# A METHOD AND SYSTEM FOR ENFORCING A SECURITY POLICY VIA A SECURITY VIRTUAL MACHINE

## TECHNICAL FIELD

[0001] The described technology relates generally to enforcing a security policy to block an undesired behavior by a computer program.

## BACKGROUND

[0002] Software systems, such as operating systems and file systems, provide application programming interfaces through which application programs can access the services of the software systems. An application program interface may provide functions with parameters for performing a specified service. For example, a file system may have a function (also referred to as a "system call") for creating a file. That function may have parameters that specify a file location, file name, file type, file size, and so on of the file to be created. When an application program invokes a function, it passes actual parameters corresponding to the formal parameters defined for the function. The function may perform certain checks on the actual parameters to ensure that they are valid. For example, the create file function may ensure that the specified file type is valid. If a parameter is not valid, then the function returns an error to the application program.

[0003] Software systems can be extremely complex for a variety of reasons. A software system may seek to be backward compatible with all prior versions of that software system. In such a case, the software system may need to support all the functions of the application program interface of the prior versions. Since new functions are typically added with each version, the number of functions can be very large and their interactions complex. Some software systems may also include software components developed by different divisions of the same

company or by different companies. The interactions of these components can also be quite complex.

[0004] It has become increasingly important for software systems to ensure that their application programming interfaces are not vulnerable to either inadvertent misuse or intentional attack. One vulnerability of a software system may be through the parameters of its application programming interface. When the functions of prior versions are combined with those of newer versions, and when components from different developers are integrated, the existing parameter validation performed by the functions may not be sufficient to ensure correct behavior of the functions. For example, an application programming interface of a file system developed by one company may be integrated with a file server developed by another company. The maximum file size parameter of the application programming interface may be larger than that supported by the file server. In such a case, a file size that is acceptable to the application programming interface may cause problems with the file server. As another example, a system administrator may want to limit the maximum file size even further, but the system administrator may have no available means to enforce that limitation.

[0005] It would be desirable to have a mechanism for enforcing security policies and, in particular, security policies that specify valid conditions for parameters of application program interfaces.

## SUMMARY

[0006] A method and system for enforcing a security policy is provided by a security virtual machine. The security virtual machine includes a processor engine, an instruction store, a data store, and an instruction pointer. The security virtual engine executes a security program specified using an intermediate language that is compiled from a high-level language representation of the security policy. The security program is loaded into the instruction store for execution. When a security enforcement event occurs, such as the invoking of a function of an application programming interface, data from the security enforcement event is

stored in the data store and the processor engine starts fetching and executing the instructions for the security program from the instruction store. The instructions specify actions to be taken to enforce the security policy based on the data of the security enforcement event.

## BRIEF DESCRIPTION OF THE DRAWINGS

[0007] Figure 1 is a block diagram that illustrates components of the security system in one embodiment.

[0008] Figure 2 is a block diagram that illustrates the data structure of the data store in one embodiment.

[0009] Figure 3 is a diagram that illustrates the layout of an instruction of the security virtual machine in one embodiment.

[0010] Figure 4 is a flow diagram that illustrates the processing of the load program component in one embodiment.

[0011] Figure 5 is a flow diagram that illustrates the processing of the system call intercept component in one embodiment.

[0012] Figure 6 is a flow diagram that illustrates the processing of the processor engine of the security virtual machine in one embodiment.

[0013] Figure 7 is a flow diagram that illustrates the processing of the dereference component of the security virtual machine in one embodiment.

[0014] Figure 8 is a flow diagram that illustrates the processing of the evaluate test component of the security virtual machine in one embodiment.

[0015] Figure 9 is a flow diagram that illustrates the processing of the test component of the security virtual machine in one embodiment.

[0016] Figure 10 is a flow diagram that illustrates the processing of the match string component of the security virtual machine in one embodiment.

[0017] Figure 11 is a flow diagram that illustrates the processing of the nontest component of the security virtual machine in one embodiment.

## DETAILED DESCRIPTION

[0018] A method and system in a computer system for enforcing a security policy encoded in an instruction set of a security virtual machine is provided. In one embodiment, a security system provides a security virtual machine that executes security programs expressed in the instruction set (i.e., intermediate language) of the security virtual machine. The security system may include a compiler that receives a security policy represented in a high-level language and generates a security program that is a representation of the security policy in the intermediate language of the security virtual machine. The security system stores the security program in an instruction store of the security virtual machine. When a security enforcement event occurs (i.e., an action that needs to be checked to ensure it complies with the security policy), the security virtual machine executes the instructions for the security program from its instruction store using data from the security enforcement event to enforce the security policy. If the security enforcement event indicates that an attempt is being made to perform an undesired behavior (e.g., an action that might exploit a vulnerability of an operating system), then the security program may block the attempt. In one embodiment, the security virtual machine of the security system executes in kernel mode of an operating system to identify and prevent attempts by applications and external systems to perform an undesired behavior for the computer system on which the security system is executing.

[0019] In one embodiment, the security system identifies when parameters of a system call issued by an application to a system service (e.g., file system and memory management system) may lead to an undesired behavior. The security policy contains rules that each specify a condition based on parameters of the system call and an action to take when the condition is satisfied. For example, the condition of a rule may be satisfied when a file create system call is issued with a parameter specifying a file size greater than 1GB. The action associated with that rule may be to block the creation of the file. The high-level language may be an XML-based language, and each rule may be identified by rule, condition, and action tags. When the rules of the security policy represented in the high-level

language are compiled, each rule is translated into instructions of the intermediate language to form the security program. For example, a rule may be translated into a series of instructions. One that compares the system call to a value indicating that the function is "file create," one that compares a file size parameter to a constant with the value of 1GB, and one that outputs an indication to block the system call.

[0020] In one embodiment, the security virtual machine includes a processor engine, an instruction store, an instruction pointer, and a data store. When initializing the security virtual machine to enforce the security policy, the security system loads the instruction store with a security program that implements the security policy. The security system also loads data of the security program into the data store. When a system call is received, the security system stores the parameters of the system call (including a value that identifies the system call) into the data store. The security system may also store process control information of the process that issued the system call into the data store. The security system initializes the instruction pointer to point to a start instruction of the security program. The processor engine starts the security program by fetching and executing the start instruction pointed to by the instruction pointer. Each instruction that is executed modifies the instruction pointer to point to the next instruction to be executed. The instructions reference the data of the data store to enforce the security policy. The execution of the security program generates an output action set that specifies the actions (e.g., block system call) that are to be taken to enforce the security policy.

[0021] In one embodiment, the security policy may comprise a sub-policy for each system call whose parameters are to be checked. The security system may compile each sub-policy into a separate security sub-program that can be loaded into the instruction store independently of the other security sub-programs. Each sub-policy may correspond to the security enforcement for a single system call. The security system may maintain a mapping of each system call to a start instruction pointer in the instruction store and a start data pointer to the data in the data store of the corresponding security sub-program. When enforcing the

security policy on a system call, the security system initializes the instruction pointer to the corresponding start instruction pointer and the data pointer to the corresponding start data pointer. The security sub-programs can use instruction and data reference techniques that are relative to the instruction pointer and the data pointer. In this way, the instructions and data of each security sub-program are relocatable. In the following description, a single security program handles all system calls by decoding each system call and jumping to the appropriate instruction location to process that system call. One skilled in the art will appreciate that this described system can be adapted to support a sub-program for each system call.

[0022] Figure 1 is a block diagram that illustrates components of the security system in one embodiment. The security system includes some components that execute in user mode 100 and others that execute in kernel mode 120. A policy compiler 102 is a component that executes in user mode to compile security policies of a policy store 103 into security programs that are executable by the security virtual machine. The kernel mode components include a system call intercept component 121, a load program component 122, and a security virtual machine 125. The security virtual machine includes an instruction pointer 126, a processor engine 127, an instruction store 128, and a data store 129. The load program component loads security programs compiled by the policy compiler into the instruction store of the security virtual machine, stores program data into the data store, and sets the start instruction pointer. The system call intercept component intercepts system calls issued by application programs 101 and stores the parameters of each system call and process control information of the application programs into the data store. After intercepting a system call and storing its parameters, the system call intercept component instructs the processor engine to execute the security program of the instruction store. The processor engine executes the security program by fetching instructions pointed to by the instruction pointer and performing the operations specified by the instructions. The operations include storing actions to be taken to enforce the security policy in an output action set of the data store. Upon completion of the execution of the

security program, the system call intercept component performs the actions of the output action set.

[0023] The computing device on which the security system is implemented may include a central processing unit, memory, input devices (e.g., keyboard and pointing devices), output devices (e.g., display devices), and storage devices (e.g., disk drives). The memory and storage devices are computer-readable media that may contain instructions that implement the security system. In addition, the data structures and message structures may be stored or transmitted via a data transmission medium, such as a signal on a communications link. Various communications links may be used, such as the Internet, a local area network, a wide area network, or a point-to-point dial-up connection.

[0024] Figure 1 illustrates an example of a suitable operating environment in which the security system may be implemented. This operating environment is only one example of a suitable operating environment and is not intended to suggest any limitation as to the scope of use or functionality of the security system. Other well-known computing systems, environments, and configurations that may be suitable for use include personal computers, server computers, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, programmable consumer electronics, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

[0025] The security system may be described in the general context of computer-executable instructions, such as program modules, executed by one or more computers or other devices. Generally, program modules include routines, programs, objects, components, data structures, and so on that perform particular tasks or implement particular abstract data types. Typically, the functionality of the program modules may be combined or distributed as desired in various embodiments.

[0026] Figure 2 is a block diagram that illustrates the data structure of the data store in one embodiment. The data store includes a literal data structure 201, a dynamic data structure 211, a system call parameters data structure 221, a

process control block data structure 231, and an action output set 241. The literal data structure includes a literal data table 202 and a literal data store 203. The literal data table contains fixed-size entries that reference literal data stored in the literal data store. "Literal data" refers to data of the security program that is constant for the life of the currently loaded security policy. Literal data can only be changed by loading a new security policy. The dynamic data structure has a similar organization to the literal data structure, but it stores "dynamic" data rather than "literal data." "Dynamic data" refers to data of the security program that can be modified without loading a new security policy. For example, dynamic data may include the names of applications that are exempt from the security policy. Each entry of the literal and dynamic data tables points to data in the corresponding data store. The data of the literal and dynamic data stores are represented in a self-describing format. That format includes data type and data size information. In one embodiment, the security virtual machine supports data types of Boolean, integer, string, and raw binary (also referred to as "blob") in addition to arrays of integers, strings, and blobs. Instructions reference the literal and dynamic data using indexes into the literal and dynamic tables. When executing an instruction, the security virtual machine dereferences the index to the literal and dynamic table by retrieving the indexed entry and using its value as a pointer to the start of the data item in the literal or dynamic data store. Since data is self-describing, the type and size of data can be determined. The system call parameters data structure includes a string table 222, an integer table 223, and a raw binary table 224. The parameters of the system call are stored in these tables according to their data type. The process control block data structure is a table that contains an entry for each piece of process control information that is available to the security virtual machine. In one embodiment, the process control information includes an application identifier and a thread identifier. The action output set may be a set of action flags that is generated during execution of the security program and that indicate actions to be performed to enforce the security policy on the system call. The actions may include blocking the system call and notifying a user.

[0027] Figure 3 is a diagram that illustrates the layout of an instruction of the security virtual machine in one embodiment. Each instruction includes an operation field 310, a parameter1 field 320, a true branch field 330, a false branch field 340, and a parameter2 field 350. The parameter1 field references data associated with the system call (i.e., stored in the system call data structure or in the process control data structure), and the parameter2 field references data associated with the security program (i.e., immediate data or data stored in the literal or dynamic data structures). The operation field includes a parameter2 descriptor 311 ("m") and an operation code 312. The parameter2 descriptor specifies how to interpret parameter2. Parameter2 can contain immediate data, a reference to dynamic data, or a reference to literal data. When a reference is specified, the security virtual machine dereferences parameter2 before performing the operation of the instruction. The operation code identifies the operation of the instruction as defined below in more detail. The parameter1 field contains a parameter1 descriptor 321 ("s") and an index 322. The parameter1 descriptor indicates whether parameter1 refers to a system call parameter or to process control information. If the parameter1 descriptor indicates a system call parameter, then the high-order bits of the index specify whether the string table, integer table, or raw binary table of the system call parameters data structure is to be indexed, and the low-order bits indicate the indexed entry of the table. If the parameter1 descriptor indicates process control information, then the index indicates the specific process control information. The true branch field and the false branch field contain offsets that specify the next instruction to execute depending on whether the condition code of this instruction evaluates to true or false. The offset is added to the current instruction pointer to point to the next instruction to be executed.

[0028] Tables 1 and 2 illustrate the instructions of the security virtual machine in one embodiment. The instructions can be classified as those that perform a test on data and those that do not. Table 1 illustrates the nontest instructions.

TABLE 1

| Operation Code | Description | Parameters |
|---|---|---|
| HALT | Stops processing | **p2**: status code |
| ACTION | Adds the action code to the output action set | **p2**: action code |
| JMP | Jumps to new instruction | **p2**: jump offset from current location |
| RESET | Clears the output action set | **p2**: ignored |

[0029] Table 2 illustrates the nontest instructions. The pattern instruction may define a regular expression for comparing two strings, which is useful for wildcard-type comparisons of file names (e.g., "*.DAT").

TABLE 2

| Operation Code | Description |
|---|---|
| TEST | boolean: test if **p2** is TRUE |
| CMPEQ | integer: compare **p1==p2** |
| CMPNE | integer: compare **p1!=p2** |
| CMPLT | integer: compare **p1<p2** |
| CMPLE | integer: compare **p1<=p2** |
| CMPGE | integer: compare **p1>=p2** |
| CMPGT | integer: compare **p1>p2** |
| BLOB_MATCH | blob: **p1==p2** |
| STRING_MATCH | string: **p1==p2** |
| PATTERN_MATCH | string: **p1** matches **p2** wildcard-pattern |
| BLOB_MEMBEROF | blob: **p1==p2**[i] |
| INT_MEMBEROF | integer: **p1==p2**[i] |
| STRING_MEMBEROF | string: **p1==p2**[i] |
| PATTERN_MEMBEROF | string: **p1** matches **p2**[i] wildcard-pattern |

[0030] Figure 4 is a flow diagram that illustrates the processing of the load program component in one embodiment. The component is invoked when a security program is to be loaded into the security virtual machine. A security program contains virtual instructions, a literal data block, and a dynamic data block. The literal data block includes the information of the literal data table followed by the information of the literal data store. The component copies the literal data block to the literal store starting at a start location. The component then adds that base location to each offset in the literal data table to convert the offsets to pointers. The component processes the dynamic data block in a similar way. The virtual instructions contain offsets that are relative to the start of the literal store and the dynamic store. In block 401, the component identifies a start location within the literal store and copies the literal data block of the security program starting at that start location. In decision block 402, the component adds the start location to each offset of the literal data table to convert the offsets of the security program to pointers. In block 403, the component identifies a start location within the dynamic store and copies the dynamic data block of the security program starting at that start location. In block 404, the component adds the start location to each offset of the dynamic data table to convert the offsets of the security program to pointers. In block 405, the component copies the virtual instructions of the security program to the instruction store and then completes.

[0031] Figure 5 is a flow diagram that illustrates the processing of the system call intercept component in one embodiment. The component is invoked when a system call is intercepted. The component initializes the system call parameter data structure and the process control data structure and then starts the security virtual machine. The component may also provide an instruction pointer of the instruction to start executing to process the system call. When security sub-programs are used, the component may have a mapping of system calls to the start instruction pointers and to start data pointers to the literal and dynamic data structures for that system call. The load program component may be called multiple times to load different sub-programs of the security program for processing different system calls. Since the instructions and data are offset

based, the instructions and data can be stored in the next available locations of the instruction store and data structures. In block 501, the component sets the current instruction pointer to the start instruction pointer of the security program and resets the action output set to an initial state (e.g., empty). In blocks 502-504, the component loops storing the parameters of the system call into the system call parameters data structure. In block 502, the component selects the next parameter of the system call. In decision block 503, if all the parameters of the system call have already been selected, then the component continues at block 505, else the component continues at block 504. In block 504, the component stores the selected parameter in the appropriate table of the system call parameters data structure and then loops to block 502 to select the next parameter of the system call. In blocks 505-507, the component loops storing process control information in the process control data structure. In block 505, the component selects the next process control information for the process that placed the system call. In decision block 506, if all the process control information has already been selected, then the component continues at block 508, else the component continues at block 507. In block 507, the component stores the selected process control information into the process control data structure and then loops to block 505 to select the next process control information. Alternatively, rather than using and initializing an internal process control data structure, the security system may retrieve process control information directly from a process control block when needed. One skilled in the art will appreciate that a process control block is created by an operating system when a process is created. In block 508, the component launches the security virtual machine to enforce the security policy on the intercepted system call. Upon completion of the execution of the security program by the security virtual machine, the component performs the actions of the action output set.

[0032] Figure 6 is a flow diagram that illustrates the processing of the processor engine of the security virtual machine in one embodiment. The processor engine initializes the system call parameter data structure based on the intercepted system call, launches the security virtual machine, and performs the actions of the

output action set. In blocks 601-607, the processor engine loops loading and executing the instructions of the security program stored in the instruction store until a halt instruction is executed. In one embodiment, the security policy may define that, whenever a condition of a rule is satisfied, no other rules are tested. In block 601, the processor engine fetched of the instruction pointed to by the current instruction pointer. In block 602, the processor engine invokes a component to dereference parameter1 and parameter2. In decision block 603, if the operation code of the instruction is a test operation, then the processor engine continues at block 604. If the operation code of the instruction is a nontest operation (except for a halt operation), then the processor engine continues at block 608. If the operation code of the instruction is a halt operation, then the processor engine continues at block 609. In block 604, the processor engine invokes the evaluate test component to determine whether the test operation evaluates to true or false. The invoked component sets a condition code flag to true or false. In decision block 605, if the condition code is true, then the processor engine continues at block 607, else the processor engine continues at block 606. In block 606, the processor engine adds the offset of the false branch field to the current instruction pointer and then loops to block 601 to fetch the next instruction. In block 607, the processor engine adds the offset of the true branch field to the current instruction pointer and then loops to block 602 to fetch the next instruction. In block 608, the processor engine invokes the perform nontest component and then loops to block 601 to fetch the next instruction. The perform nontest component performs the operation of the instruction and sets the current instruction pointer to point to the next instruction to be executed. The perform non-test component adds the offset of the true branch (or parameter2 in the case of a jump instruction) of the instruction to the instruction pointer. In block 609, the component performs the halt instruction and then completes.

[0033] Figure 7 is a flow diagram that illustrates the processing of the dereference component of the security virtual machine in one embodiment. This component dereferences parameter1 and parameter2 of the fetched instruction. In blocks 701-708, the component dereferences parameter1. In decision block 701, if

parameter1 is a system call parameter as indicated by the parameter1 descriptor, then the component continues at block 703, else the component continues at block 702. In block 702, the component sets the dereferenced parameter1 to the process control information specified by the index of parameter1 and then continues at block 709. In decision block 703, if the index of parameter1 indicates that the system call parameter is an integer, then the component continues at block 704, else the component continues at block 705. In block 704, the component sets the dereferenced parameter1 to the integer specified by the index and continues at block 709. In decision block 705, if the index of parameter1 indicates that the system call parameter is a string, then the component continues at block 706, else the component continues at block 707. In block 706, the component sets the dereferenced parameter1 to the string specified by the index and continues at block 709. In decision block 707, if the index of parameter1 indicates that the system call parameter is raw binary, then the component continues at block 708, else an error has occurred. In block 708, the component sets the dereferenced parameter1 to the raw binary specified by the index and continues at block 709. In blocks 709-714, the component dereferences parameter2. In decision block 709, if the parameter2 descriptor indicates immediate data, then the component continues at block 710, else the component continues at block 711. In block 710, the component sets the dereferenced parameter2 to the value of parameter2 in the fetched instruction and then returns. In decision block 711, if the parameter2 descriptor indicates literal data, then the component continues at block 712, else the component continues at block 713. In block 712, the component sets the dereferenced parameter2 to the literal data specified by parameter2 in the fetched instruction and then returns. In decision block 713, if the parameter2 descriptor indicates dynamic data, then the component continues at block 714, else an error has occurred. In block 714, the component sets the dereferenced parameter2 to the dynamic data specified by parameter2 in the fetched instruction and then returns.

[0034] Figure 8 is a flow diagram that illustrates the processing of the evaluate test component of the security virtual machine in one embodiment. The component

decodes the operation code of the loaded instruction and invokes a component to implement a test of that operation code. The invoked component sets the condition code to true or false. In decision blocks 801-803, the component decodes the operation code of the loaded instruction. In blocks 804-806, the component invokes the components to implement the decoded operation code and then returns.

[0035] Figure 9 is a flow diagram that illustrates the processing of the test component of the security virtual machine in one embodiment. The test component implements the test operation code. In block 901, the component sets the condition code to the value of the dereferenced parameter2 and then returns.

[0036] Figure 10 is a flow diagram that illustrates the processing of the match string component of the security virtual machine in one embodiment. The component determines whether the dereferenced parameter1 matches the dereferenced parameter2. In one embodiment, the security virtual machine may use pattern matching. For example, a parameter may include a "wild card" specification or more generally a regular expression. In decision block 1001, if the lengths of parameter1 and parameter2 match, then the component continues at block 1003, else the dereferenced parameters cannot match and the component continues at block 1002. In block 1002, the component sets the condition code to false and then returns. In blocks 1003-1006, the component loops checking each character of the strings of the dereferenced parameters. In block 1003, the component selects the next character of each string. In decision block 1004, if all the characters of the strings have already been selected, then the component continues at block 1008, else the component continues at block 1005. In block 1005, the component normalizes the selected characters. For example, the component may put each character into lowercase to implement a case insensitive comparison. In decision block 1006, if the selected characters match, then the component loops to block 1003 to select the next character of the strings, else the component continues at block 1007. In block 1007, the component sets the condition code to false and then returns. In block 1008, all the characters of the strings match and the component sets the condition code to true and then returns.

[0037] Figure 11 is a flow diagram that illustrates the processing of the nontest component of the security virtual machine in one embodiment. The component performs the operation of the nontest operation code of the fetched instruction, including setting the current instruction pointer to point to the next instruction to execute. In decision block 1101, if the operation code is an action operation, then the component continues at block 1102, else the component continues at block 1103. In block 1102, the component adds parameter2 of the fetched instruction to the action output set and then continues at block 1107. In decision block 1103, if the operation code is a reset operation, then the component continues at block 1104, else the component continues at block 1105. In block 1104, the component clears the action output set and then continues at block 1107. In decision block 1105, if the operation code is a jump operation, then the component continues at block 1106, else the component continues to decode further operation codes. In block 1106, the component adds parameter2 to the current instruction pointer and then returns. In block 1107, the component adds the value of the true branch field to the current instruction pointer and then returns.

[0038] One skilled in the art will appreciate that although specific embodiments of the security system have been described herein for purposes of illustration, various modifications may be made without deviating from the spirit and scope of the invention. One skilled in the art will appreciate that a security system using a security virtual machine can be used to enforce a wide variety of security policies. For example, the security system can be used to enforce security of messages received via a network, transactions received by a transaction processor, and more generally any application that provides an application programming interface. Accordingly, the invention is not limited except by the appended claims.

## CLAIMS

I/We claim:

[c1] 1. A method in a computer system for enforcing a security policy, the method comprising:

providing a security policy;

compiling the security policy into a security program based on an instruction set of a security virtual machine;

loading the security program into an instruction store of the security virtual machine; and

upon occurrence of a security enforcement event, executing the instructions of the instruction store based on data of the security enforcement event to enforce the security policy.

[c2] 2. The method of claim 1 wherein the security policy specifies security for system calls.

[c3] 3. The method of claim 1 wherein the security enforcement event is an application placing a system call.

[c4] 4. The method of claim 3 wherein parameters of the system calls are the data of the security enforcement event.

[c5] 5. The method of claim 3 wherein process control information of an application that places the system call is the data of the security enforcement event.

[c6] 6. The method of claim 1 wherein data of the security program is stored in a literal data structure.

[c7] 7. The method of claim 6 wherein an instruction contains a reference to literal data stored in the literal data structure.

[c8] 8. The method of claim 1 wherein data of the security program is stored in a dynamic data structure.

[c9] 9. The method of claim 8 wherein an instruction contains a reference to dynamic data stored in the dynamic data structure.

[c10] 10. The method of claim 1 wherein an instruction identifies a location of the next instruction to execute.

[c11] 11. The method of claim 1 wherein the security virtual machine supports Boolean, integer, string, and raw binary data types.

[c12] 12. The method of claim 1 wherein the security virtual machine supports regular expression pattern matching comparisons.

[c13] 13. The method of claim 1 wherein the execution of the security program creates an output action set that specifies how to handle the security enforcement event.

[c14] 14. The method of claim 1 wherein the executing of the instructions is performed in kernel mode.

[c15] 15. The method of claim 1 wherein an instruction includes an operation code, parameters, and branch fields.

[c16] 16. The method of claim 15 wherein the branch fields include a true branch field that specifies the next instruction when a condition of the instruction

evaluates to true and a false branch field that specifies the next instruction when the condition of the instruction evaluates to false.

[c17] 17. The method of claim 1 wherein the security enforcement specifies a behavior to block.

[c18] 18. A security virtual machine for detecting when an application is placing a system call with parameters that violate a security policy, comprising:

an instruction store that contains instructions that implement the security policy;

a data store that contains data of the security policy;

a parameter store that contains parameters of a system call; and

a processor engine that executes the instructions of the instruction store using data of the data store and parameters of the parameter store to determine whether the system call violates the security policy.

[c19] 19. The security virtual machine of claim 18 wherein the instructions that implement the security policy are compiled from a representation of the security policy in a high-level language.

[c20] 20. The security virtual machine of claim 19 wherein the high-level language is in an XML-based format.

[c21] 21. The security virtual machine of claim 19 wherein the high-level language has rules that are specified in terms of a condition and an action to perform when the condition is satisfied.

[c22] 22. The security virtual machine of claim 18 wherein the security virtual machine executes in kernel mode.

[c23] 23. The security virtual machine of claim 18 wherein a system call intercept component stores parameters of an intercepted system call in the parameter store.

[c24] 24. The security virtual machine of claim 18 wherein a load program component loads instructions into the instruction store.

[c25] 25. The security virtual machine of claim 18 wherein the instruction set is a reduced instruction set.

[c26] 26. The security virtual machine of claim 18 including a process control information store for storing process control information that is used when executing the instructions.

[c27] 27. The security virtual machine of claim 18 wherein the data store includes a literal data structure and a dynamic data structure.

[c28] 28. The security virtual machine of claim 18 wherein instructions contain a reference to the data store.

[c29] 29. The security virtual machine of claim 18 wherein each instruction identifies a location of the next instruction to execute.

[c30] 30. The security virtual machine of claim 18 wherein the security virtual machine supports Boolean, integer, string, and raw binary data types.

[c31] 31. The security virtual machine of claim 18 wherein the virtual machine supports regular expression pattern matching comparisons.

[c32] 32. The security virtual machine of claim 18 wherein the execution of the instructions creates an output action set that specifies how to handle the enforcement of the security policy.

[c33] 33. The security virtual machine of claim 18 wherein an instruction includes an operation code, parameters, and branch fields.

[c34] 34. The security virtual machine of claim 18 wherein the branch fields include a true branch field that specifies the next instruction when a condition of the instruction evaluates to true and a false branch field that specifies the next instruction when the condition of the instruction evaluates to false.

[c35] 35. A computer-readable medium containing instructions for enforcing a security policy, the instructions for execution by a security virtual machine and being compiled from a high-level language representation of the security policy.

[c36] 36. The computer-readable medium of claim 35 further containing a data store for data used to enforce the security policy.

[c37] 37. The computer-readable medium of claim 35 wherein the data store includes literal data and dynamic data.

[c38] 38. The computer-readable medium of claim 35 wherein the instructions form a relocatable security sub-program.

[c39] 39. The computer-readable medium of claim 35 wherein each instruction identifies a next instruction to execute.