Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution

A. Murat Fiskiran and Ruby B. Lee
Department of Electrical Engineering
Princeton University
{fiskiran, rblee}@princeton.edu

Abstract

Many computer security threats involve execution of unauthorized foreign code on the victim computer. Viruses, network and email worms, Trojan horses, backdoor programs used in Denial of Service attacks are a few examples. In this paper, we present an architectural technique, which we call Runtime Execution Monitoring *(*REM*), to detect program flow anomalies associated with such malicious code. The key idea in REM is the verification of program code at the hash block (similar to a basic block) level. This is achieved by pre-computing keyed hashes (HMACs) for each hash block during program installation, and then verifying these values during program execution. By verifying program code integrity at the hash block level, REM can monitor instructions whose behavior is typically exploited by malicious code, such as* branch, call, return *instructions. Performance degradation with REM averages 6.4% on our benchmark programs, which can be reduced to under 5% by increasing the size of the L1 instruction cache.*

1. Introduction

The increasing complexity of modern computer systems has also contributed to the increase in computer security vulnerabilities. The most dangerous type of vulnerabilities allows an attacker to cause program flow anomalies *during* program execution, leading to arbitrary code execution on the victim computer [1]. Many of the most disruptive network worms recently encountered (e.g. Blaster, Slammer, Code Red, Nimda) have exploited such vulnerabilities [1][2]. Malicious code of this kind can propagate very fast and cause severe network disruption and data loss even before it can be identified [2]. For example, the Slammer network worm (released January 2003) infected more than 90% of all the vulnerable systems in under 10 minutes, before any meaningful human response was possible [2][3].

This work was supported in part by Kodak (A.M. Fiskiran is a Kodak Fellow) and by NSF Research Grants CCR-0208946 and CCR-0326372.

We use the term *unauthorized code* to refer to any executable (or a malicious instruction sequence embedded in an otherwise legitimate executable) that was injected into a computer system without user authorization. Security threats related to unauthorized code include: viruses (excluding macro viruses); Trojan horses; spyware and adware (programs that monitor system activity such as browsing habits and display unsolicited ads); and backdoor programs used in Distributed Denial of Service (DDoS) attacks. Clearly, a reliable mechanism to detect and prevent unauthorized code execution will contribute significantly to computer security.

In this paper, we describe an architectural technique, which we call *Runtime Execution Monitoring* (REM), to monitor program execution and to detect flow anomalies that may be linked to malicious code execution. The key idea in REM is the real-time verification of program code at the hash block (similar to a basic block) level. Therefore, REM can detect program flow anomalies that occur during execution such as buffer overrun attacks commonly used by network and email worms.

The rest of this paper is organized as follows. In Section 2, we overview the related past work. In Section 3, we describe the REM architecture. In Section 4, we present performance data. Section 5 is the conclusion.

2. Related work

2.1. Code encryption and integrity checking

The line of research that most closely parallels ours is the code encryption and integrity verification methods proposed for Digital Rights Management. Earliest works in this field proposed *bus-encrypted microprocessors*, where the program code is encrypted when it is in the memory, and exists in decrypted form only inside the processor chip [4]. Subsequent studies expanding on this idea are presented in [5]-[10]. We focus on the *eXecute Only Memory* (XOM) and the *memory integrity verification* architectures described in [6] and [7].

In the XOM architecture, software is distributed in encrypted form by the vendor and decrypted during execution on the target processor using a secret key. A simplified illustration of this scheme is shown in Figure 1(a). The encryption/decryption unit is between the L2 cache and the main memory. On an L2 read miss, the data from the memory is first decrypted before it is written to the L2 cache. On a write back, the dirty L2 line is first encrypted before being written to the memory. Because the memory is untrusted, integrity verification is also required. This is done by tagging each memory block (L2-line-sized) with a keyed hash (HMAC)[1] [6][11].

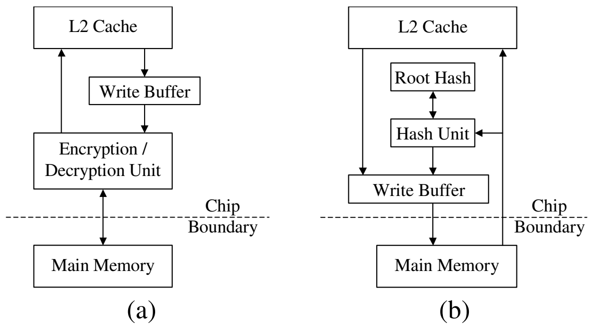


Figure 1: (a) XOM, (b) Memory Hashing

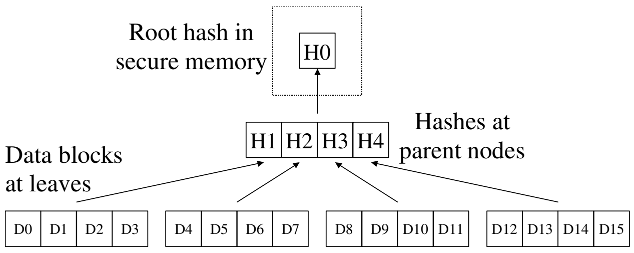


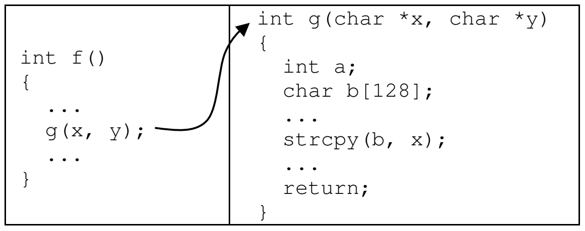
Figure 2: Memory organization in MH architecture

One of the shortcomings of XOM is that it does not completely protect against an attacker tampering with memory (in particular against *replay* attacks) [11]. To address this problem, the *memory hashing* (MH) scheme in [7] was proposed. It was later described how the two methods can be used together [10].

The MH architecture is shown in Figure 1(b). The memory is structured as a tree where the program data is placed at the leaves (Figure 2). Every node of the tree contains the hash of the nodes (or leaves) below it. At the root of the tree is the *root hash*, which is permanently kept in secure memory. On an L2 read miss, the integrity of the incoming data block is checked by recursively verifying its hash and all the hashes of its parent nodes, up to the root hash. If there is a mismatch at any level, an exception is raised. On a write back, all hashes depending on the dirty line are updated, including the root hash.

[1] A hash algorithm produces a fixed-size digest (hash) of a variable-size input. While it is easy to compute the hash of any input, it is computationally infeasible to find an input that hashes to a given value. An HMAC (Hashed Message Authentication Code, or *keyed hash*) is a hash algorithm that incorporates a secret key into the hash computation.

While XOM and MH architectures provide important security functions, they have several shortcomings that limit their usefulness. First, XOM only protects encrypted code whereas most of today's software is unencrypted and a significant fraction is open source. Second, XOM does not protect shared library code, which always exists in plaintext form, whereas virtually all modern applications rely on shared code. Third, neither XOM nor MH fully protects untrusted I/O channels, such as network interfaces. Fourth, neither architecture reliably detects flow anomalies that happen *during* program execution, which is typical of malicious code activity.



```
int f()
{
  ...
  g(x, y);
  ...
}
```

```
int g(char *x, char *y)
{
  int a;
  char b[128];
  ...
  strcpy(b, x);
  ...
  return;
}
```

Figure 3: Code with buffer overrun vulnerability

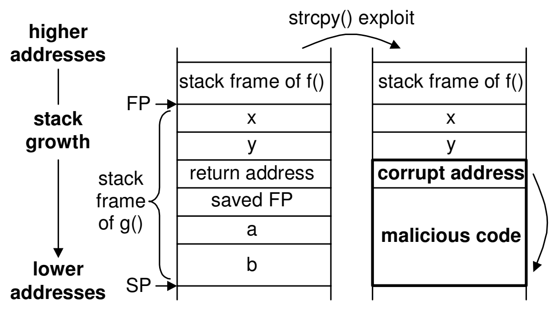


Figure 4: Buffer overrun exploit

Consider the buffer overrun exploit shown in Figure 3 and Figure 4, where the unchecked string *x* is copied into local variable *b*. Because *x* is larger than *b*, the procedure return address is overwritten, and the malicious code passed in *x* is executed following the procedure return. This anomaly is not always detected by XOM or MH because memory integrity verification is not performed when the stack frame of *g* is created. Because the hashes are updated only on L2 write back, they do not always reflect the value of the correct procedure return address, hence corruption of this address is not reliably detected.

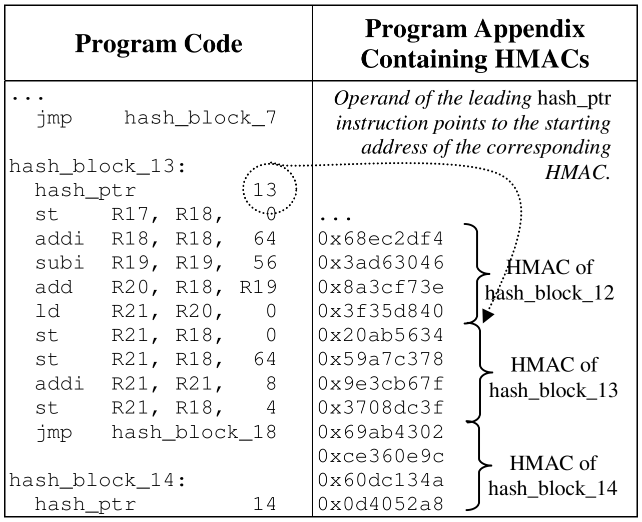
2.2. Other software and architectural defenses

Software tools and safe programming dialects have been designed to defend against vulnerabilities most commonly exploited by malicious code. *StackGhost* [12], and *Cyclone* [13] are two examples. Common drawbacks of these software methods are: protection of only selected libraries and functions; significant adverse performance impact and code bloat; and compiler modifications and/or rewriting and recompiling of legacy code.

An architectural technique is presented in [14] that guards against buffer overrun exploits. This is achieved by keeping the procedure return addresses in a *secure return address stack* (SRAS) rather than in data memory. SRAS causes no code bloat and has minimal adverse performance impact, but it does not protect against exploits other than buffer overruns nor against general unauthorized code execution.

The AMD64 architecture uses a *no-execute* (NX) flag to disable instruction execution from selected memory segments, such as the data stack [15]. However, it is still possible to inject malicious code into other unprotected memory segments, such as the heap or static/global variables [16]. Furthermore, flow anomalies can also be triggered without malicious code injection, for example by simply corrupting the procedure return address and transferring execution to a random location in the program.

There are also host-based and network-based intrusion detection (ID) tools to detect anomalous system and network activity. Examples include [17][18]. In general, ID tools can only identify intrusions with a delay [19], which is often significant. Therefore, their usefulness is limited against fast-propagating malicious code.


Program Code
Program Appendix Containing HMACs
Operand of the leading hash_ptr instruction points to the starting address of the corresponding HMAC.
HMAC of hash_block_12
HMAC of hash_block_13
HMAC of hash_block_14

```
...
  jmp     hash_block_7

hash_block_13:
  hash_ptr          13
  st     R17, R18,   0
  addi   R18, R18,  64
  subi   R19, R19,  56
  add    R20, R18, R19
  ld     R21, R20,   0
  st     R21, R18,   0
  st     R21, R18,  64
  addi   R21, R21,   8
  st     R21, R18,   4
  jmp    hash_block_18

hash_block_14:
  hash_ptr          14
```

```
...
0x68ec2df4
0x3ad63046
0x8a3cf73e
0x3f35d840
0x20ab5634
0x59a7c378
0x9e3cb67f
0x3708dc3f
0x69ab4302
0xce360e9c
0x60dc134a
0x0d4052a8
```

Figure 5: Program code in REM

3. Runtime Execution Monitoring (REM)

3.1. HMAC computation

REM verifies program execution at the basic block (hash block) level. We define a *hash block* as "*a sequence of instructions with a single entry point, single exit point, and no internal flow control instructions, such as* branch, call, return *instructions*". While this is generally identical to the definition of a basic block, we prefer to define a hash block explicitly since the basic block definition occasionally excludes the "*single entry point*" requirement.

REM involves computing an HMAC (keyed hash) for each hash block of a program when it is first installed on the host computer. The HMACs are then appended to the program. A new instruction, *hash_ptr* (hash pointer), is added to the ISA. Each hash block begins with a *hash_ptr* instruction, whose immediate operand points to the corresponding HMAC in the program appendix (Figure 5). This facilitates finding the HMAC of a given hash block during execution. Since the generation of the HMACs and the insertion of *hash_ptr* instructions can be performed directly on executable code, recompilation and compiler modifications are not necessary. This makes REM suitable for protecting proprietary and legacy code, where source code is not available.

The key used to generate the HMACs is called the REM key, which is randomly assigned to each processor and is required for software installation. HMACs can be generated using a hash algorithm (e.g. SHA-1, MD5 [11]) or a symmetric-key cipher. In this paper, we use the AES symmetric-key cipher [20] with 128-bit keys because it has very fast hardware implementations [21]. We set the default HMAC size equal to the AES block size, which is 128 bits. To compute the HMAC of a hash block, we first parse the instructions into 128-bit blocks (i.e. groups of four if the instruction size is 32-bits). Zero padding is used if a block contains uneven number of instructions. Each block is then encrypted with AES using the REM key. Finally, the encrypted blocks are XOR'ed together to generate a 128-bit HMAC. Using AES in the ECB mode is acceptable because the hash block size is usually small [11]. Therefore, we believe that the security of this scheme is not less than other 128-bit hash algorithms.

There are many hardware designs for fast AES implementation. One example is described in [21], which can perform AES encryption in 10 cycles, with an effective pipelined latency of 1 cycle. The area of this design is reported as 173,000 gates. Better performance (or smaller area) can be realized for REM by exploiting the fact that the REM key is fixed for each processor. In this paper, we will assume using an AES unit with a 20-cycle absolute latency and a 1-cycle effective pipelined latency. As we will discuss in Section 4.2, a longer encryption latency can be simply accommodated by using a larger HMAC read buffer.

3.2. REM architecture

Figure 6 shows the elements in the REM datapath. L1 I-cache is shared to store both the instructions and HMACs. The HMAC compute logic (HCL) reads instructions in 128-bit blocks and computes the HMAC corresponding to the current hash block. HCL is connected to the pipeline control and processes an instruction block only after all the instructions in the block are committed. This simplifies the handling of instructions that are speculatively issued and instructions issued in branch delay slot(s), when these may be conditionally nullified. HCL also interfaces to the L1 D-cache to save (restore) its internal state on context switches and interrupts (the dotted data line).

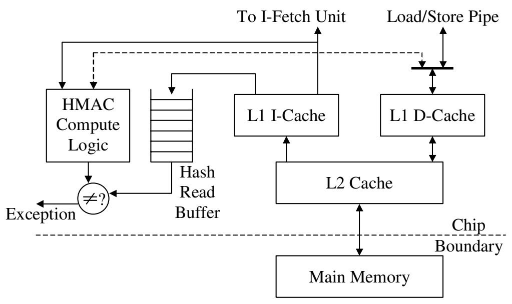


Figure 6: REM datapath

Concurrent to the HMAC computation, the stored HMAC of the current hash block is read from the memory and stored in the *first-in-first-out* (FIFO) hash read buffer. This buffer is necessary because the HMAC computation latency is longer than the HMAC lookup latency. The address of the HMAC corresponding to the current hash block can be computed simply by scaling the operand of the leading *hash_ptr* instruction, and then adding this value to the starting address of the HMAC appendix. When the HMAC computation is finished, it is compared to the stored HMAC, and an exception is raised if the values mismatch.

3.3. Security

REM can detect flow anomalies that occur *during* program execution. Consider the buffer overrun example in Figure 3 and assume that the infused malicious code attempts to spawn a command shell to execute arbitrary system commands. This normally requires using a flow control instruction to invoke the kernel mode (for example, the *int* instruction on x86 [22]). With REM, this raises an exception since there will be no HMAC(s) associated with the hash block(s) in the malicious code. Even though there will be a 20-cycle delay before this anomaly is detected, the exception will be raised before the kernel can transfer execution to the unauthorized command shell. Any attempt to skip legitimate instructions will also be detected with REM, since the HMACs are computed only on executed instructions.

To circumvent REM protection, an inside attacker (or the malicious code itself) may attempt to modify a program and also update the associated HMAC appendix. However, this requires recovering the REM key by breaking 128-bit AES encryption, which is not computationally feasible. While the default HMAC size in REM is 128 bits, 64-bit or 32-bit HMACs can also be used for storage-constrained or less security-critical systems. In this case, it may be possible to recover the REM key by examining installed code. Even in these instances, REM can still offer protection against automated attack tools on the network that cannot launch host-specific exploits.

We have so far assumed using a trusted operating system (OS) to prevent the REM key from being compromised during program installation. If the OS cannot be trusted, it is possible to guard the REM key by using a combination of public-key/symmetric-key encryption as in the XOM architecture [6]. The REM architecture does not provide memory integrity verification or program code confidentiality. If these functions are required, XOM or MH architectures may be used in combination with REM.

Table 1: Architectural parameters

Architectural Parameter	Value
L1 I-Cache	64 kB, 2-way, 32 B lines
L1 D-Cache	64 kB, 2-way, 32 B lines
L2 Cache (unified)	1 MB, 4-way, 64 B lines
L1 latency	1 cycles
L2 latency	10 cycle
Memory latency	100 cycle
Number of load/store pipes	2
Fetch/Decode/Issue/Commit Width	4
Number of Register Update Units	128
HMAC Latency	20 cycles (1 cycle pipelined)
Hash Read Buffer	16 entries

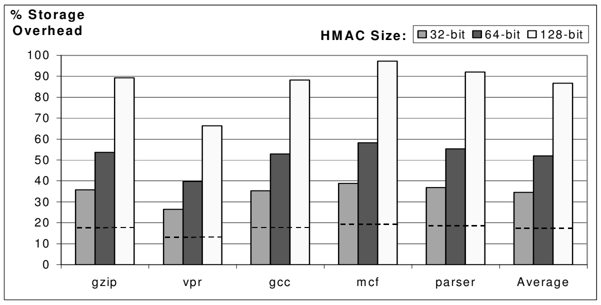


Figure 7: Code size increase in REM

4. Performance

To evaluate the performance impact of the REM architecture, we perform simulations on five SPEC 2000 integer benchmarks [23]. We use the SimpleScalar toolset, a cycle-accurate out-of-order superscalar processor simulator, configured for the PISA instruction set [24]. The default values for the architectural parameters we use are summarized in Table 1.

4.1. Code size

The REM architecture increases the program size due to the *hash_ptr* instructions and the appended HMACs. Figure 7 shows the total storage overhead when 32-bit, 64-bit, 128-bit HMACs are used. The dashed lines on the columns indicate the size increase due to the *hash_ptr* instructions, which is 17.3% on average. The total overhead is 34.6%, 51.9%, and 86.6% for 32-bit, 64-bit, and 128-bit HMACs respectively. While the 86.6% overhead of the 128-bit HMACs is high, the total size increase on a system may be limited by using REM only on vulnerable applications, such as webservers and mailservers that maintain continuous network connections. For resource-constrained environments, smaller 32-bit HMACs may be preferred to limit the total overhead to about 35%.

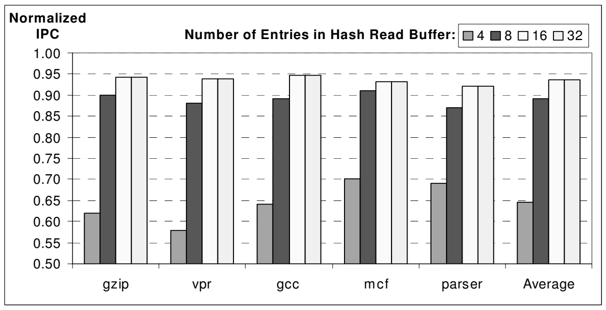


Figure 8: Impact of hash read buffer size

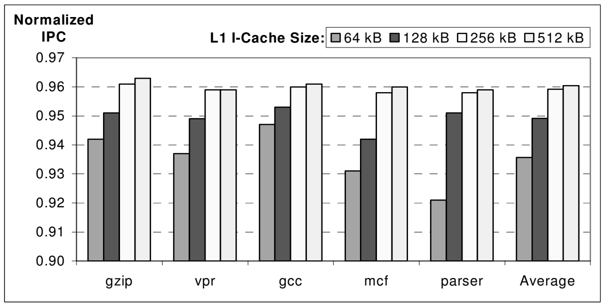


Figure 9: Impact of L1 I-cache size

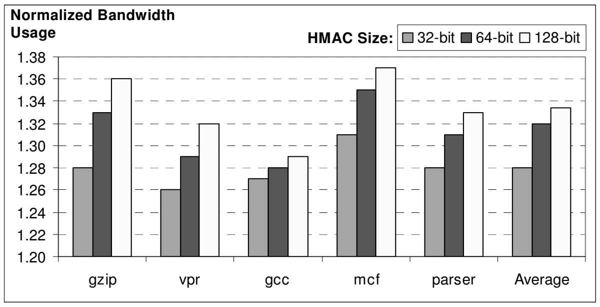


Figure 10: Off-chip bandwidth consumption

4.2. Hash read buffer size

The optimal size of the hash read buffer depends on the HMAC computation latency. In the worst case, a consecutive sequence of unconditional branch instructions is executed, thereby starting a new hash block each cycle. To avoid pipeline stalls in this case, an *n*-deep hash read buffer is necessary, where *n* is the HMAC computation latency. Because such an instruction sequence is rare, the actual read buffer size can be significantly smaller.

Figure 8 shows the performance degradation when hash buffers with 4, 8, 16, and 32 entries are used. A normalized IPC = 1 is the baseline performance of the benchmarks (i.e. without REM). With a 4-entry buffer, the performance degradation is significant for all benchmarks, averaging 35.4%. With an 8-entry buffer, the average degradation reduces to 11.0%. There is no measurable performance difference for buffers with 16 or 32 entries. The average slowdown for these cases is 6.4%.

4.3. Cache contention and memory bandwidth

Once a sufficiently large hash buffer is used, any performance degradation in REM, which averages 6.4% (Figure 8), is due to the additional memory accesses needed to bring the stored HMACs into the read buffer. The additional memory reads increase both the L1 I-cache contention and the off-chip bandwidth consumption. Figure 9 shows the impact of the former effect for different L1 I-cache sizes. When the I-cache size is increased to 128 kB and 256 kB, the average performance degradation reduces to 5.1% and 4.1% respectively. Figure 10 shows the increase in off-chip bandwidth consumption at different HMAC sizes. The average additional bandwidth consumption with REM using 128-bit HMACs is 33.4%. This reduces to 31.6% with 64-bit HMACs, and to 28.1% with 32-bit HMACs. The reduction is not proportional to the HMAC size because the contribution of the *hash_ptr* instructions to the bandwidth consumption is the same at all HMAC sizes.

5. Conclusions

We presented an architectural technique, which we call Runtime Execution Monitoring (REM), to detect program flow anomalies associated with unauthorized code execution on a computer system. The key idea in REM is the verification of program code at the hash block (basic block) level. This is achieved by pre-computing keyed hashes (HMACs) for each hash block during program installation, and then verifying these values during program execution. Significant implementation variables in REM are: (1) size of the HMACs, (2) algorithms used to compute the HMACs, (3) hash read buffer size, (4) L1 I-cache size. HMAC size depends on the desired security level of the system. While 128-bit HMACs may be needed for security-critical systems, even a smaller (32-bit) HMAC will provide protection against automated attack tools. While we have used the AES symmetric-key cipher to generate the HMACs, other symmetric-key or hash algorithms may also be used (e.g. SHA, MD5) [11].

The function of the hash read buffer is to temporarily store the looked-up HMACs until they are compared with the computed HMACs. We showed that a 16-entry read buffer is sufficient to eliminate all HMAC-related pipeline stalls when the HMAC computation latency is 20 cycles (single-cycle pipelined latency). A longer absolute

HMAC latency does not degrade performance as long as the read buffer is large enough. This is because the HMAC compute unit is not on the critical memory path. In this regard, REM differs from XOM, where the encryption unit degrades performance by increasing the memory access latency. Once a large-enough hash read buffer is used, performance degradation in REM is primarily due to the cache contention resulting from the shared use of the L1 I-cache to store both instructions and HMACs. The impact of this effect is 6.4% on average.

By verifying program code integrity at the hash block level, REM can monitor instructions whose behavior is typically exploited by malicious code, such as *branch*, *call*, *return* instructions. In this regard, REM differs from the previous works, XOM and memory hashing, which perform encryption/decryption and integrity verification on fixed-size memory blocks. Furthermore, because REM requires an explicit installation phase requiring an authorization password (the REM key) for each program, it also provides protection against other computer security threats linked to unauthorized code, such as viruses (excluding macro viruses), Trojan horses, and backdoor programs. While we do not advocate REM as a one-size-fits-all solution against all such malicious code, it can contribute significantly to computer security if it is employed as a new layer of defense concurrently with existing security tools.

References

[1] The SANS Institute, "The 20 Most Critical Internet Security Vulnerabilities", <http://www.sans.org/top20>.

[2] D.M. Kienzle and M.C. Elder, "Recent Worms: A Survey and Trends", *Proc. ACM Workshop on Rapid Malcode*, pp. 1-10, Oct. 2003.

[3] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer Worm", *IEEE Security and Privacy Magazine,* vol. 1, no. 4, pp. 33-39, Jul.-Aug. 2003.

[4] R.M. Best, "Preventing Software Piracy with Crypto-Microprocessors", *Proc. IEEE COMPCON,* pp. 466-469, Feb. 1980.

[5] T. Maude and D. Maude, "Hardware Protection Against Software Piracy", *Communications of the ACM,* vol. 27, no. 9, pp. 950-959, Sep. 1984.

[6] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software", *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 168-177, Nov. 2000.

[7] B. Gassend, G.E. Suh, D. Clarke, M.v. Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification", *Proc. Int. Symposium on High-Performance Computer Architecture (HPCA),* pp. 295-306, Feb. 2003.

[8] D. Lie, J. Mitchell, C.A. Thekkath, and M. Horowitz, "Specifying and Verifying Hardware for Tamper-Resistant Software", *Proc. IEEE Symposium on Security and Privacy (SP),* pp. 166-177, May 2003.

[9] J. Yang, Y. Zhang, and L. Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering", *Proc. Annual IEEE/ACM Int. Symposium on Microarchitecture (MICRO),* pp. 351-360, Dec. 2003.

[10] G.E. Suh, D. Clarke, B. Gassend, M.v. Dijk, S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors", *Proc. Annual IEEE/ACM Int. Symposium on Microarchitecture (MICRO),* pp. 339-350, Dec. 2003.

[11] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Oct. 1996.

[12] M. Frantzen and M. Shuey, "StackGhost: Hardware Facilitated Stack Protection", *Proc. USENIX Security Symposium*, Aug. 2001.

[13] L. Hornof and T. Jim, "Certifying Compilation and Run-Time Code Generation", *Proc. ACM Conf. Partial Evaluation and Semantics-Based Program Manipulation*, Jan. 1999.

[14] J.P. McGregor, D.K. Karig, Z.Shi, and Ruby B. Lee, "A Processor Architecture Defense against Buffer Overflow Attacks", *Proc. IEEE Int. Conf. Information Technology: Research and Education (ITRE)*, pp. 243-250, Aug. 2003.

[15] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, available at <http://www.amd.com>, 2003.

[16] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade", *Proc. DARPA Information Survivability Conference and Exposition (DISCEX),* vol. 2, pp. 119-129, Jan. 2000.

[17] W.W. Stames, "Integrity Assessment Tools: Fundamental Protection for Business Critical Systems, Data, and Applications", *Proc. Int. Conf. Information Technology Interfaces (ITI)*, pp. 465-470, Jun. 2000.

[18] C.C. Zou, L. Gao, W. Gong, and D. Towsley, "Monitoring and Early Warning for Internet Worms", *Proc. ACM Conf. Computer and Communication Security (CCS),* pp. 190-199, Oct. 2003.

[19] R.A. Kemmerer and G. Vigna, "Intrusion Detection: A Brief History and Overview", *Computer,* vol. 35, no.4, pp. 27-30, Apr. 2002.

[20] National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)", *FIPS Pub. 197*, <http://csrc.nist.gov/publications/fips>, Nov. 2001.

[21] H. Kuo and I. Verbauwhede, "Architectural Optimization for a 1.82 Gb/s VLSI Implementation of the AES Rijndael Algoritm", *Proc. Int. Workshop on Cryptographic Hardware and Embedded Systems (CHES), Lecture Notes in Computer Science,* vol. 2162, pp. 51-64, May. 2001.

[22] Intel, *IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, available at <http://www.intel.com>, 2004.

[23] Standard Performance Evaluation Corporation (SPEC), *SPEC CPU2000*, <http://www.spec.org/cpu2000>.

[24] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0", *Computer Architecture News*, pp. 13-25, Jun. 1997.