How to Hide Secrets from Operating System: Architecture Level Support for Dynamic Address Trace Obfuscation

Abstract— **The adversary model for digital rights management is much more powerful than for the traditional security scenarios. The adversary has complete control of the computing node – supervisory privileges, physical as well as architectural object observational capabilities. In essence, this makes the operating system (or any other layer around the architecture) itself the adversary. The repercussions of this observation are severe. It creates a need to "keep secrets" from the operating system (OS). It isolates the architecture from the operating system.**

We argue for the need to keep secrets from the OS in hardware. This concept is demonstrated through architectural support for the obfuscation of dynamic address traces on the memory bus. The objective is to leak as little information about the executed program sequence as possible. This is done by handing over many of the virtual memory management responsibilities from the operating system to an architecturally isolated hardware black-box (VM black-box). We provide a detailed design for the VM blackbox and some microarchitecture level simulation derived performance data. We also describe a compiler directed prefetch scheme that uses both instruction and data prefetches to obfuscate the address traces on the address bus between on-chip L2 cache and memory.

I. Introduction

Digital rights management (DRM) refers to the process of protecting the intellectual property (IP) embedded in digital domain. This paper addresses digital rights management for software. The DRM violations for software can result in either financial losses for the software developers or a loss of competitive advantage in a critical domain such as defense (for example, when an aircraft lost in hostile territory contains embedded systems with critical IP). Software piracy alone accounted for $13 billion annual loss [1] to the software industry in 2002.

Software digital rights management traditionally consists of watermarking, obfuscation, and tamper-resistance. All of these tasks are made difficult due to the power of adversary. The traditional security techniques assume the threat to be external. The system itself is not an adversary. This provides a "safe haven" or "sanctuary" for many security solutions. However, in DRM domain, the operating system itself is not trustworthy. On the contrary, the operating system constitutes the primary and formidable adversary. This is where the core of the difficulty arises. In essence, every security technique needs to hide some secrets from its adversary. Where can a DRM technique hide such "secrets"? The current system design exposes every system component to the operating system (OS). The access rights model is hierarchical with an entity at a higher level endowed with the access rights of all its children and some more. In such a model, the operating system sits at the root node. In such a scenario, there is no "sanctuary" left for the DRM technique's secrets that is not accessible to the operating system. Hence, we believe that the primary distinction between the traditional security and DRM is that the focus shifts from the problem of "protecting the operating system from adversary" to the problem of "protecting the program from the operating system". A key thesis of this paper is: *"All of the component technologies of digital rights management: specifically watermarking, static and dynamic behavior obfuscation, and tamper resistance can be made significantly more robust if they have a mechanism to hide a secret from the operating system."*.

Any software-only solution to the "secret-hiding" from OS seems to be inadequate. It leads to the classical meta-level inconsistencies encountered in classical software verification derived from Gödel's incompleteness theorem. In the end, in most scenarios, it reduces to the problem of "*last mile*" wherein only if some small kernel of values could be isolated from the operating system (as an axiom), the entire schema can be shown to work! At this point, it is worth noting that even in the Microsoft's next generation secure computing base (NGSCB) [14], the process isolation from operating system under a less severe adversary model is performed with hardware help. The NGSCB's goal is to protect the process from the operating system corrupted by external attacks by maintaining a parallel operating system look-alike called "nexus". The nexus in turn relies upon a Security Support Component (SSC), a hardware component, for performing cryptographic operations and for securely storing cryptographic keys! The trusted computing group consisting of AMD, HP, IBM, and Intel among many others is expected to release trusted platform module (TPM) [2], to provide the SSC hardware support. Our own experience with static and dynamic obfuscation and tamper resistance has convinced us that without hardware support not very robust guarantees can be made for any of these technologies.

We explore architecture level support for the one DRM component, dynamic address obfuscation. The resulting architecture collectively is called DRMAr (DRM Architecture).

A. Dynamic Address Obfuscation

Static obfuscation is designed to make it hard to infer anything more than what is already observable through a program's black-box input-output relationships through white-box examination. Even perfect static obfuscation (the one that leaks zero information statically), however, is not enough to prevent dynamic observation attacks from gaining enough information about the program to reverse engineer it. The dynamic execution, however, still generates the correct program sequence in the address traces at the memory bus. An obfuscation of this address sequence refers to the process of decreasing correlation between the true address sequence and the obfuscated address sequence as much as possible. Goldreich & Ostrovsky [11] were the first ones to handle this problem in the context of an oblivious RAM model. The model is motivated by and relies upon software-hardware (SH-) package. Specifically, it seems to assume a lightweight, embedded operating system which is not necessarily controlled by the owner of the SH-package. This is a scenario applicable to smart cards for instance. It however does not apply to a typical Linux box. The operating system sits smack in the middle of the two interactive Turing machines (ITMs), memory and CPU ITMs, defined in their setup. Operating system is the broker for virtual address translation between the communication tape of CPU ITM and the communication tape of the memory ITM. Unfortunately, this allows for many opportunities for the operating system virtual memory manager (in Linux for instance) to be rigged to collect significant amount of information about the program sequencing. This situation will arise in any interesting computing system with virtual memory support. This is the weakness we address in this paper.

Our solutions to this problem: virtual memory black-box uses reconfigurable components. The reconfigurable components provide the pseudorandom functions for dispersion.

B. *Paper Organization*

The rest of the paper is organized as follows. The related work is presented in Section II. The dynamic address obfuscation support through a virtual memory black-box is described in Section III. The prefetch based dynamic address obfuscation is presented in Section IV. We conclude the paper in Section V.

II. Related Work

The most closely related research to dynamic address obfuscation is Goldreich & Ostrovsky's work on oblivious RAMs [11]. Their work however is predicated upon a lightweight operating system embedded inside a software-hardware package. The proposed work targets the virtual memory address system of a general purpose, multiuser operating system such as Linux.

There have been several architecture level proposals for secure processors. Most notable is XOM [13], execute only memory. It provides a mechanism to isolate the processes from each other including the operating system processes. Inter-process communication can be supported through a null process broker. All the memory traffic is encrypted. Data is hashed with the corresponding address for memory integrity. Note, however, that address sequences are plainly visible to an observer. The primary design objective for XOM is not digital rights management. Static obfuscation in XOM is available only in as much as the contents of memory are encrypted. Their sequencing however is visible.

AEGIS [17] provides architecture level support for tamper-evident and tamper-resistant processing. Both software and physical tampering are included in the attack space. They too however do not consider the dynamic address obfuscation issue deferring it to oblivious RAM solution of Goldreich & Ostrovsky [11]. AEGIS also supports compartmentalized execution of the processes similar to XOM. It goes a bit further in memory integrity verification by incorporating Merkle hash tree verification [9]. Once again, the static obfuscation is only a byproduct of memory contents encryption. The sequencing information is still visible.

A recently proposed architecture, reliability and security engine (RSE) [15], provides hardware support for error detection and some of the security issues. Specifically, the security aspects are handled through a memory layout randomization module to subvert attacks such as buffer overflow attacks. Other modules handle fault tolerance aspects.

III. Dynamic Address Obfuscation: Virtual Memory Black box

The virtual memory black-box (VM blackbox) receives every virtual address within the processor microarchitecture much the same way as the translation lookaside buffer (TLB). In fact, the VM blackbox is ideally positioned in the processor pipeline at the current position of the TLB. However, the VM blackbox needs to do more than just virtual address translation of the TLB. For instance, consider program loading. Program loading is currently performed by the loader, a kernel component. With an untrustworthy operating system, this responsibility needs to be moved to the VM blackbox. However, there are subtleties about the granularity of load modules as we clarify later. Table III outlines all the tasks assigned to the VM blackbox which are traditionally performed by an operating system.

A. *Process Initialization*

Many actions are taken on a process initialization. A process descriptor (called a task descriptor in Linux) is filled in with several values at this stage. These values have to do with the many access attributes and group ID attributes. The process stack, part of the initialization, also holds the environment attributes and the initialization & termination function pointers. In a traditional OS, all these activities are performed by the OS. However, in DRMAr system, some of the actions will be performed by the VM blackbox. The most prominent attribute selected by the VM blackbox is the function that disperses the virtual pages. This function is implemented with a specialized reconfigurable (lookup table/LUT based logic) unit. The function is described by a configuration (descriptor) for this reconfigurable architecture. This function descriptor is kept with the task/process descriptor in an encrypted form.

Traditional OS task	VM Blackbox task	DRMAr OS task
process initialization (initial physical page assignment)	partial process initialization (initial physical page assignment) (process virtual page dispersion function selection)	partial process initialization
process scheduling		process scheduling
dynamic linking (load time relocation)	partial dynamic linking (load time relocation)	partial dynamic linking
virtual address translation (page table maintenance) (page fault handling) (page replacement)	virtual address translation (page table maintenance) (page fault handling) (page replacement) (page dispersion function assignment) (page block substitution key assignment)	

TABLE I
PROPOSED RESPONSIBILITY ASSIGNMENT FOR THE VM BLACK-BOX

The VM blackbox contains the private part of a public/private key pair. This pair can be renewed at bootup time or even periodically. Note that the public key is not really used by any other system component to send secure messages to the VM blackbox. VM blackbox uses it to encrypt any state before storing it in memory, and to decrypt any state loaded from memory.

We can assume that there is sufficient space within the VM blackbox to cache the VM-attributes of a small working set of process descriptors. Any spills into memory are stored in encrypted form.

B. Process Scheduling

The process scheduling is performed by the OS in the existing systems, and we have left it that way in DRMAr. All the existing priority levels can be maintained. The only malicious action that can be taken by the OS through scheduling would be process starvation. However, in the digital rights management context, process starvation does not benefit the adversary at all. It does not appear to aid the information leak for reverse engineering of the process.

C. Dynamic Linking

Dynamic linking or shared objects in Linux allows for the executable code to be linked to a program on a demand driven basis. A procedure (or a collection of procedures) is linked only if it is called. The executable (in ELF, executable and linking format) passes control to an interpreter, the dynamic linker, which itself is a shared library in Linux. The executable contains a procedure linkage table (PLT) and a global offset table (GOT). PLT keeps information about the relative locations of the procedures, whereas GOT tracks data locations. When invoked, the dynamic linker, garners information from the linked object's .dynsym (dynamic symbol table), .dynstr (dynamic string name table), and .hash ((a hash table to allow linker to quickly access symbols) sections. Another section, .dynamic, contains information about other files the linker needs. The dynamic linker is traditionally an OS component, Gnu ld.so in Linux. The main (well-known) threats in dynamic linking from a hostile dynamic linker are symbol or object hijacking. The auxiliary vector can be tampered with a wrong environment.

It is not apparent that any of these weaknesses provide an advantage to the adversary in a DRM environment. The dynamically linked objects are typically public libraries (and do not contain any intellectual property to be protected). They publish plenty of information about their symbols and dependencies in ELF to leak significant information on program structure. We have assumed that information leak from the linking object (and not necessarily from the linked object) is to be contained. In such a scenario, all the dynamic linking control can be left with the OS. The virtual to physical address mapping still resides with the VM blackbox preventing information leak from dynamic address sequences. One could even argue that the linked object dynamic address sequence revelation is not much of a threat in this model. However, just to be consistent, we will still leave the virtual to physical address mapping with the VM blackbox.

If we needed to protect the information about which shared objects are linked from the protected program, we will need to transfer significant amount of dynamic linker responsibilities to the VM blackbox. Moreover, the binary format such as ELF would have to be modified not to keep such relevant information in plaintext. This will lead into yet another problem. The dynamic linking sections of the ELF file for the protected linking program can be securely encrypted by the compiler used by the software producer. Which key should be used for this encryption? This leads into the same "last mile" issue faced by XOM and AEGIS. The initial handover of the program is done by the software vendor encrypting the program with respect to an advertised public key for the processor chip. Hence, the software is specialized on a per processor chip basis before distribution, which can be an undesirable characteristic. For the time being, we have chosen not to address this issue.

Similarly, if linked objects also needed to be protected, VM blackbox will need to undertake many of the dynamic linker activities. For the time being, DRMAr does not address the issue of migrating dynamic linking into the VM blackbox.

D. Virtual Address Translation

This is the heart of the VM blackbox responsibilities. This includes the virtual to physical address translation of TLB and much more. Program loading is one of these activities. We could let the OS interact with VM blackbox with a primitive such as `VM_ld A, size` which signifies loading of virtual addresses from the virtual address A to $A+size$. If we merely have the VM blackbox return the physical address mapping of the virtual addresses $[A : A + size]$, the OS can infer the entire mapping. Hence the actual storing of the program/data at these virtual addresses needs to be performed by the VM blackbox. Hence the `VM_ld file, A, size` can specify that the next `size` bytes from `file` be loaded at virtual addresses $[A : A + size]$. Note that the argument `file` is really a proxy for a system program with a pointer to the current location in the file so that VM blackbox can ask it to provide the next `size` bytes from this file.

If the `size` is a small value (say 32 B), the OS can observe the memory bus traffic and with some effort, reconstitute the virtual to physical address mapping. Hence, we need some type of "shelter" or "dummy" word storage of Goldreich & Ostrovsky [11] to build up the combinatorial choices of the mapping. We propose to do the load on the page granularity, *i.e.*, `VM_ld file, virtual_page` will load a virtual page into memory. Our assumption is that a typical page is about 64KB (if necessary such a restriction can be placed on the OS).

Let a virtual address A correspond to the block $\#B(A)$ in the virtual page number $V(A)$. The VM black box will randomly permute the block $B(A)$ into block $f_R(B(A))$ through the use of block address translation reconfigurable logic.

a) **Reconfigurable Block Address Translation Logic:** Note that the objective is to permute the blocks within a page. However, we wish the permutation to be different for each page. This is so that the adversary cannot gain information monotonically over time by observing the memory traffic. Reconfigurable logic provides a good platform for such "soft" random functions. Figure 1 shows the schema for the reconfigurable random permutation function logic. This schema shows a permutation of 12 bits. With 64KB page size and 16B block size, one needs to permute the 12 address bits corresponding to the block number within a page.

Note that $f_R(B(A))$ needs to be a bijective function: both injective and onto. Hence, we cannot choose an arbitrary FPGA like structure and populate it with a randomly selected configuration. This could result in non-bijective mappings. Each column in the schema of Figure 1 consists of 4-LUT (4-input lookup table). However, We wish to implement only reversible (conservative) gates [7], [3] within these LUTs. A conservative/reversible gate does not lose any information in going from its inputs to outputs. We should be able to tell the input values uniquely by observing the output bits of such a gate. Obviously, a 2-input AND gate is not reversible (since on output 0 we cannot tell whether it was input 00, 01 or 10). In general, a reversible gate needs to have as many inputs as outputs. Hence, we need 4x16 LUTs (truth tables on 4 inputs specifying 4 output bits simultaneously). For instance a 2-input gate with 2 inputs X and Y and 2 outputs X and $X \oplus Y$ is a reversible gate. The intuition behind using reversible gates is that since each reversible gate defines a bijection, a composition of these reversible gates will automatically result in larger bijections.

Both Fredkin [7] and Toffoli [18] have defined classes of reversible gates.

Definition 1 (Toffoli, 1980): Toffoli gate Toffoli$(n, n)(C, T)$ is defined over a support set $\{x_1, x_2, \ldots, x_n\}$ as follows. Let the control set $C = \{x_{i_1}, x_{i_2} \ldots, x_{i_k}\}$ and the target set $T = \{x_j\}$ be such that $C \cap T = \emptyset$. The mapping is given by Toffoli$(n, n)(C, T)[x_1, x_2, \ldots, x_n] = [x_1, x_2, \ldots, x_{j-1}, x_j \oplus (x_{i_1} x_{i_2} \ldots x_{i_k}), x_{j+1}, \ldots, x_n]$.

A Fredkin gate is defined similarly:

Definition 2 (Fredkin, Toffoli, 1982): Fredkin gate $Fredkin(n, n)(C, T)$ is defined over a support set $\{x_1, x_2, \ldots, x_n\}$ as follows. Let the control set $C = \{x_{i_1}, x_{i_2} \ldots, x_{i_k}\}$ and the target set $T = \{x_j, x_l\}$ be such that $C \cap T = \emptyset$. The mapping is given by $Fredkin(n, n)(C, T)[x_1, x_2, \ldots, x_n] = [x_1, x_2, \ldots, x_{j-1}, x_l, x_{j+1}, \ldots, x_{l-1}, x_j, \ldots, x_n]$ iff $x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_k} = 1$. In other words, target bits are switched iff the control bits are all 1.

For the time being, we use Toffoli(4,4) gates with 4-input bits and 4-output bits in Figure 1. However, we could easily replace them by Fredkin(4,4) gates. The domain of configurations mappable to each of these LUTs consists of selections of sets T and C such that $T \cap C = \emptyset$. For a support set of 4 variables, the number of unique reversible Toffoli functions is $3 * \begin{pmatrix} 4 \\ 1 \end{pmatrix} + 2 * \begin{pmatrix} 4 \\ 2 \end{pmatrix} + \begin{pmatrix} 4 \\ 3 \end{pmatrix}$. First term captures the control sets C of size 1, the second one of size 2 and the third one of size 3. We will ignore control sets

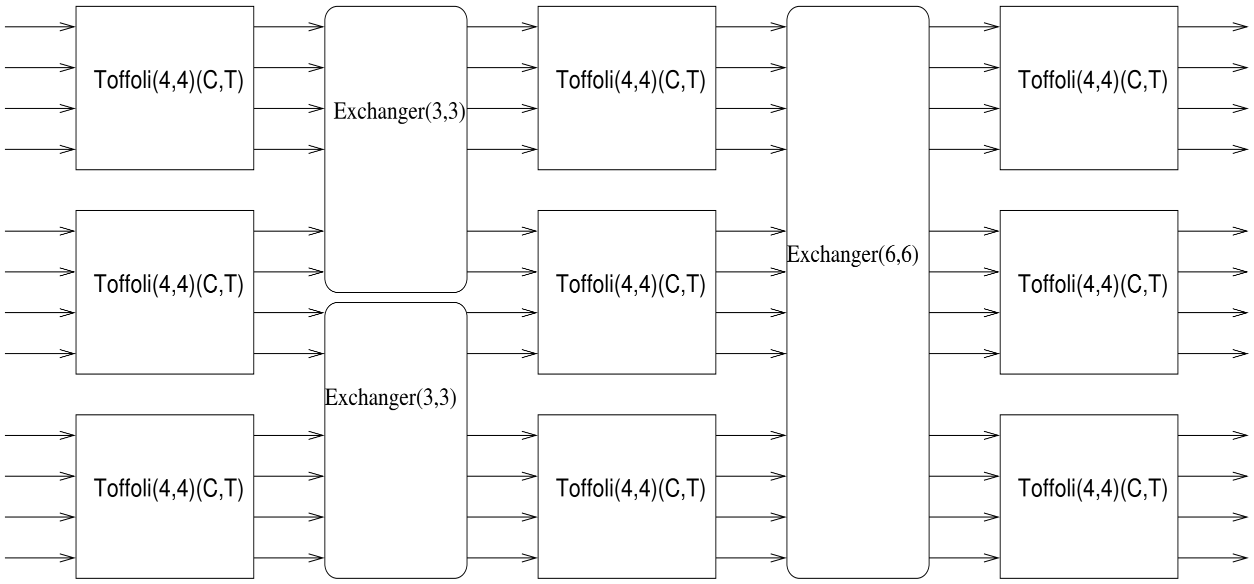

Fig. 1. Reconfigurable Block Address Translation Logic Schema

of size 1 in our discussion. Hence the number of reversible Toffoli functions derivable from size 2 & 3 control sets is $12 + 3 = 15$. Hence the number of address mapping functions derivable just from the reversible gate configuration population diversity is $(15)^9 \approx 2^{35}$. Obviously, there are several redundant configurations in this space. The exchanger blocks in between the columns provide bit shifting operation. A 3-bit exchanger block `exchanger(3,3)` has two sets of 3-bit groups, upper 3-bits and lower 3-bits. It either exchanges them (lower 3-bit group goes out on the upper 3-bits and the upper 3-bit group goes out on the lower 3-bits) or leaves them unchanged. One bit of configuration per exchanger suffices. The 6-bit exchanger similarly works on 6-bit groups. Since *exchange* is also a bijective operation, the composition of reversible (bijective) Toffoli gates and exchangers leads to bijective address mappings with a large amount of population diversity. We need to explore some other interesting routing structures that can easily guarantee bijections (without a large overhead in filtering out non-bijective routing configurations). Note that a typical FPGA routing matrix configuration will require extensive analysis to determine if a given routing configuration is bijective. That is why we did not consider it.

We could have chosen to implement the entire `Toffoli(12,12)` as a single reversible gate. However, it is impractical for the same reason 12-LUTs are impractical in reconfigurable computing world (requiring 2^{12} sized memories).

b) **Configuration Selection for the Reconfigurable Block Address Mapping:** How is the configuration of each 4-LUT selected randomly within the schema presented in Figure 1? Note that this mechanism needs to be efficient since it needs to be changed for each virtual page. A simple mechanism will be to store all the fifteen possible functions at

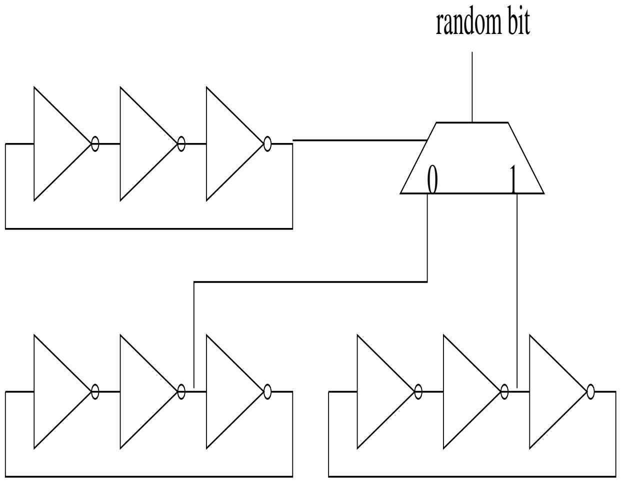

Fig. 3. Ring Oscillator Based Raw Random Bit Generator

each of the LUTs (similar to DPGA of DeHon [5]). In addition to the 4-bits coming in from the block address, each LUT will also see 4 more bits to select one of the random, reversible function (Figure 2). The bits $fs3, fs2, fs1, fs0$ select one of the 15 reversible functions $f15, \ldots, f4$, $f3$, $f2$, $f1$, $f0$. The selected truth table is used with the 4 input address bits $A3, A2, A1, A0$ giving as output the permuted address bits $PA3, PA2, PA1, PA0$.

c) Random Selection of LUT Reversible Gate Configurations per Page: Note that each of the 9 LUTs in Figure 1 requires selection of 4 bits to specify one of the reversible functions. We also need 1 bit each for the exchanger blocks. These bits constitute the configuration for the virtual address translation block. For each new allocated, physical page these bits need to be selected randomly. What would be a reasonable mechanism to select these bits which is not architecturally

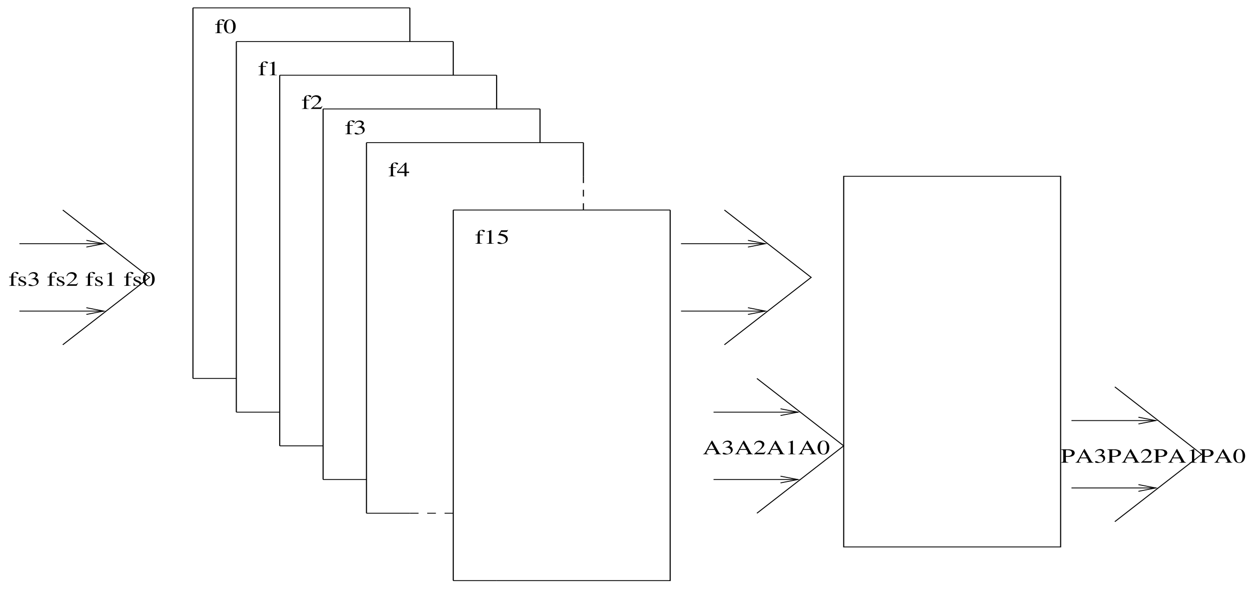

Fig. 2. Configuration Selection for each LUT

observable by the OS? Due to architectural hiding requirement, the entire apparatus for the random bit generation needs to be completely internal to the VM blackbox. This rules out software based solutions unless we wish to incorporate a private computing engine within the VM blackbox or to support compartmentalized execution model of XOM or AEGIS. Many physical processes do show random behavior. These can be thermally induced, charge decay induced, among many other existing proposals in the literature. We develop a random bit generation block based on the uncertainty of the timing in a ring-oscillator. Ring oscillators have been used in various ways to provide raw random bits [8], [19]. Our approach is slightly different than the existing ones, better suited for the VM blackbox.

Figure 3 shows the block diagram for a raw random bit generator. A three-stage ring oscillator is the basic building block for this scheme. We wish to exploit randomization in the asynchrony of the oscillator timing with the processor clock used to sample these random bits. An additional level of randomness is introduced by multiplexing two bits from two distinct ring oscillators by the output of another ring oscillator. Note that these three ring oscillators need not be physically adjacent. In fact, better randomization will be achieved due to the silicon fabrication process parameter variation by dispersing these ring oscillators in different parts of the chip. The chip temperature induced variations also will be more widely distributed through wider geographical distribution of these oscillators. Finally, the entropy of the raw bit stream could be increased further by choosing a larger number of seed ring oscillators and multiplexors.

Note that ideally we will like a provable randomness property from these bit strings as in case of pseudorandom functions [10]. Although these schemes are cryptologically secure, they are extremely expensive to implement (in addition to requiring a 1-way function implementation, significant amount of state storage, and a complex algorithm is also needed). Hence, we ruled these schemes out for a hardware implementation. Denker [6] provides a provably high entropy bit generator that works on a raw stream of random bits (potentially with low entropy density) to transform it into a high entropy density bit stream. The software driver for the entropy density enhancement is still fairly expensive, but can be considered on top of the ring oscillator generated bit streams if need be.

d) **Configuration Size and Performance of Reconfigurable Logic:** The total configuration size per virtual page then is 4-bits for each of the 9 LUTs and 3 bits for all the exchangers, which equals 39 bits. This indeed is a very reasonable and manageable configuration size. The performance of the reconfigurable address mapping logic consists of three cascaded 256×4 SRAM lookups and three multiplexors. Equating the access time of a dual-ported 32×32 SRAM array (register file) to half a clock cycle (in a typical processor), this delay is reasonably bounded by 2 cycles. The address translation logic could easily be pipelined for an initiation interval of possibly half a cycle and latency close to two cycles. Multiple address translation pipelines could be utilized to support multiple load/stores in an instruction-level-parallel (ILP) processor.

We measured the performance impact of multiple cycle latency TLBs (both ITLB and DTLB) on the execution time of SPEC2000 int benchmarks [16] though SimpleScalar simulations [4]. Since, the VM blackbox is placed in the current location of TLBs in the superscalar pipeline; the variable TLB latencies actually model the variable latencies of the VM blackbox. We simulated each benchmark with access latencies of 2, 3, and 4 cycles. The performance impact is shown in Figure 4. The average performance penalty with

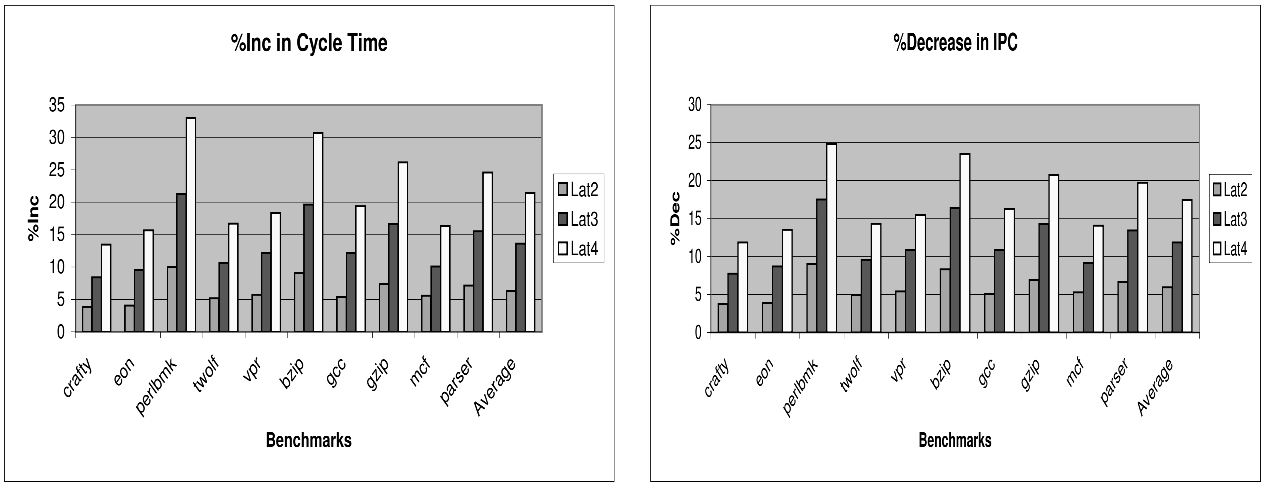

Fig. 4. Performance Penalty of Several TLB Access Times on SPEC2000 int Benchmarks

Parameter	Value
Processor	Alpha out of order 4 way issue
ITLB	256KB, 4 way, 4096B line
DTLB	512KB, 4 way, 4096B line
L1-DCache	16KB, 4 way, 32B line
L1-ICache	16KB, 1 way, 32B line
L2-Unified	256KB, 4 way, 64B line

TABLE II

SIMPLESCALAR SIMULATION PARAMETERS

the estimated 2 cycle latency is approximately 5%. All the SimpleScalar simulations (including the cache miss rate data in the following) were generated with the processor configuration shown in Table II.

e) **Cache Miss Penalty Impact of Block Address Translation:** One potential pitfall of block address translation within the VM blackbox is that it may interfere with the working sets' placement within caches and result in increased miss rates. We simulated these address translation based obfuscation methods in SimpleScalar [4] (version 3, Alpha Instruction Set) with SPEC2000 Integer benchmarks. Table III lists the miss rate of of Level-1 instruction cache. Table IV lists the miss rates of instruction TLB, which are reflective of miss rates of page table entries within the VM blackbox. Finally, Table V reports the simulated instructions per cycle (IPC). Each column in these tables corresponds to the number of address bits participating in the permutation (or the extent of dispersion). The larger the dispersion more adversely the miss rates can be impacted. For all these tables, the (leftmost) column with number of obfuscated bits equal to 0 corresponds to the situation where VM blackbox is not dispersing the page blocks at all. Note that none of the miss rate changes from block dispersion are noticeable. The only observable effect is in the IPC for `vpr` that goes down from 1.179 to 1.178 (negligible) with 13-bit dispersion. The ITLB miss rates reported by SimpleScalar [4] are not an accurate reflection of reality since it does not model the virtual memory as well.

L1 instruction cache Miss Rate							
# Obf. Bits	0	11	13	15	17	19	20
bzip	0.000	0.000	0.000	0.000	0.000	0.000	0.000
crafty	0.043	0.043	0.043	0.043	0.043	0.043	0.043
eon	0.022	0.022	0.022	0.022	0.022	0.022	0.022
gcc	0.026	0.026	0.026	0.026	0.026	0.026	0.026
gzip	0.009	0.009	0.009	0.009	0.009	0.009	0.009
mcf	0.009	0.009	0.009	0.009	0.009	0.009	0.009
parser	0.001	0.001	0.001	0.001	0.001	0.001	0.001
perlbmk	0.012	0.012	0.012	0.012	0.012	0.012	0.012
twolf	0.016	0.016	0.016	0.016	0.016	0.016	0.016
vpr	0.004	0.004	0.004	0.004	0.004	0.004	0.004

TABLE III

L1 I-CACHE MISS RATES AS FUNCTION OF NUMBER OF PERMUTED ADDRESS BITS

ITLB Miss Rate							
# Obf. Bits	0	11	13	15	17	19	20
bzip	0.000	0.000	0.000	0.000	0.000	0.000	0.000
crafty	0.000	0.000	0.000	0.000	0.000	0.000	0.000
eon	0.000	0.000	0.000	0.000	0.000	0.000	0.000
gcc	0.001	0.001	0.001	0.001	0.001	0.001	0.001
gzip	0.000	0.000	0.000	0.000	0.000	0.000	0.000
mcf	0.000	0.000	0.000	0.000	0.000	0.000	0.000
parser	0.000	0.000	0.000	0.000	0.000	0.000	0.000
perlbmk	0.000	0.000	0.000	0.000	0.000	0.000	0.000
twolf	0.000	0.000	0.000	0.000	0.000	0.000	0.000
vpr	0.000	0.000	0.000	0.000	0.000	0.000	0.000

TABLE IV

ITLB MISS RATES AS FUNCTION OF NUMBER OF PERMUTED ADDRESS BITS

Note however that the ITLB miss rates are indicative of miss rates in the cached page tables within the VM blackbox. What the data in Table IV does indicate is that the miss rates of the cached page tables within VM blackbox are not likely to go up with block dispersion.

f) **Overall Schema for VM Blackbox:** Figure 5 shows the overall schema for the VM blackbox. The TLB (if split then ITLB and DTLB) are part of the VM blackbox. The

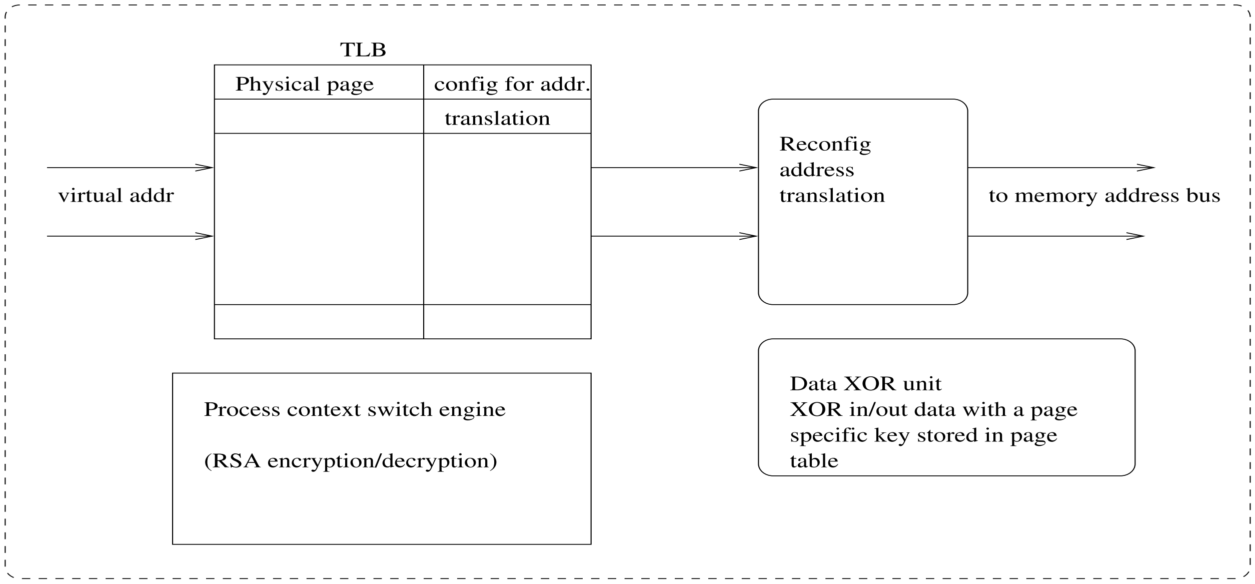
TLB
Physical page
config for addr. translation
virtual addr
Reconfig address translation
to memory address bus
Process context switch engine
(RSA encryption/decryption)
Data XOR unit
XOR in/out data with a page specific key stored in page table

Fig. 5. Overall Schema for the VM Blackbox

	Simulated IPC						
# Obf. Bits	0	11	13	15	17	19	20
bzip	1.842	1.842	1.842	1.842	1.842	1.842	1.842
crafty	1.242	1.242	1.242	1.242	1.242	1.242	1.242
eon	1.423	1.423	1.423	1.423	1.423	1.423	1.423
gcc	1.254	1.256	1.252	1.252	1.252	1.252	1.252
gzip	1.746	1.746	1.746	1.746	1.746	1.746	1.746
mcf	1.025	1.025	1.025	1.025	1.025	1.025	1.025
parser	1.508	1.508	1.508	1.508	1.508	1.508	1.508
perlbmk	1.442	1.442	1.442	1.442	1.442	1.442	1.442
twolf	1.123	1.123	1.123	1.124	1.124	1.124	1.124
vpr	1.179	1.179	1.178	1.178	1.178	1.178	1.178

TABLE V
IPC AS A FUNCTION OF NUMBER OF PERMUTED ADDRESS BITS

processor is forced to broker all the virtual addresses through the VM blackbox. All the page tables are now VM blackbox's responsibility. It performs the TLB management (caching and restoring). All the page table information going into memory is encrypted and is correspondingly decrypted on the way in to VM blackbox. The swap space is also maintained by the VM blackbox. The contents of swap space are encrypted.

There are two degrees of capability the VM blackbox can be implemented with. The physical pages can be allocated to a process at the process creation/activation by the kernel. This option simplifies the VM blackbox, and does not seem to take much away in terms of its capabilities. The other option would be to perform entire physical page management within the VM blackbox. However, that would require moving the entire process state into the VM blackbox along with process scheduling. At this point, we have decided to let the OS keep the physical page allocation responsibility. The VM blackbox needs to trust the OS about the process ID (since the OS tells the VM blackbox which process is scheduled leading the VM blackbox to load the corresponding tables). This does not appear to pose any problem in terms of the OS gaining any new information about the intra-page block ordering for the spoofed process ID.

The third data encryption block shown in Figure 5 can be used to obfuscate data (and/or instruction) based on XOR. A per page 32-bit random key can be chosen (which can be saved in the TLB entry). All the writes will XOR the data with this key before sending it to memory. All the reads will apply the XOR before forwarding the data to the CPU.

Note that we have not explicitly addresses interrupt handling in this paper. A mechanism similar to the one employed in XOM and AEGIS will work in this context as well. All the VM blackbox state (the cached page tables which include configuration bits for the reconfigurable address translation engine) needs to be saved by the VM blackbox controller in an encrypted form. The state restoration is also the VM blackbox controller responsibility. Any tampering of the saved state only leads to disabling of a program, which is not the goal in a DRM attack.

g) **Additional Address Trace Obfuscation:** Note that since the proposed address trace obfuscation scheme disperses/scatters the blocks within a page, its obfuscation is limited to log(page size) address bits. If additional obfuscation is desired, we will have either have to move up the address bits (into the virtual page bits). That, however, requires taking over more process management details from the OS than we care for. Hence, the other option is moving down address bits (towards cache block numbers). This will require some of the cache controller functions to be moved into the VM blackbox inaccessible from the OS. The simplest form will involve an cache block number translation (as long as we assume that the cache address bus is not architecturally visible to the OS, which it typically isn't). Although, we have not explored this option, it appears to be a straightforward extension of the presented scheme. The performance overhead will be incurred in the cache block number translation which already is in the critical path of cache timing.

IV. Prefetch Based Address Trace Obfuscation

In this section, we present another compiler directed, dynamic address obfuscation scheme that is orthogonal to the address dispersion of the VM blackbox. In a modern processor, L1 and L2 caches are getting very large, for instance Intel's latest processor family, Centrino [12] has a 1MB on-chip L2 cache. Our assumption is that the adversary can only observe the address bus from the processor chip to the memory subsystem. This provides yet another mechanism to obfuscate the dynamic address traces. The L2 cache already filters out a very large percentage (98%+) of the addresses generated between the CPU core and L1 caches, which itself has some obfuscation effect. However, if we could prefetch most (some) of the cache blocks needed in the near future in an obfuscated order, any information leaks between L2 cache and memory can be sealed even further. Most contemporary processors support compiler directed prefetching. We insert prefetch instructions at compile time with the explicit objective of obfuscating the L2 cache to memory address traces. The following is an outline of this prefetch algorithm. It operates on the profiled control flow graph of the program. Note that a node in the following description can be any desired granularity: basic block or hyperblock or function. Our current implementation uses function level granularity.

h) **Prefetch Allocation Algorithm:**

i) **Assumptions:**

- The graph is directed with each edge $e_{i,j}$ between nodes i and j having an instantiation probability $p_{i,j}$. The edge is directed from the parent node i to the child node j.
- There is a limit on the maximum number of cache blocks that could be prefetched at any point in time - P_{max}. This is derived from practical considerations. Higher this limit, potentially better obfuscation would be, however at the cost of resulting cache pollution leading to worse performance.
- Each node i has an attribute associated with it - the number of cache blocks it spans (accesses) n_i.
- There is a constant k_p such that, each prefetch instruction requires k_p bytes.
- One prefetch instruction is required to prefetch a cache block.
- Each cache block is of size $C_b$bytes.

j) **Objectives:**

- Annotate each node with the prefetch instructions.
- The prefetch instructions should be such that, every parent node should at least prefetch the prefetch instructions of all its child nodes.
- Then the remaining space (P_{max} - the bytes required to fill the prefetch instructions) should be filled with prefetch instructions to prefetch instructions from most probable path.

k) **Algorithm:**

```
1) For every node, construct the following list: the list contains the child nodes in the decreasing order of their cumulative probabilities ($p_i$), i.e, the probability to reach the node $i$ from the starting node. If a node $i$ is listed more than once, merge all the entries and add the cumulative probabilities.
2) Find the list of root nodes (nodes which do not have any parent)
3) For every root node, $makePrefetch(rootNode_i)$.
Function: $makePrefetch(curr_{node})$
1) Let $bytes_{max} = P_{max} * k_b$
2) While(1) till step 9
3) Let $avail_{bytes} = bytes_{max}$
4) While $avail_{bytes} > 0$ do steps 7,8
5) Create the prefetch block for the $curr_{node}$ by browsing its cumulative probability list of nodes. A node $i$ can be included only if all its cache blocks can fit in i.e., $n_i$ $<=bytes_{max}$.
6) For every node included, decrease the available $avail_{bytes} = avail_{bytes} - n_i$
7) If all the immediate child nodes (child nodes which have an edge with the $curr_{node}$) of $curr_{node}$ are covered, this is the prefetch block for the $curr_{node}$, mark all the nodes which are prefetched by this node with this node's ID. Exit the function.
8) Else, find the prefetch block for the least probable child node $j$ , i.e, call $makePrefetch(j)$.
9) Add instructions to prefetch the prefetch blocks of $j$ in $curr_{node}$ i.e, $bytes_{max} = bytes_{max} - prefetch_j$
```

l) **Experimental Setup:** This scheme has been implemented with `gcc`. We use `gprof` to collect the profiling information. However, at this point, we do not have any performance overhead data to report. The experimental setup to select a statistically valid set of inputs for profiling is being established. We should have the results on the cache pollution overhead, prefetch accuracy, overall IPC degradation, and address trace variance with respect to the original address trace in the near future.

V. Conclusions

Protection of intellectual property on a hostile computing node is emerging as one of the most difficult problems in system design. The adversary is much more powerful than the traditional security models. This brings about the need to hide "secrets" from the operating system. We have presented an architecture for providing support for one of the vital components of obfuscation, *dynamic address obfuscation*. We argue that a large part of virtual memory management responsibilities must be moved from the OS to a new, private & self-contained unit, virtual memory blackbox (VM blackbox). VM blackbox disperses the physical addresses to create obfuscation. We developed an interface for the OS, VM blackbox interaction which allows for the VM blackbox to maintain and hide its private state from the OS. We describe all the internal components of the proposed VM blackbox. We also quantify the performance overhead of such address dispersions. Another compiler directed scheme for dynamic address obfuscation that utilizes large L2 cache sizes to hide the address order further was also presented. The instructions and data needed in the near future are prefetched into L2 cache in an obfuscated order compared to the actual program order. This scheme has been implemented within gcc.

Note that this architecture does not address the threats posed by physical tampering & observation such as power profiling.

REFERENCES

[1] Business Software Alliance. 8th annual business software alliance global software piracy study: Trends in software piracy 1994-2002, 2003. http://global.bsa.org/globalstudy/2003_GSPS.pdf.

[2] Trusted Computing Platform Alliance. Trusted platform module, 2003. http://www.trustedcomputing.org/.

[3] R. Bennett and R. Landauer. Fundamental Physical Limits of Computation. *Scientific American*, pages 48–58, July 1985.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin, Madison, 1996.

[5] André DeHon, "DPGA-coupled microprocessor: Commodity ICs for the early 21st century", In D. A. Buell and K. L. Pocek, editors, Proc. of IEEE workshop on FPGAs for Custom Computing Machines, pp. 31-39, Apr. 1994.

[6] J. S. Denker. High-entropy symbol generator, 2003. http://www.av8n.com/turbid/.

[7] E. Fredkin and T. Toffoli. Conservative Logic. *International Journal of Theoretical Physics*, 21(3/4), April 1982.

[8] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9 ACM Conference on Computer and Communications Security*, 2002.

[9] B. Gassend, G. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and merkle trees for efficient memory integrity verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, 2003.

[10] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986.

[11] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[12] URL: *http://www.intel.com*

[13] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[14] Microsoft. Next-generation secure computing base, 2003. http://www.microsoft.com/ngscb.

[15] Nithin M. Nakka, Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. An architectural framework for providing reliability and security support. In *Proceedings of the International Conference on Dependable Systems and Networks, DSN 2004*, June 2004. to appear.

[16] SPEC Benchmarks URL: *http://www.specbench.org/osg/cpu2000/*

[17] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17 Int'l Conference on Supercomputing*, pages 160–171, 2003.

[18] T. Toffoli. Reversible Computing. Technical Report MIT/LCS/TM151/1980, MIT Laboratory for Computer Science, 1980.

[19] VIA Technology. Evaluation summary: Via c3n nehemiah random number generator, 2003. http://www.via.com.tw/en/viac3/via_c3_padlock_evaluation_summary.pdf.