WDL-TR
March

SECURE MINICOMPUTER OPERATING SYSTEM (KSOS)

EXECUTIVE SUMMARY

PHASE I: Design of the Department of Defense Kernelized Secure Operating System

Contract MDA 903-77-C-0333

Prepared for:

Defense Supply Service—Washington
Room 1D245, The Pentagon
Washington, D.C. 20310

Ford Aerospace &
Communications Corporation
Western Development
Laboratories Division

3939 Fabian Way
Palo Alto, California 94303

NOTICE

The Department of Defense Kernelized Secure Operating System (KSOS) is being produced under contract for the U.S. Government. KSOS is intended to be compatible with the Western Electric Company's UNIX™ Operating System (a proprietary product). KSOS is not part of the UNIX license software and use of KSOS is independent of any UNIX license agreement. Use of KSOS does not authorize use of UNIX in the absence of an appropriate licensing ageement.

This document, furnished in accordance with Contract MDA 903-77-C-0333, shall not be disclosed outside the Government and shall not be duplicated, used, or disclosed in whole or in part for any purpose other than to evaluate the contractor's performance of Phase I of the contract; upon completion of Phase I of the contract, the Government shall have the right to duplicate, use, or disclose the data to the extent provided in the contract.

The contents of this document shall be handled as proprietary information until 5 April 1978. After that time, the Government may distribute the document as it sees fit.

UNIX and PWB/UNIX are trade/service marks of the Bell System.

DEC and PDP are registered trademarks of the Digital Equipment Corporation, Maynard, MA.

KSOS Executive Summary

Ford Aerospace & Communications Corporation
Western Development Laboratories
Software Technology Department
3939 Fabian Way
Palo Alto, California 94303

ABSTRACT

KSOS is the Kernelized Secure Operating System designed for DARPA. KSOS is required to be externally compatible with Bell Telephone Laboratories' UNIX*tm, to be efficient, to satisfy certain multilevel security requirements, and to be demonstrably secure. This document provides a summary of the progress obtained in Phase I of the KSOS development by Ford Aerospace and its subcontractor SRI International under contract MDA903-77-C-0333. It gives an overview of the Phase I work, including a summary of the documentation delivered under the contract. It also outlines plans for the Phase II work.

ORGANIZATION OF THIS SUMMARY

This document is organized as follows.

INTRODUCTION

The long-term goal of the KSOS effort is to develop a commercially viable computer operating system for the DEC PDP-11/70 that

* is compatible with the Bell Telephone Laboratories' UNIX*tm,
* is capable of efficiency comparable to standard UNIX*tm,
* enforces multilevel security and integrity, and
* is demonstrably secure.

In order to achieve this goal, the Phase I effort described here has designed a trusted Security Kernel and associated trusted Non-Kernel Security-Related Software, such that the trusted software:

* provides a suitable basis for KSOS;
* intrinsically supports multilevel security/integrity,
* can be used by itself to support non-UNIX*tm-based applications,and
* is able to run efficiently on a DEC PDP-11/70.

The security of the overall KSOS system must be convincingly demonstrated. This will be accomplished by formal verification of the security properties of the design (i.e., the formal specifications) and selected proofs of correspondence between the delivered code and the design. In addition, KSOS will be rigorously tested to lend added confidence in the in the system.

Although the Security Kernel is intended initially to support an Emulator providing a UNIX*tm-like user environment, the Kernel has been designed to be used by itself, or with an Emulator providing a different user environment. Typical uses of the the Kernel by itself would be dedicated secure systems such as military message processing systems, or secure network front ends.

THE BASIC DESIGN

The design of KSOS consists of a Kernel (KSOS.K) that supports multilevel security, the trusted Non-Kernel Security-Related Software (KSOS.NKSR.T) which though outside of the Kernel, is trusted to deviate internally from the multilevel security policy to provide critical system functions, an Emulator (KSOS.E) that provides compatibility with the existing UNIX*tm user interface, and the untrusted Non-Kernel Security-Related Software (KSOS.NKSR.U) providing user-level services such as secure mail and line printer spooling. As a consequence of the requirement for a convincing demonstration of KSOS security, the trusted software should be reasonably small --in order to simplify the verification effort. However, it is neither necessary nor desirable that all security-related software be a part of the Kernel, particularly because some of the security policy may vary from one application to another. The design supports various security-related functions outside of the Kernel. Any meaningful verification of security must also consider any of the Non-Kernel Security-Related Software which is trusted to violate the strict sense of multilevel security and integrity. The FACC KSOS design encourages the minimization of such trusted software, although it makes explicit the efficiency tradeoffs that arise. Note that in the design discussed here the UNIX*tm Emulator software has essentially no effect on security, and therefore does not require verification.

A slightly simplified block diagram of the design approach is given in Figure 1, showing which levels of the design depend on which others and which design levels must be trusted. A given design level in this figure is permitted to depend only on lower design levels. In principle, a particular design level may call any lower design level directly; however, in the actual implementation there will be some restrictions imposed, as noted below.

As seen in the figure, the Non-Kernel Security-Related software for KSOS is divided into two design levels, one (KSOS.NKSR.T) trusted to violate selected parts of the multilevel security model in a controllable way, the other (KSOS.NKSR.U) not requiring any trust at all. The Emulator is seen to be nontrusted. The figure shows that the trusted KSOS.NKSR.T can call upon the Kernel. It also implies that the Emulator can call upon KSOS.K and KSOS.NKSR.T. Similarly, the nontrusted KSOS.NKSR can call upon the Kernel, the trusted KSOS.NKSR.T and KSOS.E. User applications (i.e., programs or dedicated environments) may in principle use the Kernel, the Emulator, and the Non-Kernel Security-Related Software, although in the actual implementation they can be constrained, e.g., not to use KSOS.K directly. By this means, certain Kernel primitives may be restricted to use by the trusted software, and certain Non-Kernel Security-Related functions may be restricted to use by administrative officers or system daemons. On the PDP 11/70, KSOS.K will run in Kernel mode, while the trusted KSOS.NKSR and KSOS.E will run in supervisor mode. Other systems than KSOS could be built using KSOS.K, which might or might not use portions of KSOS.NKSR and KSOS.E. Implementations of KSOS or just KSOS.K on other hardware are also anticipated. In a generalized domain architecture, Figure 1 is illustrative of how the system might be partitioned into more than just three states.

It is an engineering judgment as to what should be in the Kernel, as well as to what the specific Kernel interface should be, in order best to satisfy the system requirements. The approach taken in the FACC Phase I design is expected to provide significant advantages. In this design, the Kernel provides generality suitable for the implementation of UNIX*tm and other applications, while also being modest in size and conducive to efficient implementations for these applications. This arises from the use within the Kernel of compile-time definable types (similar to the extended type mechanism in SRI's Provably Secure Operating System, PSOS). In KSOS, this mechanism is used to support multilevel secure directories, without requiring the entire directory manager to be inside the Kernel. In the case of directories, a file "subtype" is supported by the Kernel, while the directory manager is a part of KSOS.NKSR.T. This allows the integrity of the directories to be improved while continuing to allow the Emulator to be untrusted.

The methodology employed throughout facilitates verification that the entire system satisfies the desired multilevel security properties. This verification is composed of two parts. First, that the design is consistent with the formal requirements, and second that the implementation is completely consistent with the design. As a result of the latter verification, the security of the implementation can be effectively demonstrated. Moreover, note that much more is thereby verified since the consistency proofs of the implementation guarantee not just secure operation but also correct operation, assuming the specifications are correct. That is, the demonstration that programs are consistent with their formal specifications guarantees that the implementation does what is specified, no more, and no less. It should be

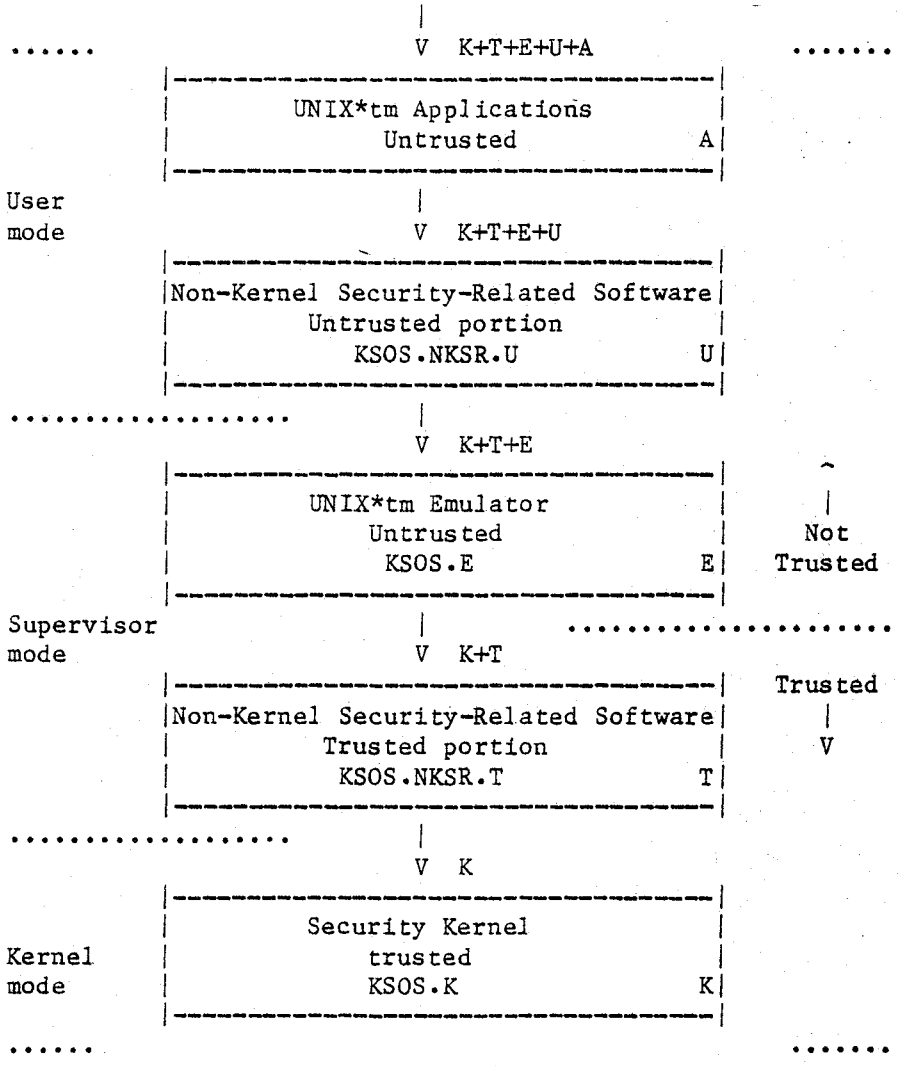

Figure 1
Block Diagram of KSOS Components

Note: K,T,E,U,A denote the functions provided by the five levels in upward order, respectively. The interfaces potentially visible at each level are cumulative upwards, e.g., as indicated by K+T+E+U+A. In actual implementation there may be restrictions on function visibility.

remarked that this two step verification, first of the design and then of the implementation, may reduce the overall verification effort. It also allows strong statements to be made about the system design whether or not full code proofs are undertaken.

The work of this contract has taken a strong systems viewpoint toward the overall development of the Security Kernel, the Non-Kernel Security-Related software, and the UNIX*tm Emulator. This viewpoint is focused around the use of a formal methodology for system design, implementation, and verification that has been developed at SRI International, and used previously on various

system designs. The methodology is called the Hierarchical Development Methodology (HDM). Its use permits a wide collection of needs arising throughout the development and subsequent use of the Security Kernel and its surrounding KSOS software to be carefully addressed or anticipated. As a consequence, the resulting KSOS design provides:

* multilevel security;
* provable security;
* high reliability and availability;
* high performance (operational efficiency) of both the Kernel and the UNIX*tm Emulator;
* flexibility of the Kernel design to be readily applicable to other hardware bases besides the PDP-11/70 (e.g., to the Honeywell SCOMP);
* generality of the Kernel design to be applicable to other security-relevant applications instead of or in addition to KSOS, e.g., a dedicated message processing system;
* controllability of the maintenance and evolution of the Kernel and Non-Kernel Security-Related software;
* ease of maintenance, evolution, and particularization to installation needs of the Emulator software, without adverse impact on the overall system security.
* ease of reverification following changes to the trusted portions of the system (KSOS.K and KSOS.NKSR.T).

It should be noted that the goal of provable security has significant implications that would affect any development process, with respect to the design, the choice of specification language, the choice of the programming language, and the choice of the verification methodology. However, these are all addressed by HDM and by the approach taken here.

THE HIERARCHICAL DEVELOPMENT METHODOLOGY, HDM

The formal methodology used in Phase I and proposed for use in the Phase II development of the KSOS system is summarized below.

* An overall systems viewpoint is maintained throughout.
* A unified methodology is used for design, implementation, and verification. This greatly increases the understandability of the design, the ease of implementation, and the verifiability of the resulting system. It includes the use of a formal specification language called SPECIAL (A SPECIfication and Assertion Language).
* The methodology encourages a hierarchically decomposed design, which itself has strong implications on initialization, shutdown, recovery from hardware and software errors, maintenance, and verification.
* A programming language is to be used that is well suited to both system programming and to eventual program verification.
* Verification is separated into two distinct stages, the first showing the correspondence between the formal specifications of the design and the formal requirements for multilevel security, the second showing the consistency of the programs with their specifications. The combination of these stages assures that the implementation completely satisfies the multilevel security requirements. This approach increases the understandability of the proofs, and also simplifies them.
* Advanced but well-debugged development tools supporting HDM have been used and will be used wherever appropriate. Existing tools used in Phase I include checkers for the hierarchical structure, the specifications, and the mappings between the state representations at different levels.

An existing theorem prover and simplifier are expected to be used in Phase II to provide verification tools supporting proofs of correspondence between specifications and the multilevel security model. Related tools -- some existing and some under development -- may be used to provide illustrative proofs of program correctness, as appropriate.

The methodology attempts to unify the entire development process. It decouples design and implementation into distinguishable stages, providing a formal definition of the design and a formal basis for implementation and proof. This approach considers the entire development process in a formal way and permits formal proofs at each stage in the process. Even in the absence of proofs, this approach seems to greatly increase the understandability and precision with which a design can be expressed, and the ability to evaluate the reasonableness of such a design with respect to stated desired properties of the system. The methodology has considerable utility throughout the development of KSOS, in Phase I, in Phase II, and in any additional efforts to provide proofs of implementation correctness. It also makes a positive contribution to various further related tasks, such as verification of the consistency of any subsequent changes affecting security, as well as implemention of the design on other hardware and verification of the resulting system. In the latter case, specifications for most of the Kernel (except for the machine and device-dependent levels) could remain largely intact, and the specifications for KSOS.E and KSOS.NKSR.T could remain unchanged. Thus the demonstration of the security of the design can carry over directly to the new implementation. The verification of consistency between code and specifications might also carry over in part, depending on the programming language used.

SECURITY

The desired multilevel security requirements demand that information at a particular security level may not move downward to a lower security level. Because of the syntax of SPECIAL, the proofs that these requirements (formally stated) are actually satisfied by the specifications follow largely from simple (i.e., mostly syntactic) checks on the specifications. Following such proofs, any implementation consistent with the specifications would itself satisfy the security requirements. That a design proved to be secure is itself correctly implemented then follows completely from proofs of the consistency of the specifications with their implementing programs and hardware. (The dependence on correct hardware is made quite explicit by this approach.) It is of course also desirable to demonstrate that the specifications --even if proved to be secure-- actually describe the desired effects. This task is aided by the understandability of the specifications, and by testing of the resulting implementation. For example, the specifications for the top-level (user-interface) can be compared with the behavior of existing UNIX*tm in the case of the Emulator. The resulting system can be compared with exisiting UNIX*tm by running programs and applications environments on both systems.

The design for the Kernel permits all of the Kernel primitives to satisfy the desired security properties completely under normal usage by users. A few relaxations of this strict behavior are necessary to support the trusted Non-Kernel Security-Related software, and are confined to the KSOS.NKSR.T by the controlled distribution of minimal privilege. These isolated relaxations can be shown to satisfy a specific subset of the security properties, in a completely controllable way, and to be masked completely by the trusted Non-Kernel Security-Related software.

THE ROLE OF SPECIFICATIONS

Formal specifications by themselves provide a significant advance in the state of the art of software system development. They provide a concise and precise functional statement of exactly what any external or internal interface is expected to do. They enforce abstraction on the design that consequently simplifies implementation, debugging, system integration, and maintenance. They greatly enhance the understandability of a design. They provide a forum for discussion of design issues. Their understandability encourages the manual discovery of design errors. They also make possible the intuitive verification of certain desired properties that the design should satisfy.

THE ROLE OF THE PROGRAMMING LANGUAGE

It is desired that the programming language used for the Kernel and the Non-Kernel Security-Related software have certain strong properties. (The Emulator may also take advantage of this language.) The desired properties include such things as

* adequate compiler support for generating efficient code,
* suitable constructs for control and data abstraction,
* type safety,
* ability to support multiprogramming, and
* ability to handle machine-dependency when necessary.

Some of these desired properties (notably type safety and support of abstraction) contribute significantly to the verifiability of the resulting code. They also contribute to the avoidance of many characteristic security flaws. At the moment, Euclid appears to be highly appropriate, with an extended Modula as an alternate choice. (It appears that some of the competitive DoD/1 languages would be appropriate, if adequate support were available.)

THE ROLE OF VERIFICATION

As noted above, specifications support proofs of specification properties, and also facilitate proofs of program consistency with the specifications. The ability to state and prove properties about a design (as represented by a set of specifications) -- before that design is ever implemented -- will have a significant impact on the system development. Nevertheless, no system can justifiably be thought to be secure unless appropriate properties of its implementation can also be proved. On the basis of the work to date, proving that the specifications for the KSOS design satisfy the required multilevel security properties can be straightforward and accomplished largely by automated tools -- many of which have already been developed at SRI. In addition, although more complex than such design proofs, proving the consistency of implementation with respect to the specifications is now becoming a realistic task, especially with the emergence of recent theoretical advances and the prospect of suitable on-line tools. Furthermore, the expected use of a language like Euclid or extended Modula would very helpful. In addition, the proposed use of review and testing is expected to increase the confidence in the implementation.

THE ROLE OF ON-LINE TOOLS

The role of computer tools is indicated above, with respect to the syntactic checking of specifications, the verification of the security of the design, and the eventual verification of the consistency of programs with the specifications. Experience in attempting to develop secure systems in the past indicates that an enormous amount of mind-numbing effort would be required under conventional approaches, and even then there is considerable doubt as to whether security flaws still remain. The approach outlined here, with its judicious use of on-line tools that support the Hierarchical Development Methodology, is expected to result in considerably more confidence in the security of the resulting system than is possible with conventional, largely manual approaches. Further, the automated approach promises to be far more cost-effective. For example, during the exercise of writing of formal specifications for UNIX*tm, various previously unknown flaws in that system were detected. In the writing of formal specifications for the KSOS Kernel, various minor flaws were detected by the hierarchical interface checker and the specification analyzer. These flaws, many of which might give rise to insecurity in the implementation, have been detected and removed during this early stage of design. This is particularly valuable for various minor typographical errors in the specifications which otherwise might result much later in significant flaws in the resulting system. In addition, because of the structure and constraints of the methodology, flaws in the implementation of even a correct design may also often be detected by the implementation tools, e.g., the compiler and simple consistency checks.

THE KERNEL

The Security Kernel (KSOS.K) is structured into a hierarchically ordered set of modules, each of which depends (for its implementation and for its correctness) solely on lower-level modules. The set of accessible Kernel calls has been chosen to be powerful and efficient for the implementation of KSOS, but general enough for the implementation of other applications (e.g., dedicated). These Kernel calls support (among other things) the creation and deletion of files and processes, the reading and writing of files, interprocess communication, and the protected invocation of trusted software.

The Kernel has a "UNIX-flavor" to it. It was designed with the actual implementation of the lower levels of UNIX*tm in mind. This, of course, does not mean that the Kernel is suitable only for creation of UNIX*tm user environments. Significant efforts have been made to make the Kernel both machine independent and UNIX*tm independent. The Kernel design incorporates many of the concepts from the existing prototype "Secure UNIX*tm" implementations. Its main departure from the prototypes is that the FACC design does not employ virtual memory. This decision was reached because existing UNIX*tm software has very large "working sets" that minimize the value of a virtual memory architecture. Also motivating against a virtual memory architecture are the long delays associated with process environment switches on a PDP-11/70. Satisfying page faults, even if the page is in core could significantly degrade system performance.

The Kernel internally supports objects of program-definable types and capability addressing. These are intended for use within the Kernel for creating Kernel-supported objects such as multilevel secure directories without requiring any of the directory mechanism to reside within the Kernel

-- the directory manager is in KSOS.NKSR.T. An overview of a proposed design decomposition of the Kernel follows, from highest level of abstraction to the lowest.

* Kernel calls
* process operators
* interprocess communication
* file capabilities
* file subtypes
* process segments
* process states
* mountable file systems
* file contents
* file states
* multilevel security
* privilege control
* device-independent functions
* type-independent information
* secure entity names

THE TRUSTED NON-KERNEL SECURITY-RELATED SOFTWARE

Only part of the Non-Kernel Security-Related Software must be trusted (and hence ultimately verified). Although most of the Non-Kernel Security-Related functions must contain a small amount of trusted code, most of the code supporting these functions need not be trusted. A spectrum of design decisions can be made either distributing or centralizing the trusted portion of each function. The FACC design permits the portion which must be trusted to be kept small. The Non-Kernel Security-Related Software as a whole supports the following functions.

* system startup and shutdown
* login and logout
* password changer
* user security-level changer
* file security-level changer
* virtual terminal handler
* mount and unmount
* line-printer daemon
* file system maintenance, dump/restore
* system administration

As noted below, the spooler and the mailer are examples of security-related programs that do not need to be trusted, because of the constraints imposed by the Kernel and the trusted Non-Kernel Security-Related software. The nontrusted functions need not be verified. Further simplifying the verification effort of the trusted portions is the fact that they are composed of autonomous modules which can be verified independently.

THE EMULATOR

The KSOS Emulator interface supports the UNIX*tm calls, and implements them in terms of the KSOS Kernel. It is protected from the user, and the Kernel is protected from it. In general, it calls the Kernel directly rather than going through the trusted Non-Kernel Security-Related software, except for certain directory operations. In essence, the Emulator does whatever it has to in order to provide compatibility with the desired UNIX*tm calls.

However, certain features of UNIX*tm have been removed from the user interface to KSOS, in the interests of providing a secure system. Most notable among these is the "superuser" facility. Also, the checks on certain user functions have been strengthened.

The Emulator contains the bulk of the support for the interface to the computer network. Only the multiplexing and demultiplexing of the data streams to and from the network are trusted. The flow control and data stream integrity functions of the network are untrusted and are supported on a per-process basis by the Emulator. This architecture is extremely attractive for a number of reasons. First the size of the trusted software is reduced to a minimum. Second, the flow control is truly end-to-end. Third, overall structure requires minimal Kernel support. Finally, the basic architecture can be easily adapted to support other networks protocols.

THE NONTRUSTED NON-KERNEL SECURITY-RELATED SOFTWARE

As noted above, many of the Non-Kernel Security-Related functions require some trusted code, although most of the code for the implementation of these functions need not be trusted. In addition, the spooler and the mail facility -- although in principle security related -- can operate entirely as untrusted programs. The design thus allows great flexibility in its implementation. It is also possible to easily extend the functions provided by the Non-Kernel Security-Related software because they are not hard coded into the Kernel.

THE WORK PROPOSED FOR PHASE II

The aim of the proposed Phase II work is to develop an effective implemention of the design Phase I KSOS design, to demonstrate that this design completely satisfies the desired properties of multilevel security, and to demonstrate the essential correctness of the implementation by illustrative rather than exhaustive means. On the basis of the design that has emerged from Phase I, and the structured methodological approach being used throughout the development, there is reasonable evidence that this aim can be accomplished in a timely and cost-effective way. The proposed work for Phase II will also provide detailed illustrations of how the implementation can be demonstrated to be correct, that is, proven consistent with its specifications.

PRELIMINARY EVALUATION

The approach used here affords various significant advantages over previous competing approaches, but avoids incurring many of the risks typically associated with high-technology attempts to advance the state-of-the-art. Considerable success has already resulted from the use of this approach, and such success is justifiably expected to continue.

From a systems viewpoint, the work described here is novel in many respects. These include the following.

* KSOS will be the first full use of the formal methodology (HDM) for a complete system development. However, HDM has been well tested in the design stage of several previous projects.
* The HDM methodology can accommodate the verification of a larger amount of Kernel and other trusted software than can other approaches. This is due to two orthogonal decompositions: the decomposition of the

verification process into stages (e.g., specification-to-model proofs, followed by code consistency proofs) and the decomposition of the design into hierarchical levels of abstraction. These both simplify the verification effort significantly. The automated tools offer a manyfold further reduction in effort. In addition, the approach is directly applicable to the verification of the security of the Non-Kernel Security-Related software.

* KSOS is likely to involve the first use in the development of a production-quality computer system of a modern programming language (Euclid, or possibly Modula) highly appropriate for such an effort. Note that each of these languages is a conservatively designed variant of an existing well-established programming language (Pascal).
* This will be the first implementation of a production system that includes a Security Kernel designed to be provably secure, and implemented using a programming language suitable for such verification.
* The design takes advantage of several innovative operating system concepts, e.g., using objects of extended type (here called file subtypes) within the Kernel. The use of Kernel-supported types is expected to produce significant advantages in flexibility and generality.
* Because of these innovations, it should be stressed that the risks are minimal. The experience to date is very promising. For example, the time required for FACC to master the methodology was shorter than expected. The approach is significantly aided by well-used supporting tools. The task of formally verifying that the specifications for the KSOS design satisfy the multilevel security requirements seems reasonable. The task of producing an efficient and secure implementation from the existing Phase I design appears to be straightforward. The task of demonstrating that the implementation is correct ultimately requires formal proofs that the programs are consistent with the specifications. While complete proofs are not proposed, it is expected that a combination of illustrative proofs will demonstrate the feasibility of carrying out complete proofs in the future.

The design takes advantage of the strengths of both of its prototype precursors, namely the UCLA Data Secure UNIX*tm and the MITRE Secure UNIX*tm, although the present approach has numerous advantages over those prototypes, as follows.

Re UCLA: The FACC design carefully considers efficiency and flexibility in advance. (Note that the use of capabilities within the Kernel is also found in the UCLA Kernel.) The use of formal specifications with a proof methodology tied to those specifications permits proofs of the intrinsic security of the design, based on the specifications, independent of subsequent implementation and verification of implementation correctness. The FACC choice of programming language seems to be better suited for implementation and for eventual program verification than UCLA Pascal.

Re MITRE: The SRI formal methodology for specification and proofs of specification properties is similar to that used by MITRE; however, the concept of hierarchy, the specification language, the program proof methodology and the tools for automatic specification checking and program verification are more advanced than MITRE's.

The FACC KSOS design does differ from the protoypes in that it does not use virtual memory. As discussed above this choice was motivated by performance considerations, and analysis and experimentation with virtual memory UNIX*tm systems.

GUIDE TO DOCUMENTATION

The following documents are included in the documentation of the KSOS Phase I effort.

KSOS System Specification (Type A)
KSOS Computer Program Development Specifications (Type B5)
KSOS Verification Plan
KSOS Implementation Plan
KSOS Maintenance and Support Plan

KSOS SYSTEM SPECIFICATIONS (TYPE A)

The System Specification (Type A) establishes the requirements for the KSOS system with respect to performance, design, development, and test. Deviations from the behavior of the existing UNIX*tm user inerface are explicitly cited.

KSOS COMPUTER PROGRAM DEVELOPMENT SPECIFICATIONS (TYPE B5)

The Program Development Specifications (Type B5) provide the detailed design of the Kernel, the Non-Kernel Security-Related software, and the UNIX*tm Emulator, with one document for each. The interface presented by the Kernel is given in detail. A draft version of formal specifications (written in SPECIAL) for the externally visible functions and many of the internal functions of the Kernel is included as an appendix to the Kernel B5 specs. These are not required in final form until Phase II, but are included at this time as illustrative of the approach, and demonstrative of the depth of consideration given to the design. Preliminary formal specifications for the existing UNIX*tm system exist and have been distributed previously, although they are not required at this time. The process of generating these latter specifications was very helpful in defining what KSOS should actually appear to do, and was also valuable in ferreting out several hitherto unknown bugs in UNIX*tm.

VERIFICATION PLAN

The Verification Plan provides the precise model for multilevel security that the Security Kernel is expected to satisfy. It also shows how the formal specifications for KSOS can be formally proven to be consistent with the formal model for multilevel security. In addition, it discusses the choice of programming language to be used in the Phase II implementation, the process of verifiying that programs are consistent with the formal specifications (in Phase II and beyond), and the tools that would be used to support the verification effort associated with the Phase II development effort.

IMPLEMENTATION PLAN

The Implementation Plan discusses programming techniques, implementation tools, testing, external configuration management, and the assurance of integrity and performance of the implementation. FACC plans to utilize its on-going work in development-support systems based on UNIX*tm to aid in the creation of KSOS. The plan emphasizes tools that are well matched to the scope and nature of the KSOS effort.

MAINTENANCE AND SUPPORT PLAN

The Maintenance and Support Plan discusses what will be required in order to test, maintain, and modify the KSOS software. The long term maintenance of KSOS is viewed as an extention to the procedures for configuation management and trouble reporting that will be routinely used during the development phase. Thus, the mechanisms will be well established and thoroughly "debugged" prior to the maintenance phase. Also discussed in this document is the mechanism for system generation of KSOS at the individual user sites. The procedures are intended to allow a security officer (or other similarly computer-naive users) to generate a KSOS system, and to be assured of its security and integrity properties.