Feasibility Study of Dynamic Trusted Platform Module

Arun K. Kanuparthi, Mohamed Zahran and Ramesh Karri
Polytechnic Institute of NYU
akanup01@students.poly.edu mzahran@acm.org rkarri@poly.edu

***Abstract*— A Trusted Platform Module (TPM) authenticates general purpose computing platforms. This is done by taking platform integrity measurement and comparing it with a pre-computed value at boot-time. Existing TPM architectures do not support run-time integrity checking of a program on the platform. Attackers can modify the program after it has been verified at the *Time Of Check (TOC)* and before its *Time Of Use (TOU)*. In this paper we study the feasibility of integrating a dynamic on-chip TPM (DTPM) into the core processor pipeline to protect against *TOCTOU* attacks. We explore the challenges involved in designing DTPM and describe techniques to improve its performance. The proposed DTPM has 2.5% area overhead and 18% performance impact when compared to a single processor core without DTPM.**

I. Introduction

A trusted platform module (TPM) acts as a root of trust for the platform that contains it, and can securely store artifacts used to authenticate the platform. These artifacts can include passwords, certificates, or encryption keys. A TPM also stores platform integrity measurements to ensure trustworthiness of the platform [1].

Current trusted platform architectures only provide load-time guarantees. Integrity is measured just before the program is loaded into memory[1]. However, an attacker can introduce malicious code after the program is checked and before it is used. This class of attacks is called TOCTOU attacks [2].

We study the feasibility of using run-time integrity measurements to counter the TOCTOU attacks. Section II presents background on TPM, TOCTOU threat model, and related work. Section III discusses the challenges involved in designing DTPM, presents our approach to designing a DTPM, and proposes techniques to optimize performance. We describe the experimental setup to evaluate the proposed approach and analyze the results in Section IV and conclude in Section V with our ideas for future work.

II. Background

A. TPM

A TPM is the root of trust for a computing platform. It builds a chain of trust by measuring various parts of the platform. The main functions of a TPM are to check for platform integrity, data sealing and binding. To check for integrity, it calculates a cryptographic hash of the platform configuration using the built-in SHA-1 hash engine and stores this integrity measurement in a protected storage called Platform Configuration Registers (PCRs). When TPM is challenged to prove the authenticity of the platform or the program running on the platform, it responds by reporting the integrity measurement stored in the PCRs. The integrity is checked by comparing the computed hash value and the value reported by the PCR. TPM contains a 2048-bit RSA engine for public key encryption, a random number generator, and a built-in RSA key known as the Endorsement Key, that is used to generate additional secret keys which can be used to provide data sealing and binding.

B. The Threat Model

PCRs store the platform integrity measurement taken at boot time. Based on this measurement, a user is forced to trust this program at a subsequent time. However, a malicious user might induce run-time vulnerabilities after TOC and before TOU. Alternatively, an attacker may modify instructions and data that were correct at TOC, before TOU. These are two instances of TOCTOU threats [2]. Thus, there is a need for run-time integrity checking.

C. Previous Work

Run-time integrity checking to detect control flow anomalies involves computing a hash value (cryptographic or non-cryptographic) of an instruction or a basic block at compile time and comparing it with the hash value calculated at run-time. An approach for run-time detection of control flow errors caused by transient and intermittent faults was proposed in [3]. A non-cryptographic hash is appended to every basic block at compile time and this is then compared against the hash generated at run-time to detect any control flow errors. A joint compiler/hardware infrastructure for run-time integrity checking, CODESSEAL, was proposed in [4]. At compile-time, CODESSEAL calculates the cryptographic hash of a basic block and embeds this information into the executable. The pre-computed hashes are stored in the memory of an FPGA that is placed between the main memory and the closest cache to the main memory. At run-time, the integrity of each basic block is verified by the FPGA. A related approach uses caches and hash trees to verify the integrity of programs and data in the memory [5]. Data are placed at the leaves of a tree in which every node contains the hash of the nodes below it. This approach performs run-time integrity checking by recursively verifying the hash of the incoming basic block and all the hashes of its parent nodes, up to the root hash.

[1]It is assumed that this program remains unchanged.

Runtime Execution Monitoring (REM) [6] modifies the micro-architecture and ISA to detect and prevent program flow anomalies resulting from malicious code injection. It verifies program code at the basic block level by pre-computing keyed hashes during program installation and then comparing these values against their hashes computed at run-time. REM controls the processor pipeline and does not commit the instructions in the basic block until integrity has been checked. This run-time integrity checking is achieved at the cost of modifying the ISA. In REM, the hashes are stored on the L1 instruction cache. This results in contention between the instructions and the hash values. None of the hashes are stored on disk. As a result, there is a significant storage overhead in the memory.

The central objective of the above mentioned work is to provide run-time integrity checking to prevent control flow errors. The approach proposed in this paper also has the same objective, but differs in the way the hash storage is handled, and also provides run-time integrity checking at a different level of granularity (at the trace level, where each trace consists of several basic blocks). A hash storage hierarchy is proposed to store the pre-computed hashes. The proposed approach does not involve making modifications to the ISA.

III. Challenges in Designing a DTPM

A. Motivational Example

Consider `403.gcc`, a SPEC CPU2006 benchmark. When `403.gcc` is run on a reference input `166.s`, it executes 82×10^9instructions. Run-time integrity checking of each of the instructions entails calculating a hash value for each instruction and comparing it with a pre-computed value. Using a cryptographic hash function, SHA-1 in this paper, on each instruction takes 80 cycles to generate a 20-byte output for each instruction [7]. Consequently, calculating a hash for every instruction degrades performance, and has large storage of 1.64 trillion bytes[2]. Hence, checking for integrity at the instruction level is not practical.

Let us consider run-time integrity checking at the basic block level. A basic block is a sequence of instructions with one entry point and one exit point [3]. When we perform run-time integrity checking at the basic block level on the `403.gcc` example, 51224 basic blocks are executed. We will now require 8004 KB to store these hashes. This is still too large to be stored on the chip. Therefore, we need to store some of the hashes in the disk (encrypted using the platform RSA keys). When we perform run-time integrity checking at the basic block level, it takes 102.1×10^9 cycles while executing 500 million instructions. This is because multiple disk accesses are required to fetch the hashes and also hash calculations for the basic blocks consume extra cycles. Fig. 1 shows that including the DTPM increases the number of execution cycles, by 250× on average for all the benchmarks [8].

[2]An average instruction is four bytes in size, which is much smaller than 512-bits. Thus, we pad the instruction to produce the 64-byte input. This underutilizes the SHA-1 hash function.

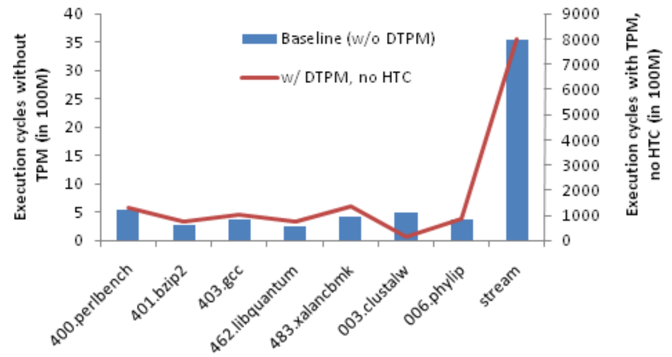

Fig. 1. Impact of including the DTPM on the execution cycles

If we perform run-time integrity checking at the function level, we need to hash all the functions in the application. The `403.gcc` benchmark has 3890 functions. If we check the integrity of every function, we will need to generate and store 3890 hashes, which is about 78 KB. This is a significant improvement. The number of cycles required to generate the hash value is proportional to the size of the input, which in this case, is the size of instructions in a function. Hashing functions with a large number of instructions in it will take a large number of cycles. So, we must stall the pipeline until all the instructions in the function are in the pipeline. This will impact the performance. Since checking for integrity at the instruction level, basic block level and function level are impractical, we propose to check at the trace level.

B. Integrity checking of Dynamic Instruction Traces

At run-time, a program takes different paths depending on the input applied. It follows a particular sequence of basic blocks depending on the control instructions that are encountered. When a control instruction is encountered, it starts executing a different basic block. This logical sequence of basic blocks is called a trace [9]. A control flow graph is a graphical representation of all the different traces that can be traversed during program execution. Fig. 2 shows an example Control Flow Graph (CFG) for a function with each basic block acting as a node. After the last instruction in each basic block, there is a control instruction (conditional or unconditional branch, a jump, or return) which transfers the control to another basic block. There are six possible traces in the CFG shown in Fig. 2: $BB0 \rightarrow BB1 \rightarrow BB4 \rightarrow BB5$, $BB0 \rightarrow BB1 \rightarrow BB3$, $BB0 \rightarrow BB1 \rightarrow BB3 \rightarrow BB5$, $BB0 \rightarrow BB2$, $BB0 \rightarrow BB2 \rightarrow BB3$ and $BB0 \rightarrow BB2 \rightarrow BB3 \rightarrow BB5$.

The starting address of the trace is the starting address of its first basic block. Each trace has a trace ID. The trace ID is the starting address of trace followed by a series of 1s and 0s indicating branch taken or not taken. The trace ends with a return instruction or when it reaches a pre-defined length in terms of number of basic blocks. If an attacker modifies the instructions without changing the trace ID, it will still result in a different hash value.

One parameter that needs to be considered is the number of basic blocks in a trace. In one extreme, each basic block

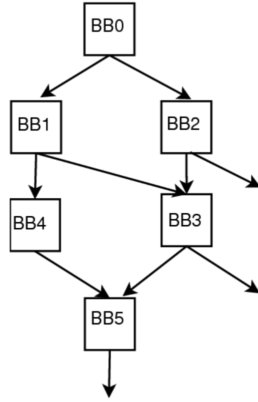

Fig. 2. Example CFG of a function

can be a trace. The obvious advantage is that this results in fewer pipeline stalls. But, this will result in a large number of hashes. On the other extreme, if each trace is composed of a large number of basic blocks, the number of hashes is reduced. However, it will increase the number of pipeline stalls. From the profiling information obtained for the benchmarks, the average basic-block size is 16 bytes. We propose that each trace is composed of up to four basic blocks. Since the input to the SHA-1 hash function will be 64 bytes, this results in efficient usage of the SHA-1 hash function as it does not involve zero padding the input. Trace size can similarly be tuned for other hash functions. This results in a small number of hashes and fewer pipeline stalls. Thus, we propose run-time integrity checking at the trace level, with each trace consisting of at most four basic blocks.

To reduce the number of traces that need to be checked, we leverage the observation that a processor spends 90% of the time executing 10% of the program code [10]. This is called the 90-10 rule. By generating hashes for the most frequent traces, we reduce the hash storage overhead. When this rule is applied to the `403.gcc` benchmark, we need only 1636 hashes at the trace level when compared to 51224 and 82 billion hashes at the basic block and instruction levels respectively. The first five columns of Table I show the reduction in the number of hashes and the size of all the hashes.

The 90-10 rule is just one approach to reduce the total number of hashes. If security is more important than performance, then security-critical functions, traces, or basic blocks may be checked for integrity instead.

C. Micro-architecture Support for Run-time Integrity Checking of Traces

There are several important design considerations here. First, where should the DTPM be placed in the core pipeline? Second, how does the DTPM know that the address currently being fetched is the start of a trace? And finally, if the DTPM finds that a trace is modified, when does it abort execution?

To address the first issue, since we compute hashes only for selected traces, we need to monitor the starting address of the trace. Thus, having the DTPM near the fetch stage of the pipeline is a good solution. Fig. 3 shows the DTPM integrated into the basic processor pipeline. The dotted line shows the chip boundary. The shaded region in the DTPM is the DTPM internal storage. The DTPM gets the program counter from the fetch stage, the branch directions from the branch predictor and the instructions from the L1 instruction cache. These are shown by arrows (a), (b) and (c) respectively in Fig. 3. These are used to identify the trace. Hashes that are stored outside the DTPM are fetched when the hash is not found in the DTPM.

The starting address of a trace can be conveyed to the processor by storing the start addresses of the traces in the DTPM internal storage, indexed by the trace ID. This is our solution to the second problem.

As far as aborting execution, we tag each instruction with a bit as it progresses through the pipeline. This bit indicates whether the instruction has been successfully verified by the DTPM. If this bit is set, it means that the DTPM is still checking the instruction and hence the instruction cannot be committed. DTPM controls the commit stage of the pipeline. This is shown by arrow (d) in Fig. 3. If the bit is clear, it means that the trace has been successfully checked and hence can be committed. If the run-time integrity checking fails, it interrupts the processor for further action. If the DTPM does not detect the start of a trace, it means that the current trace is not used frequently and the fetched instructions commit without any DTPM intervention.

Some of the delay incurred in the process of fetching pre-computed hashes, calculating hash for the current trace, and comparing them, is masked by the time taken by the pipeline to execute the instructions in the trace. Overall system performance is affected only when the instructions reach the commit stage and DTPM has not yet checked their integrity. This occurs mainly because of disk accesses to fetch the pre-computed hashes.

We propose to reduce the disk accesses by introducing a hash storage hierarchy. One solution is to store the hashes in the on-chip cache as in [6]. But this will result in contention for accessing the cache along with the instructions and data. Therefore, we propose to use a dedicated Hash Trace Cache (HTC) to store the hashes. HTC was inspired by trace caches [9]. The main goal of HTC is to cache the hashes fetched from the disk, so that the DTPM will not need to access the disk very often. If DTPM does not find the hash in its internal storage, it will look for it in the HTC. HTC is accessed using the trace ID, and if there is a hit, it returns the hash of that trace. We assume that the chip is protected from physical attacks and the hashes stored on-chip cannot be tampered with, but anything off-chip can be modified by an adversary. Thus, the DTPM does not compute the hash of the current trace if the hash of this trace is found in the HTC. It marks the trace as safe. But, if there is a miss in the HTC, the lower levels in the hash storage hierarchy are searched for and the DTPM calculates the hash for this trace. The pre-computed hash found in the lower level is fetched to the higher level of hash storage hierarchy, i.e., the HTC, decrypted using the platform RSA keys. If the calculated and

	No. of Hashes		Size of Hashes		HTC accesses	main memory accesses	disk accesses
	w/o 90-10 rule	w/ 90-10 rule	w/o 90-10 rule	w/ 90-10 rule			
400.perlbench	20149	707	3149	14	23498952	72393	667
401.bzip2	2508	198	392	4	6790463	67345	156
403.gcc	51224	1636	8004	32	60112547	374219	2487
462.libquantum	1648	151	258	3	5260921	161078	114
483.xalancbmk	24165	826	3776	17	28011925	47772	765
003.clustalw	2554	146	400	3	5683924	330177	108
006.phylip	1745	122	273	3	4883263	17249	79
stream	1248	42	195	1	36621247	165482	8

TABLE I

COMPARISON OF NUMBER AND SIZE (IN KB) OF HASHES, AND NUMBER OF HTC ACCESSES, MAIN MEMORY ACCESSES AND DISK ACCESES

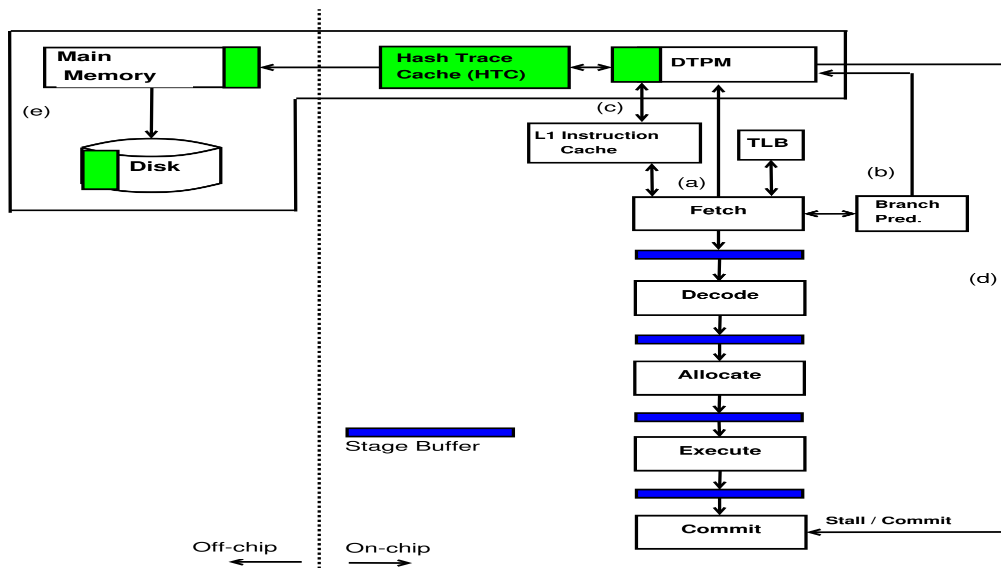

Fig. 3. Processor Pipeline with DTPM and the proposed memory hierarchy

the pre-computed hash values match, the instructions in the trace are committed. If a hash value stored in the HTC has to be replaced, it is encrypted using the platform RSA keys and stored in the lower level of hash storage hierarchy.

Based on the sensitivity study on the size of the HTC [8], we choose a 32 KB direct mapped HTC. The resulting improvement in performance is in Fig. 4. The average number of execution cycles for all the benchmarks with DTPM and HTC incorporated into the processor pipeline is 1.35 times the number of execution cycles on the basic pipeline. When the HTC is introduced, the number of disk accesses is reduced. The benchmark `403.gcc` accesses the HTC 6×10^7 times and accesses the lower level in the hash storage hierarchy that is outside the chip boundary 3.7×10^5 times. Without the HTC, there would have been 6×10^7 disk accesses. The sixth column of Table I gives the number of HTC accesses. Although the HTC reduces performance impact, its miss penalty is very high. The high miss penalty of the HTC is mainly due to the huge difference in access latency between the on-chip HTC and the off-chip disk.

D. Additional Performance Optimizations

Ideally, run-time integrity checking should not affect the overall performance. There is performance improvement when the HTC is included in the hash storage hierarchy when compared to the case when there is no HTC. But, the HTC has a high miss penalty. The latency introduced by the hash computation also affects the performance. Performance can further be improved if these two issues are taken care of. We propose the following additional performance optimizations.

1) Extending Hash Storage Hierarchy to Reduce the Miss penalty of the HTC: We propose to extend the hash storage hierarchy by using a part of the system DRAM or main memory to store the hashes. We use part of the address space of the system memory as an additional level in the memory hierarchy of hash value storage (shown by the shaded portion in the main memory in Fig. 3). On a HTC miss, the system memory will be interrogated before the disk is accessed. There are two issues that we must deal with. The first is where to store the hash values in memory. The second is how to access these hash values given a trace ID.

We propose to store the hashes in the system memory as part of the operating system address space. This is more secure because this address space cannot be accessed by the user application. However, we propose to store the hashes in encrypted form. This is because being off-chip makes them vulnerable to a TOCTOU attack.

To address the second issue, we implement a scheme similar to virtual memory management, wherein a page table is stored in memory and used to make the translations from virtual to physical pages on a TLB miss. We store a table of hashes starting with a fixed start address. This table is accessed using the trace ID. The trace ID modulo the number of entries in the table is used to get the entry in the table where the encrypted hash of the trace is stored. When the encrypted hash is fetched from the main memory (or from the disk on a miss in the table), it is brought into the DTPM, decrypted and compared against the hash of the trace computed at run-time by the DTPM. The hash storage hierarchy is shown by the region indicated by (e) in Fig. 3. The shaded regions in the main memory and the disk indicate the small portion allocated for storing the hash values. The last three columns of Table I give the number of accesses in the HTC, main memory and the disk respectively.

2) Speeding up the Hash Computation: Part of the overall latency of DTPM is to calculate the SHA-1 hash for a trace. So we expect a performance improvement if we speedup this hash calculation, using a faster hash function[3]. However, as we will see in the results section, this was true for some of the benchmarks. This is because, in case of an HTC hit, we do not recalculate the hash value for that trace. Therefore, speeding up hash calculation will not improve performance whenever there is an HTC hit. Speeding up hash calculation will improve overall performance when there is no HTC or when there are a lot of HTC misses.

IV. Experimental Setup and Results

A. Experimental Setup

A wide range of benchmarks from suites such as SPEC CPU2006, BioBench [11] and STREAM [12], have been chosen to represent different program behaviors (processing-bound and memory-bound). These benchmarks were compiled using the GNU version of GCC at O3 optimization level. For all the benchmarks, we run a representative 500M instructions. To obtain the basic block information, we profile the benchmarks with the training inputs, using exp-bbv, a basic block vector generation tool from Valgrind [13]. We used Zesto [14], a cycle-accurate out-of-order superscalar processor simulator that simulates x86 binaries. We configured the simulator to have the specifications of a single Intel Nehalem core [14]. We choose a 4 GHz processor frequency for our experiments. We assume that (i) the comparison of the hash values take 2 cycles (ii) finding the hash in the DTPM storage takes 1 cycle (iii) HTC is 32 KB in size, with an access time of 1 ns [15], which translates to 4 cycles (iv) the memory access takes 50 ns, which translates to 200 cycles (v) encrypting and decrypting the hashes takes 150 cycles (vi) and each disk access takes 2.6×10^6 cycles [8].

B. Results and Analysis

The average number of execution cycles when there is no DTPM is 7.86×10^8. However, when the DTPM is incorporated, there is a $250\times$ increase in the execution cycles. When the HTC is added to the hash storage hierarchy, the total execution cycles is now approximately 35% more than the execution cycles on a basic processor pipeline. When the main memory is included in the hierarchy, this is further reduced to approximately 18%. This performance improvement is shown in Fig. 4.

The average number of instructions per cycle (IPC) of the baseline processor is 1.211. The average IPC goes down to 0.865 when the 32 KB HTC is in the hash storage hierarchy. When main memory is added to this hierarchy, the IPC is 1.178 and is approaching the baseline IPC as seen in Fig.5.

The effect of speeding up the hash function on the overall performance is summarized in Fig. 6. As we mentioned

[3]Some of the SHA-3 round 2 candidates can compute a hash in only ten clock cycles.

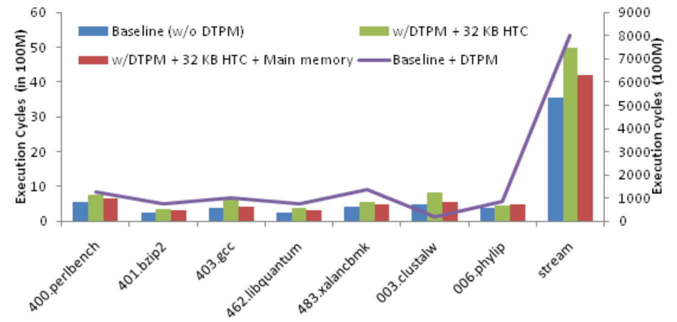

Fig. 4. Impact of different optimizations on the execution cycles

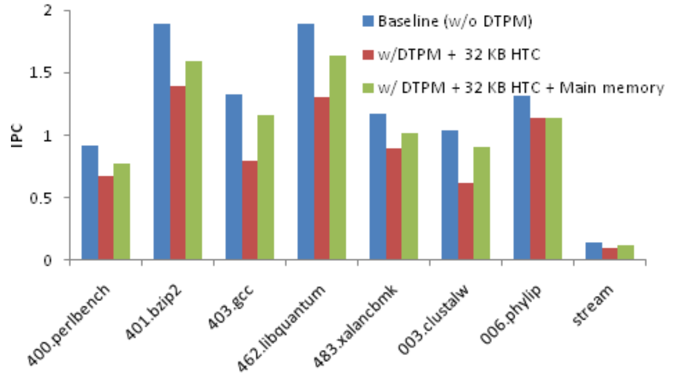

Fig. 5. Impact on Instructions Per Cycle (IPC) of the various optimizations

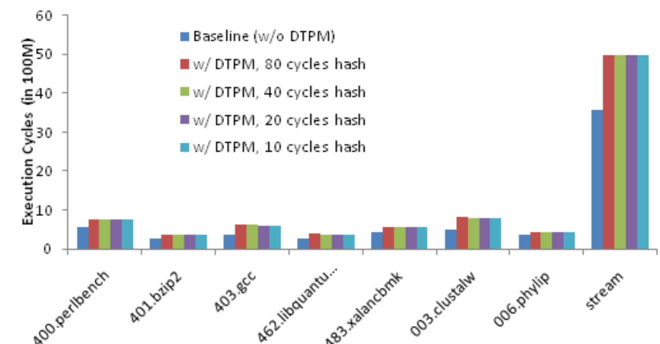

Fig. 6. Impact of different hash functions on the execution cycles

earlier, it depends on HTC performance in terms of hits and misses. `403.gcc` has the highest HTC misses, as seen from the second last column of Table I. So speeding up hash computation by 8× improves performance by 4.18%. `006.phylip` has the lowest HTC misses and also has the lowest improvement in performance due to speeding up the hash computation (0.24%). In general, the performance improvement when the hash computation is speeded up is not high. This is due to the high number of HTC hits, as shown in Table I. If HTC was not present, then the system performance would have been sensitive to hash calculation speed.

C. Comparison with related work

REM does not implement a hash storage hierarchy. The pre-computed hashes are not stored in the disk, and hence the memory requirement is expected to be high. Since the new instruction is inserted at the beginning of every basic block in the code, REM sees an 86.6% increase in the code-size for a few SPEC CPU2000 benchmarks. Our approach does not change the ISA, and does not modify the code. REM uses a small hash read buffer to store the pre-computed hashes. If the hash value goes out of the buffer, it has to be fetched again from the memory. This can degrade performance if SPEC CPU2006 benchmarks, which have several billions of instructions, are used. The 32KB HTC proposed in this paper can hold 1K hash values in comparison to REM's hash buffer.

D. Hardware Cost and Power Analysis

In this paper, we proposed an on-chip DTPM and HTC with some hashes stored in the main memory. A direct-mapped 32 KB HTC has an area of 0.2891 mm^2. The area of the RSA-2048 engine and the SHA-1 engine together is 0.26 mm^2 when implemented using 45 nm FreePDK cells. Nehalem processor core area is 24.4 mm^2 and hence, the area overhead is 2.5%. The total read dynamic power at maximum frequency is 0.0968 W. The total standby leakage power is 0.0257 W [15].

V. Conclusion

In this paper we studied the feasibility of an on-chip dynamic TPM for a single core processor. We discussed the main challenges of DTPM: high impact on performance and generation and storage of hash values. In order to make DTPM feasible, we introduced a hash value storage hierarchy that includes a Hash Trace Cache and a part of the main memory, beside the disk. We showed that a DTPM can indeed be implemented with 18% performance overhead and just 2.5% area overhead. We are currently investigating additional means to reduce the impact on performance further.

VI. 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.

VII. Disclaimer

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

References

[1] TCG, "Trusted Platform Module (TPM) Summary," http://www.trustedcomputinggroup.org/resources/trusted_platform_module_tpm_summary, April 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*, March 2008, pp. 14–32.
[3] M. Schuette and J. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Transactions on Computers*, vol. C-36, pp. 264–276, March 1987.
[4] 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.
[5] B. Gassend, G. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," in *Proc. of The Ninth International Symposium on High-Performance Computer Architecture*, February 2003, pp. 295–306.
[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] Y. Ming-yan, Z. Tong, W. Jin-xiang, and Y. Yi-zheng, "An efficient ASIC implementation of SHA-1 engine for TPM," in *Proc. of The IEEE Asia-Pacific Conference on Circuits and Systems*, vol. 2, December 2004, pp. 873–876.
[8] A. Kanuparthi, M. Zahran, and R. Karri, "Architecture Support For Dynamic Trust Measurement," http://cid-7880c71b9c4ca6e7.skydrive.live.com/browse.aspx/Public, Polytechnic Institute of NYU, Brooklyn, NY, Tech. Rep., April 2010.
[9] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," in *Proc. 30th Annual Symposium on Microarchitecture*, December 1997, pp. 138–148.
[10] J. L. Hennessey and D. A. Patterson, *Computer Architecture - A Quantitative Approach*. San Francisco, California: Morgan Kauffman Publishers, 2007.
[11] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005, pp. 2–9.
[12] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," http://www.cs.virginia.edu/stream/, University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007.
[13] Valgrind, "Valgrind exp-bbv - Basic Block Vector generation tool," http://valgrind.org/docs/manual/bbv-manual.html, .
[14] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration," in *Proc. of the International Symposium on Performance Analysis of Systems and Software*, April 2009, pp. 53–64.
[15] HP, "HP Interactive CACTI," http://www.hpl.hp.com/research/cacti, .