A Framework For Trusted Instruction Execution Via Basic Block Signature Verification

Milena Milenković, Aleksandar Milenković, Emil Jovanov
Electrical and Computer Engineering Dept.
The University of Alabama in Huntsville
{milenkm|milenka|jovanov}@ece.uah.edu

ABSTRACT

Most of today's computers are connected to the Internet or at least to a local network, exposing system vulnerabilities to the potential attackers. One of the attackers' goals is the execution of the unauthorized code. In this paper we propose a framework that will allow execution of the trusted code only and prevent malicious code from executing. The proposed framework relies on the run-time verification of basic block signatures. The basic block signatures are generated during a trusted installation process, using a signature function with secret coefficients and the address of the basic block within a program. The result of the trusted installation is the encrypted basic block signature table (BBST), which is appended to the program binary. The potential of the proposed framework is evaluated using traces of SPEC CPU2000 benchmarks. The results indicate that the proposed mechanism does not have a large negative impact on performance.

Keywords

Computer security, trusted execution, intrusion detection

1. INTRODUCTION

With more computers connecting to the Internet each day, computer system security has become a critical issue. This trend will continue in the future, with even more computer platforms connected to the Internet, including the growing number of embedded systems, from home appliances to health monitoring devices.

One of the major security problems is the execution of the unauthorized and potentially malicious code. During the execution of vulnerable programs the attacker is able to inject the code into some memory structure, for example a buffer, and then to change the code pointer, such as the return value on the stack [1]. One attack example is the so-called stack smashing: an attacker exploits a possibility for a buffer overflow in the program, by sending more data than the buffer can hold. The consequence of this attack is that a valid return address on the stack is overwritten with the malicious code address that points to the unauthorized code, also written on the stack. If the vulnerable program has root privileges, the unauthorized code will have the same privileges. Various other examples of attacks exist, such as the heap overflow and the format string attack [2].

In this paper we propose a processor architecture that will allow execution of the trusted instructions only and prevent malicious code from executing. The proposed architecture relies on the run-time verification of the signature of the last basic block in an instruction stream. A *basic block* is a straight-line code sequence with no branch instructions out except at the exit and no branch instructions in except to the entry. An *instruction stream* or dynamic basic block is a sequential run of instructions from the target of a taken branch to the first taken branch in sequence. A basic block signature is determined using a multiple input signature register, with linear feedback coefficients dependent on the processor secret key.

Our recent evaluation of SPEC2000 benchmarks indicates that an application execution encompasses a relatively small number of unique instruction streams [3], and correspondingly a relatively small number of basic blocks. Hence, a table holding all basic block signatures will occupy limited memory resources. Not all basic block signatures have to be checked, just the last one in an instruction stream. The reason is that a malicious stream cannot end with the trusted basic block, so if the last basic block in a stream is proved to be trusted, the whole stream can be trusted. The code in the cache memory does not need to be verified, which further reduces the number of verifications. In order to minimize the execution overhead, the signature verification is performed in parallel with the instruction execution. The most recently needed signatures are stored in the cache-like structure – the Basic Block Signature Table (BBST). Hence, a penalty occurs only if a signature must be fetched from the memory. The potential of the proposed framework is evaluated using traces of SPEC CPU2000 benchmarks [4], by assessing the number of the BBST misses per one million instructions.

The rest of the paper is organized as follows. Section 2 describes the related work, and the third section gives details about the proposed framework. Section 4 shows the framework potential, and the last section concludes the paper.

2. RELATED WORK

The simplest solution of the problem of malicious code execution would be to write the code that is not vulnerable to such attacks. This is of course infeasible, although the static code analysis can find a significant number of security flaws. For example, Wagner et al. propose an automated detection of code that might cause the buffer overflow [5], but the evaluation of that approach shows a relatively large number of false alarms. Another

ACM SE'04, April 2–3, 2004, Huntsville, Alabama, USA.

approach is to detect and/or prevent the execution of malicious code dynamically, in the run-time.

The research by Kirovski et al. is the most related one to our work [6]. They propose the Secure Program Execution Framework for intrusion prevention. The main idea is that the executable of a program can have different representations that produce the correct program behavior. Possible transformations include instruction scheduling, basic block reordering, branch type selection, register permutation, etc. The smallest unit of transformation is the instruction block – instructions that can fit into one cache line or prefetch buffer. During installation each instruction block is transformed in the following way. First, the domain of selected constraints is ordered. For example, in the case of instruction scheduling, a unique number is assigned to each instruction according to some policy. Then transformation-invariant (TI) hash is calculated, since some information is not dependent on the code transformation: control-data flow graphs, instruction types, values of constants, etc. The obtained hash value is then encrypted using AES or DES encryption algorithm, and the secret CPU ID as the key. The encrypted hash value defines the transformation of the instruction block. The similar process is used during program execution. The verifier component calculates the TI hash for every instruction block that is fetched after an instruction cache miss. It then encrypts the hashed value and verifies whether the obtained transformation is equal to the actual code. If there is no match, an abort signal is sent to the processor. Verifications introduce a significant overhead: for example, 33 cycles for domain ordering and TI-hash, 16 cycles for encryption, and 1 cycle for verification. That overhead can be reduced if the TI-cache is used. The approach is validated using ARM processor simulator and Mediabench set of benchmarks. With TI-cache, the performance is reduced 7.5%-17.1%. If the basic block reordering transformation is used, the code size increases on average 7.5%. This solution successfully prevents intrusion, but at the cost of relatively significant execution slowdown.

A very interesting approach is to tag all data coming from "the outside world", e.g. I/O channels, as spurious, and to prevent execution of any control transfer instruction if the target address depends on spurious data [2]. This approach may generate some false positives, since the target address may be input-dependant, for example in switch constructs. In general case, input data can propagate to a target address through a series of calculations, so this approach requires a relatively complex data dependency analysis.

Several researchers suggest intrusion detection by monitoring the system calls of a program [7][8][9][10]. If the system call sequence for a particular program deviates from the normal behavior, an intrusion is suggested. The normal program behavior is obtained either by profiling or by encoding the specification of expected behavior, using a special high-level specification language. If profiling is used, false positives may be generated when a rarely used region of the code is executed. Specification-based approach, on the other hand, is as error prone as the coding process itself. Finally, although a malicious code is very likely to encompass a system call, such as *system()* command, an attacker may potentially devise an attack with the same call sequence as the vulnerable program, or inflict some damage even without system calls. Another profiling approach [11] suggests using the values of performance monitoring registers to verify whether the program deviates from its expected behavior.

Various compilers, compiler and library patches target one or more possible vulnerabilities in the code, but so far none of them is able to cover all possible security bugs that enable malicious code execution [1]. Most of them concentrate on the detection and/or prevention of the change of the function return address on the stack. For example, the StackGuard compiler places the dummy value between the return address and the rest of the stack. A buffer overflow attack that overwrites the return address must also overwrite the dummy value.

Xu et al. propose an architectural support against the buffer overflow attack [12]. Two ideas are evaluated: to separate data and address stack, or to use a secure redundant copy of return addresses.

Our approach is somewhat similar to mechanisms used to tamper-proof the software [13], but with the different goal and different granularity. The tamper proofing protects against unauthorized changes in the code, and it usually works on the level of files or modules, while our framework works at the level of instruction streams and basic blocks, and protects against execution of the unauthorized code.

3. PROPOSED FRAMEWORK FOR TRUSTED INSTRUCTION EXECUTION

A security mechanism for trusted instruction execution should successfully prevent the execution of any unauthorized code, but the security features should not significantly increase the program execution time and the overall system complexity. In this paper we propose a framework that satisfies these requirements.

The atomic code unit protected by its signature is a *basic block*. One possibility would be to verify the signature of each executed basic block, but at the cost of an increased overhead. Assuming that the memory containing program instructions is write-protected, the number of verifications can be safely reduced if only the last basic block in an instruction stream is verified, i.e., the whole instruction stream can be trusted if its last basic block is verified. The number of necessary signature verifications can be further reduced. We can assume that the code in the cache memory can always be trusted, so the checks for signature match should be done only if there is an instruction cache miss during the execution of the last basic block in the instruction stream. The critical information on the hard disk, such as signatures, must be protected by encryption.

Figure 1 illustrates the proposed architecture for trusted program execution. The proposed framework introduces relatively modest changes in the processor organization and a new format of program executables. The processor is modified to include two hardware resources, the BBST (Basic Block Signature Table) and the BBSVU (Basic Block Signature Verification Unit). The BBST is a cache-like structure that keeps relevant information about the most recently needed basic block signatures, and the BBSTVU verifies the signatures when needed. The signature information for all basic blocks is stored in the BBST_M, prepared by the compiler and appended to the executable. The following paragraphs give a more detailed description of each phase of the proposed security mechanism: compilation and program installation, program loading in the memory, and program execution.

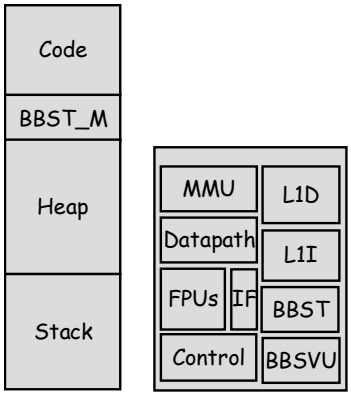


Figure 1. Architecture for trusted computing.
Legend: MMU – Memory Management Unit,
IF – Instruction Fetch Unit, FPUs – Floating Point Unit(s),
Control – Control Unit, L1D – Level 1 Data Cache,
L1I – Level1 Instruction Cache, BBST – Basic Block Signature Table, BBST_M – Basic Block Signature Table in Memory,
BBSVU – Basic Block Signature Verification Unit.

Compilation and Program Installation. The program compilation process is modified to generate the basic block signature table (BBST_M). Figure 3 shows the structure of the table. It includes the following fields: BB.SA (Basic Block Starting Address Offset) and BB.S (Basic Block Signature). The BB.SA is the address offset of the first instruction in the basic block, from the beginning of the code. The basic block signature is a function of the instruction words in the basic block. While different functions can be used for signature, we propose the use of the multiple input signature register (MISR). The MISR is frequently used in the VLSI testing, since it compresses an array of data under test into one signature, which is than compared to the signature of a known correctly functioning component. The MISR is essentially a shift register with linear feedback and external inputs. An example is shown in Figure 2. The signature of a basic block is calculated with the instruction words as consecutive inputs to the MISR. The MISR linear feedback coefficient are based on the encryption key, which is unique to each processor and hidden in hardware. The same key is used to generate the key for the signature table encryption. The encryption key can be accessed only during the trusted program installation, or during loading of the program in the memory.

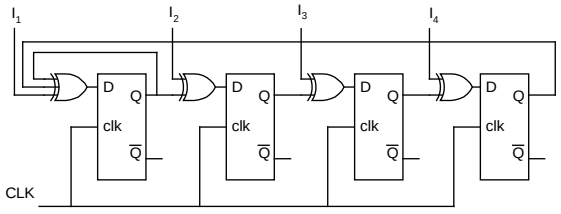


Figure 2 An implementation of a 4-bit MISR.

The library code deserves a special note. If a static library is used, only the necessary functions are linked with the rest of the application into one executable file. Basic block signatures are calculated for that file, so the signature table includes signatures of basic blocks in the used library functions. Unlike a static library, a dynamically linked library (DLL) is always completely loaded in the memory, either at load time, or in runtime. In the proposed framework each DLL has its own signature table, so all code can be safely verified. The first address of the DLL code in the memory must be kept by the system, so that address offset can be properly calculated.

At the end of compilation and program installation phase the BBST_M is encrypted using hidden processor key and some symmetric secure encryption algorithm such as AES (Advanced Encryption Standard). We do not need to encrypt the code itself, since any changes in the code will result in the signature mismatch.

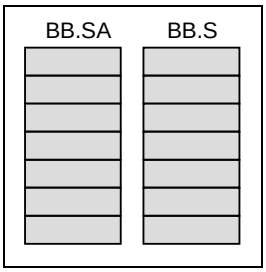


Figure 3 Basic Block Signature Table fields.
Legend: BB.SA – Basic Block Starting Address Offset
BB.S – Basic Block Signature.

***Program Loading*.** Program loading assumes decryption of the BBST_M and its loading into the main memory. A memory region with BBST_M can be accessed only by the BBSVU, and it is also write-protected. A subset of the BBST_M can be loaded into the processor's basic block signature table, and that subset can be chosen in various ways: by spatial locality, by applying profiling information, or randomly. Another option is not to preload the BBST, but to fill it dynamically, just like a regular cache structure. When a stream is executed and the BBSVU does not have the corresponding entry for the last basic block in the stream, a BBST miss is issued, and the corresponding info is brought to BBST from the BBST_M.

As long as an application and its signature table stay in the memory, there is no need for decryption process, so the decryption increases load time only. If an application has a very large signature table and/or code, or must be frequently swapped, the decryption must be performed for each load in memory. Very large signature tables can be only partially loaded, and then a special partial BBST flag must be set. If this flag is set, a miss in the memory triggers the loading of the corresponding BBST part.

Program Execution - Program-Flow Monitoring by the BBSVU. During trusted program execution, an additional set of registers keeps the track of the program flow and information relevant for basic blocks: PPC (Previous Program Counter) and CBB.SA (Current Basic Block Starting Address) (Figure 4). The decoding logic detects the instructions that can change the control flow, such as jumps, branches, returns, etc. Every such instruction marks the end of a basic block, so the next instruction is the beginning of a new basic block. If a control signal coming from the instruction decoder is set, indicating that a control flow instruction is currently being executed, than an internal flag (CntF) indicates that the difference between the address of the program start and next instruction PC will initialize CBB.SA. A signature capture block is responsible to calculate CBB.S (Current Basic Block Signature), starting from the first instruction in the block. Since the original signature is calculated using a MISR with coefficients determined by the processor key, the combinational logic for CBB.S provides the same coefficients. The CntF is also used to reset the MISR.

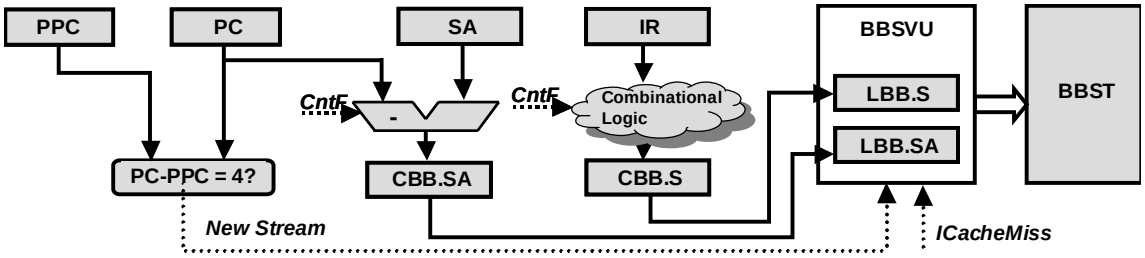


Figure 4 Program flow monitoring. Legend: PC – Program Counter, PPC – Previous Program Counter, IR – Instruction Register, SA – Program Starting Address, CntF – Control signal from the decoder, indicating the end of a basic block, CBB.SA/LBB.SA – Current/Last Basic Block Starting Address, CBB.S/LBB.S – Curren/Last Basic Block Signature Register, BBSVU - Basic Block Signature Verification Unit, BBST – Basic Block Signature Table, ICacheMiss – Instruction Cache Miss during basic block execution.

As explained before, we need to check only the signature of the last basic block in an instruction stream, and only when that basic block caused an instruction cache miss. The end of the current instruction stream is detected by comparison of the PC and PPC registers. If there was also a corresponding cache miss, current values of the CBB registers are transferred to the corresponding LBB registers in the BBSVU. The CBB registers then continue to capture the relevant information of the currently executed basic block, while concurrently the BBSVU is verifying the signature of the last basic block in the previous stream. The signature table lookup results in a miss or hit. In the case of a signature hit the program stream executed has no malicious code. A signature miss is caused by the execution of an instruction stream with no signature for the last basic block in the signature table. It could be an infrequently executed basic block, or a malicious code. If the last basic block of the previous instruction stream has an entry in the basic block table residing in the memory and there is a signature match, than the instruction stream is not malicious and the execution continues as usual. Otherwise, it means that a malicious instruction stream has been executed, so the BBSV traps the operating system. The operating system than halts the program execution, and audits the intrusion event.

At the moment when a possible intrusion has been detected the program can be well in the middle of execution of the following instruction stream. In order to reduce the overhead due to the signature checking, we do not stop the processor after each instruction stream and the checking process goes completely in the background. We assume that this delay on average will be a few instructions and that no serious harm can be made during that period of time, but a threshold of the number of streams waiting to be verified must be set. For example, we can allow only one stream to proceed with the execution during the verification of the signature for the last basic block in the previously executed stream. If we allow more than one stream to proceed unverified, we need a buffer for the last basic block signatures. Another, more secure option, is to buffer any write to the memory until the verification ends, so no untrusted stores can be executed.

When a very short instruction stream executes in parallel with the signature table lookup, the processor will be stalled until the signature processor completes the lookup. This situation can be avoided if we assume a pipelined buffer between the signature co-processor and program-flow and capture block. This implementation will slightly increase the complexity of control unit and requires extra area.

Another question is when the basic block signature should be captured, i.e., in which pipeline stage. For in-order execution, the capture block may work in parallel with the execution unit, but for out-of-order execution, the signature must be calculated for a prefetched (in-order) basic block. The system must keep track of correspondence between branches and in-order basic blocks, and check signature of the in-order basic block that includes a currently executing branch, so some additional registers are needed. Whole procedure has to be moved to early pipeline stages, e.g. after a decode stage.

4. THE FRAMEWORK POTENTIAL

In this section we will first discuss the strength of the proposed approach, and then show its potential for prevention of execution of the unauthorized code.

We use a 32-bit basic block signature, i.e., a 32-bit MISR. It means that there are 2^{32} possible combinations for the linear feedback coefficients. The attacker may have knowledge only about the program code, and not about the signatures, so it is not possible to discover the MISR function by cryptanalysis. For example, the brute force attack in a buffer overflow attack would require to overflow the buffer up to 2^{32} times to find a basic block with a signature that is accepted by the system. With a possibility of buffer overflow each second of the program execution, an attacker would need more than a hundred years for a successful attack. Nevertheless, if more security is needed, we may use longer signatures and the corresponding MISR.

The potential of the proposed framework is evaluated using SPEC CPU2000 traces for Alpha architecture [3]. Each application is traced in two segments for reference data input: the first two billion instructions (F2B), and the two billion instructions after skipping 50 billion (M2B), thus making sure that the results do not overemphasize the program initialization. We use 10 integer (CINT) and 12 floating-point (CFP) applications. The simulated BBST has 4 ways, one signature line, and 128 or 256 sets. The instruction cache size is fixed, with 64B line, 4 ways, and 128 sets. Both structures have LRU replacement policy. The BBST is filled dynamically.

Table 1 and Table 2 show the number of unique basic blocks and unique instruction streams in each traced segment. The application *176.gcc* has the maximum number of unique basic blocks, 29133 in the M2B segment, and *200.sixtrack* has the minimum number, 144 in the M2B segment. On average, CFP benchmarks have less instruction streams and basic blocks than CINT, so we may expect less misses in the BBST for CFP benchmarks.

Table 1 Basic block statistics for integer benchmarks

	No. of Instr. Streams		No. of Basic Blocks	
CINT	**F2B**	**M2B**	**F2B**	**M2B**
164.gzip	751	336	872	327
176.gcc	25416	22222	29133	25777
181.mcf	744	308	981	327
186.crafty	4122	1892	4161	1692
197.parser	4767	4200	4193	4145
252.eon	3486	588	3885	675
253.perlbmk	9034	6344	10425	7542
254.gap	3218	476	3580	542
255.vortex	5496	2644	8086	3823
300.twolf	2399	1014	2842	1195
Average	5943.3	4002.4	7476.222	5079.778

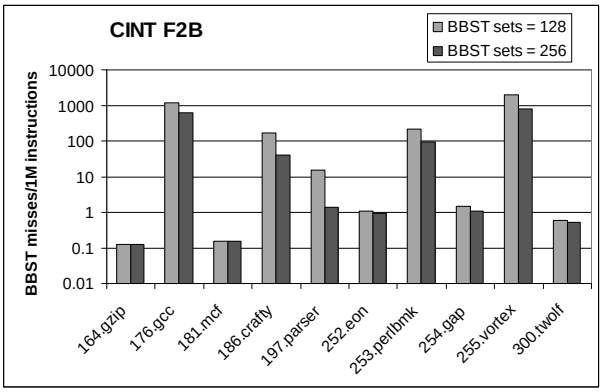
Table 2 Basic block statistics for floating-point benchmarks

	No. of Instr. Streams		No. of Basic Blocks	
CFP	**F2B**	**M2B**	**F2B**	**M2B**
168.wupwise	1563	234	2132	312
171.swim	1582	496	2268	793
172.mgrid	1457	875	1909	1082
177.mesa	1637	593	2177	763
178.galgel	1818	81	2518	166
179.art	435	341	549	502
183.equake	517	260	668	395
188.ammp	955	502	1100	566
189.lucas	964	317	1318	458
191.fma3d	2083	841	2447	1082
200.sixtrack	3532	82	4325	144
301.appsi	2439	389	2439	636
Average	1581.8	417.6	1987.5	574.9

Figure 5 and Figure 6 show the number of BBST misses per 1 million executed instructions. We use this measure instead of the BBST miss rate, because it is a better quantitative indicator of the execution time overhead due to the trusted instruction execution. On a BBST miss, the BBST_M must be searched for the missing entry. If binary search is used, the number of memory accesses in the worst case $\log_2$(BBST_M size). The average number of memory accesses can be reduced if a convenient hash function and larger BBST_M are used.

As Table 1 and Table 2 suggest, the number of BBST misses is less for the CFP benchmarks then for CINT, on average for more than two orders of magnitude. The CFP applications have not only less instruction streams, but also less instruction cache misses, and therefore less accesses to the BBST (Table 3 and Table 4).

The maximum number of BBST misses per 1M instruction is for the 128 sets, for *255.vortex*, *176.gcc*, and *253.perlbmk*, in the F2B segment: 2002, 1208, and 220, respectively. On average, the number of BBST misses is smaller in the M2B segment than in the F2B segment, probably due to the less cache misses in the M2B, when most application enter the main program loop. Doubling the number of entries of the BBST reduces the number of misses for more than half for some applications, while application with small number of misses are not sensitive to the change in BBST size.



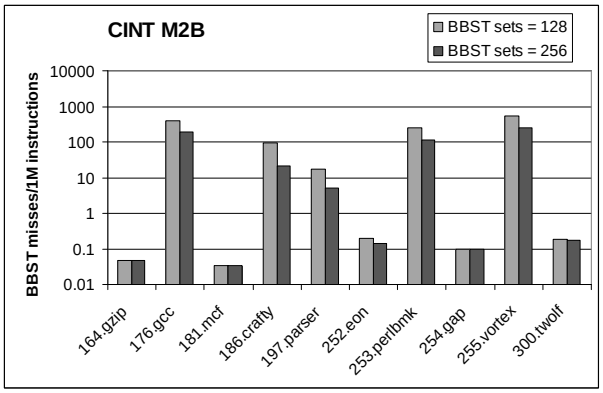
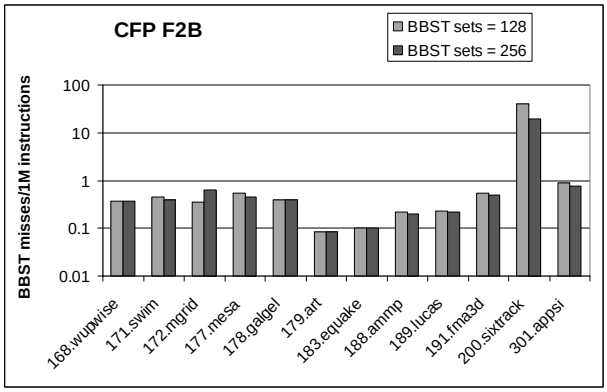


Figure 5 BBST misses per 1M instructions: CINT

Table 3 BBST Accesses, ICache Misses for SPEC2000 CINT

	ICache Miss		BBST Access	
	F2B	**M2B**	**F2B**	**M2B**
164.gzip	473	163	244	93
176.gcc	11191997	4383436	6486412	2542273
181.mcf	567	152	317	68
186.crafty	5796954	3373577	3249582	1911733
197.parser	203198	221642	136042	122151
252.eon	642317	624857	616278	610379
253.perlbmk	23773413	2639643	13212329	1437475
254.gap	735728	10634	329756	6829
255.vortex	20855690	5187588	10888457	2712472
300.twolf	88703	844	86509	403



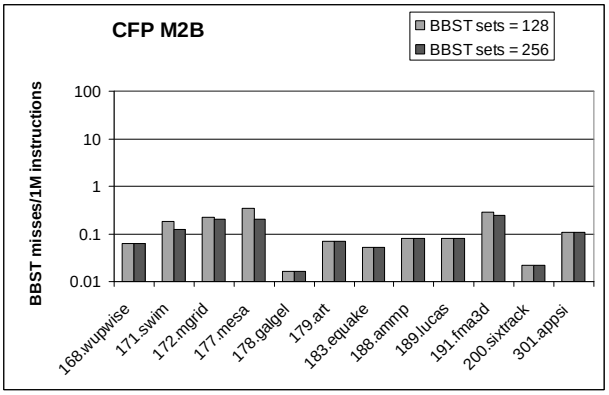


Figure 6 BBST misses per 1M instructions: CFP

Table 4 BBST Accesses, ICache Misses for SPEC2000 CFP

	ICache Miss		BBST Access	
	F2B	M2B	F2B	M2B
168.wupwise	1494	263	806	125
171.swim	5517	5443	2512	2245
172.mgrid	7919	6167	3227	2369
177.mesa	25274	32143	14695	18897
178.galgel	1826	84	893	32
179.art	363	385	167	142
183.equake	2653050	336	1776447	111
188.ammp	995	900	623	314
189.lucas	1023	556	494	169
191.fma3d	8253849	2532	4704965	1061
200.sixtrack	1743932	663	949572	87
301.appsi	4434880	4089263	1483987	1281737

5. CONCLUSION

In this paper we propose a framework for trusted program execution and evaluate its potential. The preliminary results confirm the claim that our basic block signature verification algorithm will not significantly increase the execution time.

We plan to assess the performance of even smaller BBST, and to simulate the effect of the verification algorithm on execution time on both in-order and out-of-order processors. We also plan to evaluate whether profiling information can reduce the number of BBST misses. The BBST can be preloaded with the most frequently needed signatures, with signatures used at the beginning of program execution, or randomly. Another direction of future research is to evaluate an alternative implementation of the proposed mechanism, where signatures are embedded in the code, and reside in the instruction cache memory along the instructions.

6. REFERENCES

[1] J. Wilander, M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, CA, February 2003, pp. 149-162.

[2] G.E. Suh, J. Lee, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. *Technical Report MIT-LCS-TR-912*, Computer Science and Artificial Intelligence Laboratory, MIT, 2003.

[3] A. Milenkovic and M. Milenkovic. Exploiting Streams in Instruction and Data Address Trace Compression. In *Proceedings of IEEE 6th Annual Workshop on Workload Characterization*, Austin, TX, October 2003, pp. 99-107.

[4] SPEC 2000 Benchmark Suite, http://www.spec.org

[5] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of Networking and Distributed System Security Symposium 2000*, San Diego, CA, February 2000.

[6] D. Kirovski, M. Drinic, and M. Potkonjak. Enabling Trusted Software Integrity. In *Proceedings of ASPLOS*, San Jose, CA, 2002, pp. 108-120.

[7] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 1999, pp. 133-145.

[8] I. Sato, Y. Okazaki, and S. Goto. An Improved Intrusion Detection Method Based on Process Profiling. *IPSJ Journal*, Vol.43, No.11, pp. 3316-3326, November 2002.

[9] R. Sekar, T. Bowen and M. Segal. On Preventing Intrusions by Process Behavior Monitoring. In *Eighth USENIX Security Symposium*, Washington, DC, Aug 1999, pp. 29-40.

[10] S. A. Hofmeyr, S. Forrest and A. Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*,s Vol. 6, 1998, pp. 151-180.

[11] D. L. Oppenheimer and M. R. Martonosi. Performance Signatures: A Mechanism for Intrusion Detection. In *Proceedings of the 1997 IEEE Information Survivability Workshop*, San Diego, CA, 1997.

[12] J. Xu, Z. Kalbarczyk, S. Patel and R. K. Iyer. Architecture Support for Defending Against Buffer Overflow Attacks. In *Proceedings of Workshop on Evaluating and Architecting System Dependability (EASY)*, San Jose, California, October 2002.

[13] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation–Tools for Software Protection. *IEEE Transactions on software engineering*, Vol. 28, No. 8, pp. 735-746, August 2002.