Volume 07 Issue 01 Published, February 19, 2003 ISSN 1535-766X

Intel® Technology Journal

Managed Runtime Technologies

Runtime Environment Security Models

A compiled version of all papers from this issue of the Intel Technology Journal can be found at:
http://developer.intel.com/technology/itj/index.htm

Runtime Environment Security Models

Selim Aissi, Intel R&D, Intel Corporation

Index words: security models, runtime security, access control, sandbox, CLR security, ASP.NET security, JRE security, Java security, runtime access control models

ABSTRACT

The tremendous new potential offered by distributed computing, inside and outside the home and business, also carries with it the necessity to exercise certain security safeguards. As distributed, mobile, and executable content moves among devices, the opportunity for security breaches increases dramatically. Also, as device-to-device e-Commerce services become more automated [11], new types of security threats are emerging. With these drastic changes in computing models comes a greater need for robust application security.

For example, "executable content" is the idea of sending code to a remote compute engine to be executed. In addition to flexibility and expressiveness, executable content brings new potential problems. A program received from a remote source must be regarded as non-trusted to some degree, and its access to certain resources must be restricted. However, this new execution model is not bound by the limitations of the operating system because the runtime environment enforces the security policies based on the code's origin. Both the Java* Runtime Environment (JRE) and .NET* Framework Common Language Runtime (CLR) security models have the following common security features: language type-safety, bytecode verification, runtime type checking, name space separation via class loading, and fine-grained access control.

This paper compares the JRE and the CLR evolutionary security mechanisms. The paper also compares the two models to the Clark-Wilson security model, a formal, application-level model used to ensure the integrity of commercial data. The Clark-Wilson model is a formal presentation of the security policy enforced by a system, and it is useful for testing a policy for completeness and consistency. It also helps describe what specific mechanisms are necessary to implement a security policy.

Besides exploring the nature and scope of the sandbox-based JRE and CLR security models and comparing them to the Clark-Wilson integrity model, this paper also provides some insight into the future of runtime security.

INTRODUCTION

The idea of using a sandbox to secure the threads running inside of that box is very similar to the idea of building a wall around a town to protect its inhabitants.

The concept of building a thick wall for protection is as old as history itself. Greek legend provides an interesting case of a thick wall that caused more destruction than protection: the Trojan Wall. Let's explore that security legend a bit further. During the Trojan War, the Greeks asked Epeius, an excellent craftsman, to build a wooden horse, which he did with the aid of the goddess Athene. Inside the horse were placed a handpicked group of warriors. Then the Greek fleet sailed away, leaving behind a warrior named Sinon, who pretended he had been left behind by accident. He also pretended that the huge wooden horse was an offering to Athene and that, if taken into the city, would make the city invincible. Despite warnings from some quarters, the Trojans pulled down part of their battlements and hauled the wooden horse inside the city. Lulled into a false sense of security, little watch was kept. The Greek fleet returned furtively and Sinon released the warriors from inside the wooden horse. Troy fell because the Trojans' confidence in an impenetrable wall led them to overlook the security risk in their midst [1]. When programmers (or users) fail to check inside the Horse (a metaphor for malicious code) before they roll it within the computing device, like the Greek legend, the result is unpleasant.

A Trojan Horse is an easily written security hack that has been used for years to breach traditional computer and network security barriers. The first Trojan Horses were disguised as demos, freeware, and shareware. The

* Other brands and names are the property of their respective owners.

unsuspecting victim would run the software from inside traditional security walls where the program could effectively attack. Sometimes the program just displayed a witty joke, performed some harmless mischief, installed viruses or broke into password files. Sometimes the Trojan Horse used security holes to break deeper into the computer. In all of those cases, the Trojan Horse causes some damage, including loss of productivity and confidence in the security of our systems.

Contrary to early claims, Trojan Horses and viruses are no strangers to runtime environments either [6]. In August of 1998, a proof-of-concept virus called Strange Brew appeared. While it did not carry a damaging payload, it did prove the concept that cross-platform Java* viruses and Trojan Horses could be written. Strange Brew, however, affects only Java applications, not Java applets that typically run inside a Web browser.

In January of 1999, the second known Java virus, called Java.BeanHive, was discovered. This virus was designed to infect both Java applets as well as Java applications.

The Java.BeanHive virus was, however, the first to exploit JRE's access control mechanisms by asking the user to grant the virus permission for full file access. Because the virus was a seemingly innocuous Java applet, some users inadvertently granted it full permission, not knowing it was malicious code.

In Java and .NET, the runtime environments provide security models that deal with access control to system resources. The following sections describe the capabilities offered by those mechanisms.

The .NET Framework also has had its share of security holes. In June 2002, session highjacking, information-leakage, and buffer overflow vulnerabilities were identified [12].

MOBILE CODE SECURITY: JAVA* AND THE .NET* ENVIRONMENTS

"Mobile code" denotes program code that traverses a network and executes at a remote site. The process of traversing can either be active as in the case of mobile agents which move around in a network at their own volition, or it can be passive, as in the case of user-downloaded code such as applets.

Both Java* and .NET* environments can be used as platforms for both types of code mobility, and in conjunction with the Internet, they open new possibilities for software development, software deployment, and computing architectures. The downside is that they also open new security threats. Downloaded code can include a virus or be a Trojan Horse and thus pervert the concept of code mobility over the Internet in a possibly dangerous way. Any mobile code platform, including both Java and .NET, suffers from four basic categories of potential security threats [5]:

- *Leakage.* This occurs when there are unauthorized attempts to obtain information belonging to or intended for someone else.
- *Tampering.* Tampering is unauthorized changing or deleting of information.
- *Resource stealing.* This occurs when there is unauthorized use of resources or facilities such as memory or disk space.
- *Antagonism.* These are interactions that don't result in a gain for the intruder but are, nonetheless, annoying for the attacked party.

To deal with these threats, Java and. NET environments provide special runtimes that try to protect users from erroneous or malicious mobile code and try to ensure the security and privacy of the user's system.

They both provide fairly good levels of protection against leakage and tampering but resource stealing and antagonism cannot be fully prevented since it is still hard to automatically distinguish between legitimate and malicious actions.

THE EVOLUTION OF THE JAVA* RUNTIME ENVIRONMENT'S SECURITY MODEL

In runtime environments, the security model is based on policy construction and enforcement. A security policy consists of the rules that must be obeyed by a program, the mechanisms to enforce these rules and to detect when they are violated, and the actions that are taken when a security violation is detected.

In Java*, a security policy is implemented by writing a subclass of the *SecurityManager* class and installing it as the system's security manager.

While the bottom three layers of Java's security model are fixed and defined by the Java language specification [2], the Java Virtual Machine (JVM) specification [3], and the Java API specification [4], the runtime environment is implementation-dependent. Although it is the only configurable part of the security model, this is,

* Other brands and names are the property of their respective owners.

nevertheless, sufficient for a wide range of different security policies to be implemented.

The Java Sandbox Model

Java security has undergone considerable evolution. In the JDK 1.0 security model, any code run locally had full access to system resources while dynamically loaded code had access to system resources controlled by a security manager. The default security manager sandbox provided minimal access to resources such as disk drives. In order to support a different security model, a new security manager would have to be implemented. The concept of trusted, dynamically loaded code was introduced in JDK 1.1. Any dynamically loaded code that was digitally signed by a trusted code provider could execute with the same permission as local code. JDK 1.2 introduced an extensible access control model that applies to both local code and dynamically loaded code. Fine-grained access to system resources can be specified in a policy file on the basis of the source of the code, the code provider, and the user of the code. Unlike earlier versions of the JDK, this policy file allows the security model to be adjusted without writing a new security manager. The security manager has standard access control checkpoints embedded in its code whose behavior is determined by the selection of permissions enabled in the policy file. New permissions can be defined, but explicit checks must be added to the security manager or application code if the permissions apply to application resources rather than system resources.

JVM Security

Four practical techniques for securing mobile code exist: the sandbox model, code signing, firewalls, and proof-carrying code. In order to secure mobile code, Java uses a hybrid approach, which combines sandboxes and code signatures. The Java core classes act as a security shield and enforce the sandbox model by granting or forbidding access to resources, based on a security policy. The rules specified in the security policy define the actions a piece of code is allowed to perform depending on the origin of the code and an optional signature. Not all of Java's powerful security mechanisms are in place by default when launching the JVM. While some basic checks are performed automatically, the more sophisticated concepts, including the sandbox model, have to be put into action explicitly.

Before a class is loaded, the following steps occur. First, the Verifier performs a set of security checks to guarantee properties such as the correct class file format, the correct parameter types, and binary compatibility. Doing these checks before loading enhances both security and runtime performance. They ensure the integrity of the Java runtime environment since no malformed class can be loaded that could cause a general system fault. Having passed the Verifier, the class loader loads the bytecode representation of the class and checks optional signatures. Furthermore, the source (i.e., origin) of the class's code is constructed, which consists of the location from which the class was obtained and a set of certificates representing the signature.

The source of the class code is the key input for the security policy construction for a given class. In Java 2, the security policy is defined in terms of protection domains, which define what a piece of code with a given source is allowed to do. Hence, a protection domain contains a code source with a set of associated permissions. Given the code source of a class, the security policy is searched to determine the permissions of the class.

Finally, the class is “defined,” meaning it is made publicly available and added to the class loader’s cache of classes. This is important to ensure class uniqueness. Java considers two classes equal if, and only if, they have the same name and were loaded by the same class loader.

After these initial steps, the class can be used in the Java runtime environment. However, every time the class tries to access a system resource, its permissions are checked by the security manager. If the call to the security manager returns silently, the requesting caller has sufficient permissions to access the resource, and the execution continues. If not, a security exception is raised and has to be handled by the caller or otherwise the JVM terminates.

A key question is how the security manager decides whether access to a resource is granted. Since Java 2, the security manager is mainly included for compatibility reasons and delegates nearly all of its tasks to the access controller. The access controller uses a stack inspection algorithm and the security policy to decide how to proceed.

The stack inspection algorithm is based on the call stack of the current method. Since every class is assigned an appropriate set of permissions when it is loaded, the stack inspection algorithm can use this information to make its decision.

THE .NET* FRAMEWORK COMMON LANGUAGE RUNTIME SECURITY MODEL

The Microsoft .NET* Framework offers code access security and role-based security to help address security concerns about mobile code, and to help determine what users are authorized to do. Both code access security and role-based security are implemented using a common infrastructure supplied by the Common Language Runtime (CLR).

Because they use the same model and infrastructure, code access security and role-based security share several underlying concepts described in the following sections.

The CLR Code Access Security Model

Any application that targets the CLR must interact with the runtime's security system. When an application executes, it is automatically evaluated and given a set of permissions by the runtime. Depending on the permissions that the application receives, it can either run properly or it will generate a security exception. The local security settings on a particular computer ultimately decide which permissions the code receives. Because these settings can change from computer to computer, one can never be sure that code will receive sufficient permissions to run. This is in contrast to the world of unmanaged development, in which one may not have to worry about the code's permission to run. CLR code access security is based on the following four concepts: writing type-safe code, using imperative and declarative syntax, requesting permissions for the code, and using secure class libraries.

In order to write type-safe code and to enable code to benefit from code access security, a compiler that generates verifiably type-safe code must be used.

Interaction with the runtime security system is performed using imperative and declarative security calls. Declarative calls are performed using attributes; imperative calls are performed using new instances of classes within your code. Some calls can only be performed imperatively, while others can be performed only declaratively. Some calls can be performed in either manner.

Requests for permissions in the code are applied to the assembly scope, where the code informs the runtime about permissions that it either needs to run, or specifically does not want. Security requests are evaluated by the runtime when the code is loaded into memory. The purpose of requests is only to inform the runtime about the permissions it requires in order to run. Requests do not influence the runtime to give the code more permissions than it "deserves."

The secure class libraries use code access security to specify the permissions they require in order to be accessed. The developer must be aware of the permissions required to access any library that the code uses and make appropriate requests in the code [7].

The CLR Role-Based Access Model

Roles are often used in an application to enforce some policies. Role-based security can be used when an application requires multiple approvals to complete an action.

The .NET Framework's role-based security supports authorization by making information about the principal, which is constructed from an associated identity, available to the current thread.

A principal represents the identity and role of a user and acts on the user's behalf. Role-based security in the .NET Framework supports three kinds of principals: Generic Principals which represent users and roles that exist independent of Windows NT* and Windows 2000* users and roles, Windows Principals which represent Windows* users and their roles (or groups), and Custom Principals which can be defined by an application in any way that is needed for that particular application.

The identity, as well as the principal it helps to define, can be either based on a Windows account or be a custom identity unrelated to a Windows account. The .NET Framework applications can make authorization decisions based on the principal's identity or role membership, or both. A role is a named set of principals that have the same privileges with respect to security. A principal can be a member of one or more roles. Hence, applications can use role membership to determine whether a principal is authorized to perform a requested action.

To provide ease of use and consistency with code access security, .NET Framework role-based security provides *PrincipalPermission* objects that enable the common language runtime to perform authorization in a way that is similar to code access security checks. The *PrincipalPermission* class represents the identity or role that the principal must match and is compatible with both declarative and imperative security checks. You can also access a principal's identity information directly and

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

perform role and identity checks in your code when needed [7].

Comparison Between the JRE and CLR Security Models

Figure 1 compares the Java* and .NET Framework architectures. The JRE and CLR are viewed as middle layers between intermediate languages and the underlying operating systems.

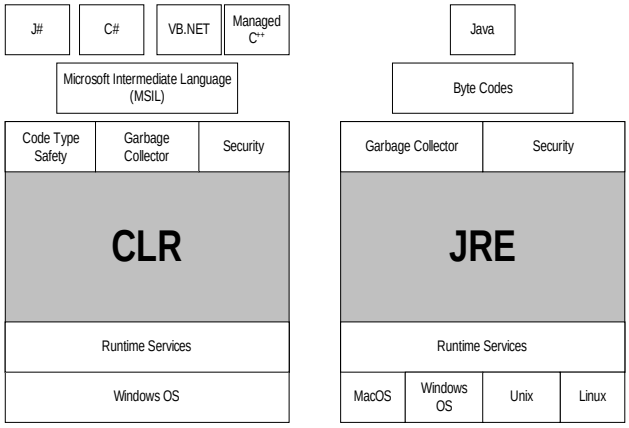

Figure 1: CLR versus JRE

In order to compare the two security approaches, the Clark-Wilson Security Model is used.

The Clark-Wilson Security Model

Integrity models [8] are used to describe what needs to be done to enforce information integrity policies. There are three goals of integrity: to prevent unauthorized modifications, to maintain internal and external consistency, and to prevent authorized but improper modifications.

To accomplish these goals, a collection of security services that embodies the properties needed for integrity as well as a framework for composing them is needed. The needed security properties for integrity include access control, auditing, and accountability.

The Clark-Wilson [9] model is an integrity, application-level model that attempts to ensure the integrity properties of commercial data, and it provides a framework for evaluating security in commercial application systems. It was published in 1987 and updated in 1989 by David D. Clark and David R. Wilson [13].

The Clark-Wilson model is based on analyses of security models actually applied within businesses. These security models aim at ensuring the integrity of resources rather than simply controlling access to them. They depend on controlling state transformations, and upon maintaining separation of duties between users of the system.

Clark and Wilson partitioned all data in a system into two types of data items for which integrity must be ensured: Constrained Data Items (CDIs) and Unconstrained Data Items (UDIs). The CDIs are objects that the integrity model is applied to, and the UDIs are objects that are not covered by the integrity policy (e.g., information typed by the user on the keyboard). Two procedures are then applied to these data items for protection. The first procedure, namely the Integrity Verification Procedure (IVP), verifies that the data items are in a valid state (i.e., they are what the users or owners believe them to be because they have not been changed). The second procedure is the Transformation Procedure (TP), which changes the data items from one valid state to another. If only a transformation procedure is able to change data items, the integrity of the data is maintained. Integrity enforcement systems usually require that all transformation procedures be logged, to provide an audit trail of data item changes.

In runtime environments, CDIs and UDIs can be mapped to fields of components (e.g., assemblies). TPs can be mapped to Java methods or .NET assemblies. An assembly is a collection of types and resources that is built to work together and form a logical unit of functionality.

A principal in this context is an authenticated Java or .NET principal where the authentication has been achieved using either the Java Cryptographic Extension (JCE) or Microsoft's Cryptographic API (CAPI*).

DISCUSSION

Resource Integrity

The basic concepts of access control in the JRE and CLR security models do not meet Clark-Wilson's requirement of resource integrity. Both environments require each controlled operation to be re-coded to include a permission check. This is not appropriate for a component that is delivered in binary form. Both the CLR and JRE also require determination of which operations update the state of an object so that only those operations that maintain the integrity of the system are allowed. This approach is error prone. A better approach would be to intercept all state accesses and allow only those made from operations that maintain integrity while blocking all others.

Execution-Time Checking

The .NET* Framework holds an advantage in the area of execution-time checking. .NET's application domains are

* Other brands and names are the property of their respective owners.

less permeable than Java's, i.e., NET code verification is stronger by default for local applications. Because a JVM verifies only remotely loaded code by default, one can run a Java* program locally without any security manager at all [10].

Data Protection

Neither environment offers a significant advantage in source code and data protection. Each platform has its strengths and weaknesses in this area. .NET's cryptography relies on the developer properly configuring the CAPI because it's so closely tied to Windows*. However, the variety of plug-ins and components available for Java makes it a more flexible environment.

The Java Cryptographic Extension (JCE) is a mechanism that allows suppliers of cryptography to integrate their libraries in a standard way with Java applications. The API is fairly flexible, allowing detailed control of the cryptographic process. However, that flexibility can lead to excessive complexity and can make the API difficult to use.

Communication Security

Developers using the .NET Framework may need to use Microsoft's Internet Information Server (IIS*) for communication protection. This strong dependency on a Web Server, such as IIS, to provide runtime security services, could restrict CLR's communication security.

Code-Access Security

The code-based security mechanisms for Java and .NET are very similar. The Windows connection gives the new platform a richer set of permissions and evidences than Java does. Java is more stripped down due to its platform independence; however, Java's code-based access control is very mature and offers several configurable policy levels. .NET provides hierarchical code groups and allows for targeted code checks.

User Authentication

The .NET Framework offers good authentication out of the box. It implements authentication through authentication "providers," such as forms, Passport, and IIS. .NET's close ties to IIS can hinder its flexibility.

Java code is easier to modify and the Java Authentication and Authorization Service (JAAS) is available for developers to modify and then plug in. JAAS also provides several levels of customization, making Java's authentication and role-based access control stronger than .NET's.

Both platforms, however, lack a mechanism for advanced user-based access control, such as permission delegation. For more complex role-based access control projects, users have to build their own layer of security for determining user-based access control.

Auditing and Tracking

Neither platform offers much support for secure authentication and tracking. Although JDK 1.4 introduces a logging package, it offers no secure facilities. .NET offers a managed wrapper around Windows *EventLog*, but that's as far as its auditing features go. Consequently, .NET applications are restricted to the functionality and limitations of *EventLog*.

Neither .NET nor Java provides acceptable support for auditing and tracking transactions. With .NET, developers can use the Windows mechanisms, but they need to go outside the .NET Framework to get them. Even when .NET and Java add logging packages, it is not clear how secure that mechanism would be.

Managed and Unmanaged Code

While the .NET Framework provides a solid security model through managed code in the CLR, the ability to run unmanaged code confers the ability to bypass CLR security through direct calls to the underlying operating system APIs. Also, in Java, signed and trusted code has unrestricted access to system resources. Java's calls to native code through the Java Native Interfaces (JNIs) confer the ability to bypass JRE's security. Similarly, running unmanaged code in the CLR can be used to bypass the .NET Framework security.

FUTURE OF RUNTIME SECURITY

The real test for runtime security facilities will be their deployment in large distributed systems. In their current models, the overall system security depends on perfect functioning of the application, the language, the virtual machines, and the underlying operating systems. It also depends on the interaction of those elements. Therefore, this kind of system security becomes very complicated and unstable if the system is very large. The experience of Java security has shown that most of the security problems reported come from defects in the implementation of the security mechanism and from malicious applets that use vulnerabilities in the applications that use the virtual machine.

The ability to encrypt communication and provide digital signatures is only part of enabling secure applications, trusted communication, and proof-of-identity. There is

* Other brands and names are the property of their respective owners.

still the issue of where and how the keys are generated and stored. Not only do the keys have to be exchanged over secure links, they have to be generated and then managed in a secure way. Hardware-based secure storage can play a major role in securing the generation and safekeeping of keys.

Security hardware may also provide more reliable random-number-generation, time-stamping, and auditing capabilities, which are crucial for cryptographic and signature functions.

Both Java* and the .NET* Framework include cryptographic capabilities based on software libraries. However, performing the encryption on hardware is inherently more secure than leaving it to the software. Furthermore, hardware-based cryptography, signatures, and key storage capabilities can provide a common security infrastructure for both Java and .NET, which can make the development of secure runtime applications much easier than having to develop code to invoke the Java or .NET cryptographic extensions.

CONCLUSION

Both Java's* JRE and .NET's* CLR do not meet Clark-Wilson's requirements of resource integrity. However, they both provide quite comprehensive security services, though each has a different focus.

Java's authentication and authorization services are fairly flexible. Although its use is not mandated, authentication and authorization functionality can be provided by the JAAS. .NET's authentication and authorization services, however, are provided through the Windows* operating system or identification stores (e.g., Passport*).

Both environments use similar concepts for handling user and code access to resources, with permissions being critical to both. The concept of roles is used to associate permissions with principals in both environments.

Common hardware-based cryptographic and key-management capabilities can drastically enhance the security of the runtime environment. However, getting the industry to agree on a common hardware architecture for mobile and non-mobile security will be a challenge for the next few years.

ACKNOWLEDGMENTS

The author extends his thanks to the reviewers who provided invaluable feedback. Special thanks to Gene Forte, Srinivasan Krishnamurthy, Joel Munter, Gururaj Nagendra, Murthi Nanja, and Carlos Rozas for their careful review of this paper. My gratitude is also extended to Norbert Mikula for generating many of the thoughts in the paper.

REFERENCES

[1] Seton-Williams, M.V., *Greek Legends and Stories*,. Barnes & Noble, Inc., New York, New York, pp. 103-111.

[2] Joy, B., Steele, G., Gosling, J., and Brasha, G., *Java Language Specification*,. Book News, Inc., Portland, Oregon.

[3] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*, Addison-Wesley, New York, New York.

[4] Gong, L., *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*,. Sun Microsystems Press, Santa Clara, California.

[5] Goulouris, G., Dollimore, J., and Kindberg, T,. *Distributed Systems – concepts and design*, International Computer Science Series, Addison-Wesley, Massachusetts and London, pp. 477-516.

[6] Schweitzer, D., *Securing the Network from Malicious Code: A Complete Guide to Defending Against Viruses, Worms, and Trojans*, John Wiley & Sons, New York, New York.

[7] LaMacchia, B.A., Lange, S., Lyons, M., Martin, R., and Price, K., *.NET Framework Security*, Addison-Wesley, Massachusetts and London, New York, New York, pp. 43-79.

[8] Summers, C. R., *Computer Security: Threats and Safeguards*, McGraw Hill, New York, page 142.

[9] Anderson, R., *Security Engineering: A Guide to Building Dependable Distribution Systems*, Wiley Computer Publishing, New York, pp. 188.

[10] Kunene, G., *Software Engineers Put .NET and Enterprise Java Security to the Test*,. http://archive.devx.com/enterprise/articles/dotnetvsjava/GK0202-1.asp.

[11] Aissi, S., Pallavi, M, and Krishnamurthy, S., "Ebusiness Process Modeling: the Next Big Step," *IEEE Computer*, May 2002.

[12] Adams, L., *ASP.NET security holes*, ZDNet, Australia, June 2002, http://www.zdnet.com.au/builder/architect/sdi/story/0,2000035062,20266124,00.htm

[13] Clark, D. D. and Wilson, D.R.,"A comparison of commercial and military computer security policies,"

* Other brands and names are the property of their respective owners.

IEEE Symposium on Security and Privacy, pp. 184-194, Oakland, CA, 1987.

AUTHOR'S BIOGRAPHY

Selim Aissi has been involved in the development of secure, safety-critical systems in the R&D sector, and in military, automotive, and wireless appliances for over twelve years. Before joining Intel in 1999, he worked at the University of Michigan, General Dynamics' M1A2 Abrams Battlefield Tank Division, General Motors' Embedded Controller Excellence Center, and Applied Dynamics International. At Intel, he played several management and senior architecture roles, and he is currently a Senior Security Architect at Intel's Research & Development group. Selim served on the review board of several publications and conferences. He currently serves on ACM's CCS'03, SAM'03, NCISSE'03, and IC'03 conference boards. He holds a Ph.D. degree in Aerospace Engineering from the University of Michigan and is a member of the IEEE, ACM, and ISSA. His e-mail is selim.aissi@intel.com.

For further information visit:

developer.intel.com/technology/itj/index.htm