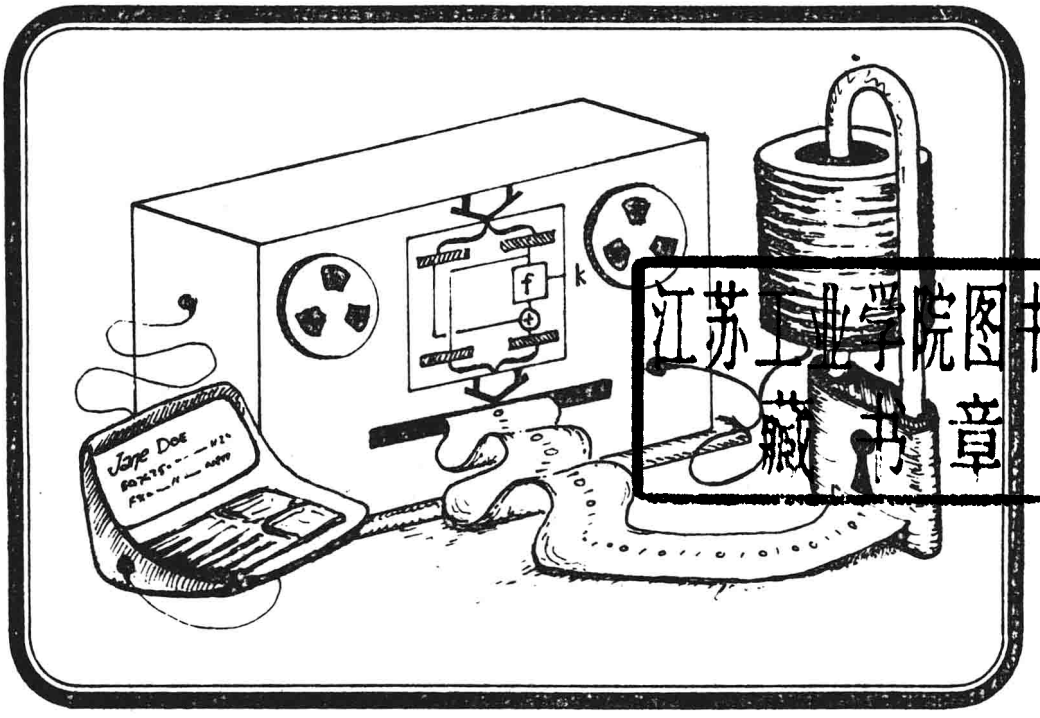
Proceedings of the 1985 Symposium on Security & Privacy

Proceedings of the 1985 Symposium on

Security and Privacy

April 22-24, 1985 Oakland, California

Sponsored by the
Technical Committee on Security and Privacy
IEEE Computer Society



ISBN 0-8186-0629-0
IEEE Catalog Number 85CH2150-1
Library of Congress Number 85-60166
Computer Society Order Number 629

IEEE COMPUTER SOCIETY PRESS

1884 1984 THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

IEEE COMPUTER SOCIETY

Published by IEEE Computer Society Press
1109 Spring Street
Suite 300
Silver Spring, MD 20910

ISBN 0-8186-0629-0 (paper)
ISBN 0-8186-4629-2 (microfiche)
ISBN 0-8186-8629-4 (casebound)
IEEE Catalog Number 85CH2150-1
Library of Congress Number 85-60166
Computer Society Order Number 629

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

IEEE
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

1985 IEEE Symposium on Computer Security and Privacy

General Chairman
Robert Morris
AT&T Bell Laboratories

Program Co-Chairmen

Jonathan K. Millen
MITRE Corp.

Virgil D. Gligor
University of Maryland

Program Committee

Stephen Kent
BBN, Cambridge, MA

Richard Platek
Odyssey Research Associates, Ithaca, NY

David Bailey
Los Alamos National Laboratory, Los Alamos, NM

Carl Landwehr
Naval Research Laboratory, Washington, DC

Steven Lipner
Digital Equipment Corp., Maynard, MA

Thomas Berson
SYTEK, Inc., Mountain View, CA

Deborah Downs
Aerospace Corp., Los Angeles, CA

John Woodward
MITRE Corp., Bedford, MA

Preface

This year's Symposium on Security and Privacy is sixth in a series of annual symposia which helped establish computer security as a distinct discipline of computer science and engineering. As a discipline of computer science and engineering, computer security demands a blend of both theoretical and practical thinking. It offers rich opportunities for contributions from either direction and for the creative application of theoretical results in real systems. Undoubtedly, this Symposium continues the established tradition of fostering this interplay of needs and ideas.

The computer security topics addressed by this year's Symposium cover a wide range, including verification methods and tools, network security, cryptographic algorithms and techniques, security policy, and OS and DBMS security mechanisms. The growing interest in this Symposium is reflected by the large number of submissions. We received 63 submissions of papers and 2 panel proposals—a substantial increase over previous years. Each paper was reviewed by at least three members of the Program Committee. The results of the review process led to the selection of the 25 papers for inclusion in the three-day, nine-session program. For the first time, the best papers from this Symposium will be published in an issue of IEEE Software during 1985.

We would like to thank all the authors who submitted papers to the Symposium and the Program Committee members who, despite their geographic distribution and a demanding review schedule, responded promptly to all our demands. Last but not least our thanks are due to Ms. Lee Blue of the IEEE Computer Society for her efforts and patience in producing the Symposium Proceedings.

Jonathan K. Millen
Program Co-Chairman

Virgil D. Gligor
Program Co-Chairman

Table of Contents

Database Security

Operating Systems Mechanisms

Security Policies I

Security Policies II

Verification Methods

Chairman: J.K. Millen, *MITRE Corp.*

Monday, April 22, 1985
9:00 am - 10:30 am

Structuring Systems for Formal Verification
R.B. Neely and J.W. Freeman

Trusted Software Verification: A Case Study
T.C.V. Benzel and D.A. Tavilla

Analysis of the Hardware Verification of the Honeywell SCOMP
V.D. Gligor

STRUCTURING SYSTEMS FOR FORMAL VERIFICATION

Richard B. Neely
(303) 594-1460
James W. Freeman
(303) 594-1536
Ford Aerospace and Communications Corporation
10440 State Highway 83
Colorado Springs, Colorado 80908

High levels of assurance for a secure system are obtained, in part, by the description of its trusted computing base in terms of a formal top-level specification. Nevertheless, the use of a single-level specification can result in an inability to link the behavior of the trusted computing base with the security policy of the system as a whole. This paper discusses some of the resulting problems and presents an approach to structuring systems that will support their verification. Such structuring is shown to be effective in bridging the gap between the trusted computing base itself and the system seen as a whole.

INTRODUCTION

It is generally accepted that formal methods can be used to increase the level of assurance that a system is secure. In spite of current improved understanding of such methods, concepts used in describing the trustworthiness of components retain the same limitations they had ten years ago. Only the most simple and monolithic of systems or components can be characterized by a single "top level specification" -- yet the attempt is still made to describe even entire operating systems by such means. Additionally, at the completion of the formal verification of a component, the relationship between the results of that verification and any increased understanding of the component and the overall system is usually not clear. Finally, the trustworthiness of an individual component is typically ensured only by assigning some sensitivity label to it and doing an analysis based on this assignment, rather than establishing its trustworthiness in relation to its individual constraints and requirements.

Some work has been accomplished in deriving security requirements from the environment of the component itself, and what the component actually does. Examples of such work include the development of the "separation kernel" as described by Rushby [10] and related work, the modeling approach described by Bartels and Dinolt [1], and various approaches being investigated within the University of Texas environment [4].

Further investigation and development of such concepts is needed. We introduce the notion of a "trust domain" to answer that need. A trust domain encapsulates a component in terms of "rule abstraction." This allows a characterization of the component in terms of expectations, both internal and external, of the component. The characterization is expressed at the component interface. Application of the trust domain concept promises reduction of proof complexity, better understanding of the formal specification and verification results, and explicit identification of underlying assumptions. If these goals are realized, then an increased level of assurance will follow.

The remainder of this paper is focused on providing the motivation, application, and exploitation of the trust domain concept. First, a statement of some problems to be solved is given. Then, the limitations of previous structuring attempts to solve the problems are recounted. This is followed by a detailed description of the concept of a trust domain together with an explanation of how the concept helps to solve the problems presented. An extended example of the use of trust domains is provided, including its embodiment in a trust domain representation language. Finally, we demonstrate the utility of the trust domain approach for formal verification in terms of the identified problems, and from this demonstration we draw several conclusions.

Problem Statement

Several problems related to the structuring of a system for verification are at least partially solved by the use of trust domains. These problems is described in this section.

Problem 1: Assessment of Verification Results. Consider a moderately complex computer system whose adherence in operation to a given security policy is critical. Suppose that the design, development, and documentation of the system follow the guidelines given in the DoD Computer Security Center's Trusted Computer System Evaluation Criteria [2]. One might now ask exactly

This research was sponsored in part by the USAF Rome Air Development Center under the Multinet Gateway Program, contract number F30602-81-C-0233.

what was verified, and further, how do the verification results really contribute to the confidence that the system will not violate the security policy. The answer can be formulated and expressed only in terms of the formal structure of the system, how that structure relates to the system environment, and how the proofs relate to the system structure. On one hand certainly, a small quantity of proof output (e.g., "TRUE" or "FALSE") is easy to understand, but one does not gain much confidence via such an understandable statement. On the other hand, a great deal of output, if it is not well-structured or does not relate cleanly to a well-structured system, is impossible to understand well enough to be sure what is proved.

At least three criteria need to be used in assessing the specification and verification output:

1. Proofs are small or at least understandable. The proofs are not just terse, but both complete and simple a possible.

2. The specification information and verification results are clearly related to the actual implemented system.

3. There are well-formed boundary conditions (environmental assumptions).

The latter two criteria are directly tied to the system's "architectural reference points" so one can more meaningfully discuss what is actualiy modeled, the underlying hardware constraints, assumptions, other non-proved components and code correspondence issues. By an architectural reference point we mean a significant aspect of a system architecture that is taken as given and so must be reflected by not only the system implementation, but also by the formal description of the system. For example, in a system of network gateways, an architectural reference point might include specific aspects of the geographical separation of the gateway nodes. Aspects of the ISO model layering of the protocols might form another example. The basic point about an "architectural reference point" is that it constrains the allowed design space of the system.

Problem 2: Verifying System Specific Characteristics. While the previous discussion was in terms of a "moderately complex" system, distributed systems typically possess an especially complex structure . A noteworthy class of examples is the class of communications systems. In an "ADP," or host, system, a case might be made to consider the interface of an operating system kernel as the exclusive subject of formal specification; but there is no analog in a communications environment. The software and hardware that implement network functions are structurally and conceptually far removed from the "system interface," i.e., the network system as seen from the "host users."

Problem 3: Ensuring Trustworthiness. Many systems, in enforcing a specific set of requirements, such as a given access control scheme based on sensitivity labels, determine properties and derived requirements that are not directly related to the labeling requirements. Sometimes the traceability between the original and derived requirements is weak because the derivation process hasn't been well documented. Second, some systems have well-defined security requirements that are not given via sensitivity labels. The result is a problem of truly understanding and agreeing on what trustworthiness means in an environment that may include but is not entirely dependent upon sensitivity labels.

Problem 4: Domain Reusability. In order to reduce the cost and, hopefully, technical risk of the specification and verification process, different components that are either identical in function, similar with only parametric differences, or significantly different while assuming or providing similar or identical interfaces -- such different components ought not to have to be specified and verified in a completely independent manner. Some means of reusing the formalism of these components is needed. This aspect has been discussed and an approach, based on a notion of reusable problem domain theories, has been suggested by Don Good [4]. Additional work needs to be accomplished in this area.

Previous and Current Structuring Attempts

Previous attempts to provide effective verification results have paved the way to the current state of the art. Although much good work has been accomplished, previous attempts, however, fall short of what we feel is possible now. It is instructive to see in which ways this latter statement is true, in terms of the three criteria presented in the discussion of the assessment of verification results. Although specific examples refer to the Kernelized Secure Operating System (KSOS), and by extension to Honeywell's SCOMP [11], they are relevant to a wider spectrum of contemporary projects.

Criterion 1: Proof Complexity. Often, the verification conditions (VCs) to be proved were linked to the formal specification in a way very difficult to trace because they were the result of much syntactic manipulation and other processing. Further, even though the most trivial of them were weeded out early, large numbers of proofs were still necessary. And yet, the true complexity of the results was much greater than it would seem from the amount of proof generated. A typical place of hiding complexity was in a special-purpose VC generator tool. Both of these observations were true, for example, for the KSOS formal verification [9]. The transition and translation steps from the SPECIAL language of HDM into VCs suitable for the Boyer-Moore theorem prover were rather large ones. The attempted examination of intermediate forms were to aid understanding of the translation process; in fact, they only added to the complexity to be digested. The Feiertag VC generator [3] is indeed a very complex program, and for the pur-

pose of analyzing complexity, must be included in the actual proofs to be examined.

In addition, the proof material was artificially small because it was not the full system that was involved. In KSOS, the kernel interface, along with relevant internal functionality, was the only part formally specified and verified. Yet what the user of KSOS depends on for enforcing security is the KSOS system as a whole, and so statements about the system as a whole is what needed proving. While the Unix emulator was not to be part of the trusted computing base, that fact needed to be a result of the specification and proof process, not an assumption of it. Further, the Non Kernel Security Related (NKSR) portion was part of the trusted computing base, yet was not part of any integrated proof process. Were the proof of KSOS, using the HDM-Feiertag technology, to be complete and integrated into a single FTLS as ordinarily conceived, the complexity of the proof would be increased by a large factor.

Criterion 2: True System Representation. Accurate representation of the target system also fell short in several ways. Security models were typically simple "flow-upward" models based on a lattice structure of security classifications. However, within the system (and sometimes visible to at least certain users), "exceptions" or special privileges had to be allowed for the sake of correct system operation. These never fit within the model, and so had to be either ignored in specification or else allowed to generate "spurious," unprovable VCs, which were then explained away informally. In addition, because of great disparity between the structure of the specification and the implemented code, the informal code correspondence argument in KSOS never became very convincing.

Criterion 3: Boundary Conditions. Finally, assumptions necessarily made about the system as a whole (but not provable) were dealt with quite informally, and in fact often were never mentioned but made tacitly. Such assumptions involved hardware and other entities with which the trusted computing base must interface and on which it depended, as well as the initial setup of file system data bases and even correct administrative procedures. By handling such issues in an ad hoc manner, more unprovable VCs were generated. Consequently the specification and verification results were confusing, lacked convincing power, and missed the opportunity to point out exactly where certain conditions had to be maintained externally for the system to remain secure.

TOWARDS A SOLUTION

Progress has been made in each of the identified four general problem areas in recent times. What is needed still is a conceptual framework to aid in organizing this type of specification and verification information. While much work remains to be accomplished in solving the problems described, we have seen initial applications of the trust domain concept to offer an effective step. The motivation for a trust domain idea is continued and provided next via a conceptual description. Having established the concept, how one actually describes such an entity is given by a description of the necessary structural and constraining relationships.

Trust Domain Description: Concept

A trust domain is characterized by the following list of attributes:

1. It is a part of a system (a component) with a well-defined functional boundary.

2. There are certain properties about its behavior that can be expected.

3. It is entitled to expect the validity of certain assertions about its environment.

4. It may have internal (non-externally visible) characteristics that are used to provide behavioral guarantees.

The "well-definedness" attribute is essential in producing the limitations on the scope of a trust domain construct to specify what is actually "trusted". A trust domain's trustworthiness is established either by assumptions that may or may not be proved (clearly identified), or else by proofs of assertions based on the environment of the trust domain. Such proved assertions are called resultant theorems of the trust domain.

Note that part of a trust domain's environment is typically some set of other trust domains. In that case, the guarantees of the other trust domains become part of the environmental assertions of the original trust domain. The assertions to be proved are termed "derived constraints" or resultant theorems. The original trust domain is then said to be constrained by each of the other trust domains. The relationship of each of the other trust domains with the original is spoken of as a constraint relationship. Other constraints external to a trust domain include assertions about the system's environment that must be taken as given in the development process.

The conceptual interface of two trust domains so related consists of functional abstraction (based on functional decomposition); data abstraction (based on private and shared abstract data types); and rule abstraction (based on constraints for function usage and interactions among functions and the data types they govern). A trust domain may be pictured as in Figure 1.

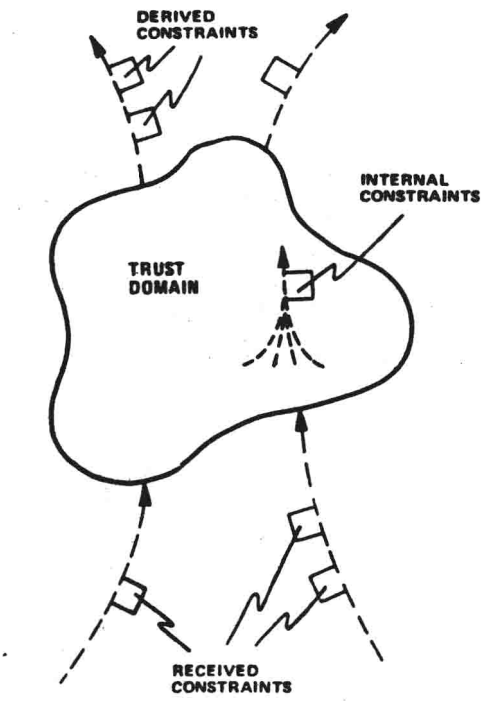


Figure 1. Trust Domain

Trust domains may have not only constraint relationships, but also containment relationships. Figure 2 provides an example of trust domains with constraint relationships; the figure also serves as a basis for illustrating containment relationships.

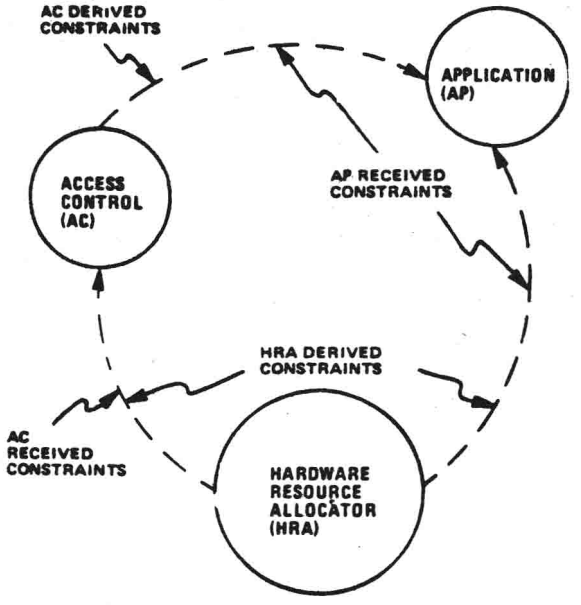


Figure 2. Trust Domain Constraint Concept

Figure 2 could be a picture of the major activities within a simple secure system. The system as a whole is a trust domain that contains the three trust domains shown. This must be so, since it is only about trust domains that properties can be proved, and it is the system as a whole that must be proved secure. Further, the individual trust domains of Figure 2 themselves may (usually at this level, will) contain interior trust domains.

It is important to note that no a priori distinction is made between "trusted" and "untrusted" components, though that distinction falls out of the interpretation of constraint rules. The idea is that external constraints (portrayed as externally applied assumptions) allow a limited scope of activity, so that all that must be proved about the trust domain itself is that, within that limited scope, it will not allow a violation of the constraint rules (its derived constraints) it is to enforce. Consequently, a trust domain is "trusted" to satisfy only the derived constraints. If a trust domain is placed so that no expectations need to be applied to its behavior (it has no derived constraints), it is "trusted" to satisfy no constraints. It is then said to be "untrusted". It is this usage in which the term "untrusted" is applied and simply means a trust domain with no resultant theorems or derived constraints. "Trusted", of course, means that derived constraints do exist. Note that this implies that "trusted" is not an absolute term, but is relative to the content of the derived constraints -- i.e., it is always in the sense of "trusted to obey what particular rules".

A key issue, then, in the application of such a concept is the facility to describe the various sets of rules or behavior properties of a given trust domain. The realization of trust domains and associated constraint rules depends on the use of certain established software design techniques, viz., functional abstraction and data type abstraction. A given formally described set of constraints or rules then is represented in terms of the visible functions and the abstract data types.

Trust Domain Description: Structure and Constraint

The trust domain notion has been described to this point as a structuring concept to of express desired properties and system structure. The previous discussion was to motivate the types of entities needed in a description of a trust domain. The explicit means by which a trust domain is described is outlined now so that specific examples can be described.

A "domain" is an entity with, of necessity, two types of relationships. The types of relationships are structural and constraint. A domain may "adjoin" another domain or "contain" or("inhabit") another domain. These two relationships, adjoin or contain, are structural. If two domains adjoin one another, they cannot contain one another. The identification of which

domains adjoin or contain other domains provides a topographical description of the system. A domain may "derive" or "receive" from another domain. These are constraint relationships. Figures 3 and 4 identify and illustrate how the two types of relationships are identified and described (denoted). The following paragraphs provide motivation for such a description.

Suppose domain A adjoins domain B and contains domain C (or C inhabits A). In order to represent actual systems and to describe communications among system entities (whether within a personal computer or among internets), domain are identified as either node or link. This identification is called the gender of the domain. Adjoined domains must be of opposite gender. Figure 3 identifies additional relationships among three domains that share adjoin and contain relationships.

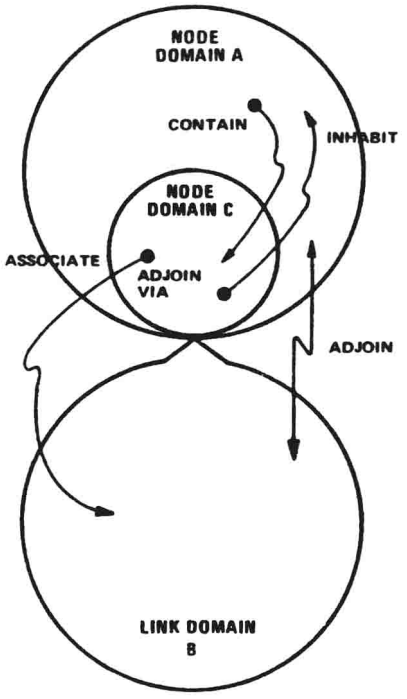


Figure 3. Trust Domain Structure Relationship

First, a domain that inhabits another domain is either a boundary or an interior domain with respect to the containing domain (C is interior to A and is, in fact, a boundary domain of A). Note also that domain A, containing domain C, "associates" C with domain B and C adjoins B "via" A. From A's point of view C is associated with B; from C's point of view C is adjoined to B via A; from B's point of view C is not present or in fact not visible. Thus "adjoin", "adjoin via" and "associates" are distinct structural relationships.

Constraint relationships are now incorporated with the structural relationships. Figure 4 identifies and illustrates this. Note that domain A "derives" constraints Y, Z for domain B, and "receives" constraint X from C; domain B "receives" constraints Y,Z from A and domain C "derives" constraint X for A. Note also that A receives constraint W from B if and only if B derives constraint W for A.

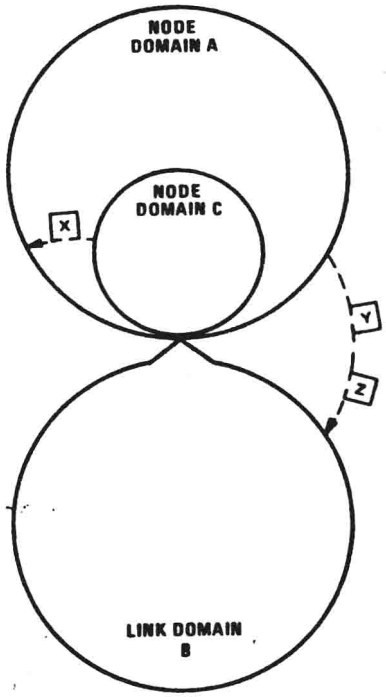


Figure 4. Trust Domain Constraint Relationship

The three domains can be described in terms of a trust domain description language as follows:

```
node domain A shall
    contain boundary node C;
    adjoin link B;
    associate C with B;
    derive for B
        constraint Y;
        constraint Z;
    receive from C
        constraint X;
end A;

node domain C shall
    inhabit boundary of node A;
    adjoin link B via A;
    derive for A
        constraint X;
end C;
```

```
link domain B shall
    adjoin node A;
    receive from A
        constraint Y;
        constraint Z;
end B;
```

It is useful to observe that although the arrows in Figure 3 illustrate the contains, adjoins and associates relationships, they are not integral to the structural description. Such arrows are redundant to the identified relationships. The dotted arrows together with the "boxes" identified in Figure 4 denote what constraints are levied between which domains. They are integral to the description.

Examples of domains using this description are given next.

EXAMPLE: STRUCTURING A SYSTEM

This section presents an example that illustrates the previously described aspects of a trust domain. The example is a network system fabricated for purposes of illustration. It includes multiple sites containing local area networks, and inter-site (presumably geographically extensive) transmission. Each site possesses a single mainframe processor operating in multilevel secure mode and multiple workstations. Figure 5 shows the topography of the system.

Note that all Processors and Workstations are considered to be "surface" system components, outside the system communications element (the Interconnect), and thus logically outside of any site. This choice of representation has been seen to provide a clear view of security-related constraints in several actual systems described in terms of trust domains. The level of detail of system representation depicted in Figure 5 might be typical of the initial structuring step performed to provide a security architecture.

Figure 6 describes a portion of the example system, introducing the constraint relationships. In fact Figure 6 is representative of a second step performed in building up a system security architecture. It is characterized by the analysis of the constraints relating the trust domains. The assignment of suggestive names to those constraints lays the groundwork for the addition of rigorous, mathematically oriented representations for each of the constraints.

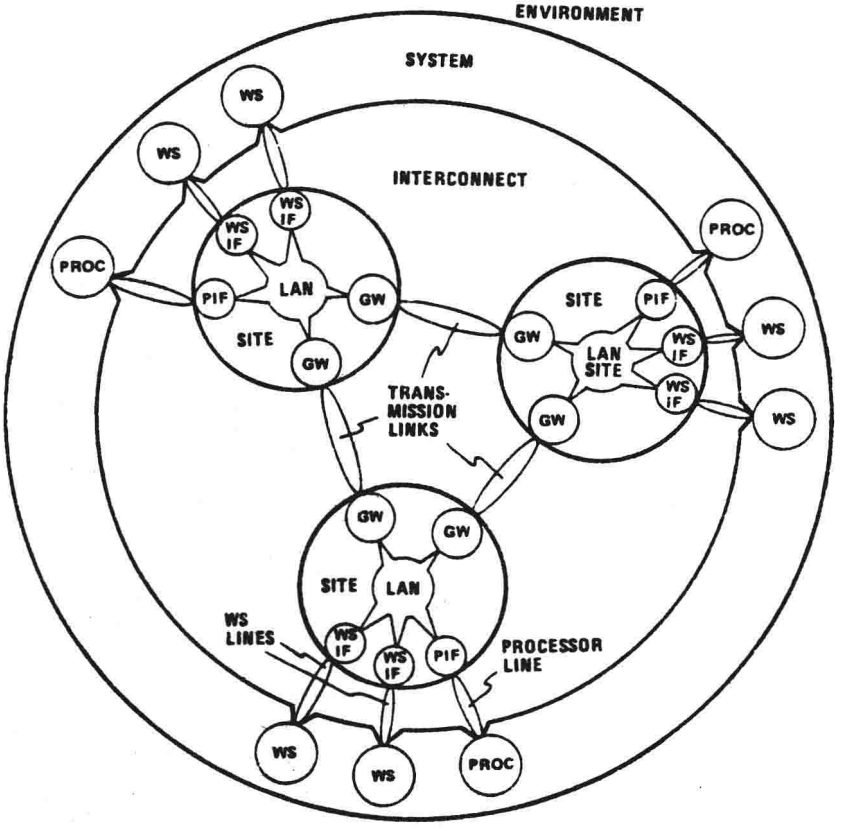


Figure 5. Example System Structure

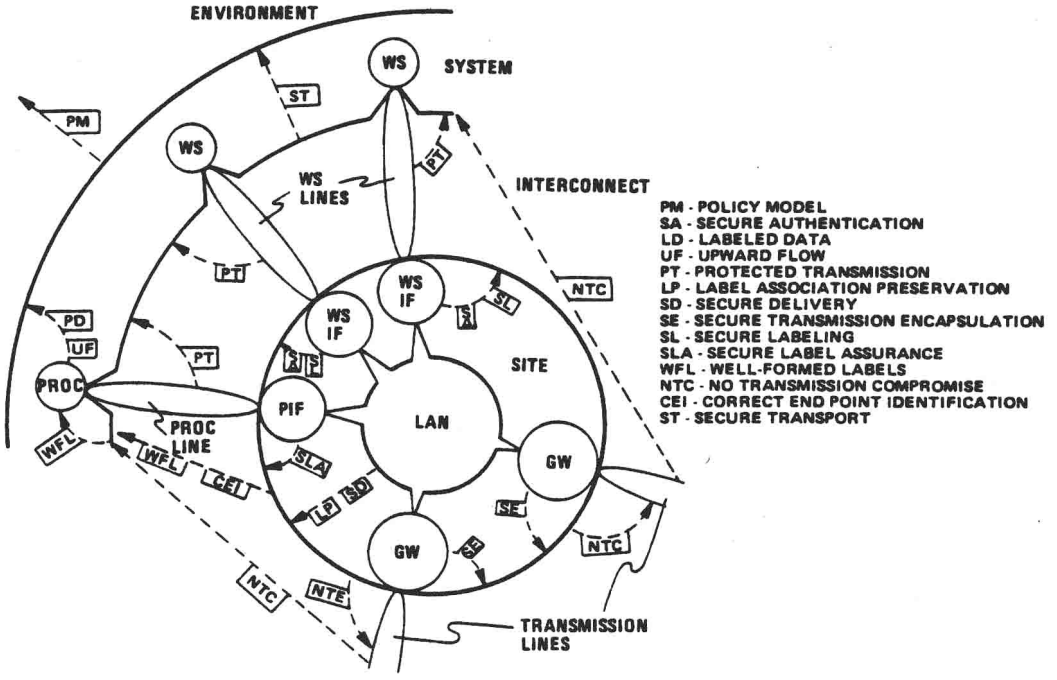


Figure 6. Example System Constraints

Several properties of the constraints in Figure 6 should be noted. A single named property is derived by the system "for the environment." This property is suggestively named "policy_model," which is the set of assertions comprising the model of the system security policy. Other constraints between domains build upon one another, finding their ultimate source in the descriptions of the components of the trusted computing base (and further in its implementation). If these source constraints are correct, and the proofs linking all the constraints are correct, and the trust domain structuring of the system faithfully represents the system's design, then the resultant constraint, the policy model, has been verified to be valid for the implemented system.

Trust Domain Description

This subsection contains the text of the description of the example system in terms of the trust domain description language introduced in section 2.

```
node domain ENVIRONMENT shall
    contain
        interior node System;
    receive from System
        constraint policy_model;
end ENVIRONMENT;
```

```
node domain System shall
    inhabit interior of node ENVIRONMENT;
    contain
        interior link Interconnect,
        interior node Processor multiple,
        interior node Workstation multiple;
    derive constraint policy_model;
    receive
        from Interconnect
            constraint secure_transport;
        from Processor
            constraint labeled_data;
            constraint upward_flow_only;
        from Workstation NO_CONSTRAINTS;
end System;

node domain Workstation shall
    inhabit interior of node System;
    adjoin link Interconnect;
    derive NO_CONSTRAINTS;
end Workstation;

node domain Processor shall
    inhabit interior of node System;
    adjoin link Interconnect;
    derive for System
        constraint labeled_data;
        constraint upward_flow_only;
end Processor;
```

```
link domain Interconnect shall
    inhabit interior of node System;
    contain
        boundary link Workstation_Line multiple,
        boundary link Processor_Line multiple,
        interior node Site multiple,
        interior link Transmission Link multiple;
    adjoin
        node Workstation,
        node Processor;
    associate
        Workstation_Line
            with Workstation one_to_one,
        Processor_Line with Processor one_to_one;
    derive
        for System constraint secure_transport;
        for Processor
            constraint well_formed_labels;
    receive
        from Site
            constraint well_formed labels;
            constraint correct endpoint ID;
        from Workstation_Line
            constraint protected_transmission;
        from Processor_Line
            constraint protected transmission;
        from Transmission link
            constraint no trans compromise;
end Interconnect;

link domain Workstation_Line shall
    inhabit boundary of link Interconnect;
    adjoin
        node Site,
        node Workstation via Interconnect;
    derive for Interconnect
        constraint protected_transmission;
end Workstation_Line;

link domain Processor_Line shall
    inhabit boundary of link Interconnect;
    adjoin
        node Site,
        node Processor via Interconnect;
    derive for Interconnect
        constraint protected_transmission;
end Processor_Line;


link domain Transmission_Link shall
    inhabit interior of link Interconnect;
    adjoin node Site multiple;
    derive for Interconnect
        constraint no_trans_compromise;
    receive from Site
        constraint no_trans_compromise;
end Transmission Link;
```

```
node domain Site shall
    inhabit interior of link Interconnect;
    contain
        boundary node Workstation_LAN IF,
        boundary node Processor_LAN IF,
        boundary node Gateway,
        interior link Local Area Network;
    adjoin
        link Workstation_Line multiple,
        link Processor_Line,
        link Transmission Link multiple;
    associate
        Workstation LAN IF
            with Workstation Line one to one,
        Processor LAN IF
            with Processor_Line one_to one,
        Gateway with Transmission_Link
                                one_to_one;
    derive
        for Interconnect
            constraint well_formed labels;
            constraint correct endpoint ID;
        for Transmission_Link
            constraint no_trans_compromise;
    receive
        from Local Area_Network
            constraint label_assoc_preservation;
            constraint secure_delivery;
        from Workstation LAN IF
            constraint secure_authentication;
            constraint secure_labeling;
        from Processor LAN IF
            constraint secure label assurance;
        from Gateway
            constraint
                secure_trans_encapsulation;
end Site;

node domain Workstation_LAN_IF shall
    inhabit boundary of node Site;
    adjoin
        link Local_Area_Network,
        link Workstation Line via Site;
    derive for Site
        constraint secure_authentication;
        constraint secure_labeling;
end Workstation_LAN IF;

node domain Processor LAN IF shall
    inhabit boundary of node Site;
    adjoin
        link Local_Area_Network,
        link Processor_Line via Site;
    derive for Site
        constraint secure_label assurance;
end Processor_LAN IF;

node domain Gateway shall
    inhabit boundary of node Site;
    adjoin
        link Local_Area_Network,
        link Transmission_Link via Site;
    derive for Site
        constraint
            secure_trans_encapsulation;
end Gateway;
```

```
link domain Local Area Network shall
    inhabit interior of node Site;
    adjoin
        node Workstation_LAN_IF multiple,
        node Processor_LAN_IF,
        node Gateway multiple;
    derive for Site
        constraint label_assoc_preservation;
        constraint secure_delivery;
end Local_Area_Network;
```

Selected Detailed Examples

The example in Figure 6 is complete in its identification of major structures and constraints in terms of trust domains. Two areas of expansion remain to be completed. The first is the elaboration of each of the constraints to include a complete mathematical representation of the constraint. Such representation will typically follow the standards of the specification language into which the trust domains are to be mapped. In general, this will involve expressions (e.g., predicate calculus), which are in fact prescribed by the existing trust domain language grammar. The second area is the description of the implementation requirements for the hardware and software interfaces so that they can be related to the rest of structure of the system, including the interface of the system itself. The following two subsections present a limited example of the identified areas.

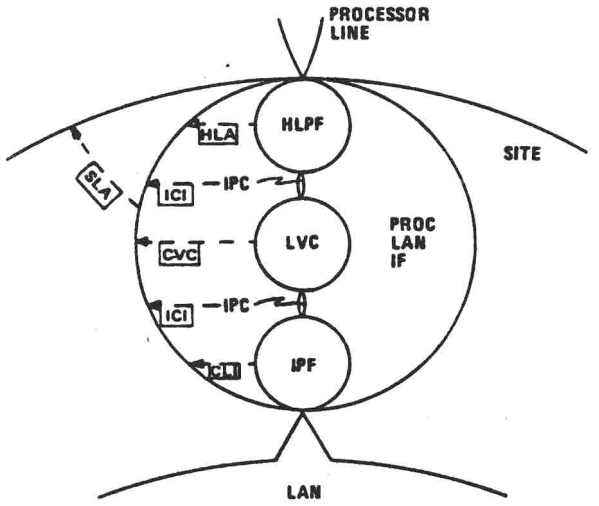
Constraint Elaboration The trust domain chosen for constraint elaboration is the Processor, depicted in Figure 6 with the notation "Proc." Following is the Processor trust domain description with constraints elaborated in terms of predicate calculus expressions.

```
node domain Processor shall
    inhabit interior of node System;
    adjoin link Interconnect;
    derive for System
        constraint labeled_data:
            FORALL datum,
                is_valid_label(class(datum));
        constraint upward_flow only:
            FORALL datum,proc,
                canread(proc,datum) =>
                    dominates(class(proc),
                              class(datum))
                AND canwrite(proc,datum) =>
                    dominates(class(datum),
                              class(proc));
    specify
        type element, proc subject,
             proc_object, label;
        variable datum: proc_object;
        variable proc: proc_subject;
        function is_valid_label(label):
                    boolean;
        function class(element): label;
        function
            canread(proc subject,proc object):
                        boolean;
        function dominates(label,label): boolean;
end Processor;
```

Note the "specify" clause added to the domain that allows truncated declarations of types, variables, and functions (also constants) to clarify the predicate calculus expressions. With the full form of the constraints, the suggestive names are retained; the two parts of each constraint thus complement each other.

Software-Related Example The example of this section has so far only been described in terms of its major system structures. This description will now be augmented by a selected example that relates the interface of a portion of the trusted computing base to the rest of the system structure.

The domain to be so augmented is the Processor LAN interface (Processor LAN IF). A pictorial description of that domain is given in Figure 7.



HLPF - HOST LABELING PROTOCOL FUNCTION
LVC - LABEL VALIDITY CHECKER
IPF - INTERNET PROTOCOL FUNCTION
IPC - INTER-PROCESS COMMUNICATION

HLA - HLP-LABEL-ASSOCIATION
CLI - CORRECT-LABEL-INSERTION
CVC - CORRECT-VALIDITY-CHECK
ICI - INTERPROC-COMM-INTEGRITY
SLA - SECURITY-LABEL-ASSURANCE

Figure 7. Software Domains of Processor_LAN_IF

It consists of three software domains, a Host Labeling Protocol Function, a Label Validity Checker, and an Internet Protocol Function. These domains communicate via a multiply instantiated Inter-Process Communication link domain. As with the previously presented portion of the example, these internal domains derive constraints that allow the proof of the constraint of Processor LAN IF derived for the Site (viz., secure label assurance).

This domain, as detailed in Figure 7, is now presented in terms of the trust domain description language.