*Journal of Information Security*, 2010, 1, 1-10
doi:10.4236/jis.2010.11001 Published Online July 2010 (http://www.SciRP.org/journal/jis)

# Micro-Architecture Support for Integrity Measurement on Dynamic Instruction Trace

**Hui Lin[1], Gyungho Lee[2]**
[1]*ECE Department, University of Illinois at Chicago, Chicago, USA*
[2]*College of Information and Communications, Korea University, Seoul, Korea*
*E-mail*: *hlin33@uic.edu*, *ghlee@korea.ac.kr*
*Received July* 12, 2010; *revised July* 16, 2010; *accepted July* 20, 2010

## Abstract

Trusted computing allows attesting remote system's trustworthiness based on the software stack whose integrity has been measured. However, attacker can corrupt system as well as measurement operation. As a result, nearly all integrity measurement mechanism suffers from the fact that what is measured may not be same as what is executed. To solve this problem, a novel integrity measurement called dynamic instruction trace measurement (DiT) is proposed. For DiT, processor's instruction cache is modified to stores back instructions to memory. Consequently, it is designed as a assistance to existing integrity measurement by including dynamic instructions trace. We have simulated DiT in a full-fledged system emulator with level-1 cache modified. It can successfully update records at the moment the attestation is required. Overhead in terms of circuit area, power consumption, and access time, is less than 3% for most criterions. And system only introduces less than 2% performance overhead in average.

**Keywords:** Integrity Measurement, Remote Attestation, Software Vulnerability, Trusted Computing

## 1. Introduction

Nowadays, computer under different platforms interacts with each other through internet environment. Although this provides convenience and increased functionality, it is necessary to securely indentify software stack running in remote systems. Effective remote attestation mechanism has drawn lots of research interests. Trusted Computing Group (TCG) first standardized the procedure to launch a remote attestation [1]. As defined, the protocol consists of three stages: integrity measurement, integrity logging, and integrity reporting [2]. The function of integrity measurement is to derive a proper measure that is an effective representation of a given platform status. In order to narrow down the range of such measures, Trusted Computer Base (TCB) is defined as hardware components and/or software modules whose integrity decides the status of a whole platform. Consequently, integrity measurement can simply based on measures from the TCB, which reduce performance overhead in measurement and attestation. Integrity logging is the process of storing aforementioned integrity measure in protected storing space. This process is not mandatory, but highly recommended to reduce the overhead due to repeated calculation for integrity measurement. The last step, which is called integrity reporting, is to attest system based on the stored or calculated integrity measures.

Computer systems emphasize different security goals per contexts. While system integrity is more important in one situation, the other may concern more about data privacy. Integrity measurement is strongly related to security policy applied to specific computer system and consequently results in different attestation mechanism. TCG's specification describes an integrity measurement during system's booting process. This mechanism is called "trusted boot". At the very beginning, a hardware signature, which is stored in some security-related hardware components, is used as the root of the trust. Current hardware vendors design Trusted Platform Module (TPM) to provide such functionality. As each entity is loaded into memory, the integrity measures on the binaries are calculated one by one and form a trust chain at last. Unlike secure booting, system takes measurements and leaves them to the remote party to determine system's trustworthiness. TCG's attestation based on such a trusted booting is also called binary attestation [2].

Other integrity measurements still follow TCG's "measure-before-load" principle. Property attestation and semantic attestation both try to extract the high level property or semantic information from binary measurement. So it will be more efficient and effective to validate whether a security policy is hold or violated on such a measured property a priori. IBM's Integrity Measurement Architecture (IMA) based on the TCG's trusted booting extends the approach into application software stack. IMA is now a security module provided by Linux kernel since version 2.30 [3].

A good integrity measurement should be able to derive a reliable measure that represents the status of computer system. From the resulting measure, a challenger (the remote entity which is interested in attesting the system) should be able to tell the system's updated security-related capability such as whether the memory has been ever corrupted by attacker, or whether programs can be properly executed in isolation, or whether cryptography keys are securely stored, and so on. On the other hand, measurement procedure should be transparent to the local user and introduces little performance overhead.

Current integrity measurements face problems of gathering sufficient history information on what has been done to the computing device. When each entity is loaded into memory, measurement of its binary codes is recorded. However, there will be a "measurement gap" at the moment when measurement results are requested. System status may be different from the recording in measure. Furthermore, measurements are made directly on program's executable code residing in main memory. There exists another "behavior gap" between instructions executed in the processor and executable codes in the memory. The integrity measure of executable code in the memory can be a good measure to represent the system state. However, as different attacks occur from internet, this is becoming less sufficient for a remote challenger. For those programs running for a long time, such as server programs, a static measurement prior to execution may have little relation to the system status at the current moment. As a result, more accurate measurement, which can include program behavior, needed to tell challenger all history of bad behavior. This results in a better decision on trustworthiness of the system.

However, with more information included, overhead to measure programs' state increases. As a result, some measurements are targeted to specific data, such as processor control data, function pointer in memory, network traffic, intrusion detection, and so on. Measurement is often restricted in order to utilize only limited amount of information. Consequently validation of system against a certain security policy introduces little performance overhead. This policy-driven attestation or validation schemes are largely based on limited information specific per intended attack scenarios. The problem is that although it is efficient in their proposed situation, portability of such measurement is very low. In different situation, attestation may require a big modification which also exerts a large performance penalty.

In order to provide updated integrity measurement as system evolves, we propose an original dynamic instruction trace measurement (DiT) to include in the metric dynamic instructions-level behaviour in the processor with the help of simple micro-architecture modification. However, instruction-level trace can vary from time to time, with some part of the program being executed more frequent than the other. Directly recording the processor behavior causes lots of performance overhead and without increasing any accuracy. In stead of applying measurement in processor, we still perform the operations on the memory. As a result, most function interfaces provided before, such as the ones proposed in TCG or IBM's IMA, can be maintained.

Cache is an evolutionary design building a bridge between the memory and processor to reduce access delay. However, in this paper, we modify the structure of the instruction cache to the one similar to the data cache. The consequence is that instructions can also be written back to the memory. As program continues its execution, code region in its address space no longer stores codes loaded before execution but records instructions which are executed. We improve the integrity measurement for trusted computing in the following aspects:

1) Extending the measurement scope. When the security-sensitive program is loaded and starts execution, DiT writes back instructions into memory. Consequently, binary code located in its address space records instructions which are actually executed.

2) Facilitating attestation for different security policy. DiT only replaces static measurement with dynamic one. As a result, it changes little on the high level interface and provides a better general solution to diverse scenarios.

3) Writing back instructions does not require the involvement of operating system. Thus, DiT builds a connection between what has seen inside processor and what resides in memory. This procedure does not require trusting operating system, which in some cases can be corrupted by attackers.

The paper is structured as follows. Section 2 presents the background on trusted computing and integrity measurement. In Section 3, we present DiT's design in details. To avoid potential hazards from attacks, we propose several hardware-wise recommendations in Section 4. The experimental results and analysis are given in Section 5. Finally, the related work and conclusion are made in Section 6 and Section 7.

## 2. Background

### 2.1. Trusted Computing

Trusted computing deals with computer system in a hazard environment. Though there is lack of ubiquitous definition of trust, this paper refers the one from Trusted Computing Group (TCG) specification. Trust is mentioned as the expectation that a device will behave in a particular manner for a specific purpose [2].

Trusted Computing Base (TCB) is specified as any hardware and/or software components within the interested platform, whose safety can affect the status of the whole system. The assumption is made that if TCB is safe, system can be trusted. However, TCB's components vary from systems. In some situations, it may work with integrity validation mechanism; as a result, run-time critical data values are included in TCB. However, on other situations, execution of security-sensitive programs, such as encryption/decryption operation, is important to system's proper function; some architecture components, which guarantee privacy of such application program, are chosen in TCB. TCG has summarized diverse application scenarios and concludes that it should include the following two characteristics:

1) Isolated Execution, or protected execution. The computing platform should be able to equip security-related application program with an isolated environment. As a result, no other legacy programs can access or corrupt information it relies on. To achieve this property, many researchers adopt the virtualization approach or hardware extension to legacy computer architecture [4].

2) Remote Attestation. Each computing platform should be able to provide mechanisms to: (1) securely measure TCB's safety state; (2) protect measure log stored locally; (3) transmit measure to remote challenger.

### 2.2. TCG's Binary Attestation

TCG defines a binary attestation to provide a trusted booting. Whenever an entity is loaded into memory from the moment machine is physically turned on, TPM applies cryptographic hash function, say *Hash*, on its executable code to make a measurement result, say *M*. The binary measurement for each entity is logged separately. Additionally, each measurement is also stored in one of Platform Configuration Registers (PCRs) in TPM by making the cryptographically *extend* operations with PCR's current value, $\mathrm{PCR}_t$, *i.e.*, new PCR values $\mathrm{PCR}_{t+1} = Hash(\mathrm{PCR}_t|M)$, where|denotes concatenation. When verifier requires attestation, TPM sends measurement logs (in local hard disk) and the corresponding PCR value to the verifier. He will recalculate hash result based on measurement logs. The comparison between newly-computed hash result and PCR value can tell whether untrusted behaviour within the environment has ever modified PCR value, measurement log, or executable code itself.

Using binary attestation facilitate verification in mainly two aspects. 1) measurement with such format hides many different high-level implementations and reduces the complexity to calculate measure log and PCR value; 2) It successfully separates measuring and verification. Attestation does not try to prevent a system from illegal behaviour that might compromise system. It only records the history of loaded code, securely sends them to the verifier and leaves the verifier to make trustworthiness decision.

### 2.3. Integrity Measurement on the Application Program

Starting from the root of trust provided by TCG, Integrity measurement architecture (IMA) from IBM takes the first step to extend measurements from booting process to application level programs. IMA is provided as a software module to Linux kernel from the version 2.30. It provides measurements regarding to current system's software stack. The whole project provides integrity measurement but does not propose any detailed attestation mechanism. Measurements provide evidences showing whether system is corrupted by certain rootkit attacks or not.

IMA measures each individual component before it is loaded. With the help of *extend* operation, trusted booting forced execution to follow only one legal order. However, in application level, programs can execute different threads in parallel; program order does not related to trusted condition any more. So IMA groups measure together instead of applying extend operation one by one.

But IMA's is following TCG's "measure before loading" principle, therefore it inevitable maintains shortcomings of the binary attestation, such as its ineffectiveness to reveal hardware attacks or the software attacks after the program is loaded and executing.

## 3. Architecture Extension to Measure Instruction Level Behaviour

### 3.1. Design of Integrity Measurement in Application Level

DiT is based on IBM's IMA which provides comprehensive measurement over software stack. In IMA, all executable codes and chosen structured data are included in the measurement log. Any data which are loaded by operating system, dynamic loaders, or applications with identifiable integrity semantics are hashed. Measurement can be made automatically at the moment when codes or data are loaded into main memory. As programs continue their execution, kernel is able to measure its own changes. Similarly, every user level process can measure its own security sensitive inputs, such as its configuration files or scripts. The consequent 160 bit value from hash calculation becomes an unambiguous identity for such software module. Challenger can distinguish different file types, versions, and extensions by this unique fingerprint.

As system evolves, IMA collects hash results into a measurement list which is stored locally. The integrity of this list is of a great importance. Therefore, IMA uses TPM to prevent any modifications made on measurement list. Platform Configuration Register whose value can only be changed by physically system rebooting or TPM extend operation provides protected storage. Extend operation is applied on each value stored in the measurement list. Since it is impossible to restrict application-level softwares into a small number of orders, order of each value in the list is not used to validate the trustworthiness of the system.

## 3.2. Writing Back the Instructions

Although IMA provides measurement of all loaded software, it still follows TCG's "measure-before-loading" mechanism. As a result, "metric gap" and "behaviour gap" can largely degrade efficacy of measure log.

The "metric gap" occurs when measurement does not represent the updated state of the system. Application program can run for a long time, such as server program. So it may be a long period since the measurement is made. During this time interval, memory is possible to be corrupted. Attacks, who can take root privilege, can modify loaded executable codes. However, it is possible to detect such modification when the codes are being executed again. This is the basic assumption made in former tamper resistance design [5]. As executable code is hashed again, resulting measure will be different. However, attestation is made asynchronously to system's operation. It is possible that attestation is made before executable codes are hashed again. As a result, measurements may give challenger a misinformation about what is running at the moment.

**Figure 1** makes a comparison between three measurement mechanisms: DiT proposed in this paper, IMA and Aegis which is a typical secure processor design to achieve tamper evidence and resistance environment [5]. When IMA measures executable code, it makes comparison to values which are calculated before. In Aegis, if software's execution relies on a program, the measurement of this program is calculated again and comparison is made to former calculated value. In these two situations, the challenger may still get measurements from which the system can regarded as trusted but actually the memory is already corrupted.

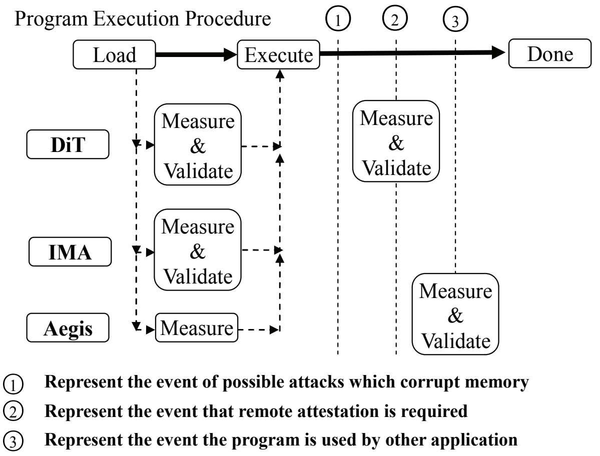

Program Execution Procedure
①
②
③
Load
Execute
Done
DiT
Measure & Validate
Measure & Validate
IMA
Measure & Validate
Aegis
Measure
Measure & Validate
① Represent the event of possible attacks which corrupt memory
② Represent the event that remote attestation is required
③ Represent the event the program is used by other application

**Figure 1. "Metric Gap" occurs in the design of IMA and Aegis.**

"Metric gap" can be resolved by applying a measure to executable code at the moment of attestation is made (which is also reflected in **Figure 1**). However, "behaviour gap" can further introduce more severe problem. This describes the fact that static codes in memory are different from instructions executed in processor. But it is instructions executed in processor finally corrupt the system. On the other words, executing instructions are truly represent the trustworthiness of the system. What makes things worse is that many attacks do not rely on the modification on program's executable code to launch malicious behaviour any longer. For example, buffer overflow attack has diverse implementations. One of them is to insert codes directly in stacks which make detection only possible for a very short period of time. Challenger should also be able to know such deleterious execution since this system is vulnerable to attacks in the future.

No matter how attacks exploit software vulnerability, it finally needs to execute its code in the processor. As a result, researchers also propose to records behaviour in the processor. To reduce performance overhead, they only analyse behaviour of critical instructions, such as indirect branch or critical data. Measuring those data may work for certain security policy but lacks of portability and extendibility to future unknown attacks.

Measuring all instructions is a challenge. Instructions are fetched from memory, but dynamic execution flow varies from situation to situation. It is impossible to provide limited number of unique state to represent safety of such execution. On the other hand, collecting all possible states are computationally impossible to make.

DiT does not directly measure all executed instruction in processor. It maintains large part of original measurement interfaces which measure codes in memory. What DiT successfully makes is to extend architecture's pipeline to build connection between processor and memory (**Figure 2**). It proposes to store back instructions into its original locations after they are fetched into pipeline. The

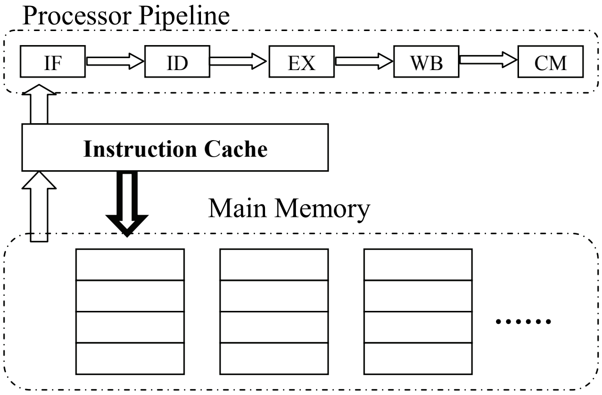


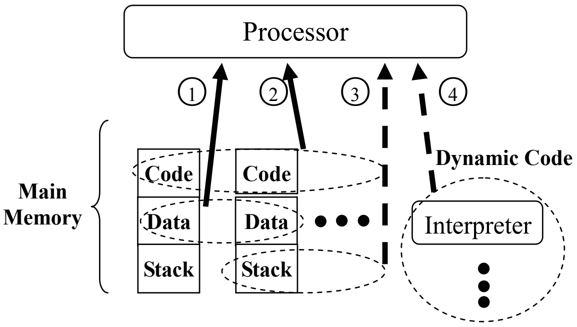
**Figure 2. Strcture to measure dynamic instruction trace.**

purpose is to resolve "behaviour gap" between processor and memory. This is not an intention to record all possible run-time execution paths but to store instructions which are truly executed into measure log.

With such modification, what to measure and when to measure have to be carefully designed. Program's address space consists of data region, code region, and stack to record program execution context. In IMA, all executable code and part of related data, which are dynamically loaded by operating system, are measured (**Figure 3**). DiT will cover all code regions, data regions and stack as long as there are some instructions being written back to them.

Due to attacks, instructions can come from other locations rather the code region. This not only makes DiT to expand measurement range to include memory region such as stack, but also require it to add several temporal points to make such measurement. We can still use the aforementioned buffer overflow as the example. Stack contents vary as program enters into different contexts. Malicious code hidden there may soon be overlapped by unrelated information, such as parameter passed by following function call. As a result, malicious code should be measured on time before it is eliminated by legal ones.

To insert proper temporal points is a trade-off between detection ability and performance overhead. The performance overhead in original integrity measurement mechanisms is amortized, which is due to the fact that hash calculation is made at the frequency of program loading. From many former anomaly-detection approaches, successful corruption usually results in some changes in instructions level behaviour, such as cache miss, prediction miss and so on [6]. Furthermore, hash operation, which calculates memory code, is easily performed in parallel with program's normal operation. In the current work, one inevitable measurement is added. DiT launches the measurement at the moment of attestation requirement is made, which at least resolve the metric gap between measure and system state.



①: **Data with integrity semantic is loaded by operating**

②: **Executable codes are fetched from memory**

③: **Malicious codes are fetched in statck or other illegal location**

④: **Codes are executed dynamically**

**Figure 3. Behavior gap occurs due to attacks or dynamic generated code.**

## 3.3. Introduce Randomization through the Use of Cache

Most personal computers usually have two level caches. Instruction and data are divided in the level-1 cache while level-2 cache is usually a unified cache which stores them together. DiT includes cache into the procedure of writing back instructions to the memory which "reverse" the procedure when instructions are fetching from it. In order to make write back work, instruction cache should be appended with few state bit just as data cache does.

By replacing structure of individual cache to the one of data cache, processor actually does not need to have the actual action of "writing back". It only needs to set a corresponding status bit and leave the work to cache and memory management unit. Whenever cache miss occurs, instruction cache first stores values in cache entry back to the memory and then read other instructions instead of overwriting it directly.

Usually, it is hard to predict cache miss. This randomizes the time to write back instructions. As a result, another level of protection which prevents attacks from learning this measurement and hide its malicious codes can be made. Besides, this operation does not need the involvement of operating system. Even when OS is not trusted, such as the kernel is corrupted, writing back operation can always be executed properly.

Current micro-architecture design can further help our design to write back instructions. Since level-2 cache is unified, only level-1 instruction cache requires modification. And the modification is restricted to small number of status bits added to each cache entry. As a result, overhead on chip area, power consumption and access time to cache entry (which is also called cache hit latency) is reasonable. Furthermore, instructions usually holds much better locality references than data cache which results in much less cache miss. Consequently, performance effect from writing back instructions is also possible to be restricted to a small amount.

## 4. Further Micro-Architecture Recommendations

With the proposed design, DiT is able to measure large amount of program's execution. However, it may still miss some situation due to current operating system design as well as diverse attacking mechanism. In this section, we propose several extra hardware recommendations to further resolve those issues.

### 4.1. Adding Measurement Point

With the aid of DiT, measurement will be recalculated with program's execution. There is still a possible hazard that attacker replaces correct codes to the malicious ones (that he injects before) in memory to avoid proper measuring (similar to the way he/she can insert malicious code) after malicious codes are stored back. As a result, adding more measurement points is necessary to provide another protection level on DiT itself.

The cache miss or branch prediction miss indicate a behaviour change in instruction level, which can be used as a point to recalculate measurement. To further reduce performance, we propose to make the measurement at the moment when the potential attacks are going to happen. However, from current study in software vulnerability, to detect the proper attacking potential is proved to be another difficult issue. As program is running, its address space records its execution state through the use of stack and/or heap and so on. However, its code space remains stable. Operating system design provides a good protection when it launches different code space to execute, such as the design of context switch. However, attackers successfully inject or exploit new or existing code space to avoid reliable operation provided from operating system.

As a result, we can make measurement when instructions are written back to the memory location which is outside of the code region (not address space) for the current running programs. As each program is loading its code, we can records its physical address in memory into a table and store it in a memory management unit. A comparison between written back instruction and each physical address of a code region can indicate which program this instruction is belong to. If it does not belong to any legal program, we can raise an exception. On seeing this exception, measurement is not also necessary since action of avoiding measurement is made.

By such architecture recommendation, DiT can achieve the validation such that every instruction executed in processor should be from executable code space which is properly loaded into memory before. Consequently, DiT can prevent injected code attacks while making measurement.

### 4.2. Measuring the Run-Time Generated Code

Different from compiler which generates executable codes, interpreter executes machine instructions on the fly. In our proposed design, integrity measurement is only capable of measuring binary codes of interpreter itself, dynamic codes generate by interpreter to processor are not recorded (**Figure 3**). On the other hand, more popular attacks begin to adopt this mechanism. Such attacks, including sql injection, cross script attacks, dominate current web applications. This presents a big challenge to provide accurate measurements to remote challenge, as malicious behaviours are extracted from user input and getting execution one instruction by another. Measuring executable codes from memory becomes impossible.

When instructions are generated from interpreter, DiT finds that there is no source memory location to which such dynamic instructions can transmit. Our proposed method is to "deceive" the interpreter that the dynamic executed codes is actually dynamic loaded. As a result, it can follow the predefined procedure to make such measurement.

This is achieved by creating a new memory region which can be linked to the memory space of interpreter's process. Current operating system, such as Linux kernel, provides safe interface to dynamically add or remove memory region from process' address space. It will be easy to include such secondary code region to interpreter's address space.

This is equivalent to adding a container to store dynamically executed code; however, the measurement will not be possible at the "load" time, since the container is empty at this moment. Only at the end of execution when all executed codes are written back, proper measure is going to be made on the full container.

## 5. Experiment and Result Analysis

In order to analyse applicability of DiT, two sets of experiments are conducted respectively. The first one simulated measurement mechanism, especially the situation to hash program's code upon asynchronous attestation. Then another set of experiments are made to detect hardware and performance overhead caused by modification on level-1 instruction cache.

## 5.1. Implementing Measurement

Different from IBM's IMA which implements all integrity measurements within Linux kernel, we implement it in the hardware level. DiT is integrated into Bochs which is a full-fledged open source × 86 PC emulator. It is used to emulate entire system from × 86 architecture to virtually instrumented monitor.

Through our experiment, we find that write back instructions to memory causes some instability for emulated system. As a result, DiT focuses on certain target program and only stores its on-fly instructions into memory. As mentioned before, TCB provides an isolation execution environment for the security-related programs. By implementing writing back instructions for only interested program, we believe that DiT can more practically simulate TCB's execution model.

We install Gentoo Linux with the kernel of version 2.6.29 in the emulation. To track process information, kernel is modified so that hardware emulator becomes aware of software context switch. Since version 2.6.x, kernel introduces the late binding for the context switch, so both *exec* () and *sched* () functions are modified. Consequently, process identity, such as Process ID and Process name is updated into a global variable as soon as process is created and loaded into memory.

Besides the operating system modification, we also implement several virtual debugging monitor. One of the most critical interfaces which DiT inserts is the one that halts the execution of current program in emulated operating system and hashes the code region in the address space of current active process. This efficiently emulates the situation that measurement is made upon the attestation request is sent from remote challenger.

## 5.2. Performance Overhead

In order to make instructions cache to write back, several extra status bits are required to each cache entry which is similar to the structure in data cache. Since in most micro-architecture design, level-2 cache is designed as a unified cache, only level-1 instructions cache needs modifications. To make a comprehensive analysis of such change, area, power consumption and access time is emulated under CACTI 5.0 [7]. The parameter of unmodified cache is the same as the one used in **Table 1**, which is also used in SimpleScalar for performance experiment. Five extra bits are added to each entry of the instruction cache to implement the write back mechanism. With the simulation results given from **Table 2**, largest power overhead is less than 10%. Overhead of other criterion is actually ignorable. Especially, modification has little effect on access time of level-1 cache.

**Table 1. Architecture parameters.**

| Parameter | Value |
|---|---|
| Fetch/dispatch/issue width | 4 |
| Instruction window | 128 entries |
| register update unit size | 128 entries |
| Load/Store Queue | 64 entries |
| I-cache | 128K 1 way set-asso., 1-cycle hit time |
| D-cache | 128K 1 way set-asso., 1-cycle hit time |
| L2 cache | Unified, 1M, 4 way set-asso, 6 –cycle hit time |
| Memory | 100 cycles access time, 2 memory ports |
| Function unit | 4 Int ALUs, 1 Int MUL/DIV, 4 FP Adder, 1 FP MUL/DIV |

**Table 2. Area, power and access time overhead for modified L1 cache.**

| Technology node | Overhead criterion | Normal L1 Cache | Modified L1 Cache | Overhead |
|---|---|---|---|---|
| 90 nm | Area (mm^2) | 2.59811765 | 2.66909173 | 2.73% |
| | Power (W) | 5.23044172 | 5.23787143 | 0.142% |
| | Access time (ns) | 1.40756434 | 1.40756434 | 0.00% |
| 32 nm | Area (mm^2) | 0.36714162 | 0.36929974 | 0.588% |
| | Power (W) | 3.54005779 | 3.87976541 | 9.59% |
| | Access time (ns) | 0.43442463 | 0.43875809 | 0.998% |

We tested SPEC2000 benchmarks running in Simplescalar which models an out-of-order superscalar processor [8]. Reference inputs are adopted and we skip instructions of the number which is specified by SimPoint [9].

Writing back instructions are not supported in Simplescalar, as a result, we modify source codes of sim-outorder (the out of order simulators) such that right after each time a read access is performed to the level-1 cache, a write access to the same entry in the cache is launched. The parameter to run Simplescalar is given in **Table 1**.

We collect all number of level 2 cache access and cache misses for each program in SPEC 2000. The number of level-2 cache access varies to different programs. In *eon*, *perlbmk* and *vortex*, the modified level-1 cache increases more than 50% of level 2 cache accesses. But for other benchmarks, the change is not that obvious. We only select the increase of level-2 cache access with more than 0.01% among all 26 programs (**Figure 4**).

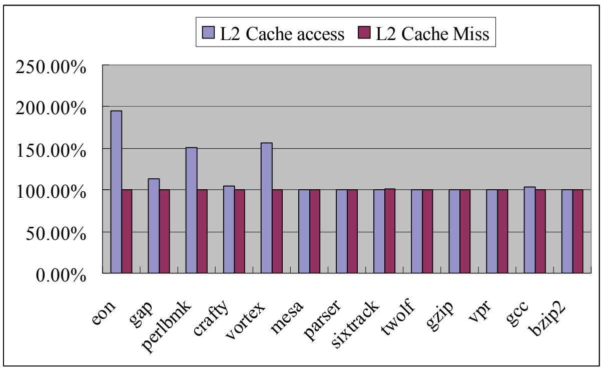


**Figure 4. Normalized Level-2 cache access and cache misss.**

Although there are big increases in level-2 caches access, this does not simply increase the corresponding cache miss. All cache miss due to the modification of level-1 cache is increased with less than 1%. This is probably due to the fact that level-2 cache holds a good locality references for instructions. As a result, performance overhead for all benchmark programs is ignorable as shown in **Figure 5**. The largest performance overhead measured in IPC is less than 5%.

## 6. Related Work

### 6.1. Tamper Resistance Design

Execute Only Memory (XOM) has included whole memory space in the trusted computing base as most adversaries launch the attacks to corrupt memory [10]. In order to guarantee both integrity and privacy of the data in memory, encryption components are included in the legacy architecture design. Data transmitted from processor to memory is encrypted and reversely, they are decrypted for execution in processor

Aegis [5] follows the same assumption that memory can not be trusted. It hashes executable code when a program is loaded into the memory for execution. At this moment, any other code and data that the program relies on is checked to guarantee that the program is started in a trusted environment. In the situation that operating system can not be trusted, Aegis introduce security related module and hardware component into the legacy processor. Tamper resistance design does not make assumption on how memory is corrupted thus it is able to detect simple hardware attacks.

Tamper resistance design is similar to our approach in the way of measuring untrusted code. However, they are holding the assumption that detection of static code can be found on moment the software is used again. As mentioned before, attestation can be made before next-use of software modules, so directly adopting tamper resistance approach introduces “metric gap”. On the other hand, they are unable to measure program’s runtime behaviour as well.

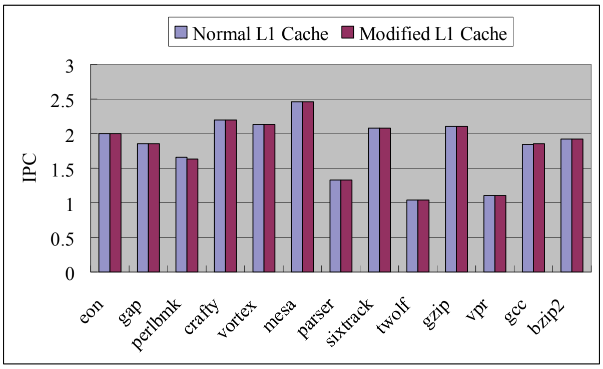


**Figure 5. Comparison of IPC number with normal Level-1 cache and modified L1 cache in processor.**

### 6.2. Integrity Measurement

TCG first standardize the procedure to make a remote attestation, besides, it also recommends an integrity measurement methods which is efficient during system booting. This binary attestation can only record what the programs are running on the platform and use the identity and the loading order of programs to system state after booting.

IBM’s IMA, Integrity Measurement Architecture, inserts measurement interface into Linux kernel. As each program is loaded into memory, its executable code is hashed. When a program is further loading other codes or security-critical data structure, measurement is made as program transfer its control flow. However, software vulnerability which is exploited by attackers during each individual program’s execution can also spoil measurement.

Based on the observation that modifications made in kernel space is usually permanent, Loscocco *et al.* propose to measure dynamic data structure which is critical to kernel control flow [11]. Such dynamic data structure is called contextual information, which is used to represent the state of the whole computing system. But this method is not efficient to be used in the user space operations.

### 6.3. Property Driven Remote Attestation

Binary measurement has the advantage of easy calculation and application-independence. Since hash calculation is irreversible, directly exploiting such metrics provides a big challenge and performance overhead. As a result, different attestation, which adopts different metrics, is proposed.

With specific security policy being set for the attesting system, property attestation and semantic attestation [12-14] propose to derive system high level information instead of the pure software stack. The extracted metrics can be directly used against security policy. Measurement methods may be implemented differently, but measurement is decided by security policy. As security policy changes, it is less flexible to change measurement implementation accordingly. As they indirectly include validation part into attesting platform, attesting platform's performance overhead is increased and validation procedure is also put under the hazardous environment. We propose DiT which designs an application-independent measurement which separates validation and measurement just as binary attestation does.

Some other researches also consider that program's run-time behaviour as a validation metrics, however, with many limitations. Alam *et al*. propose a behaviour attestation method [15]. However, the behaviour is defined as the quality of service the system can provide, connection latency, and so on. Consequently, this attestation implementation designed for web services only which lack the portability to be applied to other applications programs.

## 7. Conclusions

Ever since TCG standardized the procedure to launch a remote attestation, how to exchange the trust measure efficiently between computer systems under diverse platoforms has been a popular open research issue. Locally, attesting mechanism derives integrity measure based on software stacks on which trust decision is made. TCG introduces a binary attestation during system booting and many integrity measurement implementations are proposed following the "measure-before-loading" principle. Those measurements do not take into the account the actions after each program begins its execution. As a result software vulnerability which can corrupt both system status as well as measurement operation can introduce the "behavior gap" and the "metric gap" between program runtime behavior and consequent measurement. DiT, the dynamic instruction trace integrity measurement, is proposed as assistance to the current integrity measurement methods. By changing the structure of instruction cache, instructions are stored back into memory when cache miss occurs. As a result, code region in programs address space actually contains dynamic instructions trace executed in processor. By applying integrity measurement based on this change, DiT successfully include most updated system state to the moment when attestation is required.

We have experimented this attestation mechanism in *bochs*, a full-fledged emulator, with a current updated version of Linux kernel installed. We have successfully simulated the procedure of measuring program's code (or trace) at the time when attestation is made. To further analyze the change made in level-1 instruction cache, Cacti is exploited to check area, power consumption and access time overhead. SPEC2000 benchmarks are run on the modified Simplescalar to analyze the performance overhead. As we only limit our small modification in level 1 instruction cache, the overhead in terms of circuit area, power consumption, and access time are all reasonable, and also the performance overhead is marginal.

## 8. Acknowledgement

This work was supported by the IT R&D Program of MKE/KEIT (2010-KI002090, Development of Technology Base for Trustworthy Computing).

## 9. References

[1] "Trusted Computing Group." http://www.trustedcomputinggroup.org

[2] TCG Specification Architecture Overview Specification Revision 1.4, Trusted Computing Group (TCG), 2007.

[3] IBM Integrity Measurement Architecture (IMA). http: //domino.research.ibm.com/comm/research_people.nsf/pages/sailer.ima.html

[4] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter and A. Seshadri, "How Low can you Go Recommendations for Hardware-Supported Minimal TCB Code Execution," *Proceedings of ASPLOS*'08, Seattle, Vol. 43, No. 3, 2008, pp. 14-25.

[5] G. Edward Suh, D. Clarke, B. Gassend, M. Dijk and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *Proceedings of ICS*'03, San Francisco, 2003, pp. 160-171.

[6] Y. X. Shi and G. H. Lee, "Augmenting Branch Predictor to Secure Program Execution," *Proceedings of DSN* 07.

[7] http://www.hpl.hp.com/research/cacti/

[8] T. Austin and D. Burger, "The SimpleScalar Tool Set," University of Wisconsin CS Department, Technical Report No. 1342, June 1997.

[9] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proceedings of the* 10*th ASPLOS*, California, Vol. 37, No. 10, 2002, pp. 45-57.

[10] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, *et al.*, "Archi-

tectural Support for Copy and Tamper Resistant Software," SIGPLAN Notice, Vol. 35, No. 11, 2000, pp. 178-179.

[11] P. Loscocco, P. Wilson, A. Pendergrass and C. McDonell, "Linux Kernel Integrity Measurement Using Contextual Inspection," *STC*'07: *Proceedings of the* 2007 *ACM Workshop on Scalable Trusted Computing*, Virginia, 2007.

[12] L. Chen, R. Landfermann, H. Lohr and C. Stuble, "A Protocol for Property-Based Attestation," *Proceedings of STC*'06, the ACM Press, Virginia, 2006, pp. 7-16.

[13] A. Sadeghi and C. Stuble, "Property-Based Attestation for Computing Platforms: Caring about Properties, not Mechanisms," *Proceedings of NSPW*'04, New York, 2004, pp. 67-77.

[14] V. Haldar, D. Chandra and M. Franz, "Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing," *Proceedings of VM*'04, San Jose, 2004, p. 3.

[15] M. Alam, X. W. Zhang, M. Nauman and T. Ali, "Behavioral Attestation for Web Services (BA4WS)," *Proceedings of the* 2008 *ACM Workshop on Secure Web Services*, 2008.