ISI-TR-605

NPS-CS-05-010

Trustworthy Commodity Computation and Communication

SecureCore Technical Report

Design Principles for Security

Terry V. Benzel, Cynthia E. Irvine, Timothy E. Levin, Ganesha Bhaskara, Thuy D. Nguyen, and Paul C. Clark

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0430566 and CNS-0430598 with support from DARPA ATO. 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 National Science Foundation or of DARPA ATO.

Author Affiliations

Naval Postgraduate School:

Cynthia E. Irvine, Timothy E. Levin, Thuy D. Nguyen, and Paul C. Clark

Center for Information Systems Security Studies and Research
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

USC Information Sciences Institute:

Terry V. Benzel and Ganesha Bhaskara

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, Ca 90292

TABLE OF CONTENTS

I. INTRODUCTION 1

A. Definitions 2

B. Security Design Principles Overview 3

II. STRUCTURE 4

A. Economy and Elegance 4
Least Common Mechanism 4
Clear Abstractions 5
Partially Ordered Dependencies 5
Efficiently Mediated Access 6
Minimized Sharing 6
Reduced Complexity 7

B. Secure System Evolution 7

C. Trust 8
Trusted Components 8
Hierarchical Trust for Components 9
Inverse Modification Threshold 9
Hierarchical Protection 10
Minimized Security Elements 10
Least Privilege 10
Self-reliant Trustworthiness 11

D. Composition 11
Secure Distributed Composition 11
Trusted Communication Channels 12

III. LOGIC AND FUNCTION 12
Secure defaults 12
Secure Failure 13
Self Analysis 14
Accountability and Traceability 14
Continuous Protection of Information 15
Economic Security 15
Performance Security 16
Ergonomic Security 16
Acceptable Security 17

IV. SYSTEM LIFE CYCLE 17
Use Repeatable, Documented Procedures 17
Procedural Rigor 18
Secure System Modification 18
Sufficient User Documentation 18

V. COMMENTARY AND LESSONS LEARNED 19

Include Security in Design from the Start 19
The Philosopher's Stone 19
Other Approaches to Secure System Composition 20
The Reference Monitor 20
Conflicts in Design Principles 20

REFERENCES AND BIBLIOGRAPHY 21

I. Introduction

Security vulnerabilities are rampant throughout our information infrastructures. The majority of commodity computing and communication platforms have been designed to meet performance and functionality requirements with little attention to trustworthiness. The transformation of traditional stand-alone computers into highly networked, pervasive, and mobile computing systems profoundly increases the vulnerabilities of current systems, and exacerbates the need for more trustworthy computing and communications platforms.

While there is a significant history of secure systems design and development focusing on one or more of the triad of hardware, networking and operating systems, there are few worked examples [20]. To date, only special purpose systems begin to meet the requirements to counter either the modern or historical threats. In spite of over thirty years of research and development, a trustworthy product built at the commodity level remains elusive.

The SecureCore project is designing a secure integrated core for trustworthy operation of mobile computing devices consisting of: a security-aware processor, a small security kernel and a small set of essential secure communications protocols. The project is employing a clean slate approach to determine a minimal set of architectural features required for use in platforms exemplified by secure embedded systems and mobile computing devices.

In addition to security, other factors including performance, size, cost and energy consumption must all be reasonably accounted for when building a secure system. These factors are especially important for viability in the commodity market, where client computing devices have constrained resources but high performance requirements. Our goal is not security at any price, but appropriate levels of security that permit desirable levels of performance, cost, size and battery consumption.

As a prelude to our clean-slate design, we have reviewed the fundamental security principles from more than four decades of research and development in information security technology. As a result of advancing technology, we found that some of the early "principles" require re-examination. For example, previous worked examples of combinations of hardware, and software may have encountered problems of performance and extensibility, which may no longer exist in today's environment. Moore's law in combination with other advances has yielded better performance processors, memory and context switching mechanisms. Secure systems design approaches to networking and communication are beginning to emerge and new technologies in hardware-assisted trusted platform development and processor virtualization open hither to previously unavailable possibilities.

Our analysis of key principles for secure computing started with the landmark work of Saltzer and Schroeder [25] and surveyed the refinement of these principles as systems have evolved to the present. This report provides a distillation, synthesis and organization of key security systems design principles, describes each principle, and provides examples where needed for clarity. Although others have described various principles and techniques for the development of secure systems, e.g. [3], [9], [22], [24],

[25], [29], it was felt that a concise articulation of the principles as they are applied to the development of the most elemental components of a basic security system would be useful. In developing this report we have focused on the principles as they may be most applicable to SecureCore. A later report, "SecureCore Architecture and Requirements" [12], that uses these principles to define a high level architecture for SecureCore and a set of requirements, which will then be refined into a design specification. As a separate component of this work, a series of analysis reports will compare and contrast SecureCore to alternative modern information technology projects such as the TCG TPM [30] and various virtual machine based systems.

A common limitation of previous and ongoing efforts to articulate secure software development principles is the premise that "security vulnerabilities result from defects that are unintentionally introduced into the software during design and development" [6]. In contrast to those efforts and the software engineering "safety" paradigm upon which they rely, the articulation of design principles for SecureCore differs in two ways. First, our perspective not only acknowledges the risk of unintentional flaws, it explicitly assumes that unspecified functionality may be intentional. An adversary within the development process is assumed. Second, our analysis considers both the design of components as well as the composition of components to form a coherent security architecture that takes into account hardware, software and networking design elements.

The remainder of this section provides the definitions for commonly used terms, and an illustration of our overall taxonomy of security principles. Following this we present, in separate sections, the principles for: structure, logic and function, and system lifecycle. Finally, we end with some "lessons from the past," and identify some potential conflicts in the application of the described principles.

A. Definitions

Component: any part of a system that, by itself, provides all or a portion of the total functionality required of a system. A component is recursively defined to be an individual unit, not useful to further subdivide, or a collection of components up to and including the entire system. A component may be software, hardware, etc. For this report it is assumed that an atomic component – one not consisting of other components – may implement one or more different functions, but the degree of *trustworthiness* of the component is homogeneous across its functions.

A *system* is made up of one or more *components*, which may be linked (interact through the same processor), tightly coupled (e.g., share a bus), distributed (interact over a wire protocol), etc.

Failure: a condition in which, given a specifically documented input that conforms to specification, a component or system exhibits behavior that deviates from its specified behavior.

Module: a unit of computation that encapsulates a database and provides an interface for the initialization, modification, and retrieval of information from the database. The database may be either implicit, e.g. an algorithm, or explicit.

Process: a program in execution.

Reference Monitor Concept: an access control concept that refers to an abstract machine that mediates all accesses to objects by active entities. By definition, the ideal mechanism is protected from unauthorized modification and can be analyzed for correctness [2].

Security Mechanisms: system artifacts that are used to enforce system security policies.

Security Principles: guidelines or rules that when followed during system design will aid in making the system secure

Security Policies: Organizational Security Policies are "the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information." [28] System Security Policies are rules that the information system enforces relative to the resources under its control to reflect the organizational security policy. In this document, "security policy" will refer to the latter meaning, unless otherwise specified.

Service: processing or protection provided by a component to users or other components. E.g., communication service (TCP/IP), security service (encryption, firewall).

Trustworthy (noun): the degree to which the security behavior of the component is demonstrably compliant with its stated functionality *(e.g., trustworthy component).*

Trust: (verb) the degree to which the user or a component depends on the trustworthiness of another component. For example, component A *trusts* component B, or component B is *trusted by* component A. Trust and trustworthiness are assumed to be measured on the same scale.

B. Security Design Principles Overview

Security design principles can be organized into logical groups, which are illustrated in Figure 1. The logical groupings for the principles are in shaded boxes whereas the principles appear in clear boxes. For example, Least Privilege is a principle and appears grouped under Structure/Trust. In the case of "Secure System Evolution," the principle is in its own group.

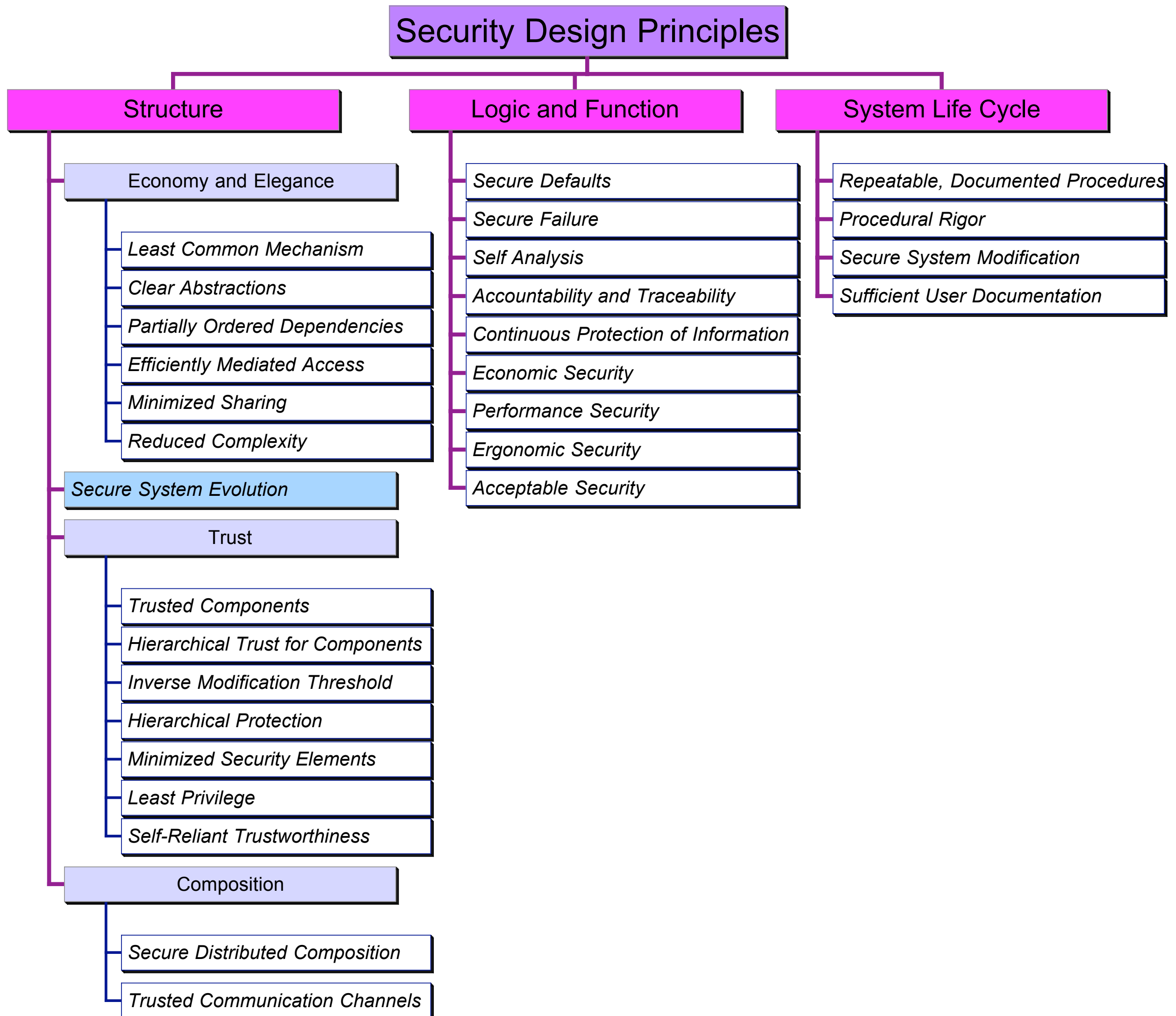

Figure 1. Taxonomy of security design principles

II. Structure

Structural design principles affect the fundamental architecture of the system: how the components relate to each other and the nature of their interfaces. We start with the fundamental need for economy and elegance in a system.

A. *Economy and Elegance*

Least Common Mechanism

The principle of least common mechanism states that if multiple components in the system require the same function or mechanism, then there should be a common mechanism that can be used by all of them. Thus, the various components do not have

separate implementations of the same function; rather, the function is created once. Examples of the application of this principle include device drivers, libraries, and operating system resource managers.

Using least common mechanism will help to minimize the complexity of the system by avoiding unnecessary duplicate mechanisms. Another benefit is maintainability, since modifications to the common function can be performed (only) once, and the impact of proposed modifications can be more easily understood in advance. Also, the use of common mechanisms will facilitate the construction and analysis of (1) non-by-passable system properties and (2) the encapsulation of data (see also "Minimized Sharing").

Consideration should be given to the problem of persistent state as it relates to a common mechanism. The common mechanism may need to retain state related to the context of the calling component. Whenever possible, the system should be organized to avoid this since: (1) retention of state information can result in significant increases in complexity, and (2) can result in state that is shared by multiple components (see "Minimized Sharing"). Sometimes various forms of linking can permit a common mechanism to utilize state information specific to the calling component, and, with sufficient low-level support, the mechanism can even assume the privilege attributes of its calling component. [10]

Clear Abstractions

The principle of clear abstractions states that a system should have simple, well-defined interfaces that clearly represent the data and functions provided. The elegance (e.g., clarity, simplicity, necessity, sufficiency) of the system interfaces, combined with a precise definition of their behavior promotes thorough analysis, inspection and testing as well as correct and secure use of the system. Clarity of abstractions is difficult to quantify and a description will not be attempted here. Some of the techniques used to create simple interface are: the avoidance of redundant entry points, the avoidance of overloading the semantics of entry points and of the parameters used at entry points, and the elimination of unused entry points to components.

Information hiding [23] is a design discipline for ensuring that the internal representation of information does not unnecessarily perturb the correct abstract representation of that data at an interface (see also Secure System Evolution).

Partially Ordered Dependencies

In applying the principle of least common mechanism, if the shared mechanism also makes calls to or otherwise depends on services of the calling mechanisms, creating a circular dependency, performance and liveness problems can result. The principle of partially ordered dependencies says that the calling, synchronization and other dependencies in the system should be partially ordered.

A fundamental tool in system design is that of layering [8], whereby the system is organized into functionally related modules or components, and where the layers are linearly ordered with respect to inter-layer dependencies. While a partial ordering of all functions in a given system may not be possible, if circular dependencies are constrained to occur within layers, the inherent problems of circularity can be more easily managed

[26].

Partially ordered dependencies and system layering contribute significantly to the simplicity and coherency of the system design (see also "Assurance through Reduced Complexity").

Efficiently Mediated Access

The mediation of access to resources is often the predominant security function of security systems, which can result in performance bottlenecks if the system is not designed correctly. The principle of efficiently mediated access [1] states that the access control mechanism for each subset of the policy should be performed by the most efficient system mechanism available while respecting layering and still meeting system flexibility requirements. A good example of this is the use of hardware memory management mechanisms to implement various access control functions, e.g. [10], [27].

Minimized Sharing

The principle of minimized sharing states that no computer resource should be shared between components or subjects (e.g., processes, functions, etc.) unless it is necessary to do so. Minimized sharing helps to simplify the design and implementation. It is evident that in order to protect user-domain information from active entities, no information should be shared unless that sharing has been explicitly requested and granted (see also "Secure Defaults"). For internal entities, sharing can be motivated by the principle of least common mechanism, as well as to support user-domain sharing. However, internal sharing must be carefully designed to avoid performance and covert channel problems [17]. There are various mechanisms to avoid sharing and mitigate the problems with internal sharing.

To minimize the sharing induced by common mechanisms, they can be designed to be *re-entrant* or *virtualized*, so that each component depending on that mechanism will have a virtual private data space. Virtualization logically partitions the resource into discrete, private subsets for each dependent component. The shared resource is not directly accessible by the dependent components. Instead an interface is created that provides access to the private resource subsets. Practically any resource can be virtualized, including the processor, memory and devices. Encapsulation is a design discipline or compiler feature for ensuring there are no extraneous execution paths for accessing the private subsets (see also "information hiding," under Secure System Evolution). Some systems use global data to share information among components. A problem with this approach is that it may be difficult to determine how the information is being managed [31]. Even though the original designer may have intended that only one component perform updates on the information, the lack of encapsulation allows any component to do so.

To avoid covert timing channels, in which the processor is one of the shared components, a scheduling algorithm can ensure that each depending component is allocated a fixed amount of time [11]. A development technique for controlled sharing is to require the execution durations of shared mechanisms (or the mechanisms and data structures that determine its duration), to be explicitly stated in the design specification, so that the effects of sharing can be verified.

Reduced Complexity

Given the current state of the art, a conservative assumption must be that every complex system will contain vulnerabilities, and it will be impossible to eliminate all of them, even in the most highly trustworthy of systems. Application of the principle of reduced complexity contributes to the ability to understand the correctness and completeness of system security functions, and facilitates identification of potential vulnerabilities. The corollary of reduced complexity states that the simpler a system is, the fewer vulnerabilities it will have. An example of this is a bank auto teller, which, due to the simplicity of its interface (a very limited set of requests), has relatively few functional security vulnerabilities compared to many other widely used security mechanisms.

From the perspective of security, the benefit to this simplicity is that it is easier to understand whether the intended security policy has been captured in the system design. For example, at the security model level, it can be easier to determine whether the initial system state is secure and whether subsequent state changes preserve the system security properties.

B. Secure System Evolution

The principle of secure system evolution states that a system should be built to facilitate the maintenance of its security properties in the face of changes to its interface, functionality structure or configuration. These changes may include upgrades to the system, maintenance activities, reconfiguration, etc. (see also, Secure System Modification, and Secure Failures). The benefits of this principle include reduced lifecycle costs for the vendor, reduced cost of ownership for the user, as well as improved system security. Just as it is easier to build trustworthiness into a system from the outset (and for highly trustworthy systems, impossible to achieve without doing so), it is easier to plan for change than to be surprised by it.

Although it is not possible to plan for every possibility, most systems can anticipate maintenance, upgrades, and changes to their configurations. For example, a component may implement a computationally intensive algorithm. If a more efficient approach to solving the problem emerges, then if the component is constructed using the precepts of modularity and information hiding [14], [15], [23], it will be easier to replace the algorithm without disrupting the rest of the system.

Rather than constructing the system with a fixed set of operating parameters, or requiring a recompilation of the system to change its configuration, startup or runtime interfaces can provide for reconfiguration. In the latter case, the system designer needs to take into account the impact dynamic reconfiguration will have on secure state.

Interoperability can be supported by encapsulation at the macro level: internal details are hidden and standard interfaces and protocols are used. For scalability, the system can be designed so that it may easily accommodate more network connections, more or faster processors, or additional devices. A measure of availability can be planned into the system by replication of services and mechanisms to manage an increase in demand, or a failure of components.

Constructing a system for evolution is not without limits. To expect that complex systems will remain secure in contexts not envisioned during development, whether

environmental or related to usage, is unrealistic. It is possible that a system may be secure in some new contexts, but there is no guarantee that its "emergent" behavior will always be secure.

C. *Trust*

Trusted Components

The principle of trusted components states that a component must be *trustworthy* to at least a level commensurate with the security dependencies it supports (i.e., how much it is trusted to perform its security functions by other components). This principle enables the composition of components such that trustworthiness is not inadvertently diminished and consequently, where trust is not misplaced.

Ultimately this principle demands some metric by which the trust in a component and the trustworthiness of a component can be measured; we assume these measurements are on the same, abstract, scale. This principle is particularly relevant when considering systems and those in which there are complex "chains" of trust dependencies.

A *compound component* consists of several subcomponents, which may have varying levels of trustworthiness. The conservative assumption is that the overall trustworthiness of a compound component is that of its least trustworthy subcomponent. It may be possible to provide a security engineering rationale that the trustworthiness of a particular compound component is greater than the conservative assumption, but a general analysis to support such a rationale is outside of the scope of this report.

This principle is stated more formally:

Basic types

component

t: integer /* level of trust or trustworthiness – this is cast as integer for convenience - any linear ordering will do */

System constant functions and their axioms

subcomponent(a, b:component): boolean /* a is a subcomponent of b */

depend(a, b: component): boolean /* a depends on b */

sec_depend(a, b: component): boolean /* a has security dependency on b */

axiom 1.. ∀ a, b: component(

sec_depend(a, b)

⇒ depend(a, b)) /* but not visa versa */

trust(a, b: component): t /* the degree of sec_depend */

axiom 2.. ∀ a, b:component(

$$
\begin{aligned}
&\quad\quad sec_depend(a, b) \\
&\quad\quad\quad \Rightarrow trust(a,b) > 0)
\end{aligned}
$$

trustworthy(a: component): t /* a is trustworthy to the degree of t */

$$
\begin{aligned}
&axiom\ 3..\ \forall\ a, b{:}component(\quad /*\ sub\text{-}component\ trustworthiness\ */ \\
&\quad subcomponent(a,b) \Rightarrow \\
&\quad\quad trustworthy(b) \leq trustworthy(a))
\end{aligned}
$$

Principle of trusted components.

$$
\begin{aligned}
&\forall\ a{:}component(\\
&\quad \exists\ b{:}component\ (sec_depend(a,b) \Rightarrow \\
&\quad\quad trust(a,b) \leq trustworthy(b)))
\end{aligned}
$$

Hierarchical Trust for Components

The corollary of hierarchical trust for components states that the security dependencies in a system will form a partial ordering if they preserve the principle of trusted components. To be able to analyze a system comprised of heterogeneously trustworthy components for its overall trustworthiness, it is essential to eliminate circular dependencies with regard to trustworthiness. Clearly, if a more trustworthy component located in a lower layer of the system were to depend upon a less trustworthy component in a higher layer, this would, in effect, put them in the same equivalence class: less trustworthy.

Trust chains have various manifestations. For example, the root certificate of a certificate hierarchy is the most trusted node in the hierarchy, whereas the leaves may be the least trustworthy nodes in the hierarchy. Another example occurs in a layered high assurance secure system where the security kernel (including the hardware base), which is located at the lowest layer of the system, is the most trustworthy component.

This principle does not prohibit the use of overly trustworthy components. For example, in a low-trust system the designer may choose to use a highly trustworthy component, rather than one that is less trustworthy because of availability or other criteria (e.g., an open source based product might be preferred). In this case, the dependency of the highly trustworthy component upon a less trustworthy component does not degrade the overall trustworthiness of the resulting system.

Inverse Modification Threshold

The corollary of inverse modification threshold states that the degree of protection provided to a component must be commensurate with its trustworthiness. In other words, as the criticality of (i.e., trust in) a component increases, the protections against its unauthorized modification should also increase. This protection can come in the form of the component's own self-protection and trustworthiness, or from protections afforded to the component from other elements or attributes of the architecture. Unauthorized modification could take place through penetration of the component (e.g., an attack that

bypasses the intended interfaces), misuse of poorly designed interfaces, or from surreptitiously placed trapdoors.

Techniques to show the absence of trapdoors and penetration vulnerabilities can be applied to the construction of highly trustworthy components. Examples of the application of this principle can be seen in the hardware, microcode, and low level software of trustworthy systems: none of these elements is easy to modify.

Hierarchical Protection

The principle of hierarchical protection states that a component need not be protected from more trustworthy components. In the degenerate case of the most trusted component, it must protect itself from all other components. In another example, a trusted computer system need not protect itself from an equally trustworthy user, reflecting use of untrusted systems in "system high" environments where the users are highly trustworthy.

Minimized Security Elements

The principle of minimized security elements states that the system should not have extraneous trusted components. This principle has two aspects: cost and complexity of security analysis. Trusted components, necessarily being trustworthy, are generally more costly to construct, owing to increased rigor of development processes (see "Procedural Rigor"). They also require greater security analysis, to qualify their trustworthiness. Thus, to reduce cost, and decrease the complexity of the security analysis, a system should contain as few trustworthy components as possible.

The analysis of the interaction of trusted components with other components of the system is one of the most important aspects of the verification of system security. If these interactions are unnecessarily complex, the security of the system will also be more difficult to ascertain than one whose internal trust relations are simple and elegantly constructed. Generally, fewer trusted components will result in fewer internal trust relationships and a simpler system. For example, a novice multilevel secure system designer may be tempted to solve every security problem with one or more "trusted subjects," creating a system that is unnecessarily complex.

Least Privilege

The principle of least privilege states that each component should be allocated sufficient privileges to accomplish its specified functions, but no more. This limits the scope of the component's actions, which has two desirable effects: (1) security impact of a failure of corruption of the component will be minimized, and (2) the security analysis of the component will be simplified. The result is a safer and more understandable system.

Least privilege is such a pervasive principle that it is reflected in all aspects of the system. For example, interfaces may be constructed that are available to only certain subsets of the user population. In the case of an audit mechanism, there may be an interface for the audit manager, who configures the audit settings; an interface for the audit operator, who ensures that audit data is safely collected and stored; and, finally, yet another interface for the audit reviewer.

In addition to its manifestations at the system interface, least privilege can be used as a guiding principle for the internal structure of the system itself. This can take several forms. Closely aligned with the notions of modularity and encapsulation [23], one aspect of internal least privilege is to construct modules so that only the elements encapsulated by the module are directly operated upon [31]. Elements external to a module that may be affected by the module's operation are indirectly accessed through interaction (e.g., via a function call) with the module that contains those elements. Another aspect of internal least privilege is that the scope of a given module or component should only include those system elements that are necessary for its functionality, and that the modes by which the elements are accessed should also be minimal.

Self-reliant Trustworthiness

The principle of self-reliant trustworthiness states that systems should minimize their reliance on external components for system trustworthiness. If a system were required to maintain a connection with another external entity in order to maintain its trustworthiness, then that system would be vulnerable to drops in the connection. Instead, a system should be trustworthy by default with the external connection used as a supplement to its function.

The benefit to this principle is that the isolation of a system will make it less vulnerable to attack. Clearly, if this were not the case, then attack scenarios would be devised to isolate the system and thus bring down its defenses. In a highly networked environment, this would be a problem for the targeted system, but also perhaps calamitous for other systems on the network.

A corollary to this relates to the ability of the component to operate in isolation and then resynchronize with other components when it is rejoined with them (see the principle of secure failures).

D. Composition

Secure Distributed Composition

Many of the design principles for secure systems deal with how components can or should interact (e.g., see Hierarchical Trust). The composition of *distributed* components can magnify the relevancy of these principles. In particular, the translation of security policy from a stand-alone to a distributed system can have unexpected "emergent" results (see also, Secure System Evolution in II.B). The principle of secure distributed composition states that the composition of distributed components that enforce the same security policy should result in a system that enforces that policy at least as well as the individual components do. For example, consider a set of components that support similar subjects and objects [16] and enforce the same access control policy on those objects. Under this principle, if the components are composed into a distributed system that supports the same policy, and information contained in objects is transmitted between components, then the transmitted information must be at least as well protected in the receiving component as it was in the sending component. Communication protocols and various distributed data consistency mechanisms can help to ensure consistent policy enforcement across a distributed system.

In another example, consider a distributed system in which all subjects and objects are associated with labels from a sensitivity hierarchy of Top Secret (TS), Secret (S), Confidential (C), Unclassified (U), and the usual multilevel mandatory access control policy is enforced. Also, since the system components are not very trustworthy, each component may only present to users information with "adjacent" sensitivity (e.g., component **A** handles TS and S, and component **B** handles S and C), so that in the event of leakage, the information does not leak too "far." The networking (i.e., composition) of components **A** and **B** can result in a "cascade" of sensitivities such that if **A** leaks TS information to an S user, TS information can wind up on **B,** presenting users with a wider range of sensitivities than the system policy allows [20].

Thus, to ensure correct system-wide level of confidence of correct policy enforcement, enforcement, the security architecture of a distributed composite system must be thoroughly analyzed.

Trusted Communication Channels

The principle of secure communication channels states that when composing a system where there is a threat to the communication between components, each communications channels must be trustworthy to a level commensurate with the security dependencies it supports (i.e., how much it is trusted to perform its security functions by other components). Several techniques can be used to mitigate threats, and enhance the trustworthiness of communication channels. Three are discussed here.

First, use of the channel may be restricted by protecting access to it with a suitable access control mechanism such as a reference monitor located beneath or within each component. By controlling and limiting access to the channel, possible misuse of the channel can be reduced. In addition, the components with authorized access to the channel may be more trustworthy than other components.

Second, end-to-end communications technologies, such as encryption, may be used to eliminate security threats in the channel's physical environment. In some cases, an alternative to platform-based encryption is the use of in-line encryption devices. When such devices are employed, they must be at least as trustworthy as the reference monitors of the linked components.

Finally, intrinsic characteristics assumed for and provided by the channel must be specified. With such documentation, it is possible for system designers to understand the nature of the channel as initially constructed and to assess the impact of any subsequent changes to the system.

III. Logic and function

The principles associated with logic and function are applicable at both the system and component level.

Secure defaults

The principle of secure defaults applies to the initial configuration of a system as well as to the negative nature of access control and other security functions. First, the "as

shipped" configuration of a system or component should not aid in violation of security policy. There have been many examples in recent years of commercial systems that have arrived in a configuration that was not adequately self-protective, resulting in security breaches before the correct configuration could be established. Some examples of mechanisms for which secure initial configuration may apply are audit, firewalls and passwords.

The second part of this principle says that security mechanisms should deny requests (e.g., to obtain access to a file) unless the request is found to be well formed and consistent with the security policy. The alternative is to allow a request unless it is shown to be inconsistent with the policy. In a large system, the conditions that must be satisfied to grant a request that is by default denied are often far more compact and complete than those that would need to be processed to deny a request that is by default granted (for example, consider the filtering rules in a firewall).

Secure Failure

The principle of secure failure states that a failure in a system function or mechanism should not lead to violation of security policy. Failure is a condition in which a component's behavior deviates from its specified behavior for an explicitly documented input (*unspecified behavior*, which includes response to inputs that do not conform to specification, is addressed in IV). Ideally, the system should be capable of detecting failure at any stage of operation (initialization, normal operation, shutdown, maintenance, error detection and recovery) and take appropriate steps to ensure security policies are not violated.

Once a failed security function is detected, the system may reconfigure itself to circumvent the failed component, while maintaining security, and still provide all or part of the functionality of the original system, or completely shut itself down to prevent any (further) violation in security policies. For this to occur, the reconfiguration functions of the system should be designed to ensure continuous enforcement of security policy during the various phases of reconfiguration. Another mechanism that can be used to recover from failures is to rollback to a secure state (which may be the initial state) and then either shutdown or replace the service or component that failed with orthogonal or replicated mechanisms.

Failure of a component may or may not be detectable to the components using it. This principle indicates that components should fail in a state that denies rather than grants access. For example, a nominally "atomic" operation interrupted before completion should not break security policy and hence must be designed to cope with interruption events by employing higher level atomicity and rollback mechanisms such as transactions, etc. If a service is being used, its atomicity properties must be well documented and characterized so that the component availing itself of that service can detect and handle interruption events appropriately. For example, a system should be designed to gracefully respond to disconnection and support resynchronization and data consistency after disconnection.

Replication of policy enforcement mechanisms, sometimes called "defense in depth," can allow the system to continue securely even when one mechanism has failed to protect the

system. If the mechanisms are similar, however, the additional protection may be illusory, as the intruder simply repeats the same or similarly difficult attacks on each mechanism. Similarly, in a networked system, breaking the security on one system or service may enable an attacker to do the same on other similar replicated systems and services. By employing multiple protection mechanisms, whose features are significantly different, the possibility of attack repetition can be reduced. However, it should be noted redundancy techniques may increase resource usage and may adversely affect the system performance.

Self Analysis

The principle of self-analysis states that the component must be able to assess its internal functionality to a limited extent (within the limits of the "incompleteness theorem") at various stages of execution and that this self-analysis capability must be commensurate with the level of trustworthiness invested in the system.

At the system level, self-analysis can be achieved via hierarchical trustworthiness assessments established in a bottom up fashion. In this approach [4], the lower level components check for data integrity and correct functionality (to a limited extent) of higher level components. For example, trusted boot sequences involve a lower level component attesting to the trustworthiness of the next higher-level components so that a transitive trust chain can be established. At the root, a component attests to itself, which usually involves an axiomatic or environmentally enforced assumption about its integrity.

These tests can be used to guard against externally induced errors or internal malfunction or transient errors. By following this principle, some simple errors or malfunctions can be detected without allowing the effects of the error or malfunction to propagate outside the component. Further, the self test can also be used to attest to the configuration of the component, detecting any potential conflicts in configuration with respect the expected configuration.

Accountability and Traceability

The principle of accountability and traceability states that actions that are security-relevant must be traceable to the entity on whose behalf the action is being taken.

This principle requires the designer to put into place a trustworthy infrastructure that can record details about actions that affect system security (e.g., an audit subsystem). To do this, the system must not only be able to uniquely identify the entity on whose behalf the action is being carried out, but also record the relevant sequence of actions that are carried out. Further, the accountability policy ought to require the audit trail itself be protected from unauthorized access and modification. The principle of least privilege aids in tracing the actions to particular entities, as it increases the granularity of accountability. Associating actions with system entities, and ultimately with users, and making the audit trail secure against unauthorized access and modifications provide non-repudiation, as once some action is recorded, it is not possible to change the audit trail.

Another important function that traceability and accountability serves is in the analysis of events leading to violation of security policy. If a security violation occurs, analysis of the audit log may provide additional information that may be helpful in determining the

path or component that allowed the violation of security policy.

Continuous Protection of Information

Principle of continuous protection of information states that information protection required by the security policy (e.g., access control to user-domain objects) or for system self-protection (e.g., maintaining integrity of kernel code and data) must be protected to a level of continuity consistent with the security policy and the security architecture assumptions. No guarantees about information integrity, confidentiality or privacy can be made if it is left unprotected while under control of the system (i.e., during the creation, storage, processing or communication of the information and during system initialization, execution, failure, interruption, and shutdown). Following the precepts of the *reference monitor* [2], to provide continuous enforcement of the security policy, every request must be validated, and the reference monitor must protect itself. Invalid requests should not result in a system state such that the system cannot properly enforce the security policy. The principle of secure failure also applies here in that it involves a roll back mechanism that can return the system to a secure state.

To ensure protection, parameters at interfaces must be chosen so that security critical values are provided by more trustworthy components. In addition, at the interface to a security mechanism, the security-relevant operation should appear atomic. This will eliminate time-of-check-to-time-of-use vulnerabilities.

Cryptography is one of the primary mechanisms used to protect information in transit and in storage.

To protect information during computation various low-level hardware mechanisms can be used such as instructions that enable switching contexts atomically and mechanisms that enable memory protection.

In some environments, it is desirable to allow the system security policies to be "modifiable" at runtime, for example to adjust to catastrophic external events. Changes to policies must not only be traceable but also verifiable, i.e., must be possible to verify that the changes do not violate security policies. The system architect should understand the consequences of allowing modifiable policies in a system i.e., depending on the type of access control and the actions that are allowed and controlled by the policies, certain configuration changes may lead to inconsistent states or discontinuous protection due to the complex or undecidable nature of the problem. One approach to this problem is the use of pre-verified configuration definitions where the transition from old to new policies is effectively atomic and any residual effects from the old policy are guaranteed to not conflict with the new policy.

Economic Security

The principle of economic security states that security mechanisms whose strength is commensurate with the level of trustworthiness should be used in components that enforce security policies.

Mechanisms involved in the enforcement of security policies incur computation and resource overhead. The strength of the mechanisms must be sufficient to satisfy the system requirements. Using security mechanisms of greater strength than necessary may

unnecessarily incurs extra overhead. For example, if a 2048 bit key may be sufficient to satisfy the requirements for policy enforcement, a 4096 bit key, though relatively more secure, only incurs more computation overhead unnecessarily.

Another prudent design guideline is that the security mechanisms used to protect an entity should be no more expensive than the entity itself, i.e., the security mechanism should not be more costly than the expected damage of a security breach. However, designers should remember that when trust chains are involved, the overall trustworthiness requirement determines the strength of mechanism needed, as the system is no more trustworthy than the least trustworthy mechanism in the chain.

Mechanisms that aid in enforcement of security may utilize many basic building blocks. Cryptography is often one of many mechanistic choices. Because it is computationally intensive, it has to be used sparingly and only when other, simpler mechanisms are insufficient. Even when encryption is used, the strength of encryption must be selected appropriately after considering key storage, key exchange, exposed attack space etc.

Application of this principle reinforces the requirement to design the system from the ground up, i.e., to incorporate simple mechanisms at the lower layers that can be used as building blocks for higher level mechanisms.

Performance Security

The principle of performance security states that security mechanisms should be constructed so that they do not degrade system performance unnecessarily. A corollary to this principle is that the demand for performance and functionality should not blind the system designer to system security requirements. For example, in a banking system, an animated user interface may not be as critical as protecting user's financial assets.

There is often a tradeoff between functionality and performance. The more functionality a system has, the more generic its components must be, and hence the system will be less optimized to perform specific functions. Similarly, there may be a tradeoff between performance and enforcement of security policy. On one extreme, complete elimination of all security checks may provide slight performance improvements. However, if enforcement of security policy is one of the intended goals for the system, the system should be designed such that enforcement of security policy does not significantly degrade system performance. This forces the designer to incorporate mechanisms that aid in enforcement of security policy, but incur minimum overhead, such as low-level atomic mechanisms on which higher level mechanisms can be built. Such low level mechanisms are usually very specific, have very limited functionality, and are heavily optimized for performance. For example, once access rights to a portion of memory is granted, hardware mechanisms may be used to ensure that all further accesses involve the correct memory address and access mode [10].

Ergonomic Security

The principle of ergonomic security states that the user interface for security functions and supporting services should be intuitive and user friendly, and provide appropriate feedback for user actions that affect policy and its enforcement. The mechanisms that enforce security policy should not be intrusive to the user and should be designed not to

degrade user efficiency.

There is often a tradeoff between usability and the strictness of policy enforcement. For example, if every action that requires authorization required the user to prove their identity explicitly, it would be intrusive, annoying, and could potentially make the system unusable. To overcome this deficiency, the designer may decide to cache credentials for an appropriate duration and use them to enforce policy. However, caching credentials adds complexity to the system and their use creates new vulnerabilities.

The system should provide the user with feedback and warnings when insecure choices are being made. For example, web interfaces often warn users that information may not be protected during transmission. In such cases, a user may reconsider entry of personal information. The designer must seek a solution that both satisfies the security policy requirements and makes the system easy and efficient to use.

Care must also be given to interfaces through which system administrators configure and setup the security policies. Ideally, system administrators must be able to understand the impact of their choices. They must be able to configure systems before startup and administer them during runtime, in both cases with confidence that their intent is correctly mapped to the system's mechanisms.

Acceptable Security

The principle of acceptable security requires that the level of privacy and performance the system provides should be consistent with the users' expectations. The perception of personal privacy may affect user behavior, morale and effectiveness. Based on the organizational privacy policy and the system design, users should able to restrict their actions to protect their privacy. When systems fail to provide intuitive interfaces or meet privacy and performance expectations, users may either choose to completely avoid the system or use it in ways that may be inefficient or even insecure.

Thus ergonomic and acceptable security, coupled with user education, are essential.

IV. System life cycle

Several principles guide the system life cycle to contribute to the initial and ongoing security of the system.

Use Repeatable, Documented Procedures

The principle of repeatable and documented procedures means that the techniques used to construct a component should permit the same component to be completely and correctly reconstructed at a later time. Repeatable and documented procedures support the creation of a component that are identical to the component created earlier that may be in widespread use. In the case of other system artifacts (such as documentation and testing results), repeatability supports consistency and inspectability.

A procedure can range from a script to compilable code, to steps taken for the reporting and remediation of system deficiencies. Procedures may be formalized and can be based upon standards. For example, the Common Criteria [13] provide a framework for the derivtion of system requirements that is comprised of the following steps: definition of

system goals and its concept of operation; identification of threats to the defined system; identification of assumptions regarding the system and its environment; identification of organizational policies external to the system; identification of security objectives for the system and its environment based on previous steps; and specification of requirements that will meet the objectives.

Procedural Rigor

The principle of procedural rigor states that the rigor of the system life cycle process should be commensurate with its intended trustworthiness. Procedural rigor defines the depth and detail of the system lifecycle procedures. These procedures contribute to the assurance that the system is correct and free of unintended functionality in two ways. First, they impose a set of checks and balances on the life cycle process such that the introduction of unspecified functionality is thwarted. Second, rigorous procedures applied to specifications and other design documents contribute to the ability to understand the system as it has been built, rather than being misled by an inaccurate system representation, thus helping to ensure that its security and functional objectives have been met.

Highly rigorous development procedures supporting high trustworthiness are costly to follow. However, the lowered cost of ownership resulting from fewer flaws and security breaches during the product maintenance phase can help to mitigate the higher initial development costs associated with a rigorous life cycle process.

Secure System Modification

The principle of secure system modification states that system modification procedures must maintain system security with respect to the goals, objectives, and requirements of its owners. Upgrades and modifications to systems can transform a secure system into an insecure one. The procedures for system modification must ensure that, if the system is to maintain its trustworthiness, the same rigor that was applied to its initial development must be applied to any changes. Because modifications can affect a system's ability to maintain secure state, careful security analysis of the modification is needed prior to its implementation and deployment. This principle parallels the principle of secure system evolution in Section II.

Sufficient User Documentation

The principle of sufficient user documentation states that users should be provided with adequate documentation and other information such that they contribute to rather than detract from system security. Even though the system may be designed for ergonomic security, its use may not be intuitively obvious. The availability of documentation and training can help to ensure a knowledgeable cadre of users and administrators. Where complexity must be minimized and where on-line documentation is inadequate, clearly written documentation and appropriate training is needed. If users do not know how to use a component properly, do not know standard security procedures, or do not know proper behavior to prevent social engineering attacks, they can easily introduce new system vulnerabilities.

V. Commentary and Lessons Learned

Include Security in Design from the Start

The problem with system security is that it is easy to find flaws, but it is difficult to find all flaws. Thus, if post-development flaw discovery and remediation is chosen as the path to achieving a secure system, then it is difficult to make a statement regarding the completeness of the security mechanism. Similarly, security functions that are added to a pre-existing system require analysis to ensure that they will perform with the level of trustworthiness intended. This analysis will extend to all elements depending on or upon which the security addition depends, as well as all resources shared by the addition, e.g. global data. Furthermore, unless the system has already been rigorously developed, the security analysis is likely to become so complex that starting anew would be more effective.

Generally, security redesign results in significant restructuring of existing systems so that they are aligned with the principles stated in Section I, e.g., [26]. Again, at a certain point it is prudent to apply the principles *a priori* rather than to attempt a retrofit.

The Philosopher's Stone

Experience has shown that when the principles described herein are applied to the construction of a system, development time and effort may rise in comparison to typical "time to market" driven commercial development practices. This conflict has resulted in various proposals for using untrustworthy components to achieve trustworthy systems, with mixed results.

Replication and other forms of fault tolerance have been described as candidates for developing trustworthy systems from untrustworthy components [21]. Although there are a number of constructs and techniques shared between security and fault tolerance [5], the presence of fault tolerance does not necessarily achieve security, and, conversely, security does not necessarily result in fault tolerance.

Similarly, the notion of "defense in depth" [7] describes security derived from the application of multiple mechanisms, e.g., to create a series of barriers against attack by an adversary. However, there is no theoretical basis to assume that defense in depth, in and of itself, could imply a level of trustworthiness greater than that of the individual security components. Without a sound security architecture and supporting theory, the non-constructive nature of these approaches renders them equivalent to temporary patches.

"Balanced assurance" [18] defines a hierarchy of security policies, where different policies may be allocated to different components of a system. In this approach, the trustworthiness of a given component must be consistent with the importance of that component's policy (i.e., greater importance requires greater trustworthiness). It is said that when the components are composed according to a precisely described set of rules, the trustworthiness of the resulting system will be equal to that of the most trustworthy component. While this approach shows promise with respect to specific examples, a coherent generalization has not been defined.

Other Approaches to Secure System Composition

A significant part of the cost of building a secure system is that of demonstrating its trustworthiness through its "evaluation" by a third party [13]. An approach to lowering these costs is to use components that have already been evaluated as to their trustworthiness, and thereby bypassing or minimizing the need to evaluate the system itself.

A variety of formal analyses has been performed to support the composition of security components by modeling the security properties that would result, e.g., [19]. Also, there has been support for "evaluation by pieces," which would acknowledge previously evaluated components, and not require their examination in the evaluation of the composite system. However, this approach has only been made available to "low assurance" systems, as it lacks a well-formed theory of correctness.

The Reference Monitor

The *reference monitor concept* [2] is an abstraction for the necessary and sufficient features of the component that enforces access control in a secure system. Its three characteristics are: the mechanism is protected from modification so that it always is capable of enforcing the intended access control policy; it cannot be bypassed, so that it is always consulted about requests to access to the information it is intended to protect; and it is understandable. To date, no viable alternative to the reference monitor concept has been proposed.

Because it is so abstract, the reference monitor concept provides no details regarding how an implementation might be constructed. Implementations will, to a greater or lesser extent, attempt to achieve the abstraction. The principles discussed in this document apply to the design and implementation of real systems. Some of those principles map directly to the reference monitor concept itself, while others provide the framework for constructing an implementation that is as close as currently possible to the idealized mechanism. For example, the principle of continuous protection of information clearly supports the reference monitor's notion of non-bypassability. Principles found in the context of Structure/Trust contribute to the ability of the mechanism to protect itself from tampering. A large number of the principles apply to the understandability of the mechanism, but some go beyond the original abstraction to address issues such as large-scale composition, maintainability, evolution, performance, and usability

Conflicts in Design Principles

Design principles need to be scoped and revisited during development, since there can be potential conflicts between their system specific interpretations. One principle can override or alter another principle. Though listing all potential conflicts is beyond the scope of this study, one example is the conflict between requirements of software engineering principle of "portability and reusability" and that of minimization; another is "minimized sharing" vs. "least common mechanism." These conflicts might not be satisfied simultaneously, but depending on the goals of the system, one principle may be emphasized to a greater extent than the other.

References and Bibliography

[1] Francis Afinidad. An Interval Algebra-Based Temporal Access Control Protection Architecture. PhD dissertation, Naval Postgraduate School, Monterey, California, June 2005.

[2] James P. Anderson. Computer security technology planning study. Technical Report ESDTR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. (Also available as Vol. I,DITCAD-758206. Vol. II, DITCAD-772806).

[3] Ross Anderson. Security Engineering. Wiley and Sons, Inc., 2000.

[4] William A. Arbaugh, D. Faber, and J. Smith. A secure and reliable bootstrap architecture. In Proceedings 1997 IEEE Symposium on Security and Privacy, Oakland, CA, May 1997, pp. 65-71.

[5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing, 1(1):11-33, 2004.

[6] N. Davis. Developing secure software. The DoD Software Tech, 8(2):3–7, 2005.

[7] Department of Defense, Directive 8500.1, Information Assurance (IA), 24 October 2002.

[8] Edsger W. Dijkstra. The Structure of the “THE”-Multiprograming System. Communications of the A.C.M., 11(5):341–346, 1968.

[9] Morrie Gasser. Building a Secure Computer System. Van Nostrand Reinhold, New York, NY, 1988.

[10] Intel. Intel Architecture Software Devloper’s Manual, Volume 3: System Programming Guide. Intel Corporation, Santa Clara, CA, 1999.

[11] Cynthia E. Irvine, Timothy E. Levin, Thuy D. Nguyen, and George W. Dinolt, The Trusted Computing Exemplar Project, Proceedings of the 2004 IEEE Systems Man and Cybernetics Information Assurance Workshop, West Point, NY, June 2004, pp. 109-115.

[12] Cynthia E. Irvine, Timothy E. Levin, Thuy D. Nguyen, and Terry V. Benzel, “SecureCore Architecture and Requirements,” *in preparation*.

[13] ISO/IEC. ISO/IEC 15408 - Common criteria for information technology security evaluation, July 2005.

[14] Philippe A. Janson. Using type-extension to organize virtual-memory mechanisms. SIGOPS Oper. Syst. Rev., 15(4):6–38, 1981.

[15] Philippe Janson. Using type extension to organize virtual memory mechanisms. PhD thesis, MIT (also MIT/LCS/TR-167), Cambridge, Massachusetts, September 1976.

[16] Butler Lampson. Protection. In Proc. 5th Princeton Conf. on Information Sciences and Systems, pages 437–443, Princeton, NJ, 1971.

[17] B.W. Lampson. A note on the confinement problem. Communications of the A.C.M., 16(10):613–615, 1973.

[18] Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Mark Heckman and W. R. Shockley, Element-level classification with A1 assurance, Computer and Security, 7:73-82, 1988.

[19] Daryl McCullough, A hookup theorem for multilevel security, IEEE Transactions of Software Engineering, 16(6):563-568, 1990.

[20] Jonathan Millen. The cascading problem for interconnected networks. In Fourth Aerospace Computer Security Applications Conference, Orlando, FL, December 1988, pp. 269–273.

[21] National Research Council, Trust in Cyberspace, ed. Fred Schneider, National Academy Press, Washington, D.C., 1999.

[22] Peter G. Neumann. Principled assuredly trustworthy composable architectures. Technical Report CDRL A001 Final Report December 28, 2004, SRI, Menlo Park, CA, 2004.

[23] David L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the A.C.M., 15(12):1053–1058, 1972.

[24] Charles P. Pfleeger and Shari Lawrence Pfleeger. Security in Computing. Prentice Hall, Inc, Third edition, 2003.

[25] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, 1975.

[26] Michael D. Schroeder, David D. Clark, and Jerome H. Saltzer. The Multics kernel design project. Proceedings of Sixth A.C.M. Symposium on Operating System Principles, pages 43– 56, November 1977.

[27] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. Comm. A.C.M., 15(3):157–170, 1972.

[28] Daniel Sterne, On the buzz word 'Security Policy', Proceedings of the IEEE Symposium on Research on Security and Privacy, Oakland, California, May 1991, pp. 219-230.

[29] Rita Summers. Secure Computing. McGraw Hill, New York, NY, 1997.

[30] Trusted Computing Group, TPM Main, Part 1, Design Principles, Specification Version 1.2, Revision 85, 13 February 2005, https://www.trustedcomputinggroup.org/downloads/specifications/tpm/tpm.

[31] W. Wulf and Mary Shaw. Global variable considered harmful. SIGPLAN Not., 8(2):28–34, 1973.