I, Nandhiney Pichumani, hereby submit this work as part of the requirements for the degree of:

Master of Science

in:

Computer Science

It is entitled:

A Framework on Security with Trusted Computing

This work and its defense approved by:

Chair: Dr. Yizong Cheng

Dr. George Purdy

Dr. Kenneth Berman
A Framework on Security with Trusted Computing

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE
in the Department of
Electrical Computer Engineering and Computer Science
of the College of Engineering
18, October, 2004
by
Nandhiney Pichumani
B.Tech (Chemical Engineering), Anna University, Chennai, India, 2000.

Committee Chair: Dr. Yizong Cheng
Abstract

The goal is to build a framework that enables secure distribution of java executables across potentially insecure networks. The model provides an end to end solution for securing applications and enables trusted computing. The model enables copy protection and prevents software piracy.

The solution requires support from both hardware and software. The software solution encompasses the usage of cryptographic techniques. It invokes a Security manager with configured security policies. The execution is done on the fly leaving no trace of the executable.

The hardware solution requires the receiver to have architectural support for tamper-resistant software. The processor core will be the only trusted component of the hardware and will support cryptographic operations. It requires that a separate mode be activated where the contents of the memory have only execution privileges.

Simulations were performed to study the performance of the architectural change. An optimal size for the built in L2 cache, which provides a good balance between cache size and performance, was determined.
ACKNOWLEDGEMENTS

I am deeply indebted to my advisor Dr. Yizong Cheng for his help, suggestions, encouragement and support during the research and writing of the thesis. I would like to express my gratitude to Dr. Franco for helping me with the topic and gearing me in the right direction.

I would like to thank Dr. Kenneth Berman and Dr. George Purdy for making the time to be a part of my thesis committee.

My sincere thanks to David Lie, Edward Suh, Stevan Vlaovic, Blaise Gassend for answering all my queries, Martin Bochnig, Jan Bruun Andersen, Christophe Bothamy, Robert Montjoy, Chris Isbell for helping me with the Bochs installation, Nidhi Kejriwal, Doug Burger, M.S. Hrishikesh for helping me with the Simple Scalar installation and Lucky, Coleman for their initial guidance.

I express my heartfelt thanks to my parents, sister and Karthik for encouraging me and providing me all the moral strength and support.

Finally, I extend my thanks to all my friends who made my stay at UC a pleasant and enjoyable experience.
CHAPTER 4: DISCUSSION OF RESULTS ............................................................................................... 31

4.1 PERFORMANCE OVERHEAD DUE TO SECURITY MANAGER: ...................................................... 31

4.2 SIM-OUTORDER: .............................................................................................................................. 32

4.3 CYCLES PER INSTRUCTION (CPI): .................................................................................................. 33

4.4 CACHE MISS RATE: .......................................................................................................................... 41

4.5 BLOCK SIZE ANALYSIS: .................................................................................................................. 43

4.6 FAILED IMPLEMENTATION: ............................................................................................................ 46

CHAPTER 5: RELATED WORK ........................................................................................................ 47

5.1 TCPA 4: .......................................................................................................................................... 47

5.2 PALLADIUM 10: ............................................................................................................................. 48

5.3 AEGIS 3: ......................................................................................................................................... 49

5.4 FUTURE WORK ............................................................................................................................... 50

CHAPTER 6: CONCLUSION ................................................................................................................. 51

BIBLIOGRAPHY .................................................................................................................................. 56

APPENDIX A ....................................................................................................................................... 57

DEFINITION OF TERMS: ....................................................................................................................... 57
LIST OF FIGURES/TABLES

Figure 1.................................................................................................................................................. 13
Figure 2: The XOM Architecture........................................................................................................... 21
Table 1...................................................................................................................................................... 29
Table 2...................................................................................................................................................... 34
Table 3...................................................................................................................................................... 37
Table 4...................................................................................................................................................... 43
Table 5...................................................................................................................................................... 44
Table 6...................................................................................................................................................... 46
LIST OF GRAPHS

Graph 1 .................................................................................................................................................. 31
Graph 2 .................................................................................................................................................. 38
Graph 3 .................................................................................................................................................. 39
Graph 4 .................................................................................................................................................. 40
Graph 5 .................................................................................................................................................. 41
Graph 6 .................................................................................................................................................. 42
Graph 7 .................................................................................................................................................. 44
Graph 8 .................................................................................................................................................. 47
CHAPTER 1: Introduction

1.1 Motivation:

The motivation for the problem stems from the rapid growth in the myriad possibilities the Internet has to offer. Companies develop products (applications) and make it available for use by others. To ensure that business is profitable and to reduce potential risks, proper management of Digital Rights and several security issues need to be addressed before the product is shipped to the end user.

The sender (vendor) has to ensure that the receiver (customer) uses the application without tampering with the source code/secret data embedded in the software. In particular, the sender needs to be assured that the receiver cannot pirate the product and take undue advantage. There should be mutual trust between the person distributing the software (vendor) and the third party (customer) using the application. The receiver has to verify that the application is genuine and will not cause any undesirable effects on their computer before using the application.

1.2 Definition of Terms

Trusted computing in the problem context refers to providing authentication, copy protection and tamper-resistance. The genuine receiver alone will be able to use the application.
The application cannot be tampered with in any way. The original source code cannot be viewed or modified but merely executed. The execution will terminate immediately if there is any evidence of tampering.

1.3 Problem Scope:

The focus is on developing a solution to securely distribute Java executables (class/jar files) from a sender to a receiver interested in using the application.

1.4 Formal Problem definition/specification:

Sender’s end:
Java class files can be easily decompiled to generate the source code. There are several decompilers like Mocha, Dejavu and WingDis available for use in the market. Therefore, java class files cannot be distributed to the customer as such without the risk of giving the end user full access to the original source code.

Receiver’s end:
The Java Virtual Machine, which does the interpretation of the byte code needs to provide a secure interface for the executable to execute. The JVM has full access to the system
resources. We want to restrict JVM’s access to all resources and give the executable just minimal privileges.

The idea is to provide an interface where we know exactly what privileges the executable has. It will not tamper with any existing files, will not produce any undesirable effects.

Tamper-resistant architecture is critical to ensure that only the legitimate receiver receives access to the application. Once the receiver receives the application, the only privilege they should have is execution. They should not be able to view/modify/copy the applications content and utilize it to their advantage.

1.5 Approach to Solve the Problem

Enabling mutual trust between the vendor and the customer requires support from both hardware and software. Current hardware architecture doesn’t provide support for completely enabling this trust. We propose a model, which would work if proper hardware support as described below were present.

The software solution to the problem relies on cryptographic techniques to securely deliver the executable to the receiver i.e. we encrypt the executables and transmit them to the receiver. It is then deciphered and executed on the fly. We want to ensure that there is no trace of the executable (class file) at the receiver’s end. To accomplish this we execute the program on the fly (by using Class Loaders in Java).
The current JDK security model is depicted in the picture below (Refer Fig 1).

To provide a secure interface to execute the application, a security manager is required. Without the security manager, only the rights of the user running the program limit the program. Security policies dictate the privileges of the executable. These are configured according to the program requirements and invoked using the Security manager.

The receiver has a tamper-resistant hardware architecture like XOM\(^1\) (execute only memory), which ensures that code cannot be copied/viewed/modified but just executed. The private key is embedded in hardware and the processor core with its built in caches is the only trusted component of the architecture. External memory is not trusted.

The architectural changes are studied by performing simulations using Simple Scalar Toolset\(^8\)

![Diagram of JDK Security Model](image-url)  

**Figure 1\(^{15}\)**
1.6 Security issues addressed:

1. Mutual authentication between the sender and the receiver
   Enabled by using public key encryption and using Digital Signatures.

2. No tampering of the distributed software/executable at all stages
   Needs encryption at the sender’s end
   Tamper resistant hardware architecture in the receiver’s end.

3. Executable should not produce undesirable effects on the receiver’s machine.
   Should not use the system’s resources to its advantage and cause harmful effects on the receiver’s machine.
   Enabled by using security manager and trusted computing architecture at the receiver's end.

4. Protect the integrity of the system. Prevent killing processes/threads, modifying/deleting files, modifying memory.
   Enabled by invoking the security manager with the configured security policies.
1.7 Thesis Outline:

Chapter 2 describes the Foundations for the Research. It provides an outline to the software and hardware mechanisms used. Chapter 3 discusses the methodologies used to accomplish the goal. It describes the algorithm and discusses the hardware details. Chapter 4 entails performance study using simulations. Chapter 5 describes related research ideas floating in the industry. Chapter 6 concludes the thesis.
CHAPTER 2: Foundations of the Research

2.1 Handling Software Security:

Securing java applications thoroughly requires monitoring security in three different contexts:

a. Virtual Machine Security:

This refers to security provided at the JVM level. The Class loader, Security Manager and Byte Code Verifier provide security at the JVM level.

*Class loader* provides separate namespaces for local and remote classes.

*Security Manager* invokes security policies that are configured for the application.

*Byte Code Verifier* verifies that code adheres to JVM specification and doesn’t violate system integrity. It tests code fragments and uses theorem prover to check for illegal code (code that violates access rights on objects and tries to cast object types or class).

b. Application Security:

Java application security can be compromised due to implementation flaws and/or design flaws.
During implementation, *insufficient input validation* and *inadequate randomness* in the cryptographic operations can lead to security vulnerabilities.

At the design level, *cloning* and *deserialization* could lead to security breaches. Visibility (whether a class member is public or private) and Extensibility (whether a subclass can extend a class or method) when not properly designed can lead to security vulnerabilities. The *final* and *private* keyword should be used judiciously to circumvent these design flaws. The applications under study were carefully designed to prevent these implementation flaws.

Java- has some inbuilt security features- no pointers and runtime casting. This prevents illegal access to memory.

c. Network Security:

This refers to securing the communication channel between the two parties. It should protect the information from being eavesdropped or modified. It should address issues of providing Authentication, Confidentiality and Non-Repudiation. Identifying access points in the network and securing them is vital. Access points provide a window for passive sniffing activity that is undesirable.

Our solution relies on Cryptography to secure the communication channel between the two entities.
2.2 Trusted Computing:

The sender needs to know that the target program delivered is being executed unmodified and no other program running on the receiver’s machine is affecting its execution in any way (trying to get data, manipulate it or interfere with its proper execution). Architecture like XOM\(^1\) (Execute Only Memory) is required at the receiver’s end for enabling trusted computing.

Any data present on the memory bus is insecure. Tamper resistant architecture trusts only the processor core. Any information (instruction/data) that leaves the processor boundary will be encrypted.

2.3 XOM\(^1\) architecture:

XOM\(^1\), an acronym for Execute Only Memory is an architecture that does not trust the operating system or physical memory. XOM\(^1\) hardware manages access control, while resource allocation alone is done by the OS. The cryptographic keys and functionality are present in the processor.

It provides a high level of security for applications by providing a microprocessor architecture where separate compartments are maintained for every program (Process in one compartment cannot read data from another compartment). Session keys enable these on-chip
compartments. Process that has access to the session key alone can be in the compartment. Refer Figure 1 for the XOM architecture

Creation of compartments:

Cryptographic techniques are used to enable the formation of compartments. Asymmetric key cipher usage is more expensive when compared to symmetric keys. Therefore, a combination of symmetric and asymmetric key cryptography is used to enable the on-chip compartments.

A user program/application intended for a XOM machine is encrypted with a symmetric key and the symmetric key is then encrypted with the public key of the XOM machine. When the target XOM machine receives the program, it decrypts the symmetric key using the private key (which is embedded in hardware directly on the main processor) of the XOM machine and uses the decrypted symmetric key for creating the compartment. Certificates on the public key and the user program provide **authentication**. The encryption and decryption capabilities are built into the processor core.

The XOM ID:

Every user program in the XOM machine has a unique compartment key (session key) that is associated with a XOM ID tag. The XOM identifier (XOM id) is a shorthand (can be 1 bit...
for simple design) for the session key. The XOM ID is an index into a session key table (stored in private memory) which maps it to the decrypted session key.

Data produced by the program during execution is tagged with the active XOM id (XOM id of the program that is executing). When data is read, the tag on the data is compared with the active XOM id. The read operation is allowed only if the comparison succeeds, otherwise an exception is thrown eventually exiting the XOM mode. This ensures that compartments are isolated from each other and no other user program can access data from this program.
Figure 2: The XOM Architecture
There is a null compartment with session id 0 where programs run without encryption.

Additional Instructions:

To enable XOM, additional instructions are added:

Enter_xom and Exit_xom instructions mark the beginning and ending of operations in XOM mode.

Enter_xom:

XOM code is preceded by this instruction. It is an indication that the code that follows code belongs to a compartment with a session key. A check is done in the session key table to retrieve the XOM ID and if found, its status is set to active.

If no entry is found in the session key table, the key hasn’t been decrypted. The private key is used to retrieve the session key, a XOM ID is associated with it and an entry is added to the session key table.

In XOM mode the instructions are decrypted using the session key and placed in the instruction stream.
Exit_xom:

This is the normal terminating execution from the XOM mode. This instruction changes the active identifier to that of the null compartment.

The store_secure and load_secure instructions are used by the currently executing XOM program to move data between registers and external memory.

Store_secure:

This encrypts the contents of the tagged register with the session key, creates a hash of the location and stores that in the external memory.

Load_secure:

This decrypts the contents of memory using the active session key, checks the hash and loads a register. The register’s tag is replaced with the active XOM ID.
Secure Storage:

All data is tagged with a XOM identifier. External memory is untrusted in XOM and therefore when data leaves the XOM machine it is encrypted with the corresponding session key. A hash of data, MAC (to prevent spoofing attacks), which also combines the address of the data (to prevent splicing attacks) is appended to the original data and then stored in external memory.

To conclude, XOM machine has a mechanism to accomplish the following:

1. Decrypt the session key using the private key embedded in the processor.
2. Decode the program using the session key
3. Data tagging for program based isolation
4. Enter and Exit XOM instructions to enter and exit XOM mode.
5. Additional instructions to encrypt and decrypt instructions to and from external memory.
2.4 *Simple Scalar Tool Set*: 


Sim-fast:

This is the fastest and least detailed functional simulator.

Sim-cache and Sim-cheetah:

These are functional cache simulators.

Sim-safe:

Functional simulator that checks for correct alignment and access permissions for every memory reference.

Sim-outorder:

This is the most complicated and detailed timing simulator.

This was chosen for performance studies.
Chapter 3: Security with Trusted Computing

3.1 Software Security:

The communication channel between the vendor and the customer is secured by encrypting the executable. The encrypted executable is then transmitted to the receiver.

Encryption:

Algorithm Description:

AES\(^9\), Advanced Encryption Standard (Rijndael) was chosen for encryption. AES is the NIST (National Institute of Standards and Technology) and Federal Information Processing (FIP) Standard to protect sensitive information.

The algorithm has the following advantages:

- Provides good security
  
  If a machine could try \(2^{55}\) keys in one second, it would take a machine approximately 149 trillion years to break a 128-bit key.

- Has good performance
Performs consistently well (both hardware and software) different computing systems

- Is Efficient
  Has low memory requirements.
- Is easy to implement
- Is Flexible (different key sizes can be used)

Algorithm details:

The algorithm divides the input into blocks of 128 bits and works on one block at a time. The 128-bit input is divided into 16 bytes and arranged in a 4x4 matrix. A substitution box called Sbox is initialized. This is a 16 x 16 matrix.

AES has the following iterative steps:

1. Add Round Key
2. Substitute Bytes
3. Shift Rows
4. Mix Column
Implementation details:

AES is a symmetric block cipher algorithm. It was implemented in CBC mode (Cipher Block chaining) with a 128-bit key size.

Why CBC Mode of Operation?

If CBC is not used, identical plaintext will result in identical ciphertext, which makes it vulnerable to codebook attacks.

With CBC, the decryption of every block is dependent on the previous ciphertext block. Any corruption in a single ciphertext block will result in affecting the decryption of all subsequent blocks.

On the other hand if ECB (Electronic Code Book) mode is used, only one block of plaintext will be affected by a corrupted cipherblock.

Error propagates to all the blocks after the corrupted block and therefore makes CBC the best choice to provide maximum security.
Securing the symmetric Key:

To transmit the symmetric key securely, public key encryption is used. Elliptic Curve Cryptography (ECC) is used to transmit the key and Elliptic Curve Digital Signature (ECDSA) to verify the authenticity of the sender. The symmetric key is encrypted with the public key of the receiver (published publicly) and sent. The receiver decrypts the symmetric key using his private key (embedded directly on the processor). The deciphered key is then used for decrypting the executable.

On-the fly execution:

Once the encrypted executable and key are received by the receiver, the key is decrypted and verified by ECDSA. All these operations occur inside the processor, which is the only trusted hardware component.

The executable is executed on the fly by using ClassLoaders and Reflection in Java. The encrypted file is read, decrypted and executed but no trace of the .class file is left behind.

Security Manager and Security Policy:

Security policies are configured to give the executable minimal privileges.
The policytool is used to create the policy file. The runtime permission to create class loaders is given to the executable. This policy is enforced by invoking the JVM’s security manager during execution.

3.2 Hardware Security:

We assume that we have architecture like XOM\(^1\) where external memory is not trusted. We have a processor core with built-in caches that is trusted. Every instruction that leaves the processor boundary will be encrypted. All instructions on the memory bus and external memory will be encrypted. The original XOM architecture as described in Hardware Support for Tamper Resistant and Copy-Resistant Software\(^4\) was modified to include a built-in L2 cache. This should theoretically provide better performance because:

— It reduces the access time to memory. More instructions can now fit inside the trusted processor core.

— The overhead involved in encrypting and decrypting instructions that leave and enter the processor is minimized.

Simulations were performed to determine the optimal size for the built-in L2 cache that yields a good balance between performance and cache size.

SimpleScalarToolset\(^8\) is configured to execute PISA binaries. PISA is a MIPS-like instruction set that is primarily used for instructional use. The benchmarks are compiled for Power PC G5 processor to obtain maximum performance.
The detailed sim-outorder simulator is used for performance studies.

The following table summarizes the simulation parameters:

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Specification</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction Cache</td>
<td>64KB, 2-way, 64B line</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32KB, 2-way, 64B line</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Unified, 512KB, 4-way, 64B line</td>
</tr>
<tr>
<td>L1 Latency</td>
<td>2 cycles</td>
</tr>
<tr>
<td>L2 Latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Memory Latency (first chunk)</td>
<td>80 cycles</td>
</tr>
<tr>
<td>Issue/Decode width</td>
<td>8 instructions/cycle</td>
</tr>
<tr>
<td>Register Update Unit Size</td>
<td>128</td>
</tr>
<tr>
<td>Load/Store Queue Size</td>
<td>64</td>
</tr>
<tr>
<td>Front Bus Speed</td>
<td>1.25GHz, 32 B , 10GB/s</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.5GHz</td>
</tr>
</tbody>
</table>

Table 1
The following SPEC (Standard Performance Evaluation Corporation) 2000 CPU benchmarks (processor intensive) were used for measuring performance.

1. Anagram
   This is a pointer-intensive benchmark suite.

2. Compress
   Compresses and Decompresses Files in memory

3. GCC
   The GNU C compiler generating Sparc Code

4. GO
   Artificial Intelligence plays the game Go.
4.1 Performance Overhead due to Security Manager:

The overhead involved in invoking the Security Manager was studied by analyzing an Application and Graphics program. Refer Chart below. A profiling analysis was done to study the application performance.

![Graph 1](chart_link)
<table>
<thead>
<tr>
<th>Program</th>
<th>Application</th>
<th>Graphics</th>
</tr>
</thead>
<tbody>
<tr>
<td>OverHead</td>
<td>30%</td>
<td>12.44%</td>
</tr>
</tbody>
</table>

**Table 2**

Clearly, the overhead of invoking the security manager is significant and depends on the nature of the application. However, to ensure that the sender’s program will not cause any undesirable effects on the receiver’s machine, invoking the security manager is critical.

### 4.2 Sim-outorder:

The detailed simulator that supports out-of-order issue and execution was used for performance study. This allows us to model a very detailed cache organization unlike the cache simulators that are suitable only for miss rate study.

The configuration file was modified for different cache sizes.

Cache configuration is:

```
<name>::<nsets>::<bsize>::<assoc>::<repl>
```

- `<name>` cache name
- `<bsize>` block size
- `<nsets>` number of sets in the cache
<assoc> associativity of the cache

<repl> replacement policy, \( l = \text{LRU}, f = \text{FIFO}, r = \text{random replacement} \)

The cache size is the product of \(<\text{nsets}>\), \(<\text{bsize}>\) and \(<\text{assoc}>\).

4.3 Cycles per Instruction (CPI):

A metric for measuring application performance is execution time. Hardware performance depends on the following:

- Instruction set architecture
- CPU cycle time and
- Bus cycle time.

CPU cycle time = Number of instructions \( \times \) Clock cycles/instruction (CPI)

CPI is the number of CPU clock cycles that is utilized per instruction.

The original XOM architecture as described in Hardware Support for Tamper Resistant and Copy-Resistant Software\(^4\) does not have a built-in L2 cache and its performance was analyzed.
The simulation verifies that including another level of cache enhances the performance. The reduction in the cycles per instruction is at least 10% going from No L2 to a 128KB L2. For the Go benchmark it is very significant at 23%.

The benchmarks (Anagram, Compress, GCC and Go) were executed with four different L2 cache sizes – 128, 256, 512 and 1024KB.

The simulation cycles per instruction (CPI) used decreases as the cache size increases. Refer graphs 1-5 below. The decrease is significant when the cache size increases from 128 to 256KB and from 256 to 512KB. Increasing it from 512 to 1MB reduces the cycles per instructions significantly only for the GCC benchmark.

Increasing the L2 cache size yields better performance because it reduces the number of accesses to external memory. However, there seems to be a saturation limit on the cache size depending on the application. Any increase beyond that limit doesn’t enhance the performance significantly.

The Anagram benchmark produces no change in performance when the cache size is increased from 512KB to 1MB. There is a 5.5% decrease in the CPI used when the cache size increases from 256 to 512KB for the Compress benchmark. The decrease is only 0.24% when the cache size is increased from 512KB to 1MB. For the Go benchmark, the decrease in CPI is 7.17% when the cache size is increased from 256 to 512KB, but from 512 to 1MB the increase is only 0.53%.
However for GCC, the decrease in CPI seems quite significant 9.4% when increasing from 512KB to 1MB. This could be attributed to GCC being a bulky program (byte size almost 2GB) and the nature of GCC – Sparc Code Generator. The results are summarized in the table below:

<table>
<thead>
<tr>
<th>% decrease in CPI for Benchmarks</th>
<th>No L2-128KB</th>
<th>128–256KB</th>
<th>256-512KB</th>
<th>512KB-1MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANAGRAM</td>
<td>12.5</td>
<td>5.6</td>
<td>1.9</td>
<td>0</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>9.8</td>
<td>12.3</td>
<td>5.5</td>
<td>0.2</td>
</tr>
<tr>
<td>GO</td>
<td>23.2</td>
<td>14.1</td>
<td>7.2</td>
<td>0.5</td>
</tr>
<tr>
<td>GCC</td>
<td>11.2</td>
<td>13.6</td>
<td>9.1</td>
<td>9.4</td>
</tr>
</tbody>
</table>

Table 3
Graph 2
Graph 3
Graph 4
Go Benchmark

![Bar Graph]

Graph 5
Comparison of all Benchmarks

Graph 6
<table>
<thead>
<tr>
<th>Code Size</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anagram</td>
<td>85536</td>
</tr>
<tr>
<td>Compress</td>
<td>103136</td>
</tr>
<tr>
<td>GCC</td>
<td>2177840</td>
</tr>
<tr>
<td>GO</td>
<td>621056</td>
</tr>
</tbody>
</table>

Table 4

4.4 Cache Miss Rate:

The performance of a 2-level cache is evaluated as follows:

Average memory access time = Hit time\textsubscript{L1} + Miss Rate\textsubscript{L1} * Miss Penalty\textsubscript{L1}

Miss Penalty\textsubscript{L1} = Hit time\textsubscript{L2} + Miss Rate\textsubscript{L2} * Miss Penalty\textsubscript{L2}

To obtain good performance, low memory access time is desirable. To obtain lower memory access time, low cache miss rate is desirable.

The L2 Cache miss rate decreases as the cache size increases and saturates beyond a particular cache size depending on the nature of the application. Refer graph below:
Cache Miss Rate

Graph 7

<table>
<thead>
<tr>
<th>L2 Cache Miss Rate</th>
<th>128 KB</th>
<th>256KB</th>
<th>512KB</th>
<th>1MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>For Benchmarks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANAGRAM</td>
<td>0.7522</td>
<td>0.3541</td>
<td>0.2199</td>
<td>0.2199</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>0.3042</td>
<td>0.1658</td>
<td>0.016</td>
<td>0.0071</td>
</tr>
<tr>
<td>GO</td>
<td>0.3303</td>
<td>0.1929</td>
<td>0.1226</td>
<td>0.0425</td>
</tr>
<tr>
<td>GCC</td>
<td>0.2614</td>
<td>0.089</td>
<td>0.0086</td>
<td>0.0027</td>
</tr>
</tbody>
</table>

Table 5
The L2 cache miss rate is very minimal for all the applications when the L2 cache size is 512KB. There seems to be no difference in the cache miss rate for Anagram benchmark when the cache size is increased from 512KB to 1MB. This could be attributed to the benchmark being pointer intensive.

For ideal cache designs, higher hits and fewer misses are desirable. For L1 Caches, fast hits are desirable. For L2 Caches, there are fewer hits when compared to L1 Caches and therefore fewer misses are desirable. To achieve this goal, higher associativity and larger block sizes are desirable for the L2 cache.

**4.5 Block Size Analysis:**

The effect of varying the cache block size on a L2 cache size of 512KB was studied. In theory larger block sizes provide better performance. Different block sizes 64, 128, 256 and 512 were studied keeping the L2 cache size at 512KB.

The following table and graph summarizes the results:
Effect of Cache Block Size on Cycles per Instruction:

<table>
<thead>
<tr>
<th>Block Size/Benchmarks</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anagram</td>
<td>0.5885</td>
<td>0.5875</td>
<td>0.587</td>
<td>0.5868</td>
</tr>
<tr>
<td>Compress</td>
<td>0.6572</td>
<td>0.6577</td>
<td>0.6586</td>
<td>0.6611</td>
</tr>
<tr>
<td>GCC</td>
<td>0.9192</td>
<td>0.9016</td>
<td>0.8939</td>
<td>0.9023</td>
</tr>
<tr>
<td>GO</td>
<td>0.8285</td>
<td>0.8277</td>
<td>0.8284</td>
<td>0.8305</td>
</tr>
</tbody>
</table>

Table 6

Varying the block size doesn’t seem to affect the Cycles per instruction in a uniform way and does not produce significant results for the benchmarks under study.
Effect of Block Size on Cache Miss Rate:

The cache miss rate decreases as the block size increases. At 512 block size, all the benchmarks under study have a cache miss rate lower than 0.05.

A 512KB Cache with 512-block size seems like the ideal choice for the built-in L2 cache size.
Bochs\textsuperscript{10}, an x86 PC emulator was used to simulate the hardware. Bochs was on installed on a Solaris box (host operating system) and Windows was installed as the guest operating system. The installation was tweaked to emulate a CD-ROM drive. A disk image of JRE was created and installed on the guest Windows OS. Disk images are iso files created using mkisofs utility. The application program was also loaded as a disk image on the emulated drive. Bochs turned out to be an extremely slow functional simulator. The caches could not be simulated. It was not suitable for hardware performance studies.
Chapter 5: Related Work

5.1 TCPA:

The Trusted Computing Platform Alliance is an industry-wide initiative to provide a secure computing platform for providing information security. It was formed by HP, Compaq, IBM, Intel and Microsoft in 1999. It adds an encryption chip to the computer which would safely store the machine’s identity. The goal is to ensure that user authentication can be done to concerned parties. It would provide the ability to protect the platform secrets by using tamper-resistant chips. TCPA adds two security components TPM (Trusted Platform Module) and CRTM (Core root of trust measurement) to the existing platform and the first thing that is run during the boot process is CRTM. The CRTM takes the hash value of the next component to be loaded and reports it to TPM, which stores it in one of its registers and passes the control to the BIOS which recursively keeps loading components till the entire OS is loaded. A third party can get info about the integrity of the platform (i.e. is it in a trusted state) by providing the values stored in the TPM registers. Thus TCPA provides Platform attestation and has a mechanism for reporting Platform integrity. It provides protected storage using the tamper-resistant chip. Unlike XOM, the underlying OS and external memory are trusted and therefore it doesn’t provide fool-proof security.
5.2 Palladium:\(^{10}\):

Palladium, now called NGSCB (Next Generation Secure Computing Base) is a Microsoft initiative to protect software from software. Palladium is built on top of hardware similar to TCPA. It is a larger initiative than TCPA and includes changes to CPU, Memory and I/O.

Palladium adds a new bit to the architecture to distinguish b/w trusted and standard mode. In the trusted mode, the processor boots NEXUS, which is the secure OS (a security kernel). The SSC (Security Support Component, similar to TCPA’s TPM) verifies the identity of NEXUS by computing and checking its hash, which is then stored in a read-only register. The nexus interacts with the Palladium h/w and provides services to agents. The nexus implements access control and memory isolation. It executes in Kernel mode within Curtained memory.

Agents:

Agents are applications, which may be standalone or may provide some services to other applications running in the standard mode.

Palladium provides new security features like

Sealed Storage:

Seal storage so that only some programs can access it. This is enabled by using encryption with a key generated by using the SSC.
Attestation:

This is provided by checking the values in the read-only register after the boot of NEXUS.

Curtained memory:

Physical memory is separated into standard and trusted mode. Only processes in trusted mode can access pages marked as trusted. DMA is blocked using a DMA Exclusion Vector at the page level. This specifies which is part of curtained memory and therefore can’t accept DMA. A hardware memory map of 1 bit per page marks curtained pages.

Secure I/O:

User I/O is encrypted in the trusted mode so that there can be no spoofing /sniffing.

Palladium still has to trust the NEXUS and Curtained memory doesn’t provide program-based isolation like XOM.

5.3 AEGIS:

AEGIS is architecture for single-chip processors that can be used to build secure computing systems. Any component external to the processor is untrusted. It implements two solutions, one in which a part of the OS is trusted (Security Kernel Solution, which is very similar to Palladium) and the other where the OS is untrusted like XOM. The memory verification scheme used by AEGIS is different from XOM. AEGIS uses Cached hash trees for memory integrity verification.
5.4 Future Work

The idea could be extrapolated to distribute other executables securely. The focus of the study was distributing java executables. Other executables (C, C++, WINDOWS, DOS etc) could use the similar approach. The performance overhead of the additional security could be studied.

The performance impact of modifying L1 Cache parameters could be studied. The L1 cache size for our analysis was kept fixed at 64KB. The cache size and block size could be varied and the optimal size could be determined.

The Security Manager could be optimized to minimize performance overhead. The overhead of using the security manager is very significant. It could be optimized to mitigate the security overhead.
Chapter 6: Conclusion

A security framework was modeled to safely distribute java executables and enable trusted computing. At the software layer, security policies were configured and invoked with a security manager. Execution was done on-the-fly so that no trace of the executable was left behind.

At the hardware layer, a processor core with an embedded private key and built-in instruction level encryption and decryption capability is required to achieve copy protection and tamper resistance. A Level-2 built-in cache reduces processing time and memory access time. Performance studies were carried to determine what the ideal built-in L2 cache should look like.

The Simulation Cycles used per instruction and the L2 Cache Miss Rate were chosen for performance analysis. The CPI is a good measure of the overhead due to the additional security. The number of CPU cycles used per instruction should be small to achieve good performance.

The miss rate for the L2 Cache gives a good indication about the memory access time. For L1 caches, high hit rates are desirable. For L2 caches, there are fewer hits when compared to the L1 caches, therefore, lower miss rates are desirable.

Four L2 cache sizes 128, 256, 512 and 1024 KB were used keeping the block size at 64.
From the Cycles per instruction and L2 cache miss rate study, 512KB L2 cache size seems to be a good balance between performance and cache size.

Four block sizes 64,128,256,512 were evaluated keeping the cache size at 512KB. Simulations show that at 512 block size the cache miss rate is under 0.05 and optimal.

The incorporation of a 512KB L2 built in cache provided a 19% performance improvement for Anagram, 25% for Compress, 30% for GCC and 39% for the GO benchmark which is very significant.

Companies can employ this model to ensure that their products are delivered to people keeping their product secrets intact. The end user cannot tamper with their product in any way. This will prevent software privacy and ensure copy protection.

The model provides a means to enforce Digital Rights Management. The owner of the program can configure his own policies like play once, fixed trial period etc. and send it across to the end user. Since the security is built into the hardware, and we have a tamper resistant environment, we can be guaranteed that the owners policies are properly enforced.

The model can be employed for secure distributed computing. Pieces of a huge application can be distributed to different users and the results of computation can be verified. The end
user can compute a signature using his private key (in hardware) and send it along with the computed results to the owner. The owner can verify the signature and be assured that his computation took place in a tamper resistant environment. Since trusted computing is enabled, they have the guarantee that the computed result is genuine.


5. Omar Bakr, Hareesh Nair, Aman Narang, Tony Scelfo. *TCPA and Palladium: Are the benefits of hardware changes enough to justify an upgrade?*. MIT Course 6.805, December 11, 2002


Appendix A

Definition of terms:

Cloning:

Objects can be cloned by implementing the Cloneable interface. Internally it returns things of type object. The object type is the super class of all classes. This needs to be cast to the proper subclass type before it is used.

Deserialization:

This is the reverse process of Serialization. Serialization is the process by which objects are stored in file/stream before they are transferred to other processes across a network. This is accomplished by implementing the Serializable interface and using the Object Input and Output Stream classes. Serialized objects need to be deserialized before they are used.

Confidentiality:

This is the guarantee or assurance that information will not be disclosed to unauthorized personnel.
Authentication:

This determines the user’s identity and their access rights. Passwords are commonly used as authentication schemes. Digital Certificates and Digital Signatures provide authentication.

Non-repudiation:

This provides proof of the origin/delivery of data to protect the sender against a false denial by the recipient and vice versa. Digital Signatures provide Non-repudiation.

Digital Signatures:

This is an electronic signature that is used to authenticate the identity of the sender.

Digital Certificate:

This is an electronic means of establishing the credentials. A Certification Authority issues it.

Block Cipher:

Block cipher is a mathematical function that takes a plaintext block and a key as input and produces ciphertext block as output.
The mathematical function works as follows:

It is easy to compute the ciphertext block knowing the plaintext and the key

It is easy to generate the plaintext knowing the ciphertext and the key

It is computationally not feasible to find the key knowing the plaintext and ciphertext.

*Symmetric key:*

Same key is used for encryption and decryption.

*Cipher Block Chaining Mode:*

Each plaintext block is XOR’d with the previous ciphertext block before encryption. An initialization vector of a certain size is used for the first plaintext block.

This hides patterns in the plaintext block (randomizes the input).

*Public Key Cryptography (Asymmetric key):*

Different keys are used for encryption and decryption. A public, private key pair is used for encryption and decryption.