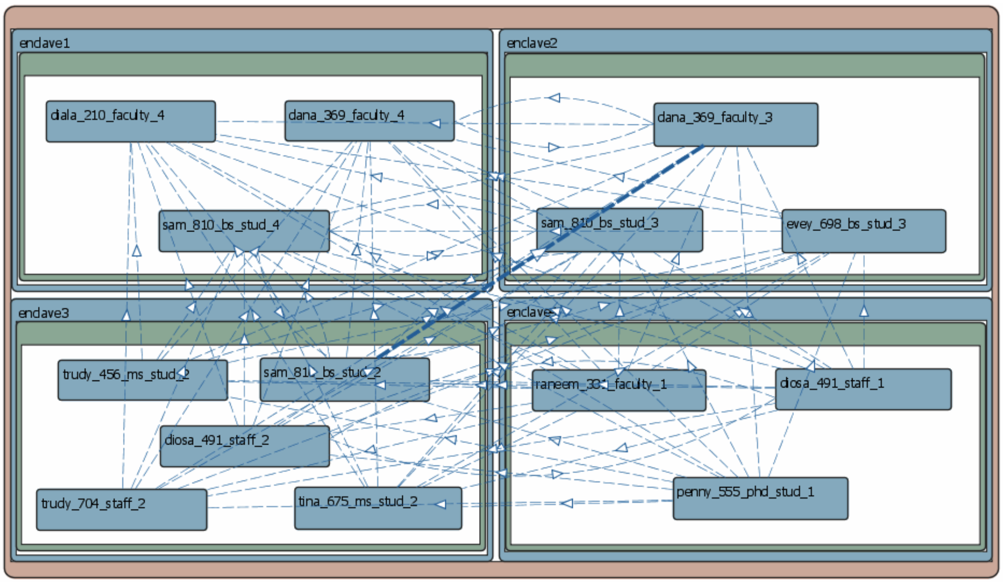

Figure 13. Class 3.b write operation.

REFERENCES

[1] C. Heitmeyer, “Managing complexity in software development with formally based tools,” *Electronic Notes in Theoretical Computer Science*, vol. 108, pp. 11–19, December 2004.

[2] L. A. Wahsheh and J. Alves-Foss, “Specifying and enforcing a multi-policy paradigm for high assurance multi-enclave systems,” *Journal of High Speed Networks*, vol. 15, no. 3, pp. 315–327, October 2006.

[3] IEEE Standards Board, “IEEE standard glossary of software engineering terminology,” IEEE Standard 610.12-1990, September 1990.

[4] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor, “The MILS architecture for high assurance embedded systems,” *International Journal of Embedded Systems*, vol. 2, no. 3/4, pp. 239–247, 2006.

[5] J. Alves-Foss, C. Taylor, and P. Oman, “A multi-layered approach to security in high assurance systems,” in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, January 2004.

[6] W. S. Harrison, N. Hanebutte, P. Oman, and J. Alves-Foss, “The MILS architecture for a secure global information grid,” *Crosstalk: The Journal of Defense Software Engineering*, vol. 18, no. 10, pp. 20–24, October 2005.

[7] P. White, W. Vanfleet, and C. Dailey, “High assurance architecture via separation kernel,” October 2000, draft.

[8] J. M. Rushby, “Design and verification of secure systems,” in *Proceedings of the 8th ACM Symposium on Operating System Principles*, December 1981, pp. 12–21.

[9] “Avionic application software standard interface (Draft 3 of Supplement 1) (Specification ARINC 653),” 2003, ARINC Standards.

[10] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, 2nd ed. The MIT Press, 1994.

[11] J. Loeckx and K. Sieber, *The Foundations of Program Verification*, 2nd ed. John Wiley & Sons and B. G. Teubner, 1987.

[12] P. Civera, G. Masera, G. Piccinini, and M. Zamboni, *VLSI Prolog Processor, Design and Methodology: A Case Study in High Level Language Processor Design*. North-Holland, 1994.

[13] A. Lin, “Integrating policy-driven role based access control with the common data security architecture,” Hewlett-Packard Laboratories, Tech. Rep. HPL-1999-59, April 1999.

[14] D. E. Bell and L. J. LaPadula, “Secure computer systems: Unified exposition and MULTICS interpretation,” MITRE Corporation MTR-2997 Rev. 1, Tech. Rep. ESD-TR-75-306, March 1976.

[15] CHISEL research group, “SHriMP,” Department of Computer Science, University of Victoria. Retrieved December 20, 2007, from http://www.thechiselgroup.org/shrimp.

[16] A. Winter, B. Kullbach, and V. Riediger, “An overview of the GXL graph exchange language,” in *Software Visualization*, vol. 2269, 2002, pp. 324–336, Lecture Notes in Computer Science (LNCS), S. Diehl (Ed.), Springer-Verlag.

[17] C. Best, M.-A. Storey, and J. Michaud, “Designing a component-based framework for visualization in software engineering and knowledge engineering,” in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, July 2002, pp. 323–326.

[18] L. A. Wahsheh and J. Alves-Foss, “Using policy enforcement graphs in a separation-based high assurance architecture,” in *Proceedings of the IEEE International Conference on Information Reuse and Integration*, August 2007, pp. 183–189.

[19] L. A. Wahsheh and J. Alves-Foss, “Policy-based security for wireless components in high assurance computer systems,” *Journal of Computer Science*, vol. 3, no. 9, pp. 726–735, 2007.

[20] J. D. Moffett and M. S. Sloman, “Policy hierarchies for distributed systems management,” *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1404–1414, December 1993.

[21] E. C. Lupu and M. Sloman, “Conflicts in policy-based distributed systems management,” *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 852–869, November/December 1999.

[22] J. DeTreville, “Binder, a logic-based security language,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002, pp. 105–113.

[23] R. R. Kasemsri, “A survey, taxonomy, and analysis of network security visualization techniques,” Master’s thesis, Georgia State University, December 2005.

[24] S. Bassil and R. K. Keller, “Software visualization tools: Survey and analysis,” in *Proceedings of the 9th International Workshop on Program Comprehension*, May 2001, pp. 7–17.

[25] R. Koschke, “Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 2, pp. 87–109, March 2003.

[26] S. P. Reiss, *FIELD: A Friendly Integrated Environment for Learning and Development*. Kluwer, 1994.

[27] C. Tobkin and D. Kligerman, *Check Point NG/AI: Next Generation with Application Intelligence Security Administration*. Syngress, 2004.

[28] J.-M. Favre, “G^{SEE}: A generic software exploration environment,” in *Proceedings of the 9th International Workshop on Program Comprehension*, May 2001, pp. 233–244.

Luay A. Wahsheh is an assistant professor of computer science who will be joining the University of Baltimore in Baltimore, Maryland, USA in August 2008. His research involves computer security, with an emphasis on the design and implementation of security policies in high assurance computer systems. He received a Ph.D. in computer science from the University of Idaho in Moscow, Idaho, USA in 2008, a master’s degree in computer science from Stephen F. Austin State University in Nacogdoches, Texas, USA in 2000, a postgraduate diploma in computer science from the University of Essex in Colchester, United Kingdom in 1994, and a bachelor’s degree in computer science from Mutah University in Karak, Jordan in 1992.

He worked in the Center for Secure and Dependable Systems at the University of Idaho from September 2003 to May 2008. Prior to joining the University of Idaho, he spent three years as a lecturer in the Department of Computer Science at Stephen F. Austin State University, where he taught classes in computer science.

Dr. Wahsheh is a member of Upsilon Pi Epsilon computing sciences honor society.

Daniel Conte de Leon is an assistant professor of computer science in the Division of Natural Sciences and Mathematics at Lewis-Clark State College in Lewiston, Idaho, USA. His current research interests are in the creation of novel extended programming and specification methods and languages for the development of safe and secure high assurance and critical computing systems. In addition, he is interested in developing novel approaches for the efficient use, conversion, and conservation of energy. He received Ph.D. and master’s degrees in computer science from the University of Idaho in Moscow, Idaho, USA in 2006 and 2002, respectively.

Prior to joining Lewis-Clark State College, he worked at the Center for Secure and Dependable Systems at the University of Idaho, where he designed a novel methodology for the specification and visualization of safe and secure system architectures.

Dr. Conte de Leon is a member of the ACM and the IEEE.

Jim Alves-Foss is a professor of computer science in the Department of Computer Science at the University of Idaho in Moscow, Idaho, USA. His main research interests are in the design and analysis of secure distributed systems, with a focus on formal methods and software engineering. He received Ph.D. and master’s degrees in computer science from the University of California – Davis in Davis, California, USA in 1991 and 1989, respectively, and a bachelor’s degree in mathematics, computer science, and physics also from the University of California – Davis in 1987.

He is the director of the Center for Secure and Dependable Systems at the University of Idaho and has been a faculty member in the Department of Computer Science since August 1991.

Dr. Alves-Foss is a senior member of the IEEE.