DESIGN ISSUES: A VIRTUAL MACHINE FOR SECURITY IN A RELATIONAL DATABASE MANAGEMENT SYSTEM

Wynand van Staden
RAU-Standard Bank Academy for IT
Rand Afrikaans University
Johannesburg
`wvs@adam.rau.ac.za`

Martin S Olivier[1]
Department of Computer Science
University of Pretoria
Pretoria
`http://www.mo.co.za/`

ABSTRACT

Virtual Machines (VMs) have been applied in many situations in order to solve computing problems. One of the more interesting aspects of the application of VMs is in the arena of computer security. A prime example of the application of VMs as security mechanisms is that of packet filters, in which theVM is used to execute domain specific programs to determine the fate of all packets that arrive on a network interface.

Another example in which programs are used to enforce security is the Java VM. The Java security model makes use of a `SecurityManager` class that is used by the Java libraries to ensure that local security policies aren't violated. The `SecurityManager` class is compiled to Java bytecode that is executed by the Java VM in order to enforce security. This clearly indicates that using programmable security in a VM in a commercial environment can be accomplished readily.

In a similar fashion it is possible to create a VM that can be used for access control in a relational database management system. The use of a VM in such a scenario can provide more expressiveness and flexibility than standard access control mechanisms are capable of.

This paper reflects upon the nature of such a VM by considering aspects of the VM that are important for its application in a relational database management system. This is done in order to create a framework on which a possible implementation can be built.

[1]This material is based upon work supported by the National Research Foundation under Grant number 2054024. Any opinion, findings and conclusions or recommendations expressed in this material are those of the author(s) and therefore the NRF does not accept any liability thereto.

1 INTRODUCTION

The implementation of security in a Relational Database Management System (RDBMS) is (in most cases) accomplished using rigid structures that store the access rights that users have on objects in the database. Along with a relational algebraic language such as SQL, access control can be accomplished in a very efficient and manageable way.

Unfortunately because of the strict nature of the structure storing the access control information, it is often difficult to implement complex access control policies. The implementation of these access control policies are typically removed to an application layer.

This illustrates that programmable security would certainly provide significant benefits with regard to expressiveness as far as access control is concerned. This idea is of course not new at all, and two superb examples are that of the security manager class available to the Java VM, and the BSD Packet Filter (BPF) which is used to determine if data arriving on a network interface should be accepted or rejected.

It is possible to view SQL statements in a similar fashion as network packets. Whereas network packets contain information such as the originating host IP etc., SQL statements contain information such as the objects that access are requested to, the user that issued the access request (provided by the `AuthID` SQL built-in variable [1]), and the access mode that is requested (read, update, insert, etc.). Thus, given the power that programmable security would impart to the access control process, it is desirable to create a VM which can be used for access control in an RDBMS.

In this paper, the design issues of such a VM are considered. These considerations entail the most important aspects of a VM in general, such as the architecture of the VM, instruction size, the size of the instruction set, the amount of addressable memory, the supported registers, supported data-types, domain specific variables, and the possible extension of the VM's capabilities.

These aspects are considered to be of great importance since each affects the effectiveness of the VM, as well is its capability to provide the desired functionality – access control.

It must be emphasised, however, that this paper reflects upon aspects in order to provide guidelines for a possible implementation. It is not intended that this paper should describe a standard which should be followed when attempting to implement such a mechanism.

The rest of this paper is structured in the following manner. Section 2 provides background information on security applications of VMs as well as the functioning of access control in relational databases. Section 3 examines the architectural considerations of the VM. Section 4 considers the instruction size of the VM, and in doing so it considers aspects such as the number of supported operations, the amount of addressable memory, and the number of supported registers. Section 5 considers the data types the VM should support. Section 6 reflects upon the environmental variables that should be provided to Access Control Programs (ACPs), section 7 considers the possibilities of extending the functionality of the VM by allowing the incorporation of pre-written binary functions. Finally section 8 concludes the paper.

2 BACKGROUND

Access control in RDBMSs is based on a model introduced by Griffiths and Wade in 1976 [2]. This model allows for the storage of access control information in a relation in the system catalogue [2–4]. This relation is referred to as the *authorisation ruleset*. Important data such as the objects a user has access to, the mode of access users are allowed on that object, as well as the user that granted the access and the time at which it was granted is stored in the authorisation ruleset [3, 5].

Access control takes place when an SQL statement is passed to the Access Control Subsystem (ACS). A device commonly referred to as the *reference monitor* [3, 6] consults the authorisation ruleset to determine if access can be granted based on the SQL statement that is presented to the ACS. The information presented to the reference monitor will reflect the user, the object and access mode referred to in the previous section. Additionally, in the case of Role Based Access Control (RBAC), session information may also be present. Unfortunately, the authorisation ruleset does not allow for complex access control policies to be implemented. It has therefore been extended [7, 8], to allow for more efficient access control.

These models have proven effective; however the unconventional requirements of database vendors often force the implementation of access control policies on an application layer level. This access control layer can be incorporated effectively in the access control system by making use of a VM. An example of using a VM for "access control" is the UNIX network packet filter first suggested by Mogul et al [9] as the CMU/Stanford Packet Filter (CSPF). The basic idea was to place the filtering portion of the packet filter in the UNIX kernel, removing the need to perform context switches before being able to examine the network data[2].

In order to accomplish this, Mogul et al proposed that a VM capable of executing small programs be placed in the kernel as part of the packet filter core. Whenever network data arrived, these small programs would execute, returning (among others) either a true or false value indicating that a packet should be delivered to its endpoint or not.

The CSPF unfortunately had some severe shortcomings and was replaced by the BPF. The BPF introduced by McCanne et al [10], makes use of a VM to execute small bytecode programs that can accept or deny data packets based on information such as the originating address, the destination as well as the source port and destination port; there are of course other information which can also be used. The BPF uses a register machine which makes it more efficient than the CSPF [10].

The BPF supports roughly twenty-two instructions [10]. Thus it has a small instruction set, which allows for rapid execution. The instruction set is also rich enough to support packet filtering.

Another example of security through VMs is the Java VM [11], which provides a special class for the security management of applets [12]. Whenever an applet calls a function that executes a privileged instruction (such as write to disk, or read from disk, or making a network connection to a remote machine) the security class is invoked. Methods within this class examine a local

[2]The context switch was necessary because network data had to be copied into user space where a user application programme examined the network data to determine if it could be accepted or not.

security policy to determine if the instruction can be executed.

3 ARCHITECTURE OF THE VM

Perhaps the most important choice to make when designing any piece of software is the architecture. The important choice regarding architecture for the VM considered here is whether or not it is based on a stack machine or on a register machine.

A stack machine makes use of a stack to place operands, as well as operations of a program on. Execution takes place by popping the operand (operands) and operator from the stack, performing the operation and pushing the result back on the stack.

Unfortunately this method makes it virtually impossible to successfully implement branching instructions, and therefore all the operations (consisting of operators and operands) have to be evaluated. This means that execution of the program can only stop when all of the statements in the program has executed [10]. A stack architecture thus has severe overhead, and will not be considered in the rest of this paper.

A register machine can perform branching more easily and will therefore allow programs to terminate without executing unnecessary statements. For rapid response when determining authorisation for an access request, it is therefore assumed that the VM will be based on a register machine.

Another important aspect which also has to be considered is whether or not the VM supports procedure calls. Even though it may be argued that this is not a distinct architectural choice, it is discussed here since it will most definitely influence implementation issues of the architecture.

3.1 Procedure calls

Allowing ACPs to call other ACPs as procedures allows the implementation of complex access control policies, since ACPs can be strung together forming one complex ACP. This means, however, that management of access control policies will involve the Database Administrator (DBA) or System Security Officer (SSO) modifying ACP source code in order to modify the access control policy's implementation.

A better way to accomplish this is by extending the authorisation ruleset in such a way that a single access request triggers a set of ACPs to be executed [13]; all these ACPs have to return true if the access request is to be granted. Using this technique, modification of the access control policy only involves modification of the set of ACPs, clearly resulting in easier access control management.

It is therefore not expressly necessary that the VM should provide procedure call support, and for the purposes of the VM discussed in this paper it is not considered further.

4 INSTRUCTION SIZE

The instruction size of the VM, *ie* the number of bits that make up each instruction in an ACP, is dependent on the number of operations supported by the VM, the number of registers provided by the VM, and the total amount of addressable memory [14].

There are two different categories in which instruction sizes can be placed. Firstly there is a fixed instruction size, in which the supported operations, registers and amount of available memory is carefully selected in order to ensure that every instruction will be of a predetermined fixed number of bits.

The second method is a variable length instruction size, in which the size of each instruction is determined by the particular operation for that instruction.

An implementation of a VM may therefore choose to have a fixed instruction size, catering for a more generic set of instructions being used, or the implementation may alternatively choose to have a variable length instruction size.

Since the application domain of the VM – access control – is extremely specific, it stands to reason that there should be no need for large amounts of addressable memory, or an extremely large register set. Programs should ideally be relatively small for rapid execution. As far as the number of operations are concerned, it is once again noted that the target domain of the VM is nowhere near as general as that of VMs like the Java VM.

The Java VM supports no more than two-hundred and fifty-six instructions. It is therefore doubtful that the access control VM should support an instruction set nearly as large as that, thus both the fixed length and variable length instruction sets are viable options.

The following sections provide a brief consideration of the aspects which influences the size of the VM's instruction set.

4.1 Size of the instruction set

The VM should support a rich but small instruction set. It is after all the size of this instruction set which will determine the capability of an ACP to support a particular access control policy.

Another point to consider is possible future expansion of the virtual machine to include a larger collection of fundamental instructions.

It is clear that the fundamental instructions that should be supported by the VM, can in large be based on the instruction set supported by a more general VM. Clearly the VM must also include instructions that are specific to access control processing. The instructions that must be supported by the VM can be placed in nine categories:

1. Moving values between registers and variables.
2. Branching operations (conditional and unconditional)
3. Integer and Floating point arithmetic.
4. Bit manipulation operations (such as AND and OR).
5. Instructions that terminate an ACP.
6. Variable declaration
7. Instructions to parse SQL statements.
8. Instructions to execute extensions to the VM fundamental instructions.
9. Operations to execute SQL statements from within the ACPs as well as operations to navigate the result of these statements.

As is expected, the operations from the first four categories operate in the same way as the more general VM. These operations will therefore not be discussed in this paper. The operations from the last five categories are specific to a VM that supports access control, and will thus be considered in more detail below.

Termination Termination of ACPs happen when it can be determined that access should be allowed or denied. A statement indicating that access should be allows returns a **true** value to the reference monitor, otherwise a **false** value is returned. Presently, it is assumed that the termination operation only returns a **true** or **false** value.

It is conceivable that this return value can be extended to include a value of **maybe**, to indicate that at present the ACP cannot determine if access should be granted or not. This may be effectively employed in a transaction based scenario. This case is however, not considered in the rest of this paper.

Variable declaration The VM should provide additional storage space aside from the registers. This allows the temporary storage of values during the execution of an ACP. In order to facilitate effective use of the VM, two types of variables are proposed.

Firstly, the VM must support local variables. That is, variables that only have scope during the execution of a particular ACP. Secondly, variables that have scope even after the ACP they were declared in has terminated. Variables of this type are deemed to be *persistent variables*, and these variables are discussed in more detail in section 4.2.2.

The VM should also provide an instruction which can be used to retrieve the address of a particular variable. This functionality will be required when, for instance, `string` variables have to be manipulated (4.1).

Extensions to the VM Future expansion of the VM should always be kept in mind, as well as the fact that the particular requirements of the VM may differ from enterprise to enterprise. It is therefore desirable to provide a way of extending the functionality of the VM. Thus the VM must provide instructions which allow ACPs to indicate that a particular extension should be executed. This idea is explored further in section 7.

Instructions to parse an SQL statement Determining if access should be granted or denied may require more information than just the object, user, and access mode. The VM should therefore ideally also provide instructions that allow ACPs to parse SQL statements.

For example, consider a scenario where *Bill* is not allowed to change the current savings account interest rate of a bank's database by more than one percent at a time. If ACPs are to be considered useful they should ideally be able to be used in the creation of an access control policy that enforces this restriction. In order to accomplish this it is necessary that there are instructions which allow an ACP to parse SQL statements to, for example, determine the values associated with columns in INSERT and UPDATE statements.

Navigating the result of an embedded query In a similar fashion to triggers which can access data stored in relations in the database, it would also be beneficial if ACPs could access data stored in the database. This would allow ACPs to examine data, and based on it grant access or deny access.

Consider for example a database which stores information on the movement of employees. As soon as *Joe* enters the front door of the enterprise owning such a database, he swipes his access control card. The fact that he has entered the building is recorded in the database. When he tries to access a database (the same database that logs the movement of employees or a different one) as part of his daily duties, an ACP can execute an SQL query to determine if *Joe* is in fact on the enterprise's premises. It is obvious that these types of access control policies can provide tremendous benefit to the enterprise.

4.2 Amount of addressable memory

The amount of memory that has be available to ACPs is of course largely determined by the actual application of ACPs. Because ACPs are not nearly their general application software counterparts, it stands to reason that a large amount of memory should not not be a mandatory requirement.

The only important aspect to consider is that there is more than one memory area that should be supported by the VM. There must be an area for local variables, and a memory area for persistent variables—the Persistent Storage Area (PSA).

Bearing all of the above in mind, the fact that ACPs require minimal resources to accomplish their task and the fact that there are at least two different memory areas that can be used for the storage of data it is highly unlikely that anything larger than a sixty-four kilobyte segment for each of the aforementioned memory areas should be required.

The two different memory sections introduced will be discussed next.

4.2.1 Local Variables

Local variables that form part of an ACP is exactly the same as the normal definition of a local variable. That is, these variables have local scope with respect to the program.

These variables are declared (and can also be defined) within the *local* segment of an ACP. The amount of global variables that can be declared is dependent on the amount of addressable memory, and it should also be noted that using the Von Neumann architecture, this memory is also dependent on the size of a particular ACP.

The final type of variable is the persistent variable which forms part of the PSA, and will be discussed in the following section.

4.2.2 Persistent Variables

Simply having one ACP that is used to determine if access can be granted or not is of course not enough. Using a combination of ACPs provides even more power. Thus it becomes necessary to be able to save a previous state, which can be used in subsequent ACPs. Consider for example an access control policy which restricts the number of times that *Charlie* may access a certain relation per day. In order to realise this policy it is necessary to keep track of the number of times *Charlie* has accessed the relation. This information can be stored in a relation (by issuing an SQL statement from inside the ACP), however, this results in additional overhead. A faster way would be to store the information in a special memory area (designated the PSA).

Obviously it becomes necessary that ACPs declare certain variables to be persistent, distinguish them from local variables. It is also necessary (for security purposes) that persistent variables are declared with access modifiers, such as read-only or read-write.

Sharing variables among ACPs, means that these variables have to be protected. This can be handled in one of two ways.

Firstly, if it is assumed that only one person creates the ACPs then there is obviously no danger of the illegal use of a particular shared variable. On the other hand if more than one person is allowed to create a program, then it can be assumed that the creator of a specific program will find out from the creator of another program which variables are shared, and thus there is again no violation of confidentiality.

Secondly, shared variables can be considered entities within the RDBMS which have to be protected in the same fashion as base relations are protected. In this case ACPs can be created to protect the variables; access to these variables is accomplished in much the same way in which access is accomplished to a base relation or view. Persistent variables provide a way for ACPs to store information in a fast and efficient way between executions. These type of variables can be of any data type that is supported by the VM.

4.3 Registers

For the purpose of access control, the VM's register set does not have to be large. The only requirement on the VM is that it should provide registers that are large enough in order to store constant values that correspond to the data types supported by the VM (section 5).

Obviously this does not mean that the VM should support an internal register that is capable of storing string values; and as has been mentioned the VM should provide an operation which allows ACPs to get the memory address of a particular variable. As base types, we consider that the VM should only provide registers that are capable of storing integers and floating point values. There need not be a distinction between the registers other than the fact that registers used for floating point numbers are wider than those used for storing integers.

Registers must be wide enough to store large values. The reason for this is obvious: when working with data stored in a database then large values frequently appear as a result of some manipulation (such as an aggregate operation on data).

In addition, since it is also envisaged that the VM will provide programs with access to information such as the IP of the client issuing the access request, proper care has to taken to ensure that the VM supports registers that are large enough to manipulate IP addresses. The current implementation of the IP address range limits this registers size to thirty-two bits; this size will of course change as the IPv6 protocol is used more in the near future.

4.3.1 The Flag Register

ACPs which execute may at certain times require information regarding certain states of the VM. A prime example of this is determining whether the execution of a particular SQL query was successful, whether the last tuple of the result of a previously executed SQL query has been reached, or if a “division by zero” was attempted. Another use of the register is to indicate that a particular hook program (section 7) failed to execute properly, or that the requested hook program does not exist.

The VM indicates these states by maintaining an internal *flag register*. The register operates by making use of particular bits that form part of the register in order to indicate that a particular error condition exists or that a particular error has occurred.

4.3.2 The Instruction Pointer

It is also conceivable that the VM provides an instruction pointer register, which is used to keep track of the current instruction being executed. If the VM supports procedure calling, then a procedure and operand stack may be used. Thus the instruction pointer can be stored in the operand stack.

5 DATA TYPES

The VM need only provide support for atomic data types such as integers, floating point values and string literals[3]. Effectively this means that more complex types like structures and classes are not supported (this is obviously because the the ACPs are intended to be low level access control facilitation mechanism).

Arrays can be handled in much the same way as arrays are handled in a general assembly language. The first element can be accessed by acquiring the “array” variable’s address. Subsequent indices in the array are reached by adding an offset value to the variable’s address.

After having examined the data types that are supported by the VM, the following section elaborates on the use of variables that provide information of the RDBMS environment.

6 ENVIRONMENTAL VARIABLES

Environmental variables are variables which can provide more information regarding the environment of the RDBMS. This information is sometimes necessary to base access control policies on. A simple example is the existence of the SQL built-in operators `CURRENT_TIME` and `CURRENT_DATE`. The VM must provide this information as read only variables to ACPs. As these variables are intended to be read only, their values can never be changed by any ACPs.

As the information that is provided by the global variables may never be constant (for example the current time) instructions in which these variables appear as source operands are handled

[3]These types can of course, and will no doubt, be expanded to include the fundamental data types supported by the RDBMS that utilises the VM itself.

by the VM in a special manner. There are at least three categories of global variables which the VM must supply to the ACPs. These are:

1. The current date and time.
2. Information on the connection to the RDBMS by the current subject.
3. Information regarding the query issued by the current subject.

These variables are all discussed in the following sections.

6.1 Date and Time

Time is of critical importance for security purposes. To this extent the VM should provide the current date and time to ACPs.

For ease of use, keywords can be used to return the atomic parts of the time and date (such as the current hour, or current day) rather than requiring ACPs to manipulate date and time values in order to obtain these atomic values.

Internal storage of the date and time can be accomplished by using the UNIX time standard, which only requires thirty-two bits to store the current date and time. The largest unsigned value that can be represented using thirty-two bits is roughly four billion. Thus, this implementation should be adequate for the next one-hundred and thirty years.

The following section provides more information on the global variables that provide information regarding the connections made to the database server.

6.2 Connection information

In order to provide more expressiveness and power, the VM should ideally provide ACPs with information regarding the connection that is made to the server by the client. This can include information such as the IP address of the client's machine, and the type of connection being made (secure or not). This will allow ACPs to deny access if for instance the client's machine is on a different (presumably untrusted) subnet. Or perhaps, certain tuples in a relation can be returned provided the connection made to the database is secure.

Global variables can be extended to include much more information than that presented here. The core idea behind the introduction of the global variables is simply to show that the environment of the database server can be used by ACPs in order to enforce certain access control policies.

6.3 Information on queries

The VM should also provide the ACPs with information regarding the current query. For this purpose the VM should provide:

1. A string representation of the complete query.
2. A list of all the attributes that access is requested to in the query.
3. A list of all the objects that access is requested to in the query.

4. The subject that is executing the query.

Operations that allow the accessing of arrays can be used to manipulate these variables (section 4.1).

7 VM SPECIAL FUNCTIONALITY (HOOKS)

Hooks provide additional as well as extensible support for custom functionality of the VM. By writing a specific function either in a programming language such as C or C++, or a trigger written in SQL, the functionality of the VM can be extended.

These hooks are called by making use of an instruction which is recognised by the VM as a “hook calling” instruction. Hook programs can be made available to ACPs by loading binary object files that contain the definition of hook functions.

The extension proposed here is similar to the extensions that can be accomplished readily in the TCL programming language introduced by Ousterhout [15].

It is obvious that hook functions should be able to receive parameters. For this purpose a special instruction should be provided which indicates that certain variables are to be passed to the hook functions either by value or by reference. In order to avoid the unnecessary overhead involved in the passing of variables, all environmental variables should be made available to hook functions by default. Values can then either be returned by modifying parameters which were passed by reference, or by placing the result in a predetermined register.

8 CONCLUSION

In this paper significant design issues surrounding a VM that can be used for access control in an RDBMS were considered. In particular aspects such as the instruction size, the amount of addressable memory, and the number of registers were considered. Special instructions that are central to the requirements of an access control VM were also examined.

Reflection of these issues has indicated that implementation of such a mechanism can be accomplished, and that such a mechanism can be beneficial in the realisation of complex access control policies for several reasons.

Firstly, the VM assumes the RDBMS as a primitive environment, allowing it to utilise data from relations in the database as well as functional information for SQL statements. Thus affording more expressive power to ACPs. Secondly, where normal access control in a relational database is reliant on algorithms (for example Multi-Level Secure (MLS)) that are implemented as part of the RDBMS software, the VM approach proposed here allows easy modification of the particular type of access control. Thirdly, by using a PSA, ACPs can be made “session aware”, allowing access control policies to be implemented that span multiple access requests. Finally, using a well known technique, the functionality offered by the VM can be arbitrarily extended to ensure future use.

There are of course other relevant issues raised in this paper that can be categorised as future research. Exactly how security of each ACP is ensured has to be considered. It is also assumed that ACPs are not malicious. Research done on proving the “safety” of programs by Necula et al [16] and Metz [17] can be considered to solve this issue. Future work can also be aimed at creating a high level language that targets the VM proposed in this paper. This will allow more efficient use of the capabilities offered by the VM.

References

[1] Christopher J Date and Hugh Darwen. *A guide to the SQL standard*. Addison-Wesley, fourth edition, 1997.

[2] Patricia P Griffiths and Bradford W Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems (TODS)*, 1(3):242–255, 1976.

[3] Christopher J Date. *An introduction to Database systems*, volume 2. Addison–Wesley, 1983.

[4] Elisa Bertino. Data security. *Data and Knowledge Engineering*, 25(2):199–216, 1998.

[5] Christopher J Date. *An introduction to database systems*, volume 1. Addison-Wesley, seventh edition, 2002.

[6] Terry Rooker. The reference monitor: an idea whose time has come. In *Proceedings on the 1992-1993 workshop on New security paradigms*, pages 192–197. ACM Press, 1993.

[7] Silvana Castano, Mariagrazia Fugini, Gaincarlo Martella, and Pierangela Samarati. *Database Security*. ACM Press, 1994.

[8] Ravi S Sandhu. The typed access matrix model. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 122–136, 1992.

[9] Jeffrey C Mogul, Richard F Rashid, and Micheal J Accetta. The packet filter: An efficient mechanism for user-level network code. *ACM Operating Systems Review, SIGOPS*, 21(5):39–51, 1987.

[10] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, April 1999.

[12] Java^{TM} 2 platform, standard edition, v 1.4.2 api specification. http://java.sun.com/j2se/1.4.2/docs/api/java/lang/SecurityManager.html, 2003.

[13] Wynand van Staden. The use of a virtual machine as an access control mechanism in a relational database management system. Master's thesis, Rand Afrikaans University, 2003.

[14] Irv Englander. *The architecture of computer hardware and systems software*. John Wiley and Sons, third edition, 2002.

[15] John K Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing, March 1994.

[16] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation*

(OSDI ’96), October 28–31, 1996. Seattle, WA, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[17] Craig Metz. Safety checking of kernel extensions. In *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*, June 2000.