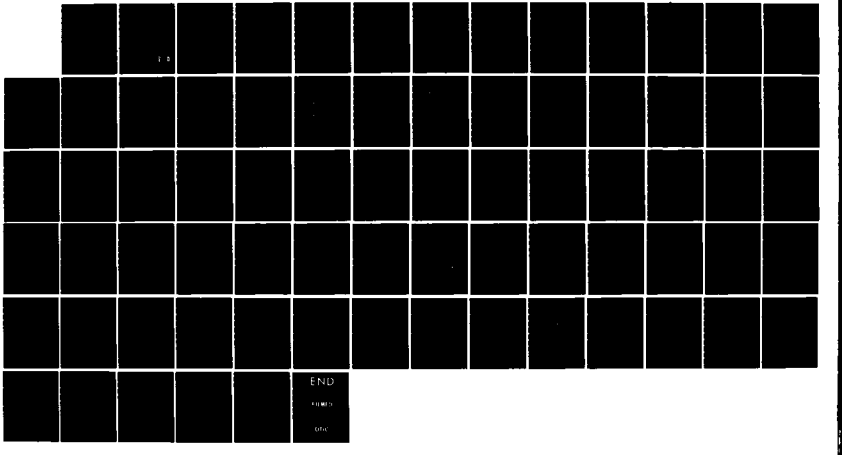
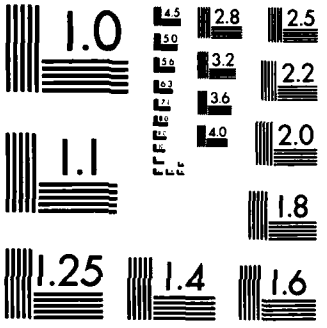
AD-A155 194 SECURITY CONCEPTS FOR MICROPROCESSOR BASED KEY GENERATOR CONTROLLERS(U) SYTEK INC MOUNTAIN VIEW CA R K BAUER ET AL. 24 APR 84 SYTEK-TR-84009 1/1

UNCLASSIFIED MDA904-82-C-0449 F/G 9/2 NL





(2)

SECURITY CONCEPTS FOR MICROPROCESSOR BASED KEY GENERATOR CONTROLLERS

SYTEK TR-84009

R.K. Bauer
R.J. Feiertag
B.L. Kahn
W.F. Wilson

24 April 1984

Contract MDA904-82-C-0449

A004: Final Report

Prepared for:

Mr. Howard S. Weiss
Maryland Procurement Office
Attn: L433 (JLC)
9800 Savage Road
Ft. Geo. G. Meade, MD 20755

SYTEK, Incorporated
1225 Charleston Rd.
Mountain View, California 94043
(415) 966-7300

CURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
;TEK TR-84009	S-226,997	

TITLE *(and Subtitle)*

:CURITY CONCEPTS FOR MICROPROCESSOR BASED
'Y GENERATOR CONTROLLERS

5. TYPE OF REPORT & PERIOD COVERED: Final Report, 30 Sep 82 to 31 Mar 84.

6. PERFORMING ORG. REPORT NUMBER:

AUTHOR(s)

. K. Bauer, R. J. Feiertag, B. L. Kahn,
F. Wilson

8. CONTRACT OR GRANT NUMBER(s): MDA904-83-C-0449

PERFORMING ORGANIZATION NAME AND ADDRESS

:TEK, Incorporated
:25 Charleston Road
:untain View, California 94043

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: SHELBURNE

CONTROLLING OFFICE NAME AND ADDRESS

:aryland Procurement Office
:TTN: L433 (JLC) 20755
:800 Savage Road, Ft. Geo. G. Meade, MD

12. REPORT DATE: 24 April 1984

13. NUMBER OF PAGES: 71

MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*

15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

DISTRIBUTION STATEMENT *(of this Report)*

:pproved for public release; distribution unlimited.

Accession For	
NTIS GRA&I	X
DTIC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A/	

DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

SUPPLEMENTARY NOTES

KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

security
microprocessors
secure architectures
software verification
security fault analysis
secure controller design
iAPX-286

ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

:e use of microprocessors in key generator controller designs can improve device :roughput, reduce size and heat dissipation, and provide for greater function-:lity while reducing cost and energy requirements. However, there are many :nherent difficulties in using microprocessors in a key generator controller. :is study investigates how the three disciplines of architecture, software :erification, and security failure analysis can be applied in a mutually support-:g manner such the resulting microprocessor based controller could be attested :o provide the level of security and reliability needed for correct operation. :e architecture study identifies the importance of relatively isolated process-

)D , FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

BLOCK 20. (Cont'd.)

ing domains. Every controller architecture must provide highly controlled intercommunication between several processing domains in order to provide for red/black isolation, separation of security and nonsecurity processing, etc. The software verification study reviewed standard techniques and found them entirely appropriate for this application. A high level security model is developed describing controller security. The traditional security fault analysis subdivides a circuit according to a block diagram or grid pattern to isolate the security relevant portions. The study analyzed these portions in a "bottom-up" approach by assessing the effects of individual (and combinations) or component failures on the circuit operation over a range of external conditions. The study provides a theory that a system is "safe" if it will continue to meet it's security requirements throughout its lifetime with some stated probability.

CONTENTS

1. EXECUTIVE SUMMARY

The use of microprocessors in key generator controller designs can improve device throughput, reduce device size and heat dissipation, provide for greater functionality while reducing cost and energy requirements. However there are many inherent difficulties in using microprocessors in a Key Generator Controller. Microprocessors are more difficult to analyze for correct and secure implementation due to the tremendous number of active devices, their failure modes and microscopic size. The introduction of software raises issues of software correctness and failure effects on software operation. While advancement of controller technology through use of microprocessors is important, it requires reexamination of security architecture concepts and assurance techniques.

This study investigated how the three disciplines of architecture, software verification, and security failure analysis could be applied in a mutually supporting manner such that the resulting microprocessor based controller could be attested to provide the level of security and reliability needed for correct operation.

The architecture study rapidly identified the importance of relatively isolated processing domains. Every controller architecture must provide highly controlled intercommunication between several processing domains in order to provide for red/black isolation, separation of security and nonsecurity processing etc. The simplicity of this mechanism plays a fundamental role in simplifying security fault analysis and software verification. The Intel iAPX-286 architecture was examined in detail as it incorporates significant security mechanisms into the microprocessor architecture and associated firmware kernel. The study discovered that while the 286 was extremely adept at creating, manipulating and deleting domains and objects, these mechanisms had complex implementations, were nearly impossible to monitor, and were not designed to meet the strict security requirements of the agency. An operational example using the 286 is developed in this paper and used for comparison against alternative architectures.

Processor-per-Domain architectures were investigated. These were found to have limited but adequate functionality for most applications. On the other hand, confidence in fundamental domain isolation is greatly improved due to static domains (no need for processor sanitization) and implementation entirely in hardware. An alternate Access Map architecture was developed which retains the high security confidence of the processor-per-domain approach while providing a large number of domains. The domain isolation is accomplished in hardware apart from the microprocessor(s), allowing it to be monitored or built in a redundant fashion.

Traditional Security Fault Analysis (SFA) subdivides a circuit according to a block diagram or grid pattern to isolate the security relevant portions. These are then analyzed in a "bottom up" approach by assessing the effects of individual (and combinations) of component failures on the circuit operation over a range of external conditions. This requires computer support for usable controllers, and becomes intractable when highly integrated components are used. To overcome the difficulties of this approach, the study proposed and developed a theory of safe machines.

A system is "safe" if it will continue to meet it's security requirements throughout its lifetime with some stated probability. When a component fails in a controller, it is in a sense, a new machine which is either safe or unsafe. If a controller in normal operation is "safe" (determined through design verification), each subsequent component failure changes the machine to a safe or unsafe state. Markov processes provide a mathematical framework for analysis of problems of this type. If probabilities are assigned to state transitions, a Markov analysis will yield the probability that the system will remain safe for a specified period of time. The study develops an example and explores some of the difficulties involved in characterizing the "components" of a machine which by design can be fairly coarse.

Software verification is the third topic of the study. Standard techniques are reviewed and found entirely appropriate for this application. A high level security model is developed describing controller security.

In summary, the study identified several acceptable controller architectures and evaluated their relative merits. Software verification techniques were applied and found satisfactory. Traditional SFA analysis was supplemented by the proposed theory of safe machines.

2. INTRODUCTION

Microprocessors are creating a revolution in computer systems and data communications equipment. Most new computer and data communications equipment contain microprocessors as main components. Microprocessors have been widely accepted because they are compact, high-speed, relatively energy efficient, provide many useful functions, and can be programmed to fit a wide variety of applications. However, microprocessors have been slow to be adopted for use in secure communications equipment and have not been used to their full advantage in this application. This caution is well advised as several microprocessor advantages become drawbacks when they are applied to security critical equipment. For example:

- The high-speed with which microprocessors operate allows data to be processed quickly and in large amounts, but it can also allow data to be compromised quickly and in large amounts.

- Microprocessor complexity which allows them to replace large amounts of circuitry also makes them difficult to analyze for correct and secure operation.

- The nature of the silicon medium and the microscopic size of individual transistors within the microprocessor makes them susceptible to many new types of failure that are often difficult to detect.

- Accurate means for evaluating the reliability and correct function of microprocessor controlled software requires further development.

Furthermore, the architectures of most microprocessors were not designed for secure environments and their highly integrated nature and pinout constraints make it extremely difficult to retrofit security into such computer systems.

This study investigates how the three disciplines of architecture, software verification , and security failure analysis can be applied in a mutually supporting manner such that the resulting microprocessor based controller can be attested to provide a level of security and reliability as needed for correct operation.

The body of this report is presented in three major sections: KGC Architecture, Verification and Safety Analysis.

The Architecture section describes how verification and safety issues present architectural constraints. The Intel iAPX-286 microprocessor is reviewed, a sample KGC architecture developed, and certain difficulties observed. A processor-per-domain (PPD) type architecture is used as a starting point for the development of an Access Map Architecture which combines the

security features of the Intel iAPX-286 chip with the verification and safety benefits of the simpler PPD architecture.

The Verification section concentrates on techniques required for formal verification of domain isolation and intercommunication properties of Domain Isolation Machines.

The Safety section develops a top down view of hardware fault analysis as opposed to the bottom up view adopted by existing SFA techniques. The section begins by developing a model of hardware faults in which a KGC with a failed component is represented as a different machine or state. It is noted that with suitable restrictions, Markov Chain analysis techniques may be used for analyzing the probability that a KGC will transform into an unsafe machine after a certain period of time. The section concludes by noting directions for continued development of this technique.

2.1 CONCEPTS

For purposes of exposition, a sample application has been selected which consists of a medium speed (19.2 kb) encryption device using a key generator controller shown in Figure 1:

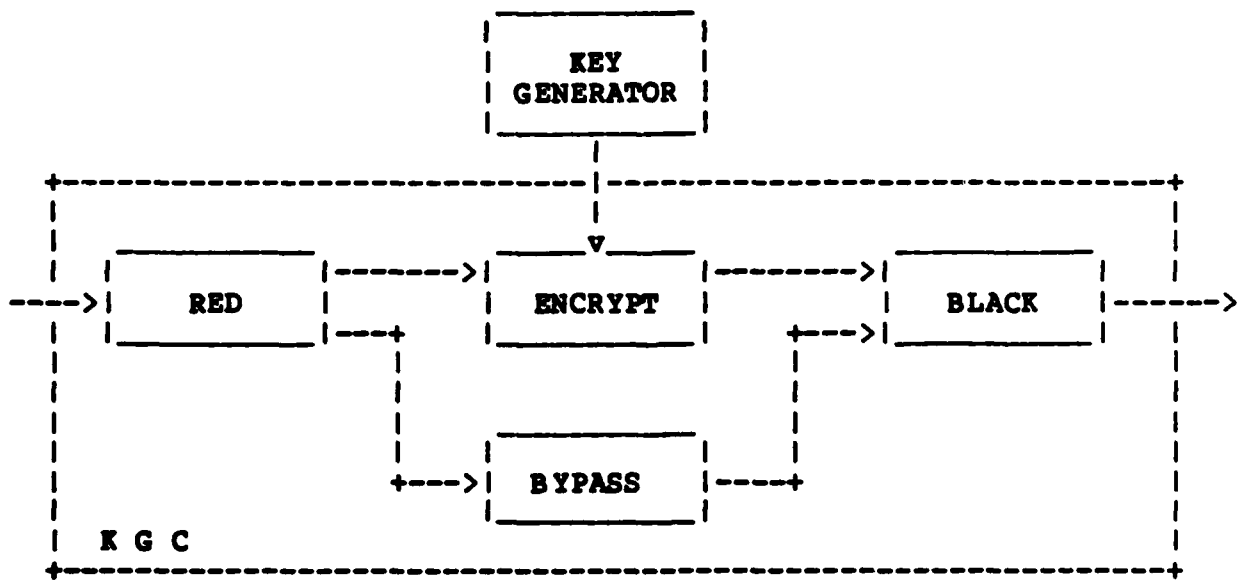


Figure 1. Key Generator Controller Application

The KGC contains four distinct processing environments: a RED domain which accepts unencrypted data input, a BLACK domain which transmits encrypted data output, an ENCRYPT domain which converses with the key generator and performs the actual encryption and finally a BYPASS domain which allows selected information to pass directly from the RED to the BLACK domains. This section of the report describes and recommends particular

architectural implementations for this application. However the reader should bear in mind that this application is very easily generalized and that the architectures described here are suitable for a wide range of security applications.

The primary function of a key generator controller (KGC) is to provide data security. The system must not improperly release any data. If a high degree of confidence cannot be gained in the secure operation of the KGC, then the system does not fulfill its primary purpose and is not useful. Historically, security has been obtained by using careful design and implementation techniques and by using redundancy to help detect and compensate for component failures. Confidence has been heightened through the use of Security Fault Analysis (SFA) techniques which analyze the effects of hardware component failures.

The use of microprocessors and software in KGCs, while providing many benefits, greatly complicates the analysis process required for security assurance. SFA analysis is rendered inadequate by the explosion of individual devices within microprocessors and its inability to address defects in software. So dramatic is the impact of microprocessors and their associated software that it is necessary to reexamine the entire confidence building process. In a system incorporating programable microprocessors, security analysis must rest evenly upon an analysis foundation of software verification, testing and security fault analysis. Each of these techniques has particular strengths, but no single method provides all the assurances required of KGC's.

2.1.1 Processing Domains

The complexities of software verification can be eased by appropriate choice of KGC architecture. The KGC architecture should provide for multiple domains of execution with limited, controlled intercommunication. This allows security critical software such as the BYPASS and ENCRYPT domains to be separated from software which deals with nonsecurity issues such as communications. This simplifies verification by eliminating the detail and complexities of communications software from the software requiring verification.

Similarly from the standpoint of safety analysis we soon realize the advantages of this divide and conquer approach. Isolation and minimization of security critical componentry is a mainstay of Security Fault Analysis. Critical circuitry can employ high reliability components and can be supplemented by physically and/or functionally redundant circuitry. The need for relatively isolated domains of computation is a focal point for our architectural discussion in that the overall quality of the verification and safety analysis for an application is governed by the quality of the underlying domain isolation and communications mechanism. This effectively motivates the layered architectural view illustrated in Figure 2:

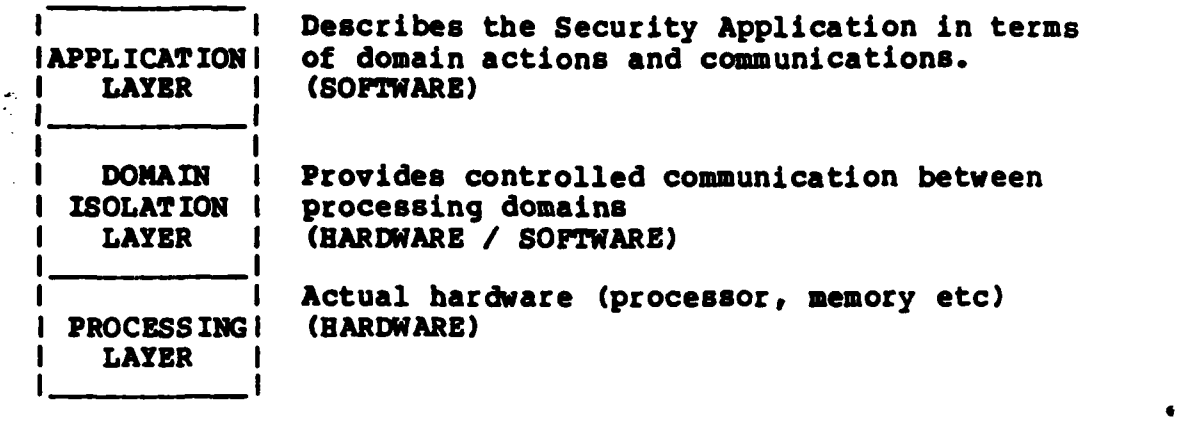


Figure 2. KGC Application as a Layered Machine

Each layer is dependent upon the services and security correctness of the layer(s) below. Starting at the bottom, the physical components must correctly implement their function and cannot incorporate design or implementation flaws which might affect their security performance. (E.g. if software memory addressing mechanisms are relied upon for domain isolation, the processor ALU must operate correctly in so far as it is used for address computations.) SFA and functional testing play important roles at this level.

The domain isolation layer uses the hardware devices of the processing layer to establish separate domains of computation with controlled intercommunication. This always requires specific hardware support, and the simplest and perhaps best approaches build the entire layer of hardware with no software what so ever. To the extent that software is used towards this goal, it must be rigorously verified, as all applications layer software is dependent upon the correctness of the domain isolation layer hardware and software. Again SFA plays an important role in assessing the strength of the hardware supports for domain isolation. Active health check testing is also important as it can continually attempt to breach the domain isolation and cause a shutdown if it is successful. Finally, if software must be used to achieve the goals of this layer, software verification techniques are also employed. The application layer describes the security application, in this case a Key Generator Controller. Actions are described in software as computations in various domains and interdomain transfers of information.

It is quite desirable to encapsulate the bottom two layers into a Domain Isolation Machine (DIM) which would export a standard software interface usable as a base for implementing security applications. Such a machine would enjoy several advantages over existing hardware bases. First, it would be reusable. This would allow much of the software and hardware analysis efforts

invested at these levels to be shared amongst current and prospective applications. Second, different Domain Isolation Machines could be built providing equivalent security but varying in implementation details such as data throughput potential, tempest characteristics etc. These could be relatively interchangeable amongst applications, providing the same functionality and security but differing in terms of throughput, cost etc.

The concept of independent domains of computation, and our confidence in the hardware and software mechanisms providing this feature, is a security concern which overrides the security details of the application built upon it. For this reason, the architectures described later in this section dwell heavily on this topic.

2.1.2 Standard Interface for a Domain Isolation Machine

Let's take a few moments to examine Figure 2 in more detail. Figure 3 illustrates the interface objects and actions present at the applications interface.

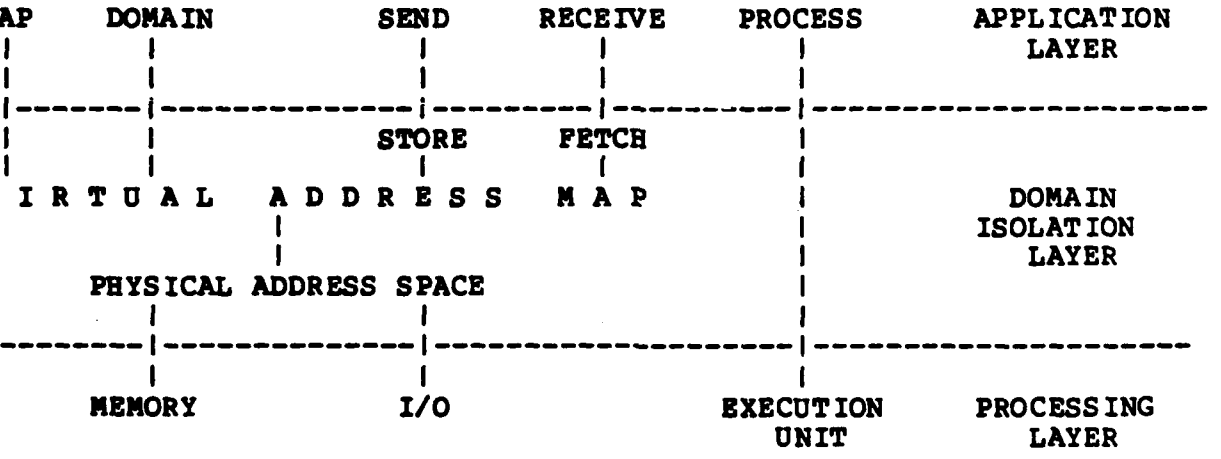


Figure 3. Application/Domain Machine Interface

A domain is a set of computation resources, and the DOMAIN object is a description of those resources to include quantity and location of memory and I/O ports. MAP is an object defining allowed inter domain communication. It can conveniently be thought of as a matrix with sending domains on one axis and receiving domains on the other. Presence of a one or zero in a cell designates whether that one way communication path is allowable or not. At this level the only appropriate actions are SEND some information to domain X, RECEIVE information from domain Y or PROCESS some information in the current domain. It is interesting to note that the means of implementing interdomain communication, whether it be shared memory or I/O based, is strongly rooted the Virtual to Physical Address mapping. Furthermore that a main source of processor complexity (e.g. everything else) is separate from this

fundamental issue of domain isolation and intercommunication.

2.1.3 Domain Machine Service Requirements

Different applications can place more or less strenuous service requirements upon the domain isolation layer. For instance a particular application may need very fine grained security levels requiring a large number of processing domains. Or even worse, the application may need to dynamically create processing domains and destroy them. Such requirements can be met with existing technology but always at the expense of complexity in the domain isolation layer and confidence in its security. Fortunately the KGC example, and for that matter the majority of communications security applications, require only a limited fixed number of processing domains. Figure 4 provides an overview of Domain Machine Service requirements.

CRITERIA	COMPLEX APPLICATION	SIMPLE APPLICATION
Granularity of Control	5 - 20 Domains	2 - 5 Domains
Dynamic Domains	Creation / Deletion	Static
Dynamic Objects	Creation / Deletion	Static
Implementation Issues		
o Domain Switch Time	Fast	One Processor per Domain OK
o Processor State Sanitization on Switch	Complete	Unnecessary
o Interdomain Communication Speeds	Fast	Slow

Figure 4. Processing Domain Requirements Issues

Granularity of control refers to the number of domains which are necessary or useful. In the case of our KGC application at least three are required: RED, BLACK and ENCRYPT/BYPASS. RED and BLACK must be separate with no intercommunication allowed. The ENCRYPT/BYPASS domains may be combined although there may be some utility to splitting them or having an even greater number of available domains. Dynamic Domains refers to the ability to dynamically create or delete a processing domain. Certain processing bases such as the Intel iAPX-286 provide this capability. Dynamic Objects refers to the ability to dynamically create or delete objects such as I/O buffers. Finally Implementation issues address certain aspects of implementing multiple domains on a single processor. How long should an interdomain control transfer take? How completely should the processor's externally accessible and internal registers cleared upon a domain transfer?

know" concept familiar to anyone in the security community.

The descriptor tables may be altered dynamically to pass a buffer from one domain to another. The descriptor tables described in the Task Definition section are only useful if they are protected. Only a very few routines are normally allowed to change the descriptor tables.

This sort of block transfer will occur frequently in a security application. In any sort of guard device, security filter, or KGC system, there are many separate tasks that operate on the same blocks of data. These are easily and securely passed from one task to another by switching the access rights. This is described in more detail in the KGC design analysis below.

We always want to keep our domains as isolated as possible, to preclude the possibility of unexpected data leakage between domains. On the other hand, for a domain to be useful, there must be some outlet to other domains.

The options available to the clever programmer are extensive. This is good in the programmer's eyes, but can be a demon in disguise. Here are a few of the possibilities offered by this descriptor table scheme:

- Buffers of any size up to 64K can be shared by one or more domains.
- Buffers can also be passed from one domain to another.
- Data segments may overlap, so that the buffers are partially shared.
- Partial buffers, pieces of buffers from one word to the entire buffer, can be passed to another domain.
- The access rights of read, write, and execute are associated with the domain, not the buffer.
- Any number of tasks may share a buffer.
- The access rights may differ for each task.
- The buffer size may differ for each task.
- Any of these may be changed during execution by altering the descriptor tables.

3.2.3.3 Data Flow Control We have stressed isolation and privacy over and over. Tasks often need to communicate with each other, and sometimes share data.

The Memory access section above showed the advantages in minimizing the amount of shared memory. There still must be some channels for inter-task communications.

There are three general inter-task communications channels controlled by the 286 microprocessor:

- Data created by one task and used by others
- Message and data blocks sent from one task to another
- Parameters passed to a routine accessed through a call gate

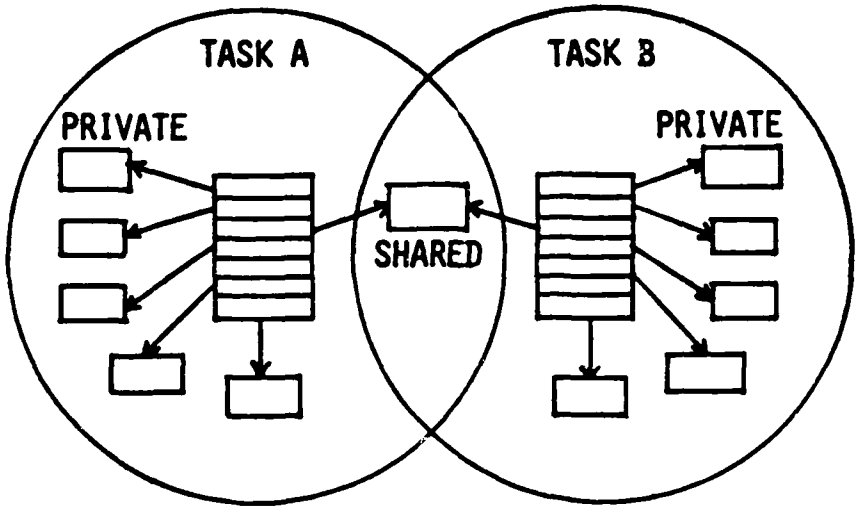


Figure 11. Tasks Sharing a Data/Code Area

The Memory access section above explained how each task on the 286 has certain access rights to specified segments of memory. A task is not allowed to read any segment of memory it does not have 'read' rights for, and likewise it cannot alter memory without 'read/write' rights. However, nothing prevents us from granting read/write access on some memory to one task, and read only access on the same memory to other tasks.

Since the separate tasks in a security application are actually sub-tasks of a single job, it is more common for tasks to share information. It must be considered common for data to be used by several tasks, but only modified by one or two. In general, a task that isn't required to alter some data will be required NOT to alter that data. This is similar to the "need to

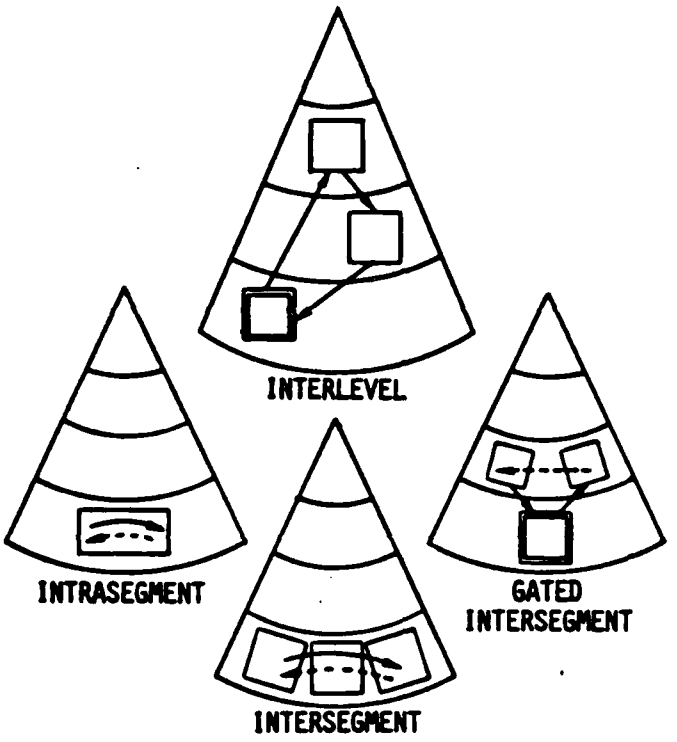


Figure 10. The Various kinds of Calls

These are calls that can be made within a domain. There is no restriction on calls made to another code segment within the current sub-domain, that is within the current domain at the same or lower privilege level. However, calls to a higher privilege domain require the use of a call gate, and the other kinds of calls may be gated if desired.

Calls to a higher level are a special case of inter-task calls. We are actually making a call to another sub-domain (see Privilege Levels, above) which shares all of the current resources, and has a few more of it's own. An inter-level call activates the program stack used for the called level, automatically copies a specified number of bytes from the old stack to the new one, and transfers control to a predefined entry point. An inter-task call, on the other hand, performs a complete task switch, swapping and storing all CPU registers and changing to the new LDT (see Task Definition, above).

Calls within the same subdomain may be gated, but this only helps when the caller doesn't know the level of the called routine.

example because there is rarely a mix of trusted code and untrusted code within a single domain. The four privilege levels within each task are handy for isolating data, restricting code access, and protecting the I/O ports from unauthorized use.

If the Intel task is considered to be a domain, the four privilege levels can be thought of as nested sub-domains. Code executing at a given privilege level can access data segments at the same level or at lower levels.* Separate stacks are maintained for each privilege level, but the higher levels can still access the stacks at the lower levels. Functionally, level 2 can be thought of as everything in level 3 plus some new things only available in level 2.

WARNING: Level 0 (the highest level) can bypass the protection mechanisms. Level 0 has some added abilities in every domain. There are extra registers and functions available at level 0 that can change the operation modes of the CPU. Level 0 must be used with caution! Any code operating at level 0 must be exhaustively analyzed. In a secure operation level 0 must not be used for anything except those functions which can only be performed at level 0. The cost, in verification effort, is very high for anything done at this level.

3.2.3.2 Logic Flow Control The 286 controls execution in more ways than just restricting memory access. The Task Definition section above explained how each task on the 286 has certain access rights to specified segments of memory. The 286 also controls branching between domains and program segments. A task must have permission to branch between code segments or to another task. A previous section described what happens on a task switch. This section describes when such a transfer is allowed.

To facilitate this, the 286 maintains a set of tables describing who calls who. Intel refers to these as "call gates" and "task gates". These gates are used for calls to routines outside of the current memory segment, to a higher privilege level, or to another task.

* Levels are named in a strange and unfortunate way. Higher privilege levels have a lower number. Level 0 is the highest.

levels, described in the next section below. Briefly, each of the higher privilege levels has a separate stack. The Back Link (offset 0) identifies the previous task, to make returning from a task easier.

The register labeled Task LDT Selector (offset 42) is very important. This is a pointer to the task's Local Descriptor Table (LDT). The TSS and the LDT together completely describe the individual characteristics of a task at any point in time.

3.2.3.1 Multiple Privilege Levels The 286 recognizes four "privilege levels". These levels form a hierarchy, which is best visualized as four concentric rings. This forms a figure resembling an archery target, in which the bullseye is the most privileged level. Each ring can access itself and any rings outside of itself.

Everything in the system has a privilege associated with it: blocks of memory, routines, entry points (Logic Flow, below), etc.

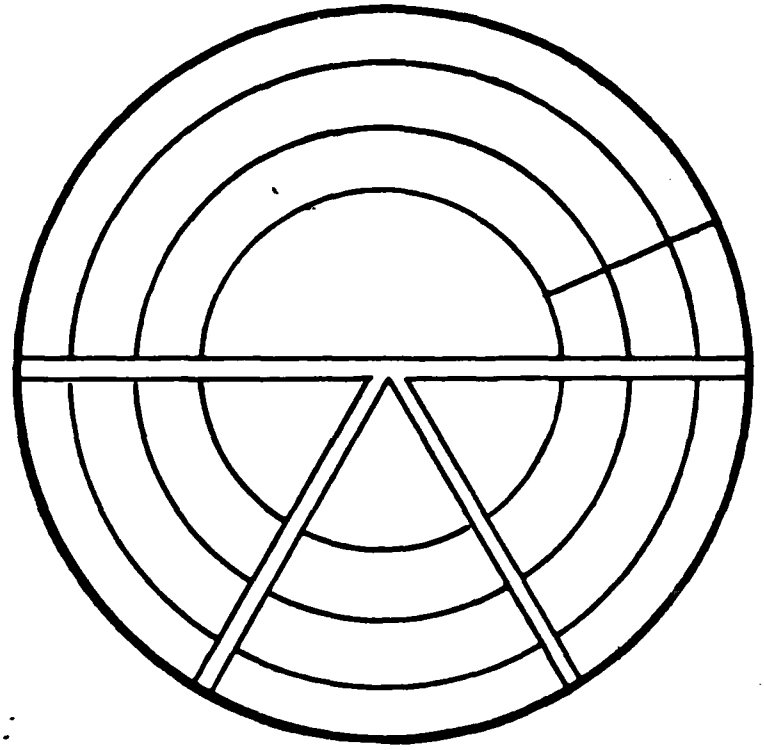


Figure 9. Privilege Levels in the Tasks

The multiple levels is an easy way to restrict access within a domain. As such, the effect on verification is similar to using a high level programming language - the code will be easier to read and well structured. Verified code differs from the OS

practically implemented as separate tasks.

The 286 performs a full sanitization during a task switch. All registers are marked during a task switch, and each must be associated with a descriptor from the GDT or the new LDT before they can be used.

The task state is saved during a task switch. Every task has a Task State Segment (TSS) associated with it. The TSS describes the current state of the task.

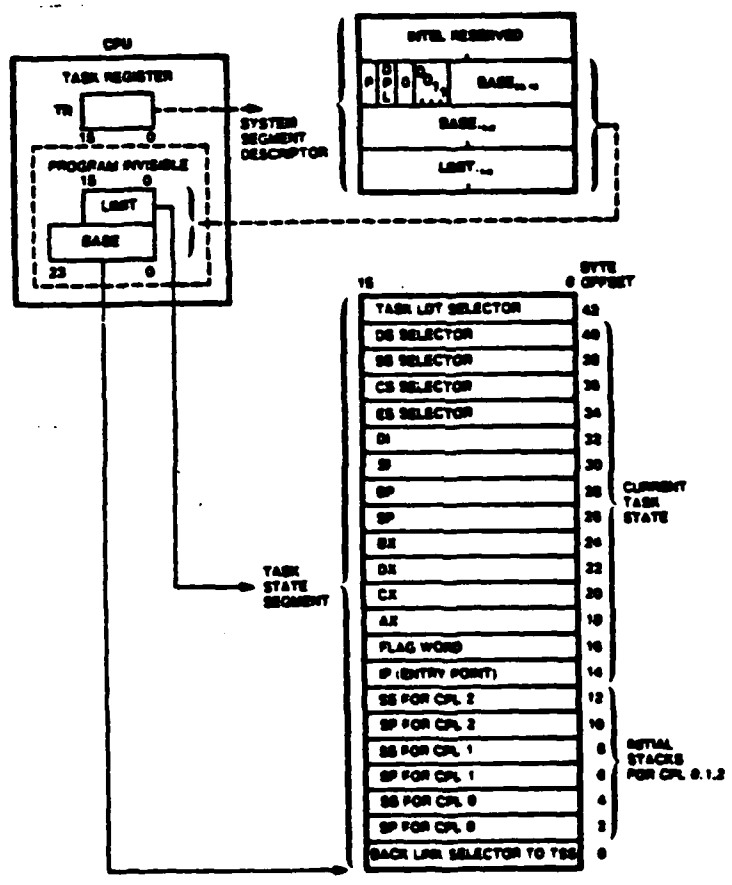


Figure 8. The Components of the TSS

Whenever control transfers to another task, the TSS of the exiting task is updated to show the current state. When the task regains control of the CPU, all the internal registers are loaded in from the TSS, and execution continues as if control had never left.

Most of the registers are pedestrian. The IP through DS registers (byte offsets 14-40 above) are the same as those found in the Intel 8086. The three extra sets of Stack pointers (offsets 2-12) are initial values for the additional privilege

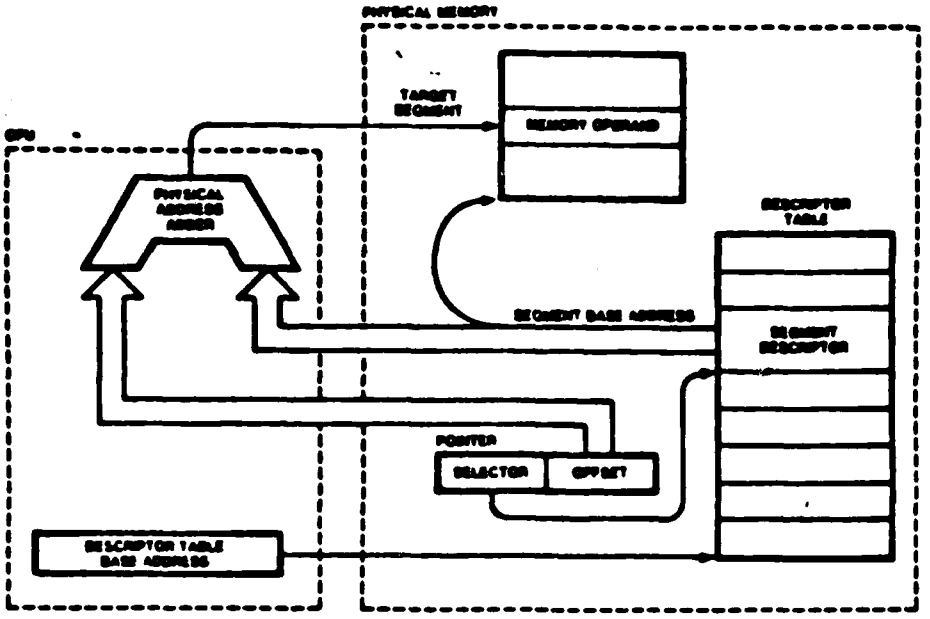


Figure 7. Computing the Physical Address

Individual tasks don't know what memory addresses they are actually using, because memory references are mapped into real memory. This facilitates the use of modular programs, because the code does not need to know where it will eventually reside.

A code or data descriptor indicates where the logical memory segment actually begins, how large that segment is, and what access rights are associated with that segment of memory. There are four possible values of the access rights field: read only, read/write, execute only, execute/write. These access rights cover almost anything a programmer might want to do. Every memory access is checked for legitimacy. Any attempt to use memory illegally generates a hardware interrupt, and stops the process.

The task gate descriptors provide the means for transferring control to another task. The only way for one task to make a jump or call to another task is by using a task gate. This means that for any given task, examination of the descriptor tables will provide a list of the tasks that may be branched to. This can simplify code verification considerably.

Transition from one task to another is done as a result of a single instruction, and is thus very fast. Most of the work involved in switching from one task to another is performed in 22 microseconds. This allows extensive use of task isolation without incurring a serious speed sacrifice. Even interrupt handlers, normally kept as short and fast as possible, can be

It is important to understand the two descriptor tables that pertain to any given task, the Global Descriptor Table (GDT) and that task's Local Descriptor Table (LDT). The GDT is shared by all tasks. Any descriptors (resources) listed in the GDT are shared by all tasks. The LDT is associated with exactly one task, and the descriptors listed in an LDT apply only to that task. The LDT gives each task it's individual characteristics. The LDT makes it possible to isolate the individual tasks to whatever extent desired or to share memory between a few tasks. Putting a descriptor into the GDT is equivalent to putting that descriptor into every task's LDT. The LDT and the GDT together completely describe everything that can be done by the task.*

A descriptor table may contain any of the following types of descriptors:

- Code Segment
- Data Segment
- Call Gate
- Task Gate

The code and data segment descriptors are machine resources in the traditional sense. These are physical blocks of memory, identified as either code or data. We will describe these first.

The call gate descriptors are an unusual kind of machine resource; the processor must have permission to branch to code in another code segment. These descriptors provide entry points to other procedures and programs within the same domain. These are covered in the section below on Logic Flow Control.

The task gates provide the capability to transfer control to another task. This is described later in this section.

First, the code and data segment descriptors. The 286 incorporates a memory management facility that provides thorough control over memory use. The processor maintains a set of tables that map logical memory to physical memory. The term "logical memory" means that when a task tries to access memory in one segment of memory, the processor automatically and invisibly routes the access to another segment of memory.

* This discussion ignores the Interrupt Descriptor Table for simplicity's sake. The interrupt gates are similar to call gates, and deserve no special attention.

The Intel task definition scheme allows the creation of separate and useful execution domains. A domain is a set of machine resources, and the 286 fully defines the machine resources available to each task. This definition is detailed enough to allow a full range of definition. Domains may share no resources at all, they may share a single data buffer and two code segments, or they may share all resources except two code segments, whatever is desired. This flexibility and power of the 286 task definition scheme is sufficient to create the Domain Isolation Layer described above (see Figure 5).

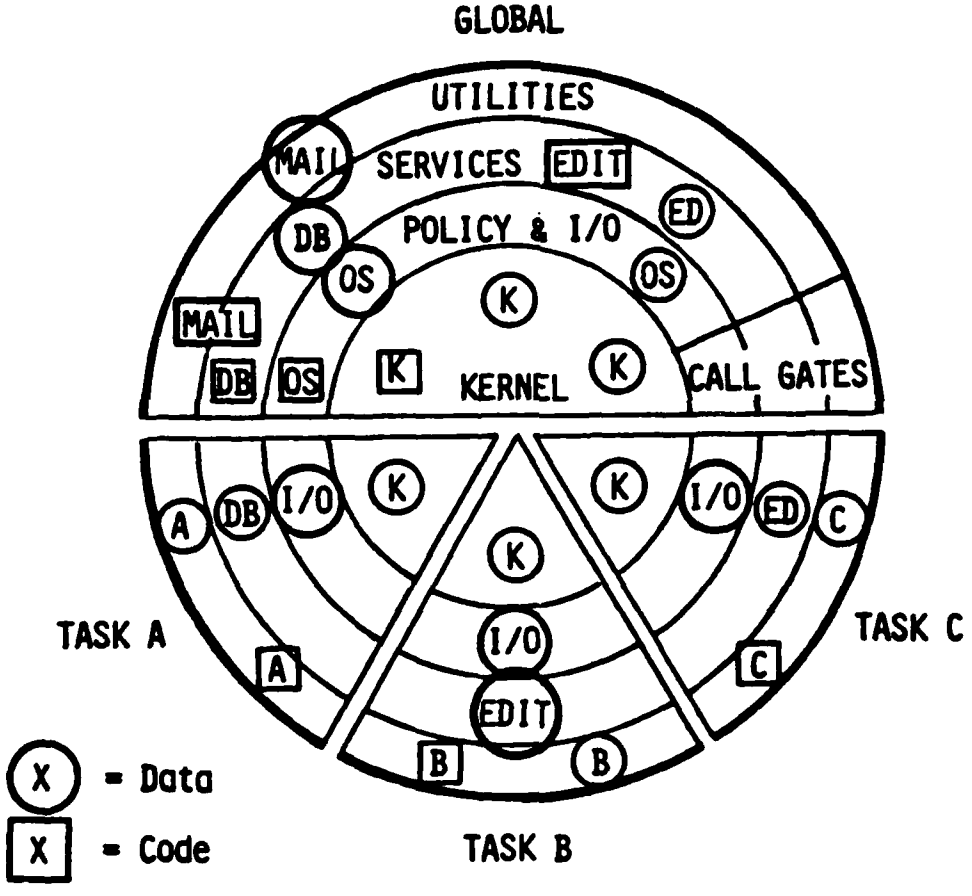


Figure 6. Tasks in the O.S. example

The Intel task is defined by two Descriptor Tables that list the resources of that task. Each descriptor defines some object, or resource. A descriptor is a pointer to a data segment or an entry point into some code. Combining the resources listed on the two tables with the internal CPU registers gives a complete list of the task's resources, and so it is a complete definition of the domain.

ever mess up the disk. To keep everything in order, the routines that actually manipulate the disk are hidden from the users. Calls are only allowed to high level, idiot-proof routines.

DATA FLOW CONTROL: Blocks of memory are frequently passed between a user and the OS. Whenever possible, the OS will handle input and output (I/O) in chunks, using blocks of memory usually called "buffers". Typical parameters used in a call to the OS I/O routine are the size and location of the buffer, not the buffer itself. The OS can temporarily change the access rights of that segment, perform the I/O, and restore the original access rights.

3.2.2 286 - Security Relevant Features

The 286 has four general features that are security relevant. We will discuss both how these features work, and what they are good for. The next five sections will explain the security features of the 286, and will also outline the underlying concepts of the Domain Machine. With these concepts in mind, we will be able to discuss both the 286 and other possible domain machines.

- Task Definition

  Intel tasks have firm perimeters, with no loopholes. Tasks are defined by the objects that they have access to.

- Multiple Privilege Levels

  The four privilege levels are actually a hierarchy of sub-tasks. Each privilege level has access to everything in the levels below it.

- Logic Flow Control

  There are several controls on who calls who, especially jumps between tasks or privilege levels.

- Data Flow Control

  Tasks can be totally isolated from each other, or they can share a selected set of objects.

3.2.3 Task Definition

We are examining the Intel 80286 because of its ability to isolate tasks from one another. The 286 is designed to support a large number of tasks in multi-tasking systems. The 286 allows the scope and capabilities of these tasks to be completely defined. If the tasks in the system are carefully defined so that they are mostly isolated from each other, with very little resource sharing, they will be the sort of domains that we are discussing in this paper.

systems. The requirements of the market niche that Intel originally targeted have been met. The requirements for a secure KGC system, however, are far stricter.

3.2.1 An Overview of the 286

The security relevant features of the 286 which we will discuss in the next section are rather detailed and technical. We will ease into it by presenting a short example. We will illustrate the "normal" use of the 286's capabilities through details of an imaginary Operating System (OS) that uses all four of the security features discussed below. This common and hopefully familiar example should aid understanding. Examples drawn from an imaginary multiuser OS demonstrate the significance of these features.

TASK DEFINITION: A multiuser OS provides timesharing of the single CPU. The OS will switch rapidly between users, granting each user a short burst of microprocessor time. This is invisible to the individual user. A user may notice that his job is taking longer one time than it did on some previous run, but otherwise everything looks the same.

It is important to the individual user that nothing gets changed in his individual area while the CPU is off serving someone else. This constitutes leakage from one task to another, and would cause behavior certain to confound and bewilder the unknowing user. All CPU registers, as well as the private data and code, must be intact after any number of "invisible" task switches.

Who uses what is a major issue in any shared environment. In general, every user would like to imagine himself the only one on the machine, with all of the machine's resources at his disposal. Any user would be very unhappy to discover that his data has been altered by someone else. Some users would be equally disturbed by the thought that others might even be reading their data.

MULTIPLE PRIVILEGE LEVELS: Generally speaking, a multi-user operating system has two responsibilities: provide user services and coordinate use of the system resources.

The system can rely on the users' good sense and hope that everyone asks the OS to provide these services. After all, users know they will cause problems if they blithely utilize resources whenever and however they want to. On the other hand, the system can reserve certain privileges for itself. Needless to say, the latter is the most commonly chosen path.

LOGIC FLOW CONTROL: The OS performs centralized, coordinated services for the users. Consider the case of disk usage. The OS wants to provide fast and easy access to the mass storage disk. Since the disk is shared by all, it is important that no user

Isolation layer in Figure 5.

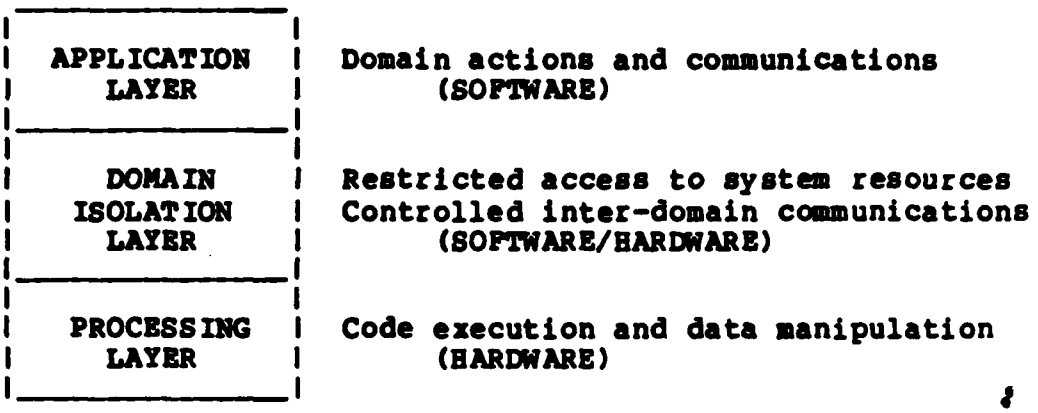


Figure 5. Domain Isolation Machine

There are many ways hardware can help provide this layering. The goal is to find a hardware/software combination which is capable of dependably "shaping" the domain, yet is easy to understand, use, and verify.

3.2 THE 80286 MICROPROCESSOR

The 80286 microprocessor was designed to support process separation. The 286 has many features not usually found on a microprocessor chip which make process separation much easier. Some of these features, such as the virtual memory management facility, have many uses. Other features are oriented entirely towards process separation.

Intel contemplated security issues from the earliest design stages. Intel had design analysis performed by security experts to determine the performance of their planned 80286 in a security environment. Advanced modeling was performed by this group in an effort to uncover flaws and weaknesses in the 286 design. Intel considered the results of these studies in their final design.

Intel's goal was to build a fast processor with the ability to enforce varying degrees of isolation on individual processes. The market they targeted wanted a processor that would handle a complex system of interlocking processes. They wanted to satisfy a wide range of applications, some of which were quite elaborate. Intel planned a chip with tremendous flexibility and power.

Intel came very close to their goal The 286 is very good for many customers, with applications ranging from PBX switching networks to small multi-user business computers. The process isolation built into the iAPX-286 system is stronger than on any other commonly used microprocessor. Customers can have higher confidence in the hardware based protection mechanisms on the 286 than in the software protections that must be used in other

3. ARCHITECTURE

3.1 OVERVIEW

This chapter will dive into the architecture of Key Generator Controllers (KGC). We will refer to verification and safety issues where these affect the hardware and software design.

We have presented the conceptual base. The following chapter will explain in more detail what a domain is and how domains are used in verification. The chapter on Safety will show the benefits of distributing responsibility over small modules or "chunks" in the hardware design. This chapter will demonstrate how the theories of Safety and Verification can be applied to useful, functioning architectures. We now tie the theory to the real world.

We will cover four areas in this chapter:

- The 80286 Microprocessor

  Security features on the 286. Also presents the basic concepts needed to discuss the implementation of a domain machine.

- A Sample Application

  A KGC system software design. This example is used to illustrate the use of a domain machine in a secure environment.

- Using the 286 as a Domain Machine

  How to use the security features of the 286 for domain isolation. Describes specifics of the sample application. Problems with the 286 are summarized.

- Alternative Domain Machine Architectures

  Two alternative hardware designs are described and compared to the 286. The designs are then abstracted so they can be implemented in other ways.

At this point the reader should have an understanding of what a domain is and why domains are desirable. Since a domain is a limited set of resources, there is less much less to prove about a section of code executing inside of a domain. This simple fact will eliminate many pages of complicated argument during code verification and results in a much stronger proof.

This chapter will show how much easier it is to implement domains on a machine designed to support them. Through example and analysis, we will show what must be provided (through some combination of hardware and software) to create the Domain

The main issue here is that different applications may make greater or lesser demands upon the Domain Isolation Machine base. The more demands, the greater the complexity of the underlying DIM, and presumably the more complicated its verification and safety issues. In the architecture descriptions which follow, to the complexity of supported applications is addressed.

3.3 A SAMPLE APPLICATION

We now present a software design for a KG Controller. The sample KGC will help us demonstrate the use of a domain machine in a secure application. We will discuss the implementation of this KGC design on the Intel iAPX-286 system, and compare this with an implementation on a sophisticated alternative architecture of our own design.

This example will illustrate the usefulness of the domain machine in security critical applications. We will also derive the characteristics that are most important in the design of the domain machine itself. These characteristics will provide us with a yardstick for evaluating domain machines. Later in this chapter, we will take this 'yardstick' and evaluate several possible designs for domain machines.

3.3.1 Software Design Using a Domain Machine

The domain machine is a unique environment. A software designer working on a domain machine has control unlike any other programming environment. The modern high level, block structured programming languages offer powerful data scoping and logic flow controls. These structured and clearly defined tools speed applications development and enhance confidence. The domain machine takes the same concepts several steps further. Software design on a domain machine is like having an extraordinary high level language built specifically for the application at hand.

Isolation provides both protection and restraint. Each individual domain receives a double benefit from domain isolation. First, the integrity of the domain's private data and code space is guaranteed. There is no way that any other domain can access private areas. Second, the domain is only allowed access to certain restricted shared areas. This makes it very easy to satisfy requirements that a particular domain does not do certain things. In many cases, the domain machine will prevent the domain from doing these forbidden things. For example, it is easy to demonstrate that a certain module doesn't corrupt the disk if the domain doesn't have write access.

The domain machine allows tailored capabilities. The definition of each domain identifies what that domain can or cannot read, write, call, etc. In any high level language, there are complicated rules that identify the same thing for every procedure, but these rules are much less flexible and somewhat more difficult to understand. Only the most elaborate languages, such as DOD's ADA, even approach the thoroughness and flexibility of the domain machines we are discussing here.

The domain machine provides better security properties than any high level language can. The exacting code verification required for KGCs demands advanced data and logic flow analysis. Languages such as Pascal simply don't offer the data hiding and definable scoping needed to implement this level of design.

Additionally, it is not possible to verify the operation of compilers. Compilers are far too complex. A language such as ADA makes a fine companion for the domain machine, but it is not an adequate to replacement.

Good design will make verification much easier. As the Verification chapter will show, the work involved in code verification is considerable. The use of a high level language can cut the verification effort to a fraction of the work required for a low level language. The benefits from using a domain machine are similar. The money saved by building on a domain machine is surpassed only by the increased confidence in the security of the system.

3.3.2 The Requirements For Our Sample Application

We will now begin using a sample application for illustration of concepts. A few specifics are needed.

The sample application will encrypt messages travelling from a highly trusted (RED) environment to some less trusted communications net (BLACK). The messages will have a maximum size of four thousand characters (4K). The message headers will vary in length, and the headers will not be encrypted. The messages will be released with an unencrypted header and an encrypted body.

The messages will arrive continuously. The data rate will not be exceptionally high, but the KGC must have as little impact on the sender as possible. Causing the RED sender to wait for the KGC to encrypt a message is unacceptable.

3.3.3 The High Level Design

The first step in any high level design is to determine the major functions of the target system. When working with a domain machine we also want to break down the job into blocks with as little functional overlap as possible, so that we can isolate them. The sharing of data or low level code must be kept to a minimum. In this case, the first step is fairly easy. The Introduction to this paper described the job of a KGC with the diagram below.

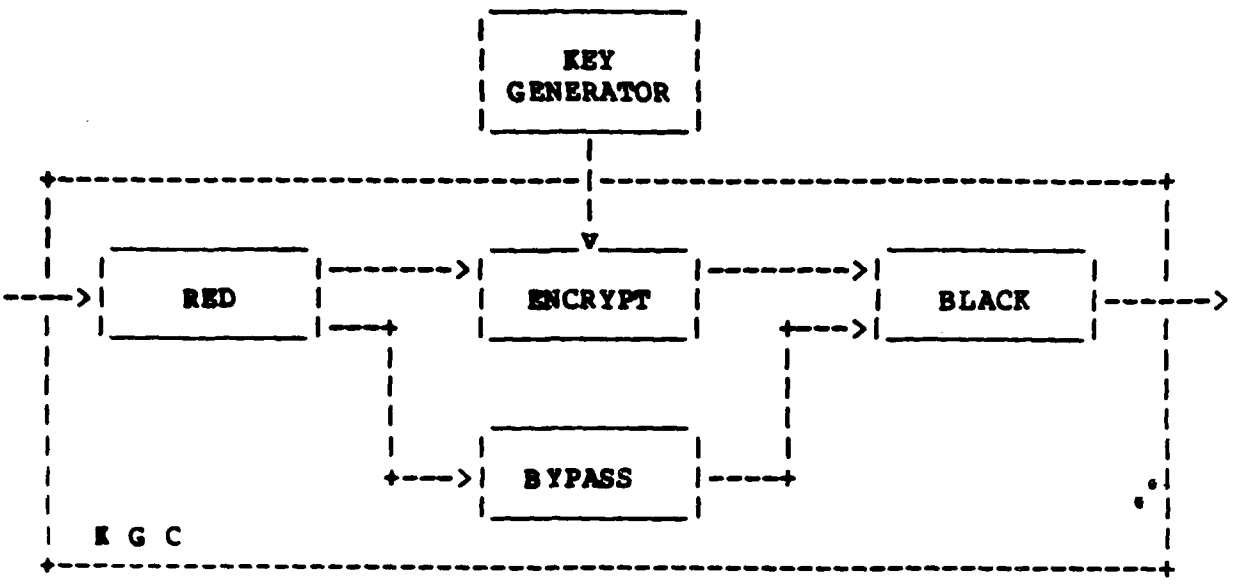


Figure 12. EGC Conceptual Block Diagram

This is a fine conceptual breakdown, but it needs to altered slightly to work as our design breakdown. Taken literally, this diagram suggests that data somehow sends itself to either the ENCRYPT or BYPASS blocks. Curiously, the BYPASS block does nothing but let data pass through. There is an intelligence implied for the system by the figure above, but that intelligence doesn't reside in any of the conceptual blocks. The main ideas of the system are shown, but a Functional Block Diagram must assign all the decision making to some block to be complete.

The design pictured below contains a block with the responsibility for whether data will be encrypted or bypassed. This is the PROCESS block. It replaces the BYPASS block, because bypass is one of it's functions. The PROCESS block acts as the system's traffic cop, routing the correct portions of messages to the right places. The main difference between the two diagrams is that one block is responsible for analyzing the message and routing it properly.

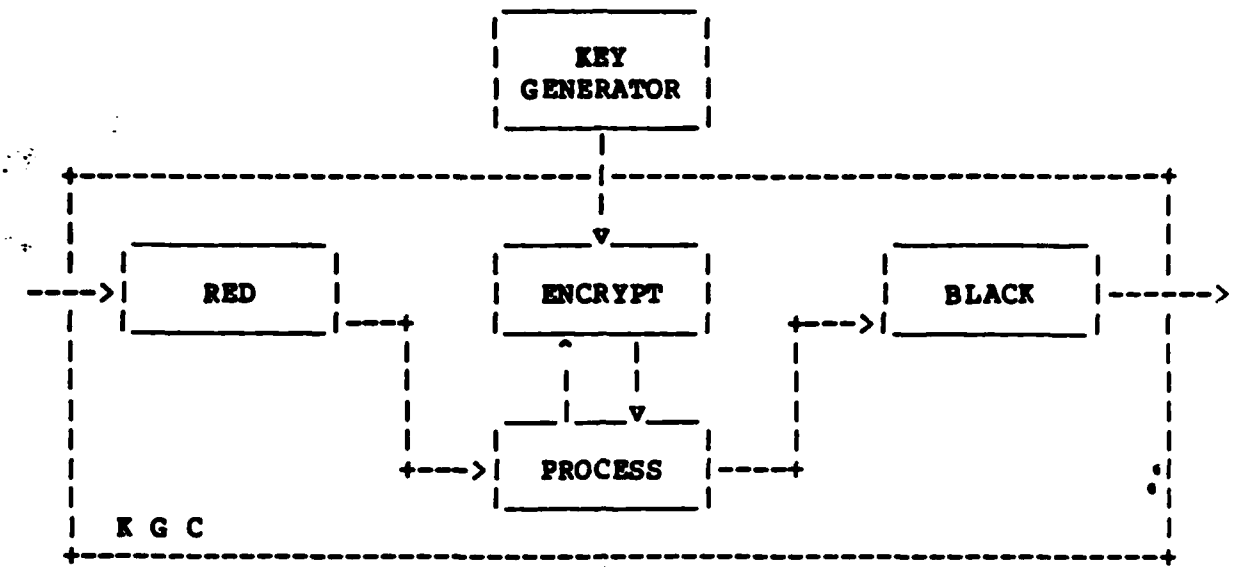


Figure 13. KGC Functional Block Diagram

The design at this point encompasses four tasks. The four tasks correspond closely to the original four functions: Red, Black, Process, and Encrypt. RED and BLACK handle the input and output, respectively. Neither has any inter-domain communication skills. RED writes the input data into a buffer and flags PROCESS when the message is completely received. BLACK outputs a buffer, and flags PROCESS when the message has been completely transmitted. ENCRYPT utilizes the Key Generator to encrypt a buffer, and flags PROCESS when the encryption is complete. PROCESS takes a message from RED, sends the body of the message through ENCRYPT, and releases the message to BLACK.

The next step in design breakdown is to simplify each of the domains as much as possible. Every capability added to a domain increases the effort needed to verify it. Only RED and ENCRYPT can write into the message buffers, while all of them can read the message buffers. Only PROCESS needs to recognize the flags that RED, BLACK and ENCRYPT use to indicate when their tasks are completed. Only PROCESS is allowed to transfer buffers between domains. There is no sharing or communication between domains except for the buffers moved by PROCESS and the flags recognized by PROCESS. This will simplify the verification of any domain machine implementation of this system.

One very good example of restricting the capabilities of domains is demonstrated by our handling of the ENCRYPT block. It is tempting to allow the ENCRYPT block to exit directly to the BLACK output block instead of returning to the PROCESS block. (Note that this is actually implied by the Conceptual Block Diagram above.) This would be a mistake, because the ENCRYPT

block does not need the ability to move messages between domains. The PROCESS block is the only one who can move messages, so all requirements regarding the release of messages to the BLACK output block can be satisfied by analyzing the PROCESS block code.

We will take the idea one step further. In each of the four blocks above, there is some smaller portion of the block that must handle the message buffer itself. These portions do require access to the same areas as the whole block, but they have one extra capability. Since handling the message buffer is important and dangerous, it may be useful to handle these portions separately when possible. In the diagram below, each domain is split into two sub-domains. The inner ring in each domain contains the code which is allowed to manipulate the message buffers. Each of the several implementations discussed later in the chapter will handle this part of the design differently.

Note that the code in the higher levels is not more trusted than the code in the lower levels. The ring concept is used to add an extra measure of data/code isolation. The advantage gained is a limiting of what needs to be proven about each section of code during the verification process. The message buffers will always be security critical, so any code which manipulates them will be closely restricted.

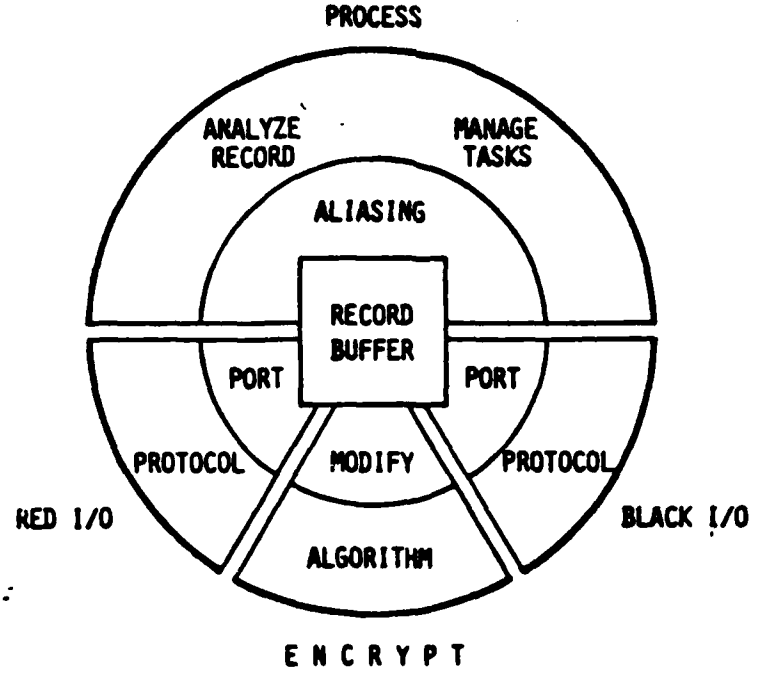


Figure 14. KGC Software Design

The highest level design is very important step, and should be performed carefully. The implementations of this design will

be somewhat different on the different domain machines, but the isolation in this high level design will be maintained in each implementation.* Notice how much attention was given to security issues before the target machine had been determined! This is a crucial element in successful creation of a secure system.

3.4 USING THE 286 AS A DOMAIN MACHINE

First we must outline the hardware design for a domain machine using the 286. The 286 can be used to provide a fairly substantial Domain Isolation Layer. There are several important hardware requirements, but overall the design is quite straightforward.

The most important restriction is that the 286 must be the only intelligence in the system. This means that no other chip will be able to take control of the bus. This precludes the use of Direct Memory Access, intelligent floating point chips that can access memory, and slave processors.

The I/O must be memory mapped. The I/O handlers are something of a problem because the I/O ports are not mappable. Unlike memory, I/O ports cannot be restricted to one domain. The 286 supports an I/O privilege level (IOPL), which restricts the use of I/O opcodes to code executing at the IOPL or higher. Unfortunately, this means that any routine which can access a port can read or write ANY port. Because of this, our hardware design will use memory mapping for I/O. Memory mapped I/O provides the I/O handlers with as much protection as the rest of the system. The main disadvantage of this approach is the I/O instructions which perform block moves cannot be used.

With these hardware restrictions in mind, we can create a sample hardware to run the sample application.

* Any domain machine will handle the domain isolation, but each domain machine will differ, just as computers differ.

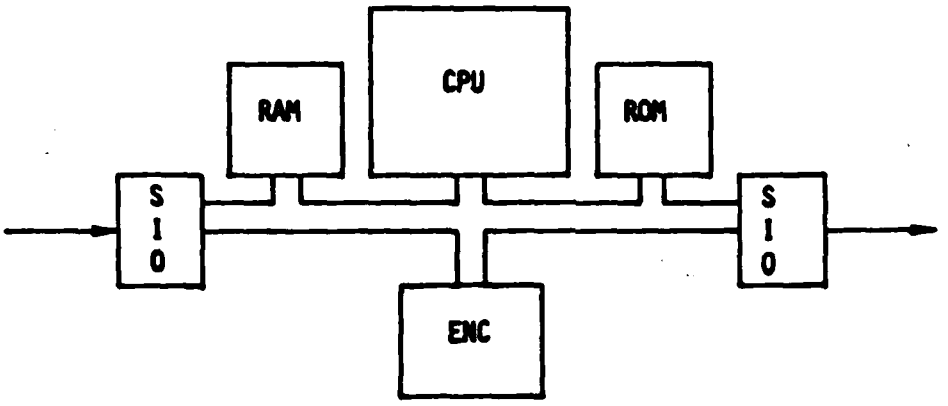


Figure 15. KGC Hardware Design Using a 286

The Serial I/O (SIO) chips are the connections to the outside world. The RED and BLACK domains will each have access to one of these chips. The box labeled ENC is an interface to a Key Generator, which will be used in the encryption process. The ENCRYPT domain will have access to this.

Now that we have a hardware base, we can proceed to the software portion of the domain machine. The 286 offers a great deal of support for the domain machine in hardware, but there is still a great deal that must be done in software. The method chosen for buffer passing must be described and the domains must be defined.

The buffers are passed between domains by modifying the domain definitions. The 286 allows very large (up to 64K) segments of memory to be logically transferred from one domain to another almost instantaneously. This capability is exploited to save the time involved in copying buffers from one domain to another.

All it takes to pass a buffer from one domain to another is changing pointers in the Local Descriptor Tables (LDT) of the domains involved. Of course, there is nothing new about passing pointers to data instead of the data itself. The difference here is that the 286 memory management maps all logical addresses to other physical addresses based on the contents of the pointers in the current task's LDT. When we change the LDT, the 286 will automatically reroute the same logical memory accesses to a different section of physical memory.

When the RED domain signals that it has a full message buffer, the PROCESS domain trades it for an empty buffer. From the point of view of the RED I/O domain, that previously full buffer has been instantaneously replaced by an empty buffer. The

task itself, the RED domain, works with a logical object. When the definition of that logical object changes, the task can't even tell. The RED domain doesn't know where the full buffer went, and could not access the buffer even if it did know.

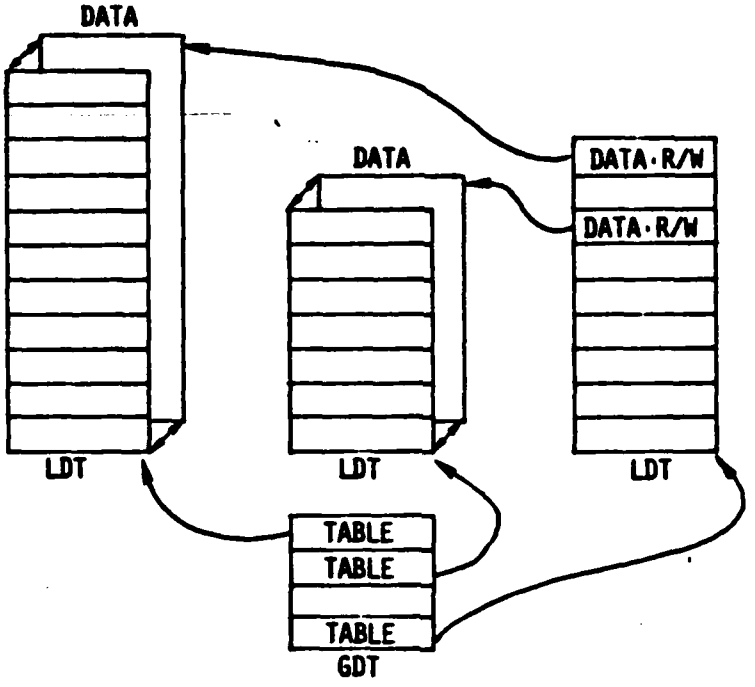


Figure 16. Modifying the Domain to Pass a Buffer

This is all performed through manipulation of the Local Descriptor Tables (LDT) in the various tasks. The figure above shows three LDTs and the Global Descriptor Table (GDT). There are descriptors in the GDT that identify each of the LDTs. This is the normal operating condition. There must be a Global Descriptor for every LDT. The unusual activity here is the aliasing of two of the LDTs through the third LDT. Aliasing refers to situations where the same object is referred to by two different names, often with different characteristics, as is being done in the diagram above. In this situation, the third task views the LDTs of the other two tasks as Data Segments. This allows the third task to alter the LDTs of the first two tasks.

The PROCESS domain manages the movement of messages in the KGC. It does this by altering the LDTs of the RED, BLACK, and ENCRYPT domains. The LDT for the PROCESS domain gives PROCESS Read/Write access to the other three LDTs. When PROCESS moves a buffer from RED to ENCRYPT, it moves the descriptor pointer to that buffer from RED's LDT to ENCRYPT's LDT.

There is one other aspect to aliasing worth mentioning. Nothing says that buffers need to be made entirely available to each domain. For example, the messages have headers which are not to be encrypted. Descriptors associate a task with a segment, not with an object. This allows painless dissection of buffers. There is no reason for the ENCRYPT domain to ever see that header. This is illustrated in the figure below.

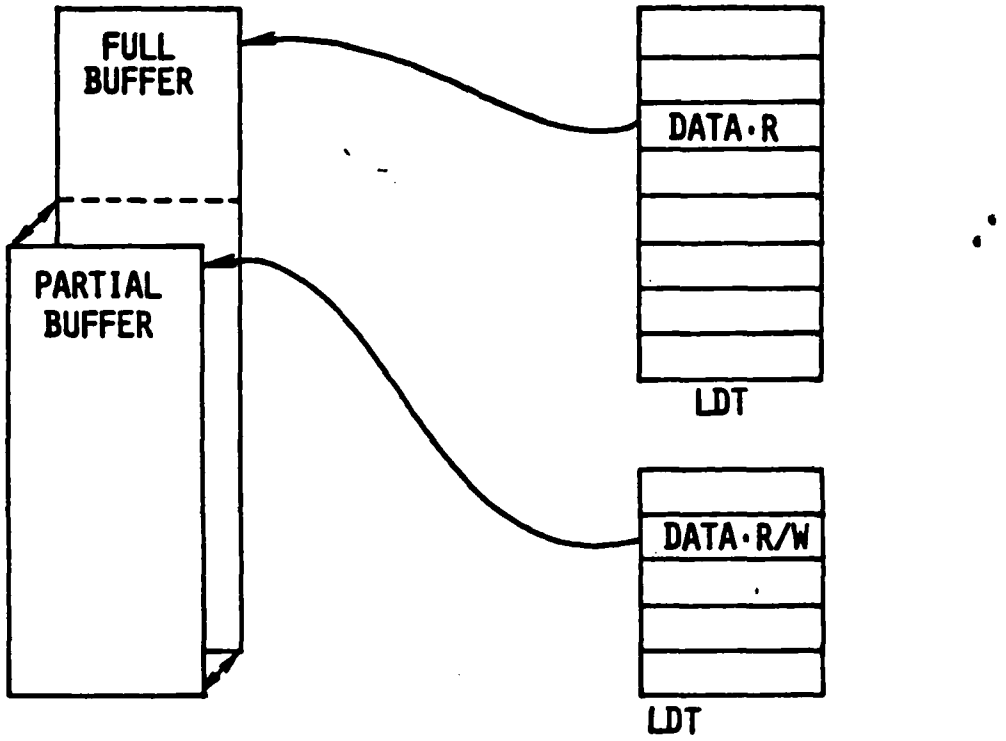


Figure 17. Aliasing of Partial Buffers

The descriptor tables must be properly handled by the software. Passing buffers from one domain to another is the fastest method for sharing data objects. The alternative to passing the buffer is copying one buffer into another, which can take considerable time.

The access rights and buffer size are associated with the domain, and so can differ for the same object.

A walk-through of the code, tracing a record through the KGC, should clear up any remaining questions.

The 286 is in unprotected mode at power-on. The start-up code must set up all the descriptor tables properly, and then jump into protected mode. The correctness of all code associated

with the descriptor tables must be solidly proven, because this is part of the domain machine and the verification will depend heavily on this.

The contents of the descriptor tables themselves must be verified. The construction of these tables cannot be trusted to Intel's automatic tools. The output from Intel's support tools could be verified, but it is probably easier to build these complicated tools by hand than to wade through a data file. To simplify the analysis of these domains, the Sample Application will put nothing in the GDT except what must be there - descriptors pointing to each of the tasks. No code or data segments, no call gates will be allowed in the GDT. The LDTs will be built as structures in the C programming language and linked to the Pascal used for the rest of the code. LDTs described in Pascal or in assembler are both very difficult to comprehend, although for opposite reasons. Pascal hides the machine representation of data from the programmer, and assembly requires the specification of every byte. Considerable support will have to be shown during code verification to argue that this is all done properly. The tables are full of pointers that refer to each other, and many pointers that must refer to the same area of memory.

Once in protected mode, the PROCESS domain must allocate buffers. PROCESS has the power to alter the other domains, and it uses this ability to allocate buffers. Since it would be too dangerous for PROCESS to be able to alter it's own domain, the choice was made to give PROCESS Read access to all buffers at all times. PROCESS gives an empty buffer to the RED, BLACK, and ENCRYPT domains. BLACK and ENCRYPT ignore empty buffers, but RED will fill up it's empty as data comes in. The start-up procedure is now complete.

The RED domain fills the buffer as the message arrives. RED takes care of all the low level protocol, and manages the Serial I/O (SIO) port. When the message is completely received, RED will set the staus flag "message complete" and wait for the status flag to reset.

The PROCESS domain waits for this flag to be set. When PROCESS sees this flag, it replaces the descriptor in RED's LDT pointing to this full buffer with a descriptor pointing to an unused buffer. PROCESS then determines where the message body begins within the buffer, and manufactures a descriptor that only refers to this part of the buffer. This new descriptor replaces the empty buffer in the ENCRYPT domain, and the status flag is set to "please perform encryption".

The ENCRYPT domain uses the Key Generator to perform encryption on the message, and then sets the status flag to "ready to go". ENCRYPT will now wait until the staus flag is reset to "please perform encryption".

PROCESS will now release the message by swapping the descriptor held by BLACK's LDT with a descriptor for the current message, header and body. If RED filled another buffer in the interim, the PROCESS task will shoot this over to ENCRYPT. If there are no new messages, ENCRYPT will get a null buffer again.

The BLACK domain performs a job similar to RED, except that it cannot alter the buffer. The message is released, and any protocol is handled as necessary. Again, BLACK will alter the status flag when it is finished, and wait for the flag to be reset by a new buffer.

If BLACK or ENCRYPT are still busy with the last message when a new one is ready for them, PROCESS will place that message's descriptor onto a waiting queue. There must be a waiting queue for both BLACK and ENCRYPT. If we run out of memory, the RED domain will have to wait for a buffer to free up.

We have to stress the importance of the proper handling of these buffers. Any mistakes in the buffer passing could compromise security. The algorithms themselves are not too difficult, but there is concern that human verifiers may have difficulty tracing through the Descriptor Tables. Well written code should support verification, but it has been our experience that the programmers must have a good understanding of the verification process to produce code clear enough for high confidence. Any project using the 286 must take great care in this area.

3.4.1 Problems With The 286

A number of problems have been noted so far. We will summarize these and discuss problems with the 286 in more detail here. There are five classes of problems with the 286:

- Invisible - "Too Integrated"

  Is it designed right? Is it working?

- Invariable - "Too Bad"

  You had better like Intel's ideas.

- Inextensable - "Too Lonely"

  security properties cannot extend to other chips.

- Incomplete - "Too Few"

  Design compromises cause security problems.

- Incomprehensible - "Too Tricky"

  Complexity reduces confidence - mistakes can be subtle.

3.4.1.1 Invisible Intel has packed an astounding level of functionality onto a single chip. This has benefits in higher speed and lower cost. Unfortunately, for the purposes examined in this paper there are many difficulties caused by the high integration.

We need the highest confidence in the domain machine for our KGC controller. The verification for many portions of the software will depend heavily upon flawless operation of the domain machine. The confidence of the whole system will increase with testing and analysis of the domain mechanisms.

There is no reasonable method for testing the domain mechanisms on the 286. The activity takes place entirely within the processor chip. There are no external manifestations. If there were some diagnostic mode on the 286 that would send information about the domain isolation and mapping activities to pins on the edge of the chip, we could make a start towards testing the domain mechanisms. This would still be uncertain however, because there is no way to tell exactly what the domain mechanism would be using as it's inputs. The 286 has no such diagnostic mode, and Intel has no public plans of doing anything like this in the future.

Analysis of the inner workings of the 286 is also very difficult. First, the internals of any processor chip are enormously elaborate. The 286 is one of the densest and busiest processors being made. Second, the domain isolation section of the CPU is deeply intertwined with the CPUs processing section. Most of the major components of the 286 processing section are in some way connected to and controlled by the domain isolation mechanisms. Even if Intel were willing to sell a detailed, carefully documented picture of the screen used to make the chip, analysis would be close to futile.

The 286 is Too Integrated.

3.4.1.2 Invariable Intel did their best job at designing the 286. Now it is set in silicon, and will experience only minor changes. There are a number of characteristics of the 286 which are not optimal for our domain machine. The next two paragraphs detail a few of these shortcomings. Undoubtedly, any designer using the 286 will find other problems. Some things simply cannot be done. Other things must be done, even if they cause problems.

Possibly the most extreme case of this is the required use of privilege level 0. There are a number of things that can only be done at privilege level 0; flags that must be set and cleared, modes that can only be altered in that privilege level. Unfortunately, level 0 is very dangerous. There is an "undocumented" instruction called LOADALL, which allows any task running at level 0 to bypass the protection mechanisms. One may wish to simply avoid using level 0, but sometimes this is impossible.

Using other designs, such as the ones presented in the following sections, the domain machine might be changed to suit particular needs of the application.

If you don't like the way the 286 was designed, Too Bad.

3.4.1.3 Inextensable The 286 must be the only intelligence on the system bus to maintain the security properties. The memory access controls built into the 286 can only govern the 286 itself. Because of this the 286 must not be used with any chip able to perform Direct Memory Access (DMA), with advanced floating point chips that access memory, or with other CPUs performing slave processing. If the 286 is mixed with other intelligent chips, some other mechanism must be used to ensure data integrity.

Just as the processing power of the 286 cannot be extended, it is also impossible to extend the protection mechanisms. If we are concerned about the proper operation of the domain isolation, one reasonable approach is to build another sub-system that performs exactly the same functions, but is implemented slightly differently. This is called "functional redundancy". The idea is that the likelihood of the two different implementations having the same design error is very small. Unfortunately, this is impossible on the 286 because of the way the domain isolation is interwoven with the processing section.

The 286 must be the only processor in the system, so we call it Too Lonely.

3.4.1.4 Incomplete The 286 is both too elaborate in pursuit of flexibility and too restricting. The design of a general purpose microprocessor, especially one targeted for such a large market, requires a series of design compromises.

The I/O scheme built into the 286 is a mess. The only protection offered to the 256 I/O ports is the I/O Privilege Level (IOPL) register. The 286 forbids any process operating at less than the required level cannot execute the special I/O instructions. The two main problems with this scheme are:

1. The IOPL must be set while in level 0, which is a very dangerous place to be. Level 0 should be avoided because it can bypass all the protection mechanisms. Also, any other entry to Level 0 might change the IOPL.

2. Any I/O routine has read and write access to all ports and control over the interrupt structure. This will cause a considerable amount of extra work in verification.

The 286 instruction set is lacking desirable features for using memory mapped I/O. Block moves to and from a port are not possible when the I/O is mapped into the main 286 memory space.

The 286 also lacks modern bit manipulation instructions, which are useful in many applications besides I/O.

Each memory segment in the 286 has one of four attributes: Read Only, Read/Write, Execute Only, Read/Execute. It is not possible to grant Write Only, Write/Execute, or Read/Write/Execute status to a segment. Of these, Write Only is the most useful.

Our sample application contains a use for write only buffers. If the RED domain can only write to the message buffers, it would not matter if there were information left over in the buffers from a previous operation. There would be no requirement on the PROCESS domain to clear buffers before giving them to RED. This may not seem like a difficult requirement in our system because of it's simplicity, but if the same application were extended to multiple REDs and BLACKs with bidirectional flow (encryption/decryption), the requirement would become more difficult to prove. The ability to label some segments as Write Only can be very useful. There are probably applications that would also benefit from all combinations of attributes.

The Descriptor Tables are dynamically altered by the 286, even when the domains themselves are static. There are a number of flags set and cleared in the descriptors themselves by the 286 during normal operation. The location of the descriptor tables being used is also dynamically determined, as a special register must be loaded from Level 0 to point to the Global Descriptor Table (GDT). Because of these two quirks, it is difficult to put the descriptor tables into ROM and execute with a set of domains determined entirely by the hardware. In fact, all Level 0 code and any code which locates the descriptor tables in RAM must be scrutinized before analysis of the domain definitions can even begin.

The 286 has many fine features, but when it is missing the one an application needs we can only say that it has Too Few.

3.4.1.5 Incomprehensible Complexity in hardware or software reduces confidence in the system, because mistakes can be very subtle and yet tragic. The 286 is extremely complex. One amusing analogy compares the flexibility of the 286 to a car with two steering wheels - one for each front tire. It certainly is more flexible, but who can use such a thing? This is overly harsh because the flexibility of the 286 can be very useful, but it is a two edged sword.

There are two general problems caused by the 286's complexity. First, it is hard to program correctly. This will extend development time for a secure application, and will likely generate more errors that the code verification process can only hope to catch. Second, analysis of the domain structures and their interrelations are quite difficult. This undermines the

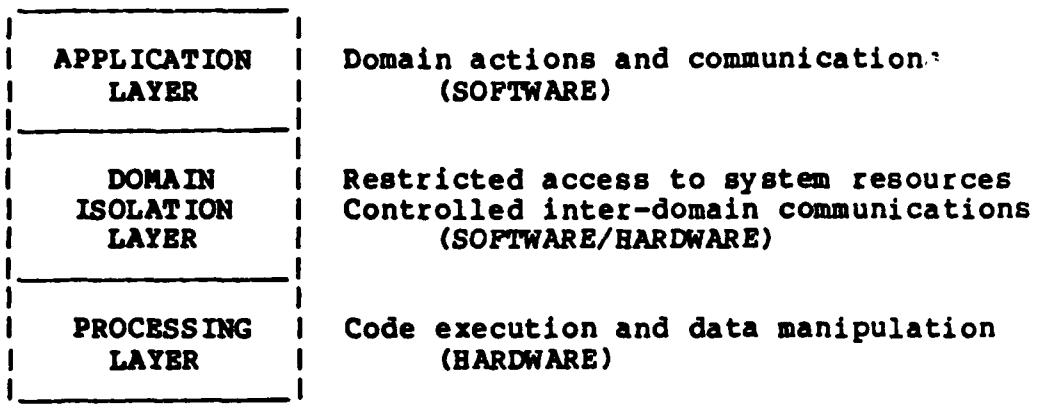


Figure 22. Domain Isolation Machine

The problem of verifying security properties of systems structured with a domain isolation layer can then be divided into the following steps. First, verify that the domain isolation mechanism actually implements domains which are isolated except for the communication paths explicitly specified. Second, decompose the overall security requirements on the system into requirements on application programs running in separate domains. Third, verify that the application programs satisfy their requirements.

In the remainder of this section we will discuss each

4.1 VERIFICATION OF DOMAIN ISOLATION

The essential property which must be proved about the domain isolation mechanism is that the only interdomain communications it allows are those which have been specified. In order to precisely state this property it is necessary first to have a means of specifying intended communications and second to have a way to determine the communications which the domain isolation mechanism allows. The communication map described below gives a method for specifying intended communications. Some definitions are then developed which allow a precise statement of the communication allowed by a domain isolation mechanism. The correctness property for domain isolation can then be stated in terms of the relation between the desired communication map and the communication allowed by the mechanism.

4. VERIFICATION

The purpose of security verification is to demonstrate that a system meets its security requirements. It has long been recognized that to make a credible demonstration possible, a divide and conquer approach is required. That is, the system is divided into parts which have distinct responsibilities with respect to enforcing the security requirements of the system as a whole. Some parts of the system may bear no responsibility for security enforcement, and hence not require any verification. The division of the system should isolate the substantive, difficult to verify properties into as small a portion of the system as possible.

One way to achieve this division of responsibility is to consider the system to be layered with each layer implementing an abstract machine which the next layer can use. This is the idea behind the security kernel approach to developing general purpose operating systems. The security kernel provides an abstract machine on which operating system utilities and application programs can run. Ideally the security kernel would enforce a security policy in such a way that programs using the abstract machine it creates bear no responsibility for security enforcement. In practice it turns out that some of the responsibility for security in a kernelized system resides in trusted processes with specialized requirements.

For small special purpose systems such as a KG controller there is no natural division into policy enforcer and application programs such that the application programs have no responsibility for enforcing security. However, if communication between application programs is completely unrestrained, verification will be extremely difficult because of the complex interaction possible between different parts of the system. The divide and conquer approach will not be available.

Thus a layered architecture is still appropriate for small special purpose systems, but the layering is different than for a general purpose operating system. An appropriate architecture should include a layer which implements separate virtual machines for different programs or program groups so that the communication between the virtual machines is explicitly specified. We refer to these virtual machines as domains, and we refer to the architectural layer which implements these machines as the domain isolation layer.*

* John Rushby has advocated a similar approach to building verified systems using a separation kernel to creates regimes which are analagous to our domains.

- The AMA protection mechanism can surround any group of processors or intelligent chips.
- The protection mechanism is separate and visible, so it is available for safety analysis.
- The protection mechanism itself can implemented with component and/or functional redundancy, to guard against hardware failure and design error.

complexity.

Only a few chips are required for the AMA protection mechanism. The chip count for any system is moderate.

An AMA design allows any number of processors to be used together, in many combinations. If more processing power is needed for an application, more CPUs can be added in. In this way, an AMA architecture can far exceed the performance of a 286, or any other single CPU.

The design of an AMA determines the style of buffer passing available. This is the main reason to choose dynamic domains over static domains. If the the speed advantage of passing buffers instead of copying is required for an application, the AMA is implemented with some alteration or selection possible in the Access Map.

The AMA seems to be the best of both worlds. In every case, AMA either is rated as good as the PPD or 286, or the AMA offers a range of implementations that will satisfy any application with extra money and effort. We must remember that PPD is one very simple case of an AMA. It is not thoroughly fair to compare AMAs in general with PPD architectures.

3.6 ARCHITECTURE CONCLUSIONS

The Intel iAPX-286 is a significant player in the security arena, but it does not provide the confidence desired for a state of the art Key Generator Controller. However, it is in many ways a useful chip, and it may be adequate for some applications. The 286 is a reasonable choice in applications other than the KGC, applications that don't require as high a level of confidence.

- The 286 is a high performance chip.
- The 286 transfers buffers between domains quickly.
- The 286 is available in off the shelf computer boards, and will soon be used in commercial systems.

Access Map Architecture (AMA) is one general class of designs which is superior to the 286. Using an AMA design has many advantages over using the 286:

- The design may utilize any processor (even the 286!).
- The design can be optimized for the application, increasing speed or decreasing complexity as desired.
- AMA designs are not inherently expensive, and in fact can be much cheaper because of their flexibility.

Domain Latch, the Processor Block must reset all internal resources to some known state. In a simple system, this amounts to nothing more than saving the current contents of the internal CPU registers and then clearing these registers. The requirement is that anything within the processor block that contains state, such as high speed cache RAM or slave processors, must be set to a known state. Many chips will only need their RESET pins toggled to go through this zeroization process.

3.5.4 AMA vs. 286

The following chart summarizes the advantages and disadvantages to the two systems we have discussed so far.

PPD vs. 286 vs. AMA

	PPD	286	AMA
Number of Domains	Hardware	Software	Either
Domain Definition	Hardware	Software	Either
Safety Granularity	Fine	Course	Fine
Software Complexity	Low	High	Medium
Chip Count	# of Domains	Moderate	Moderate
Processing Load	Distributed	System Load	Distributed
Data Flow	Buffer Copy (slow)	Domain Modify (fast)	Either

The number of domains and the domain definitions can be determined either by hardware or software on an AMA architecture, depending on the implementation. Probably the most common implementation would be a mix of these two. The crucial section is the Access Map. This map can be implemented with RAM, allowing total flexibility as on the 286, implemented with an 8 bit latch manipulating one portion of the Access Map structure, or implemented entirely in ROM for the solidity of the PPD architecture.

An AMA has a Fine safety granularity, because the protection mechanisms are so accessible and distributed. Any spot where a single failure can compromise the system, component or functional redundancy may be used to increase the safety.

The AMA is listed as Medium software complexity because of the possible manipulation of the Access Map. Even using dynamic domains, an AMA architecture is not as complex as the 286, because the mapping mechanism is straightforward and direct. If the dynamic domains are not used, the software remains at a Low

3.5.3.1 Multiple Domain A small, reasonably simple addition to the Functional Block Diagram turns the generic PPD architecture into a generic Access Map Architecture, supporting multiple domains in a single processor block.

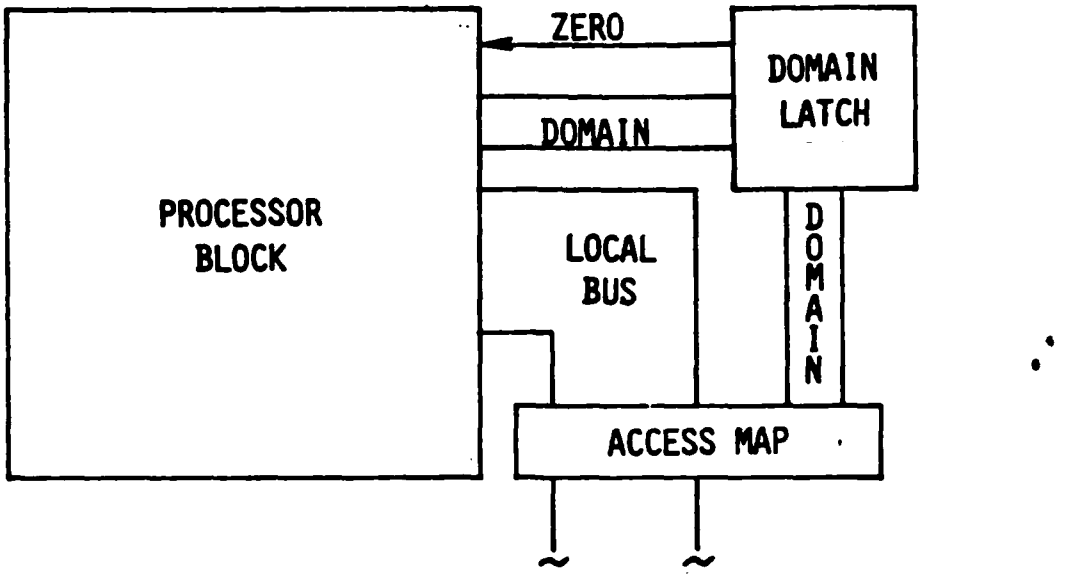


Figure 21. Generic Multiple Domain AMA

There are only two new functions needed to transform our PPD Access Map Architecture into a multiple domain machine.

1. Zero Processor Block

   Any resources within the Processor Block must be reset to some known state. This usually consists of clearing the CPU registers.

2. Map Selection

   The Access Map contains a map for every domain. The Access Map must select the map that applies to the current domain.

The Functional Block Diagram for the multiple domain Access Map Architecture above shows a Domain Latch which performs these two functions. The Domain Latch remembers the current domain, and provides the Access Map with this information. If the Access Map is implemented by a ROM, as above, the output lines from the Domain Latch are simply treated as more inputs to the ROM.

The Domain Latch also sends a Zero signal to the Processor Block whenever the domain changes. This is a new requirement on the Processor Block. Whenever a Zero signal comes from the

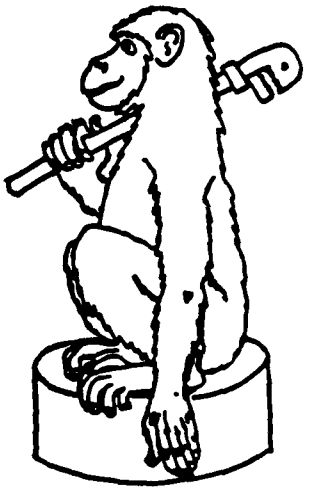
3.5.3 Access Map Architecture (AMA)

The Processor Per Domain (PPD) architecture is one very simple implementation of a more general design approach we call the Access Map Architecture (AMA). The PPD provides rock solid domain isolation without complication by restricting the the resources that can be physically accessed by each processor.

The isolation may be physical, as above, or it may be logical. We can reimplement the dual-port RAM interface using a logical Access Map instead of a physical Access Map. The processor in the ENCRYPT domain is given physical access to the system bus, but it's use of that bus is monitored. If the ENCRYPT domain tries to access any resource except the allowed dual-port RAM, the system is halted.

The logical Access Map is functionally identical to the physical Access Map. The question is, can this logical mechanism be made as simple and trustworthy as the physical isolation? The answer is Yes! The simplest Access Map requires only a single, off the shelf chip.

The Access Map can be implemented by one PROM. For simplicity in explanation, let's assume for a moment that the only system resources available are the CPU registers and 64K words of memory. When the LOCAL BUS goes through the Access Map, the 16 address lines are connected in parallel to the 16 (address) inputs of a 64K x 1 bit PROM. The one bit output of the PROM indicates whether the memory access is a legal one or not. If the output of the PROM is ever low, the system clock is stopped, killing the whole system. The Access Map never interferes under normal operation. The Access Map sits by and observes, making sure that nothing illegal is attempted. If an anomaly is detected, the Access Map throws a monkey wrench into the works.



This example is simplified, but the same procedure can be followed to monitor more signal lines. The Read/Write, Data/Code, or Memory/I-O lines can be included as inputs to the PROM. The memory can be monitored in larger chunks by ignoring the lower address lines. Two PROMs can be used if there is a need to monitor more than 16 bus lines.

In summary, the PPD is superior for verification and safety reasons, but due to performance and design restrictions PPD cannot support some applications which the 286 can support.

3.5.2.1 Generalized PPD The processors in each domain under PPD are physically isolated from most of the resources. Examination of the board layout quickly reveals which resources are available to each Processor Block. The RED and BLACK domains are isolated from all machine resources except for the single bit SIO connection to the PROCESS domain. It is physically impossible for RED and BLACK to communicate directly.

The diagram below shows a Functional Block Diagram of a generic PPD domain.

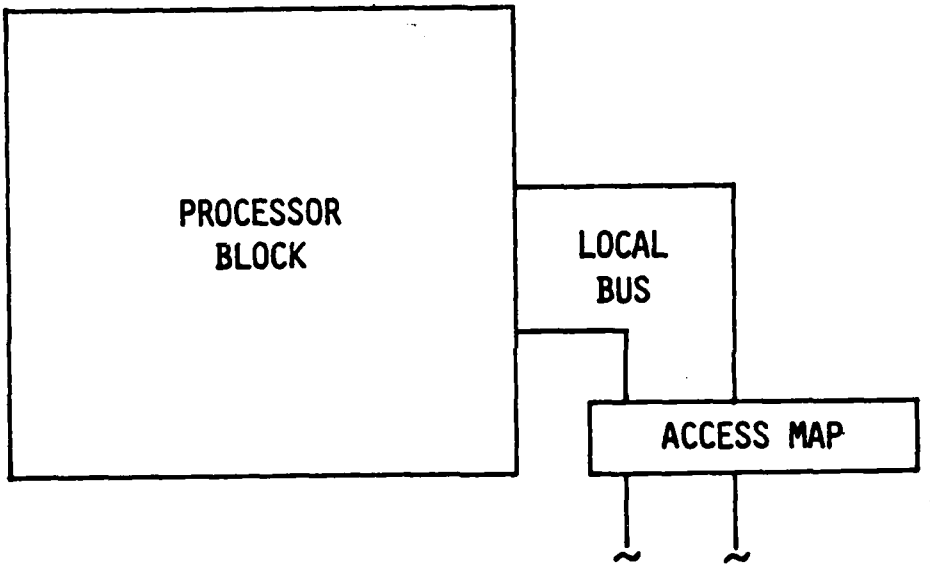


Figure 20. Generic PPD Architecture

Let's compare Figures 19 and 20:
In Figure 20, the Access Map controls which system resources are available to the Processor Block. The Access Map represents the isolation method used on a particular domain. The processor, RAM, and ROM in the I/O Domains from Figure 19 implement the Processor Block shown in Figure 20. The SIO chips (and the lack of any other bus interface) implement the Access Map.

The Access Map can be implemented in many ways. The simplest is to disable control and address lines leaving the domain. Imagine if 16 address lines go into the Access Map, but only the lower four come out - the Processor Block will only share the lowest 16 bytes of memory with other domains. The dual port RAM method is an extension of this.

should be fine for KGCs, and verification of these static domains is much more convincing.

The safety granularity of the 286 is listed as Course because almost all of the protection mechanism is included in the CPU. Strictly speaking, the memory containing the Descriptor Tables is also a component of the protection mechanism, but it is very difficult to even double up on the crucial chips (component redundancy). Component redundancy on the entire protection mechanism dramatically increases the safety of the system, because "bit-hits" can be detected. Unfortunately, this requires duplication of the entire system when using the 286. The 286 is especially vulnerable to this sort of error, because once a descriptor is loaded into the 286 registers, it remains unchanged for an extended period. The results of a single corrupted bit on a single memory access can be tremendous.

The PPD design has a Fine safety granularity, because the protection mechanisms are so accessible and distributed. Any spot where a single failure can compromise the system, component or functional redundancy may be used to increase the safety.

The term "functional redundancy" refers to the reimplementation of some system component, so that the function is performed in two different ways. This has all the advantages of component redundancy, and also provides extra protection against design error. Functional redundancy is possible for crucial portions of a PPD design (such as the dual port RAM arbiter), but is totally impossible for the crucial portions of the 286.

The software complexity of the two designs has been discussed extensively. Listing them as Low and High is generous.

The PPD is not suitable for a system with a large number of domains. The chip count in a PPD system is not unreasonably high. The microprocessors can be easily replaced by microcomputers, eliminating a number of support chips. This would decrease the chip count for a three domain PPD system to around twice that for the 286. In a system with many domains, however, the 286 design remains unchanged while the PPD design increases many times over.

The PPD design is inherently suited to distributed processing, sharing the workload between a number of processors. It is unreasonably difficult to use multiple 286s in a single system. (Unless they are used as the CPUs in a PPD Processing Block!)

The main performance problem in the PPD design is the flow of data between domains. In the simplest PPD designs, the only method available to transfer data from one domain to another is the relatively slow block copy method. This is fine for a KGC running at a low transmission rate such as 19.2 Kb, but too slow for higher speeds. The 286 is descriptor based, and can transfer buffers by changing descriptors.

in dual port RAM, and then resets the semaphore.

The PROCESS domain then retrieves the encrypted message body, and sends the entire message out through the SIO chips to the BLACK domain.

The BLACK domain buffers up the message sent by the PROCESS port on, and then sends the message out through the other port on it's SIO chip. The BLACK Processing Block handles the low level protocol as necessary.

Comparing this with the section on Using the 286 as a Domain Machine, one can't help but be impressed by the elegant simplicity of this design. There are no problems with initialization, no tricky maneuvering to keep the buffers straight. This level of clarity affords a marked decrease in the cost and effort of verification, and a corresponding increase in security confidence.

3.5.2 PPD vs. 286

The following chart summarizes the advantages and disadvantages to the two systems we have discussed so far.

PPD vs. 286

	PPD	286
Number of Domains	Hardware	Software
Domain Definition	Hardware	Software
Safety Granularity	Fine	Course
Software Complexity	Low	High
Chip Count	# of Domains	Moderate
Processing Load	Distributed	System Load
Data Flow	Buffer Copy (slow)	Domain Modify (fast)

The number of domains and the domain definitions under PPD are determined entirely by the hardware. In the 286 these are both determined by the contents of the descriptor tables. If the application requires dynamic domain creation and destruction, as in a general purpose operating system, the PPD architecture will be inadequate. It is, however, extremely difficult to prove the correctness of a system using dynamic domains. As was mentioned in the 286 Sample Application walkthrough, the initialization of domains in the 286 can also be very tricky. Static domains

1. The hardware is static. The domain definition can be determined without examining the code. The domains cannot be altered by an "updated" version of the software.

2. The hardware isolation mechanisms are accessible to validation testing. The isolation mechanisms are not hidden inside some other chip.

3. The isolation mechanisms can be made simple and obvious. Simpler designs are resistant to hardware failure. Obvious designs are resistant to design error.

Two domain isolation methods are shown here. Neither one is inherently superior. We show both to clarify the concepts involved. The two most common isolation methods are demonstrated here, but many other methods can be imagined.

The I/O blocks are isolated by an SIO channel. The RED and BLACK Processor Blocks share nothing with the ENCRYPT domain, and the only connection to the PROCESS domain is a single SIO channel. If buffer chips are only included in one direction, then RED can only send over the channel (Write Only) and BLACK can only receive (Read Only).

The ENCRYPT block is isolated by dual-port RAM. The ENCRYPT domain and the PROCESS domain share this block of RAM. The ENCRYPT Processing Block shares no other resources with any other domain.

3.5.1.1 Using PPD as a Domain Machine The PPD is so simple that a short walkthrough of our Sample Application as it would be implemented should adequately describe the use of the PPD as a domain machine.

On power-up, each domain comes alive under it's own code. Each domain waits until it has something to do. Naturally, there is not much activity until a message arrives.

The message arrives, one byte at a time, into one port of the RED's SIO chip. The RED Processor Block performs the low level protocols as necessary, and buffers up the entire message. When the message is complete, RED sends the entire message to PROCESS's SIO chip.

The PROCESS domain buffers up the message and determines where the beginning of the message body is. PROCESS then copies the body of the message into the dual port RAM that the PROCESS domain shares with the ENCRYPT domain. PROCESS then sets a semaphore in dual port RAM to indicate a request for encryption and waits for ENCRYPT to respond.

The ENCRYPT domain monitors the semaphore. When it is set, ENCRYPT utilizes the Key Generator to encrypt the buffer residing

3.5.1 Processor Per Domain (PPD)

This architecture is well described by it's name - Processor Per Domain (PPD). Each domain in the high level software design is supported by a separate Processor Block. A Processor Block is some combination of chips with enough intelligence to perform the task required by that domain. Typically, a Processor Block will consist of a CPU, some ROM and RAM, and I/O support chips. A rough block diagram is shown below.

The technique for creating a PPD architecture is easy to see. The central portion of this figure is very similar to the figure used earlier in the implementation of the 286 system, "KGC Hardware Design Using a 286". The difference is simply stated: A Processor Block has been added for every domain.

There are four Processor Blocks in this architecture, one for each domain. The central block, resembling the 286 architecture, executes the PROCESS domain. A block containing a CPU, some RAM and ROM, and some Serial I/O (SIO) support chips has been added on each side. These two Processor Blocks execute the RED and BLACK domains. There is a fourth Processor Block separated from the PROCESS block by dual ported RAM. This block executes the ENCRYPT domain.

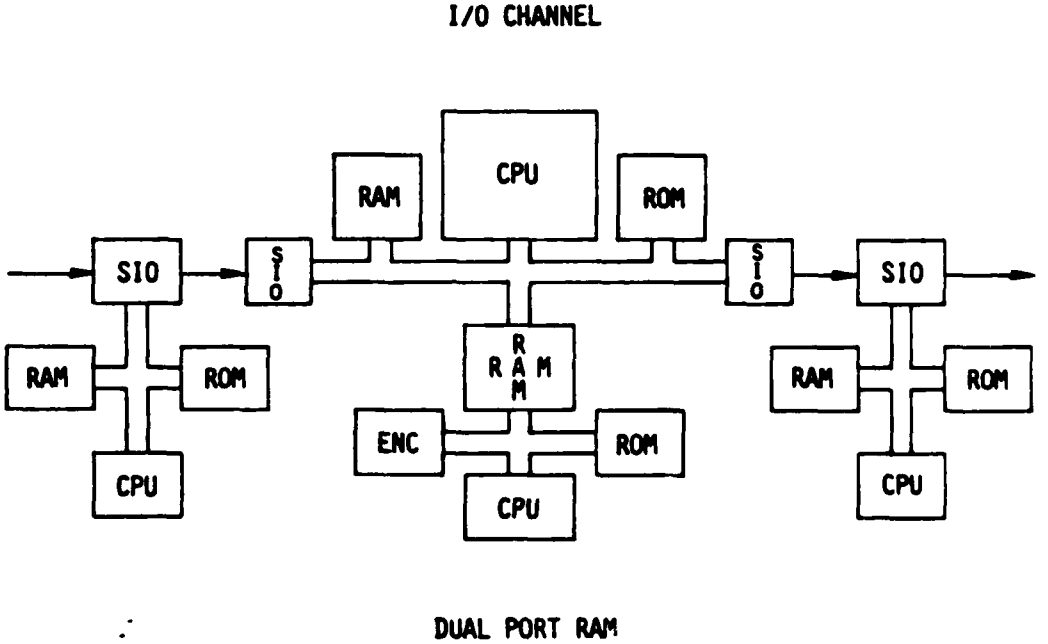


Figure 19. Processor Per Domain - Isolation Methods

The hardware provides domain isolation. The domains are defined by the resources physically shared by the Processor Blocks. This is a significant advantage for several reasons:

built into the chip. Unfortunately, the infrequently documented instruction called LOADALL can bypass all the protection mechanisms from Level 0. This level violates the concept of a domain machine, and will be a glaring exception to all other rules.

The 286 supports another feature called Requested Privilege Level (RPL). The full implications of this feature are as convoluted as it's implementation. Roughly speaking, the RPL allows a process operating at one privilege level to temporarily operate at a lower privilege level. It is tempting to use this feature to dodge verification of highly privileged code. One task can wear many hats, depending upon some variety of variables. Unfortunately, the argument rapidly becomes insubstantial because it is difficult to trace when and where the process is wearing which 'hat'. Static analyses such as those presented in the verification chapter below are rendered useless.

Because of the secretive nature of these little tricks, we may have missed a few. One can be certain that inventive programmers will find and exploit any of the little loopholes that have been missed in this paper. This is perhaps the most damaging statement we make: The 286 is so complex that after studying the security properties of this CPU for over a year, we cannot state with certainty that there aren't more problems yet to be found.

With all the little things the programmers must watch out for, the chip is just Too Tricky.

3.5 ALTERNATIVE DOMAIN MACHINE ARCHITECTURES

This section will present alternatives to the 286. We will apply strict requirements to these architectures. To be seriously considered as a replacement, alternative architectures must solve the problems identified in the 286 without significantly increasing the chip count or cost. The architecture also must be able to equal or surpass the performance of the 286.

We will present two complete architectures here. The first, called Processor Per Domain (PPD), is extremely simple and secure. Although it is best suited to a limited set of applications, it's function and design are so clear and indisputable that the PPD design is very impressive. The PPD design is then abstracted to show the general principal. This same principal is then altered to support multiple domains on a block of processors. A design is then presented that easily and cheaply implements this new concept, called the Access Map Architecture (AMA). Both the PPD and AMA designs are compared to the 286.

solidity of the domain machine, our motivation for using the 286. These two problems translate to an increase in cost and a decrease in confidence.

Figure 18 only shows the main components involved in a task switch. This is just one aspect of the interrelationship and structure of domains in the system. These are the basis of the domain isolation, and so require a perfectly clear comprehension. The task switch pictured here is actually one of the easier domain characteristics to map, but this graphically illustrates the difficulty of one highly important step in verification.

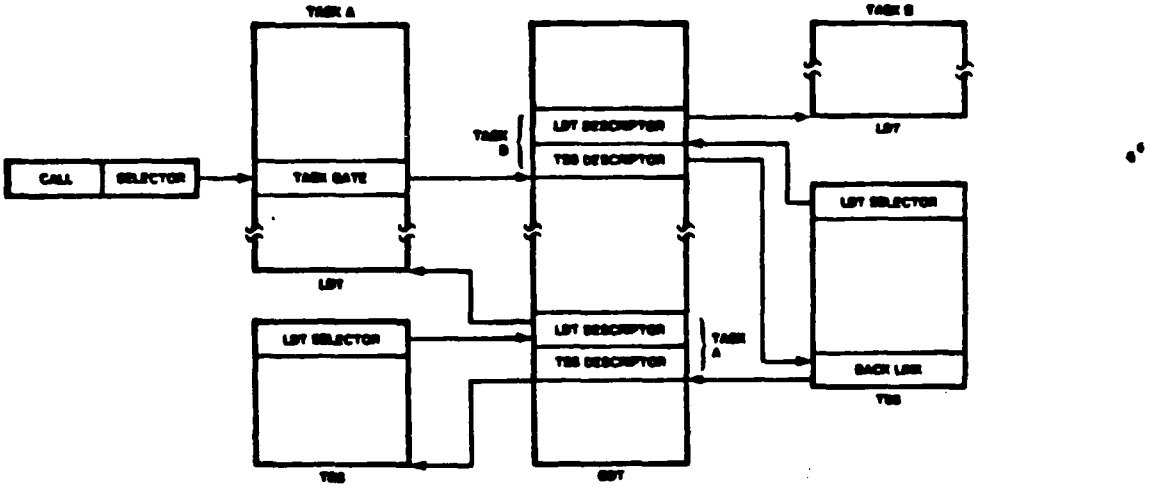


Figure 18. The Complexity Involved in a Task Switch

Any security conscious application using the 286 will have to forbid the use of some of the features available on this CPU. How can anyone be sure that nothing was overlooked? How can a project manager enforce the absence of certain instructions with the certainty required?

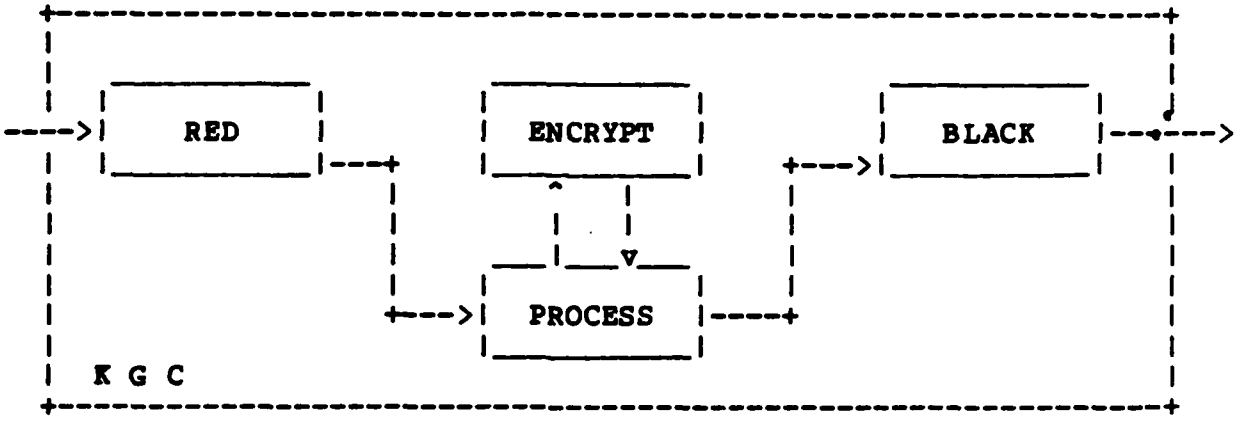
As mentioned before, there are problems associated with using the I/O ports, and restrictions on the use of intelligent support chips or multiple processors. These requirements are fairly straightforward, assuming that the security assurance team has veto power over the hardware designers. (A rather large assumption!)

The software issues are subtler. As with the hardware design, an active and highly expert security assurance team may be able to catch all mistakes. Nonetheless, we consider this to be another major failing of the 286.

The use of Level 0 is recommended by Intel frequently in their otherwise excellent literature for the 286. In a two privilege level system, Intel recommends using levels 0 and 3. Of course, Level 0 must be used for certain operations, this is

4.1.1 Communication Maps

The explicitly allowed communication between domains can be represented by a communication map. This map is a directed graph with domains for nodes and communication paths for edges. For example, consider a KG controller for encrypting messages entered at a red interface and sending the encrypted message out a black interface. In addition, a bypass is included for control information. The communication map is shown in Figure 23.



Figur 23. KGC Communications Map

A directed graph can also be shown as a matrix. The communication map can be represented by a matrix whose rows and columns are labelled by the domains. A one in position (a,b) indicates communication is allowed from domain a to domain b, and a zero indicates no communication is allowed. The matrix representation of the communication map in Figure 23 is given by the MATRIX in Figure 24. Since this will typically be a sparse matrix, it may also be convenient to simply list the permissible communications using the notation A -> B to indicate communication from A to B is allowed. For example, the communication map shown above is given by the LIST in Figure 24.

MATRIX

	Red	Encrypt	Process	Black
Red	1	0	1	0
Encrypt	0	1	1	0
Process	0	1	1	1
Black	0	0	0	1

LIST

```
Red ----------> Process
Process-------> Encryption
Process-------> Black
Encryption----> Process
```

Figure 24. MATRIX and LIST Representations of a Communications Map

The example above is typical in that there are domains (the interfaces) such that information can flow from one to the other but only by going through intermediate domains. Software in the intermediate domains can control what information is passed. The black interface domain is relatively isolated from the red interface domain in the sense that information can not be passed directly from the red interface domain to the black interface domain. Domains B is absolutely isolated from domain A if there is no route for information to flow from A to B. These ideas are formalized below.

4.1.2 Formalization of Domain Isolation

In order to state what it means for a domain isolation mechanism to implement a particular communication map, a few definitions are needed. We consider the abstract machine created by the domain isolation mechanism as a state machine. A state machine M can be characterized as a four-tuple, M = (V,I,O,init) where V is a set of variables, I a set of input ports, O a set of operations, and init is the initial state of the machine. For each variable v in V, there is an associated set of possible values called the type of v and denoted T(v). A state of the machine is a function s on V such that for all v in V, s(v) is in T(v). Like variables, input ports have associated types indicating the set of possible inputs to that port. An input value is given by a function c on I such that c(i) is in T(i) for all i in I. Let C stand for the set of all such functions. An operator o in O is a function from SxC to S. This functions determines the

new state from the old state and the input value.

We have not explicitly mentioned either outputs or the determination of the next operation. To keep the notation from becoming more cumbersome than it already is, we will assume that among the variables are histories for each output port. Also, we will assume that the state includes sufficient information to determine the next operation, denoted next(s).

We can now define a domain. In order to consider the sequencing of functions it is convenient to consider a next operation to be defined for the domain A for any state of M, although this operation will only coincide with the next operation for M when next(s) is an operation of A. A domain A of M is a machine together with a next operation function for A satisfying the following requirements:

1. The variables of A are variables of M.

2. The input ports of A are input ports of M.

3. The operations of A are restrictions of operations of M which do not depend on inputs or variables outside of A and don't change variables outside of A.

4. The initial state for A is the initial state of M restricted to variables in A.

5. When the next operation of M corresponds to an operation of A, then it equals the next operation for A.

Formally the definition can be stated as:

A domain A of M is a state machine $A = (V_A, I_A, O_A, init)$ and a function $next_A$ from the set of states of M to O_A such that:

1. $V_A \subseteq V$

2. $I_A \subseteq I$

3. For every o in O_A, there is an o' in O such that for any state s of A, input value c of A, state s' of M with $s(v) = s'(v)$ for v in V_A, input value c' of M with $c(i) = c'(i)$ for i in I_A: $o'(s',c')(v) = o(s,c)(v)$ for v in A and $o'(s',c')(v) = s(v)$ for v not in A. These conditions determine a unique o' and by identifying o with o' we may simply say the operations of A are operations of M

4. $init_A(v) = init(v)$ for v in A

5. $next_A(s) = next(s)$ if next(s) is in A.

M is said to be divided into domains A1 ... An if

1. $V = V_{A1} U \cdots U V_{An}$
2. $I = I_{A1} U \cdots U I_{An}$
3. $O = O_{A1} U \ldots U O_{An}$
4. For all j, k with $j \neq k$, $O_{Aj} \wedge O_{Ak} = \{\}$.

If A is a domain of M, the state of A corresponding to a state of M is the restriction of the state of M to the variables in A. With these definitions we can now state what it means for domains to be isolated and then give a correctness property for a domain mechanism. Domain B is said to be isolated from domain A relative to a set of domains D if for any starting state s and any sequence of operations from domains in D and inputs from D, the resulting state of B and the next operation for B are the same as for the sequence with operations from A removed. B is said to be locally isolated from A if B is isolated from A relative to {A,B}. B is said to be absolutely isolated from A if B is isolated from A relative to the set of all domains of M. The correctness property for a domain isolation mechanism can now be stated.

The domain mechanism M is correct with respect to the communication map CM if: For every pair of domains A,B of M, B is locally isolated from A if and only if A -> B is not in CM.

The definition above gives a model of domain isolation. Since the application programs will typically want to rely on information about the objects used for communicating between domains, a natural next step in modelling domain isolation would be to model domain communication in terms of communication with respect to particular variables in the state machine.

In practice the approach to verifying the correctness of a domain isolation mechanism depends on the architecture used for the mechanism. If the access map architecture is used, most of the domain mechanism is implemented in hardware. Although hardware verification has not been extensively studied, the properties of the hardware which are used for maintaining domain isolation are simple enough that inspection would yield a degree of credibility greater than any currently possible software verification. The access map would have to be checked to determine that it corresponds to the desired communication map, but this comparison would be significantly easier than software verification.

For the IAPX 286 architecture, some software would definitely be required to implement the domain machine. Current approaches to software verification could be used. In fact the verification would be a form of flow analysis. However, if the verification of a domain machine implemented on the IAPX 286 is to be done to an equivalent degree of detail to that for the access map architecture, all the software for the domain machine would have to be verified at the code level. This is a formidable task.

4.1.3 Verification of Application Programs

The verification of a secure system built on a trusted domain isolation mechanism can make important use of the controlled communication provided by the domain isolation mechanism. The security requirements for the system form a system-specific security policy which can be formalized as a security model.

Because the domain isolation mechanism allows pieces of the system to be divided into programs running in separate domains, the policy can be decomposed into requirements for the software in the various domains. This decomposition must be done in such a way that the totality of requirements for the various domains implies the system security policy. A proof that the requirements for individual domains implies the system policy must be part of the system verification. The decomposition may proceed in such a way that there are no security requirements on the software in some domains. This software is called untrusted software. Software in domains where there are some security requirements is called trusted software, and its requirements must be verified.

It is useful to return to the example described by the KGC Communications Map (Figure 23). The overall security policy to be demonstrated is that no information is passed from the RED input to the BLACK output except ciphertext and specified control information. This system requirement will be enforced if the ENCRYPT domain sends only ciphertext to the PROCESS domain and the PROCESS domain sends only information received from the ENCRYPT domain and the specified control information to the BLACK domain. Notice that the software in the RED and BLACK domains is untrusted and hence need not be verified.

4.2 VERIFICATION CONCLUSIONS

The use of a domain isolation mechanism provides a convenient factoring of system verification. In fact if the domains are used effectively, significant amounts of software will be untrusted and not need to be verified. The domain mechanism itself must, of course, be verified. This is a consideration in the choice of architecture for the domain isolation mechanism. The simpler the mechanism the easier the verification. If a limited set of static domains will suffice, a simple hardware

mechanism may be used to create the domain machine. The analysis of this mechanism will be much easier than verification of software required for more complex domain machines. An area which needs further study is a technique for formally stating the system requirements and domain requirements of typical systems which might use a domain mechanism. This is necessary to allow formal proof that the system requirements have been properly decomposed; that is, the domain requirements imply the system requirements.

5. SAFETY ANALYSIS

As discussed in the previous section, verification techniques are used to demonstrate that a system operates in a manner consistent with its security requirements. However, one of the assumptions of the verification is the absence of hardware failures. That is, the verification demonstrates that the system is secure when functioning as designed. Hardware failures alter the behavior of the system. Generally the system will not exhibit its originally intended functionality in the presence of hardware failures. For secure systems it would be desirable to show that the system will not allow violations of its security requirements even in the face of failures. In practice it will generally only be possible to show that the probability of a combination of failures which would permit security violations is acceptably low. In this section we will present an approach to determining an upper bound for the probability of failures resulting in a security violation. We will also give an example illustrating the application of this technique. However, further work will certainly be needed to determine the practicality and usefulness of this technique in the development of real systems.

5.1 Overview of Safety Analysis

The ultimate goal of verification and safety analysis is to allow the development of systems which can be relied on to meet their security requirements throughout their lifetime with a high degree of probability. There are two types of problems which can allow the system to violate its security requirements. First, there may be errors in the design and implementation of the software or hardware. Verification addresses this problem. Of course, even if formal code verification is used, there are still some aspects of the systems of operation which cannot be formally verified with the current state-of-the-art. For example, the compiler and hardware are not usually formally verified. One technique which can be used to decrease the impact of errors in the design and implementation of the system is to use functional redundancy for critical system elements. That is, the elements are independently designed and implemented to perform the same function, both are used in the system, and their operation is compared to determine any inconsistencies. While verification and functional redundancy can be used to decrease the likelihood of residual design and implementation, it is not possible currently to quantify the probability of errors in the implemented system.

The safety analysis described here is concerned with the second type of problem, failures which occur after the system is implemented. There are techniques for lessening the probability of such failures or at least reducing the impact of the failures on security. These techniques include redundancy, special robust construction techniques, rigorous testing under extreme conditions and simplicity of design. However, in order to determine

which of these techniques should be applied, to what degree, and to which parts of the system, it is necessary to have a way of quantifying the probability of a security related failure in the system. That is the purpose of safety analysis.

The problem of assessing the impact of hardware failures has been addressed by security fault analysis and by general reliability analysis. The safety analysis described in this section is closely related to both techniques. The distinction between the approach discussed here and a typical security fault analysis lies in the complexity of the circuits which are being analyzed. A system using currently available microprocessor technology is simply too complex to allow analysis of the consequences of the failures at the level of discrete devices. Although information about the probability of failure within a given period of time can be obtained for integrated circuits, this information typically says little about the exact nature of the failure. Thus the system must be analyzed by a technique which can be applied at a larger level of granularity than the discrete device level which has been used in security fault analysis. Safety analysis differs from general reliability analysis in that certain failure modes are of paramount interest. The failures of interest are those which allow some violation of the system security requirements.

5.1.1 Definition of Safe Systems

The first step in developing a technique to analyze safety is the definition of a safe system. We can consider a hierarchy of information compromise properties for communication systems. Ranked in order of decreasing acceptability these are:

1. Always operates without information compromise.
2. Shuts down or ceases communication if it cannot operate without information compromise.
3. If compromised, notification is given.
4. Can be compromised without notification.

In practice there will be a non-zero probability of a combination of faults which results in compromise without notification. Thus the definition of safe system must be stated in terms of a maximum probability that the system violates its security requirements. Moreover, since we are dealing with safety analysis for hardware failures the calculation of probability must be in terms of requirements for the hardware. For a given system there will be several aspects of the operation of the hardware which are important for security. The types of hardware failure which are important are those which could result in a security compromise of types 3 or 4 in the list above. As noted in the introduction some types of hardware failure are more

likely than others to result in significant information compromise. Thus it may be appropriate to require that the hardware meet different requirements with different probabilities. The required probability of meeting specific requirements must be stated in terms of a period of operation. That is because the probability of any possible random event occurring within a given time interval approaches certainty as the time interval approaches infinity. We can now state our definition of a safe system.

> A system is safe with respect to requirement R with probability P for time T if during operation for a period of length T the probability that the system does not violate R at any time is greater than or equal to P.*

5.1.2 Model of Hardware Faults

In order to analyze safety it is necessary to have a model of the the evolution of a system as faults occur. The approach used here is the state machine approach. With a given hardware status the system can be modelled as a state machine. That is, it can be represented by a set of variables, each of which can take on values from some set. The current state of the machine is determined by the values of the variables. The operations on the machine would be represented by transitions in the machine state. The state machine approach forms the basis for modelling systems in some formal specification languages such as SPECIAL and Ina Jo. For our purposes, we must take into account the possibility that a fault or intentional reconfiguration will occur which will change the machine to some different machine. The ordered pair consisting of the particular machine which currently exists and its current state forms the overall system state. Safety analysis then requires that machines be categorized into "secure" and "insecure" with respect to the requirement under consideration. Once this is done, the next step is to calculate the probability of reaching the insecure region.

Figure 25 shows this model of safe systems in its most general form. Each circle represents a particular machine. The arrows represent possible migration paths from one machine to another. With each such path there will be an associated probability representing the likelihood that the transition represented by the arrow occurs. In some instances the probability of moving to another machine may be dependent on the state of the current machine. For example, the system may be built so that certain states automatically trigger a shutdown. In this

---

* The definition could also be stated in terms of a minimum requirement for the mean time M until requirement R is violated. For a typical system the two forms of definition would be interchangeable in the sense that for a given value of M there would be values of P and T such that the two definitions are equivalent.

case, the machine would move to the "shutdown" machine. The dependence on the current state of the machine is represented by the broken lines dividing some machines into multiple state regions.

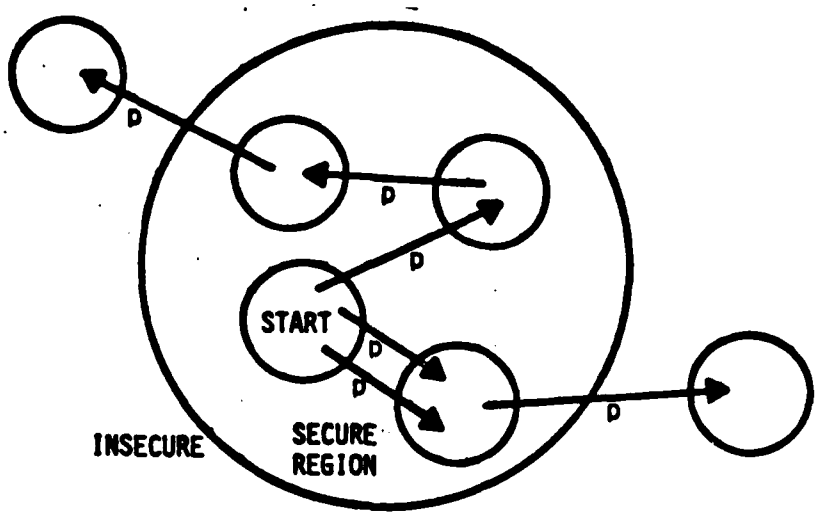


Figure 25. Model for Quantifying Safety

The correct determination of secure and insecure machines depends on the absence of design and implementation errors affecting security. Finally, the probability of occurrence of a particular kind of failure in a component is frequently difficult to determine. Thus, in general it will be necessary to overstate the probability of reaching an insecure machine. That is, precise calculation of the probability of reaching an insecure state will not generally be possible. However, if the analysis yields an upper bound for the probability which falls in the acceptable range, the system will have been shown to be safe for the particular requirement.

5.2 Application of Markov Chains

The evolution of machines in the presence of hardware faults can be considered a stochastic process. A stochastic process has a set of possible states which the process can be in at any given time. Stochastic processes can be divided into continuous stochastic processes and discrete stochastic processes depending on whether the process is observable at any time after some initial time or observable only at specified times. Although a hardware fault might occur at any time, it is reasonable to only consider the state of the system at discrete times. The interval between times can be made small enough that the behavior of the system only matters at these discrete times. For example, the time interval could correspond to a clock cycle although a larger time interval may simplify calculations. The advantage of considering the transition from one machine to another as a discrete

stochastic process is that it much easier to calculate properties of a discrete stochastic process. A discrete stochastic process is characterized by the set of possible states and probabilities for each observation time indicating the probability that the process will be in a particular state at the next observation.

In general the probability of being in a given state i at time t + 1 may depend on the entire history of the process up through time t. However, an important special case of stochastic processes is the Markov Chain. A Markov Chain is a stochastic process in which the probability that the process is in state i at time t + 1 depends only on the state of the process at time t. The sequence of system states in our analysis of safe systems is a Markov Chain. The significance of viewing the sequence of system states as a Markov Chain is that there are well known methods for calculating the probability that a Markov Chain reaches a particular state in a given amount of time. This will allow the computation of the probability of reaching an insecure state, and hence determination of whether the system is safe.*

Since the probability that a Markov Chain is in state j at time t+1 depends only on the state i of the system at time t, the probabilities can be arranged in a matrix. Let p(i,j) represent the probability that the next state of the Markov Chain is j given that the current state is i. If there are n states, these probabilities form an nxn matrix called the transition matrix. (See Figure 26.) The individual probabilities are called transition probabilities.

* There is precedent for this approach in a somewhat different context. The observation that Markov Chain analysis could be applied to the measure of probabilities of types of system failures was made on the Software Implemented Fault Tolerant computer project. (See Wensley et. al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," Proceedings of the IEEE, Vol. 66, No. 10, October 1978.) In that project the focus was on correct computation of flight control functions rather than secure operation.

States $s_1, \ldots, s_n$

Transition probabilities p_{ij}

Transition Matrix

$$\begin{bmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & & \vdots \\ p_{n1} & & p_{nn} \end{bmatrix}$$

Figure 26. Markov Process

In order to use Markov Chain analysis, it is necessary to determine what the states of the Markov Chain will be. Of course, we could consider each possible system state to represent a different state for the Markov Chain. This would mean that the state of the Markov Chain would consist of an ordered pair with the first element a machine resulting from the initial machine changed by some sequence of failures and reconfigurations, and the second element the execution state of that machine. The problem with this selection of states is that the number of possible states is too great to permit analysis.

As a result the states must be considered at a larger level of granularity. Instead of considering the exact nature of the machine resulting from failures, it is sufficient to consider the current machine to be characterized by the components which have failed. The execution states can also be grouped together. Typically all that will be important about the current execution state is the configuration, for example whether the system is shutdown, has its external communication disabled, or is in a full functioning mode. Thus the states of the Markov Chain can be considered as ordered pairs of the form (failure set, configuration). The choice of components is important in the practical application of this approach. Since any failure of a component must be considered as the worst possible failure from the point of view of security, components which are too large will result in an overly pessimistic calculation of the probability of reaching an insecure machine. In order to make the calculation of the transition probabilities easier, it is desirable to pick components whose failure is mutually independent.

Even with this aggregation of system states into states for the Markov Chain analysis there still would be so many states that the computations required for the determination of the safety of a system would be very difficult. Fortunately, in particular instances it is possible to group the states even further. This is best illustrated with an example.

5.3 Safety Analysis Example

Consider the system shown in Figure 27. In this example system a single CPU accesses memory through an access map which divides the memory into domains. The domain changer records the current domain and resets the CPU when the domain is changed. For robustness the domain change and access map are duplicated. If the CPU tries to access memory outside of the current domain, the access maps signal a violation. The outputs of the two access maps are compared and a violation signal from either access map causes the comparator to prevent the memory access. If only one map signals a violation, the system is shutdown. We will analyze the safety of this example with respect to the domain isolation mechanism.

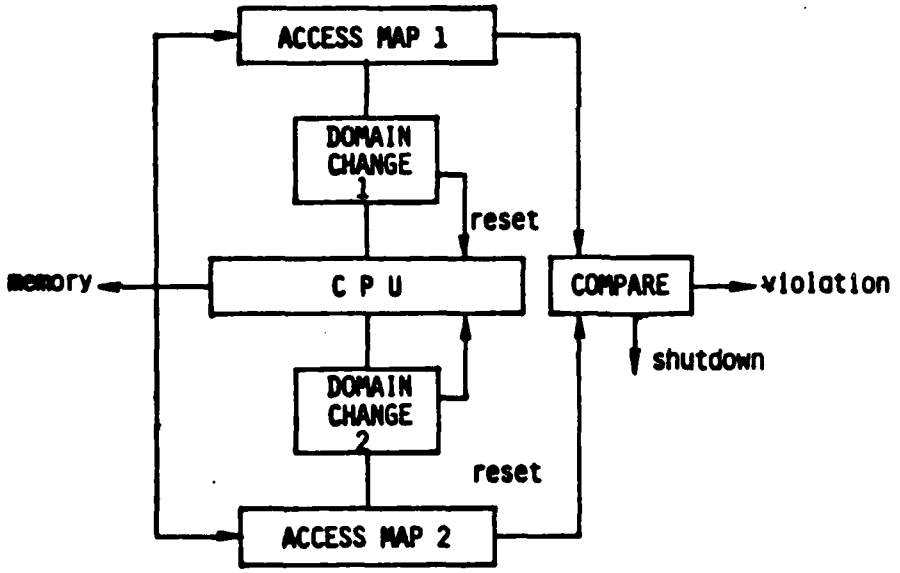


Figure 27. System Diagram with Component Redundancy

5.3.1 Components

We now need to consider the "components" of the system. A failure of any part of the CPU other than the reset mechanism can not result in a violation of domain separation. Hence we do not need to consider the CPU as a component here. In addition to the components of the system drawn in Figure 27, we must consider a memory failure which causes a write access to one memory location to modify another location or a read access to one memory location to depend on another memory location. A failure in either a domain changer or its associated access map will have the same effect of making unreliable the calculation of whether the memory address requested is in the current domain. Hence we can consider the combination of a domain changer and an access map to be a single component for our analysis. The probability of failure of this combined component can be calculated in terms of the probability of failure of its two individual hardware entities. We will refer to the combination of an access map and domain changer as a domain mechanism. The list of components to be considered then is:

- Domain mechanisms
- Reset mechanism
- Comparator
- Memory (addressing)
- Shutdown mechanism

5.3.2 States of the Markov Chain

Since there are six components, there are 2^6 = 64 possible sets of failures. There are two configurations, shutdown and operational. The result is 2*64 = 128 states for the Markov Chain. Fortunately, a preliminary analysis allows many of the states to be grouped together. Since we are interested in the probability of reaching an insecure state, we can group all the insecure states together. A failure of the comparator, the memory address mechanism or the comparator yields an insecure state unless the system is shutdown. If both the domain mechanisms fail, the system is insecure. Because of the symmetry of the example system, a failure of either domain mechanism is equivalent. Finally, if the system is shutdown, the system behavior does not depend on the failure set. We will consider shutdown as a single state. Thus the preliminary analysis yields six states: no failures, failed domain mechanism, failed shutdown mechanism, failed domain mechanism and shutdown mechanism, system shutdown, and system insecure. We will denote these states by nf, dm, sm, dm-sm, shutdown, and insecure respectively. The next step in our analysis is to calculate the 6x6 transition matrix.

5.3.3 Transition Matrix

The calculation of probabilities for the transition matrix must be based on standard data which will be available for hardware components. The information we will use in this example is the rate of failure for the components. Data about failure rates is typically available in the form of mean time between failures. With the assumption that the failure rate a standard exponential distribution, this data can be converted into a figure for the probability of failure in a fixed period of time. In particular the time can be chosen to be the unit of time selected for transitions of the Markov Chain. We will use P[i] to denote the probability of failure of component i in unit time.

Several of the 36 values in the 6x6 transition matrix can be quickly determined. We will make the assumption that once a component has failed it can no longer be trusted and hence will be considered failed from that time on. This means, for example, that P(dm,nf) = 0 because the fact that the first state has a failed domain mechanism means the domain mechanism must be in the failure set in the next state as well. The only exception to the rule that no components are removed from the failure set is the case where the system is shutdown. In this case components will be repaired or replaced before the system is re-activated. As a result the next state will certainly be the no failure state. The row of the transition matrix corresponding to the shutdown state will have a 1 in the no failure column and 0 in other columns. If the system is in the insecure state we will assume that it remains insecure in the next state.* The row of the transition matrix for the insecure state will have a 1 in the column for insecure and 0 elsewhere. P(nf,shutdown) = 0 because the system will not shutdown when all components are working properly. If the shutdown mechanism has failed, we will make the conservative assumption that the system can no longer shutdown. If more data is available about the nature of failures of the shutdown mechanism, the data could be used instead of this assumption and the calculations would be more precise. All assumptions have been made to overstate the probability of reaching the insecure state.

After this preliminary analysis, the transition matrix can be written in the form shown in Figure 28.

---

* In fact, if we are only interested in the probability that the insecure state is reached, it makes no differences what probabilities are used for moving to other states from the insecure state.

$$
\begin{aligned}
m_{1j} &= (1 - p_{1j}m_{jj}) + p_{11}m_{1j} + p_{12}m_{2j} + \cdots + p_{1N}m_{Nj},\\
m_{2j} &= (1 - p_{2j}m_{jj}) + p_{21}m_{1j} + p_{22}m_{2j} + \cdots + p_{2N}m_{Nj},\\
m_{3j} &= (1 - p_{3j}m_{jj}) + p_{31}m_{1j} + p_{32}m_{2j} + \cdots + p_{3N}m_{Nj},\\
&\vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad\qquad \vdots\\
m_{Nj} &= (1 - p_{Nj}p_{jj}) + p_{N1}m_{1j} + p_{N2}m_{2j} + \cdots + p_{NN}m_{Nj}.
\end{aligned}
$$

Figure 28. Computation of Mean First Passage Time

There are 14 entries which must be calculated from information about the failure rates of components. Most of these can be calculated from the probability of failure in unit time, F[i]. For those entries which depend on the probability of reaching the shutdown state we must use a value for the mean time between accesses to memory. The smaller this value, the less likely the system is to reach the insecure state instead of the shutdown state in the event of failures. This is an example of an instance where a simple self test, attempting a memory access during a prolonged idle period, can increase the safety of the system. The probability F'[i] of failure of component i in the mean time between memory accesses can then be calculated from the MTBF for components.

5.4 Calculations with Markov Chains

Once the transition matrix has been determined, standard techniques can be used to calculate the probability of reaching the insecure state in a given period of time. Alternatively, the transition matrix can be used to calculate the mean time to reach the insecure state.

The entry P(i,j) in the transition matrix T gives the probability of going from state i to state j in one unit of time. The probability of going from state i to state j in k units of time can be easily determined from the kth power of T. The (i,j) entry of T^k is the probability that if the Markov Chain is in state i at time t, it will be in state j at time t + k. The probability needed to determine the safety of a system with respect to a particular requirement is the (nf,insecure) entry of T^k where nf indicates no failures (the initial state) and k is the length of time from the definition of safety given in units of time used for the Markov Chain.

The mean time to reach the insecure state can be calculated by using the concept of mean first passage time. The mean first passage time from state i to state j, denoted m(i,j), represents the mean time required to reach state j starting in state i. The m(i,j) can be computed from the entries in T by using the equations given in Figure 29. For a fixed j, there are n linear equations in n unknowns where n is the number of states in the Markov Chain. The value of interest for safety is m(nf,insecure).

	nf	dm	sm	dm-sm	shutdown	insecure
no_failure	$P_{nf\ nf}$	$P_{nf\ dm}$	$P_{nf sm}$	$P_{nf\ dm\ sm}$	0	$P_{nf\ insecure}$
domain mechanism	0	$P_{dm\ dm}$	0	$P_{dm\ dm\text{-}sm}$	$P_{dm\ shutdown}$	$P_{dm\ insecure}$
shutdown mechanism	0	0	$P_{sm\ sm}$	$P_{dm\ sm}$	0	$P_{sm\ insecure}$
domain and shutdown mechanisms	0	0	0	$P_{dm\text{-}sm\ dm\text{-}sm}$	0	$P_{dm\ sm\ insecure}$
shutdown	1	0	0	0	0	0
insecure	0	0	0	0	0	1

Figure 29. Transition Matrix

For the large time values of interest in the calculation of safety the probability of reaching the insecure state at a given time will approximate an exponential distribution. As a result the mean time to reach the insecure state can be calculated directly from the probability of reaching the insecure state in a given period of time and vice versa.

5.5 SAFETY CONCLUSION

This section has introduced a technique for determining an upper bound for the probability of security compromises as a result of hardware failures. Application of the technique will certainly identify useful refinements. However, certain conclusions can be drawn which will be valid for any analysis of safety with respect to hardware failures.

It is important to choose a reasonable level of granularity for the analysis. The safety analysis described here does not dictate a particular granularity. The choice of components can be optimized for the particular application. In general, components which are too large will result in an overstatement of the probability of a security failure. The result will be that it is impossible to demonstrate that the system meets its safety requirements. If the components are too small, the analysis will be too complex to be practical.

The need to be able to choose reasonable components for the analysis puts some demands on the architecture chosen. If the architecture makes use of devices which integrate important

security controls with other complex functions, it will be difficult to get information about the failure characteristics of those portions of the device which implement the security controls. The failure rate of the device will be the only available measure of the robustness of any portion of the device. The only choices are to do a detailed determination of the failure characteristics of the security relevant portions of the device or consider the entire device as a component in the safety analysis. The first option will be impractical for most modern microprocessors. The second option will yield a measure of failure probability which will be too high for some very critical application. Thus, just as software verification places some constraints on the structure of verifiable programs and their implementation languages, the need to demonstrate safety puts some constraints on the architecture of a system. Of course, the separation of important security functions onto separate devices is not only useful for demonstrating safety. An architecture which uses separate devices for security functions also allows for a more credible demonstration that the functions are correctly implemented.

Finally, the safety analysis must be closely coordinated with the design and implementation verification. The verification process must include identification of those properties of the hardware which are required for the verification. This is the first step in determining which hardware requirements must be subjected to safety analysis.

6. FINAL CONCLUSIONS

This study investigated how the three disciplines of architecture, software verification, and security failure analysis could be applied in a mutually supporting manner such that the resulting microprocessor based controller could be attested to provide the level of security and reliability needed for correct operation. We successfully achieved this goal. We formulated safety and verification techniques for analyzing the correctness and reliability of secure communications equipment. We identified the domain machine and isolation kernel as an architectural approach for building secure communications equipment to which the above techniques are applicable in practice.

6.1 FUTURE STUDIES

We can now identify several avenues for future study. Each of these should take up where this paper leaves off.

- Further Work in Safety Analysis

  This new science needs significant work on techniques and tools. Safety analysis needs development of the general approach and further definition of the different types of failures. The ideas presented here demonstrate the possibility of such an analysis, but the data needed for evaluation is either unavailable or in some unusable form.

- Linguistic Constructs for Security Specifications

  The very best efforts in architecture, safety, and verification will fail if the Security Specifications for an application are imprecise or misstated. The current state of the art is to construct formal models of english specifications. The models are exact, but the step from a formal model to an english description is long and unsure.

- Design of a Security Chip Set

  The Access Map Architecture is simple enough that fully functional, high performance systems can be put together today. However, the basic concepts can be used to guide the development of VLSI circuits combining significant power into single chips without making the 286's mistakes. For example, a two chip set implementing a powerful Processor Block and a flexible, straightforward Access Map could be designed that allows validation of the protection mechanisms, functional and component redundancy, and high speed performance.

This paper shows how orchestrating architecture, verification, and safety will deliver the confidence required for a KGC utilizing high speed, low cost microprocessor components. As the work and study continues, so will the benefits continue.

END

FILMED

7-85

DTIC