Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs?

Corey Malone
ECE Department
Polytechnic Institute of NYU
New York, NY
cmalon01@students.poly.edu

Mohamed Zahran
ECE Department
Polytechnic Institute of NYU
New York, NY
mzahran@acm.org

Ramesh Karri
ECE Department
Polytechnic Institute of NYU
New York, NY
rkarri@poly.edu

ABSTRACT

In this paper, we propose to use hardware performance counters (HPC) to detect malicious program modifications at load time (static) and at runtime (dynamic). HPC have been used for program characterization and testing, system testing and performance evaluation, and as side channels. We propose to use HPCs for static and dynamic integrity checking of programs.. The main advantage of HPC-based integrity checking is that it is almost free in terms of hardware cost; HPCs are built into almost all processors. The runtime performance overhead is minimal because we use the operating system for integrity checking, which is called anyway for process scheduling and other interrupts. Our preliminary results confirm that HPC very efficiently detect program modifications with very low cost.

Categories and Subject Descriptors

K.6.5 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Security and Protection

General Terms

Security

Keywords

hardware performance counters, integrity

1. INTRODUCTION

Programs can be maliciously modified either when the program resides in storage (on disk for example) or at runtime. To detect malicious modifications while the program is on disk, many motherboards include a Trusted Platform Module (TPM) [1] that checks for program integrity at load time. The TPM does not check for program integrity at runtime.

Dynamic integrity checking of programs is needed to detect runtime modifications [2]. Signature instruction stream is an approach for runtime detection of control flow errors caused by transient and intermittent faults [3]. A non-cryptographic hash of the basic block is appended at its end at compile time, and this hash is then compared against the hash generated at runtime. Secure program execution framework [4] uses a hash function along with a cryptographic transformation. CODESSEAL [5] is a joint compiler/hardware infrastructure for dynamic integrity checking of basic blocks. The pre-computed hashes are stored in the memory of an FPGA which is placed between the main memory and the last level cache. Runtime Execution Monitoring (REM) [6] modifies the processor microarchitecture, and the instruction set architecture (ISA) to support dynamic integrity checking. Dynamic integrity checking has a performance overhead as it involves hash fetch from memory or disk, hash calculation, and comparison with fetched hash. Moreover, these techniques require modifications to the microarchitecture or the instruction set architecture or both.

In this paper, *We propose a novel application of hardware performance counters (HPC) -which are built into almost all mainstream processors for performance tuning purposes- for static and dynamic integrity checking.* HPCs have been included in microprocessors from almost two decades. HPCs have been used for performance evaluation of hardware systems [7], characterization of software applications [8], and testing of software applications [9, 10]. Using side-effects, like checksum calculation or performance counters, was discussed in [11, 12]. In [11] the authors propose remote authentication system called "Genuity" that verifies a system (the hardware as well as the software that runs on it) by running validity tests (sending an input) and a checksum (output) is sent back over the network. The checksum takes into account not only hashing locations in memory but also DTLB miss performance counter. The paper [12] shows that the system in [11] can be attacked, and argues that a software-only solution to software authentication faces numerous challenges, making success unlikely. However, in this paper we use performance counters to *build a model* that is used to check the integrity of the program dynamically. Our proposed scheme is hard to attack using the scheme in [12] because we use *relationship among performance counters* not only raw numbers from performance counters like in [11].

2. HPCS FOR INTEGRITY CHECKING

HPCs are a set of special-purpose registers built into processors. HPCs store the counts of software and hardware related events. Examples of such events include cache misses, instructions committed, and branch instructions committed. Although a typical processor has the ability to measure a large number of events, (for example, 133 events on the Nehalem architecture [13]) all of them cannot be monitored at

STC'11, October 17–21, 2011, Chicago, Illinois, USA.

the same time; this is determined by the number of HPCs supported. For example, since Ultra Sparc II has 2 HPCs we can monitor 2 events at the same time and since AMD Athlon has 4 HPCs [14], we can monitor 4 events at the same time.

2.1 The Threat Model

Similar to other integrity checkers, we target attacks that alter the instructions of a program before or during execution. This means that our threat model allows an attacker to tamper with the disk and make modifications to the program's binary [16]. Modifying the program during execution [2] is another class of attacks that we target. Specific instances of this class include buffer-overflow attack [17] and return-to-libc attack [18].

2.2 HPC-based Integrity Checking

State-of-art static and dynamic integrity checkers compute a cryptographic hash of the program periodically and compare against a pre-computed hash of the original program. It is impossible for an attacker to modify the program without modifying the generated hash.

In HPC-based integrity checking, we monitor event counts (using the built-in HPCs) and relationships between these event counts for the program being checked. The attacker does not know which events we monitor, which inputs we use to count events, and which relationships we model and monitor. We make three important assumptions before describing our static and dynamic integrity checking schemes. Our first assumption is that the OS is trusted. Second, we assume that the additional integrity information we store with each program binary is encrypted and hashed; this ensures that the attacker cannot alter this information when it is loaded from the disk to memory and then to the processor. Moreover, this step happens only once at the beginning of the program execution. After that, this information is kept in the OS address space and cannot be read or modified by other programs. Finally, similar to most integrity checking schemes, we check the integrity of programs and not the associated data.

2.2.1 Which HPCs to Monitor?

In the context of integrity checking, we shortlisted processor events to monitor as follows.

- They are independent of processor clock frequencies. Different processors have different clock frequencies and the same processor can vary its clock frequency to reduce power. So events that count cycles are not very revealing in terms of program behavior. A program that runs when the clock frequency is high may have cycle counts totally different from the same program and same input executed on the same processor with lower frequency.
- They are independent of performance enhancement structures, like TLBs, branch predictors, and caches as they may not provide repeatable numbers. These event counts may wary from one execution to next as there are a lot of predictions (branch predictions, prefetches, etc). This means the count of any event happening during fetch or execution stages of instructions can vary among the same program runs with the same input. Microarchitecture independent measures may characterize programs best [15]. So we avoid events based on these structures, including hits and misses into TLBs and the caches.
- They track malicious program modifications. For example, such modifications change either the number of instructions executed, their type, or both.

The shortlisted events include the total number of instructions retired, the number of branch instructions retired, cache stores (they are better measures than loads because loads can be speculative), completed I/O operations, and number of floating point operations.

2.2.2 HPC-based Static Integrity Checking

The left part of Figure 1 shows the main workflow of the static integrity checking scheme. Static integrity checking is done in two steps.

- **Step 1: Profiling at installation time**– The trusted OS executes the program on selected inputs without user interaction and collects the HPC values of events at the end of the execution. In the end, the OS has the final counts of the selected events which are then stored encrypted and hashed on the disk together with the program binary. The attacker does not know when the profiling is done or what are the inputs being used in profiling.
- **Step 2: Profiling at check time**– Anytime the user (Here, we mean anybody who has legitimate control over the machine, such as the owner, system administrator, or legitimate user.) initiates the program execution, or just needs to check the integrity of this program, the profiling step is repeated by the OS using the same inputs it has used in that first step, measuring the same events. The OS then compares the values of the HPCs with the ones stored from the initial profiling.

We do not expect the HPC values to be identical from one run to the next, even if the program is not modified and the same input is used. This is because program execution depends on a number of factors. For example, programs are loaded into different parts of memory, which is different with each run. Also, some programs use random number generators in their functions, which can also be different among runs. These factors, together with the noisy nature of HPCs [19, 20] affect the HPC values. However, we found from our experiments, that if the difference is within 5% the program can be assumed safe.

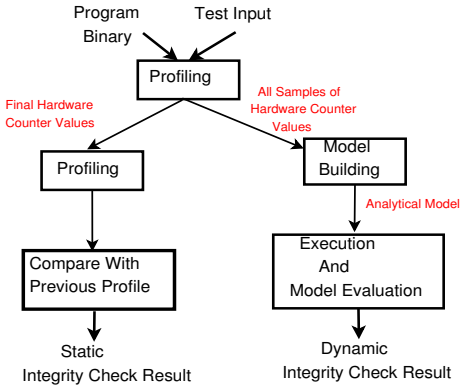

Figure 1: HPC-based Integrity Checking

2.2.3 HPC-based Dynamic Integrity Checking

Our hypothesis is that *a program can be approximated as mathematical relations between program event counts (such as the number of instructions committed, the number of func-*

tion calls, etc) to each other. Any program can be presented as a control flow graph (CFG) where each node is an instruction and an arrow going from instruction A to instruction B means that execution of B follows execution of A. Some instructions such as branches and function calls/returns break this sequentiality. Executing a program means going on a path through this CFG. For all legitimate paths in this CFG (i.e. no malicious alterations to the program) the number of instructions executed and their types are known.

1. **Example:** Suppose through all legitimate paths of the CFG we encounter at most 6 branches. If at runtime we count 8 committed branches, we can suspect something malicious.
2. **Example:** If there is a loop, and each loop iteration contains 3 branches, then at runtime the number of branches after executing this loop (no matter how many iterations) must be a multiple of 3.

By executing a program several times with different inputs, we can get an *approximation* of these mathematical relationship between program events measured by the HPCs.

The right part of Figure 1 shows the main workflow of the proposed scheme. The scheme is divided into three main steps: profiling, model building, and execution.

- **Step 1: Profiling** is done either at compile time, installation time, or at load time (if the program is small enough not to increase load time substantially). It involves executing the program with small an input set and gathering event counts. This is very similar to the first step of our static scheme and also under the OS control. The main difference is that at this step the system gathers not only the final values of the counters but reads the counters at fixed intervals and stores all the readings till the end of execution. Reading those HPCs at the end of the execution does not give an indication of the dynamic program behavior.
- **Step 2: Model building** is done at the end of the profiling phase using the periodic HPC values. We then unearth relationships between the different events. The analytical model can be of any sophistication. But for simplicity, we have chosen to generate linear relationships. For this we use Eurequa [21], a software tool for detecting equations and hidden mathematical relationships in data. We tuned Eurequa to generate a linear relationship among the HPCs we measured: INS (number of instructions retired), BR (number of branches retired), WR (L1 data cache writes), IO (complete I/O operations), FP (number of floating point operations retired), and FN (return and call instructions executed). We tried predicting INS as a function of FN, WR, BR, IO, and FP. But the detection accuracy was not very high; some malicious program modifications, such as modifying a single operation, may produce very similar number of retired instructions as a correct program. So we modeled, after several exploratory experiments, two events: INS and WR, in terms of the other events. The exact relationship is program dependent.
- **Step 3: Runtime checking**– The trusted OS uses the mathematical model to check for program integrity as follows. At load-time, the operating system (OS) reads the analytical model and stores it in the OS address space. Periodically, during execution, the OS reads the HPCs, and uses the model to calculate INS and WR event counts and compares these predictions with the actual numbers read from the HPC. We do not reset the counters every time the integrity is checked. If the deviation is higher than some threshold, the OS flags this program as *possibly malicious.*

The main advantage of our technique is that does not require additional hardware or ISA modifications. It leverages the HPCs built-in to almost all processors. So all that we need is a profiling step and a small patch to the OS.

2.3 Usage scenarios of the techniques

The proposed HPC-based integrity checking techniques can be used in many different ways.

- **Scenario 1: Embedded systems with no integrity checking support**– In this case, the proposed technique is the only integrity checking scheme. Low cost embedded systems are widely deployed and the technique can be easily added to these fielded systems without significant costs.
- **Scenario 2: High end systems with a static TPM but no dynamic integrity checking** – Most desktops and laptops come with a TPM-based static integrity checking. The proposed approach can be used to provide dynamic integrity checking.
- **Scenario 3: High end systems with a static TPM and dynamic integrity checking** – Very few systems fall into this category. In this case the proposed approach can be used in tandem with the sophisticated integrity checking techniques. The sophisticated and expensive techniques may be disabled by default. They are enabled *only* when the HPC-based techniques detect *malicious program modifications.*

3. RESULTS AND ANALYSIS

3.1 Experimental Setup

Our experimental setup has three parts: The first one is to read the HPCs. For this we used perf_event tool which is now part of the Linux kernel. We modified perf_event to read the HPCs of interest every 10,000 cycles and output the readings to a file. The choice of 10,000 comes from the fact that it is usually the time slice given by the OS to an application in a round-robin scheduling. We set perf_event such that the HPCs reported are for our applications only and not the OS. We have statically compiled our benchmarks to isolate any side effects from dynamically linked libraries. The second part has been explained in Section 2. The third part is to use our analysis to detect possible malicious modifications to the programs even if they use different inputs than the ones used in the analysis phase.

- **Benchmarks** We use programs from the CTuning suite [22]. We have chosen a blend of different programs with different characteristics and instruction mixes. For each program, we profile the correct program using an input set different than the one used at runtime. Then at runtime, we test our model with the correct program but with a different input set.
- **Malicious modifications** For each program, we insert two types of malicious modifications. Malicious modification 1 is an extra function call. This simulates such attacks as buffer overflow and return-to-libc. Malicious modification 2 is a operation modification (such as an addition changed to subtraction). This simulates attacks that try to bypass a security check (such as password check), denial of service attacks, or attacks that make the system leaks

information. We have taken great care that the modifications do not cause the program to crash. Also the modifications are placed after careful consideration of each program's CFG to ensure the modification will be executed no matter the input. In the tables, if digit 1, representing malicious modification 1, is appended to the program name, it means an extra function has been added to the program, while 2, representing malicious modifications 2, means an operation has been modified.

We tested our technique with a wide range of modifications. All the experiments were performed on Ubuntu with kernel version 2.6.38 running on an Intel Quad-Core 2 Q9400 (Yorkfield).

3.2 Static Integrity Checking

Table 1 shows the static integrity checking results. The table shows the percentage deviation of each HPC compared to the correct program running on the same inputs. We used two different inputs inp1 and inp2 in all the tables. Column 1 shows the benchmark used. For example, bzip2-1-inp1 represents the program bzip2 with an extra function added to it (malicious modification 1) and executed using input inp1. Subsequent columns present the percentage deviation of each HPC from the values of the correct program running on inp1.

We have 4 benchmarks, with 2 types of malicious modification, and 3 different inputs. This means we have 24 readings for each program event. We compare each malicious program with the corresponding benign program running on the same input. *A deviation of more than 5% means program modified.*

We can see from the table that the IO count (i.e. bus transactions) is the most sensitive to program modifications; it has detected 19 out of the 24 different malicious modifications. This is because most programs have a high percentage of I/O operations. And those operations require several bus transactions. The total number of instructions (INS) is the least sensitive. This is because of the 90/10 rule; most programs spends 90% of their times in 10% of the code. So if the modification does not hit the 10% of the code it will not have a big effect on the total number of instructions. Floating point count (FP) is useful only when a program does a lot of calculations; bzip2 as it has the largest number of floating point operations in all the benchmarks shown. The number of branch instructions (BR) is significant in detecting modifications if the CFG of the program is complicated with many decision points. This is very obvious in the first three benchmarks. Which type of malicious modification has the highest effect on the counters is program dependent. For instance, bzip2 is more sensitive to adding extra functions, while gsm is more sensitive to the operation modification. This behavior is related to the complexity of the CFG of each program. Bzip2 has less complicated CFG and hence an extra function makes enough disturbance to the flow of execution to be detected. But the more complicated CFG of gsm means it has more decision points. So a modified operation that affects the decision at some points of the CFG can be easily detected. Depending on the input dijkstra can be through complicated path in the CFG with a lot of decision points (like what happens with inp2); or it can go into a straightforward path of sequential function calls which makes it harder to detect. The sensitivity also depends on the position and type of the operation modification. With the 5% threshold deviation, we can see from the table that we have been able to detect all modifications. If any counter for an application goes above the threshold, the program is flagged as maliciously modified.

Finally, if we decrease our detection threshold from 5% we will get a lot of false positives. If we increase this threshold, some malicious modifications can go undetected.

Table 1: HPC-based static integrity checking: Deviation from correct execution (naming convention of leftmost column: benchmark-malicious modification type-input

Benchmark	INS	BR	FP	FN	IO	WR
bzip2-1-inp1	26.37	22.38	35.98	31.57	2843.59	9.18
bzip2-1-inp2	0.87	2.46	11.95	11.37	248.03	8.73
bzip2-1-inp3	21.30	14.93	50.88	46.01	4588.91	0.40
bzip2-2-inp1	0.53	0.09	3.66	1.14	1.91	5.52
bzip2-2-inp2	2.47	2.38	1.03	2.72	4.25	5.76
bzip2-2-inp3	2.70	1.13	13.75	7.37	3.94	6.42
dijkst-1-inp1	3.70	3.62	1.80	1.26	66.91	15.18
dijkst-1-inp2	10.15	10.15	13.86	0.00	35.16	0.44
dijkst-1-inp3	0.64	0.64	2.86	0.64	58.93	3.61
dijkst-2-inp1	10.19	10.56	0.65	8.30	30.98	14.23
dijkst-2-inp2	58.00	59.79	33.86	61.70	66.89	62.79
dijkst-2-inp3	62.11	64.98	24.24	58.18	65.57	65.04
gsm-1-inp1	6.51	25.40	16.92	18.92	2194.55	35.75
gsm-1-inp2	1.83	2.20	3.70	2.10	37.02	1.02
gsm-1-inp3	4.93	0.57	4.21	2.59	601.46	10.21
gsm-2-inp1	14.75	17.40	8.20	10.22	0.94	3.20
gsm-2-inp2	13.75	10.04	1.36	0.69	16.44	12.09
gsm-2-inp3	15.73	12.82	9.45	10.21	4.72	2.89
lame-1-inp1	1.07	4.95	15.46	7.26	1245.90	3.85
lame-1-inp2	0.13	0.10	6.28	6.77	49.95	5.81
lame-1-inp3	0.09	2.40	17.68	3.61	91.05	0.33
lame-2-inp1	1.12	1.07	9.74	1.42	65.30	5.44
lame-2-inp2	1.02	0.85	6.87	6.75	40.37	0.60
lame-2-inp3	4.86	5.03	14.44	0.24	49.85	0.53

3.3 Dynamic Integrity Checking

Table 2 summarizes the results of the profiling and model building phases for the benchmarks. The first column presents the benchmark and the input set. The second column shows the number of samples gathered during profiling. These samples are used to build the program model and hence are not stored. The third column shows the two linear relationships for INS and WR for each of the benchmarks. Any program starts execution with a startup phase that does not represent its real behavior [23]. For example, the startup phase of bzip2 is where the program loads the input file to be compressed. The startup phase is program-dependent and is shown in the fourth column. This information along with the models are stored with program binary.

The last column shows the threshold deviation between the value calculated, during real execution, at runtime and the value measured by HPC. If the measured deviation is above this threshold, the program has been modified. This deviation depends on the mathematical model we choose and the program. So it is also an experimental number. For example, if the calculated INS is higher than the measured one by more than 10% *and* WR measured is different from the calculated one by more than 100% than the program is

Table 2: HPC Based Dyamic Integrity Checking: Profiling Statistics and Model Building

Bench-inp	Num samples	Model	Samples to skip	thresh. error (%)
bzip2-inp1	65535	INS=10BR -1288FN WR=103FN +50IO	2500	INS: 10 WR: 100
dijks-inp1	15387	INS=266FN -20WR WR=BR +55IO	3000	INS: 25 WR: 20
gsm-inp1	65536	INS=6BR -FP WR=-10FN +150IO	1500	INS: 25 WR: 30
lame-inp1	65536	INS=21BR -11270FP WR=25FN -53IO	1500	INS: 30 WR: 10

Table 3: HPC Based Dynamic Integrity Checking: Runtime Checking

Bench	INS Dev. (after startup)	WR dev. (after startup)	Detected?
bzip2-inp1	5.6%	68.2%	no
bzip2-inp2	78.46%	27.48%	False +ve
bzip2-1-inp1	59.66%	1750.84%	yes
bzip2-1-inp2	0.56%	1164.30%	yes
bzip2-2-inp1	4.23%	1736.40%	yes
bzip2-2-inp2	0.25%	1164.30%	yes
dijks-inp1	21.4%	19.26%	no
dijks-inp2	99.01%	26.05%	False +ve
dijks-1-inp1	41.85%	20.84	yes
dijks-1-inp2	116.66%	61.20%	yes
dijks-2-inp1	25.36%	2.53%	yes
dijks-2-inp2	99.02%	50.57%	yes
gsm -inp1	24.73%	5.22%	no
gsm -inp2	19.71%	29.15%	no
gsm-1-inp1	58.14%	284.72%	yes
gsm-1-inp2	19.84%	154.39%	yes
gsm-2-inp1	67.10%	27.35%	yes
gsm-2-inp2	32.39%	41.23%	yes
lame-inp1	29.27%	0.35%	no
lame-inp2	42.18%	23.64%	False +ve
lame-1-inp1	35.10%	50.74%	yes
lame-1-inp2	56.74%	52.86%	yes
lame-2-inp1	7.02%	2.21%	no
lame-2-inp2	42.73%	12.99%	yes

flagged. The main reason for the high threshold of WR for bzip2 is that Eurequa was not able to find a very accurate model. It is important to notice here that the mathematical model does not generate the exact count, and it is not meant to do so. It is meant to generate an approximation that is within the threshold.

We executed several versions of the same application. Table 3 shows the runtime results. The first two rows for each benchmark are the correct application (non-modified) but with different inputs. The following four rows contain malicious modifications and are run on different inputs. Columns 2 and 3 show the deviation from the model generated at profile time for the INS and WR HPCs.

Following is a summary of the results.

- The first row for each benchmark is a validation experiment to check the mathematical model. We re-executed the program, unmodified, with the same input (inp 1). This shows that the correct program with the same input will pass the test, as the deviation is less than the threshold.
- Ideally, we would like to avoid false positives. The second row for each benchmark is the case where the program was executed, unmodified, with a different input (inp 2). Each of the results came out as a false positive (i.e. we detect a malicious modification when there is none).
- All modifications have been detected for both types of malicious modifications except lame-2. From the twenty tests presented in Table 3, the proposed technique detected the malicious modifications in 16 cases and failed to detect 2 malicious modifications.

A detailed study of the applications, the profiling process and the model building should further improve the accuracy of this technique.

3.4 Overhead: Time and Storage

There is no hardware cost for our scheme because HPC exist already in almost all processors. There is minimal overhead for storage and time. For the static scheme, the system only needs to store the last values of the counters at the end of profiling. So with X counters, we need to store X values with the program binary. In the experiments conducted in this paper we keep track of six counters each of which is 8 bytes. So our overall storage is 54 bytes. There is no performance overhead for the static scheme because the whole scheme is offline. For the dynamic scheme the storage requirement is also minimal. During profiling, the system needs to keep all the samples (i.e. HPC values) measured during the profiling. But these samples are kept only until the system builds the mathematical model. After that the samples can be deleted and only the model and the thresholds, which don't take more than 40 bytes, need to be stored with the program binary. During execution, there is little performance overhead (less than 10%) due to operating system interrupts to read the counters and compare the read values with the pre-computed model. Part of that overhead is hidden because the operating system overhead takes place anyway (for scheduling and other type of interrupts) whether our scheme is there or not.

4. CONCLUSIONS

We showed how HPC can be effectively used for both static and dynamic integrity checking. There are several outstanding challenges that we are currently addressing:

- Dynamic integrity checking when several programs are running at the same time. In this case, the HPC will not be able to keep track of the counts of the events for all the programs. In fact, the HPC may measure cumulative counts that do not represent any single program. There are HPC monitoring tools, like the one we used in this paper, that can monitor and gather measurements for a

targeted program. But in this case, the other programs will not be checked. We plan to extend these tools to save the measurements of a program when it is sleeping (during process scheduling for example), reset the counters, and use the HPC for the current active program. In this way, the HPC can then be used to monitor different programs. When the programs are running at the same time, we plan to explore HPC scheduling, where HPCs are used to measure events for a program for sometime. Then those measurements are saved, HPCs reset, and then used to measure another program, and so on.

- The number of HPCs in a processor is much smaller than the events that can be measured. This can impact the accuracy of the scheme because the more events that can be tracked the more accurate the model built. For static integrity checking we can profile several times measuring different events each time. For dynamic integrity checking, we are exploring schemes where the OS and the tools can change the events to be measured by HPCs dynamically at runtime.

Our future research agenda includes a thorough analysis of the program's CFG to study how the position of the modifications in the program and the number of modifications relate to the sensitivity of our technique. We also plan to investigate more microarchitecture-related counters. This will be very beneficial to check the integrity of the system itself in addition to the program. Finally, we will expand our research to include multithreaded applications.

Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-09-1-0146. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Also this work is partially funded by GAANN fellowship.

5. REFERENCES

[1] R. Ng, "Trusted Platform Module - TPM Fundamental," http://www.asprg.net/aptiss2008/slides/TPM Fundamentals-raymond ng.pdf, August 2008.

[2] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "TOCTOU, Traps, and Trusted Computing," in *Proc. of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, 2008.

[3] M. Schuette and J. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Transactions on Computers*, vol. C-36, no. 3, pp. 264–276, March 1987.

[4] D. Kirovski, M. Drinić, and M. Potkonjak, "Enabling Trusted Software Integrity," in *ASPLOS*, vol. October. New York, NY, USA: ACM, 2002, pp. 108–120.

[5] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno, "CODESSEAL: Compiler/FPGA Approach to Secure Applications," in *Proc. of the IEEE International Conference on Intelligence and Security Informatics*, May 2005, pp. 530–535.

[6] A. Fiskiran and R. Lee, "Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution," in *Proc. of IEEE International Conference on Computer Design*, October 2004, pp. 452–457.

[7] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *International Symposium on Computer Architecture (ISCA)*, 2011.

[8] Y. Luo and K. W. Cameron, "Instruction-level characterization of scientific computing applications using hardware performance counters," in *Proceedings of the Workload Characterization: Methodology and Case Studies*, 1998.

[9] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2008.

[10] C. Yilmaz, "Using hardware performance counters for fault localization," in *Proceedings of the 2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, 2010.

[11] 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*, 2003.

[12] U. Shankar, M. Chew, and J. D. Tygar, "Side effects are not sufficient to authenticate software," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, 2004.

[13] "http://www.intel.com/products/processor/manuals/."

[14] "http://developer.amd.com/assets/intro_to_ca_v3_final.pdf."

[15] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *IISWC*, 2006.

[16] A. Tereshkin, "Evil Maid Goes after PGP Whole Disk Encryption," *International Conference on Security of Information and Networks*, 2010.

[17] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and Defenses for the vulnerability of the decade," in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, February 2004, pp. 227 – 237.

[18] D. J. Day, Z. Zhao, and M. Ma, "Detecting return-to-libc buffer overflow attacks using network intrusion detection systems," in *Proceedings of the 2010 Fourth International Conference on Digital Society*, ser. ICDS '10, 2010, pp. 172–177.

[19] V. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *IISWC*, 2008.

[20] M. Kuperberg and R. Reussner, "Analysing the fidelity of measurements performed with hardware performance counters," in *Proceeding of the second WOSP/SIPEW international conference on Performance engineering*, 2011.

[21] M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," *Science*, vol. 324, no. 5923, pp. 81–85, 2009.

[22] "http://ctuning.org/."

[23] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th annual international symposium on Computer architecture (ISCA)*, 2003, pp. 336–349.