TEL AVIV UNIVERSITY
The Iby and Aladar Fleischman Faculty of Engineering

The Zandman-Slaner School of Graduate Studies

Security Applications for Hardware Performance Counters: Software Attestation and Random Generation

A thesis submitted toward the degree of
Master of Science in Electrical and Electronic Engineering

by
Eyal Ronen

February 4, 2012

TEL AVIV UNIVERSITY

The Iby and Aladar Fleischman Faculty of Engineering

The Zandman-Slaner School of Graduate Studies

Security Applications for Hardware Performance Counters: Software Attestation and Random Generation

A thesis submitted toward the degree of
Master of Science in Electrical and Electronic Engineering

by
Eyal Ronen

This research was carried out in the Department of Electrical Engineering -Systems under the supervision of Prof. Avishai Wool

February 4, 2012

Abstract

Performance monitors (also known as Hardware Performance counters, Perfmons or PMONs) are internal hardware counters built into the CPUs of the X86 family and other architecture families such as ARM and PowerPC. The PMONs store the counts of hardware-related activities within the CPU, and can be used to analyze different aspects of applications running on that hardware. PMONs were designed for advanced software performance analysis and this is their main usage.

Many events measured by PMONs are very hard to simulate accurately in software due to the high complexity of the monitored hardware operation (for instance, counting cache misses, or branch prediction errors). Moreover, the monitoring process is done by hardware with minimal software involvement. Thus, PMONs provide extremely precise counters of hard to predict hardware events and this makes them attractive for security applications.

The goal of this article is to provide a testing framework for different PMONs for two computer security applications: software attestation and true random generation. We use this framework to identify the best PMONs to use, and evaluate the performance that can be obtained.

Contents

List of Figures

Chapter 1

Introduction

1.1 Performance monitoring

Performance monitoring was introduced in the X86 family with the Pentium processor with a set of model-specific registers (MSRs) used as performance-monitoring counters. These counters permit a selection of processor performance parameters to be monitored and measured. According to Intel literature the information obtained from these counters can be used for tuning system and compiler performance[Int09]. Similar features are now available in the ARM [ARM] and PowerQUICC [Sem07] architectures.

The PMON events that can be measured vary among different Intel CPU types, and include CPU functions such as: CPU ticks, TLB misses, branch predictor misses, number of floating points operations and many more. There is a large variety of possible PMONs: e.g., the Intel Atom CPU offers 239 different PMON events to choose from.

We can divide the PMONs into 2 main families:

1. Architectural PMONs — PMON events that are guaranteed to provide fixed results on measurements of the same code over different machines of the architecture (e.g., all Atom CPUs with different memory, cache size etc.).

2. Micro-Architectural PMONs — PMON events that measure model specific functions and are dependent on specific hardware implementation of the CPU (e.g., PMONs that measure the behavior of the cache, branch predictor etc.).

Beyond their original use for system tuning, PMONs offer some unique advantages for security applications. This is because many of the events measured by PMONs are very hard to *accurately* simulate in software due to the high complexity of the hardware operation being monitored. In the paper we suggest to incorporate PMONs into two different security applications: software attestation and true random generation.

1.2 Software Attestation

1.2.1 Software Attestation Case Study - The Evil Maid Attack

The need for the ability to test whether a remote computer is indeed running directly on its designated hardware, using the software it was configured to run without any modifications, is well understood by now.

The "Evil Maid" attack described in [Ter10] is an effective method to circumvent most of today's full disk encryption (FDE) solutions. Although all of the user operating system and data is encrypted, the FDE program's loader and disk decryption code is saved in plain-text for the CPU to run in boot time. An "Evil maid" that has a physical access to a laptop, can easily modify its disk contents (e.g. an FDE loader code), leaving the laptop to the unsuspecting owner. Next time, when the password or a key will be provided by the owner, the code left by the attacker may silently record the decryption key and send it to the attacker. This is why it is essential for a FDE system to assure the user that the system that just booted is actually the system that he or she wanted to boot (i.e. the trusted one) and not some modified system (e.g. compromised by an MBR virus).

A boot time software attestation solution can help protect such FDE solutions.

1.2.2 Software Attestation Solutions

Over the years different solutions have been suggested for the basic problem of boot time software attestation. These can be divided into 2 main categories:

1. Hardware / Root of trust – solutions based on the assumption that the system includes a Root of trust, or dedicated security hardware such as a TPM. ([Tru03, SZJV04])

2. Software only attestation – ([KJ03, SLS$^+$05, SDGB11]). Such schemes typically include an entity outside the system (usually an authentication server) that sends a challenge to the tested system, verifies the response, measures the response time and uses this parameter to attest the software on the remote client. The main goal of the software attestation code is to provide a maximum-duration time penalty on a specific calculation to any code other than the original one. The input for the calculation is a specific challenge and the software code itself. All the solutions are based on the full knowledge of the system hardware. There are also a few proposals for generic attacks against software attestation schemes cf. ([SCT04, WvOS05]).

The basic underlining idea of software only attestation is to create a calculation that depends both on an input challenge, and the calculation code. The attacker's goal is to to replace or modify the code, while providing the same output in the same amount of time. The attestation code's goal is to force a time penalty as large as possible for any code change the attacker may use.

1.2.3 Using PMONs for Software Attestation

We argue that PMONs offer an attractive building block for software attestation code. By incorporating the results obtained from certain PMONs in the response, we force malicious code attempting to pass the test to simulate the PMONs. This is difficult to do both accurately and efficiently at the same time. If the attacker's simulation is not accurate it will produces a wrong output and will fail the verification. On the other hand, if the attack accurately computes the correct result it must work hard - which should noticeably slow the response time.

For this type of application the PMON value must be unpredictable (for different challenges) - but stable (should always return the same value for the same challenge).

The idea of using the CPU's meta-data for software attestation is not new in itself. E.g. Kennel and Jamieson [KJ03] first proposed using the translation look-aside buffer (TLB) in a software attestation code by reading the relevant data from the CPU MSR. They chose the TLB as it was affected by the attestation pseudo-random memory access, but was still possible to simulate by the authentication server. In contrast, we do not limit ourselves to the PMONs we can simulate but propose an attestation scheme that enables us to use any PMON that passes our tests.

Seshadri et al. [SLS+05] rejected the use of CPU meta-data, claiming that the reading time of the MSR is very long, and that the attacker can fully simulate the result of the read in less than the reading time. We argue that technological advances have now reduced the strength of this criticism: E.g., the newer "RDPMC" instruction can read a PMON value in around 50 cycles, rather than the $\sim$ 300 cycles using the older RDMSR commands.

Recently Weaver and Dongarra [WD10] explored the determinism of PMON's results. They looked for PMONs that can be predicted, are not micro-architectural and are not effected by a multitasking OS. We can ignore these constraints as our attestation scheme will run under native real mode, and we do not require the results to be predicable.

1.3 True random generation

1.3.1 True Random Generation Case Study - ASLR

Address space layout randomization (ASLR) is a computer security method which involves randomly arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, in a process's address space ([Wik]).

Using ASLR to protect the OS kernel requires a good source of entropy available at boot time. Without this entropy an attacker can guess the position of the kernel data areas.

A boot time True Random Generator solution can help to strengthen such ASLR solutions.

1.3.2 True Random Generation Solutions

All modern operating systems provide a standard API for a Pseudo Random Number Generator (PRNG) that is used mainly for cryptographic functions. In recent years faults were found in the randomness properties of the PRNG both in Windows and in Linux ([DGP07, GPR06]). Faults or wrong usage of the OS PRNG can lead to severe security breaches in the cryptographic services provided by those operating systems [USN]. Therefore modern OSes incorporate sources of "True Random" unpredictable events into their PRNG mechanisms.

Current sources for randomness are typically based on events outside the CPU, such as hard-disk ([SE05]), network or user activity ([ZLW+09]). Unfortunately these sources are not always available (e.g., in disk-less systems and different kinds of embedded systems), or may not be trustworthy ([GPR06]). The CPU clock is also sometimes sampled during the system run time for added entropy(Windows API CryptGenRandom() [LH02] and more), or as the sole entropy source of a random number generator[SS03]. As we shall see, there are PMONs that are much more effective then the system clock.

Intel has designed a hardware TRNG (True Random Numbers Generator) [SMR+10] to be incorporated in future CPUs, our approach worked on current of the shelf hardware.

1.3.3 Using PMONs for True Random Generation

Our aim is to explore the inert entropy stored inside the CPU internal state machine without the interference of the OS or any software apart from our attestation code. Unlike [SS03], we test this entropy directly under a non multitasking OS.

We argue that for this application too, PMONs are a useful tool since they may offer sources of unpredictability within the CPU, even when it is not attached to a hard-disk or a network. For true random generation we need different properties from the PMON's value: we need to have values with high variability and high entropy, i.e., PMONs that are unstable.

1.4 Contributions

In this article we study the feasibility and the added value of incorporating PMON reads into software attestation software, and for using PMON reads as a true random generator.

We also provide a testing framework for finding the best PMONs that can be used for such purposes. For every PMON we tested 2 main characteristics:

1. Stability: Does the PMON predictably return the same value for repeated calculations based on a fixed input.

2. Sensitivity: Does the PMON value vary when different inputs are used in the calculation. Sensitivity is quantified using the entropy (in bits) of the PMON value.

We created a code framework that performs a computation that depends both on an input challenge, and on the code's own machine instructions (and location in RAM). We used this code to exercise each of the PMONs, and analyzed the stability and sensitivity offered by each PMON.

The results are that for software attestation purposes, there exist several PMONs, that are stable yet provide a sensitivity of up to 6.6 entropy bits-per-sample assuming different challenge values. Incorporating repeated PMONs reads inside the attestation code accumulates entropy and forces an attacker to fully simulate the PMONs value to be able to correctly recreate the calculation.

Further, for true random generation purposes we identify several other PMONs, that are highly unstable, and provide entropy of up to 6.7 bit-per-sample for repeated samples starting with the same value. Our framework can extract randomness at a rate of 5.7 Mbps of real entropy—within the CPU itself, even on disk-less or embedded systems. Note that this rate can be increased even further by reading multiple PMONs simultaneously

In comparison to disk-based true random generation, our 5.7 Mbps raw entropy rate extraction rate compares very favorably to the 825 Kbps reported by [SE05]. Even if we follow the "paranoid" mode of [JSHJ98] and "whiten" the raw bits using [JJSH02] to produce cryptographically-strong unbiased random bits, we can still achieve a very high rate: If we distill a single bit out of every 30 8-bit measurements we obtain 29Kbps of strong random bits - in comparison to the 100 bits/minute with FFT for "whitening" [DIF94], or 577 bits/minute reported by [JSHJ98].

Organization: In the next section we describe our work environment and setup. Section 3 describes our basic PMON-using test code. Section 4 describes our experimentation with PMON-incorporating software attestation and Section 5 describes our true random generation. We conclude and suggest some future research directions in Section 6. Appendix A includes some technical details about using PMONs.

Chapter 2

Preliminaries

2.1 Software development environment

In order to obtain meaningful measurements using PMONs with minimal noise we needed a controlled environment without OS activity, and without multitasking. We tested a few OS choices with no multitasking, that still offered some basic running and debugging capabilities. We also looked for an environment that will be as similar to the boot process environment that our software attestation code would run in.

We decided to use Microsoft DOS 6.22 [Mic] for compiling, debugging and running our tests. DOS is a simple to use non-multitasking environment, that runs in real mode. We used DJGPP [djg], a DOS cross compiler version of the GCC compiler, with a DOS native Integrated Development Environment (IDE). Other environments we considered and rejected were Paradigm [par](a commercial environment for embedded system development), embedding our code in the Linux GRUB project[GRU], and native real mode programming.

2.2 Hardware Environment

All the tests were run on an Acer net-book with a N270 1.6 GHz Intel ATOM processor with 1 GB RAM. We chose this specific CPU since at the time it had the most advanced PMON architecture available and offered the possibility of measuring multiple PMONs at the same time. Furthermore, the processor has only a single core, which can reduce some of the complication to software attestation that result from the fact that different code can run on the different cores.

Chapter 3

Writing the basic test code

3.1 General

To test the possibility of using PMONs we constructed a simplified version of software attestation code, as done in [SLS$^+$05]. The input of the attestation code is a start seed (the challenge), and the output results from incorporating the initial seed with a hash of the software code.

Our version was designed for the purpose of exercising PMONs and testing their added value for attestation. To simplify the code writing process we used only 32 bit registers, and did not take into account different attestation code requirements (e.g., code optimizing size, run time, using all of the CPU registers, etc.) as they are not relevant to our goal. Before every PMON measurement we set it to 0 before running the algorithm, and measure it at the end of the run.

3.2 The Attestation code base

The code we used (see Algorithm 3.1) is a simplified version of the general design of [SLS$^+$05]. The code starts with the input challenge as a starting point, and performs N-ITER iterations. The attestation code is composed of 4 code blocks and a random traversing code. We wrote the code in AT&T assembly language—for clarity Algorithm 3.1 shows pseudo-code for one PMON read. The code has two goals:

1. To ensure that the attestation code itself, and the surrounding application have not been tampered with or moved to a new RAM location.

2. To exercise the PMONs.

Self-validation of the code is achieved by sampling the instructions stored in a pseudo-random sequence of locations, and hashing them into the current hash value (together with the iteration counter).

Algorithm 3.1 Attestation Code

```
h <- Challenge //Storing the challenge into the current hash value
instr <- p_instr <- 0
Initialize PMON //Start  of PMON exercise
for (iter = N_ITER; iter > 0; iter--)
     switch(h%4) //Pseudo random data depended jump
     {
        case 0:

            //Adds the side effect of incorporating currently
            //measured PMON value
            register <- current PMON value
            h <- h ⊕ (register & UNSTABLE_BITS_MASK)
            //Code validation
            p_instr <- memory base + h % code_size
            instr <- *p_instr
            h <- rotate_left(h ⊕ instr , 17) ⊕ iter
            break
        case 1:

            //Adds the side effect of an extra integer division
            h <- h ⊕ instr % code_size
            //Code validation
            p_instr <- memory base + h % code_size
            instr <- *p_instr
            h <- rotate_left(h ⊕ instr , 17) ⊕ iter
            break
        case 2:

            //Adds the side effect of a floating point multiplication
            h <- h ⊕ ((float)instr * (float)p_instr)
            //Code validation
            p_instr <- memory base + h % code_size
            instr <- *p_instr
            h <- rotate_left(h ⊕ instr , 17) ⊕ iter
            break
        case 3:
            //Adds the effect of an extra rotation
            h <- rotate_left(h,17)
            //Code validation
            p_instr <- memory base + h % code_size
            instr <- *p_instr
            h <- rotate_left(h ⊕ instr , 17) ⊕ iter
            break
     }
Challenge response <- h;
Read PMON; // For our testing purposes only
```

To exercise the PMONs we use a data-dependent jump to one of 4 possible blocks. As the jump is based on the 2 LSB of the current hash, it is pseudo random. Inside each block, specific instructions are used to exercise the PMON and to incorporate the current value of the PMON into the current hash.

The attestation code exercise the PMONs in several ways:

1. The code duplicates the basic validation code block 4 times rather than keeping the instructions outside the switch statement. The pseudo random code jumps among the validation code blocks affect the branch predictor and pipeline.

2. Pseudo random memory accesses affect the cache mechanism.

3. Different instructions are used in each code block.

As shown in [SLS$^+$05], for the attestation code to cover the entire attested code with high probability it should be run at least $N \log N$ iterations when N is the code size. As we propose incorporating the PMONs inside the iteration loop throughout the attestation process we wanted to study the effect that a small number of iterations has on PMONs measurements. For that reason we set N-ITER to be a number much smaller than $N \log N$. In this way each run of our code represents a a small part of the attestation process. By measuring the PMON over this small part we can accurately evaluate the added entropy of repeated PMON measurements have on the attestation process.

As we shall see in Section 5 we used the same code for random generation too. For the purpose of true random generation we chose to use a small number of iterations to increase the overall rate of the true random generator.

Initial testing showed us that it is important to set N-ITER, the number of iterations to a value that is relatively prime to the number of codes block, which is 4. Therefore for our tests of both applications we set N-ITER to 11. Note that the value of of N-ITER should not be confused with the value N (the size of code) mentioned above. For a full attestation our code would need to be executed N/N-ITER times.

Chapter 4

PMON-incorporating Attestation

4.1 Overview

As described by [SLS$^+$05] an attestation system includes the following:

1. Attestation Client - This is the system being authenticated. In our approach the client attestation software uses the PMONs.
2. Attestation Server - The server needs to be able to produce random challenges, send and receive messages, and accurately time the authentication process.

This scheme can be applied to scenarios such as validation of a disk encryption program by a Smart Card or USB token before delivering encryption keys; Validation of ATM software before connection to the central banking system; or testing an embedded system for evidence of tampering.

Our goal is to use PMONs into our calculation in a way that will require an attacker to waste a long time in either simulating them, or hide the effect is changed code had on them.

4.1.1 The Learning phase

Since the attestation server will not be able to simulate or execute the client's attestation code locally, we propose a learning phase that should be executed while the client is in a known safe configuration - e.g., before it leaves the factory. The learning phase works as follows:

1. Repeat for a predetermined number of times
   (a) The server sends a challenge to the client.
   (b) The client performs the attestation calculation and sends back the result to the server.
   (c) The server records the challenge, the result, and the time delay into a "Code Book".

4.1.2 The Operations phase

There are many possible variants of how the operations phase may work. As this is not the main topic of our work we describe one of the simpler choices.

Once the client is fielded the attestation needs to be run immediately on client startup. One possible mechanism is as follows:

1. The server sends a recoded challenge (from the "Code Book") to the client.
2. The client performs the attestation calculation and sends back the result to the server.
3. The server compares the result and time delay with the records in the "Code Book":
    (a) If the result is the same and the time delay is within an acceptable margin, the client is verified.
    (b) If not the server retries with the next recorded challenge for a predetermined number of retries (or possibly 0 retries). After which the verification fails, and the clients needs to be reinstalled.
4. If the client is verified one or more challenges are sent by the server, to replace the used challenges in the "Code Book".

An alternative is to send multiple challenges and require the client to respond correctly to *all* of them, and once it is verified - refresh the "Code Book" with multiple new challenges. Selecting the number of rounds that are required to validate the client, and the number of challenge-response tuples stored in the Code Book, are left for future work.

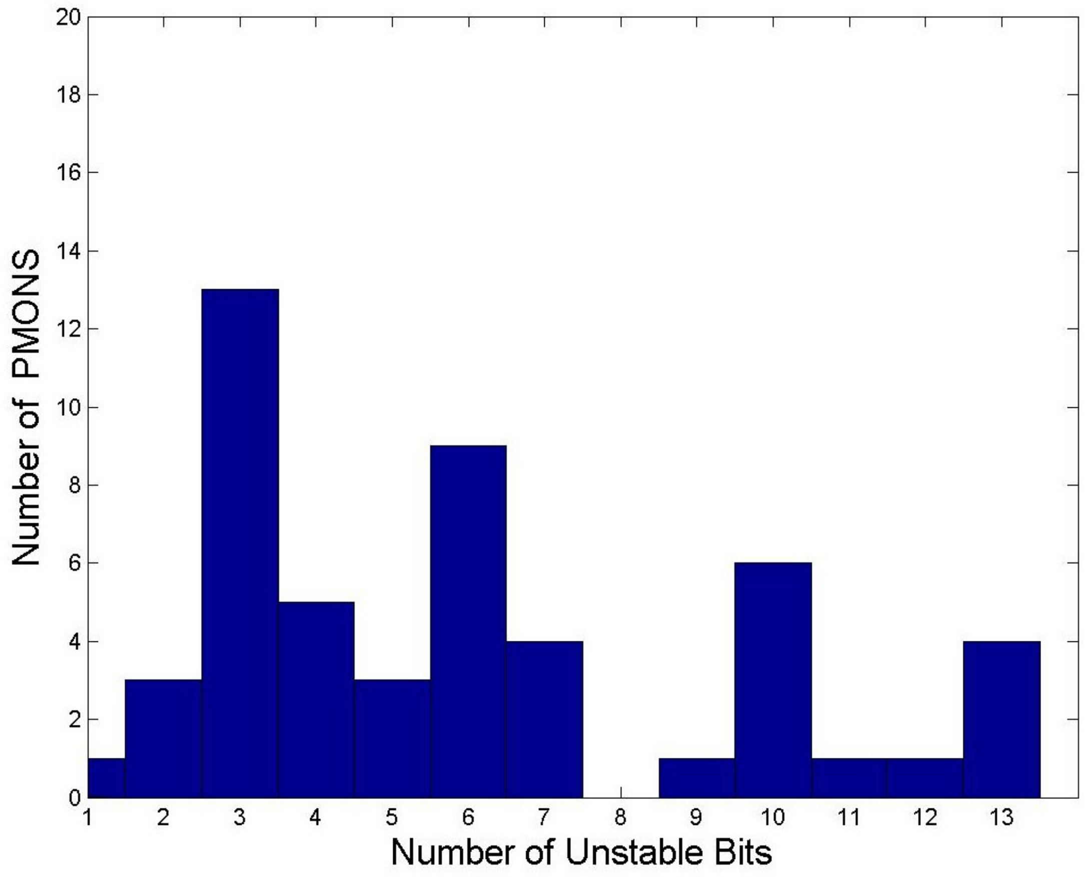
4.2 PMONs Stability measurements

In order to use PMONs in such an attestation framework we need PMONs that are Stable (return the same value in repeated runs using the same input) but Sensitive (return different values for different inputs).

To evaluate PMON stability we repeatedly measured each PMON with the same start seed (h = 1234567) 150 times. We discovered that even in the most controlled environment, and running the same code over the same input, there was some instability in many PMONs. In order to quantify this instability, and to identify the most stable PMONs, we determined the number of Least Significant Bits (LSBs) that change between measurements. If used for attestation such bits need to be zeroed out before the measurement is incorporated into the calculation to ensure the stability of the results.

We discovered that 188 of the 239 PMONs always returned a fixed value and require no bits to be masked out. In Figure 4.1 we can see the distribution of unstable bits per

Figure 4.1: Unstable Bits per PMON



PMON for the other 51 unstable PMONs. The figure shows that some of the PMONs exhibit up to 13 bits of instability.

4.3 Finding usable bit entropy for attestation PMONs measurements

Once we quantified the number of unstable bits in each PMON, we could measure the sensitivity to the input. We repeatedly measured each PMON with 150 different start seeds. To be able to use the PMON for attestation we need to get different results while measuring with different seeds. But we need to take into account only the bits that were shown to be stable. Thus we used our previous results, to identify and to mask out the unstable bits.

To quantify the remaining sensitivity to the input we used the entropy function $H(x) = -\sum p_i \log_2 p_i$ where p_i is the frequency of value i in the sample. After masking out the unstable bits from Section 4.2 we checked the entropy left in the measurement's next 8 bits.

In Figure 4.2 we can see the distribution of entropy bits per PMON. After masking out the unstable bits as appropriate all of the PMONs were completely stable. The figure shows that several PMONs exhibited significant sensitivity to the input, while remaining stable when used on fixed input.

4.4 Taking a closer look at the "good" PMONs

We decided to take a closer look at the PMONs with usable attestation entropy. To do so we considered PMONs exhibiting at least 0.4 usable entropy bits.

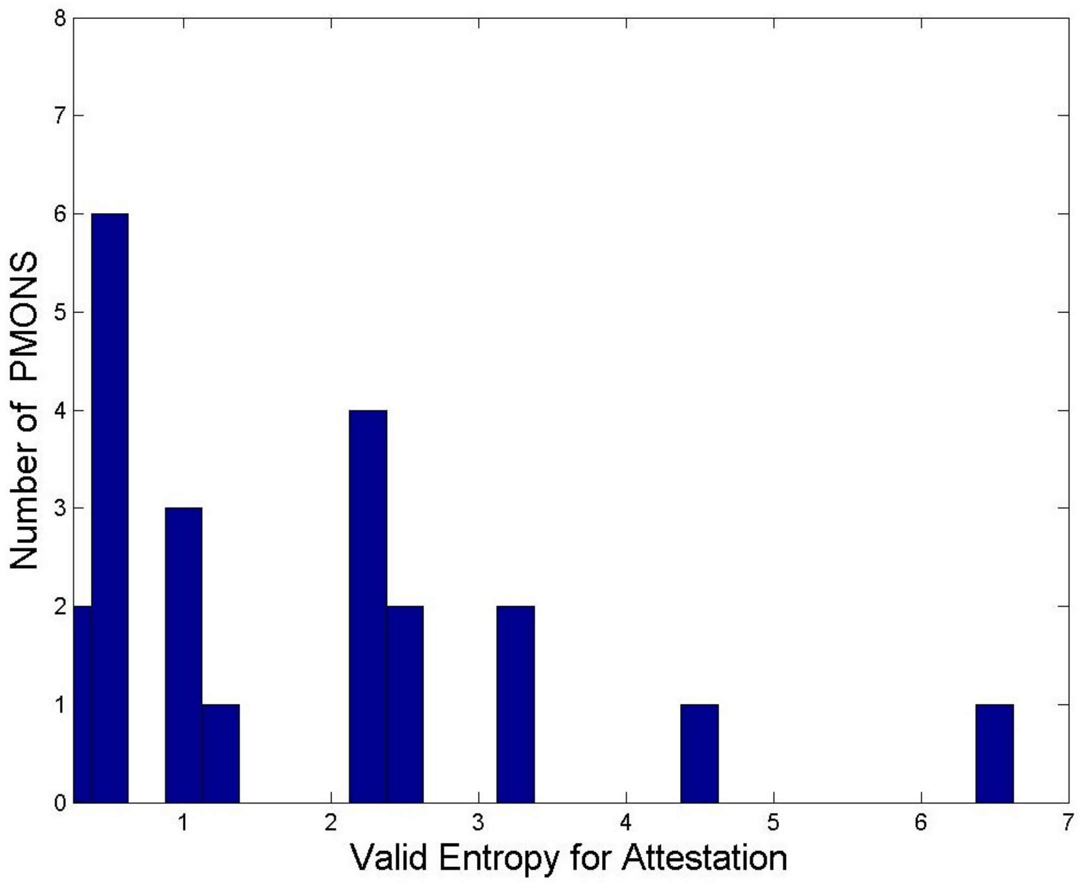
As we shall see all of the completely stable PMONs are architectural ones. The micro architectural PMONs are generally less stable, and require some bits to masked out.

We observed that the usable PMONs can be divided into a few categories showing the PMONs events with high entropy. For further information about each PMON see [Int09].

Micro-Ops related The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences and in some cases micro-op sequences are fused or whole instructions are fused into one micro-op[Int09]. The number of micro ops retired per measurement is very stable for a fixed input, yet the value is highly sensitive to different inputs. The best PMONs in this category are:

1. UOPS-RETIRED.ANY - Number of micro-ops the CPU retired. This is the PMON with the highest usable entropy overall (6.6 bits-per-sample with 0 unstable bits

Figure 4.2: Valid Entropy Bits per PMON



masked). This PMON event is architectural and depends solely on the code. In our code each code block comprises of a fixed number of micro-ops. Any instructions added or removed by the attacker will affect and change this PMON value. An attacker that wishes to provide the same result as the original calculation has to either fully simulate the PMON (adding extra instructions per block), or “fix” the PMON value after running any added code. Both options require the attacker to add extra instructions, while the legitimate code only incurs the time it takes to read the PMON. To use this PMON in attestation the right number of iterations between PMON reads needs to be set so that the estimated “simulation” time will be more than the time it takes to read the PMON.

2. UOPS-RETIRED.STALLS - Periods no micro-ops retired (4.6 bits-per-sample with 1 unstable bits masked). Unlike UOPS-RETIRED.ANY this PMON event is micro-architectural and can be relatively hard to simulate. Across different machines the number of unstable bits may increase. On machines like ours that give good “attestation” results this PMON is one of our the best candidates as it is hard to simulate and will record every instruction added to the code.

Branch related Super scalar CPUs can process several commands at the same time, and use a pipeline to prefetch the next commands from memory. The branch predictor is a complicated hardware module that tries to anticipate the results of code branching, and prefetch the following commands into the pipeline and even execute commands before it is certain if they will be needed or not. There are many PMONs dedicated to analyze issues related to the branches and branch predictor. As the branching in our attestation code is very hard to predict, several branch related PMONs pass our testing paradigm. The best PMONs in this category are:

1. INSTRUCTIONS-RETIRED - Number of CPU instruction retired (3.3 entropy bits-per-sample with 0 unstable bits masked) . This is also an architectural PMON and its result is linear with the number of instructions in the code. Each code block adds a fixed value to this PMON.

2. BR-INST-RETIRED - Number of branch instructions retired by the CPU (2.2 entropy bits-per-sample with 0 unstable bits masked). An architectural PMON that is effected by the code ran directly after each branch, and can help in identifying added branches in the code.

3. BR-INST-DECODED - Number of branch instructions decoded, including instructions for mis-predicted branches (1 entropy bits-per-sample with 2 unstable bits masked). This is a micro architectural PMON, that is hard to simulate and may discover added code in branches either taken or not taken.

Memory related Modern CPUs have a very complicated memory cache algorithm to optimize work with memory. There are several PMON events dedicated to analyzing memory usage issues. As the memory access in our attestation code is pseudo random with respect to different seeds, such PMONs perform well in our testing. The best PMONs in this category are:

1. LAST-LEVEL-CACHE-REFERENCES - number of references to L2 cache (2.2 entropy bits-per-sample with 0 unstable bits masked).
2. MEM-LOAD-RETIRED.L2-HIT - number of retired loads that hit the L2 cache (1.1 entropy bits-per-sample with 0 unstable bits masked).

Both of those PMONs are micro architectural and are affected only by the way we access the memory (and for that reason are very stable for fixed inputs). Both of these events are complicated to simulate, in particular the first one as it includes memory requests from mis-predicted branches.

Mathematical calculation related Our attestation code has 2 blocks that perform mathematical calculations: one integer DIV and one floating point MUL. Because both the values that are given to the calculation and the number of times it is done (the number of times the block is run) differ between different runs this allows us to use the following PMONs:

1. DIV - Divide operations executed (2.4 entropy bits-per-sample with 0 unstable bits masked).
2. X87-COMP-OPS-EXE - Floating point computational micro-ops executed (2.4 entropy bits-per-sample with 0 unstable bits masked).

These PMONs are architectural and their value is linear with the number of the mathematical operation they count.

Chapter 5

Random Bits Generation

5.1 Overview

The second security application we propose using PMONs in is true random generation. Recall that a good PRNG needs two parts:

1. A good seed with enough true entropy.
2. A good cryptographic model to produce a long sequence of random numbers from the seed.

There are many tests for the quality of output of a PRNG[PUB01, Mar96]. Our goal is to provide the seed of the process, and the seed quality is determined by the level of entropy of the seed.

To evaluate the best PMONs we again used the code in Algorithm 3.1. Since there is no external challenge we ran the code with a fixed starting point h = 1234567. Further, since we do not require any stability (on the contrary the less stable PMONs are better), we did not zero out any PMON's bits.

We wrote the testing framework in a way that will simulate our test case goal. Our code was programmed to run automatically at boot time, measure the PMON's value, and then restart the computer before the next run.

5.2 Single Read Entropy Test

For the random bits test we repeatedly measured each PMON with the same start seed 1024 times.

For each PMON the number of random bits in the measurement was calculated with the entropy function $H(x) = -\sum p_i \log_2 p_i$ on the LSB byte of the PMON results (after initial evaluation we saw that the entropy is in the LSB byte only).

As before, of the initial 239 PMONs 51 PMONs had entropy higher then 0. The entropy level in those PMONs is shown in Figure 5.1. The figure shows that several PMONs display

Figure 5.1: Distribution of Entropy bits in PMONs

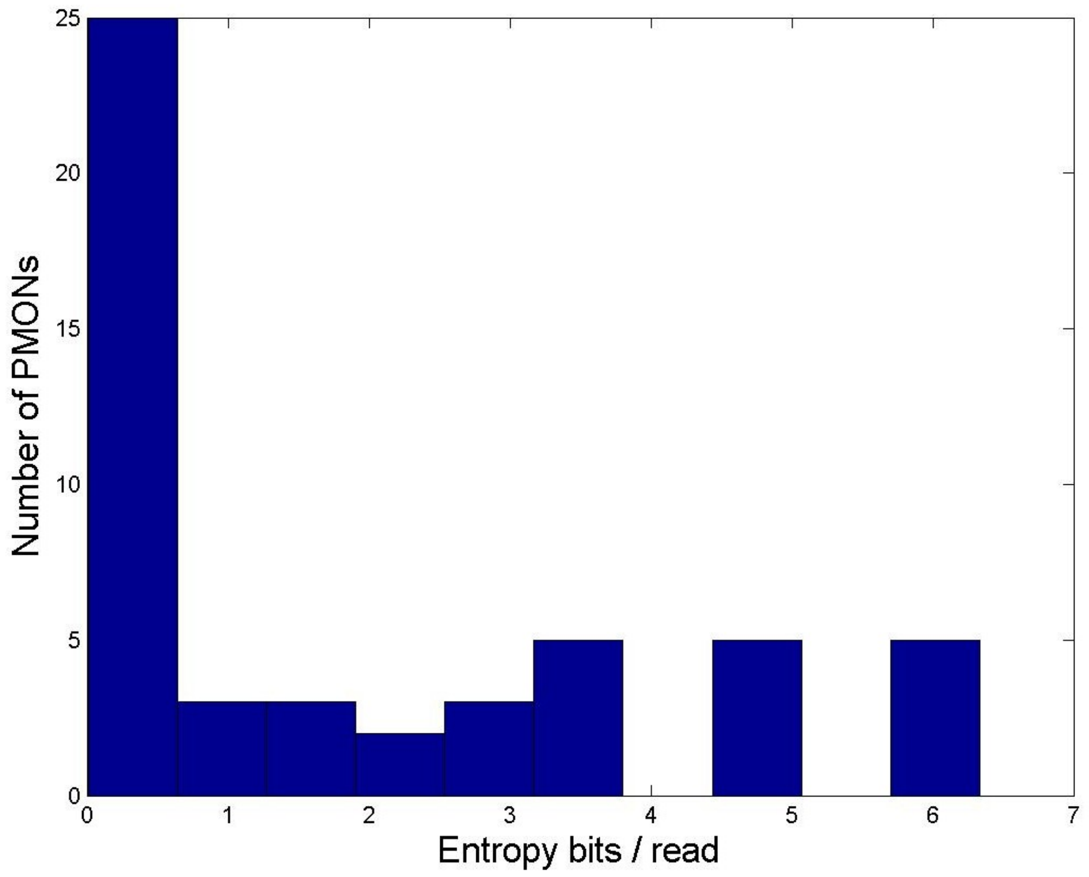
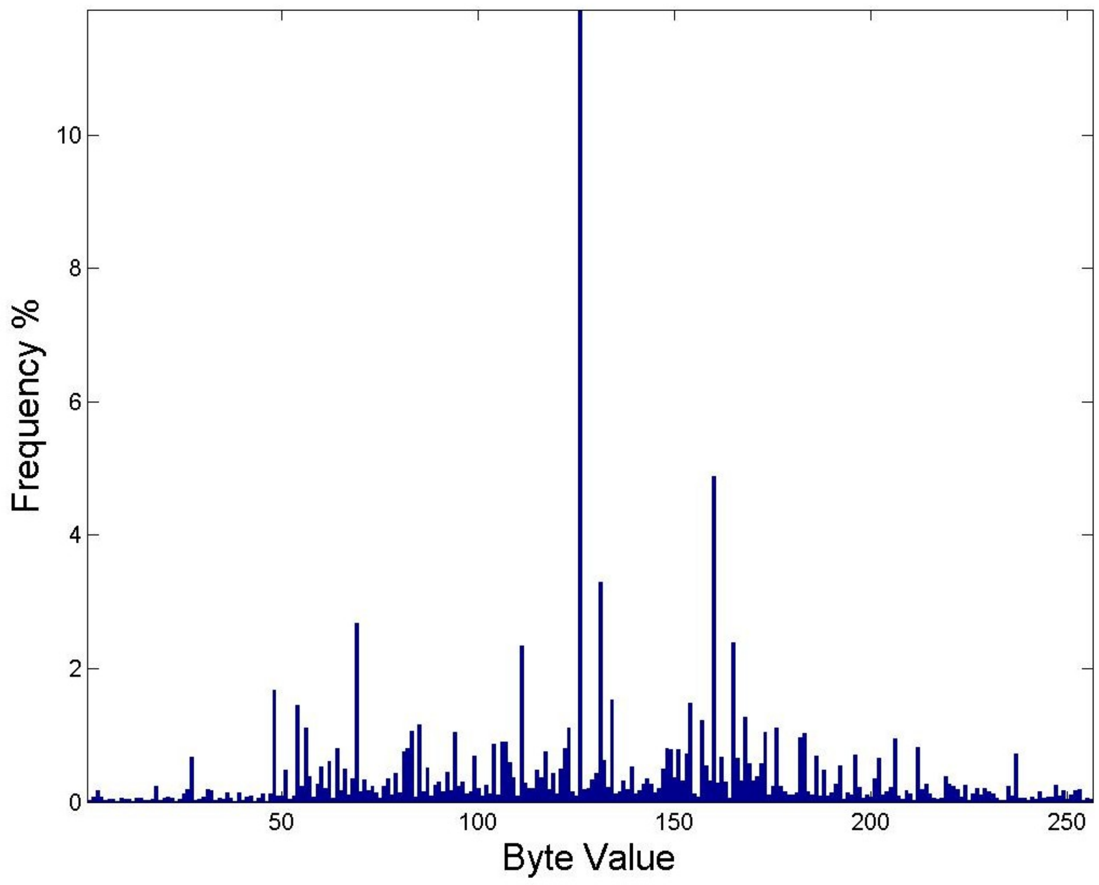


Figure 5.2: Value frequencies of the maximum entropy PMON



high instability despite the fact that exactly the same code is executed each time, with same input. The best performing PMONs exhibit more than 6 entropy bits-per-sample.

To obtain further insight we concentrated on the PMON with the highest entropy (UOPS-RETIRED-STALLED-CYCLES). For better accuracy we reran our test with 80000 repeated measurements and received 6.7 entropy bits-per-sample.

The byte value distribution of reads of the maximum entropy PMON (UOPS-RETIRED-STALLED-CYCLES) is shown in Figure 5.2. The figure shows that there is a value (125) that accounts for 12% of the distribution, the remaining 88% of the distribution are spread out quite uniformly across the values 0-255.

5.3 Taking a closer look at the “good” PMONs

CPU Stalled related The number of cycles in which the CPU was stalled waiting for new instructions depends on the timing of many hardware components in the CPU (e.g., the cache system, the processor pipeline, external memory both volatile and nonvolatile, the branch predictor and more) and their complex interaction from the boot time to the start of the code run. It is to be expected that small jitters in the timing of those components (that can be induced by physical noise like internal clock jitter due to thermal changes, or even jitters in the wakeup time of different system components) will have large effect on the measurement. The PMONs that are related to these events are micro architectural PMONs. Indeed the PMONs related to CPU stalling have the highest entropy we measured:

1. UOPS-RETIRED-STALLED-CYCLES - Cycles no micro-ops retired (6.7 bit-per-sample).

2. L2-NO-REQ - Cycles no L2 cache requests are pending (6 bit-per-sample).

3. CPU-CLK-UNHALTED - Core cycles when core is not halted (6.3 bit-per-sample).

5.4 Testing the PMON inter-read independence

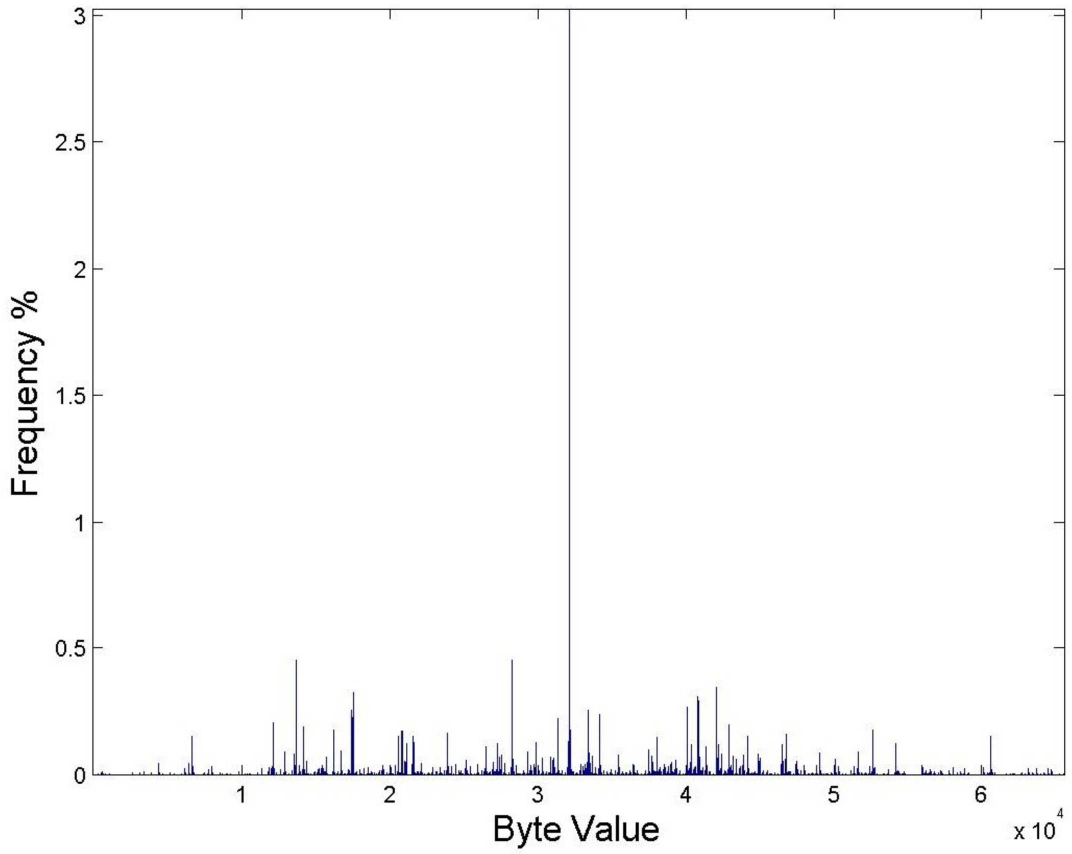
We also wanted to evaluate whether consecutive reads from the same PMON were statistically correlated. To do so we examined the joint entropy of consecutive read results.

It is well known that If X and Y are random variables, then $H(X, Y) \leq H(X) + H(Y)$, with equality if and only if X and Y are independent. So to test for the independence of each specific read, we tested the entropy on the 2 bytes made by the concatenation of the LSB of each read sample with the LSB of the next sample. We assume that if the entropy of the concatenated pairs is close to double the single byte entropy then w.h.p the results are i.i.d in nature.

We re-analyzed the data of Section 5.2 for the max entropy PMON (UOPS-RETIRED-STALLED-CYCLES). We found an entropy of 12.7 bits-per-pair of joint 2-byte values from consecutive reads, which is very close to the predicted value of $6.7*2 = 13.4$ bits-per-pair assuming i.i.d values.

The two-byte value distribution of the concatenated reads of the maximum entropy PMON is shown in Figure 5.3. The figure shows that there is a value that accounts for 3% of the distribution that is $32125 = 256*125 + 125$ — i.e., both samples had a value of 125. We saw that this was the most popular value in Section 5.2, with a frequency of 12%, so a joint distribution of $Pr(125,125) = 0.03$ is reasonably close to the predicated value of $0.12*0.12=0.0144$. Thus we see that both from the overall joint entropy, and from the behavior of the most popular value, consecutive reads are fairly independent.

Figure 5.3: Value frequencies of the concatenated consecutive two-bytes samples of the maximum entropy PMON



5.5 Random Bits Generation Rate

The maximum entropy of 6.7 bits per read was found in the UOPS-RETIRED-STALLED-CYCLES PMON.

Each PMON read took about 2000 CPU ticks that on our 1.6 GHz CPU means a sample rate of about 0.86 MHz. Therefore, using the PMON showing the maximum entropy we can get to up to $6.7_{bit} * 0.86_{Mhz} \sim 5.7_{MBit/s}$ of entropy. Considering the fact that we can measure more then 1 PMON simultaneously the maximum rate may be much higher. Even if we follow the "paranoid" mode of [JSHJ98] and "whiten" the bits by XORing all the bits of 30 8-bit samples into a single bit, we can still achieve a rate of $1_{bit} * 0.86_{Mhz}/30 \sim 29_{KBit/s}$: 3 orders of magnitude faster then their disk-based approach.

Chapter 6

Conclusion

Both random number generation and software attestation are key problems in today's personal computer security. PMONs can be used to help solve those problems, using hardware that is already available on all modern computers. Moreover further research can help CPU manufactures to add PMONs that can be used for those purposes with little or no extra cost to end users.

We have demonstrated that PMONs are an effective tool for both security applications.

We have shown that we can find PMONs that are suitable for incorporating in attestation code, and presented some PMON families that show good results. We have also shown PMON families that are good source of entropy for true random generation, without the need for any external hardware and at very fast rates.

We believe that PMONs offer an interesting topic for further work, with extended research on the following points:

1. The effect of different program code size and memory area on PMONs related to memory functions. E.g., running attestation code on a program that extends over more than the memory cache size.

2. Testing the specific PMONs that were found suitable for attestation against known generic side attacks on software attestation. PMONs that are proven to be affected by such attacks could provide a good defense.

3. Finding the best the number of block iterations between PMONs reads. The trade off should be between the PMON reading time and the added unpredictability for the attestation process.

4. Today's modern embedded processor families like ARM and Power-PC also include performance monitor capabilities similar to those found in x86 architecture. Further work on those processors can expand our work into embedded device world.

5. Although we believe that the entropy measured is due to the asynchronous nature of different hardware components in the CPU (such as instruction pipe line, memory

controller, branch predictor etc.), further research to the exact sources of randomness is needed to better and evaluate the quality of our TRNG.

Acknowledgments

We would like to thank Ronny Ronen from Intel Corporation for his valuable feedback on this paper and his insight on PMONs and the different events they measure.

Bibliography

[ARM] ARM. Arm architecture reference manual performance monitors, v2 supplement, http://infocenter.arm.com/help/, 2010.

[DGP07] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the windows random number generator. In *Proceedings of the 14th ACM conference on Computer and communications security*, page 485. ACM, 2007.

[DIF94] D. Davis, R. Ihaka, and P. Fenstermacher. Cryptographic randomness from air turbulence in disk drives. In *Advances in Cryptology, Crypto94*, pages 114–120. Springer, 1994.

[djg] Djgpp, http://www.delorie.com/djgpp/.

[GPR06] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. 2006.

[GRU] Gnu grub's, http://www.gnu.org/software/grub/.

[Int09] Intel. *Intel Architecture Software Developer's Manual 3*. Intel, 2009.

[JJSH02] A. Juels, M. Jakobsson, E. Shriver, and B.K. Hillyer. How to turn loaded dice into fair coins. *Information Theory, IEEE Transactions on*, 46(3):911–921, 2002.

[JSHJ98] M. Jakobsson, E. Shriver, B.K. Hillyer, and A. Juels. A practical secure physical random bit generator. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 103–111. ACM, 1998.

[KJ03] R. Kennell and L.H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, page 21. USENIX Association, 2003.

[LH02] David LeBlanc and Michael Howard. *Writing Secure Code*. Second edition edition, 2002.

[Mar96] G. Marsaglia. DIEHARD: a battery of tests of randomness. *See http://stat.fsu.edu/~geo/diehard.html*, 1996.

[Mic] Microsoft, http://www.microsoft.com.

[par] Paradigm C++ Professional, http://www.devtools.com/default.htm.

[PUB01] NIST PUB. 140-2: Security Requirements For Cryptographic Modules, 2001.

[SCT04] Umesh Shankar, Monica Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *USENIX Security Symposium*, pages 89–102. USENIX, 2004.

[SDGB11] R. Srinivasan, P. Dasgupta, T. Gohad, and A. Bhattacharya. Determining the Integrity of Application Binaries on Unsecure Legacy Machines Using Software Based Remote Attestation. *Information Systems Security*, pages 66–80, 2011.

[SE05] E. Schreck and W. Ertel. Disk drive generates high speed real random numbers. *Microsystem Technologies*, 11(8):616–622, 2005.

[Sem07] Freescale Semiconductor. Performance Monitor on PowerQUICC II Pro Processors, www.freescale.com, 2007.

[SLS$^+$05] 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. *ACM SIGOPS Operating Systems Review*, 39(5):1–16, 2005.

[SMR$^+$10] S. Srinivasan, S. Mathew, R. Ramanarayanan, F. Sheikh, M. Anders, H. Kaul, V. Erraguntla, R. Krishnamurthy, and G. Taylor. 2.4ghz 7mw all-digital pvt-variation tolerant true random number generator in 45nm cmos. In *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, pages 203 –204, june 2010.

[SS03] A. Seznec and N. Sendrier. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Transactions on Modeling and Computer Simulation*, 13(4):334–346, 2003.

[SZJV04] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, page 16. USENIX Association, 2004.

[Ter10] Alexander Tereshkin. Evil maid goes after pgp whole disk encryption. In *Proceedings of the 3rd international conference on Security of information and networks*, SIN '10, pages 2–2, New York, NY, USA, 2010. ACM.

[Tru03] *Trusted Computing Group (TCG) https://www.trustedcomputinggroup.org/*, 2003.

[USN] Ubuntu security notice usn-612-1 may 13, 2008 openssl vulnerability cve-2008-0166.

[WD10] V. Weaver and J. Dongarra. Can hardware performance counters produce expected, deterministic results? In *3rd Workshop on Functionality of Hardware Performance Monitoring*, December 2010.

[Wik] Wikipedia. Address space layout randomization wikipedia page. http://en.wikipedia.org/wiki/ASLR.

[WvOS05] G. Wurster, PC van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, pages 127–138. IEEE, 2005.

[ZLW+09] Q. Zhou, X. Liao, K. Wong, Y. Hu, and D. Xiao. True random number generator based on mouse movement and chaotic hash function. *Information Sciences*, 179(19):3442–3450, 2009.

Appendix A

Using the CPU PMONs

Intel CPUs organize the PMONs into 2 classes, as follows:

1. Fixed type PMONs - a set of PMONs that can measure only one specific event each, predefined by Intel.
2. General Purpose (GP) PMONs - a set of PMONs that can be defined separately to measure any event(architectural or not) as chosen by the user.

Intel processors allow activating several PMONs at the same time. For example, the ATOM CPU allows using 5 PMONs simultaneously: 3 fixed and 2 GP.

The PMONs measurement process includes 3 steps:

1. Initial Setup - This is done by writing specific values to the PMONs control MSRs using the “WRMSR” command. The relevant MSRs are :

   (a) IA32-PERFEVTSELx - a 64 bit control register for each GP PMON (where ’x’ can be 0..(Number of GP PMONs - 1)).

   (b) IA32-FIXED-CTR-CTRL - one 64 bit control register for all fixed PMONs.

   For example: Setting the IA32-PERFEVTSEL1 MSR to 0x00004305 starts measuring PMON event number 5 (BRANCH-INSTRUCTIONS-RETIRED).

2. Reading\Initializing starting value - This is done by either writing specific values, or reading the start values of the PMONs counter MSRs using the “WRMSR” \ “RDMSR” commands:

   (a) IA32-PMCx - a 64 bit counter register for each GP PMON(where ’x’ can be 0..(Number of GP PMONs - 1)).

   (b) IA32-FIXED-CTRx- 64 bit counter register for each Fixed PMON (where ’x’ can be 0..Number of Fixed PMONs - 1).

3. Reading the final result - this is done by either reading the counter MSR using “RDMSR” or the PMON specific “RDPMC” command.

The specific values and different configuration options, and a full description of available PMONs events that can be measured in each type of Intel CPU can be found at [Int09].