Detecting Code Alteration by Creating a Temporary Memory Bottleneck

Ryan W. Gardner, Sujata Garera, and Aviel D. Rubin

***Abstract*—We develop a new technique whereby a poll worker can determine whether the software executing on electronic voting machines on election day has been altered from its factory version. Our generalized approach allows a human, using a known challenge–response pair, to detect attacks that involve modification or replacement of software on a computer based on the time it takes the computer to provide a correct response to a challenge. We exploit the large difference between main memory access times and cache memory access or CPU clock cycle times to significantly increase the time required to compute the right response when the software has been changed.**

***Index Terms*—Computer security, computer viruses, software protection, software verification and validation.**

I. Introduction

CONSIDER a poll worker preparing for an election where votes are recorded and tabulated electronically. To date, there is no known mechanism that enables the poll worker to verify that the software responsible for processing these votes is the same, untampered code that was certified for the election. Rather, despite repeated studies demonstrating the ability to install covert, malicious code on voting machines [1]–[3], poll workers and voters must simply *hope* that votes are counted correctly by legitimate software.

While significant research in trusted computing has provided several methods for establishing the authenticity of software running on remote machines, establishing the integrity of software running locally is a different problem. Existing hardware-based solutions for establishing trust [4]–[11], such as those using trusted platform modules (TPMs), do not apply well in local settings because they require a separate computational device to verify the attestations produced by the trusted hardware (generally digital signatures). This is relevant in voting, for example, where it is desirable to keep the machines completely isolated to reduce the possibility of exploitation. More importantly, even if direct, trusted access to a separate, verifying machine were available, current hardware solutions require placing trust in that machine to provide reliable responses. Hence, we are back at the original problem of establishing that software is authentic. We consider an approach for terminating this chain of trust by using humans as a critical part of the attestation process.

Manuscript received February 15, 2008; revised August 26, 2009. First published September 29, 2009; current version published November 18, 2009. This work was supported by the National Science Foundation Grant CNS-0524252. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Poorvi L. Vora.

The authors are with the Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218 USA (e-mail: ryan@cs.jhu.edu; sgarera@cs.jhu.edu; rubin@jhu.edu).

Digital Object Identifier 10.1109/TIFS.2009.2033231

Recently, Franklin *et al.* introduced an approach by which a human can personally verify the software running on a local machine [12]. Human-verifiable attestations eliminate the need for trust in a separate verification device by using people in the chain of trust. Because these people can have offsetting interests, like members of opposing political parties, and can verify the software independently, they may be more trustworthy than black-box machines. This class of attestations allows someone to issue a challenge to a machine and determine whether or not its software is authentic based on the time it takes to compute a correct response. The technique for obtaining these responses, however, fundamentally depends on the existence of a software-based attestation primitive that yields humanly detectable increases in run-time under attempts to attest unauthentic code.

We present a new approach to software-based attestation primitives [12]–[15] that may be applicable to the design of systems where humans can verify that software has not been altered. The primitive we design, Endor, takes an unpredictable challenge as input and computes a checksum over a specific portion of memory and the state of the executing processor in a fixed amount of time. When this memory or processor state is altered, Endor yields an incorrect checksum with overwhelming probability. Furthermore, using code other than Endor to compute a correct checksum over the altered memory or CPU state results in a significant increase in the time to compute the checksum. We refer to this act of computing a correct checksum over altered memory or processor state as *forging* a checksum. On our four test processors, under conservative assumptions, forging a checksum requires at least 50% more time than computing a checksum with Endor. Because a human, like a poll worker with a stop watch, can distinguish a 15-s (forgery) run-time from a 10-s (legitimate) run-time, for example, this provides a realistic time constraint. We implement Endor for the 32-bit x86 due to the architecture's popularity and also because of its high complexity, which makes the design task more difficult but easier to adapt.

Our approach is based on the fact that main memory accesses are significantly slower than cached memory accesses or instruction executions. Benchmarks in previous work and on our test processors show that memory read times are at least 50 times greater than the time of a CPU clock cycle on the same machine (and as much as 500 times greater on some systems) [16]. Endor exploits this fact by requiring memory-bound work of one method for forging a checksum. Attempts to forge a checksum require an adversary to make a significantly increased number of accesses to main memory, to simulate Endor via some form of interpretive execution, or to use frequent hardware breakpoints. We require known methods of forgery to perform one of these actions partly by utilizing a new technique to create a memory bottleneck. In particular, Endor regularly ac-

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

cesses a portion of memory the size of the CPU's highest-level cache to fully utilize the cache. At the same time, it pseudo-randomly modifies a large portion of its own code during the calculation of the checksum.

The primary focus of this paper is the design and analysis of the attestation function, Endor. Before we describe Endor, we provide some background on software-based attestation and our threat model. Then, we outline properties of Endor that enable it to meet its goals. With these properties in mind, we explain how Endor is constructed. As we examine one property in particular, we describe how Endor's structure limits the adversary to three general methods of forging a checksum. We also explain how it takes advantage of the large delay caused by an access to main memory. Finally, we implement small, simple portions of code that are necessary for each of the three methods that an adversary could use to forge a checksum.[1] We run this code on processors of several micro-architectures to approximate a lower bound on the time required to compute a correct checksum when memory or processor state has been altered. This time is at least 50% greater than the run-time of Endor even in the worst case, potentially enabling human verification of software tampering.

II. Related Work

Research in trusted computing can be broadly classified into hardware-based techniques and software-based techniques. Existing hardware solutions require the use of a separate, verifying device that must be *trusted* [5]–[11], [17], [18], or they require trust in some small portion of software [19]. These solutions do not directly apply well to attesting software that executes on sensitive, stand-alone systems, like voting machines, since a trustable verification device may not be available.

The idea of software-based attestation is introduced and examined by Kennell and Jamieson with Genuinity [13]. Genuinity relies on frequent inspection of CPU performance counters to detect the execution of altered software. It is later shown to succumb to an attack by Shankar *et al.* [20] who are able to write code that does not affect these counters while computing portions of a forged checksum. Moreover, as argued by the researchers [20], performance counters are typically inconsistent on modern processors[2] rendering Genuinity unusable with those CPUs. Seshadri *et al.* design SWATT for embedded devices [14], and it requires knowledge of the *entire* state of memory on the machine it attests. While it forces a 13% run-time overhead from forgeries on an Atmel ATMega163L microcontroller, that overhead is likely to be almost negligible on standard systems with dedicated CPUs, which would have to make many slow accesses to main memory during normal execution.

Pioneer, by Seshadri *et al.* [15], is implemented for the 64-bit x86 Pentium 4 Xeon. It is susceptible to an attack that redirects execution using a hardware breakpoint [21]. Computation of a forged checksum on Pioneer also results in a 1.23% overhead in execution time, so its code must run for over 5 min to produce only a 4-s increase in run-time. Lastly, Pioneer's security relies on perfect, instruction-level optimality of the checksum function, which mandates primitive revisions for every minor processor version. When run on an Intel Core 2 processor, checksums can be forged with a run-time overhead of 0.16% (an approximate increase of 3 s over a 31-min total execution) [22]. PRISM [12], by Franklin *et al.*, is primarily a study of human verifiability but also presents an overview of an attestation function for the Intel XScale-PXA255 processor. The function utilizes a technique that relies on consistent performance counters, the ability to conduct program counter (PC)-relative memory reads,[3] and the ability to use the PC as a source register in arithmetic operations. The authors claim that the best *complete* forgery of a PRISM checksum produces execution time overheads of approximately 33% although the code for computing the forgery is necessarily very complex and it is difficult to analyze its optimality.

Further, Seshadri *et al.* utilize software-based attestation functions to ensure the integrity of software on sensor nodes with SCUBA [23]. They use a function (ICE) very similar to Pioneer although it again requires PC-relative memory reads. Forgery attacks against it yield overheads in run-time as low as 3%. In a later study, Seshadri *et al.* also use software-based attestation to establish shared secret keys between sensor network nodes without any prior secrets or side channels and without the assumption that nodes are uncompromised [24].

Recently, Gratzer and Naccache prove the existence of a software-based attestation function under a very simple, theoretical processor model using a quine [25]. This is later extended by Franklin and Tschantz to eliminate the need for a constant time processor reset [26].

III. Preliminaries

In this section, we provide relevant background before describing the design of Endor.

A. Terminology

We use several terms throughout the paper to discuss the design and analysis of our attestation function. We define these terms below.

General

- **Exe**—A small piece of software in which we want to establish authenticity (using Endor).
- **Endor**—Our attestation function. Takes an input challenge and computes a checksum over a predefined region of memory and the state of the executing processor.
- **correct checksum**—The checksum that Endor computes over the authentic Exe for a given challenge.
- **forge**—The act of computing a correct checksum when Exe is not authentic.
- **EndorTime**—The time in which Endor computes a correct checksum.
- **CompTime**—The observed time that an executing piece of code takes to compute a correct checksum.
- **MaxTime**—The maximum perceived CompTime for which a checksum is considered valid.

Memory spaces

[1]None of these portions of code are sufficient for a correct forgery. For demonstration purposes, we also include an implementation of a complete forgery.

[2]This is true of the Intel Pentium 4 model 0 and 3, the Core, and Core 2 CPUs. We have not tested on others.

[3]I.e., memory reads where the value of the PC register is used in the computation of the source memory address.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

- **ExeMem**—A predetermined, contiguous range of memory where Exe is expected to reside and execute.
- **EndorMem**—A predetermined, contiguous range of memory where Endor is expected to reside and execute.
- **AttMem**—A predetermined, contiguous range of memory that is attested by Endor. Consists of both ExeMem and EndorMem.

AUTHENTIC DATA/CODE

- **ExeDat**—The authentic, Exe code and data before Exe executes.
- **EndorDat**—The authentic, Endor code and data before Endor executes.
- **AttDat**—The authentic, attested code and data. Consists of ExeDat and EndorDat.

PROCESSOR STATE

- **ExeProc**—The state of the executing processor when Exe is run. This includes processor state variables such as interrupt vector definitions, hardware breakpoint settings, location of the stack, flag status, and others.
- **EndorProc**—The state of the executing processor when Endor is run with a given challenge.

NOTATION

- [**XMem**]—The contents of memory space XMem.

B. How Trust is Established

Endor takes a challenge and computes a checksum over a predetermined area of memory, AttMem, to establish that a piece of software, Exe, is authentic and resides in an expected location, ExeMem. In particular, it is designed so that the time required to compute a correct checksum significantly exceeds EndorTime when ExeMem does not contain the authentic data, ExeDat. In this section, we give a high-level overview of how Endor could be used to establish the authenticity of a given piece of software, Exe. We consider the example of attesting software running on voting machines on election day. A sequence of steps that could allow for such attestation is as follows:

1) Election officials choose an appropriate run-time for Endor, EndorTime. This is chosen to allow reliable human verifiability and can be based on usability studies. Increasing EndorTime can make human verification easier because it increases the total time overhead that results when code attempts to forge a checksum. For example, on the processors we use for testing, a 1-s EndorTime results in forgery computation times of at least 1.5 s, but a 10-s EndorTime results in forgery computation times of at least 15 s (Section VI). Endor is iterative as we discuss in Section V, and EndorTime can be specified by setting the number of iterations Endor executes.[4] (EndorTime is approximately the same for all challenges as we show in Section VI.)
2) Endor is integrated into voting machines' software in a way that it effectively bootstraps trust when the machines are used. We refer the reader to previous work [22] for details on effectively deploying software-based attestation functions to establish trust. This step determines and fixes the values of ExeMem, EndorMem, AttMem, ExeDat, EndorDat, AttDat, ExeProc, and EndorProc. For example, at this point, EndorMem is fixed to the exact location from which Endor is run.
3) Software reviewers receive the source code for all the voting machine software including Endor and the development environment. After reviewing and eventually certifying the software for the election, they publicize hashes of the correct voting-software binary images.
4) Prior to an election, each political party receives a voting machine as it would be used in an election. These parties verify that the software on the machine's storage matches the publicized hashes. Each party then runs Endor with multiple, randomly chosen challenges to determine corresponding checksum values and the time to compute them, EndorTime. Furthermore, the value of MaxTime is established based on the minimum time required to compute a forged checksum with the voting machine's processor. For example, MaxTime is set to $1.25 * \mathsf{EndorTime}$ if the voting machine uses one of the processors on which we tested.[5] Finally, each party provides pollworkers from its party with cards containing correct challenge–checksum pairs and the value of MaxTime. The pollworkers are instructed to maintain the secrecy of these cards.
5) Fig. 1 illustrates the basic attestation process on election day. After a voting machine is turned ON, Endor runs and prompts a poll worker for a challenge. A poll worker enters a challenge off of the card provided to him in Step 4 and starts a stopwatch.
6) Endor computes a checksum over the memory region AttMem. Upon completion, it displays the checksum to the pollworker.
7) The poll worker stops his stopwatch to obtain CompTime. The poll worker verifies that $\mathsf{CompTime} < \mathsf{MaxTime}$ and that the checksum matches the checksum on his card. If both conditions hold, the poll worker can be assured that the voting machine runs authentic software. Otherwise, he declares that it does not.
8) Steps 5 through 7 are repeated by a poll worker from each political party.

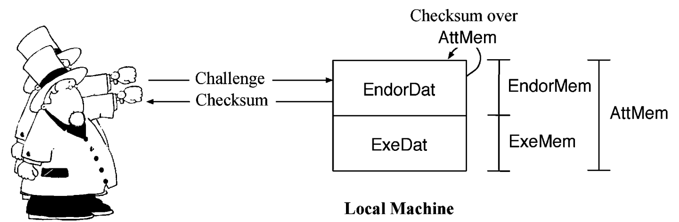

Fig. 1. Attestation process.

Although Exe is a generic piece of software, it (ExeDat) must be fairly small (less than approximately one fourth the size of the CPU cache or 512 KB on a system using a Core 2 processor,

[4]On a Core 2 processor, for example, $\mathsf{EndorTime} \approx \mathsf{iterations} * .1510\ \mu\text{s}$.

[5]In Section VI, we show that on our test processors the fastest method for forging a checksum has a CompTime that is at least 50% greater than EndorTime. We choose MaxTime as the midpoint between the true EndorTime and the minimum CompTime for forging a checksum. If other processors are used, the minimum CompTime for forging a checksum must be determined by running code for the types of forgery discussed in Section VI.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

for example) as we discuss in Section V. In practice, it may be useful to include a hash function as part of Exe. Then, once authenticity of the hash function code has been established, it may be used to attest other portions of the machine [22].

C. Threat Model

Endor's purpose is to establish the authenticity of Exe. It does not address Exe's quality or functionality. In particular, if Exe is malicious or easily exploitable even after security reviews (Step 3, Section III-B), establishing its authenticity may provide no benefit.

The specific methods for deploying Endor in real systems are largely application-dependent and are not the focus of this paper. We assume that Endor is deployed in a manner that prevents unattested code or unattested process data from being used after verification. Other work has discussed approaches for such deployment [22]. Furthermore, we do not discuss the details of input and output to and from Endor, including some critical timing sensitivities. Franklin *et al.* address most of this issue in their study of human verifiability [12].

We assume that at least one person verifying Endor's output behaves honestly, has accurate information, has a secret challenge and checksum, and has a reliable timing device. For example, in the voting scenario, we assume that at least one political party obtains a correct challenge–checksum pair with an accurate MaxTime. Further, the secrecy of that challenge and checksum is maintained within the party until a poll worker properly uses them and an accurate stopwatch for verification.

We assume that an adversary does not replace any hardware components of the system. Unlike previous techniques, ours enforces a large number of main memory accesses under a method of forging a checksum that we describe in Section V-F. Because memory access times have varied significantly less than CPU speeds over time [16], this property, in combination with large forgery overheads, may allow for broader hardware assumptions in the future [16], [27], [28], but we have not explored the possibility in this work. We also assume that the attesting machine has a single CPU core without hardware virtualization support. Voting machines, and potentially other specialized, security-critical systems such as ATMs, can easily be built with middle- or low-grade hardware meeting these requirements. Further, we assume that the adversary cannot offload computation to a faster machine. This is also easy to verify in the voting setting, for example, where machines are generally not networked and can be disconnected during verification if they are. Lastly, we assume an adversary does not trigger an external system management mode (SMM) interrupt.[6] This interrupt could be physically disabled through hardware design.

In our model, an adversary can, however, perform modifications to software. We assume that an adversary can alter any portion of the system software including the BIOS, bootloader, operating system, user applications, and Endor itself. Such modifications could occur in voting machines, for instance, while the machines are in storage or left at the polling places overnight the evening before an election. Alternatively, an insider at the manufacturer might install software other than the reviewed Exe (Step 3, Section III-B) during machine production. We discuss the precise means that we consider for successfully altering Exe in the following section.

[6]The SMM interrupt puts the processor in system management mode to execute code that is generally intended for power management and control of some system hardware. It could allow an adversary to execute unattested code without detection because the interrupt handler executes in an isolated memory space (SMRAM), which Endor does not attest.

D. Types of Attack

Consider a system in our threat model on which one intends to execute code ExeDat from memory space ExeMem with initial processor state ExeProc. We consider three ways that an adversary could execute code other than ExeDat:

1) **Altering memory**—An adversary could change instructions or data in ExeMem.
2) **Altering location of execution**—An adversary could execute code from memory other than ExeMem, from the start.
3) **Altering processor state**—An adversary could change the state of the processor. Within this type of attack, there are several possibilities including modifications of:
   a) hardware breakpoints;
   b) hardware watchpoints;
   c) paging;
   d) segmentation;
   e) interrupt vectors;
   f) flags;
   g) floating point unit (FPU);
   h) system call (`sysenter`) destination address.

We consider only these attacks in our analysis. In Section V-E, however, we also discuss why it is likely that Endor can be modified easily to account for new, critical processor state variables should they be discovered or added to CPUs. With these methods of attack in mind, we next outline properties of Endor that allow it to detect them.

IV. Security Properties

We outline key properties we aim to achieve with Endor.

1) *Flexible yet Consistent Run-Times*: The run-time of Endor should be consistent. This way users can be provided with values EndorTime and MaxTime that they can reliably distinguish when they observe executions of Endor. We should be able to easily adjust the run-time of Endor to adapt it to various applications and hardware.
2) *Strongly Ordered Execution*: The result of each computation and memory read executed by Endor should depend on the computations and memory reads that precede it [15]. Such dependence ensures that the result of each step of Endor cannot be obtained without first executing the steps before it (with some probability) and allows one to easily verify that Endor cannot be sped up by skipping or reordering operations. More specifically, to minimize the probability of an adversary predicting future states of Endor without execution, we want Endor's state at any time to be approximately pseudorandom with respect to the results of previous instructions.
3) *Altered Memory Detection*: If $[\mathsf{AttMem}] \neq \mathsf{AttDat}$, Endor should produce an incorrect checksum with high probability.
4) *Altered Location Detection*: If Endor is executed from memory other than EndorMem or computes a checksum

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

over memory other than AttMem, it should produce an incorrect checksum.[7]

5) *Altered Processor State Detection*: If the state of the processor is not equal to EndorProc, Endor should produce an incorrect checksum.
6) *Limited Methods of Forgery*: The ways by which an adversary can forge checksums should be as few as possible. The availability of only a limited number of forgery approaches makes an otherwise intractable analysis of forgery attacks possible. Furthermore, we want the execution overhead incurred by each method of forging a checksum to be as large as possible because greater increases in time can be distinguished more easily than small ones. In particular, Endor restricts an adversary forging a checksum to maintaining a second copy of a large portion of attested memory, simulating large portions of Endor using some form of interpretive execution, or using hardware breakpoints. This property requires careful consideration because it is difficult to achieve and crucial to analysis.
7) *Fast Function Iterations*: Each iteration of Endor should be designed to execute as fast as possible. Forging checksums requires more work than computing them with Endor. A faster Endor makes the overhead induced by that additional work proportionally larger.

V. Design

We now describe the design of Endor. We begin by briefly outlining the general concepts behind its construction. Then we present its high-level structure and execution process. Finally, we explain the design in further detail with respect to how Endor meets our target properties.

General Concepts

At a high level, there are four fundamental principles of Endor's design:

1) **Attest sufficiently and pseudorandomly**—Endor attests all known aspects of machine state (Section III-D) that could critically affect the execution of Exe. It does this primarily by pseudorandomly inspecting parts of the machine state, based on the given challenge, and adding the resulting values into a checksum (Sections V-C, V-D, and V-E). For example, Endor attests memory by reading from pseudorandom locations within AttMem [14] and adding the values read into the checksum (Section V-C). Such pseudorandom selection prevents a checksum-forging adversary from predicting when Endor might attest a part of the machine state that she has altered.
2) **Self-attest**—Like all software-based attestation functions [12]–[15], Endor attests its own memory, EndorMem, as it attests Exe's memory, ExeMem. This fundamental design aspect prevents an adversary from forging a checksum using a trivially modified version of Endor that simply addresses every alteration made to Exe. Every modification to Endor's code potentially adds another alteration that must be addressed.
3) **Use the entire cache**—Endor fully utilizes the CPU's entire, highest-level cache by pseudorandomly accessing a portion of memory, AttMem, that is the same size as the cache.[8] The prominent method for forging checksums with software-based attestation functions [12], [14], [15] has involved using a significant portion of additional memory (primarily in what has been referred to as a "memory copy attack" [12], [15]). By utilizing the entire cache, we force forging code that uses additional memory to start making accesses to main memory. This results in large run-times of the forging code because accessing main memory is very slow. (In particular, the time required to access main memory is at least 50 times greater than the time of a CPU clock cycle on the same system [16].)
4) **Self-modify**—Endor's code modifies itself. Specifically, the vast majority of the code that reads the attested memory and processor state is regularly modified during checksum computation. Because Endor attests its own code, each modification affects both what will be *executed* and what will be *attested*. Checksum forging code must somehow separate attested memory from what is executed because it must attest AttDat but execute malicious code in order to compute a correct checksum. Frequent self-modification, results in slower checksum forging code because the code must reflect every modification in both what it executes and what it attests (Sections V-F and VI).
Furthermore, the portion of code that Endor modifies is large, one half the size of the cache (one half the size of AttMem). This hinders forgery methods that keep two copies of the modified code (one for attesting and one for executing) because the second, large copy must be frequently accessed. As a result, the code uses significantly more memory than fits in the cache and it must make many, slow accesses to main-memory (Section V-F).

Next, we continue with the construction of Endor.

Function Overview

The basic memory layout around Endor is depicted in Fig. 2. Endor's execution is deterministic for a given challenge. As illustrated, Endor begins with initialization code, which executes first. This code reads the challenge from the user and sets up processor states including registers and breakpoints as well as some memory. It also relocates the stack and interrupt descriptor table (IDT) to within attested memory and disables interrupts to ensure constant Endor execution.

The initialization code is followed by the main code block, which is in turn followed by s modifiable code blocks, where $s = \mathsf{cache_size}/(2 * \mathsf{cache_line_size})$. Each of the modifiable code blocks has portions of code that are overwritten. The source of the instructions for these modifications is the opcode table, which is a large table of elements that each contain a widely variable set of instructions. The opcode table is generated automatically by pseudorandomly selecting operations and operands from predefined sets and ranges. The resulting instructions are used to fill 8-byte cells of the table. In summary, the instructions in the opcode table inspect AttMem and the state of the executing processor and add the results into the checksum. We give an overview of this table in the full version of the paper.[9] Together, the main block, modifiable blocks, and opcode table comprise the core Endor code. The main and

[7]This is technically part of the processor state, but we address it separately for clarity.

[8]The cache on x86 processors is managed internally and transparently from the user. When main memory is read, the corresponding data is automatically saved to the cache. If the cached data is later read, it is automatically pulled from the cache rather than from memory.

[9]Available at http://cs.jhu.edu/~ryan/attestation_code/

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

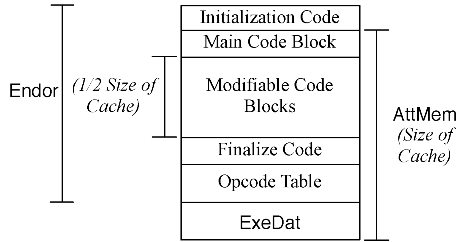

Fig. 2. High-level structure of the attested code (not to scale).

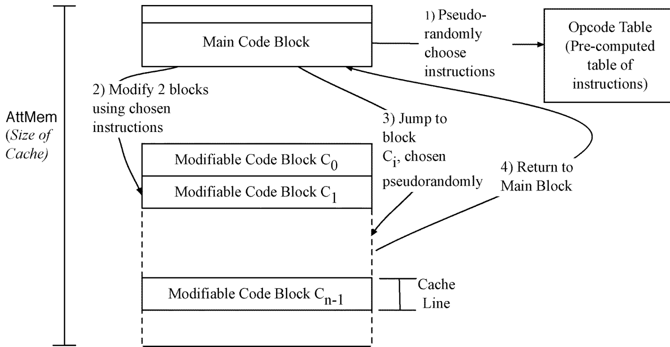

Fig. 3. Execution process of the core Endor code.

modifiable blocks execute iteratively to attest AttMem and the processor state to compute the checksum.

Fig. 3 illustrates Endor's primary iteration. Each iteration begins in the main code block. As depicted, the code pseudorandomly chooses two modifiable code blocks and modifies them by overwriting a portion of each with sets of instructions that are pseudorandomly chosen from the opcode table. Then, it chooses one more modifiable code block C_i, again pseudorandomly, and transfers execution to it. C_i executes a variety of instructions, some of which are constant and some of which have been modified to contain instructions from the opcode table. Many of these instructions inspect the state of AttMem and the processor as they are executed, and add resulting values into the checksum to conduct the attesation. For example, the instructions include memory reads from pseudorandom addresses within AttMem, and the values read are added to the checksum. The modifiable block finally returns execution to the top of the main code block to start another iteration.

After the last iteration has completed, execution continues to the finalize code. The finalize code reports the checksum to the user and sets the processor state to ExeProc. It then transfers execution to Exe.

As discussed above, the memory in AttMem fills the CPU's highest level cache exactly, and the modifiable code blocks fill one half of it. We discuss how these properties help force large overheads from forgeries in more detail in Sections V-F and VI.

Commented pseudoassembly for the main code block and the modifiable code blocks are presented in Fig. 4. We refer to it in the following sections and explain the rationale behind the instructions. The actual assembly code is listed in Appendices A and B. We now explain the design of Endor.

A. Flexible Yet Consistent Run-Times

Endor achieves flexible and consistent run-times by its straightforward, iterative structure [lines 3–5 of Fig. 4(b)].

```
//Input: k the number of iterations to be executed by Endor (ecx on CPU)
//          general purpose registers eax, ebx, edx, ebp, esi, edi set
//               to pseudorandom values
//          initialized stack pointer register p_stk (esp on CPU)
//Output: k, p_stack, and registers eax, ebx, edx, ebp, esi, edi

//ensure that the stack does not overflow by circularly wrapping stack pointer
 1:  p_stk = (stack_size/2) or p_stk        //wrap stack pointer if too low
//diffuse register values among each other
 2:  rotate(edi)                            //rotate bits of edi right by 1
 3:  eax = eax − esi                        //diffuse esi val into eax
 4:  esi = esi + edx                        //diffuse edx val into esi
 5:  ebp = ebp ⊕ esi                        //diffuse esi val into ebp
 6:  ebx = ebx ⊕ edi                        //diffuse edi val into ebx
//pseudorandomly modify one block
 7:  edi = edi and subblock_msk             //select modifiable subblock
 8:  edi = edi + subblock_base              //add subblock base addr
 9:  esi = esi and opcode_msk               //select instruction set
10:  esi = esi + opcode_base                //add op table base addr
11:  eax = eax and read_msk                 //select read src addr
12:  mem[edi]=mem[esi]                      //copy instruction set to subblock
13:  if parity(esi)==EVEN                   //if we did not write a read instr
14:      edi = p_stk                            //point edi to stack
15:  mem[edi − src_addr_disp] +=            //add the read src addr to
         eax                                //   instruction or stack
//diffuse register values
16:  ebp = ebp + (8 * eax)                  //diffuse eax into ebp
17:  ebx = ebx + esi                        //diffuse esi into ebx
18:  eax = eax + ebx                        //diffuse ebx into eax
19:  edi = edi ⊕ ebp                        //diffuse ebp into edi
//pseudorandomly modify a second block
20:  edi = edi and subblock_msk             //select modifiable subblock
21:  edi = edi + subblock_base              //add subblock base addr
22:  esi = esi + instr_set_size             //point esi to next instr set
23:  eax = eax and read_msk                 //select read src addr
24:  mem[edi]=mem[esi]                      //copy instruction set to subblock
25:  if parity(esi)==EVEN                   //if didn't write a read instruction
26:      edi = p_stk                            //point edi to stack
27:  mem[edi − src_addr_disp] +=            //add the read src addr to
         eax                                //   to instruction or stack
//diffuse register values slightly more
28:  esi = esi ⊕ eax                        //diffuse eax into esi
29:  edi = edi + ebp                        //diffuse ebp into edi
//put a destination modifiable block addr on stack so we may repoint ebx
30:  ebx = ebx and subblock_msk             //select destination block
31:  ebx = ebx + modblock_base              //add block base addr
32:  push(ebx)                              //push dest block addr on stack
//point ebx to part of the relative read instruction in the destination block
33:  eax = eax and rel_read_msk             //select portion of rel read instr
34:  ebx = ebx + eax                        //point ebx to part of rel. read
//jump to selected modifiable block
35:  goto(pop())                            //jump to addr saved on stack
```

(a)

```
//Input & Output: (same as the Main Block)

//modifiable subblock 1
 1:      (pseudorandom instruction set 1 from the opcode table)
//constant subblock 1, decrement iteration counter
 2:  eax = eax ⊕ rand_const1             //xor pseudorandom value into eax
 3:  ecx = ecx − 1                       //decrement ecx
 4:  if ecx == 0                         //if we've executed all iterations
 5:      goto finish                         //jump to the end
//modifiable subblock 2
 6:      (pseudorandom instruction set 2 from the opcode table)
//constant subblock 2, diffuse values
 7:  eax = eax ⊕ edx                     //diffuse edx into eax
 8:  esi = esi + rand_const2             //add pseudorandom value into esi
//modifiable subblock 3
 9:      (pseudorandom instruction set 3 from the opcode table)
//constant subblock 3, relative read from attested memory range
10:  eax = eax and read_msk              //select read src addr
11:  edi = edi− mem[p_stk + eax]         //read val and subtract from edi
//modifiable subblock 4
12:      (pseudorandom instruction set 4 from the opcode table)
//constant subblock 4, diffuse and return to main block
13:  ebx = ebx + rand_const4             //add pseudorandom value into ebx
14:  goto main_block                     //return to the main block
```

(b)

Fig. 4. Endor pseudocode. (a) Main block pseudocode; (b) modifiable block pseudocode.

To obtain humanly verifiable run-times, many iterations of Endor's primary loop are executed (50 000 000 in our data). As a result, the observed mean execution-time per iteration has low variance (law of large numbers). The run-time of Endor on a Core 2 processor, for example, can be approximated as EndorTime $\approx$ iterations $*$ $.1510$ μs. Clearly, the run-time can be increased or decreased by adjusting the number of iterations.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

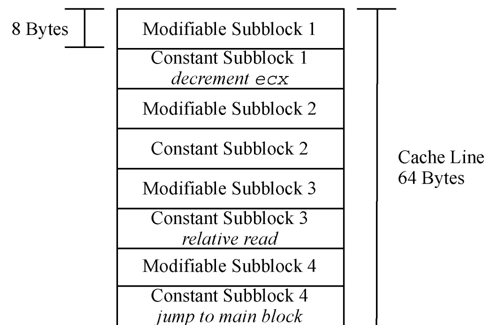

Fig. 5. Structure of a modifiable block.

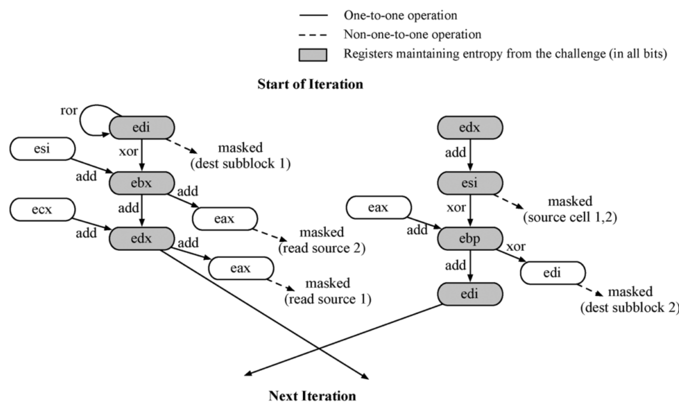

Fig. 6. Flow of values through processor registers during one iteration of Endor.

B. Strongly Ordered Execution

The execution of each iteration of Endor is dependent on the results of previous iterations and the challenge. Operations within each iteration are also strongly ordered. This ensures that the time to compute the checksum is bound by the complete execution time of all the Endor code with correct memory reads and processor state. In particular, different registers have pseudorandom contents at various points of an iteration with respect to the results of previous operations and memory reads. We use these registers to accumulate information about the state of AttMem and the state of the processor as we describe in Sections V-C, V-D, and V-E. The registers' concatenated values comprise the checksum itself.

The general structure of a modifiable code block is illustrated in Fig. 5. It is 64 bytes long, the size of a cache line. This length ensures that forgery code that must use more than the allotted portion of memory for a modifiable block will need to access significantly more main memory than it would otherwise because memory is accessed one cache line at a time. Furthermore, restricting the size of the block to no more than a single cache line minimizes its size and optimizes the run-time of Endor. The modifiable block consists of eight subblocks, four that are modifiable and four that are constant. Modifications made by the main code block consist of overwriting individual modifiable subblocks with 8-byte instruction sets from the opcode table. Although we examine other important qualities of the subblocks in further sections, with respect to strongly ordered execution, we note that constant subblocks 1, 2, and 4 each ADD or XOR a pseudorandom constant value into a register [lines 2, 8, and 13 in Fig. 4(b)]. These pseudorandom constants are each set independently during initialization using a pseudorandom number generator (PRNG) that is seeded by the user challenge.[10] This helps us achieve strong ordering, as we describe below.

Conjecture 1: If one bit of the checksum is altered during Endor execution, all bits of the checksum are pseudorandomly altered within 35 iterations.

Prior to executing the main Endor loop, the initialization code sets all the CPU's general purpose registers (GPRs) except `ecx` (the iteration counter) with the 32-bit output of the PRNG that is seeded by the user challenge.

After initialization, Endor incorporates the values of each memory read and processor register into other registers through each checksum iteration using one-to-one arithmetic operations. This is similar to the approach used in Pioneer [15]. The flow of these values is illustrated in Fig. 6. The addition of `edx` to `esi` in the upper right of the figure, for example, corresponds to line 4 in the pseudocode [Fig. 4(a)] or line 3 in the assembly (Appendix A).

The gray registers in Fig. 6 represent registers that store 64 bits of entropy (two 32-bit registers) that are continually maintained from the challenge. At any point during the iteration, at least one gray register in the left column and one gray register in the right column stores these bits (the checksum). The registers pass the entropy to other registers by adding or XORing through the iteration, as illustrated. The code is designed such that the illustrated operations between these gray registers all occur before any *non*-one-to-one (entropy destroying) operations do. For example, `edi` is masked (some of its bits are zeroed using a bitwise and operation)—line 7 of Fig. 4(a)—only *after* being XORed to `ebx`—line 6 of Fig. 4(a)—thus passing its entropy onto `ebx`. These 64 bits from the registers represented in gray are used to aggregate effects from operations and memory reads during Endor's execution and comprise the checksum itself. Specifically, we say a value is *incorporated into the checksum* to refer to that value being accumulated into this 64-bit aggregation using a one-to-one operation into one of the registers in gray. For example, a 32-bit value read from memory could be incorporated into the checksum by adding it into `edi` at the very end/beginning of an iteration.

The alternation of ADD and XOR operations provides a weak form of strong ordering over the short term because the two operations are not associative in general [15]. However, Endor achieves a stronger dependence on previous operations after several iterations, by altering the registers with pseudorandom values.

If the flow of values in Fig. 6 is extended to three iterations, a value in any register shown is propagated to any other register through exclusively one-to-one operations. Therefore, a single bit error in any value incorporated into the checksum propagates through all registers in gray in a maximum of three iterations. Lastly, we note that the modifiable block to which the main block jumps is dependent on the value of `ebx` each iteration [lines 30–32 and 35 of Fig. 4(a)]. Although this destination block is not dependent on the value of all bits of `ebx`, the rotate bits right (`ror`) operation requires that any single bit error in a register propagates through all other bits in a maximum of 32 iterations. Thus, after $32 + 3 = 35$ iterations, a bit error in

[10]We use RC4, discarding the initial 2048 bytes.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

any register affects a bit of `ebx` that determines the modifiable block to which the main block jumps. From this point, the values in the registers rapidly diverge from the correct values since each modifiable block incorporates different, pseudorandomly chosen constants into the values of the registers as explained early in this section. Hence, a single bit error in any value incorporated into the checksum causes a quick divergence in all the checksum bits.

We applied statistical tests to verify that the distribution of the register values is approximately pseudorandom. We ran randomness tests [29] on the concatenated values of `edi` and `edx` at the start of 5 000 000 iterations of Endor from each of five challenges.[11] The results of five different tests[12] stated that truly random data would perform "worse" (i.e., appear less random) than our recorded registers at least 26% of the time.

The values in the checksum registers are heavily relied upon by Endor. They are used, for example, to select source and destination code blocks for modification and target code blocks for execution. Thus, the execution path and process of Endor itself also strongly depend on the results of all its previous operations.

C. Altered Memory Detection

We have established the way that values are incorporated into the checksum and the way that Endor's execution is made highly dependent on the correctness of these values. We now show how we can use this property to inspect the state of a system's memory to ensure that any alterations in the memory yield an incorrect checksum when Endor is run.

There are two means by which Endor reads memory. The reasons for both of these are covered in Section V-F when we discuss limiting the methods of possible forgery.

The first method is enabled when modifiable subblocks are modified to contain memory reading instructions that read from direct addresses (i.e., the source address is an integer value encoded directly into the read instruction—e.g., `<add (0x4010ee3), %eax>`). Every entry of the opcode table that has an index with odd parity contains such a memory reading instruction. When making modifications to the modifiable blocks, the main code block checks the parity of the opcode table entry that it uses for the modification. When it is odd, it writes a pseudorandom source address into the newly written instruction [lines 7–15 and 20–27 of Fig. 4(a)]. As a result, on average, one half of the modifiable subblocks have memory reading instructions.

The second method of reading memory uses a register-relative read instruction in the third constant subblock of every modifiable block [Fig. 5 and lines 10–11 in Fig. 4(b)] of Endor. (A register-relative read uses the value of a register in the computation of the source address—e.g., `<add (%ebx), %eax>`.) This read instruction is executed once per iteration.

Conjecture 2: The probability P of an adversary modifying attested memory without it being incorporated into the checksum is bounded by

$$P \leq \left(1 - \frac{.587159}{M}\right)^{4n} \left(1 - \frac{3.52295}{M}\right)^{n}$$

where M is the size of the attested memory in bytes and n is the number of iterations executed by Endor.

Following Conjecture 1, if Endor reads any incorrect bit of memory and incorporates it into the checksum, then Endor yields an incorrect checksum with high probability. We also ensure that memory reads occur with a uniform distribution over a memory region that covers all attested code since the source addresses come from registers which have an approximately random distribution (Section V-B). As a result, we can derive the probability P that an adversary can modify only a single bit of attested memory without it being incorporated into the checksum. The derivation is given in the full version of the paper.[9] For our test machines with the largest cache size, a Core and Core 2 Duo with 2 MB of cache, the code attests 2 MB of memory. We chose to run the code for 50 000 000 iterations during our tests because the number provided reasonable time properties for human verifiability. Thus, in our examples, the probability of an adversary modifying a single bit without it being incorporated into the checksum is less than 1.597×10^{-61}.

D. Altered Location Detection

Endor uses many registers as both sources in operations whose results are incorporated into the checksum and as memory addresses for accessing data and code. For example, `ebx` is used as the destination address for jumping to a modifiable block [lines 30–32 and 35 in Fig. 4(a)], and it is used as a source in many instructions in the opcode table and hence in many executed instructions in the modifiable subblocks. Thus, if one of these execution or data addresses is altered, an incorrect checksum results with high probability.

E. Altered Processor State Detection

We use several approaches to ensure that the processor is in state EndorProc during checksum computation. First, there are several processor state variables we do not need to directly account for at all. For example, modifications to paging are not problematic in general because they do not alter the virtual memory space, which is what all the code operates in. Similarly, the FPU and the system call (`sysenter`) destination address only become relevant when they are explicitly used by the attested executable. As a result, rather than checking the state of these processor elements in Endor, we simply place a requirement on Exe that it not conduct any operations that would be affected by these CPU state variables without first explicitly setting them to a known state.

Segmentation is similar to paging in the sense that it mostly does not affect the virtual memory space. One difference between segmentation and paging, however, is that segmentation operates independently for data and code and can thus separate the physical memory spaces for each. The self-modifying structure of Endor prevents this possibility by modifying memory used as both code and data.

We account for several processor state variables by explicitly checking them and incorporating them into the checksum. For example, the opcode table includes some instruction sets that read the address of the IDT and XOR it into one of the GPRs.[13] Our use of the opcode table provides several benefits. First, since

[11]We used a slightly modified user-mode version of Endor to obtain the values.

[12]These include a binary matrix rank, discrete Fourier transform, Maurer's universal statistical, random excursion, and linear complexity test [29].

[13]Recall that Endor relocates the IDT within attested memory during initialization to verify that no malicious interrupt handlers are set during checksum computation.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

the table contains on the order of tens of thousands of instruction sets (for cache sizes of 512–2048 KB), it allows us to choose approximate probabilities for the execution of various instructions by including those instructions in an appropriate number of opcode table entries. In particular, we can include instructions that are executed with very low probability each iteration. This is important for reading processor structures such as the IDT address and model specific registers (MSRs) because reading them is often very slow (on the order of hundreds of CPU cycles). However, the possibility that they may be read during the execution of any modifiable block and incorporated into the checksum requires a forging adversary to also modify memory. Specifically, if an adversary changes these processor structures, she must alter Endor's code to avoid computing an incorrect checksum. The second benefit of the opcode table in this context is its flexibility. If a security-critical processor structure is discovered or added to CPUs and needs to be checked, the opcode table can easily be changed to include instruction sets that check the new processor structure (more details on the opcode table are included in the full version of the paper[9]). The state of the flags are similarly checked in this manner.

Endor must also prevent an adversary from setting hardware breakpoints or watchpoints of her choice. We could check the state of these mechanisms by including instructions in the opcode table that explicitly inspect them and incorporate their state into the checksum the same way we check the IDT.[14] However, we ensure the integrity of hardware breakpoints and watchpoints by using them instead to ensure a large workload for an adversary attempting to forge a checksum using the breakpoints, which we discuss further in Section V-F. All of our test processors support a combined total of up to four hardware breakpoints and watchpoints. We set memory-read breakpoints on four pseudorandomly chosen memory addresses within attested memory. We set the interrupt handler so that every time memory is accessed from one of these addresses, a register value is incorporated into the checksum. The read addresses that trigger the breakpoints are selected pseudorandomly using the PRNG that is seeded by the challenge during initialization. An adversary wishing to use the hardware breakpoints herself must omit at least one of the breakpoints used by Endor. Therefore, under this solution, the adversary must check whether the omitted breakpoint should have been fired upon every memory read so she can potentially emulate its effects on the checksum. Because memory reads occur three times per iteration, on average, this forces significant overhead on the adversary as we show in Sections V-F and VI.

F. Limited Methods of Forgery

We have shown how Endor yields incorrect checksums when an adversary modifies processor state or attested memory. Now we examine the way it limits the methods an adversary can use to forge a checksum.

We restrict an adversary who is attempting to forge a checksum to 1) regularly accessing a significant amount of main memory, 2) simulating execution of the modifiable blocks using some form of interpretive execution, or 3) using hardware breakpoints. In Section VI, we show that each of these requires significant slowdowns in the computation of a forged checksum.

First, recall that an expected one half of the modifications made by the main code block consist of writing memory read instructions with direct source addresses to the modifiable subblocks (Section V-C).[15] For an adversary to successfully forge a checksum, we have shown that she must modify Endor and hence modify at least one bit of attested memory. Simultaneously, these read instructions access values from the modified memory with overwhelming probability (Conjecture 2) and incorporate the altered value into the checksum, causing it to diverge. Hence, the read instructions or their source addresses must be changed or replaced to prevent an incorrect checksum. However, since these read instructions with their directly coded source addresses may be read themselves, this structure creates an immediate conflict between memory that is read and memory that is executed during a forgery. One method of forging a checksum follows:

1) In addition to a copy of the legitimate Endor modifiable code that is read, the adversary maintains a second, altered copy of the code that is executed (a *memory-copy forgery* [15]). In this case, for every modification made to the code during execution, the adversary needs to reflect the change in the copy being attested as well as the copy being executed since the modified code could be read and/or executed from that point. Because the attested memory fills the entire CPU cache, and modifications must continually be made to both copies of the modifiable code, this approach requires a significant increase in main memory accesses compared to unmodified Endor.

Next, recall that the third constant subblock of each modifiable code block contains a read instruction that reads from an address that is contained in a register [*register-relative read*-Fig. 5 and lines 10–11 of Fig. 4(b)].[16] Hence, the result of the read is dependent on the value of the register, `eax`, at the time of the read. Furthermore, approximately 35.3% of the instruction sets in the opcode table, and thus in the modifiable subblocks modify `eax`. Similarly, other modified instructions are likely to affect various values before they are used in arithmetic that affects `eax` (as in a typical dependency tree). If we temporarily disregard hardware breakpoints and interrupts, two alternatives to forgery type 1 now follow, only one of which offers a potential advantage over the memory-copy forgery:

2) The adversary avoids maintaining a separate copy of the modifiable code. In this case, she can keep either the legitimate Endor modifiable blocks in memory or altered versions of the blocks.

Suppose the adversary keeps the legitimate blocks in memory. If we ignore breakpoints, she cannot directly execute a block unless she can first determine that its register-relative read does not read from memory that she has altered. However, doing so involves ascertaining the effects of the block execution on the register that provides

[14]One caveat here is that the number of instructions that check this in the code would need to be greater than the number of hardware breakpoints supported by the processor. This is necessary to prevent an adversary from using the breakpoints by setting a hardware breakpoint at the instructions where they would supposedly be inspected and then forging the result of the inspection within the breakpoint interrupt handler.

[15]Recall that these are read instructions that have the source memory address directly encoded into the instruction, e.g., `<sub (0x400472d), %ebp>`.

[16]Instructions in approximately 25% of the opcode table cells make checks that the register relative read instruction is in place in the executing block.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

TABLE I
TEST PROCESSORS

CPU Core	Clock Speed	Cache Size
Pentium 4 Model 0	3.2 GHz	512 KB
Pentium 4 Model 3	3.2 GHz	1024 KB
Core	1.67 GHz	2048 KB
Core 2	1.86 GHz	2048 KB

the read's source address (`eax`). Hence, the adversary must simulate the execution of the block to determine the address used by the relative read or to emulate the block's execution entirely (a *simulation forgery*).

Alternatively, the adversary may be able to keep only altered versions of the modifiable blocks in memory. If she does not simulate the blocks' execution, each memory read must be expanded with code to allow translation of altered memory to AttDat as it is read, since the correct AttDat blocks are not resident.[17] However, an unaltered Endor block exactly fills a small cache line (64 bytes—Section V-B) before being expanded. Since any modifiable block contains as many as five memory read instructions,[18] even small amounts of additional code (13 bytes or approximately two to five instructions) per read instruction have the effect of doubling the size of each block. As a result, this approach requires frequent access of approximately as much memory as the memory-copy forgery (forgery type 1), and at the same time, it requires significantly more computation.

A reduction to the methods of forgery above applies when an adversary must modify memory read instructions or adjacent code to verify that she is not reading modified data. Breakpoints have potential to violate this condition by enabling redirection of execution without a modification to instructions and provide a third approach to forging a checksum:

3) The adversary uses hardware breakpoints (a *breakpoint forgery*). The possibilities with breakpoints are complex, but because we use all possible processor breakpoints in Endor, an adversary trying to use a breakpoint for forgery must remove one of the legitimate breakpoints. As a result, any forgery computation that uses breakpoints must use one at least once each iteration because it must make sure that the register-relative read instruction in the modifiable subblock does not read from an address that should have a breakpoint set on it. If it does not, it must modify or interpret the read instruction and is reduced to one of the two methods of forging above, respectively.

Memory access times are typically at least 50 (on some systems as much as 500) times greater than the time of a CPU clock cycle on the same system [16], and this is true of our test processors. This relationship makes memory-copy forgeries very costly since the cache is exactly filled with attested memory during execution of an unmodified Endor, and the continual use of a large portion of additional memory (another half the size of the cache) necessitates many accesses to main memory. Simulation is also expensive as we will see in Section VI, requiring instruction interpretation and code writing or conditional execution. An adversary can also always simulate by the basic principles of a Turing machine [30], [31]. Lastly, we show in Section VI that firing interrupts such as breakpoints also causes significant delays in execution.

G. Fast Iterations

Recall that we want Endor to be fast because delays caused by attempts to forge checksums are proportionally larger for shorter Endor run-times. Below, we mention some aspects of Endor's design that contribute to its speed.

Aside from memory accesses, the majority of Endor's instructions require only a single CPU cycle (`add`, `and`, `xor`, `ror`, `lea`, `sub`, `or`, `dec`). The ability to use such fast operations is a product of a bottom up design. For example, every pseudorandom address (to be modified, to be read from, or to be jumped to) is chosen by an `and` and an `add` or `lea` instruction. We achieve this with careful sizing of the modifiable subblocks, the modifiable blocks, and the opcode table.

The code for each iteration is also short. The main code block is 128 bytes in length,[19] and each modifiable code block is 64 bytes long.

Furthermore, we avoid explicitly setting flags in all but one case. Instead, our conditions are checked automatically through instructions with other purposes. Like the low-latency operations, this is partly enabled by the higher-level design. For example, allowing the parity of an opcode-table entry's index to indicate a read or nonread instruction provides a method for determining these instructions without the need for any additional lines of code. We also preserve flags for prolonged series of instructions by using `lea` for addition rather than the typical `add` because it does not affect the flags.

Lastly, despite some relatively complex conditional behavior, the code utilizes conditional moves to completely avoid conditional jumping so that the prefetch queue is always filled with instructions that will be executed.

VI. IMPLEMENTATIONS

We now evaluate the run-time overhead of checksum forgery computations. Our objective is to find lower bounds on the overhead that forging code must incur. To do this, we implemented Endor and very simple operations that represent a subset of the tasks required to forge a checksum.[20] In accordance with our discussion of possible methods of forging a checksum in Section V-F, we implement simple portions of code to compute a forgery using the memory-copy method, two possible simulation methods, and a method that utilizes hardware breakpoints. Further, to account for the possibility of an adversary gaining an advantage by compressing data in any of these methods of forging (up to a reduction of one half), we reduce the memory occupancy of the code for the simulation and breakpoint methods by half. That is, for modeling those approaches, our forgery code only attests a range of memory equal to the size of half the CPU cache, and its modifiable blocks occupy only one quarter of the cache. We do not need to compress the code for the memory-copy forgery in our benchmarks since decompression is actually a subset of simulation in

[17]The adversary may be able to use `call` instructions to transfer execution to routines, which can translate the altered memory. These instructions also require additional code, however, because pushed return addresses overwrite attested values in the stack (Section V).

[18]The probability that any specified modifiable block contains five memory read instructions at any given time during Endor's execution is 1/16.

[19]A power of 2 only by coincidence.

[20]All of our code is implemented as Linux kernel modules and is available at http://cs.jhu.edu/~ryan/attestation_code/

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

TABLE II
RUN-TIME OF Endor AND FORGERY CODE. (ALL TIMES ARE IN SECONDS.)

	Pentium 4 Model 0			Pentium 4 Model 3			Core			Core 2		
	Mean	StdDev	Overhd	Mean	StdDev	Overhd	Mean	StdDev	Overhd	Mean	StdDev	Overhd
Endor	8.327	0.872	0.0%	8.915	0.391	0.0%	8.658	0.662	0.0%	7.548	0.349	0.0%
Mem Copy	17.090	0.568	105.24%	14.734	0.51	65.3%	14.140	0.201	63.32%	11.698	0.348	54.98%
Sim Copy	15.805	0.330	89.80%	19.668	0.114	120.62%	13.846	0.177	59.93%	11.332	0.128	50.15%
Sim Cond	14.096	1.315	69.28%	17.242	1.12	93.41%	17.316	0.562	100.0%	15.010	0.632	98.87%
Breakpoint	31.171	0.522	274.34%	40.737	0.244	356.94%	22.885	0.221	164.34%	19.838	0.196	162.84%
Complete	28.956	0.292	247.74%	24.937	0.099	179.72%	23.463	0.200	171.01%	20.176	0.110	167.31%

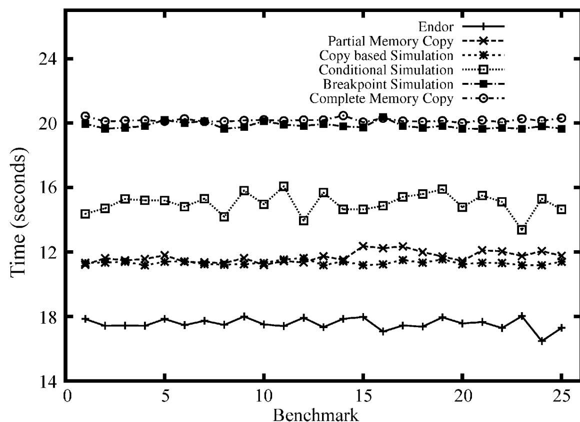

Fig. 7. Run-time of code on Core 2 CPU.

the basic sense that we consider it. Specifically, for our analysis, we consider simulation of a piece of code to be any effort to dynamically obtain its effects without directly executing it from its original location in memory. Hence, decompressing and executing compressed code is actually a form of simulation, and a memory-copy forgery involving compression/decompression is a superset of the work for the simple simulation forgeries we analyze. Additionally, for the purpose of demonstration, we have implemented complete code to correctly forge a checksum although it is not part of our analysis.

We measured the run-times of 25 executions of each implementation on the processors listed in Table I. All but one CPU core and hyperthreading were disabled when applicable. Each of the 25 runs used a different challenge, and the size of the code was adjusted to correspond to the cache size of the respective executing processor as described in our architecture. We set the number of iterations to 50 000 000 because this number seemed to provide reasonable timing properties for human verifiability. Our results are presented in Table II and Fig. 7. We briefly describe each implementation below.

A. Memory-Copy Forgery

The idea behind the memory-copy forgery is to maintain two copies of the modifiable blocks, one for attestation and one for execution. While the complete version of such a forgery involves many complex aspects as discussed briefly in Section VI-E, one requirement is that every modification made to the modifiable blocks is reflected in both copies of the blocks in order to ensure correct execution and attestation.

In the memory-copy code we used for our benchmarks, we added eight instructions to the main block of Endor's code. For each of the two modifications, we added two `movsd` instructions and two `lea` instructions to modify the second copy of the blocks (which are never actually used) and then correct the two registers that were changed respectively.

B. Simulation Through Code Modification

Without breakpoints, if the adversary does not maintain a separate copy of memory from which to execute, then she must simulate the execution of the modifiable blocks to avoid reading from a modified address, as discussed in Section V-F. There are effectively two ways code can be simulated. After reading the code to be simulated, one can either write code to memory that mimics the actions of the simulated instructions and then execute it or conditionally execute code already resident in memory (or some combination of the two).

In this section, we discuss simulation through the write-then-execute approach. Any simulation using the technique must read the code it wishes to simulate, write some new code to emulate it, and then execute it. Our partial forgery code is Endor with a small addition to the end of the main block. The additional code blindly copies the intended destination modifiable block to a temporary block that resides in cache and transfers execution onto it. The copying process omits any bytes of code that are consistent among modifiable blocks, and we succeed the copying code with a minimal length series of no-op instructions to avoid invalidating the prefetch queue. Recall that the code is also only half the size of the real Endor code for the same machine. The copying process involves ten `movsd` instructions and ten `add` instructions.

This simple process of effectively executing the target modifiable block from a different location is a subset of the requirements to fully simulate code using the write-then-execute approach.

C. Simulation Through Conditional Execution

We examine the second possible approach to a simulation forgery where, rather than writing new code, the adversary conditionally executes code that is already resident in memory. This implementation conservatively models a basic process whereby simulation code reads each modifiable subblock of the simulated modifiable block, computes a fast hash of the subblock, and then jumps to code that appropriately emulates it according to the hash. The implementation uses a copy of the corresponding legitimate opcode table where each instruction set of the table is spaced 8 bytes apart so it may be followed by a jump instruction to return execution control to the "simulator." It also models each hash look-up function with only an `and` and `add` instruction. As a result of the simplified hash, the code does effectively no real simulation of the block. It simply jumps to a table cell approximately pseudorandomly for each subblock it pretends to simulate, which then jumps back to

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

the simulator. Again, recall that it is only attesting a range of memory half the size of the cache.

D. Forgery Using Breakpoints

Rather than interpreting or using a second copy of memory, this method of computing a forgery utilizes a hardware breakpoint per iteration. Disregarding the breakpoints that are normally used by Endor (Section V-E), each iteration, the main block of this implementation sets a hardware breakpoint at the relative-read instruction in the modifiable block that will be executed. The debug interrupt handler that is called when the breakpoint triggers simply returns. The code does not set up the breakpoint trigger to cause an expensive task switch as would be necessary to prevent alteration of attested stack data nor does its interrupt handler actually aid in computing a forgery.

E. Complete Memory-Copy Forgery

In the code discussed above, we implemented simple subsets of requirements to successfully forge a checksum using each method. We also implemented code that computes a correct, forged checksum for the purpose of demonstration. It is the fastest code we were able to develop but is quite complex and not a part of our analysis. We highlight some of its aspects.

The complete forgery code uses the memory-copy method where the correct modifiable blocks are maintained in the correct location and a second copy is executed from outside AttMem. In addition to the added memory writes made by the partial memory-copy implementation (Section VI-A), the most expensive portion of computing the complete forgery is checking whether or not each written instruction contains a read because it must be done twice per iteration, and the conditional jumps involved have potential to invalidate the prefetch queue. When a modifiable block is to be modified with a read instruction, the code must also generate appropriate addresses for both the attested and executed blocks. Several other modifications were necessary.

F. Benchmark Summary

Our results, timed over 25 executions on each processor, are presented in Table II and Fig. 7. We see that the most effective portion of forgery code incurs at least a 50% overhead in execution time on all processors.

Overall, the run-time overheads created by the portions of forgery code seem sufficient for human verification. The minimum overhead incurred by forgery code on the Core 2 is approximately 3.8 s. This range seems distinguishable fairly easily by a human with a stopwatch over a 7.5-s EndorTime and could also easily be increased by increasing the number of iterations.

The overhead of the hardware breakpoint method is 162.84% or more of the standard run-time on all processors, which suggests that frequent use of hardware breakpoints is impractical. Large overheads are caused by writing to the breakpoint register and the invalidation of pipelined data preprocessed by the CPU upon each breakpoint interrupt.

The primary source of overhead for the copy-based simulation (Sim Copy) forgery is derived from the delay of copying and invalidation of the instruction cache while the conditional execution based simulation (Sim Cond) is mainly slowed by increased memory accesses and the processor's frequent inability to fully prefetch conditionally executed code. The partial memory-copy (Mem Copy) method saw the significant slowdowns we expected from accesses to slow, main memory. Lastly, the run-time overhead caused by computing a complete forged checksum is sharply greater (167.31% on the Core 2 architecture) than that of the simpler code portions.

VII. Conclusion

We have presented a new approach to software-based attestation primitives that relies on the basic principle that accesses to main memory are significantly slower than accesses to cache or CPU clock cycles. We implement a construction for the 32-bit x86 instruction set. Our empirical results demonstrate that simple portions of code required to compute a forged checksum cause significant overheads in Endor's run-time. Because the overheads are large enough that they can be detected by a human with a stop watch, the techniques may eventually be able to contribute to the design of voting and other systems where people can establish the authenticity of software on a local machine.

Appendix

A. Main Block Code

The code for the main block is included below. It corresponds to the process outlined in Fig. 4(a). Most of the operations are not coded in the same order as they appear in the pseudocode of Fig. 4(a) to avoid dependencies between nearby instructions. The code is written in AT&T syntax (i.e., instructions are of the format `<op src, dest>`).

```
 1: sub    %esi, %eax
 2: ror    %edi
 3: add    %edx, %esi
 4: or     $stack_size/2, %esp
 5: xor    %esi, %ebp
 6: xor    %edi, %ebx
 7: and    $mod_sub_mask, %edi
 8: and    $read_mask, %eax
 9: and    $table_cell_mask, %esi
10: lea    modblocks_start(%edi), %edi
11: lea    table_base(%esi), %esi
12: lea    (%ebx, %ecx), %edx
13: movsd
14: lea    (%ebp, %eax, 8), %ebp
15: movsd
16: cmovp  %esp, %edi
17: add    %esi, %ebx
18: add    %eax, disp_offset(%edi)
19: add    %ebx, %eax
20: xor    %ebp, %edi
```

```
21: and    $read_mask, %eax
22: and    $mod_sub_mask, %edi
23: and    $blk_start_mask, %ebx
24: add    $modblocks_start, %edi
25: test   $table_entry_mask, %esi
26: movsd
27: lea    $modblocks_start(%ebx),
             %ebx
28: movsd
29: push   %ebx
30: cmovp  %esp, %edi
31: xor    %eax, %esi
32: add    %eax, disp_offset(%edi)
33: and    $rel_read_mask, %eax
34: add    $4, %esp
35: add    %ebp, %edi
36: lea    rel_read_base(%ebx, %eax),
             %ebx
37: jmp    *-4(%esp)
```

B. Modifiable Block Code

The code for each modifiable block is included below. It corresponds to the process outlined in Fig. 4(b). Large portions of the code are filled in pseudorandomly from the opcode table and with pseudorandomly from the pocode table and with pseudorandimly chosen constant during initializaing.

```
;modifiable subblock 1
1: .byte  0,0,0,0,0,0,0,0
;constant subblock 1
2: xor    $rand_const1, %eax
3: dec    %ecx
4: jz     128-2
;modifiable subblock 2
5: .byte  0,0,0,0,0,0,0,0
;constant subblock 2
6: xor    %edx, %eax
7: add    $rand_const2, %esi
```

```
;modifiable subblock 3
1: .byte  0,0,0,0,0,0,0,0
;constant subblock 3
2: and    $read_mask, %eax
3: sub    (%esp, %eax), %edi
;modifiable subblock 4
4: .byte  0,0,0,0,0,0,0,0
;constant subblock 4
5: add    $rand_const4, %ebx
6: jmp    main_block
```

Acknowledgment

The authors would like to thank F. Monrose for his helpful insights. They are also grateful for K. Amari's and Z. Crisler's thoughtful comments and discussions.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.

REFERENCES

[1] H. Hursti, Critical Security Issues the Diebold Optical Scan Design Jul. 2005 [Online]. Available: http://www.blackboxvoting.org/BBVreport.pdf

[2] H. Hursti, Diebold TSx Evaluation: Critical Security Issues With Diebold TSx May 2006 [Online]. Available: http://www.blackboxvoting.org/BBVreportIIunredacted.pdf

[3] A. J. Feldman, J. A. Halderman, and E. W. Felten, "Security analysis of the Diebold AccuVote-TS voting machine," in *Proc. EVT '07:USENIX/ACCURATE Electronic Voting Technology Workshop*, Boston, MA, 2007.

[4] TPM Main Part 1—Design Principles, Specification Version 1.2. Trusted Computing Group, Mar. 2006, revision 94.

[5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proc. ACM Symp. Operating Systems Principles*, Bolton Landing, NY, 2003.

[6] J. D. Tygar and B. Yee, "Dyad: A system for using physically secure coprocessors," in *Proc. IP '94: Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment*, Boston, MA, 1994.

[7] B. Yee and J. D. Tygar, "Secure coprocessors in electronic commerce applications," in *Proc. USENIX Workshop on Electronic Commerce*, New York, 1995.

[8] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn, "Attestation-based policy enforcement for remote access," in *Proc. ACM Conf. Computer and Communications Security (CCS '04)*, Washington, DC, 2004.

[9] B. Chen and R. Morris, "Certifying program execution with secure processors," in *USENIX HotOS Workshop*, Lihue, HI, 2003.

[10] E. Shi, A. Perrig, and L. van Doorn, "Bind: A fine-grained attestation service for secure distributed systems," in *IEEE Symp. Security Privacy*, Oakland, CA, 2005.

[11] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, "The digital distributed system security architecture," in *Proc. NIST/NCSC National Computer Security Conf.*, Gaithersburg, MD, 1989.

[12] J. Franklin, M. Luk, A. Seshadri, and A. Perrig, PRISM: Enabling Personal Verification of Code Integrity, Untampered Execution, and Trusted I/O on Legacy Systems or Human-Verifiable Code Execution CMU Cylab, Tech. Rep. 07-010, 2007.

[13] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *Proc. USENIX Security Symp.*, Washington, DC, 2003.

[14] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Sofware-based attestation for embedded devices," in *Proc. IEEE Symp. Security and Privacy*, Oakland, CA, 2004.

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proc. ACM Symp. Operating Systems Principles (SOSP '05)*, Brighton, U.K., 2005.

[16] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," in *Proc. Network and Distributed System Security Symp. (NDSS '03)*, San Diego, CA, 2003.

[17] P. S. Tasker, "Trusted computer systems," in *Proc. IEEE Symp. Security and Privacy*, Oakland, CA, 1981.

[18] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A trusted open platform," *IEEE Spectrum*, vol. 36, no. 7, pp. 55–62, Jul. 2003.

[19] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proc. IEEE Symp. Security and Privacy*, Oakland, CA, 1997.

[20] U. Shankar, M. Chew, and J. D. Tygar, "Side effects are not sufficient to authenticate software," in *Proc. USENIX Security Symp.*, San Diego, CA, 2004.

[21] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems—Codebase 2005 [Online]. Available: http://www.cs.cmu.edu/~arvinds/pioneer.html

[22] R. Gardner, S. Garera, and A. Rubin, "On the difficulty of validating voting machine software with software," in *Proc. USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*, Boston, MA, 2007.

[23] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Proc. ACM Workshop on Wireless Security (WiSe '06)*, Los Angeles, CA, 2006.

[24] A. Seshadri, M. Luk, and A. Perrig, "Sake: Software attestation for key establishment in sensor networks," in *Proc. Int. Conf. Distributed Computing in Sensor Systems (DCOSS '08)*, Santorini Island, Greece, 2008.

[25] V. Gratzer and D. Naccache, "Alien versus quine, the vanishing circuit and other tales from the industry's crypt," in *Proc. Advances in Cryptology (EUROCRYPT '06)*, St. Petersburg, Russia, 2006.

[26] J. Franklin and M. C. Tschantz, On the (Im)possibility of Timed Tamper-Evident Software in (A)synchronous Systems 2008 [Online]. Available: http://www.cs.cmu.edu/~jfrankli/unpublished/timed-tamper-evident-softwa%re.pdf

[27] S. Doshi, F. Monrose, and A. D. Rubin, "Efficient memory bound puzzles using pattern databases," in *Proc. Applied Cryptography and Network Security (ACNS '06)*, Singapore, 2006.

[28] C. Dwork, A. Goldberg, and M. Naor, "On memory-bound functions for fighting spam," in *Proc. Advances in Cryptology (CRYPTO '03)*, Santa Barbara, CA, 2003.

[29] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications NIST Special Publication 800-22, 2001.

[30] E. Bernstein and U. Vazirani, "Quantum complexity theory," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1411–1473, 1997.

[31] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," in *Proc. London Mathematical Society*, 1937, vol. 43, no. 2, pp. 230–265.

Ryan W. Gardner received the B.A. degree in mathematics and the B.S. degree in computer science from the University of Colorado at Boulder, in 2005, and the M.S. degree in computer science from Johns Hopkins University, in 2007. He is currently working toward the Ph.D. degree in the Computer Science Department, Johns Hopkins University.

His primary research interests include software integrity, high-assurance protocols, and resource optimization. He has also worked for VMware and participated in state-sponsored security reviews of Florida voting machines.

Sujata Garera received the Masters degree in information networking from Carnegie Mellon University, and the Ph.D. degree in computer science from Johns Hopkins University.

She is a Postdoctoral Research Fellow at the Computer Science Department, Johns Hopkins University. Her research interests span the areas of systems, network, and software security.

Aviel D. Rubin received the B.S., M.S.E., and Ph.D. degrees from the University of Michigan, in 1989, 1991, and 1994, respectively.

He is Professor of Computer Science and Technical Director of the Information Security Institute, Johns Hopkins University. He directs the NSF-funded ACCURATE Center for Correct, Usable, Reliable, Auditable and Transparent Elections. Prior to joining Johns Hopkins, he was a Research Scientist at AT&T Laboratories. He is also a cofounder of Independent Security Evaluators (securityevaluators.com), a security consulting firm. He has testified before the U.S. House and Senate on multiple occasions, and he is author of several books including *Brave New Ballot* (Random House, 2006), *Firewalls and Internet Security*, second edition (with Bill Cheswick and Steve Bellovin, Addison Wesley, 2003), *White-Hat Security Arsenal* (Addison Wesley, 2001), and *Web Security Sourcebook* (with Dan Geer and Marcus Ranum, Wiley, 1997). He is Associate Editor of IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, Associate Editor of *IEEE Security and Privacy*, and an Advisory Board member of Springer's Information Security and Cryptography Book Series.

In January 2004, *Baltimore Magazine* named Dr. Rubin a Baltimorean of the Year for his work in safeguarding the integrity of our election process, and he is also the recipient of the 2004 Electronic Frontiers Foundation Pioneer Award.

Authorized licensed use limited to: Johns Hopkins University. Downloaded on November 19, 2009 at 01:32 from IEEE Xplore. Restrictions apply.