The goal of the Department of Defense Kernelized Secure Operating System (KSOS) project is to design, implement and prove a secure operating system. Specifically, it is desired that KSOS be designed and proven to enforce a security model, derived from the security practices of the Department of Defense, referred to as "multilevel security."

The proof required for KSOS is rigorous proof in the mathematical sense. The necessity of preparing for proof of a program so large and complex as an operating system has led to the adaptation and, where necessary, development of design and implementation methodologies for KSOS which are departures from the usual methods of systems programming. Specifically, KSOS has required formal specification of operation system design, automatic theorem generation, automatic theorem proof, selection and use of a verifiable programming language, and verification of operating system programs. KSOS represents the first industrial application of many of these techniques, and is breaking new ground in the construction and proof of large scale computer systems.

This paper describes what methods were chosen for KSOS and how they are being applied.

BACKGROUND—THE MODEL AND THE SYSTEM ARCHITECTURE

The multilevel security model attaches a tag known as an access level to every object managed by the system and places constraints upon the valid relationships between the access levels of interacting objects.

The design of KSOS has been described in detail elsewhere. A brief description of its architecture provides sufficient background for a discussion of the methodology. A KSOS system is comprised of

1. A kernel, which performs operating system functions and which has the responsibility of enforcing the security policy. The object types supported by the KSOS Kernel are processes, process segments, files, devices and subtypes. The Kernel is motivated to perform actions upon these objects by sequences of calls to the routines which the Kernel provides at its interface with the rest of the system. An important mechanism for enforcing the security policy is provided by comparisons, made at kernel call time, between the access levels of the caller and of the objects the caller seeks to manipulate.

2. An emulator, which uses the facilities of the Kernel to fabricate an (arbitrary) environment for user programs. The emulator being prepared initially for KSOS emulates the UNIX operating system. The use of an emulator is convenient for applications which seek to exploit existing software, but is not strictly necessary. The KSOS design envisages that certain applications will make direct use of Kernel facilities.

3. Support software to aid in the day-to-day operation of the system, (e.g. secure spoolers for line printer output, dump/restore programs, portions of the interface to a packet-switched communications network, etc). These are collectively referred to as "Non-Kernel System Software" (NKSS). Because of its varied responsibilities, portions of the NKSS must from time-to-time be allowed to violate the security model in order to operate correctly. These portions are referred to as privileged NKSS. To a large extent, they represent a mismatch between the idealizations of the multilevel security model and the practical needs of a real user environment. The design of KSOS allows for these violations but seeks to minimize them by providing for the economical definition of finely grained privileges and mechanism for Kernel security support of user defined extended types.

The KSOS components for which proof is required are those which are responsible for enforcing the multilevel security model, i.e. the Kernel itself and those portions of the privileged NKSS.

The remainder of this paper is organized around a schema
for the construction of provable systems. Initially only a brief overview of the schema is presented. This is followed by an elaboration of each step of the schema, first in general terms and then in terms of its impact upon KSOS methodology.

CONSTRUCTION OF PROVABLE SYSTEMS

Figure 1 illustrates a general schema which, in principle, can be used to construct proofs that systems conform to arbitrary policies. It will be seen that the overall proof is constructed from two sub-proofs, namely

P1—Proof that the system design conforms to the desired policy (property).

P2—Proof that the implementation conforms to the proven design.

From these there follows

P3—Proof that the implementation conforms to the desired policy.

Successful application of this schema requires that a great many careful preparations be made before the proofs are attempted. The various methodologies adopted for KSOS were chosen to ease the burden of these preparations. These methodologies have so far been useful in this role. In addition, the discipline of following rigorous design methodologies has yielded software engineering benefits which were unanticipated at the time the methodologies were adopted. These will be discussed in more detail below.

FORMAL STATEMENT OF DESIRED PROPERTY

Consider first those preparations involving the desired property of the design. Generally, there exists some informal statement of this property. Informal statements, written in natural language, are designed and may be adequate for human interpretation. They are however unsuitable as a touchstone for mathematical proof as they lack sufficient precision and are not in an easily manipulatable form. The informal statement of the property must therefore be converted into a suitable formal statement. Great care is required at this stage to ensure that the formal statement accurately and adequately represents the intention of the informal statement.

In the case of KSOS, informal statements of the desired security property are to be found in regulatory documents. These informal statements embody an intuitive notion of military security policy. They are widely applied and well understood. A mathematical model approximating this policy was developed by Bell and LaPadula. This model has been utilized as a formal policy statement in the design and verification of a security kernel during a project which was a predecessor to the present KSOS project. Another similar model was described by Walter. The formal policy statement being used for KSOS was prepared by generalizing from these two models and formulating the generalization in terms amenable to proof. A informal description of the model may be found in a companion paper. Full details of the KSOS formal statement are shown in Reference 13.

FORMAL SPECIFICATION OF SYSTEM DESIGN

The next step in construction of a provable system is expression of the system design in a fashion which is suitable for the construction of a proof. Such an expression is referred to as the system's formal specification. A formal specification may be viewed as a set of equations which describe the possible "states" of the system.

Viewed this way, the preparation of formal specifications is not attractive to system designers. There are two main problems. The first is that the sort of state abstraction required to formulate the equation set is not the same sort of functional or data abstraction in which the designer is trained and which (following tradition) he would otherwise use to express his design. The second problem is that adequately detailed specification of useful systems requires a large and unwieldy set of equations throughout which it is difficult to maintain consistency.

Hierarchical development methodology

Fortunately both problems may be alleviated by use of appropriate computer-based techniques. The techniques chosen for use in KSOS are embodied in SRI International's Hierarchical Development Methodology (HDM). HDM addresses formal specification problems by providing mechanization for each of a series of steps required for system design and production. These steps are the decomposition of a design into a partially ordered hierarchy of modules, the specification of each module, the specification of the interfaces and mappings between modules and, eventually, the proof. All of these functions are performed by an integrated collection of supporting programs.

In HDM as used for KSOS, each module is considered to represent an incremental abstract machine. This is implemented upon the abstract machines coming below it in the hierarchy. Each module is specified in terms of the data
types it manipulates, and in terms of functions. There are several kinds of functions. Primitive V-functions represent the state of the abstract machine. They have a value. Derived V-functions also have a value which is computed from the value(s) of primitive V-functions. O-functions represent operations upon the state of the abstract machine by specifying how the machine’s state after the operation is related to the machine’s state before the operation. OV-functions combine operation and state. This model of functional decomposition follows roughly from the early ideas of Parnas. A large reduction in the problems of abstraction is due to this model.

The abstract machines are specified in a nonprocedural language called SPECIAL (for SPECification and Assertion Language). SPECIAL has the advantageous property that the effects of a computation may be specified independently from that computation’s implementation. This is a vital pre-condition to design proof (PI in Figure 1). In addition, this model of functional decomposition allows the designer to concentrate upon the structure and functionality of his system, and to defer decisions about data representation and even algorithm choice until implementation. SPECIAL is supported in HDM by a language processor, called the Specification Checker. This provides a number of largely syntactic tests which aid the designer in maintaining consistency of definition within individual modules and throughout a hierarchy of modules.

Formal specification of KSOS

The externally visible design of the KSOS kernel was decomposed and specified using HDM. Twenty modules were used to establish and support the functionality of the kernel interface. The hierarchy of these modules is shown in Figure 2. Each of these modules has been specified in sufficient detail to allow all the externally visible effects of each kernel call, or of any sequence of kernel calls, to be determined by inspection of the specifications. The size of each module is dependent upon the nature of its abstraction and the ease of specifying how that abstraction might be produced in terms of functions available from lower level abstractions.

There are 34 KSOS Kernel calls. The specification of the kernel’s visible actions contains the definition of about 240 functions, comprised of roughly 3000 lines of SPECIAL. The complete formal specifications of the KSOS Kernel have been published in Reference 2, and those of the privileged portions of the NKS in Reference 3.

A small example (the single function SEGrendezvous) taken from the Kernel formal specifications is included as Figure 3 to give the reader exposure to the style and content of the work. SEGrendezvous is part of the mechanization of shared segments. The function is a derived V-function. It returns a value computed from the values of other V-functions. It takes three parameters, psSeid, rdvSeid, and da. The names “sSeid,” “daType,” and “RdvType” are user-defined type names. The semantics of SPECIAL call for the EXCEPTIONS to be evaluated in turn. If any of them are found to be TRUE, its associated label is “returned” as an error value and no further EXCEPTIONS are checked. If all of the EXCEPTIONS are FALSE the return value, segSeid, is calculated according to the specification in the DERIVATION section of the function.

Stated in slightly different terms, the precondition of a SPECIAL function is the conditional conjunction of the negations of its EXCEPTIONS. The postcondition of a SPECIAL function depends upon the type of function being considered. In the case of derived V-functions (such as this example) the postcondition is the DERIVATION. In the case of O-functions, the postcondition is the conjunction of the function’s EFFECTS.

Drawbacks and benefits of formal specifications

It is unfortunately true that the formal specifications of KSOS are difficult to read. This point is amply illustrated by Figure 3. Their poor reliability is due in part to the syntax of the SPECIAL language, and in part to the specification style adopted (style continues to evolve; see Reference 19). The specifications are nevertheless popular with KSOS designers and implementors. This is because they provide a medium in which design decisions can be expressed, discussed and recorded with precision and with assured continuation of design consistency. The designers and implementors communicate effectively in terms of the formal specifications. This is certainly a major benefit and it was unanticipated when the methodology was adopted.

Additional unanticipated benefits derive from the constraint of working with a hierarchical decomposition. We have found it extremely difficult to make a clean decomposition and formal specification of a kludge. On several occasions during work on KSOS, difficulty in formulating a specification for a design has encouraged prompt reexamination and subsequent simplification of that design. In other words, formal specifications help designers to understand and evaluate their product.

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>NAME</th>
<th>ABSTRACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>KER</td>
<td>kernel call interface</td>
</tr>
<tr>
<td>18</td>
<td>SPF</td>
<td>special functions</td>
</tr>
<tr>
<td>17</td>
<td>PRO</td>
<td>process operators</td>
</tr>
<tr>
<td>16</td>
<td>IPC</td>
<td>interprocess communication</td>
</tr>
<tr>
<td>15</td>
<td>FCA</td>
<td>file capabilities, open files</td>
</tr>
<tr>
<td>14</td>
<td>SUB</td>
<td>file subtypes, type extension</td>
</tr>
<tr>
<td>13</td>
<td>MFS</td>
<td>mountable file systems</td>
</tr>
<tr>
<td>12</td>
<td>PST</td>
<td>process state</td>
</tr>
<tr>
<td>11</td>
<td>PVN</td>
<td>process virtual memory</td>
</tr>
<tr>
<td>10</td>
<td>SEG</td>
<td>segments</td>
</tr>
<tr>
<td>9</td>
<td>FIL</td>
<td>file contents</td>
</tr>
<tr>
<td>8</td>
<td>FST</td>
<td>file state</td>
</tr>
<tr>
<td>7</td>
<td>SMX</td>
<td>security model</td>
</tr>
<tr>
<td>6</td>
<td>PRV</td>
<td>privilege control</td>
</tr>
<tr>
<td>5</td>
<td>DIF</td>
<td>device independent functions</td>
</tr>
<tr>
<td>4</td>
<td>TII</td>
<td>object type independent information</td>
</tr>
<tr>
<td>3</td>
<td>SYL</td>
<td>system level</td>
</tr>
<tr>
<td>2</td>
<td>SEN</td>
<td>secure entity names, system namespace</td>
</tr>
<tr>
<td>1</td>
<td>DIF</td>
<td>device dependent functions</td>
</tr>
<tr>
<td>0</td>
<td>MAC</td>
<td>machine</td>
</tr>
</tbody>
</table>

Figure 2—KSOS Kernel abstraction hierarchy.
VFUN SEGrendezvous(seid pSeid, rdvSeid; daType da) -> seid segSeid;

EXCEPTIONS
KEsegBadName: ~(EXISTS rdvTe x INSET SEGrdvTable():
    x.nameSeid = rdvSeid);
KEsegBadLevel: ~(EXISTS rdvTe x INSET SEGrdvTable():
    x.nameSeid = rdvSeid
    AND SEGshareCheck(pSeid, x.segSeid, da));

DERIVATION
LET rdvType x = SOME rdvType y | y INSET SEGrdvTable()
    AND y.nameSeid = rdvSeid
    AND SEGshareCheck(pSeid, y.segSeid, da)
    IN x.segSeid;

Figure 3—Formal specification of the function SEGrendezvous.

DESIGN PROOF

Consider now the next step in the provable system schema—the proof that the design conforms to the desired property. Slightly more rigorously, we wish to prove that the formal specification of the design implies the formal statement of the desired property. In practice, this inference is not proven directly. Instead, the formal specifications are processed to generate formulas relating the states of the specified design to the desired property. An attempt is then made to prove these formulas. For KSOS, the generated formulas are such that, for each specified function, the access levels of the objects manipulated are related to the access level of manipulator in accordance with the formal statement of the security policy. The formulas themselves take the form of inequalities upon access levels.

The formula generator has information about the computational model of HDM and about the “semantics” of SPECIAL. It also has implicit information about the formal model of security. An anticipated generalization of the formula generator would accommodate arbitrary formal models, perhaps expressed in SPECIAL.

KSOS exhibits considerable novelty in its use of an automatic design proof environment. During a predecessor project,11 the generation of formulas and their proof as design theorems was done manually. This manual work was labor intensive and mind-numbing; it required great vigilance against error. In KSOS the cost of proof and the risk of error are reduced by utilizing an automatic formula generator coupled to an automatic general-purpose theorem prover.28 To our knowledge, KSOS is the largest program for which automatic design proof along the lines sketched above has been accomplished.

Proving the published KSOS design entails the generation and proof of about 500 formulas. This is routinely accomplished in an entirely automatic fashion. The complete process requires about 10 CPU-minutes on a DEC KL-10. This dramatic reduction in proof cost has made it feasible to include a feedback path not shown in Figure 1 whereby formal specifications giving rise to unprovable theorems are modified and the proof then retried. By this mechanism even details of the design can be coerced into conformance with the desired property.

IMPLEMENTATION

Proof P1, that a formally specified design implies a desired policy, is a major milestone in the schema. The next logical step is to produce an implementation of that design. For this, an implementation language must be chosen. The translation from formal specification to implementation is, in part, automable. However, completion of the task requires application of traditional inspection, review and testing methods. As the KSOS implementation effort is only just beginning at this writing, we are forced, in this section and the next on Program Proof, to discuss our plans, not our results.

Choice of implementation language

The peculiar nature of operating system programming places some well known demands upon programming languages. In addition to these, KSOS places the additional requirement that the system implementation be verifiable, i.e., P2: proof that the implementation conforms to the specification.

The following are the requirements for the KSOS pro-
gramming language:

a. The language must be well defined and be supported by a stable, efficient compiler, which produces efficient code.
b. The language must provide "modern" control structures, data structures, abstract types, type safety and machine-dependent scopes.
c. The language must be compatible with HDM and be amenable to axiomatization.

A short list of likely system implementation languages was prepared. These were Euclid, Modula, ILPL, Gypsy, Pascal, C and Ada. Of these, the most suitable on technical grounds appears to be Euclid. However, difficulties encountered by the implementors of a compiler for Euclid have led to our choice of Modula as the KSOS implementation language.

Mapping a formal specification into code

Several levels of documentation and specification have been developed for KSOS. First, a system-level specification was produced. Next, a design specification was developed that included both prose and formal specifications for the major components of the system (e.g., Kernel, Emulator, NKSS). And finally, a product specification was developed for each of the major components of KSOS. Each of these specifications is more detailed than its predecessor and defines the implementation more exactly. In concept, each successive specification provides a refinement of the ideas presented in earlier, higher-level specifications.

Care must be taken to avoid the constant danger of inconsistency, both within a given specification level and between levels. To guard against this, we have established the primacy of formal specifications in all questions. Thus, each of the managers, designers and programmers on the KSOS project has at least a reading knowledge of SPECIAL.

There are some aspects in which any formal specifications bind their admissible implementations. Specifications written in the style used for KSOS bind the structure, functionality and local assertions of the implementation. In order to exploit this binding we plan to create and use a software tool which will map from the formal specification domain into the implementation domain, producing implementation language skeletons for use and refinement by the implementors.

There is, in principle, no requirement that a given formal specification be implementable. Neither is there a requirement that the structure of an implementation follow that of its specification. It seems to us, however, that both of these requirements increase the practical utility of incorporating formal specifications in a methodology for program development, and we therefore strive to meet them. We have shown how the KSOS formal specifications are used both manually and automatically to provide implementation guidance. The goal of decomposing the system specification into an easily implementable hierarchy was identified early in the KSOS project. The effectiveness of this choice will not be known until the implementation is complete.

Implementation environment

KSOS is being implemented in the environment provided by the Programmer's Workbench (PWB/UNIX). This program-development, management, maintenance and testing tool provides the facilities needed to carry out the complex development and maintenance activities required by the KSOS project.

The development plan for KSOS requires that several small programming teams concurrently write and test portions of the system. Configuration control will be maintained through use of the PWB/UNIX's Source Code Control System (SCCS). SCCS will be used to control all forms of machine-readable text (e.g., design notes, formal specifications, implementations, test plans) created in conjunction with KSOS implementation. Use of SCCS will provide a complete audit trail of systems development, the ability to reconstruct any version of the evolving system and the basis for subsequent maintenance of KSOS.

Inspection and Review

The accuracy with which verified formal specifications can be implemented has been the subject of much concern to the KSOS project. How can one ensure that an implementation will perform exactly the specified function and no other? Complete code proofs (i.e., P2) of the implementation would perhaps answer this question, but they are not anticipated for KSOS.

The function of ensuring an accurate match between the formal specifications and their implementation is therefore assigned to a formal inspection process using the techniques described by Fagan. Two levels of formal inspection are planned. The first, called Design Completion Review, authorizes release of module designs for Critical Design Review by the KSOS customer. This review takes place when the detail design has reached the level that each design statement corresponds roughly to ten or fewer statements in the implementation language. The second inspection, called Code Completion Review, is scheduled after the first diagnostic-free compilation of the complete module.

The focus of each inspection is to ensure conformity of the detailed design and implementation to the proven formal design specification. Every module must pass these two inspections.

Testing for specification compliance

The quality assurance efforts for KSOS seek to provide a series of convincing demonstrations of the security and completeness of the system. Inspection and review are two contributors to quality assurance; testing is another.

The formal specifications are a useful guide to test case
selection. In particular, these test cases are selected such that there is at least one test case for:
  a. The TRUE and FALSE cases of every specified EXCEPTION condition.
  b. Both \( x = \text{TRUE} \) and \( x = \text{FALSE} \) conditions of every specified IF \( x \) THEN \( \ldots \) ELSE.
  c. Every specified \( (x) \Rightarrow Q \).
  d. Every possible type of \( x \) in every specified TYPE-CASE \( (x) \) OF \( \ldots \).

Testing of a function is based on the proven formal design specification of that function. Test cases are automatically generated from the formal specifications and are then subject to an inspection process similar to those used for inspection of design and of code.

IMPLEMENTATION PROOF

The final step required to complete the schema for production of proven systems is to prove that the implementation conforms to the proven specification. Hoare has shown how such proofs may be constructed. In practice, these proof techniques have been successfully applied to isolated algorithms (e.g., Reference 27) and have led to spectacular insights about program construction. However, there have not been any implementation proofs of operating systems.

All the necessary methodological preparations for a complete implementation proof of KSOS are being made. There exist, of course, formal specifications. The implementation language was chosen to allow the formulation of proof rules. A theorem prover (the same one which is used for the design proof) is at hand. The KSOS contract calls for axiomatization of the KSOS implementation language and for creation of the necessary verification condition generator. However, it requires only "illustrative" code proofs, i.e., only portions of the implementation will be proved.

These proofs will not be sufficient to complete P2, proof that the implementation conforms to the design. Nonetheless, they will serve a very important function. They will illuminate the state-of-the-art in automatic program proof, providing not only experience in the necessary techniques but also quantitative data as to the tractability and economics of proving large programs. There is no doubt that an estimate of the effort required to perform a complete implementation proof of KSOS will be made, based upon data derived from the illustrative proofs.

SUMMARY

The KSOS project is extremely significant in the field of program development methodology. It makes initial industrial use of a number of techniques which were previously used only in academic and research environments. In particular, it is novel in its large-scale use of formal specification, automatic theorem generation, language axiomatization and automatic theorem proof. These new techniques have been successfully integrated with more traditional ones such as programming teams, inspection and review. Careful utilization of these combined methodologies allows construction of a rigorous proof that the KSOS system meets its stringent security requirements. More generally, the methodology mix and the experience gained in applying them to KSOS open the way for routine construction of computer systems whose vital properties can be convincingly proven concurrently with the development of the system.

REFERENCES

8. Executive Order 11652, Office of the President, Washington, DC.
22. Dolotta, R. H. and J. R. Mashey, "The Programmer's


