# Symmetric Behavior-Based Trust: A New Paradigm for Internet Computing

Vivek Haldar Michael Franz
University of California
Irvine, CA 92697
+1-949-824-7308
{vhaldar, franz}@uci.edu

## ABSTRACT

Current models of Internet Computing are highly asymmetric – a host protects itself from malicious mobile Java programs, but there is no way to get assurances about the behavior of a program running remotely. The asymmetry stems from a behavior-based security model: hosts ensure conformance to a given security policy by restricting the actions of programs. In contrast, security models that are based on cryptography (including code signing) are inherently symmetric by design but do not match the open architecture of the Internet and are unsuitable for reasoning about program behavior. We propose a new paradigm that combines the openness of the former with the symmetry of the latter and thereby enables completely new applications in a globally connected world.

## 1. INTRODUCTION

Research in Cyber-Security has been approached from two opposite sides. The first thread concerns itself with *entities* and their integrity and authentication. It involves certifying that various parties (persons, machines, programs) are who they claim, and then securing communication among them and protecting their information. Techniques developed for these purposes are encryption, digital signatures, cryptographic protocols, and recently, biometrics.

A second, complementary thread of research has focused on *behavior* and is concerned with ensuring that the entities in a system behave within the limits of a well-defined policy. This thread has resulted in static analysis techniques (which broadly includes methods as diverse as proof-carrying code [2] and meta-compilation [3]) and dynamic enforcement techniques such as inline reference monitors [4] and system-call interposition.

Security, however, is a system-wide concern, and does not cleave so neatly into these two domains.

It is instructive to note that many security vulnerabilities are caused when underlying design assumptions in a system stress one mode of security, while ignoring the other. Malicious code in email attachments, for example, completely gets around the authentication problem (by tricking gullible users) and exploits the lack of fine-grained resource usage policy enforcement in the operating system to run unchecked with broad privileges. Thus, simply entity-based protection is inadequate against malicious-code attacks. Traditional solutions such as firewalls cannot stop incoming executable code, nor prevent its execution on a compromised host.

The present inadequacies in behavior-based security are directly visible in vulnerabilities in common Internet infrastructure software. CERT issued twenty-eight advisories in 2003 [1]. The underlying cause for twenty-three of those advisories was incorrect memory management. Of these twenty-three, twenty-one were buffer overflow vulnerabilities, and the remaining two were buffer mismanagement vulnerabilities such as freeing a pointer twice. This clearly shows that the overwhelming majority of attacks exploit programming errors and fundamental flaws in the underlying memory model of the language being used, which is most cases is C. Authentication and integrity, secured by strong cryptographic methods, are not the weak link in security, and are hardly ever even targeted.

Intuitively, "authentication" of an entity should have a broader meaning than it does currently. It should encompass not just cryptographically verifying its origin, but also include verifying or proving that its behavior conforms to a required security policy. For example, when entities are Java bytecode programs, we should be able to send an object or program to a server and ensure that all the abstractions of the program are respected. The remote server should not, for instance, be able to read private variables, even though it is hosting the object.

We are currently investigating novel techniques that link entity-based and behavior-based security at a finer granularity than existing approaches, namely at the object level of a programming language. This is also the natural level at which to express policies. Combining fine-grained object-level authentication with expressive policies will allow us to move away from the current client-server network computing model, which assumes a trusted server and untrusted (even malicious) clients. Many useful applications do not follow this model:

- How can a distributed computation grid be hardened against rogue servers returning wrong partial results?
- When sending a negotiating e-procurement agent program to a supplier to negotiate with, how can the supplier be prevented from performing some analysis or replay attacks to extract the highest price from the agent?
- When buying services from an Application Service Provider, how can one ensure that one actually got the quality of service paid for?


*Conference'04*, Month 1–2, 2004, City, State, Country.

- When implementing a database as a service, storing data in encrypted form on servers of questionable integrity, how can stored database trigger functions be implemented?

Answering these questions will have a profound impact on the applicability of grid computing. It is exactly this issue of *fine-grained, two-way trust* that has so far hampered the adoption of distributed Internet architectures for security-critical applications. Solving the underlying security and trust issues will allow networking together large grids of computers of different levels of trustworthiness running diverse applications without compromising their security.

The need for addressing these questions is highlighted by the recent trend towards Trusted Computing. Trusted Computing is an attempt to embed a hardware-based secure sub-system into commodity platforms. Combined with software that uses it, it forms the root of trust for an array of security functionality.

While platforms incorporating Trusted Computing components will start appearing in the market in the next few years, there is a marked lack of systematic knowledge that would enable us to use such systems to their fullest potential. At present, purely cryptographic methods are used in Trusted Computing, and the stress is on identifying and authenticating entities. The question of how to handle behavior-based security is not addressed.

Before Trusted Computing can reach its full potential, questions such as the following need to be addressed:

- How do programs running on trusted platforms authenticate each other in a manner that ensures that each party satisfies some security criteria, while leaving room for different implementations?
- The current client-server network computing model assumes a trusted server, and untrusted or even malicious clients. Thus, even though a significant fraction of work is done at the clients, all the trust resides at the server. How can we design new network protocols, or adapt existing ones, to work in an environment that allows a more flexible partitioning of trust?
- Moving away from the model of having a fully trusted server, and a fully untrusted client, how do we design models and applications that use them, and that can broker trust in more flexible and dynamic ways than is possible today?

Answering these questions is a prerequisite to tackling the problem scenarios highlighted at the beginning of this paper. These questions expose some of the more practical, real-world aspects of the larger challenge – to meaningfully unify entity-based and behavior-based security.

The rest of this paper is organized as follows: in Section 2 we give a brief overview of Trusted Computing, explain some of its shortcomings, and explain our approach to tackling them – this forms the core of this paper; Section 3 discusses some of the consequences of our proposed solution; Section 4 presents the status of our work, and explores avenues for future work; Section 5 briefly surveys related work; and Section 6 summarizes and concludes.

# 2. TRUSTED COMPUTING

In this section we give a brief overview of Trusted Computing, followed by our efforts to mitigate some of its shortcomings using language-based techniques.

One way to get security assurances is to use *closed systems.* They enforce compliance with a certain security policy by being tightly controlled. They are usually manufactured by a single vendor to rigid specifications. Designers have complete control over the whole system, from hardware to software, and build it specifically to conform to a given security policy. When one closed system communicates with another, it knows within very tight bounds the expected behavior of the remote party. Common examples of closed systems are automated teller machines and proprietary game consoles

*Open systems*, on the other hand, have no central arbiter. Commodity personal computers and handhelds are examples of open systems. An open system can be easily changed to behave maliciously towards other systems communicating with it. Two communicating open systems cannot assume anything about each others' behavior, and must be conservative in their assumptions.

Trusted Computing [5] is an effort to bring some of the properties of closed, proprietary systems to open, commodity systems. Trusted computing introduces mechanisms and components in both hardware and software that check and enforce the integrity of a system, and allow it to authenticate itself to remote systems.

The root of trust is a tamper-resistant hardware device with approximately the same functionality as a cryptographic token. It has a random number generator, an RSA engine for encryption and signing, and some non-volatile storage. Its purpose is to check the system's integrity. A *secure booting* procedure makes sure that the operating system has not been tampered with. Using a chain of reasoning that starts from a trusted hardware module, we can arrive at a conclusion about the state of a system after boot-up. *Strong isolation* between the system and applications, as well as between applications themselves, prevents their integrity from being compromised. And a process called remote attestation is used to authenticate software.

*Remote attestation*, one of the core features of a trusted computing infrastructure, is the process by which software authenticates itself to remote parties. When asked to authenticate itself, an application asks the operating system for an endorsement. The operating system signs its *integrity credentials*, which is just a hash of the executable of the application. The entire certificate chain, starting from the trusted module all the way up to the application, is sent to the remote party. The remote party verifies each certificate of this chain, and also checks that the corresponding hashes are of software it approves. The attestation process must result in the client and server sharing a secret, or else the session can be easily hijacked (e.g. by performing attestation using one program, and further communicating using another).

## 2.1 The problems of remote attestation

This standard method of performing remote attestation suffers from several critical drawbacks. Briefly, they are:

- It says nothing about program behavior
- It is static, inflexible and inexpressive
- Upgrades and patches to programs are hard to deal with
- It is fundamentally incompatible with a widely varying, heterogeneous computing environment
- Revocation is a problem

We discuss each of these in more detail below.

The most critical shortcoming of remote attestation is that it is not based on program behavior. Even though what is fundamentally sought is some assurance of program *behavior* with respect to some security policy, remote attestation certifies something completely different. It simply certifies what exact executable is running. Any assurances about the behavior of the program are taken on trust. It is possible for an attested program to have bugs, or otherwise behave maliciously.

Remote attestation defined in this way is completely static and inflexible. It can convey no dynamic information about the program – such as its runtime state, or the properties of the input it is acting upon. It is a one-time operation done at the beginning of a network protocol.

Another problem is that upgrades and patches are hard to deal with. Linear upgrades from one version to the next can be accommodated by simply updating the list of "approved" software that a verifier uses. In closed and tightly controlled systems such as ATMs, this is tractable. The situation with widely available commodity software is completely different. As is increasingly common today, upgrades and patches are released very frequently. Also, software is patched more often than it is upgraded. There are usually multiple patches for multiple bugs and insecurities for a given program. Any subset of these patches may be applied in any order. This results in an exponential blowup in the space of possible binaries for a program. In such a scenario, remote attestation faces problems at both ends of the network. Servers have to manage the growing intractability of maintaining a very large list of "approved software", which is likely to always be behind the current state. Clients, on the other hand, may have to hold off on applying patches or on upgrading, simply to be able to work with remote attestation framework.

Today's computing ecosystem is extremely varied and accommodates a spectrum of heterogeneous systems with widely varying capabilities. These systems range from high-end supercomputers, to consumer devices such personal computers, handhelds, cell phones and watches, and even ubiquitously embedded microprocessors. In such a scenario, a high premium is placed on portability and interoperability. This is one reason why cross-platform portable solutions such as Java are so popular. Remote attestation, however, with its stress on certifying particular platform-specific binaries, is fundamentally incompatible with this reality. Just as with managing upgraded and patched versions of software, certifying programs that run on a variety of platforms and that must inter-operate with each other, quickly becomes intractable.

Remote attestation inherits a problem from public-key cryptography – revocation. Once a certification authority issues a certificate, it is very hard to revoke. One method is to have publicly available certificate revocation lists (CRLs) which are looked up at regular intervals. Thus, there may be some time lapse between a certificate being revoked, and access being denied to it. Checking with some revocation infrastructure (such as a CRL) at every attestation would be very inefficient.

## 2.2 Semantic Remote Attestation

The shortcomings of traditional ways of remote attestation can be traced back to one root cause – *what is desired is attestation of the behavior of software running on a remote machine, but what actually gets attested is the fact that a particular binary is being run.*

We are working on a technique called *semantic remote attestation* [6] that attempts to alleviate these shortcomings of standard remote attestation. The core idea behind our technique is to use a language-based virtual machine (a trusted virtual machine, or TrustedVM) that executes a form of platform-independent code. Software up to and including this virtual machine is trusted. However, the virtual machine can certify to remote parties various properties of code running under it by explicitly deriving or enforcing them. This can be done in many ways, such as observing the execution of programs running in a VM, or analyzing the code before execution. This is particularly easy to do with high-level platform-independent code that has a lot of information about the structure and properties of code.

Some examples of properties that a trusted virtual machine can attest are:

- **Properties of classes:** the remote party may require class A to subclass a well-known class B, or some interface C. This may be because extending B or C constraints the behavior of A in some way. For example, C may be a restricted interface for input-output operations that disallows arbitrary network connections.
- **Attesting dynamic properties:** the program being attested runs under complete control of a TrustedVM. Thus, a TrustedVM can attest to dynamic properties. This includes the runtime state of the program and properties of the input of the program.
- **Attesting arbitrary properties:** A TrustedVM has the ability to run arbitrary analysis code (within the limits set by the security policy of the local host) on the program being attested on behalf of the remote party. Thus the remote party can test for a wide variety of properties by sending across code that does the appropriate analysis.
- **Attesting system properties:** a remote party can send across code that tests certain relevant system and virtual machine properties, and the TrustedVM can attest its results. For example, before running a distributed computing program (such as SETI@Home, or Folding@Home), the server may want to test the floating point behavior of the system and virtual machine by having the TrustedVM run a test suite of floating point programs
- **Information flow properties:** when handling sensitive data on behalf of a remote party, the proper containment of information is important. Using a TrustedVM that supports fine-grained mandatory access control, a remote party can specify constraints on the propagation of its data.

Attestation thus defined is a much more fine-grained and semantically richer operation than signing the hash of an executable image. What is now attested is not the presence of a particular binary executable, but relevant properties of a program.

This has the effect of explicitly separating two concerns that were earlier merged into one – identity and behavior. Claims about code behavior are now made by the trusted virtual machine explicitly checking or deriving them. Cryptography now plays the part of binding this claim about code behavior to an entity which is qualified to make such claims – a trusted virtual machine.

A direct consequence of this is that now a variety of different implementations of the same functionality are able to function within our remote attestation framework, as long as they satisfy the properties required of them.

This technique leverages the trend of more and more application code being targeted at high-level language runtimes and virtual machines that execute some form of safe, platform-independent bytecode. The most prevalent examples of this are the Java virtual machine [8], and the more recent .NET common language runtime [9]. Such code platforms offer a number of advantages over native code. The virtual machine performs a number of static and dynamic checks to ensure a basic level of code safety – type-safety, and control flow safety. Type safety ensures that operators and functions are applied only to operands and arguments of the correct types. A special case of type safety is memory safety, which prevents reading and writing to illegal memory locations – for example, beyond the bounds of an array – and thereby also provides separation between different processes without the need for hardware-based memory management. Control flow safety prevents arbitrary jumps in code (say, into the middle of an instruction, or to an unauthorized routine). These basic properties of safe code are enforced by a combination of static (e.g. bytecode verification) and dynamic (e.g. array bounds checks) techniques. Thus, safe code does away with a major source of bugs and vulnerabilities in current systems that stem from unsafe memory operations in C – such as buffer overruns and format string attacks.

# 3. DISCUSSION

The fundamental motivation for our work is that Trusted Computing is a solution to the trust problem – it is not a solution to the larger problem of end-to-end security and program-behavior.

The classes of attacks that Trusted Computing hardware and software prevents are those that rely on spoofing the authenticity and integrity of system software. For example, the Trusted Platform Module's boot-time integrity checks will disallow booting into a corrupted copy of the operating system. However, this still does not rule out the large majority of bugs caused by insecure memory handling in C.

As a thought experiment, consider the following: what if today's system software was simply moved over to run on a Trusted PC? We would be able to get guarantees about the integrity of the system, and its authenticity when it communicated with other systems, but no assurances about its (lack of) vulnerabilities, or its behavior towards other systems.

Using semantic remote attestation deeply changes the way trust is handled in networked applications. The current model is one of a completely trusted central server, with numerous untrusted clients. Also, these trust relationships are usually static and cannot be changed at runtime, across different executions of a program, or over the lifetime of a system. Semantic remote attestation can change this lop-sided balance of trust. Implementing applications within our framework achieves two benefits:

- Trust relationships between peers, or between clients and servers, are made *explicit*, and then *checked* or *enforced* by the TrustedVM. Typically, they are implicit and taken on trust.
- Making the trust relationships explicit results in having some knowledge of *degree of trustworthiness* of clients and peers (for example, knowing which properties were satisfied, and which were not). This allows the applications to make informed decisions about the "goodness" of a result, and dynamically adjust its trust relationships.

The fact that trusted computing, and its core technique, standard remote attestation, can lock consumers into a particular program or platform has been a very widely expressed fear [10]. A key advantage of our approach is that reasoning about the behavior of a program is not tied to a particular binary. Semantic remote attestation checks for program properties, and works with different implementations of the same program as long as they satisfy the security criteria required of them.

Semantic remote attestation also completely turns on its head the established goal of language-based security – to protect the local host from downloaded malicious code – and uses it to certify properties of code running locally to remote parties.

# 4. STATUS AND FUTURE WORK

To gain experience with semantic remote attestation, we have implemented a prototype TrustedVM on top of a Java virtual machine. Two techniques that a trusted virtual machine uses to certify properties of code running on it are: installation of a runtime monitor; and running various test suites. We have implemented two example applications on our prototype that take advantage of these techniques. The first application is a simplified peer-to-peer networking protocol, and the second is a distributed computing client-server application. The P2P client uses a runtime monitor to enforce some high-level constraints, such as checking that its replies to P2P search queries are indeed true. The distributed computing client uses test suites to determine a client's capabilities. These are then used to compute error margins of results that client nodes return. A full discussion of their implementation is beyond the scope of this paper – see [6] for details.

Another technique that can be used by a TrustedVM is using mandatory access control on objects in a trusted virtual machine [7]. The goal is to certify to remote parties communicating with a TrustedVM that the information they provide is being handled according to a policy also specified by them. Consider a network exchange between some remote party and a TrustedVM that involves the exchange of some sensitive data. In such a scenario, the remote party would like to have some means of constraining how the information is handled by the TrustedVM. Taking advantage of MAC support in a VM, the remote party could specify an information flow policy for the TrustedVM to enforce.

There are also many avenues for future work that we would like to explore.

A TrustedVM is capable of attesting the results of some static analysis. However, there are not many static analyses of code for properties of interest to a remote party. Most static analyses and runtime enforcement policies so far have been geared towards protecting a host from malicious mobile code. Thus, the emphasis has been on type-safety, information-flow, and resource control and other safety issues. The emphasis is different for remote attestation. Servers want to know if the application is obeying some high-level semantic rules. One candidate for an analysis that may be of interest to servers is information flow [11]. Such an analysis would convince the server that a client is not leaking the results of some confidential computation, or sensitive data. As mentioned above, we are currently working in this direction by adding object-level mandatory access controls in a Java virtual machine.

The ability to communicate to a server what particular property of a program could not be certified can be very useful. Using TrustedVMs, this information can be communicated, and the server can get detailed information about what desired properties are not present in a client program. It can then make an informed decision about either decreasing its trust in this particular instantiation of a protocol, or stopping altogether. Thus, the server gains some dynamic feedback about the trustworthiness of its clients. We believe this property can be fruitfully exploited to "port" a variety of untrusted network protocols (TCP, HTTP etc.) to a trusted computing framework in a gradual manner, and yet have various implementations of them inter-operate. This is in stark contrast to the all-or-nothing model that standard "signed-hash" remote attestation provides – attestation either passes or fails – there is no gradation. This would also provide a gentler upgrade path for applications as trusted hardware becomes increasingly available in the market.

Trusted computing systems use *trusted paths* between input devices and applications or device drivers to prevent spoofing as well as eavesdropping. For example, a fully encrypted and authenticated channel is used between a password-prompt dialog and the application asking for it. We would like to implement corresponding functionality in a virtual machine. Currently, the dynamic nature of the Java virtual machine makes is easy to do things like modify the class hierarchy, or use reflection to interpose wrappers around method calls – both at runtime. For example, dynamic method wrappers (also known as dynamic proxies) are frequently used to add a layer of logging around method calls. Such techniques could also be used to eavesdrop on the transfer of confidential data between objects. Implementing a trusted path mechanism for object communication would be a step towards solving this problem.

In our current prototype, security policies are simply programs. For example, runtime monitors or test suites are sent to a TrustedVM as code that it installs and runs. We would like to explore the design of succinct, yet expressive, policy specification languages that can be used for this purpose.

# 5. RELATED WORK

There have been a number of approaches to building trusted systems. While it is generally agreed that ultimately some trust must reside in a tamper-proof hardware device, different approaches vary the degree of trust placed in that hardware.

At one extreme, systems such as XOM and Cerium [12] put *all* trust in hardware. The trusted computing base consists entirely of hardware and no software at all is trusted. Everything outside the main CPU is fully encrypted. The disadvantage of these approaches is that they require a complete overhaul of the architecture of current systems.

On the other hand, the TCPA platform module is relatively lightweight. It has roughly the same architecture and complexity as a cryptographic token or smartcard. It has cryptographic engine to perform encryption and digital signing, a random number generator, and a small amount of non-volatile storage. The most compelling advantage of the TCPA architecture is that it does not require overhauling changes to the architecture of widely available commodity computers. The platform module is essentially just another component on the motherboard.

The TCPA specification [5], in turn, is based on earlier work. The concept of secure booting was pioneered by Arbaugh et al [13]. Their Aegis system checked the integrity of system software in a sequence of incremental steps by using signed hashes. The Digital Distributed System Security Architecture [14] had many of the features of today's TCPA specification, including secure bootstrapping, and remote attestation of system software using signed hashes.

Virtual machines have also been used for Trusted Computing, albeit at lower levels of abstraction. Garfinkel et. al. [15] have proposed the TerraVM [16] virtual machine monitor architecture to interface with underlying trusted hardware. Their architecture provides two VMM abstractions to software – an open box VMM, and a closed box VMM. The open box VMM simply provides a legacy, untrusted interface. This allows old operating systems and software to run unmodified on it. The closed box VMM, however, provides an interface to underlying trusted hardware that new software can use. A number of such VMMs can execute on bare hardware. They are strongly isolated from each other, and have their own encrypted storage.

The goal of TerraVM is similar to Microsoft's proposed Palladium architecture. Palladium is said to have a high-assurance trusted microkernel running on hardware (called the *nexus*) that provides strong isolation between legacy untrusted applications and newer trusted applications, as well as among trusted applications. These two distinct execution environments are called the *left-hand side* and *right-hand side*, respectively.

Our work is largely orthogonal to these efforts. While these focus on providing strong isolation, and abstractions and techniques for using strongly isolated execution environment. They do not tackle the problem of remote attestation, which has been our primary focus. They also work at a lower level of abstraction. Our semantic remote attestation framework could run atop all these architectures.

# 6. CONCLUSION

Current Trusted Computing initiatives do not present a new paradigm but merely a rehash of the age-old code-signing idea. Remote attestation, one of Trusted Computing's core techniques is static, inflexible, unable to reason about program behavior, and fundamentally incompatible with today's heterogeneous computing environments.

Our alternative mechanism, *semantic remote attestation*, combines both cryptography as well as language-based techniques to constrain the behavior of programs, and attest this to remote parties. We use language-based security techniques to certify properties of code running locally to remote parties. This new paradigm allows flexible, dynamic and symmetric trust relations, and enables a range of new applications.

# 7. REFERENCES


[1] Computer Emergency Response Team (CERT); CERT/CC Annual Report, 2003, http://www.cert.org

[2] George C. Necula; A Scalable Architecture for Proof-Carrying Code; 5th International Symposium on Functional and Logic Programming, March 2001.

[3] Ken Ashcroft and Dawson R. Engler; Using Programmer-Written Compiler Extensions to Catch Security Holes; IEEE Symposium on Security and Privacy, 2002.

[4] Úlfar Erlingsson, Fred Schneider; IRM Enforcement of Java Stack Inspection; IEEE Symposium on Security and Privacy, 2000.

[5] Trusted Computing Group (TCG); TCG PC Specific Implementation Specification; August 2003.

[6] Vivek Haldar, Deepak Chandra, and Michael Franz; Semantic Remote attestation: A Virtual Machine Directed Approach to Trusted Computing; USENIX Virtual Machine Research and Technology Symposium, May 2004.

[7] Vivek Haldar and Michael Franz; Mandatory Access Control at the Object Level in the Java Virtual Machine; Technical Report 04-06, Information and Computer Science, University of California, Irvine.

[8] Tim Lindholm and Frank Yellin; The Java Virtual Machine Specification; Addison-Wesley, April 1999.

[9] David S. Platt; Introducing Microsoft .NET; Microsoft Press, May 2003.

[10] Ross Anderson; Cryptography and Competition Policy - Issues with Trusted Computing; 2nd Annual Workshop on Economics and Information Security, May 2003.

[11] Andrei Sabelfeld, Andrew C. Myers; Language-Based Information-Flow Security; IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security, 21(1), January 2003

[12] B. Chen and R. Morris; Certifying program execution with secure processors; USENIX HotOS Workshop, May 2003.

[13] W. Arbaugh, D. Farber, and J. Smith; A secure and reliable bootstrap architecture; IEEE Symposium on Security and Privacy, 1997.

[14] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson; The digital distributed system security architecture; 12th NIST-NCSC National Computer Security Conference, pages 305-319, 1989.

[15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh; Terra: A virtual machine-based platform for trusted computing; 19th Symposium on Operating System Principles(SOSP 2003), October 2003.

[16] T. Garfinkel, M. Rosenblum, and D. Boneh; Flexible os support and applications for trusted computing; 9th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2003.