—TCB penetration may be caused by flawed implementation of the penetration-resistance properties in trusted processes that may, otherwise, legitimately bypass the RMM (viz., Experiments 7 and 8). In such cases, the RMM is, by definition, unable to mediate these accesses;

—TCB penetration may be caused by inadequate context-dependent checking of parameters by trusted processes (viz., Experiment 9). In such cases, the RMM is unable to enforce TCB isolation properties, by definition, since it cannot be expected to understand context dependencies of trusted processes.

Experience with the penetration analysis method described above shows that the set of penetration-resistance properties for a TCB are a strict superset of the RMM requirements, and these properties have to be observed to ensure the security of the TCB and entire system. Furthermore, the tamperproofness and noncircumventability properties of a system (e.g., TCB) are a strict superset of the TMM requirements, in the sense that they apply to the whole TCB as opposed to only the RMM. Thus, a penetration scenario caused by a violation of (1) consistency of global system variables, (2) timing consistency of validation checks, and (3) undesirable system/user dependencies, may illustrate TCB penetration without RMM-penetration.

(2) Validation-Check Dependencies on Design and Programming Disciplines

The validation checks that should be included in a system's source code (and specified for use by the FDM **550**) depend on both the design and programming disciplines used in system development. Design-level validation dependencies are common to most operating systems, whereas programing-level dependencies are specific to the actual disciplines used. Identifying both of these types of dependencies is important for the correct (automated or manual) generation of validation-check specifications for use by the FDM **550**. Without the benefit of a correct set of validation-check specifications, the penetration analysis process loses precision.

Validation checks for penetration resistance are of two major types: interface validations involving parameters and privileges at the TCB interface, and functional (correctness) validations required elsewhere within the system. Typically, the isolation property dictates the validations required at the interface, whereas the other properties determine the validations required elsewhere.

(a) Examples of Design-level Dependencies

The penetration analysis experiments reveal several types of design level dependencies of validation-checks. Combinations of such dependencies may arise in specifying validation checks for individual system calls.

Call-Dependencies of Validation Checks: Validation checks (or condition check statements) required by penetration-resistance are sometimes dependent on the particular system call in which they occur. For example, in Unix, the global variable inode may be altered via both the mknod and write system calls. Due to the difference in the semantics of the two system calls, however, write alters an inode that already exists, whereas mknod alters an inode it has just created on behalf of the user invoking the systemcall. Thus, the validation-check specification for the altering flow through write must state that a validation of the write access to the inode is required (based on some policy (Note that the policy under which the access is deemed valid is irrelevant

to whether the validation is performed)). In contrast, the validation-check specification for the altering flow through mknod will not include this write access check. Validation checks that depend on the system call in which they occur are named "call dependent." It should be obvious that all interface validations are call dependent, since the very definition of the parameters and privileges are implicit within the system call in which they occur. However, functional validations may or may not be call dependent.

Type-Dependencies Validation Checks: Validation checks may also be dependent on the type of object the system kernel) is working on or on the type of command/argument specified at the user interface. For example, within the Secure Xenix system call semctl, user parameter are specifies a pointer to a buffer to be either read or written depending on the value of the parameter cmd. If cmd is equal to IPC_SET, the system call reads from the buffer, and hence a validation check for reading from the buffer is required (based on some policy); whereas if cmd is equal to IPC_STAT, the system call writes to the buffer, and hence a validation check for writing to the buffer is required (based on some policy). Thus, depending on the type of the parameter cmd, the parameter arg is validated in different ways and, thus, different validation specifications must be provided. Similarly, in the system call aclquery of Secure Xenix, which queries the access control list of an object, the validation checks are different for different object types specified via the user interface. This kind of dependence of a validation check is termed "type dependence."

Since "call dependence" and "type dependence" are both related to the context in which the user is requesting service from the kernel, we will group them together as context dependence. In general, validation-check specifications may be either "context dependent" or "context independent" (see FIG. **18**). The "context" of those specifications is provided by an execution path, which is determined by the kernel entry point and the entry point parameters.

Dependencies of Interface Validations: The validations required at the user interface include, (i) parameter validations, (ii) privilege validations and (iii) user/kernel address space separation checks. System call parameters can be passed either by value or by reference. The latter type of parameters have to be read into system space to convert them to parameters by value. During the reading in of these parameters, the system should check for (segment) read access as well as whether it is reading from the address space of the invoker, as illustrated in FIG. **18**. Parameters passed by value can be further classified as either "type independent" or "type dependent" (e.g., parameter arg in semctl), or as requiring no validation whatsoever (e.g., when a parameter specifies a numerical value that has no legality bounds associated with it).

Semantic Dependencies of Parameter Classes: During the analysis of the parameter validation for the Secure Xenix kernel, it has been determined that it is possible to classify the system call parameters based on call semantics, and that, in the majority of cases, parameters belonging to the same semantic class are validated in an almost identical way. Such classification simplifies the process of generation of the set of parameter validations that need to be performed at the user interface. For example, in Secure Xenix, there are a large number of system call parameters that specify a filename in the form of a character string. Every single such filename parameter is validated in exactly the same way (through the internal function namei) to ensure that the string indeed represents a valid filename in the system and that the calling process has search access to all the directories in the pathname. Similarly there are other obvious semantic classes such as read and write buffers, file descriptors, message or semaphore identifiers.

Theoretically, every single system call parameter can be placed in a semantic class and the interface validations required for each such class can be clearly specified. When a complete set of semantic classes is obtained, the automated analysis of system call parameter validation becomes possible for a given system by making the specifications of the required validation checks available to the Automated Penetration Analysis tool **500**.

Semantic classes with a large number of members are especially suitable for automated analysis. However, some of the semantic classes will have a single member, and hence, in those cases, automated analysis will involve as much effort as manual analysis of the source code. System call parameters that are type independent can usually be placed in large-sized semantic classes. In contrast, type dependent parameters are the ones that are the most difficult to classify and usually belong to single member semantic classes.

(b) Examples of Programming Discipline Dependencies for Trusted Processes

Explicit Object Sharing Among Trusted Processes: Trusted processes of Secure Xenix, and those of most Unix systems, rarely interact with each other except inside the kernel via system calls. Shared global variables between trusted processes exist in a few cases but, in general, trusted processes share no global variables, except in the form of shared (system) objects such as files, pipes, and these objects are shared through system calls. Since the kernel is analyzed for penetration-resistance separately, trusted processes that share no global program variables can be analyzed as separate entities. However, trusted processes that do share global program variables amongst themselves (such as the "lp" subsystem commands) must be analyzed collectively as a subsystem. Their relationships with respect to the shared objects must be explicitly defined such that the required validation checks can be specified.

Explicit Parameter Passing to Trusted Processes

Parameter Passing: Unlike the user specified parameters of kernel cells, the user-specified parameters of trusted processes are not as easily identified syntactically in the code. This is because a single variable might specify a string of parameters passed by reference. When these parameters are validated, the validation condition statements refer repeatedly to the same string identifier (the pointer to the string identifier is advanced successively to refer to the successive parameters referred by the string). Thus, it becomes very difficult to uniquely identify the individual user parameters, and hence to derive the syntactic form of the penetration-resistance specifications, whenever parameter passing is not explicit.

Interactive Input: Many user parameters to trusted processes are input interactively from the user interface and are copied into local variables of trusted processes before validation. The identification of which local variables receive user-specified values via interactive input should be explicit. Otherwise, deriving the penetration-resistance specifications becomes as difficult as analyzing the source code manually and is hence not cost effective.

Explicit Specification of Trusted Process Protection Mechanisms: The majority of the protection mechanisms used by the trusted processes depend on properties possessed by the executable modules of trusted processes and are not always apparent by an analysis of the code. Such mechanisms include setuid/setgid mechanisms, special.users and groups (representing administrative roles) with discretionary access to special administrative files and data structures, and special privileges which are endowed directly (and not explicitly acquired). The use of such mechanisms in trusted processes should be made explicit. Otherwise, a potentially large number of extra facts must be manually fed into the FDM **550** thereby reducing the degree of automation possible.

Whenever trusted processes are relatively small (in terms of lines of source code) it may be easier to perform the flaw detection manually by assuming that the trusted process is endowed with power which is not obvious through code inspection. In such cases, automated analysis of trusted processes may not always be cost effective. In these cases, trusted processes can be analyzed manually using the model and the penetration-resistance properties as guidelines.

(3) Interpreting the Penetration-Resistance Properties

To generate penetration-resistance (i.e., validation-check) specifications at the source-code level of a system, we must interpret the abstract penetration-resistance properties discussed in Section II above in source code. In conducting the experiments presented in Section V, it was determined that instead of trying to interpret the abstract properties in the source code directly in a single step, it is easier to first interpret these properties using internal design-level specifications. This additional step allows the present invention to generate a set of concrete properties from the abstract properties, through a study of the system documentation, and then to interpret these concrete properties in the source code. The interpretation of the abstract properties and the of integrated flow paths could be used to determine whether validation checks in source code satisfy the abstract properties. This approach would be advisable when an attempt is made to formally verify the penetration-resistance properties of a kernel.

System Isolation: Validation of system call parameters refers to legality checking of user specified parameters. This usually involves checking identifiers specifying system objects and path-names, address space separation checking for segment selectors, range checking for numerical values, etc. FIG. **7** illustrates a system call parameter validation flaw, where the user supplied parameter buf is not validated (to point to a readable segment in user space) before it is used as an argument to internal function copyseg which performs a copy operation between memory segments.

The semantics of a system call may dictate the type of check required. Parameter validation checks are necessary to ensure that the system does not behave abnormally even if supplied with unexpected or unallowed parameters. For system calls that may be invoked only by users possessing special privileges, privilege checking must be done (e.g., as illustrated in FIG. **13**(a), the calling process must possess the PRIV_MOUNT privilege in order to execute the smount system call).

System call parameters can be classified by call semantics as well as by type dependence. Parameter fname in FIG. **11**(*a*) and parameters spec and dir in FIG. **13**(*a*) are context independent parameters belonging to the same semantic class of filename strings. Validation checks are not required for parameters rwflag of FIG. **13**(*a*) and msgflg of FIG.

14(*a*). Parameter addr in FIG. **12**(*a*) is context independent and belongs to the semantic class of write buffers and so on for the other parameters.

Whenever the property of isolation is applied to the trusted processes, it mandates the validation of user-supplied parameters (e.g., length checking) and of privileges for the invocation of privileged functions. generation of design-level (concrete) properties in the context of Secure Xenix is illustrated below. For example, the abstract property of noncircumventability can be interpreted to generate the properties of inter-process signaling and inode alterability, among others. These design-level, penetration-resistance properties can then be used to derive the set of validation-check specifications for verifying that the actual source-code level checks for alter/view/invoke accesses are correctly performed within the system.

The process of interpreting abstract penetration-resistance properties in design- level specifications and then in source code is analogous to that of performing model interpretation in a systems' descriptive/formal top-level specifications (DTLS/FTLS) and source-to-code correspondence. However, the DTLS/FTLS differ from the design specifications need for interpreting penetration resistance properties. In general, the DTLS/FTLS are intended to define the system behavior in terms of user-level objects (e.g., processes, files, directories) and, therefore, do not include specifications of internal system behavior. Thus, DTLS/FTLS are usually not detailed enough to reveal the flaws that cause system penetrability. For example, no amount of analysis at the DTLS/FTLS level documentation would have revealed the timing consistency flaw illustrated in Experiment 6, or the use of copyseg to copy to/from a user supplied address illustrated in Experiment 4. Instead of DTLS/FTLS, internal system specifications should be used to derive the concrete properties of penetration- resistance from the abstract properties.

Referring to FIG. **19**, the penetration analysis of a given system can be done in either of two ways. You can start from the abstract properties to generate the concrete properties, which in turn may be used to generate the set of required source code level checks. This is the approach adopted in developing the Automated Penetration Analysis tool implementation, where the method of the present invention ensures that the required checks are indeed present in the integrated flow paths. Alternately, the set System Noncircumventability: The property of noncircumventability is exhibited within operating system kernels in various ways. For example, in Secure Xenix, reference to object status variables is equivalent to access to the various fields within the inode structure for a file; reference to object contents is equivalent to invoking internal functions readi and writei; and reference to object privileges can be thought of either as read/write access to the access control list (ACL) file, or access to the mode field of the inode variable. Reference to subjects (i.e. subject-to-subject signaling) is the same as access to the signal field within the process structure for the receiving subject. Noncircumventability is maintained in each of these cases if the access involved is preceded by a validation check to verify that the calling process has the adequate access privileges in a time-consistent manner. However, segment read/write validation check for user supplied addresses does not fall under our noncircumventability property; rather, it is a validation check for system-call parameters, and is imposed by the isolation property (which also requires validation checks for user/system address space separation for all user specified addresses).

As illustrated above, validation checks may be context (call or type) dependent, meaning that access to the same variable or internal function may require different access checks based on the "context" of the access. For example, the global proc→p_sig variable may be altered through system calls kill, rexit, and wait. However, interprocess signaling requires that the validation checks involved in each of these altering access paths be different since this is a call dependent check. The proc→p_sig variable is altered through rexit to send kill child-process signal to the parent process. Since in the context of this call it is known that the owner of the parent and child processes is one and the same, this validation check can be, and is, eliminated. Similarly, this validation check is unnecessary when the proc→p_sig variable is altered through the wait system call. This is the case because this variable is altered when the process structure of a "zombie" child is freed, and again, the owner of parent and child processes is one and the same. In contrast, as illustrated in FIG. **10**(a), the altering access to the proc→p_sig variable in the context of the system call kill must be preceded by a validation check to verify that the calling process has signaling access to the receiving process since there can be no a priori assumptions regarding the owners of the sending and receiving processes.

The validation check required to alter the contents of a file through invocation of internal function readi is also a call-dependent check. When readi is invoked through system call erece, the system verifies whether the calling process has execute access to the file in question (as illustrated in FIG. **11**(*a*)). However, invocation of readi through the read system call involves checking for the FREAD flag in the file table entry for the given file. Another validation-check example is illustrated in FIG. **14**(*a*), where write access to a given message queue is verified before altering the fields of the message structure for that queue.

In Secure Xenix, some trusted processes are endowed with special privileges to bypass the kernel access-control policy modules (i.e., the MAC_EXEMPT and DAC_EXEMPT privileges). These trusted processes must themselves enforce part of the noncircumventability policy in the bracketed sections of code where they possess these privileges. In other words, the property of noncircumventability translates to access-control policy checks inside trusted processes only in sections of code possessing special privileges that allow the TP to unconditionally "fall through" the kernel policy modules. Validation-check specifications must cover all checks in bracketed code areas.

Consistency of Global Variables: Validation checks that ensure consistency of global variables and tables usually include checking for table overflow and underflow, checking for duplicate table entries, and checking for availability of system resources before allocation. FIG. **13**(*a*) illustrates that the global variable mount→m_dev is altered only after checking that the mount table limits are not exceeded, and that the mount table does not contain duplicate entries. In FIG. **14**(*a*), we see that the global variable msgque→msg_cbytes (the number of bytes on the queue) is altered only after checking that there is available space on that queue to add the new message of the given size.

Validation checks for shared data structures defined only inside the trusted process modules must also be specified. These checks must ensure that the consistency of the shared global data structures is maintained.

Timing Consistency of Validation Checks: In a Unix type system, a kernel mode process is non-preemptible. In a single processor system, a kernel process relinquishes control of the processor only when it voluntarily goes to sleep or when it terminates. Thus, conditions validated by a kernel process are guaranteed to hold true as long as the kernel has control of the processor since no other process can change the variables involved in the validation check. To ensure timing consistency of validation checks, it is therefore required that all the validation checks associated with an information or function flow be conducted in an atomic sequence with the flow itself; i.e., the process should not invoke sleep in the middle of this sequence (viz., FIG. **20**).

The timing consistency property is more difficult to achieve in trusted processes than in kernels of Unix-based systems. In these systems, as in most other operating systems, the kernel maintains timing consistency mainly through a discipline of non-preemption. Thus, while integrated flow paths in the kernel are atomic unless there is an explicit function call to sleep, integrated flow paths within trusted processes are assumed to be non-atomic unless explicit steps are taken to ensure timing consistency of the objects or variables which are involved in the validation checks.

Note that the timing consistency property applies to the validation checks derived from all the other properties and, as such, need not be specified explicitly. The requirement for the timing consistency of the validation checks is derived directly from the penetration resistance model.

Undesirable System/User Dependencies: FIG. **9**(b) illustrates an undesirable system/user dependency that exists in an Unix type system where an ordinary user is able to crash the system by directly or indirectly invoking the internal function panic. This is viewed as unsatisfactory, since in a multi-user system, a crash could lead to denial-of-service to other users of the system.

VIII. Conclusion

The Automated Penetration Analysis tool based on the penetration-analysis method described above can be used to detect violations of additional penetration resistance properties. It may also be used for other Unix systems implementing the same set of properties. Furthermore, by merging new flows with old flows, and new checks with old checks of the same system (e.g., TCB), the tool can be used for incremental analysis (of penetration resistance) of updates. Lastly, it can be used for penetration analysis in other applications; e.g., database management systems.

The observations regarding the separability of the policy concerns from those of penetration resistance, and the insufficiency of the reference monitor mechanism in providing assurance regarding penetration resistance of a system, helps delimit the usefulness of the Reference Monitor properties in penetration analysis. Designs of secure systems can also benefit from the observation that the design and programming disciplines of a system have a large impact on the ease (or difficulty) of performing (automated) penetration analysis. Finally, the observation that the assurance process for penetration resistance is similar to that for policy implementations will allow the use of well-accepted assurance techniques for the purpose of penetration analysis.

While the invention has been particularly shown and described with reference to preferred embodiment thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the invention.

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is:

1. A method for preventing users of a computer system from exploiting system flaws to gain illegal or unintended access to system variables, objects and/or operations, comprising the steps of:

(1) generating a set of interpretation constants by applying a set of penetration-resistant properties to a given system, wherein said set of interpretation constants represent a database of required conditions, parameter validations and privilege checks that are associated with access to each abstract cell and critical function in said given system, wherein said given system has previously been determined to be penetration-resistant;

(2) generating an integrated flow path within said given system which records information regarding flows and condition checks that would be encountered along a given integrated flow path to an alter operation or a view operation on a particular abstract cell or an invoke operation on an internal system function;

(3) applying, in response to said alter operation, said view operation, or said invoke operation, a set of model rules to said given integrated flow path to determine whether said given integrated flow path conforms to said penetration-resistant properties, wherein said model rules are based on said interpretation constants: and

(4) allowing said alter operation, said view operation, or said invoke operation to proceed if said given integrated flow path was in conformity with said penetration-resistant properties.

2. The method of claim **1**, wherein said interpretation constants include an alter set, a view set, a critical function set, and an entry point set.

3. The method of claim **1**, wherein said penetration-resistant properties include (a) system isolation, (b) system noncircumventability, (c) consistency of system global variables, (d) timing consistency of system condition checks, and (e) elimination of undesirable user and system dependencies.

4. The method of claim **1**, wherein said integrated flow path is modified if all conditions of an alter model rule are met, said alter model rule comprising the steps of:

(a) determining whether said given integrated flow path includes all required entry point parameter validation checks;

(b) determining whether said given integrated flow path includes all required conditions that need to be checked to alter a given abstract cell; and

(c) determining whether all of said required conditions are checked and said given abstract cell is altered in a sequence that does not invoke a sleep function;

wherein said alter model rule is applied when said alter operation attempts to alter said given abstract cell through a system entry point along said given integrated flow path.

5. The method of claim **1**, wherein said integrated flow path as modified if all conditions of a view model rule are met, said view model rule comprising the steps of:

(a) determining whether said given integrated flow path includes all required entry point parameter validation checks;

(b) determining whether said given integrated flow path includes all required conditions that need to be checked to view a given abstract cell; and

(c) determining whether all of said required conditions are checked and said given abstract cell is viewed in a sequence that does not invoke a sleep function;

wherein said view model rule is applied when said view operation attempts to view said given abstract cell through a system entry point along said given integrated flow path.

6. The method of claim **1**, wherein said integrated flow path is modified if all conditions of an invoke model rule are met, said invoke model rule comprising the steps of:

(a) determining whether said given integrated flow path includes all required entry point parameter validation checks;

(b) determining whether said given integrated flow path includes all required conditions that need to be checked before a critical function can be legally invoked; and

(c) determining whether all of said required conditions are checked and said critical function is invoked in a sequence that does not invoke a sleep function;

wherein said invoke model rule is applied when said invoke operation attempts to invoke said critical function through a system entry point along said given integrated flow path.

* * * * *