(12)

CARNEGIE MELLON

Department of Electrical and Computer Engineering

"Behavior-Based Fault Monitoring"

Principal Investigator: **John P. Shen**

Center for Dependable Systems
Department of Electrical and Computer Engineering
Carnegie Mellon University
Schenley Park, Pittsburgh PA 15213
(412) 268-3601
INTERNET address: shen@ece.cmu.edu

CMU Research Center for Dependable Systems
Carnegie Mellon University
Pittsburgh, PA 15213-3890

(12)

Updated Final Report to Office of Naval Research
Contract N00014-86-K-0507
December 3, 1990

(1)

"Behavior-Based Fault Monitoring"

Principal Investigator: **John P. Shen**

Center for Dependable Systems
Department of Electrical and Computer Engineering
Carnegie Mellon University
Schenley Park, Pittsburgh PA 15213
(412) 268-3601
INTERNET address: shen@ece.cmu.edu

Abstract

An approach is developed which exploits the deterministic behavior of a processor to perform concurrent fault monitoring. A very low cost and highly effective technique, called Continuous Signature Monitoring (CSM), has been developed. This technique is capable of detecting transients with very low detection latency, and requires very minimal memory overhead and performance penalty. This technique has been applied to both CISC and RISC type processors. Both analytical and experimental results have been obtained in validating the effectiveness of the approach. CSM has been adopted by two aerospace companies in their design of a 32-bit RISC processor targeted for avionics and space applications. It appears that the signature monitoring technique can be extended to detect computer viruses as well via a form of program encryption.

I. Summary of Accomplishments

This section presents the technical motivations for fault monitoring, summarizes our signature monitoring technique called Continuous Signature Monitoring, and compares our results with other techniques. A list of publications resulting from our current contract is provided.

1. Motivation for Behavior-Based Monitoring

Concurrent error detection is necessary to ensure reliable computer operation. Although permanent hardware faults can be detected using built-in self-test (BIST) or an external tester, concurrent detection must be used to detect errors caused by transient faults. Based on a number of experimental studies, transient faults constitute the dominant fault type in most systems during system operation.

Traditional approaches to concurrent error detection add redundancy based on a computer's structure. The most common approach is structural duplication. Although effective, duplication is too expensive for all but a few applications. Redundancy can also be incorporated via the use of error checking codes. However, most techniques based on error checking codes are only effective against very specific error types, e.g. single or double bit errors.

We propose an approach to concurrent error detection, or fault monitoring, in processors which uses behavioral abstraction of the executing program that is monitored for run-time violations. Such behavior-based approach has the advantage that errors from any source are potentially detectable, including software and hardware design faults, as well as permanent and transient faults. Abstractions can be formed using various aspects of program behavior, including control flow, memory access, input-output, and object type or range. Experimental comparison of various abstractions shows that processor control flow offers the most error-detection potential. A number of researchers have proposed techniques that detect control-flow related errors using a simple monitor and signatured programs. We called these *signature monitoring* techniques.

2. Continuous Signature Monitoring

During the past several years, we have developed a new signature monitoring approach for processor fault monitoring that uses a simple hardware monitor and signatures embedded into the executing program. Signature-monitoring techniques detect a large portion of processor control errors at a fraction of the cost of duplication. Analytical methods developed in this work show that the new approach, Continuous Signature Monitoring (CSM), makes major advances beyond existing techniques.

A signature-monitoring technique's effectiveness can be characterized by five properties: (1) error-detection coverage, (2) memory overhead, (3) processor-performance loss, (4) error-detection latency, and (5) monitor complexity. Existing signature-monitoring techniques improve upon the original basic technique in one or more of these properties. However, all of the proposed improvements degrade one or more of the other properties. CSM approach makes major improvements in all signature-monitoring properties.

CSM reduces the fraction of undetected control-flow errors by orders of magnitude, to less than 10^{-6}. The number of signatures reaches a theoretical minimum, lowered by as much as three times to a range of 4-11%. Signature cost is reduced by placing CSM signatures at locations that minimize performance loss and, for some architectures, memory overhead. CSM exploits the program memory's SEC/DED code to decrease average error-detection latency by as much as 1000 times, to 0.016 program memory cycles, without increasing memory overhead. This short latency facilitates quick recovery in the tolerance of transient faults.

Figure 1 below compares the effectiveness of the CSM technique with three other signature monitoring techniques. The basic technique is the technique originally proposed. Path Signature Analysis (PSA) was developed at Stanford. The Signatured Instruction Streams (SIS) technique was developed at CMU and is the predecessor to the current CSM technique.

	Basic	PSA	SIS	CSM
Total Memory Overhead	10-25%	12-21%	6-15%	4-11%
Latency in PM Cycles	2-5	7-17	7-17	0.016-1.0
Control-Flow Error Coverage	96-99%	99.5-99.9%	85-93%	99.9999%
Control-Bit Error Coverage	99.9999%	100%	85-93%	99.9999%

Figure 1. Comparison of CSM to Other Signature Monitoring Techniques.

3. Resulting Publications

1. M.A. Schuette, J.P. Shen, D.P. Siewiorek and Y.X. Zhu, "An Experimental Evaluation of Two Concurrent Error Detection Approaches," ***Proc. of 16th Int. Fault Tolerant Computing Symp.***, July 1986.
2. J.P. Shen and M.A. Schuette, "Processor Control Flow Monitoring Using Signatured Instruction Streams," ***IEEE Trans. on Computers***, March 1987.
3. J.P. Shen and S.P. Tomas, "A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems," ***Microprocessing and Microprogramming: The Euromicro Journal***, Special Issue on Fault Tolerant Computing, North-Holland, May 1987.
4. K.D. Wilken and J.P. Shen, "Embedded Signature Monitoring: Analysis and Techniques," ***Proc. of Int. Test Conf.***, September 1987.
5. K.D. Wilken and J.P. Shen, "Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors," ***Proc. of Int. Test Conf.***, September 1988.
6. K.D. Wilken and J.P. Shen, "Concurrent Error Detection Using Signature Monitoring and Encryption," ***Int. Conf. on Dependable Computing for Critical Applications***, August 1989.
7. K.D. Wilken and J.P. Shen, "Continuous Signature Monitoring: Efficien. Concurrent Detection of Processor Control Errors," ***IEEE Trans. on Computer Aided Design***, June 1990.
8. K.D. Wilken and J.P. Shen, "Detecting Processor Hardware Errors and Computer Viruses Using Program Encryption and Signature Monitoring," submitted to ***IEEE Trans. on Computers***, 1990.

II. Presentation of Technical Results

This section is a compendium of major papers published through the support of this research contract. These papers document the key results of our research on Continuous Signature Monitoring as well as our earlier work on Signatured Instruction Streams. In total, three journal papers and four conference papers have resulted from this work. One more paper on extending CSM to cover a more generalized fault model and to detect computer viruses has been submitted to IEEE Transactions on Computers.

Processor Control Flow Monitoring Using Signatured Instruction Streams

MICHAEL A. SCHUETTE, STUDENT MEMBER, IEEE, AND JOHN PAUL SHEN, MEMBER, IEEE

Abstract—**This paper presents an innovative approach, called signatured instruction streams (SIS), to the on-line detection of control flow errors caused by transient and intermittent faults. At compile time an application program is appropriately partitioned into smaller subprograms, and cyclic codes, or signatures, characterizing the control flow of each subprogram are generated and embedded in the object code. At runtime, special built-in hardware regenerates these signatures using runtime information and compares them to the precomputed signatures. A mismatch indicates the detection of an error. A demonstration system, based on the MC68000 processor, has been designed and built. Fault insertion experiments have been performed using the demonstration system. The demonstration system, using 17 percent hardware overhead, is able to detect 98 percent of faults affecting the control flow and 82 percent of all randomly inserted faults.**

Index Terms—**Control flow monitoring, error detection coverage and latency, fault insertion experiments, roving monitoring, signature analysis, signatured instruction streams, transient and intermittent faults.**

I. INTRODUCTION

TRANSIENT and intermittent faults as defined in [2] play a major role in undermining the reliability of digital systems. It is estimated that they occur 10 to 30 times more frequently than permanent faults [18]. When testing for transient and intermittent faults, the system must be tested in its operational environment and concurrent with its execution of the application task. This type of testing is referred to as concurrent testing or on-line monitoring.

There is much interest in developing on-line monitoring schemes for general-purpose processors. Traditionally, massive redundancy, e.g., duplication or triplication, and error-detecting codes are used to implement on-line checking [5]. Error coding techniques can be extended to the design of self-checking logic circuits [21]. Error checking schemes based on codes usually assume rather restrictive error models and have limited fault coverage. Massive redundancy is an effective means of detecting transient errors. However, massive redundancy can be very costly and can reduce reliability when the redundancy is exhausted.

Recently, the idea of using a small amount of added hardware and/or software to continuously monitor the operation of a general purpose processor has become popular. Masson *et al.* suggested certain abstractions of the correct behavior of a processor [16] and proposed mechanisms to expose deviations from these abstractions. Lu proposed an approach called structural integrity checking which uses a watchdog processor to check the correctness of high-level control flow structures at runtime [11]. Namjoo and McCluskey proposed the use of a watchdog processor to detect malfunctions which cause illegal access to the memory subsystem [15]. A method introduced by Sridhar and Thatte [19] and the path signature analysis method proposed by Namjoo [14] both involve the encoding of the instruction stream at compile time and using this code at runtime to check the program control flow. Several other techniques for checking program control flow have been proposed [22], [4], [3], [12], [9].

This paper presents another processor monitoring approach called signatured instruction streams (SIS). The SIS approach focuses on the monitoring of program control flow in real time and mission-oriented systems. The overall research project has the following features and contributions.

1) Unlike most traditional fault-tolerant computing techniques, SIS employs a combination of hardware and software techniques. The combination of these techniques results in an approach which has low hardware and performance overhead yet achieves a high degree of fault coverage.

2) Although the SIS approach is similar to the methods presented in [11] and [14], it does not require the use of a watchdog processor. Instead, it uses a small, built-in monitor. Furthermore the SIS scheme reduces the memory overhead by using a technique called branch address hashing (BAH).

3) An implementation of the SIS approach has been applied to an actual processor, the MC68000, in order to demonstrate its feasibility and practicality. All the necessary software tools have been developed so that the entire approach is completely transparent to the application programmer.

4) A hardware demonstration system based on the MC68000 processor has been built and is fully functional. This demonstration system has been used in conjunction with a programmable hardware fault inserter in the performance of extensive fault insertion experiments to accurately determine the transient error coverage and the error detection latency of the SIS approach.

Section II of this paper presents the basic concepts of the SIS approach. Section III describes the SIS implementation details for the MC68000 processor and the MC68000-based SIS demonstration system. In Section IV, the fault insertion

Manuscript received December 8, 1984; revised December 20, 1985. This work was supported by IBM and by the Semiconductor Research Corporation under Contract SRC-83-01-022, and by the Office of Naval Research.

The authors are with the Department of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 8612055.

experiments involving the SIS demonstration system are described. Based on the analysis of the results from the experiments the error coverage and detection latency of the SIS approach are determined. Finally, Section V describes several extensions to this work which have been or are being performed. Section VI summarizes the major contributions of this paper.

II. Basic Concepts

On-line monitoring approaches can be viewed as spanning a hardware to software spectrum. At the hardware end of the spectrum are the purely hardware approaches such as hardware N-modular redundancy (NMR). Typically, the hardware approaches have low performance degradation but high hardware overhead. At the software end of the spectrum are the purely software approaches like that of Yau and Chen [22] and Chen and Avizienis [6]. These approaches incur substantial performance degradation and relatively low hardware overhead. The SIS approach lies between these two extremes. Both software and hardware techniques are used, so that the software is used to reduce hardware overhead while hardware is used to reduce performance degradation.

A. Basic Program Partitioning Concepts

The principal aim of the SIS approach is to check the correct sequencing of instructions of an application program. This can be done by partitioning the program into blocks of instructions. The blocks are chosen such that they have only one entry point and there exists only one valid sequence of instructions from the entry point to any of the exit points. The instructions of each valid sequence are encoded into a signature which is then stored at the end of that sequence. At runtime, the signatures for each block are regenerated by using the actual instructions as they are fetched from memory. By comparing the signatures generated at runtime to those generated at compile time, the correct sequencing of instructions within each block can be checked. Fig. 1(a) shows an example of a program that has been partitioned into blocks and a signature embedded at each exit point of each block.

The memory overhead due to the embedding of signatures can be significantly reduced through a technique called branch address hashing (BAH) [17]. BAH is used whenever the exit point of a block corresponds to a branch instruction. BAH involves hashing, e.g., bit-wise Exclusive-OR, the associated signature with the branch address of the branch instruction. Consequently no additional memory word is used for storing the signature; instead, the signature bits are encoded with the branch address. Fig. 2 contrasts the memory overhead of using only signature embedding with that of using signature hashing. Fig. 1(b) shows the same program as in Fig. 1(a) but with BAH used.

At compile time, branch address hashing causes the branch address in the object code to be incorrect. However, at runtime the runtime generated signature is used to rehash the hashed branch address. If the runtime generated signature is error free, then the rehashing will return the branch address to its original correct value. This rehashed correct address is then used by the processor.

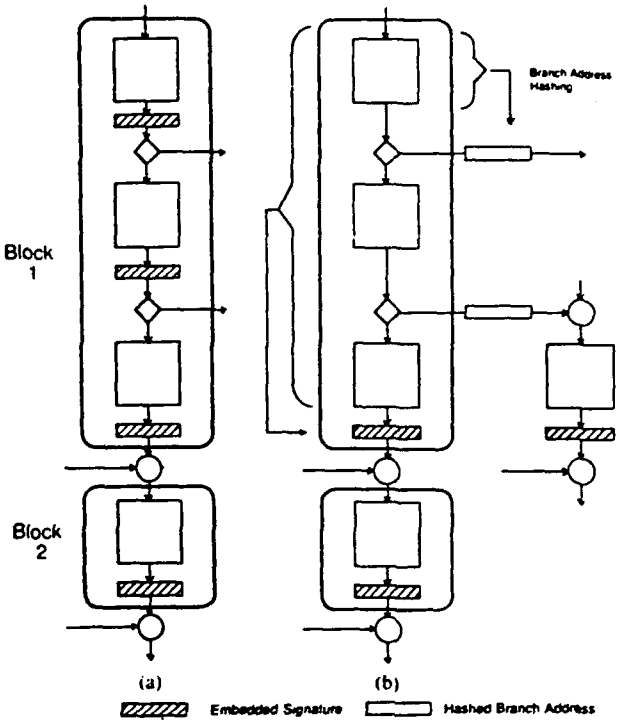


Fig. 1. Basic partitioning.

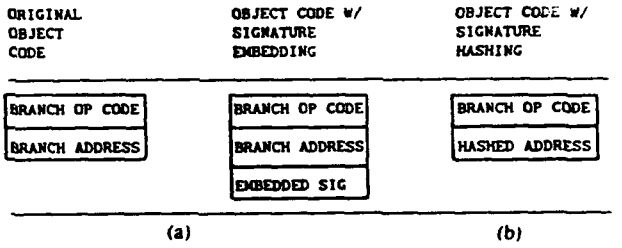


Fig. 2. Reduction in memory overhead due to signature hashing.

If the runtime generated signature differs from the compile time generated signature, the rehashed branch address used by the processor will be incorrect. If the branch is taken, then control flow will go to an erroneous destination. In this case the error will be detected at the next exit point where there is an explicitly embedded signature. It is possible for the erroneous destination to lie outside the address bounds of the program. To promptly detect such an occurrence, a program bounds detector should be employed.

B. Partitioning Concepts in Detail

In order to ensure that there is only one valid sequence of instructions between the entry point and each exit point of a block, the control flow of the application program must be analyzed. All possible valid sequences of instructions of an application program can be represented by a directed graph, called the *program graph*. Each node in the graph represents a single instruction and each arc represents valid control flow between two instructions. Checking of control flow involves verifying that the sequence of instructions executed by the processor corresponds to a path in the program graph. For example, given the program in Fig. 3(a), the instruction

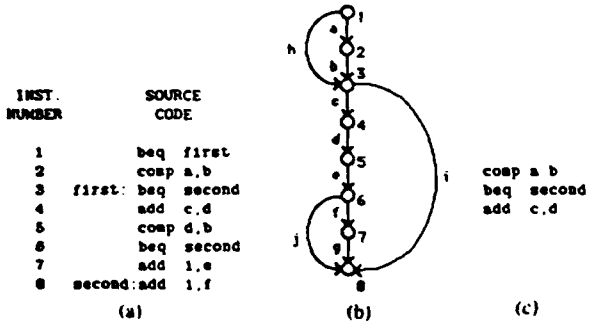


Fig. 3. Example program graph.

sequence given in Fig. 3(c) corresponds to a directed path, 2-*b*-3-*c*-4, in the program graph. A node in a program graph is called a *merge node*, as in [19], if it has more than one incoming arc. By disconnecting from every merge node all of its incoming arcs, a program graph is partitioned into a *collection of disconnected subgraphs*. As long as the original program graph has at least one merge node, it can be shown that each subgraph is connected and contains exactly one merge node of the original program graph. Because incoming arcs to merge nodes are removed, each node in a subgraph has one and only one incoming arc, except the merge node which has none. Using this and the fact that each subgraph is connected it can be shown that each subgraph is a rooted tree with the merge node being the root node and all the arcs being *directed away from the root node*.

With such a partition, each subgraph corresponds to a set of instructions or a subprogram. These subprograms are similar to the blocks mentioned in the previous subsection. Each such subprogram has exactly one entry point, corresponding to the merge node, and one or more exit points, including but not necessarily limited to the leaf nodes of the subgraph tree. Since each subgraph is a tree, there is a unique path, and hence a unique sequence of instructions, from the entry point to each of the exit points. The instructions associated with each path can be cyclically encoded into a signature which can be stored at the corresponding exit point. Control flow through a subprogram can be checked by checking the signature at each exit point of the subprogram.

Without loss of essential control flow information, the program graph can be compacted into a streamlined equivalent graph called the *control flow graph* (CFG). Each subgraph of the program graph is represented as a single node in the CFG. Arcs in the CFG represent valid control flow between subgraphs. Each arc also corresponds to an exit point of a subgraph, represented by the source node of the arc, and hence has a signature associated with it.

An example of a program segment and its corresponding program graph and CFG is shown in Fig. 4(a), (b), and (c), respectively. In Fig. 4(d) the set of instructions comprising each subprogram is shown. Instructions corresponding to entry and exit points are noted. Notice that there exists valid control flow between instructions 4 and 1 in node 3, creating what would appear to be a cycle in the subgraph represented by this node. However, the arc representing control flow from instruction 4 to instruction 1 was removed when the program

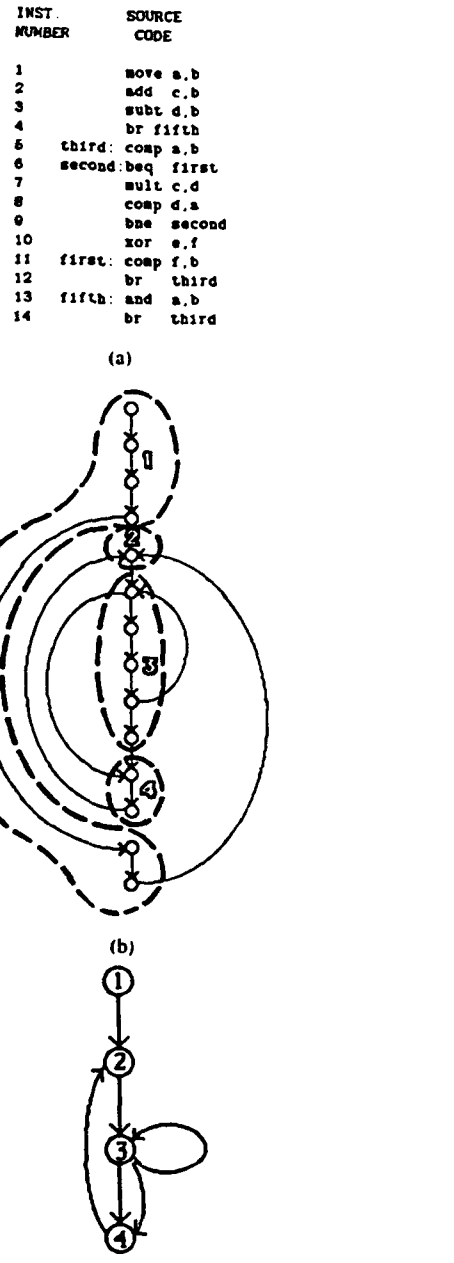


Fig. 4. CFG construction from the application program.

was partitioned into subgraphs. As a result, this arc exists as a loop in the CFG and not as an arc in the subgraph.

C. Generation and Embedding of Signatures

The actual partitioning of a program into subprograms and the generation of signatures for each subprogram can be performed by the assembler and loader such that it is

ompletely transparent to the application programmer. Additions and modifications to both the assembler and the loader re needed. The assembler must identify all the instructions orresponding to merge nodes in the program graph and artition the program into maximum-sized subprograms. The ssembler next needs to identify all of the instructions orresponding to exit points of each subprogram and reserve nemory locations in the object code for storing the signatures. After the object code has been relocated, the loader must then generate a signature for each exit point and embed or hash it nto the program code. It may place the signature next to its exit point or place information in the program code linking the exit point to its signature. A mechanism must be provided to ensure that the signature, or the linking information, will not be executed by the processor as a normal instruction.

As an example, the program of Fig. 4(a) has its object code modified from that shown in Fig. 5(a) to that shown in Fig. 5(b). In this example extra space is created by the assembler in the object code immediately after the exit point instructions 5 and 10. The remaining exit point instructions, 6, 9, 12, and 14, are branch instructions and so have their branch addresses hashed. Each of the signature embedding locations is then filled by the loader with the signature of the path associated with the exit point. The words in solid boxes in Fig. 5(b) are the embedded signatures. Those in dashed boxes are hashed branch addresses.

INST. NUMBER	ORIGINAL OBJECT CODE	SOURCE CODE	OBJECT CODE W/EMBEDDING AND HASHING
1	2fab	move a,b	2fab
2	5dcb	add c,b	5dcb
3	5edb	subt d,b	5edb
4	6000 000c	br fifth	6000 000c
5	77ab	third: comp a,b	77ab [3478]
6	6100 0006	second: beq first	6100 [F5E4]
7	abcd	mult c,d	abcd
8	77da	comp d,a	77da
9	6200 fffc	bne second	6200 [F2bd]
10	b1ef	xor e,f	b1ef [f486]
11	77fb	first: comp f,b	77fb
12	6000 fff7	br third	6000 [F450]
13	b3ab	fifth and a,b	b3ab
14	6000 fff4	br third	6000 [Fsaa]
	(a)		(b)

Fig. 5. Program code with signatures and hashed branch addresses.

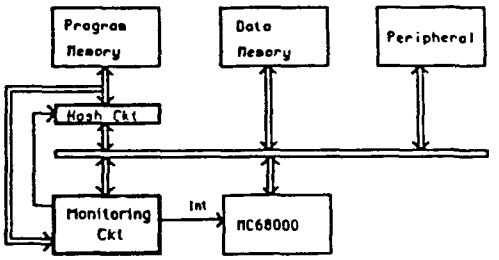


Fig. 6. Generic SIS system configuration.

D. Typical SIS Monitoring Scenario

In a self-monitoring SIS system, monitoring hardware is added to the processor as shown in Fig. 6. During normal operation, while the processor fetches instructions from the program memory, the monitoring hardware encodes these instructions into the runtime signature register. The monitoring hardware also detects the occurrence of an exit point instruction. When such an instruction is detected, the signature in the runtime signature register is compared to the embedded signature fetched from program memory. After each successful comparison the runtime signature register is reset and the signaturing of a new instruction sequence commences. A mismatch of signatures indicates the occurrence of a control flow error. When a control flow error is detected, the monitoring hardware will signal the processor by generating an interrupt to the processor. The processor can then invoke an error handling routine or notify other fault tolerant hardware in the system in order to recover from the error.

III. MC68000 Implementation

This section presents the implementation of the SIS concepts for an actual processor, namely the Motorola MC68000. Based on the implementation details presented in this section, a self-monitoring MC68000 incorporating SIS has been designed and built. This demonstration system is fully functional.

A. MC68000 Features

The MC68000 is a general-purpose processor with an external 16-bit data path and an internal 32-bit data path. Its address space is broken up into four distinct spaces, each 16 Mbits in size, that are addressed based on the values of three function code lines. The four address spaces are: user program and data spaces, and supervisor program and data spaces. Both program spaces are read only. Instruction op codes, immediate data, absolute addresses, displacements, and data referenced via program counter relative addressing modes all reside in the program space. All other information resides in the data space. There are 19 registers internal to the MC68000: a status register, a stack pointer register, a program counter, eight address registers, and eight data registers.

Each MC68000 instruction consists of one or more instruction words. The first instruction word is the op code word and all succeeding instruction words are extension words. Extension words contain such information as operand addresses and immediate data which cannot be included in the op codeword. The word size is 16 bits. It is not possible to differentiate between the fetching of an op code word and an extension word by using the information supplied at the IC package pins of the MC68000 processor chip at the time of the fetch. The MC68000 processor always performs prefetching of the next instruction word.

B. Modifications to the Assembler and Loader

The existing MC68000 assembler and loader have been modified to support SIS requirements. The original assembler has been modified to enable it to construct the CFG of an application program. It does this by adding extra information to each entry in the symbol table. The extra information increases the storage space required by the symbol table by 50 percent. The assembler depends heavily on the use of labels to indicate potential merge points. In addition, the assembler allocates memory locations in the object code for the embed-

ding of signatures and hashed branch addresses. The modifications resulted in an expansion of the assembler code from 54000 bytes to 68000 bytes, a 26 percent increase. The original assembler was a two-pass assembler. It now requires three passes.

Signatures cannot be generated until relocation of the object code has been performed as relocation alters some of the bits in the object code. The signatures are 16-bit cyclic codes, using the generator polynomial $x^{16} + x^{12} + x^3 + x + 1$. This primitive polynomial was chosen because it has relatively few terms which tends to reduce the amount of hardware required in the linear feedback shift register implementation. A routine is added to the loader to generate signatures and then either embed or hash them into the object code once relocation has been completed. This process requires an arbitrary number of passes depending upon the branching structure of the object code. Typically, for code produced by a compiler, the number of extra passes is two. For hand written assembly code this number is usually one. Modifications to the loader resulted in an expansion of the loader code from 23000 bytes to 31000 bytes. The entire signature embedding process is transparent to the user.

The embedded signatures must be fetched from program memory so they may be used by the monitoring hardware. However, the embedded signatures must be prevented from being executed as instructions by the processor. This is accomplished in the following way. Following each exit point instruction, a one-word unconditional branch instruction with a branch displacement of two bytes, henceforth referred to as a *pseudobranch* instruction, is inserted in the program code. The 16-bit signature is then stored in the word immediately following the one-word pseudobranch instruction, see Fig. 7. As it is executed, the pseudobranch effectively causes the MC68000 to skip over the next word, that is, the word containing the embedded signature. However, because the MC68000 always performs a one word prefetch, the signature will always be fetched from memory. The monitoring hardware uses the bit pattern of the pseudobranch instruction as a flag indicating that the next word to be fetched is an embedded signature. As it is fetched, the signature is latched by the monitoring hardware and used for signature comparison.

Frequently the exit point of a subprogram is a return from subroutine (RTS) instruction. In the case of an RTS instruction the pseudobranch instruction can be omitted and the signature can be embedded immediately after the RTS. In this case, the monitoring hardware must be designed to also recognize the bit pattern of the RTS instruction.

For exit points which are branch instructions with 16-bit branch addresses, BAH can be used instead of signature embedding to reduce the memory overhead. MC68000 instructions for which BAH can be performed are listed in Fig. 8. Of course the monitoring hardware must be made capable of recognizing all of these instructions so as to perform the branch address rehash when a hashed address is fetched.

Interrupts produce control flow which is not deterministic at compile time. Such control flow can occur at arbitrary locations within the program, causing temporary suspension of that program's execution. To allow interrupts in a self-monitoring processor with SIS, the monitoring hardware is made capable of pushing the current state of the runtime signature register onto a signature stack. Signaturing of the interrupt service routine is performed as though it were a separate program. When the interrupt service routine is finished, the monitoring hardware pops the previously stored signature off the stack and resumes signaturing of the interrupted program.

PSEUDO BRANCH

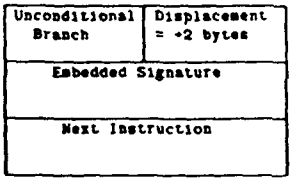


Fig. 7. Implementation of the embedded signature.

INSTRUCTION	SOURCE CODE	OBJECT CODE
long branch	bra(16)	6000 YYYY
long subr. call	bsr(16)	6100 YYYY
cond. long branch	bcc(16)	6x00 YYYY
jump	jmp	4EEx YYYY
subr. call	jsr	4EAx YYYY 4EBx YYYY

YYYY - Hashed Branch Address

Fig. 8. Possible BAH MC68000 Instructions.

Both the modified assembler and loader are fully functional and a number of example programs have been assembled. The memory overhead required due to signature embedding for three example programs is illustrated in Table I. As can be seen from this table, for typical programs with 25 percent branch instructions, a quite reasonable 10 percent memory overhead can be expected. If the instruction format of a processor is designed with SIS taken into consideration, this memory overhead for signature embedding can be significantly reduced.

C. Monitoring Hardware

The SIS monitoring hardware consists of eight functional units: a prefetch queue, a signature generator, an address rehasher, an op code decoder, a signature stack, an out-of-program bounds detector, a program counter emulator, and a controller. Refer to Fig. 9 for an illustration of the hardware during the subsequent explanation of its operation.

1) Normal Operation: The normal operation of the processor involving the monitoring hardware is outlined below. First, the MC68000 places an address on the address bus. The out-of-program bounds detector checks this address against the values of the upper and lower program bounds stored within it. If either of these bounds is exceeded, an error is signalled. Otherwise, data are placed on the data bus and pass unaltered through the rehash unit. If an op codeword is fetched, the op code decoder determines the length of the instruction. This is so that the monitoring hardware can

TABLE I
SIS MEMORY OVERHEAD

MC68000 PROGRAM	1	2	3
PROGRAM SIZE WITHOUT SIS (in bytes)	1176	2759	4387
PROGRAM SIZE WITH SIS (in bytes)	1298	3019	4785
% BRANCH INSTRUCTIONS (incl. rts)	26.2	23.7	22.3
% MEMORY OVERHEAD	10.3	9.4	9.1

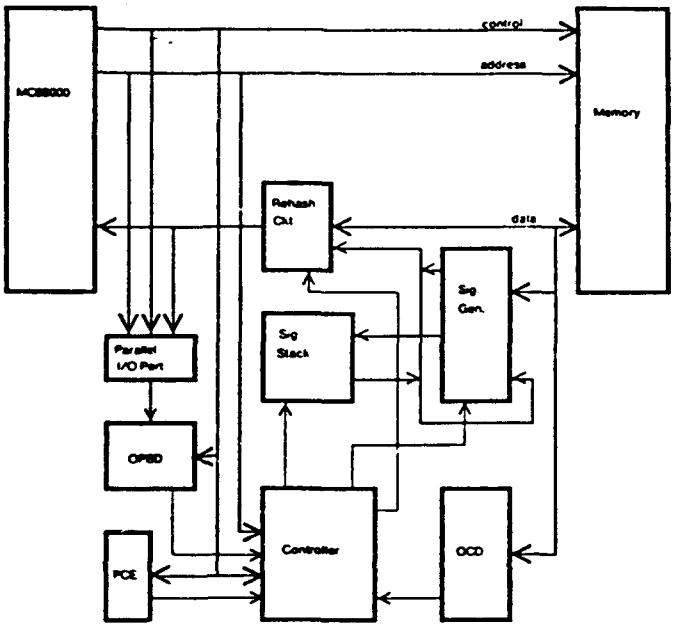


Fig. 9. The SIS monitoring hardware.

determine when the next op codeword will be fetched. In parallel with the data bus contents being latched by the MC68000, they are also latched by the prefetch queue within the signature generator of the monitoring hardware. The purpose of the prefetch queue is to prevent words from being encoded by the signature generator until the MC68000 has actually used them. Exceptions may prevent prefetched words from being used by the MC68000. As each instruction word is fetched, the contents of the program counter emulator is incremented. The fetch of the next instruction word is detected when the contents of the program counter emulator match that of the address bus.

2) Branch Instructions: The fetching of branch instructions requires additional actions from the monitoring hardware. Different actions are required for pseudobranches, short branches, and long branches. A pseudobranch is used to signify the embedding of a signature and to prevent the processor from executing the embedded signature as an instruction. Whenever the op code decoder indicates that a pseudobranch instruction is fetched, the next word fetched from program memory will be a signature. After the signature is fetched it is immediately encoded by the signature generator. The 16-bit bit pattern stored as the embedded signature is actually the inverse of the cyclic code. Hence, if no error has occurred, the encoding of the fetched signature will make the runtime signature register contents become zero. Thus only zero detection hardware is needed for error checking. This also removes the need to clear the signature register after checking is performed.

Long branches have their branch addresses in extension words. All such branches have their branch addresses hashed. When the op code decoder signals that a long branch op code is being fetched, the controller directs the rehash circuit to rehash the next word fetched from program memory using the contents of the signature register. Since the next word fetched from program memory is the hashed branch address, it will be rehashed to its correct value if no errors have occurred. To keep the contents of the program counter emulator consistent with the processor's program counter, the monitoring hardware must detect when a branch is actually taken by the program. This is accomplished by comparing the contents of the program counter emulator with those of the address bus after a long branch instruction is executed. A mismatch indicates that a branch has been taken. The program counter emulator is then loaded with the new program counter value. No additional action is taken otherwise.

Short branches have branch addresses located in the op codeword. It is difficult to detect such instructions and rehash their branch addresses before the processor receives the instruction word. Therefore, branch addresses of short branches are never hashed. Consistency of the program counter emulator contents is maintained in the same manner for long branches.

Conditional short branches to the program counter + 2 are altered to conditional short branches to the program counter + 4 with an NOP inserted. The reason being that the monitoring hardware is unable to determine whether or not branching occurred for branches to the program counter + 2. Due to prefetching by the MC68000 the fetching sequence of subsequent instruction words will be the same in both cases.

3) Exceptions: Occurrences of exceptions are recognized by the monitoring hardware controller when the processor executes a unique sequence of memory fetches. When an exception occurs, the contents of the signature register are pushed onto the signature stack and the signature register is cleared. When the first word of the exception routine is fetched, the program counter emulator is loaded with the contents of the address bus and incremented immediately. Signaturing and checking of the exception program follows. Interrupts are a special class of exceptions and are handled similarly.

4) Return from Exception and Return from Subroutine: The return from exception (RTE) and return from subroutine (RTS) instructions require special actions. The word in program memory immediately following the RTE/RTS instruction is always an embedded signature. When the op code decoder signals that an RTE/RTS is being fetched, signature checking is done in the same manner as with the pseudobranch. In the case of the RTE instruction a POP operation is performed on the signature stack after checking is done. The program counter emulator is loaded with the address of the first instruction fetched after the embedded signature is fetched. Most exception processes are terminated with an RTE

instruction. To simplify the hardware, the execution of an RTE instruction always causes a POP operation to be performed on the signature stack. Thus, returns from interrupt exceptions do not have to be differentiated from other returns from exceptions. As a result, all exceptions cause the signature register contents to be pushed onto the signature stack. Our current implementation will not support a multiprocessing environment where returns from interrupts do not return the processor back to the interrupted routine.

5) Status Register Altering Instructions: Instructions which use the processor status register as their destination operand cause the MC68000 to do an instruction refetch. When the op code decoder indicates that such an instruction has been fetched, the next program fetch is ignored by the monitoring hardware. The word is not encoded by the signature generator nor is the program counter emulator incremented. Upon refetch of the word, normal operation is resumed.

D. Overhead Summary

A self-monitoring demonstration system, based on SIS and a 4 Mhz MC68000 processor, has been designed, constructed, and is fully functional. The SIS monitoring hardware realization requires a total of 172 TTL SSI/MSI packages. The realization requires 3947 gates and 5435 bytes of memory. Compared to the approximately 23000 gates of the MC68000 processor, this represents an overhead of 17 percent in gate count and a higher overhead when including memory. The SIS hardware is highly modular, making it amenable to implementation on a single LSI chip or on the processor chip itself. Such an implementation would allow the SIS hardware to have a minimal impact on the system interconnect. Much of the memory overhead is unnecessary if the monitoring hardware is incorporated on the MC68000 chip or if the start of an instruction fetch were readily identifiable from information at the MC68000 pins. The basic clocking rate of the processor is unchanged because the added SIS hardware is not in any of the critical delay paths. Processor performance degradation, due to the added pseudobranch instructions, is estimated to be about 10 percent although this number is highly program dependent. If SIS is considered in the original design of the processor, these overhead figures can be significantly reduced.

Although the SIS monitoring hardware was implemented for the MC68000, it should be easily extendible to other processors as well. The only portions of the SIS monitoring hardware that are MC68000 dependent are the op code decoder and the prefetch unit for the signature generator. The main purpose of the op code decoder is to allow the SIS monitoring hardware to determine when a new instruction begins in the program memory so that special instructions, like the pseudobranch, can be identified. The op code decoder will be less complex for processors which provide direct external information concerning the start of an instruction fetch. The prefetch unit may be more complicated depending on the size of the prefetch queue in the monitored processor. A larger prefetch queue will complicate the determination of whether an instruction has been executed or not. This is important for purposes of determining when an instruction should be used as input to the signature register. If the processor does not have prefetching, then a technique other than the use of pseudobranches to permit signature embedding will be needed.

IV. Error Coverage Analysis

Transient errors are not very well understood. Hence, determination of transient error coverage based on an analytical model is difficult. However, attempts have been made to analyze error coverage analytically using very restrictive error models [13], [14]. The accuracy of these error models may be difficult to verify. An alternate means of determining the transient error coverage is by simulation. This requires the availability of a logic simulator and detailed models of the hardware in the system, the MC68000 in particular. Such models were not readily available at the time the experiments were performed.

An assessment of the error coverage can be obtained using hardware fault insertion experiments. Hardware fault insertion experiments have been used previously [16]. This is the approach used for determining the error coverage of the SIS approach. A general-purpose fault inserter (GPFI) was designed and built to insert faults into the self-monitoring MC68000 demonstration system. The GPFI required only three months to design and build. It uses a single board microprocessor and a small amount of custom built hardware which consists of 49 MSI/SSI chips with a total gate count of approximately 1200 gates. The results of the fault insertion experiments are analyzed to show the significant improvement in transient error and intermittent fault coverage by incorporating the SIS approach into an MC68000-based system. Analytical results are obtained which show the effect of SIS on the overall reliability of the system.

A. The Fault Inserter

A programmable hardware fault inserter has been designed so that the faults inserted could be selected based on several parameters. Faults are inserted on the external data, address, and control busses of the MC68000 chip. Each fault can be inserted on any one of 11 data, address, and control lines. The duration of the fault can be selected to be 1, 2, or 4 bus cycles. Insertion of the specified fault can be initiated when a preselected address appears on the MC68000 address bus or a selected number of bus cycles later. If an error caused by the fault is detected, then the fault inserter is able to record the detection mechanism, either one of the already existing MC68000 mechanisms or the SIS monitor, based upon information supplied by the monitored MC68000. The fault inserter is also able to record the error detection latency, that is, the time from when the fault is inserted to the time when it is detected.

Fig. 10 illustrates the incorporation of the GPFI into the MC68000 system. The GPFI is interposed between the MC68000 processor and its memory. Thus, faults are inserted on the various busses prior to reaching the address decode circuitry. Specification of the faults to be inserted is done through a terminal interface provided by the single-board microprocessor. This microprocessor is also linked with a

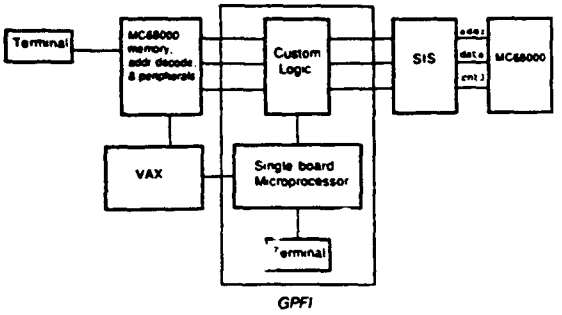


Fig. 10. Incorporation of the GPFI in the MC68000 system.

VAX® 11/785 to permit permanent storage of the fault insertion results.

B. Similarity of Inserted Faults to Actual Faults

Before making comparisons between the inserted and actual faults, the basic structure of the monitored system must be examined. For purposes of this examination, it is assumed that the monitored system is composed of only the MC68000 processor and its memory. It is not expected that the inserted faults will closely model faults in peripheral devices.

The MC68000 can be assumed to consist of a data and a control path. The data path consists of the ALU, data registers. The control path consists of the op code decoder, controller, and program control unit (PCU). The PCU contains the bus interface circuitry (BIC), program counter (PC), instruction prefetch queue, and a controller. The PCU contains circuitry for address calculations. The PCU is assumed to be responsible for the reading and writing of operands and instructions. Interrupt handling and bus arbitration circuitry will not be considered because the inserted faults will not closely model faults within these units.

1) Instruction Cycle-Data Bus Faults: Faults inserted on the data bus, while an instruction word is being read, model faults in the memory and external data bus. They do not model faults within the MC68000 because the inserted faults appear external to the MC68000. There they can be directly observed by the SIS monitoring hardware.

2) Data Cycle-Data Bus Faults: Faults inserted on the data bus, while an operand word is being read or written, do model faults internal to the MC68000. This is due to the fact that the SIS hardware does not monitor the data bus at this time. Actual faults that are modelled include: 1) faults on the external data bus and memory; 2) faults on the internal data bus and the data bus section of the BIC; 3) faults in a data register, if a data register is being written to or read from; 4) faults in the PC or address calculating circuitry, if the operand is used in determining a branch address; and 5) faults in the ALU, if the operand is used by the ALU or is the output of the ALU.

3) Instruction Cycle-Address Bus Faults: Faults inserted on the address bus, while an instruction word is being read, model: 1) faults in the PC and address bus section of the BIC; 2) faults in the PCU that cause incorrect address calculations; 3) multiple faults in the op code decoder or controller that cause the instruction to be misinterpreted; 4) multiple faults in the external data bus, memory, and memory select circuitry; 5) multiple faults in the PC or address calculation circuitry, if the operand is used in address calculations; and 6) faults in the external address bus.

4) Data Cycle-Address Bus Faults: Faults inserted on the address bus, while an operand word is read or written, model: 1) faults in the external address bus; 2) multiple faults on the internal and external data bus, memory, and memory select circuitry; 3) faults in the PCU and op code decoder which cause incorrect address calculations; 4) faults in the PC and data registers if they are used in calculating the current address; and 5) faults in the address bus section of the BIC.

5) Instruction Cycle and Data Cycle-Control Bus Faults: Finally, faults inserted on the control lines model: 1) faults in the memory and memory select circuitry; 2) faults in the PCU that prevent adherence to proper bus protocol; and 3) all of the faults listed for the address bus faults.

6) Other Classifications: Many faults were inserted with a duration that spanned several bus cycles. These can be considered to be composed of a series of the above type of faults. Such faults which have an instruction cycle fault as one or more of their components will be referred to as *instruction-type faults*. All other faults will be referred to as *data-type faults*.

C. Fault Insertion Experiments

Fault insertion experiments have been performed involving the insertion of 2891 faults. Five benchmark programs constituted the software that executed on the MC68000 while the faults were inserted. These benchmarks include the string search, bit set, linked list insertion, quicksort, and bit matrix transposition programs given in [8].

The fault insertion experiments can be divided into two sets. The first set was performed with the SIS monitoring hardware in operation. A second set of experiments was performed with only the MC68000 operating. The MC68000 has several error detection mechanisms built in which cannot be disabled. The primary purpose of this second set of experiments was to determine the degree to which the MC68000 error detection mechanisms affected the fault coverage obtained in the first set of experiments. A secondary purpose was to determine and evaluate the error coverage of the MC68000 built-in error detection mechanisms. Approximately half of the faults inserted in the first and second sets of experiments were identical in terms of location and time of insertion within the benchmark programs. These identical faults are used in an attempt to decouple the effect of the MC68000 error detection mechanisms on the results obtained for SIS.

The locations within the benchmarks at which faults were inserted were chosen at random. All lines were faulted in turn at each location. Only single faults were inserted. Typically, only one duration was selected for all faults inserted at a given location. A fault's type was not predetermined, but depended upon the instruction mix of the benchmarks and the randomly selected time at which the fault was inserted into the benchmark programs.

D. SIS Error Detection Results and Analysis

1454 faults were inserted with the SIS monitoring hardware in operation. 1124 of these turned out to be instruction-type

TABLE II
FAULT COVERAGE AND DETECTION LATENCIES WITH THE SIS MONITOR

FAULT DURATION	Number Inserted	Number Detected	% Det.	Avg. Latency (in μs)	Std Dev (in μs^2)
1 cycle	517	343	66	5400	61,000
2 cycles	462	402	87	5200	60,000
4 cycles	475	442	93	1400	27,000
BUS TYPE					
Inst	1124	1097	98	38	36
Data	330	90	27	52,000	180,000
LINE FAULTED					
D0	132	110	83	74	210
D7	132	111	84	79	230
D8	132	106	80	91	480
D15	132	108	82	65	190
A1	132	112	85	110	410
A4	132	103	78	65	200
A8	132	104	79	160	1000
A12	132	115	87	39,000	160,000
LDS	132	108	82	41	190
UDS	134	105	78	50	200
DTACK	132	105	80	79	190
TOTAL/AVERAGE	1454	1187	82	3800	51,000

faults. 1097 of the 1124 instruction type faults were detected, representing a fault coverage of 98 percent. Most instruction-type faults result in control flow errors, thus SIS provides a good control flow monitoring capability as it was intended. In addition to providing a high degree of coverage for instruction-type faults, SIS detects the errors due to such faults with a very short detection latency. The average detection latency, or mean time to detect (MTTD), for instruction type faults is only 38 μs. Considering all 1454 faults, a total of 1187 faults were detected, representing a 82 percent fault coverage for all fault types. When all faults are considered the average detection latency is 3.8 ms with a large standard deviation.

Table II shows how the fault coverage and detection latency vary with duration, the type of line faulted, and the fault type. There is a heavy dependence on the fault type and duration of the fault but very little dependence on the line faulted. The average detection latency of SIS is significantly increased by the detection latencies of the data-type faults. In fact, data-type faults which affect line A12 make the primary contribution to the longer average detection latency of the data-type faults. Table III shows this influence. Data-type faults on line A12 are more likely than faults on any other lines to cause references to locations outside of the region where program data are stored. If the faults inserted on line A12 are not considered, then the average detection latency for all faults is only 81 μs.

Fig. 11 shows the distribution of the detection latency when the SIS monitoring hardware is in operation. Notice that the vast majority of the detection latencies for the instruction-type faults are less than 100 μs. The detection latencies for the data-type faults are concentrated in two groups: one in the 20–60 μs range and the other in the $\geq$ 100 μs range.

E. Decoupling MC68000 Error Detection Mechanisms from SIS

The MC68000 was designed with several error detection mechanisms built-in to the processor. Examples of such mechanisms are the address error, illegal op code, and line emulator detection mechanisms. In addition to these mechanisms, the board on which the MC68000 resides provides bus timeout circuitry to detect the absence of a memory response during a bus cycle. The aggregate of these mechanisms is referred to as the MC68000 error detection mechanisms.

TABLE III
DETECTION LATENCIES OF DATA-TYPE FAULTS

Line Faulted	Number Inserted	Number Detected	Avg. Latency (in μs)
D0	30	9	372
D7	30	10	499
D8	30	6	1219
D15	30	8	417
A1	30	11	828
A4	30	6	444
A8	30	7	1894
A12	30	15	294,000
LDS	30	8	550
UDS	30	6	452
DTACK	30	4	602

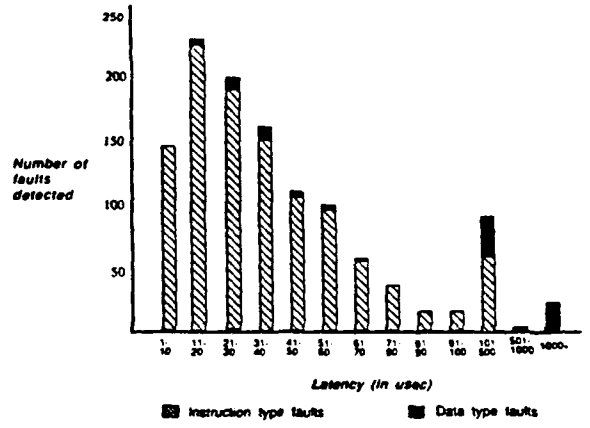


Fig. 11. Distribution of SIS monitor detection latencies.

790 faults were inserted at identical locations and times in the first and second sets of experiments. Fig. 12 shows a breakdown of the detection mechanisms for these faults in both the first and second sets of experiments. The first column states the means of detection for faults inserted in the first set of experiments. The second column states whether or not errors caused by the fault were detected by the MC68000 in the second set of experiments. The category where the errors caused by inserted faults are detected by the MC68000 in both experiments is broken into three subcategories: those which took greater, lesser, and the same amount of time to detect in the first set than in the second set of experiments.

638 faults were detected in at least one of the two sets of experiments. Of these, 281 were detected by SIS in the first set of experiments. The MC68000 error detection mechanisms do not affect the detection of a fault by SIS. Therefore, these faults would have been detected by SIS if the MC68000 error detection mechanisms had been disabled. The detection latency of 254 faults was changed by SIS. In this case, SIS at least had some influence on the detection of these faults. Although they may not have been eventually detected by SIS, there is some indication that they were not transparent to SIS. The most likely cause of the change in detection latency is the branch address hashing (BAH), used in SIS. The remaining

EXPERIMENT 1 (W/SIS) DETECTED BY	EXPERIMENT 2 (W/O SIS) DETECTED BY	NUMBER DETECTED	EXPERIMENT 1 LATENCY	EXPERIMENT 2 LATENCY
MC68000	MC68000 (same latency)	103	19μs	19μs
MC68000	MC68000 (greater latency)	117	5 0ms	7 8ms
MC68000	MC68000 (lesser latency)	92	210μs	170μs
SIS	MC68000	122	9 5ms	23 0ms
NONE	MC68000	7	-	51 0ms
MC68000	NONE	38	20μs	-
SIS	NONE	159	63μs	-
NONE	NONE	152	-	-

Fig. 12. Detection breakdown for experiments 1 and 2.

TABLE IV
FAULT COVERAGE AND LATENCIES FOR THE MC68000

FAULT DURATION	Number Inserted	Number Detected	% Det.	Avg. Latency (in μs)	Std Dev (in μs^2)
1 cycle	493	202	41	13,000	85,000
2 cycles	484	298	61	12,000	70,000
4 cycles	460	313	68	14,000	91,000
BUS TYPE					
Inst	1053	724	69	4100	42,000
Data	384	89	23	82,000	200,000
LINE FAULTED					
D0	131	79	60	4300	23,000
D7	131	78	60	1200	9000
D8	131	63	48	1600	10,000
D15	130	81	62	210	550
A1	131	72	55	450	950
A4	131	67	51	24,000	110,000
A8	129	76	59	1600	10,000
A12	131	94	72	85,000	210,000
LDS	131	97	73	98	520
UDS	130	92	71	51	200
DTACK	131	14	11	43,000	100,000
TOTAL/AVERAGE	1437	831	58	13,000	82,000

103 faults were unchanged in their detection latencies in both experiments. The fact that they were transparent to SIS is likely to be due to their extremely short detection latency, 19 μs, by the MC68000 error detection mechanisms. Further analysis and collection of data needs to be done in order to accurately assess the influence of the MC68000 error detection mechanisms on the fault coverage obtained in the first set of experiments.

F. MC68000 Error Detection Results and Analysis

1437 faults were inserted with the MC68000 operating without the SIS monitoring hardware. Of the 1437 faults, 831 were detected by the MC68000 detection mechanisms, representing a 57 percent fault coverage. The average detection latency was 13 ms with a large standard deviation. This average detection latency is over four times the detection latency when SIS is operating. Table IV shows how the detection percentage and latency vary with duration, the type of line faulted, and the fault type. It can be concluded that unlike most commercially available microprocessors, the MC68000 has reasonably effective fault detection mechanisms built in. However, comparing the distributions with and without SIS operating shows that SIS, detects a much higher percentage of faults and significantly reduces the average detection latency of most fault types.

This data shows that the fault coverage varies directly with the duration of the fault. This is to be expected since faults of longer duration cause more errors. The fault coverage also depends heavily upon the bus cycle type of the fault. The MC68000 detection mechanisms are better suited to detecting instruction-type faults than data-type faults. Not only are more instruction-type faults detected, but they are detected much faster.

Faults on the DTACK line are difficult to detect. A possible explanation for the difficulty in detecting the DTACK faults is that they merely cause the MC68000 to read from the memory a clock cycle earlier. As long as the memory has valid data at its outputs at this time, no error will occur. Errors will still be caused during write cycles since the write cycle time requirement will not be observed.

G. Analysis of Effects on System Reliability

The effect of the SIS monitoring hardware on the reliability of the system can be determined analytically. The analysis presented here uses a simple error model. The use of more complex and accurate models can be used once further studies have been performed. The assumptions used in this analysis are listed below.

1) The system consists only of an MC68000, its associated program memory, and the SIS monitoring hardware.

2) The SIS logic gate and memory overhead constitute approximately 17 percent of the logic gate and memory count of the MC68000 and its program memory.

3) The MC68000, the program memory, and SIS monitoring hardware are assumed to have constant failure rates.

4) The failure rate of a hardware module is a direct function of its gate count.

5) The failure rates of the MC68000 and SIS monitoring hardware are independent.

6) All detected errors are assumed to be recovered from.

Let $R(t)$ represent the reliability of the MC68000 with no detection mechanisms, that is $R(t)$ is the probability that an error will not occur before time t. Let E represent the error coverage of the system with SIS. F represents the relative size of the SIS monitoring hardware with respect to the size of the MC68000. The reliability of the SIS monitoring hardware is then $R^F(t)$.

In addition to the overhead for the SIS monitoring hardware, there is also overhead associated with the recovery hardware. A determination of the recovery hardware overhead has been made for incorporating SIS in a processor that is being designed at IBM-Federal Systems Division [1]. Based on this investigation, we estimate that a similar recovery scheme for the MC68000 will require about 8 percent hardware overhead. This overhead must be added to the overhead of the SIS monitoring hardware in determining the system reliability. The reliability of the entire system is then

$$\text{system reliability} = R(t) * R^F(t) + (1 - R(t)) * R^F(t) * E.$$

Fig. 13 shows a plot of the reliability of the MC68000 with and without the SIS monitor incorporated, for $E = 0.82$ and F

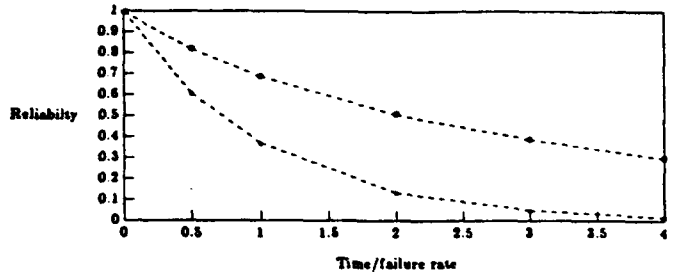


Fig. 13. System reliability.

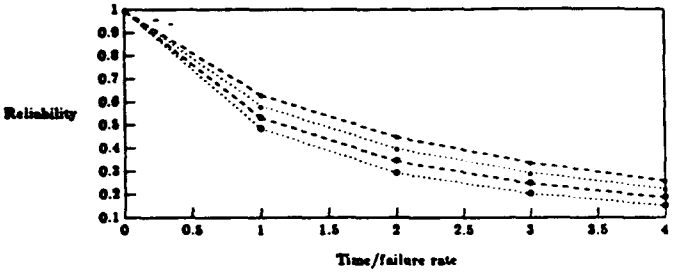


Fig. 14. System reliability with varying error coverages.

$= 0.17 + 0.08 = 0.25$. It can be seen that SIS provides a significant enhancement in the overall system reliability. Less than 100 percent error recovery coverage can be reflected in reductions of the errot coverage. Fig. 14 illustrates the effect of reductions in the fraction of recoverable errors by reducing E.

V. Extensions to SIS

Several extensions to the SIS approach have been pursued or are being pursued at CMU. This section briefly describes some of these follow-on efforts.

A. On-Chip Implementation

Enhancements to the error coverage and reduction of the overhead can be made if the SIS monitoring hardware is integrated into the design of the processor chip. This would make all internal signals of the processor visible to the monitoring hardware and allow much of the monitoring hardware in the present design to be eliminated.

First, more internal signals can be monitored to increase error coverage. Activities that could be monitored include: fetching and sequencing of all microcode instructions, internal control signal sequencing, and setting of condition codes. Operation of the ALU could be verified by passing known data through it during periods in which it is idle. The results of the ALU operation could then be incorporated into the signature. The output of a parity checker for the data registers could also be signatured providing a means for checking the operation of the processor registers by the monitoring hardware.

When implementing the monitoring hardware on the processor chip, much of the hardware in the present design can be eliminated. There would be no need for a separate op code decoder. There would be no need for a separate signature generator prefetch queue. The signature stack can be placed in main memory and a stack pointer added to the processor registers. The program counter emulator would not be necessary since the actual program counter can be used. An on-chip implementation of SIS would require approximately 2000 nMOS gates. This is equivalent to an overhead of approximately 9 percent of the silicon real estate. Because of the modularity of the SIS hardware, the impact of this added overhead on the routing area should be commensurate with the gate overhead.

Fig. 15 shows the reliability of a self-monitoring processor with on-chip implementation of the SIS monitoring hardware, assuming independence of errors in the SIS monitoring hardware and the MC68000 and that $E = 0.82$ and $F = 0.09 + 0.08 = 0.17$. It is also conservatively assumed that there is no increase in the error coverage. Since some of the circuitry is shared between the SIS monitor and the MC68000 in the on-chip implementation, all errors will not be independent. This will cause the reliability of the system to be somewhat less than that shown. This effect will be counteracted by an expected increase in the actual error coverage due to the on-chip implementation.

B. Roving Monitoring

A second and related approach for doing control flow monitoring has also been developed, called asynchronous signatured instruction streams (ASIS) [7]. With ASIS a small amount of hardware, called a hardware signature generator, is dedicated to each application processor. A second hardware unit, called the roving monitoring processor (RMP), is time shared amongst several application processors to check their control flow. Relative priorities can be assigned to each of the monitored processors. The order in which signatures from the various hardware signature generators are to be processed by the RMP can be made to reflect the relative priorities. A roving monitoring demonstration system for MC68000 processors has been constructed and is working.

C. Architecture of a Roving Monitoring Processor

Given the potential performance and cost advantages of a high-speed roving monitoring processor, an innovative data flow-based architecture for the RMP has been developed [20]. This architecture allows a streamlined instruction set to be defined which executes quickly and reduces context switching overhead. It also incorporates a mechanism for efficiently managing the execution of several instruction streams and providing high performance through pipelining. This RMP has also been constructed and is capable of simultaneously monitoring up to sixteen 8 Mhz MC68000 processors.

D. Recovery

Issues pertaining to error recovery are being investigated [1]. Work has been done by Lee, Ghani, and Heron [10] on recovery caches which with only slight modification appears to be applicable here. Basically, a recovery cache is used to store the current values of variables which have been altered since the last signature check. The contents of the cache are written back to the main memory only when a signature check indicates that no error has occurred. This is unlike the scheme presented in [10] where the cache is used as the backup storage. By using the cache to hold the current values it should be possible to obtain the performance advantages of a cache

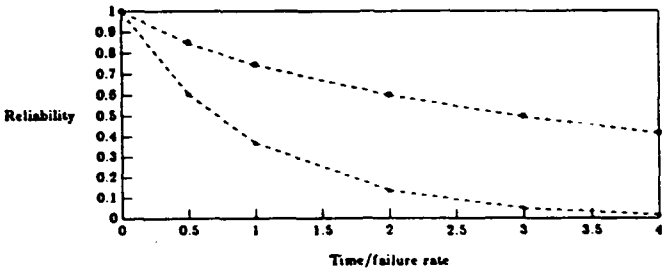


Fig. 15. MC68000 reliability with on-chip SIS.

along with the recovery capability. In addition to the cache, a set of shadow registers would be needed to provide backup storage for the processor's registers. These could be located on the processor chip. Finally, interfaces with other devices would have to be altered to allow for recovery attempts. More work in the recovery area is needed.

VI. Summary

The concepts and implementation details of the signatured instruction streams (SIS) on-line monitoring approach have been presented in this paper. A demonstration system based on the MC68000 processor has been built to demonstrate the practical feasibility of the SIS approach. This approach, if implemented on the processor chip, requires approximately 10 percent in hardware overhead. If hardware for error recovery is included, the total overhead is expected to be around 20 percent. The performance penalty depends upon the application program and is expected to be less than 10 percent.

Fault insertion experiments have been performed on the demonstration system. The results indicate that SIS provides a reasonably good error coverage of 82 percent of all fault types. For the more critical instruction-type faults it provides an error coverage of 98 percent. The error detection latency for all faults is relatively short and the average is 3.8 ms. For instruction-type faults the average error detection latency is only 38 μs. In other words, with SIS, almost all control flow errors caused by instruction-type faults can be detected within a small number of instructions from where the error occurred. Analytical results have been derived for the reliability of a system incorporating SIS. These results clearly show the benefit of the SIS approach in increasing the overall reliability of a SIS self-monitoring system.

References

[1] B. Aglietti, M. A. Schuette, and J. P. Shen, "Concurrent error detection and recovery using signatured instruction streams," Dep. Elec. Comput. Eng., Carnegie-Mellon Univ., May, 1985, Pittsburgh, PA, Tech. Rep.

[2] A. Avizienis, "Architecture of fault-tolerant computing systems," in *Proc. 5th Int. Fault-Tolerant Comput. Symp.*, 1975, pp. 3-16.

[3] J. M. Ayache, P. Azema, and M. Diaz, "Observer: A concept for on-line detection of control errors in concurrent systems," in *Proc. 9th Int. Fault-Tolerant Comput. Symp.*, June 1979, pp. 79-86.

[4] S. Bologna and W. Ehrenberger, "Possibilities and boundaries for the use of control sequence checking," in *Proc. 8th Int. Fault-Tolerant Comput. Symp.*, June 1978, p. 226.

[5] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Rockville, MD: Computer Science, 1976.

[6] L. Chen and A. Avizienis, "*N*-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th Int. Fault-Tolerant Comput. Symp.*, 1978, pp. 3-9.

[7] J. B. Eifert and J. P. Shen, "Processor monitoring using asynchronous signatured instruction streams," in *Proc. 14th Int. Fault-Tolerant Comput. Symp.*, June 1984, pp. 394-399.

[8] R. D. Grappel and J. E. Hemenway, "A tale of four microprocessors: Benchmarks quantify performance," *EDN*, Apr. 1981.

[9] V. S. Iyengar and L. L. Kinney, "Current fault detection in microprogrammed control units," *IEEE Trans. Comput.*, vol. C-34, pp. 810-821, Sept. 1985.

[10] P. A. Lee, N. Ghani, and K. Heron, "A recovery cache for the PDP-11," in *Proc. 9th Int. Test Conf.*, Oct. 1979, pp. 3-8.

[11] D. J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, pp. 681-685, July 1982.

[12] A. Mahmood, E. J. McCluskey, and D. J. Lu, "Concurrent fault detection using a watchdog processor and assertions," in *Proc. 13th Int. Test Conf.*, Oct. 1983, pp. 622-628.

[13] A. Mahmood and E. J. McCluskey, "Watchdog processors: Error coverage and overhead," in *Proc. 15th Fault-Tolerant Comput. Symp.*, June 1985, pp. 214-219.

[14] M. Namjoo, "Techniques for concurrent testing of VLSI processor operation," in *Proc. 12th Int. Fault-Tolerant Comput. Symp.*, June 1982, pp. 461-468.

[15] M. Namjoo and E. J. McCluskey, "Watchdog processors and capability checking," in *Proc. 12th Int. Fault-Tolerant Comput. Symp.*, June 1982, pp. 235-248.

[16] M. E. Schmid, R. L. Trapp, A. E. Davidoff, and G. M. Masson, "Upset exposure by means of abstraction verification," in *Proc. 12th Int., Fault-Tolerant Comput. Symp.*, June 1982, pp. 237-244.

[17] J. P. Shen and M. A. Schuette, "On-line monitoring using signatured instruction streams," in *Proc. 13th Int. Test Conf.*, Oct. 1983, pp. 275-282.

[18] D. P. Siewiorek and L. K. Lai, "Testing of digital systems," *Proc. IEEE*, vol. 69, pp. 1321-1331, Oct. 1981.

[19] T. Sridhar and S. M. Thatte, "Concurrent checking of program flow in VLSI processors," in *Proc. 12th Int. Test Conf.*, Nov. 1982, pp. 191-199.

[20] S. P. Tomas and J. P. Shen, "A roving monitoring processor for detection of control flow errors in multiple processor systems," in *Proc. ICCD*, Oct. 1985.

[21] J. Wakerly, *Error Decting Codes, Self-Checking Circuits and Applications*. Amsterdam, The Netherlands: North-Holland, 1978.

[22] S. S. Yau and F. C. Chen, "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126-137, Mar. 1980.

Michael A. Schuette (S'80-M'86) received the B.S. degree in electrical engineering from Michigan State University, East Lansing, in 1982, and the M.S. degree in electrical and computer engineering from Carnegie-Mellon University, Pittsburgh, PA, in 1984.

During the Summer of 1981 he was employed at Bell Laboratories, Naperville, IL, as a digital designer. In the Summer of 1982 he was employed at McDonnel Douglas Research Labs, St. Louis, MO, as a Research Assistant. In the Summer of 1984 he was employed at General Electric Microelectronics Center, Research Triangle Park, NC. At that time he was responsible for implementing the prototype of MAST, an automated design for testability tool. Currently, he is a Ph.D. candidate at Carnegie-Mellon University, Pittsburgh, PA, doing work in system level fault detection and tolerance.

Mr. Schuette is a member of Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.

John Paul Shen (M'81) received the B.S. degree from the University of Michigan, Ann Arbor, in 1973, and the M.S. and Ph.D. degrees from the University of Southern California, Los Angeles, in 1975 and 1981, respectively, all in electrical engineering.

From 1973 to 1975 he was with the Hughes Aircraft Company where he participated in the design of fault detection/isolation and built-in test circuits for avionic systems. In 1977 he was with the Systems Group of TRW, Redondo Beach, CA, where he was involved in the study and preliminary design of a local computer network. From 1977 to 1981 he performed research on multicomputer interconnection networks in the Department of Electrical Engineering-Systems, University of Southern California. Currently, he is an Assistant Professor in the Electrical and Computer Engineering Department, Carnegie-Mellon University, Pittsburgh, PA. He has consulted for the IBM Federal Systems Division and the General Electric Microelectronics Center. His research interests include computer-aided design and test of VLSI circuits, parallel architectures, and fault tolerance of real-time and mission-oriented systems.

Dr. Shen is a member of the Association for Computing Machinery, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi. He is a recipient of a National Science Foundation Presidential Young Investigator Award.

EMBEDDED SIGNATURE MONITORING: ANALYSIS AND TECHNIQUE

Kent D. Wilken and John Paul Shen

Department of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213 U.S.A.

Abstract -- *A new method is presented for analyzing the effectiveness of Embedded Signature Monitoring (ESM) techniques at concurrently detecting processor control flow errors. Application of this method to previously reported ESM techniques shows that error detection coverage is limited by program characteristics and does not improve for signatures larger than 8 bits. This analysis helps to explain the less than expected coverage results from experiments performed on an ESM technique that used a 16-bit signature. A new ESM technique is introduced that achieves coverage of $1\text{-}2^{-w}$ for a w-bit signature. Use of this new technique with signatures of typical size reduces undetected errors by orders of magnitude compared to previous techniques. A method is introduced for assessing the memory overhead required by an ESM technique. The new ESM technique is shown to require the least overhead based on a sample set of program statistics. Previous work has reported that embedded signatures can decrease the performance of the monitored processor by as much as 10%. Analysis presented here suggests that the new ESM technique can substantially reduce performance losses.*

1. Introduction

As integrated circuits increase in complexity the problem of insuring correct operation at their point of manufacture or during their operating life becomes more difficult. At the same time, devices are shrinking and becoming more susceptible to transient errors while in use [12]. These trends make the role of concurrent testing increasingly important and challenging.

Concurrent Processor Testing

Concurrent detection of processor errors has traditionally relied on the addition of redundancy based on the processor's structure. Replication of the entire processor with a check on the pairwise result is a common approach. Processors have been scrutinized to determine which sub-structures can make use of more economical coding techniques to detect errors. Any remaining sub-structures are error checked through duplication. While effective, these approaches may not be sufficiently economical for many applications.

The search for improved approaches to processor error detection has led researchers to propose monitoring abstractions of processor behavior [8]. Besides the prospect of new economies, this approach can be technology and implementation independent. Several specific techniques have been presented that help to insure correct processor operation by encoding a program with signatures that are verified during execution [5], [6], [9], [13], [14]. This approach to concurrent processor testing will be referred to as *signature monitoring*.

Signature monitoring uses a compiler to generate and store signatures based on the content and structure of a monitored program. A simple hardware monitor regenerates the signatures at run-time and compares them with the pre-computed version. An error is declared when a difference occurs. Errors that can be detected by signature monitoring have been divided into two categories [13]. *Bit errors* occur when the program is executed in correct order but the value of one or more program memory bits has been altered. Failures that result in incorrect program flow are classified as *sequence errors*.

Previous work has analyzed signature monitoring error coverage [4]. The model that was proposed for sequence error coverage produces results that are not consistent with recent experimental data [9]. This paper proposes a new method for analyzing sequence error coverage. This method is applied to previously proposed signature monitoring techniques. A correspondence is shown between the results obtained with this method and experimental data. This method and the accompanying analysis lead to the proposal of a new signature monitoring technique.

Signature monitoring has been applied to both assembly level programs and microprograms [13], [14]. Wide microprogram words make techniques that use one signature field per word feasible at that level. For assembly level programs, one signature must cover a number of words in order to achieve low overhead. Techniques that are effective at the assembly level are also applicable to the microprogram level. Without loss of generality this paper will focus on assembly level signature monitoring.

The analytical methods developed here can be applied to specific programs to generate quantitative results based on statistical data from the programs. Example statistics are used in this paper to generate numerical results in order to give some indication of the relative and absolute effectiveness of various signature monitoring techniques. While indicative, these numerical results are not definitive. Specific results will show application dependencies.

Embedded Signature Monitoring

Assembly level signature monitoring techniques have been proposed that store signatures within the processor's program space [5], [9], [13] [14] or in a separate memory structure [6], [11]. The former approach, which will be termed *Embedded Signature Monitoring* (ESM), has the potential for lower memory overhead because it does not require storage for a duplicate set of program control flow information. The latter approach, which will be termed *Disjoint Signature Monitoring* (DSM), has an inherent performance advantage because the pre-computed signatures do not consume processor memory bandwidth. This paper will restrict its analysis to ESM techniques although some of the analysis is relevant to DSM techniques.

Figure 1 shows a section from a program that has been signatured using a basic ESM technique. A program is analyzed by the compiler and divided into *blocks* [13]. A block is a program segment that starts at the destination of a branch instruction. In the context of this paper the term branch refers to all control flow altering instructions, including subroutine calls and returns. A block ends at the first occurrence of a branch instruction or at the location preceding the next branch destination. A block is also referred to as a branch-free interval or straight-line code.

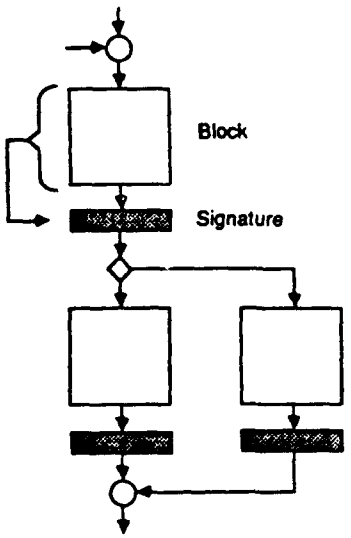


Figure 1: Basic Embedded Signaturing

An encoding function is selected and at compile time is used to compute a signature based on the bit values of the block's instructions. A Cyclic Redundancy Code (CRC) [7] is a typical encoding function. The pre-computed signature is embedded within the block, usually at the beginning or the end. Information delimiting block boundaries must be generated by the compiler and is also embedded within the block.

Dedicated hardware is used to re-compute the signatures at run-time. A Parallel-input Linear Feedback Shift Register (PLFSR) is typically used for this purpose. The PLFSR is initialized at the beginning of a block. The calculation proceeds, operating on the fetched instructions until an embedded delimiter indicates that the end of a block has been reached. The resulting signature is compared with the embedded version and if unequal, an error is declared. At run-time the embedded signatures and delimiters must be explicitly or implicitly ignored by the processor's execution unit.

Several variations from the basic ESM technique have been proposed. Different encoding functions can be selected based on the expected error mechanisms. Carter [2] discusses functions that can detect all double or triple bit errors within a specified block. Other functions from coding theory can be used for ESM to guarantee the detection of distinct bit error patterns over specified block lengths, e.g. a single burst, multiple phased bursts, multiple single bit errors, etc. [7]. Mahmood and McCluskey [4] propose generating signatures based on the column order of bits within a block rather than row order. Codes can then be applied that guarantee the detection of similar column bit error patterns.

The program segment that is covered by an embedded signature will be termed an *interval*. In the basic ESM technique an interval is the same as a block. All of the proposed ESM techniques allow for an interval to span more than one block. This reduces the number of embedded signatures which in turn decreases memory and performance overhead. Sridhar and Thatte [13] only place signatures in blocks that precede a location where the program flow merges. Namjoo [5] proposes signaturing paths, that may consist of multiple blocks, based on an analysis of the program flow graph. Schuette and Shen [9] introduce a method that eliminates the storage requirement for signatures which can be combined with a branch address.

Various methods have been proposed for embedding the information that delimits an interval. Sridhar and Thatte [13] and Schuette and Shen [9] make use of a unique opcode to indicate the end of an interval. The interval's beginning is implied. A single dedicated bit column is used by Sridhar and Thatte [13] and Tung and Robinson [14] for a similar purpose. Namjoo [5] proposes the use of two bit columns which allows both the interval's beginning and end to be explicitly indicated.

2. Coverage Analysis

Mahmood and McCluskey [4] proposed that sequence errors can be modeled as memory errors and concluded that they go undetected at a rate of 2^{-w} for a w-bit signature. Experimental results reported by Schuette and Shen [9] show that the fraction of undetected errors is much higher than predicted by this model. This section develops a new method for analyzing sequence error coverage that is shown to be more consistent with the experimental observations.

Undetected Sequence Error Estimation

Signature generation consists of a series of intermediate calculations based on the series of words within an interval. The result of each intermediate calculation corresponds to the signature of a sub-interval. For an interval [0,j] consisting of words W_0, ..., W_j, a location k in the range [1,j] has an implicitly associated *intermediate signature*, I_k, that is based on the encoding of the words in the sub-interval [0,k-1]. The value of I_k is equal to the kth intermediate calculation:

$$I_k = f(I_{k-1}, W_{k-1})$$

where W_{k-1} is the value of the word at location k-1, f is the signaturing function and I_0 is a specified initial value, e.g. 0. The last intermediate signature, I_j, is the interval's signature. Figure 2 shows an interval and the intermediate signatures that are associated with each location.

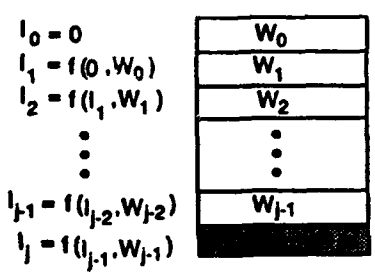


Figure 2: Intermediate Signature Generation

Intermediate signatures can be used to estimate the fraction of undetected sequence errors. A correctly operating processor will always transition from the current location S to the correct succeeding location D. After completing an intermediate calculation or an initialization at S, the PLFSR will contain the intermediate signature I_D. A sequence error will cause the program to transition to a different location D*. If $I_{D^*} = I_D$, the signature calculation that continues from D* will yield a correct result at the end of that interval and the error will go undetected. If $I_{D^*} \neq I_D$, the PLFSR will contain an incorrect value and the error will persist until it is detected at the end of the interval.

The intermediate signatures for all locations in a particular program can be determined at compile time. All locations with intermediate signature value V are placed into a group D_V. All locations that can transition to a member in D_V are placed in a group S_V. It is assumed that a sequence error can emanate from any source location with equal probability and errantly transition to any destination location with equal probability. Let m denote the total number of memory locations of a fully occupied program space, d_v denote the number of locations in D_V and s_v the number in S_V. The fraction of undetected sequence errors originating from a specific member of S_V is $(d_v-1)/(m-1) \approx (d_v-1)/m$ for typical values of m. The probability that the sequence error came from a member of S_V is s_v/m. The estimated fraction of uncovered sequence errors associated with the intermediate signature value V is e_v:

$$e_v = (s_v)(d_v-1)/m^2$$

Summing the result of this expression over all possible intermediate signature values V will yield an estimate for the total fraction of undetected sequence errors e:

$$e = \sum_v s_v(d_v-1)/m^2 \qquad (1)$$

Correlated Intermediate Signatures

Using (1) a random distribution of intermediate signatures over 2^w memory locations will yield a sequence error coverage of $1-2^{-w}$ for a w-bit signature. Any correlation among the intermediate signatures will increase the size of certain intermediate signature groups and hence decrease the coverage. The ESM techniques proposed by Sridhar and Thatte [13], Namjoo [5] and Schuette and Shen [9] use the same initial intermediate signature value, I_0, for each signature interval. This implies that all locations that begin an interval reside in the same group D_{I_0}. The intermediate signature for the second interval location, I_1, is a function of I_0 and the value of the first word, W_0. Any correlation among the W_0 values of the intervals will cause the I_1 values to be correlated. Similarly, if correlated sets of consecutive word values are found to begin intervals the intermediate signatures that are generated will be correspondingly correlated. Kobayashi [3] reports that a strong correlation exists among entire blocks of instructions.

A precise estimate for undetected errors caused by intermediate signature correlation would require considering the contribution of all intermediate signature values V. However, a lower bound can be established by considering only the effect of I_0. Several studies reported block sizes in the range of 4-10 words [3], [4], [5], [9]. Adding a signature word to each block increases the size to 5-11 words. From the method

developed above, this implies that $(d_{I_0}-1)/m$ is 1/5 to 1/11. Since the last word of each block is a source for a transition to the beginning of a block, s_{I_0}/m is also 1/5 to 1/11. Therefore a low estimate for the fraction of undetected sequence errors, $(s_{I_0})(d_{I_0}-1)/m^2$, is 1% to 4%.

This result applies to the basic ESM technique that has a signature for every block. The various proposed techniques reduce the total number of signatures, which in turn reduces the undetected errors due to I_0. Schuette and Shen [9] introduced a technique referred to as Branch Address Hashing (BAH) that reduces the number of signatures by about 50% [10]. As shown in Figure 3 a signature is used to cover the interval that resides between two program *merge nodes*. A merge node is a program location that can be reached by both sequential execution and a branch. A branch operation within this interval would normally constitute the end of a block and require a signature. This need is eliminated by replacing the branch address with a hashed address that is the bit-wise XOR of the branch address and the location's intermediate signature. Under normal operation, when the branch is taken the correct intermediate signature is generated and is XORed with the hashed address to extract the actual branch address. If an error occurs that leads to an incorrect intermediate signature, the decoded branch address becomes an arbitrary value and an errant jump is taken to a location where the error may be detected. Such an event is termed an *induced sequence error* and is analyzed in the next subsection.

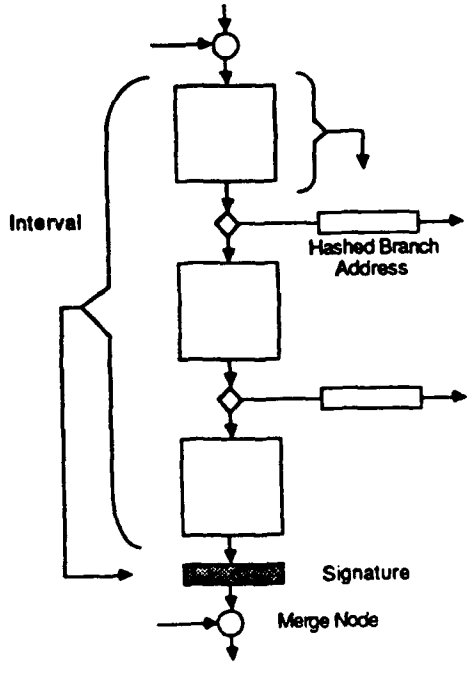


Figure 3: Branch Address Hashing

The reported 50% signature reduction [10] due to BAH might suggest that both $(d_{I_0}-1)/m$ and s_{I_0}/m will be halved. This would result in a four-fold reduction in the undetected sequence error rate that was estimated above. However hashed signatures associated with subroutine calls and unconditional branches create both the end and start of a signature interval in the same manner as an explicit signature. A location where a signature is hashed onto a conditional branch does not constitute the beginning or end of a signature interval. However when that branch is taken the PLFSR is reset, making it an I_0 source location. The net effect is that the lower signature count resulting from the use of BAH reduces the number of undetected errors due to I_0 by roughly 1.5 times. This does not appreciably change the range of undetected error rates estimated above. Any reduction in undetected errors due to a lower signature count appears to be even smaller for the other techniques.

If intermediate signatures are randomly distributed, an 8-bit signature would exhibit a 2^{-8} (0.4%) undetected sequence error rate. Coupled with the above analysis this suggests that coverage for these techniques will be dominated by the effects of intermediate signature correlation. Signatures larger than 8 bits will yield no coverage improvement. This in part explains why the experimental data from Signatured Instruction Streams (SIS) [9] indicated 2% undetected "instruction errors" [9] rather than 2^{-16} (0.0015%) as would be expected from Mahmood and McCluskey's model [4] for the 16-bit signature used.

Induced Sequence Errors

The induced sequence error described above is another major cause of undetected sequence errors. Figure 4 is a Markov model showing how an error is handled by the SIS technique. The original sequence error in state A will go undetected into state W due to correlated intermediate signatures at a rate e determined in (1). Otherwise the error persists until the end of the current block, state B. Only a fraction b of the blocks reached by the sequence error end with an explicit signature, state C. Here the error is either detected with probability $1-2^{-w}$ in state Y or goes undetected with probability 2^{-w} in state X. The remaining $1-b$ blocks end with a hashed branch, state D. Of the hashed branches a fraction c are conditional branches, state E. Because a random block is reached by the sequence error, the branch condition is not strongly correlated with the processor state. The branch condition is therefore assumed to be satisfied at a random rate equal to one half. This implies that $c/2$ of the total hashed branches are not taken. For these occurrences execution increments into the next block, state B, where resolution of the error continues.

The remaining $1-(c/2)$ fraction of the branches are taken and result in an induced sequence error, state F. When an induced sequence error occurs, the PLFSR

contains the reset value, I_0. The error will go undetected if the errant branch arrives at the beginning of an interval. For an average interval length L the probability of this occurrence is 1/L, in which case state Z is reached. $1/L \approx (d_{I_0} - 1)/m$.

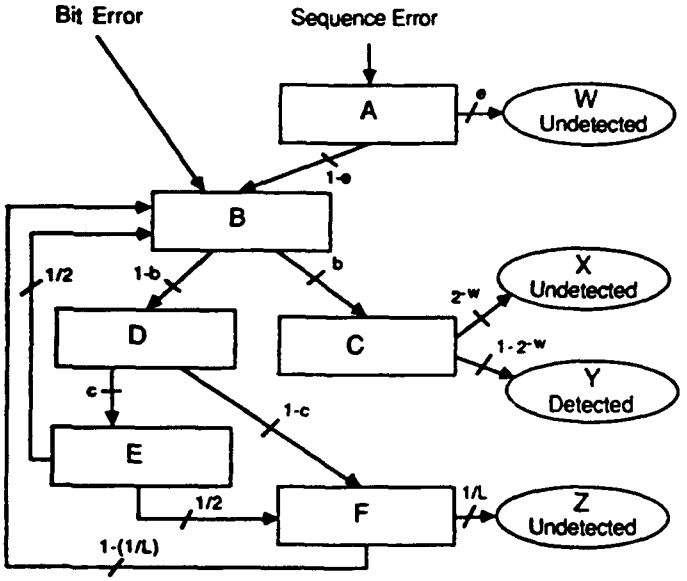


Figure 4: Markov Model of SIS Error Handling

If the induced sequence error does not arrive at the beginning of an interval, execution proceeds at a new arbitrary block and the process continues in state B. The error persists until one of the absorption states X,Y or Z is reached.

Along with sequence errors, bit errors go undetected by the SIS technique due to induced sequence errors. A bit error starts in state B and from there transitions through the Markov model in a manner similar to that of a sequence error.

Data can be applied to the above Markov model in order to estimate the total fraction of sequence errors not detected by the SIS technique. A low estimate for *e*, the undetected sequence errors due to correlated intermediate signatures, has been determined to be 1% to 4%. Shen and Schuette [10] estimate *b*, the fraction of blocks with signatures, to be 1/2. Alexander and Wortman [1] report *c*, the fraction of conditional branches, to be roughly 1/3. L, the average interval length, can be determined as follows. For the SIS technique it is estimated that 4 out of 5 blocks begin a signature interval. The remaining 1/5 are contained within an interval that is started by another block. As established earlier, typical block sizes range from 4 to 10 words. The SIS technique adds a two word opcode/signature to half of the blocks. The other blocks contain hashed signatures. This suggests that the interval size L ranges from 6 to 14 words.

A simulation of the Markov model shown in Figure 4 using these parameters estimates the total fraction of sequence errors that are undetected by the SIS technique to be in the range of 6% to 16%. Of this total 5% to 12% are due to the induced sequence error mechanism. As determined earlier, 1% to 4% are due to correlated intermediate signatures.

This result is much higher than the 2% undetected instruction errors measured experimentally for SIS. This discrepancy is explained as follows. In addition to the ESM mechanism, the SIS experimental system contains program-bounds checking hardware. Because the programs used in the experiment were small (1-4K) compared to the 64K PC relative address space, there is a high probability that an induced sequence error will cause a branch out of the preset bounds and be detected by this auxiliary mechanism. Similar tests done with larger program sizes are expected to result in lower coverage. In addition, as reported in [9], the processor's built-in error detection mechanisms could not be disabled and contributed to error coverage beyond that possible by the ESM mechanism alone. While the use of BAH makes a significant reduction in memory overhead, this analysis suggests that error coverage is sacrificed.

Uncorrelating Intermediate Signatures

Tung and Robinson [14] propose an ESM technique that improves on the coverage achievable with SIS. A value is embedded at the destination of any branch operation. Each embedded value constitutes the beginning of an interval. The embedded value is equal to the bit-wise XOR of the interval's signature and the value's memory address. During execution the monitor independently generates the branch destination address. The monitor extracts the destination interval's signature by XORing the destination address generated by the monitor with the embedded value. A branch that transitions to the beginning of the wrong interval can be detected by this technique but is not detected by SIS. Encoding signatures in this manner effectively uncorrelates intermediate signatures thereby eliminating the associated undetected sequence errors.

To minimize memory overhead Tung and Robinson employ BAH. This causes the proposed technique to be vulnerable to induced sequence errors as analyzed above. Because induced sequence errors appear to be the dominant component of undetected sequence errors this technique may achieve only a modest coverage improvement. Furthermore, as shown in Section 4, even with BAH this technique necessitates an increase in memory overhead compared with SIS. This is caused by a higher signature count. This technique requires an embedded value at each branch destination. SIS requires a signature for each merge node. All merge nodes are branch destinations but there are branch destinations that are not merge nodes.

The hardware monitor needed for this technique is necessarily more complex than any other that has been proposed. It must recognize the different addressing

modes that exist at the assembly program level, e.g. absolute, Program Counter (PC) relative and register relative. Each effective branch destination address must be computed in real-time and stored in a special register. This requires more extensive opcode decoding and a new monitor register. Program counter emulation and an adder are needed to independently generate PC relative addresses. The contents of registers must be emulated in order to accommodate register-relative addressing modes. Additional control circuitry is necessary to orchestrate these extra resources.

Other Failure Modes

A key premise of signature monitoring is that it insures correct program sequencing. Only single sequence errors have been considered thus far. This could be generalized to include multiple sequence errors. However, multiple sequence errors may not be independent, an assumption necessary to extend the single error analysis. A notorious example of dependent sequence errors is the stuck PC. This has been identified as an error that must be cover in safety critical systems [15]. None of the proposed ESM techniques detect this error. It is possible for an ESM technique to be augmented with a mechanism that specifically addresses the stuck PC. However, depending on the mechanism, this still may not be sufficient to insure correct program sequencing. The stuck PC can be viewed as an infinite loop containing one address that effectively circumvents the ESM technique. This failure mode can be generalized to include infinite loops containing more than one address. These failure modes should also be detected by any such auxiliary mechanism.

A second premise is that ESM is effective at detecting errors in program memory. Memory array errors generally occur as single or multiple bit, row or column errors. Mahmood and McCluskey [4] discuss the effectiveness of ESM at detecting single bit and row errors. Carter [2] proposes a technique that covers multiple bit errors. [4] notes the importance of detecting memory column failures and suggests that codes can be selected which guarantee coverage of certain column failures.

Notwithstanding the proposal by Mahmood and McCluskey [4], several ESM techniques remain vulnerable to column failures. Tung and Robinson [14] suggest the use of a single memory column to distinguish between a signature and an ordinary instruction. If this column is stuck-at the value that indicates "ordinary instruction", the monitor will never receive a signature check indication and the error will go undetected. Sridhar and Thatte [13] and Schuette and Shen [9] make use of a unique opcode followed by a signature. If any of the columns are stuck-at a value that is the complement of the corresponding bit value in the unique opcode, signature checking will likewise be disabled. This suggests that these latter techniques do not cover half of all possible stuck-at column failures.

Namjoo [5] proposes the use of two columns to indicate the start and end of a signature interval. The encoding of the columns is done in a manner that insures that the failure of either column will be detected. While effective at detecting column failures, this method introduces significant overhead. For the 16-bit system studied, the two additional columns represent a 12.5% memory overhead. This overhead was not included when the paper [5] suggests that the technique's overhead could be as small as 10% to 12%.

Furthermore, this accounting of column overhead should be considered a lower bound. In a system with multiple processes, certain processes may not require the increased integrity offered by ESM or may exclude signaturing for performance reasons. For example, it might be desirable to protect the operating system or other critical processes while the remainder are not monitored. The use of dedicated memory columns to delimit signature intervals means that these latter processes would unnecessarily incur memory column overhead. Similarly, if the memory structure is homogeneous across the program and data spaces, the data space is burdened with unnecessary overhead. Because the data space is often larger than the program space this could add a significant amount of overhead. A proper accounting would determine the total column overhead relative to the fraction of the memory that requires signaturing and include that with the total overhead estimate.

3. A New ESM Technique

The previous section demonstrates that existing techniques can at best detect 99% of sequence errors and cannot effectively use signatures larger than 8 bits. While this is adequate for many applications, some require much higher levels of coverage. This section introduces a new ESM technique that takes full advantage of the error detection potential offered by larger signatures.

The Basic Scheme

The strong intermediate signature correlation is caused by an initial intermediate signature that is a constant value and by short signature intervals. This results in a high density of identical intermediate signatures. This problem can be rectified by randomizing the initial intermediate signature for each interval and/or increasing the size of an interval. A new ESM technique is proposed that has both of these desired attributes.

The intermediate signature for the first location of a program to be monitored by the new technique is selected at random. The remaining intermediate signatures are determined as before except that the entire program is treated as a single interval. A properly selected signaturing function should generate a set of intermediate signatures that are randomly distributed.

If the program flow is strictly sequential, no embedded information is necessary for a monitor to regenerate these intermediate signatures. However, any branch will cause the value in the monitor's PLFSR to differ from the branch destination's intermediate signature. The program is made monitorable by adding a *justifying signature* to each non-return branch instruction as shown in Figure 5. The justifying signature is equal to the bit-wise XOR of the intermediate signature of the location following the branch and the intermediate signature of the branch destination. When a branch is taken the justifying signature is XORed with the contents of the PLFSR. Under normal operation the correct destination intermediate signature is extracted. If a conditional branch is not taken, the justifying signature is ignored and the monitor retains the already correct intermediate signature for the following location. In all cases the justifying signature is ignored by the processor's execution unit. Hashing the two intermediate signatures in this manner insures that an error propagates across branch operations

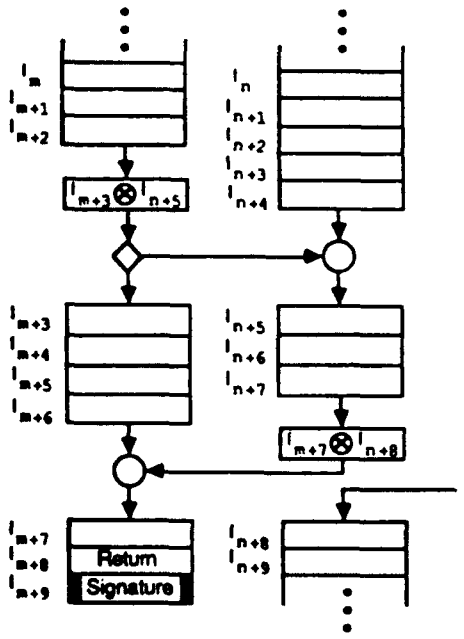


Figure 5: The New ESM Technique

In order to insure a random distribution of intermediate signatures, the monitor must push the current intermediate signature onto its stack when a subroutine call is executed. When a return from subroutine is executed, the monitor pops the intermediate signature value for the location following the subroutine call. The justifying signature associated with the subroutine call will cause an error to propagate into the subroutine. As shown in Figure 5, a signature is embedded following the return from subroutine instruction. Errors that propagate into or occur inside the subroutine can be detected at this point. This signature check point is the only one required by this technique.

The signature of an interrupted program is saved on the monitor's signature stack. The error does not propagate into the interrupting program but is preserved to be detected when the program resumes following the return from interrupt.

The memory overhead required for this system consists of one signature for each return instruction and one justifying signature for the remaining branch instructions. The average error detection latency is equal to half the mean time between the execution of return operations. If this latency is too long, signatures can be embedded at other selected locations.

Refinements to the Basic Scheme

The intermediate signature at a merge node is the same whether location is accessed sequentially or by a branch. The intermediate signature value at a merge node is determined by the sequential access path. The justifying signature associated with the branch is selected so that after decoding, the value matches that determined by the sequential access path.

A program location that can be reached by a branch but cannot be accessed sequentially will be termed an *isolated node*. Because its value is not determined by a sequential access path, the intermediate signature for an isolated node can be selected arbitrarily. The only requirement is that all branches to the isolated node must have the same decoded justifying signature value.

A random distribution of intermediate signatures can be generated for intervals that are smaller than the entire program. An interval can start at any isolated node and must end at a location preceding an isolated node. The intermediate signature for the starting isolated node is selected at random. The intermediate signatures for the rest of the interval are generated as before. This insures a random distribution of intermediate signatures. A subroutine is a typical interval over which intermediate signature generation might occur.

A branch that does not have a merge node as a destination can have its justifying signature eliminated. Figure 6 shows the flow diagram of an If-Else construct. The destination of the conditional branch is an isolated node. As discussed above, the intermediate signature of an isolated node can be selected arbitrarily. The proposed refinement selects the intermediate signature of this isolated node to be the same as the intermediate signature of the location following the branch. Because each path leading from the branch instruction has the same intermediate signature, a justifying signature is not necessary and memory overhead is reduced. An isolated node whose intermediate signature has been selected in this manner must start a new interval. This optimization implies that separate opcodes are necessary to indicate to the monitor whether or not the branch instruction is followed by a justifying signature. These two opcodes are indicated in the Figure 6 as *branch** and *branch* respectively.

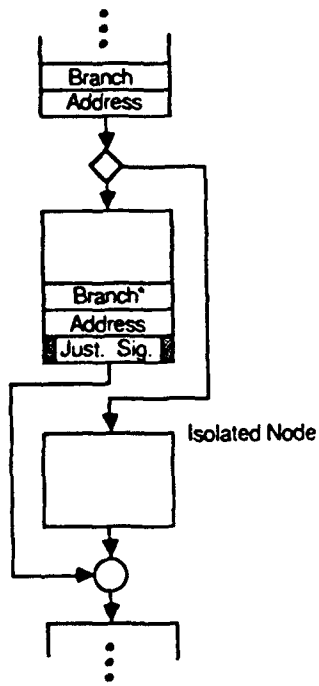


Figure 6: Refinement Applied to If-Else Construct

Coverage and Latency

This new ESM technique's random distribution of intermediate signatures suggests that sequence errors will go undetected at a 2^{-w} rate. Conditional branches without justifying signatures have two destinations with the same intermediate signature. This introduces a slight correlation among the intermediate signatures. Similarly, a register-relative branch that might be generated by a compiler for a Switch statement will have a larger number of destinations with the same intermediate signature. This suggests that the sequence error coverage for the new ESM technique will be close to but somewhat less than $1-2^{-w}$. For a 16-bit signature this corresponds to more than a two order of magnitude improvement compared with existing techniques.

For the new ESM technique, signature checks only occur at return instructions. This reduces the signature density compared with previous techniques and results in a larger error detection latency. The size of the latency increase can be estimated. Except for the procedure call, SIS contains one signature per high level language statement that contains a branch instruction. The statistical distribution of high level language statements presented in [1] implies that 1/6 of the SIS signatures are associated with the return statement. This suggests that the new ESM technique's latency will be roughly six times that of a system using SIS.

4. ESM Overhead Analysis

The original motivation for new approaches to concurrent processor error detection was to provide approaches that are more economical. The two major ESM costs are the added memory that is needed to store the embedded information and the performance lost by the monitored processor. This section analyzes these costs for the ESM techniques that have been proposed and shows that the new ESM technique introduces the least amount of overhead in both categories.

Memory

A new method for comparing memory overhead among the existing and the new ESM techniques is introduced. The program to be monitored is assumed to be written in a high level language using structured programming methods. The HLL constructs that include branch instructions are identified. A typical list consists of the Subroutine (procedure) Call, If, If Else, Return, For, While, Switch, and Do constructs. A flow diagram is created for each of these constructs. Signaturing is then applied to each flow diagram according to the ESM techniques proposed in [5], [9], [13], [14] and the new technique. Figure 7 shows the For construct as signatured by each technique. Table 1 shows the number of words of overhead required by each technique for each construct. The data for the Switch construct assumes four associated cases.

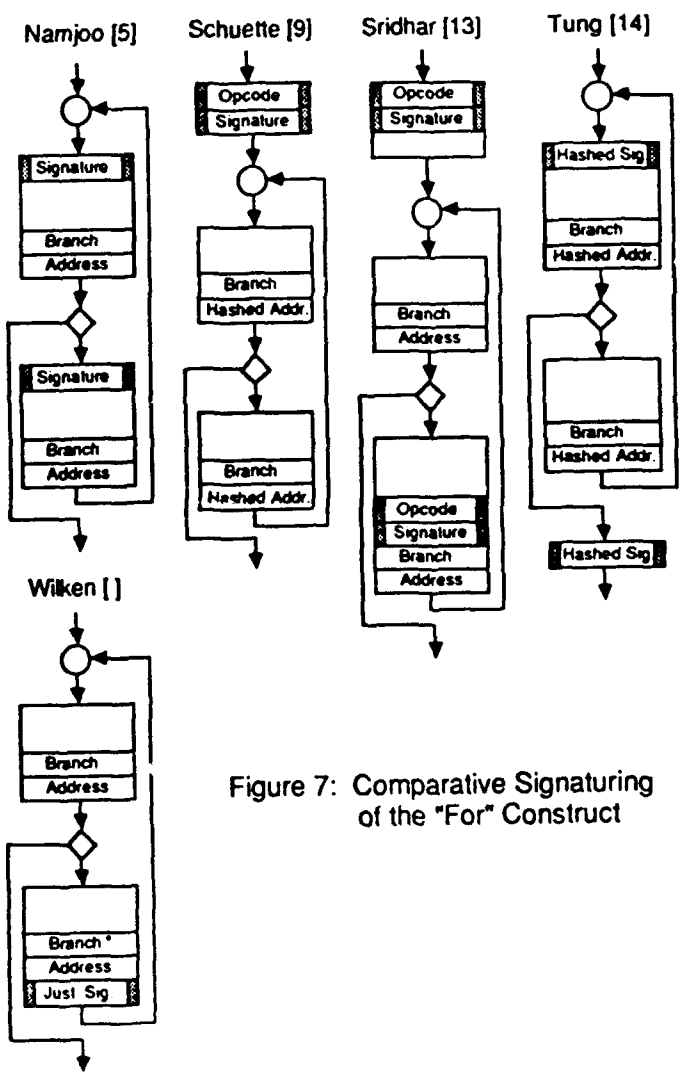


Figure 7: Comparative Signaturing of the "For" Construct

	Namjoo [5]	Schuette [9]	Sridhar [13]	Tung [14]	Wilken [ ]
JSB	1	0	2	0	1
IF	2	2	4	1	1
IF ELSE	2	2	4	2	1
RETURN	1	1	2	1	1
FOR	2	2	4	2	1
WHILE	2	2	4	2	1
SWITCH	4	2	8	4	3
DO	2	2	4	1	1

Table 1: HLL-Construct Memory Overhead

The average overhead per branching statement can be derived based on a statistical distribution. Table 2 shows one such distribution from [1]. Multiplying the distribution in Table 2 by the overheads in Table 1 yields the weighted average overhead per branching statement for each technique as shown in the first row of Table 3. Schuette and Shen [9] report that the SIS overhead is roughly 10%. This data point can be used to convert all of the overhead estimates into percentages. The techniques proposed by Tung and Robinson [14] and Namjoo [5] include one and two bit columns respectively to indicate a signature interval's beginning and end. This overhead is added to each of these techniques assuming a 32-bit processor. The resulting memory overhead percentages are shown in the second row of Table 3.

JSB	IF	IF-ELSE	RETURN	FOR	WHILE	SWITCH	DO
0.38	0.23	0.14	0.10	0.08	0.05	0.02	0.00

Table 2: Statement Distribution

Namjoo [5]	Schuette [9]	Sridhar [13]	Tung [14]	Wilken [ ]
1.56	1.14	3.12	0.95	1.04
20%	10%	27%	12%	9%

Table 3: Weighted-Average Memory Overhead

The new technique results in the least overhead. This occurs for two reasons. First, the signatures are sparse because they only occur following return from subroutine instructions. Second, the locations of actual signatures and the justifying signatures are implied by the preceding processor instruction. No memory overhead in the form of special opcode words or a dedicated column(s) is required to delimit a signature interval. The refined version of the new ESM technique does require a second conditional branch opcode to allow the monitor to determine whether a justifying signature follows. This use of the opcode space is a form of overhead that is not included in the above estimate. While this overhead appears to be small, it is difficult to quantify because it is architecture specific. Excluding this refinement and requiring a justifying signature for all branches would increase the new ESM technique's overhead from 9% to 12% based on the data used above.

As shown earlier, Namjoo [5] proposes the only ESM technique that is immune to column errors. Conventional parity can be added to the new ESM technique to gain this capability at a memory overhead cost of 3% for a 32-bit machine. Unlike the added columns proposed by Namjoo the parity column has utility for programs that are not signatured and for the data space.

Performance

Performance will be lost at points where the processor must ignore embedded signatures and execute a null operation. It has been suggested that processor performance will degrade in rough correspondence to signature overhead [9], [13]. Because the new technique and SIS have similarly low overheads it is expected that they will have similar performance losses. A closer examination shows that the new ESM technique has a decided performance advantage.

Most contemporary processors use a pipeline or a prefetch queue to increase instruction throughput. A branch operation that is taken will cause the contents of the pipeline or queue to be flushed. Each of the new technique's embedded signatures follows a branch instruction. It is possible for a signature to be read from memory and be available to the monitor without impacting processor performance. This occurs when the signature is flushed following the branch and is never executed by the processor.

The frequency at which this occurs can be estimated based on a statistical distribution of branch instructions. The HLL construct flow diagrams and the data from Table 2 suggest that roughly 3/4 of the new technique's signatures follow unconditional branches. These signatures will always be flushed. Half of all conditional branches are assumed to be taken which implies that an additional 1/8 of the signatures are flushed. This suggests that only 1/8 of all signatures used by the new ESM technique impact processor performance. In contrast, roughly 9/10 of the SIS technique's embedded words impact performance. This suggests that the new ESM technique reduces the performance loss by about seven times compared to SIS. Alexander and Wortman [1] report data that shows the dynamic occurrence of unconditional branches is much higher than their static occurrence. This suggests that the estimated reduction in lost performance is conservative.

5. Conclusions

Embedded Signature Monitoring is an efficient and effective approach for concurrent detection of processor control flow errors. This paper presents an analytical method which shows that coverage better than 99% is not achievable with existing techniques. A new ESM technique is introduced which has coverage that approaches $1\text{-}2^{-w}$ when w-bit signatures are used. This corresponds to coverage that is better than 99.99% for a 16 bit signature. A method is also introduced for determining the memory overhead needed by a particular ESM technique. The new ESM technique is shown to be quite efficient, requiring the least memory overhead based on one reported set of program statistics. For a typical contemporary processor, the new technique is shown to substantially reduce performance overhead, better than a seven-fold improvement based on one comparison. A disadvantage of the new ESM technique is that error detection latency increases due to the reduced signature density. A comparative estimate shows a six-fold increase.

The new ESM technique appears to be superior if high coverage, low memory overhead and low performance overhead dominate the system requirements. The technique proposed by Schuette and Shen [9] has similarly low memory overhead and would seem to be the most appropriate if more emphasis is placed on minimizing latency and if coverage and performance are of lesser importance.

Steady improvements have been made in ESM analysis and technique since the approach was introduced five years ago. This paper contributes to both analysis and technique. Research in this area continues. A new approach is being investigated that promises to go significantly beyond the group of techniques analyzed here by way of its ability to expand coverage, reduce latency and allow for less memory and performance overhead. The results of this investigation are forthcoming.

Acknowledgments

This work was supported by the Office of Naval Research (ONR) under contract N00014-86-K-0507. Kent Wilken was partially supported by a graduate fellowship from the General Electric Foundation. A special thanks is extended to Michael Schuette for his discussion and review of this work.

References

[1] Alexander, W. G. and D. Wortman, Static and Dynamic Characteristics of XPL Programs, *IEEE Computer 8*, 11 (November 1975), 41-46.

[2] Carter, W., *Improved Parallel Signature Checkers/Analyzers*, pp. 416-421, Proc. 16th FTCS, IEEE, (1986).

[3] Kobayashi, M., Dynamic Profile of Instruction Sequences for the IBM System/370, *IEEE Transactions on Computers C-32*, 9 (September 1983), 859-861.

[4] Mahmood, A. and E. McCluskey, *Watchdog Processors: Error Coverage and Overhead*, pp. 214-219, Proc. 15th FTCS, IEEE, (1985).

[5] Namjoo, M., *Techniques for Testing of VLSI Processor Operation*, pp. 461-468, Proc. 12th ITC, IEEE, (1982).

[6] Namjoo, M., *Cerberus-16: An Architecture For a General Purpose Watchdog Processor*, pp. 216-219, Proc. 13th FTCS, IEEE, (1983).

[7] Peterson, W. and E. Weldon Jr. , *Error-Correcting Codes*, (MIT Press, 1972).

[8] Schmid, M., R. Trapp, A. Davidoff and G. Masson, *Upset Exposure by Means of Abstraction Verification*, pp. 237-244, Proc. 12th FTCS, IEEE, (1982).

[9] Schuette, M. and J. Shen, Processor Control Flow Monitoring Using Signatured Instruction Streams, *IEEE Transactions on Computers C-36*, 3 (March 1987), 264-276.

[10] Shen, J. and M. Schuette, *On-Line Self-Monitoring Using Signatured Instruction Streams*, pp. 275-282, Proc. 13th ITC, IEEE, (1983).

[11] Shen, J. and S. Thomas, A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems, *Microprocessing and Microprogramming 20*, 4 & 5 (May 1987), 249-269.

[12] Siewiorek, D. and R. Swarz, *The Theory and Practice of Reliable System Design*, (Digital Press, 1982).

[13] Sridhar, T. and S. Thatte, *Concurrent Checking of Program Flow in VLSI Processors*, pp. 191-199, Proc 12th ITC, IEEE, (1982).

[14] Tung, C. and J. Robinson, *On Concurrently Testable Microprogrammed Control Units*, pp. 895-900, Proc. 16th ITC, IEEE, (1986).

[15] Turner, D., R. Burns, and H. Hecht, Designing Micro-Based Systems for Fail-Safe Travel, *IEEE Spectrum 24*, 2 (February 1987), 58-63.

CONTINUOUS SIGNATURE MONITORING:
Efficient Concurrent-Detection of Processor Control Errors

Kent Wilken and John Paul Shen

Center for Dependable Systems
Department of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract -- *This paper presents an efficient approach to concurrent detection of processor control errors using signatured programs. The new approach, called Continuous Signature Monitoring (CSM), makes significant advances beyond the existing signature-monitoring techniques. For typical programs, CSM decreases average error-detection latency by as much as 8 times, down to 1.2 to 1.6 program memory cycles. Memory overhead for storing signatures reaches a theoretical minimum, lowered as much as 4 times, down to 3 to 7%. The CSM monitor is less complex by more than half, and processor-performance loss is reduced as much as 10 times, down to 0.6 to 1.5%. CSM increases coverage of control-flow errors and detects certain types of errors not detected by the existing techniques, including a stuck program counter.*

1. Introduction

Concurrent error detection is necessary to insure dependable processor operation. Although permanent processor faults can be detected using built-in self-test (BIST) or an external tester, concurrent detection must be used if errors caused by permanent and transient faults are to be detected. Transient faults that result from decreased device size are a growing problem [18]. Smaller devices are more susceptible to transient faults because the energy difference between logic levels is lower and because the higher possible speeds reduce timing margins. At the same time, the number of devices per processor is increasing, more processors are subjected to noisy environments, and processor dependability requirements are more stringent.

Structure-Based Error Detection

Traditional approaches to concurrent error detection add redundancy based on the processor's structure. The most common approach is structural duplication, comparing the output of two identical modules. Although effective, duplication is too expensive for most systems.

Redundancy can also be added by decomposing the processor into smaller structures, and then applying the most efficient error-detection technique to each sub-structure. Redundancy is less than 100% if any sub-structure is checked using a technique that is more efficient than duplication. At the highest level, a processor can be decomposed into control and data sections. Data section errors can be detected by parity and arithmetic codes using a small amount of redundancy [18]. Detection of control section errors has been expensive because the section's structure is less regular, although a control section designed using self-checking PLAs and parity-checked microcode may add less redundancy than duplication [7].

Behavior-Based Error Detection

Researchers have proposed detecting processor errors by monitoring the behavior of an executing program [3, 9, 10, 12, 13, 14, 15, 16, 20, 21]. The compiler abstracts the program's behavior and the abstraction is monitored for run-time violations. Abstractions can be formed using various aspects of program behavior, including control flow, memory access, control signals, object type, and object range. Figure 1 shows the typical organization of a processor and its monitor.

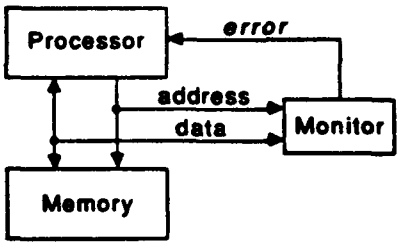


Figure 1: Typical Processor-Monitor Organization.

An advantage behavior-based error detection has over the structure-based approach is that it can insure end-to-end integrity of the abstraction from the point of compilation to the point of execution. Program execution errors caused by any source between these points are potentially detectable. These sources include software errors and hardware design errors, as well as permanent and transient hardware faults.

Behavior-based error detection may prove to be more cost effective than the structure-based approach for many applications. To maximize cost effectiveness, the selected abstraction must allow high error-detection coverage using a simple monitor. Experiments reported by Schmid et al. [15] compared various abstractions and showed that control flow offers the

1988 International Test Conference

most error-detection potential. Several researchers have proposed control-flow monitoring techniques that use a simple monitor and signatured programs [5, 13, 14, 16, 17, 20, 21], an approach called *signature monitoring* [21]. An experimental system showed that signature monitoring is viable and provides significant detection coverage for a small cost [16].

Signature Monitoring

Signature monitoring can be viewed as concurrent *signature analysis* [6], with the analyzer and the reference signatures included in the system, and the executing program used as the stimulus. The compiler computes and stores reference signatures of bit sequences that can be expected during program execution. Dedicated monitor hardware generates signatures of the run-time sequences. The monitor compares the run-time signatures with the reference signatures and declares an error if a difference occurs.

Signature monitoring is effective at detecting processor control errors but not data errors [16], because many control sequences can produce a constant signature where as data sequences generally do not. Bit sequences from any control level that produce a constant signature can be monitored: assembly code [5, 13, 14, 16, 17, 20, 21], microcode [20], or hardware control lines [3]. The best error-detection coverage is provided by signaturing sequences from all control levels and embedding the reference signatures into the assembly code. Without loss of generality, the remainder of this paper will refer to signatures of assembly-level instruction sequences, which could also incorporate lower-level control sequences.

Figure 2 shows a segment of assembly code that is signatured using the *basic technique*. The compiler divides the code into *blocks*, groups of instructions that start at a branch destination or at the location that follows a branch instruction, and end at the next branch instruction or the location that precedes the next branch destination. The reference signature of the instructions within each block is computed and then embedded at the block's end, at which point the code can not be modified. The signature function S is typically a polynomial that generates a cyclic code [18]. An *indicator bit* is set in an added memory column at each reference-signature location, which allows the monitor to distinguish program code and signatures. The processor ignores reference signatures fetched during execution, e.g. it executes NOPs, which reduces its performance.

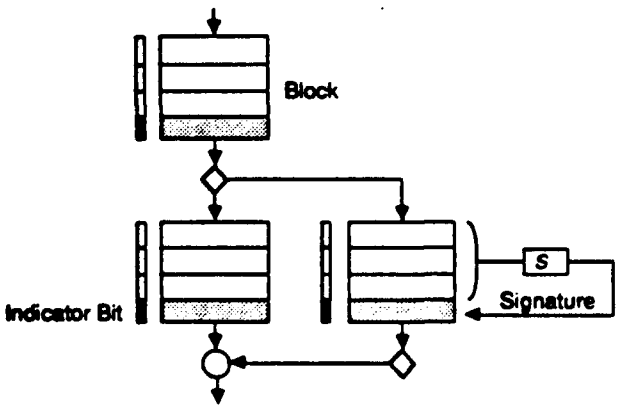


Figure 2: The Basic Signature-Monitoring Technique.

Existing Signature-Monitoring Techniques

Signature-monitoring techniques can be characterized by five properties: (1) error-detection latency, (2) memory overhead, (3) error-detection coverage, (4) monitor complexity, and (5) processor-performance loss. Several researchers have proposed signature-monitoring techniques that improve upon the basic technique in one or more of these properties. Namjoo [13] proposed a technique that encodes into each reference signature an *interval* of instructions, which can include multiple blocks. This technique reduces memory overhead and performance loss because not every block requires an embedded signature. Namjoo [14] also proposed eliminating performance loss by storing signatures in memory that is local to the monitor. Schuette and Shen [16] proposed a technique called *Branch Address Hashing* (BAH) that eliminates the signature storage location that normally follows a branch instruction, thus reducing memory overhead and performance loss. Wilken and Shen [21] showed that all techniques that begin each interval's signature computation with a fixed value, reduce control-flow error coverage. They proposed a technique that increases coverage by randomizing the initial values, and that also decreases memory overhead and performance loss [21].

However, each of these proposed improvements seriously degrades one or more of the other signature monitoring properties. The techniques that reduce memory overhead also increase detection latency because the distance between reference signatures expands. Namjoo's proposal to reduce performance loss increases memory overhead because the compiler must add links between reference signatures to allow the monitor to find the correct reference signatures as the program executes. Branch Address Hashing significantly reduces error coverage because, following the initial error, secondary errors are induced, and a large fraction are undetected [21].

Continuous Signature Monitoring (CSM), the new approach introduced in this paper, makes major improvements in all signature monitoring properties. The following five sections present the techniques that provide the improvements along with methods for quantifying the improvements. The final section summarizes the results and outlines plans for future work.

2. Decreased Error-Detection Latency

This section presents a new signature-monitoring technique that significantly decreases error-detection latency. A short latency can prevent error contamination from spreading and rendering the system undependable. If latency is short, recovery from transient errors can be done with small recovery buffers and can occur within the deadlines imposed by real-time systems.

Vertical Signatures

Existing signature-monitoring techniques encode an instruction sequence by adding a reference signature to each interval in the vertical direction. These are termed *vertical signatures* and are illustrated in Figure 2. Error-detection latency can be long when vertical signatures are used because detection is deferred until the interval's end.

Latency is measured in *program memory cycles* starting with the cycle that follows the cycle containing the error. A program memory cycle is the period from the start of one program-memory access until the start of the next. One program memory cycle may consist of multiple clock cycles for processing complex instructions, operand access, etc. The average detection latency for an error that occurs in an interval of length i is $(i\text{-}1)/2$ program memory cycles, assuming the error is equally likely at all locations. Interval length is both program and technique dependent. Several studies show that the average block size typically ranges from 4 to 10 words [4, 11, 13, 16]. Thus, including the signature added to each block, the basic technique typically has a latency of 2 to 5 program memory cycles.

Detection latency and the fraction of memory used for vertical signatures are inversely related. Existing techniques reduce memory overhead by increasing interval size to include multiple blocks [13, 16, 21], which also increases latency. Conversely, vertical signatures must be added to reduce latency, which also lowers performance and increases cost. Furthermore, for techniques that begin each interval's signature computation with a fixed value [13, 16, 20], adding signatures reduces coverage, because a control-flow error from the end of one interval that lands at the beginning of a wrong interval is undetected [21]. Shorter intervals increase this event's probability.

Horizontal Signatures

The technique proposed here uses *horizontal signatures* to reduce detection latency. Figure 3 shows the h bits added to each word in the horizontal direction that store a horizontal reference signature. The function H generates the horizontal signature for word j by operating on the instruction sequence from the interval's beginning up to and including word j.

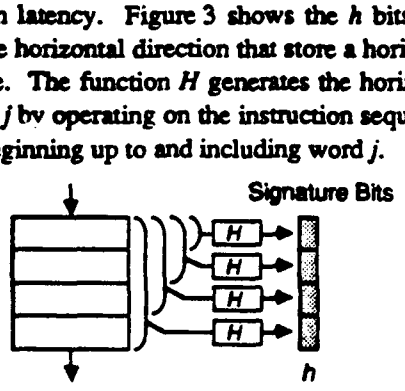


Figure 3: Horizontal Signatures.

Horizontal signatures reduce detection latency because the monitor checks a signature during each program memory cycle. There are 2^h possible horizontal signatures. A random run-time signature produced by an error matches the reference signature at the word where the error occurs with probability 2^{-h}, and the error is undetected. The error remains undetected with probability 2^{-h} at each following word in the interval, because the error causes the function H to produce a pseudo-random run-time signature at that word. The average latency l can be estimated by assuming the interval is infinitely long:

$$l = \sum_{j=1}^{\infty} (j-1)\, 2^{-jh} \qquad (1a)$$

$$= [2^{-h}/(1-2^{-h})]^2 \qquad (1b)$$

Replacing vertical signatures by horizontal signatures significantly reduces detection latency if memory overhead remains constant. For constant memory overhead, a w-bit word, and average interval length i, $i\text{-}1 = w/h$. For a typical value of w (32 bits) and the minimum size for h (1 bit), i is 33 words. This corresponds to an average vertical-signature latency of 16 program memory cycles, compared with a 1 cycle average horizontal-signature latency. As horizontal bits are added, the decrease in horizontal-signature latency is roughly exponential. Increasing vertical overhead by an equal amount decreases vertical-signature latency only linearly.

Horizontal signatures have the added advantage that no performance is lost because the signatures are fetched in parallel with the assembly code. Furthermore, the expected horizontal-signature latency is consistent across the entire program. In contrast, the vertical-signature latency can differ markedly between program sections with different interval sizes.

Horizontal signatures have the drawback that they provide lower error-detection coverage than vertical signatures if equal memory overhead is used. Horizontal signature coverage varies significantly with location in the interval and with interval length. An interval's first word is included in all horizontal signatures in that interval, as illustrated in Figure 3. A random error that occurs at the first word is detected by a horizontal signature check in an interval of length i with probability $1\text{-}2^{-ih}$. The interval's last word is only included in the last horizontal signature. An error occurring there is detected with probability $1\text{-}2^{-h}$. In contrast, a w-bit vertical signature provides coverage of $1\text{-}2^{-w}$ at any interval location for errors that create a random run-time signature, because all but one of the 2^w possible error signatures will differ from the reference signature. If horizontal overhead is equal to the vertical overhead, then $h = w/i$ or $w = ih$. Thus, vertical signatures using equal overhead provide the same high coverage ($1\text{-}2^{-ih}$) for errors at all locations that is provided by horizontal signatures only for errors at the first word of each interval.

Horizontal-Signature Function

Detection latency and coverage can be improved by tailoring the horizontal-signature function H to detect single-bit errors, because single-bit errors occur with a higher frequency than the preceding random-error assumption implies. The function H can use sub-functions P and H^* joined by the XOR operator to

generate one horizontal signature bit for each word, as illustrated in Figure 4. The sub-function P generates the parity of word j. The sub-function H^* generates a one-bit signature of the instructions from the interval's beginning up to but not including word j. Using the sub-function P, the monitor detects with zero latency all single-bit errors and one half of the random errors that occur at word j. Random errors not detected using P are detected using H^* with probability 1/2 at each word in the interval that follows j. The sub-function H^{**} generates the remaining h-1 signature bits. Thus, using the composite function H, the monitor still detects random errors with probability $1\text{-}2^{-h}$ during the first and each subsequent cycle, plus it detects all single-bit errors with zero latency.

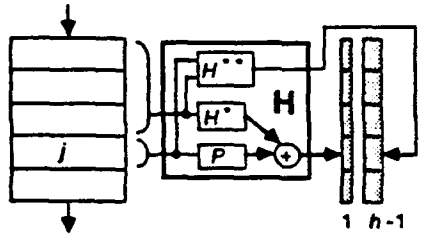


Figure 4: A Refined Horizontal-Signature Function.

One-bit horizontal signatures can be used in the typical computer system without adding memory overhead. Most systems include a parity-bit column in main memory. The one-bit horizontal signature generated by the function $P \oplus H^*$ can replace each program word's parity bit. This expands parity-column usage to include a parity check and a one-bit signature check at each word. Without loss of generality, the remainder of this paper will assume that one-bit horizontal signatures are used and are stored in the existing parity-bit column.

Two-Dimensional Signatures

Horizontal and vertical signatures can be used together to provide short latency and high coverage. The horizontal signatures insure short error-detection latency while the vertical signatures allow high error-detection coverage. Figure 5 shows an interval encoded with signatures in two dimensions. The compiler first uses the function S to generate the vertical signature, then uses the function H to generate a horizontal signature for each word, including one for the word containing the vertical signature. During execution, the monitor uses the two functions to generate both run-time signatures, which are compared with their respective reference signatures.

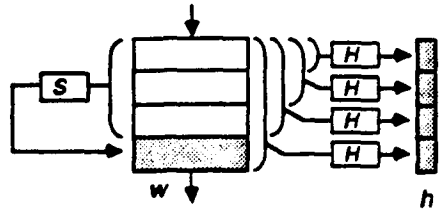


Figure 5: Combining Vertical and Horizontal Signatures.

Although horizontal signatures can be combined with any of the existing vertical-signaturing techniques, the technique proposed by Wilken and Shen provides the highest coverage and the lowest memory overhead [21]. Adding horizontal signatures to this technique results in a combination that provides the shortest latency, the highest coverage, and the lowest memory overhead. The approach that uses this combination of techniques is termed Continuous Signature Monitoring (CSM) because the signatures are continuously checked using the horizontal signatures, and because the vertical signatures maintain error-detection continuity across block boundaries by using *justifying signatures* [21], as explained in the next section. The associated techniques proposed in the remainder of this paper further improve CSM's efficiency and effectiveness.

The function H^* can be selected so that the bit it generates is equal to one of the bits of the *intermediate signature* [21] generated by the function S. The intermediate signature for each word j is the vertical signature from the interval's beginning up to and including word j-1. This selection reduces the time for compiling reference signatures because one of the intermediate signature bits generated for the vertical signature is also used as the horizontal signature bit. Similarly, the monitor hardware for run-time signature generation is less complex than if the two functions were independent. Moreover, the horizontal signatures maintain error-detection continuity across block boundaries along with the vertical signature.

3. Reduced Memory Overhead

The primary limitation on the widespread use of concurrent error detection is cost. Because memory overhead is signature-monitoring's major cost component, minimum memory overhead is necessary if signature monitoring is to reach its full potential. This section presents a technique that reduces vertical memory overhead to the theoretical minimum that is necessary to achieve coverage of $1\text{-}2^{-w}$ using a w-bit signature. A second technique is presented that reduces horizontal overhead by eliminating the indicator-bit memory column.

Minimum Vertical-Overhead Theorem

A program can be represented by a *program graph*, a directed graph that represents each block by a node and each possible transition between blocks by an arc. Figure 6 shows a program graph that represents the program segment shown in Figure 2 and the block that follows it. The program-graph representation will be used for developing a minimum-overhead theorem and an overhead-reduction technique.

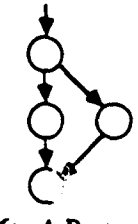


Figure 6: A Program Graph.

The following restrictions are placed on the program to be sig-

natured:

1. Branches are either one-way (unconditional) or two-way (conditional). Where necessary, multi-way branches can be decomposed into two-way branches.
2. The program graph can be determined at compile-time and does not change during execution.
3. There is one entry node, and a path exists from the entry node to all program locations.

A lower bound on the vertical memory overhead required to achieve error-detection coverage of $1\text{-}2^{-w}$ using a w-bit signature can be shown assuming that: (a) the error causes the run-time signature to be a random value, (b) the signature function generates a random intermediate-signature distribution, (c) the monitor contains no memory other than a register for accumulating the run-time signature, and (d) the intermediate signature at each location is unique.

This bound is shown using *maximal paths*, any path in the program graph that starts at a designated location and ends at: (i) an exit node, (ii) a node that is contained in another maximal path, or (iii) an arc where the path cycles back onto itself.

Each maximal path in any set of maximal paths that covers the entire program requires one w-bit signature. If a maximal path ends at an exit (i), a reference signature must be added to and checked at the end of the exit node, otherwise the program can terminate with an undetected error. The reference signature *must be w-bits for coverage of* $1\text{-}2^{-w}$ *at this location. If a maximal* path ends at a node that is contained in another maximal path (ii) or ends by cycling back onto itself (iii), the intermediate signature of the destination of the arc that ends the path must be embedded in the path, so that at run-time the monitor can use it to continue the signature calculation at that destination. Because the intermediate signatures are randomly distributed, the stored intermediate signature must be w-bits. The assumed random distribution of intermediate signatures implies coverage of $1\text{-}2^{-w}$ [21]. Therefore one signature is required per maximal path to achieve the desired detection coverage.

A program with n conditional branches can be partitioned into $n+1$ maximal paths. The first maximal path is signatured starting at the entry node. If an un-signatured maximal path still exists, there must be a conditional branch node that contains one signatured and one un-signatured outgoing arc, otherwise it is not possible to reach the un-signatured sub-graph from the entry node. Maximal paths are signatured starting at the unsignatured outgoing arc of a conditional branch node, until the entire program is signatured. For n conditional branches, n maximal paths are added to the first, for a total of $n+1$ maximal paths. Because each maximal path requires one signature, $n+1$ signatures are required for the program. This argument leads to the following theorem:

Theorem 1: To achieve signature-error coverage of $1\text{-}2^{-w}$ using w-bit signatures, at least $n+1$ signatures are needed, where n is the number of conditional branches in the program.

Minimum Vertical-Overhead Technique

A technique is presented that signatures an arbitrary program graph and meets the bound of Theorem 1. To satisfy the coverage requirement, the intermediate signatures must be randomly distributed, and errors that occur in a path that does not contain a reference signature must be detected. The latter requirement is met by embedding a w-bit justifying signature [13] into each path that merges with another path or itself, instead of embedding the intermediate signature of the merge location. The justifying signature is the XOR of the source-path's signature and the intermediate signature of the merge location. During normal operation, the run-time signature XORed with the justifying signature yields the correct merge-location intermediate signature. When an error occurs, the justifying signature allows the error to propagate from the source path to the destination path, where it can be detected.

CSM Signaturing Procedure. The following signaturing procedure adds only one signature per conditional branch. Intermediate signatures are randomly distributed because only one intermediate signature is a fixed value, therefore the bound of Theorem 1 is met. The procedure places every signature after a branch instruction, and maximizes the number of signatures that follow unconditional branch instructions. Later, this placement will be shown to be useful for reducing horizontal memory overhead and performance loss.

1. The program graph is grouped into *straight paths*, maximal sub-graphs containing nodes that are connected in the program graph and that have contiguous program locations.
2. The arc that enters the program is selected and is labeled with a fixed value, e.g. 0.
3. The straight path that contains the node where the selected arc merges is signatured such that the node's initial intermediate signature equals the incoming arc's label. If all straight paths are signatured, go to Step 5.
4. (*a*) An outgoing unlabeled arc that merges with an unsignatured node is selected from a signatured conditional-branch node; or (*b*) If no arc is selected, an outgoing unlabeled arc that merges with an unsignatured node in another straight path is selected from an unsignatured conditional-branch node. The straight path containing the conditional branch node is signatured using x_j as the initial intermediate signature; or (*c*) If no arc is selected, an unlabeled arc that merges with an unsignatured node in another straight path is selected from a signatured unconditional-branch node. (*d*) The selected arc is labeled with the intermediate signature from the end of the branch node. Go to Step 3.
5. An unlabeled arc that merges with a node whose intermediate signature is a function of some x_j is selected from a branch node whose signature is determined. The arc is labeled with the intermediate signature from the end of the branch node. The variable x_j is resolved by equating the arc's label with the intermediate signature at its merge location.
6. A reference signature is embedded after each program exit.

A justifying signature is embedded after each branch instruction that has an unlabeled arc outgoing from a straight path.

Using this procedure, a justifying signature that follows a conditional branch is always associated with the arc that leaves the straight path, which represents a branch that is taken. Therefore, the monitor must add the justifying signature to the run-time signature if the branch is taken and must skip over the justifying signature if execution is sequential.

Signatured Subroutines with Minimum-Overhead

The subroutine CALL and RETURN form a special class of branch instructions. Because neither is a conditional branch, Theorem 1 suggests that CALL and RETURN should add no signatures. However, because the destination of RETURN can be one of many locations, the two-way branch assumption is violated and the preceding theorem and technique do not apply. The theorem and the technique that follow show that subroutines can be signatured by adding one w-bit signature per conditional branch. The intermediate signature distribution is random, so the bound from Theorem 1 is extended.

A subroutine can be viewed as a control sequence that occurs during a CALL's execution. As with other intra-instruction control sequences, e.g. microinstructions and hardware control signals, if the signature of the sequence is constant, it can be included in the signature of the interval that contains the instruction. The signature of a subroutine is constant if all paths through the subroutine produce the same signature.

Using maximal paths and the assumptions used for Theorem 1, a lower bound on the number of signatures that must be added so that a subroutine's signature is constant can be shown. A maximal path exists from the entry to any exit. One such path is found and its signature is computed. This signature is termed the subroutine's *characteristic signature*. As before, each conditional branch adds a maximal path that starts at one of its arcs, and ends when condition (i), (ii), or (iii) is satisfied. If the added maximal path ends by merging with another maximal path (ii) or with itself (iii), a justifying signature must be added so that the run-time signature calculation can continue at the merge location. If the added maximal path ends at a subroutine exit (i), a justifying signature is added so that the path's signature equals the characteristic signature. Therefore one signature is added for each conditional branch, and any path from the entry node to any exit produces the characteristic signature during error-free operation. This argument leads to the following theorem:

Theorem 2: A constant signature can be obtained for each subroutine by adding one justifying signature per conditional branch in the subroutine.

Figure 7 illustrates a technique that uses characteristic signatures to signature subroutines using minimum memory overhead. The intermediate signature at the return location is the XOR of the preceding interval's signature and the subroutine's characteristic signature. The subroutine's initial intermediate signature is a fixed value, e.g. 0. The procedure described earlier is used to signature the subroutine. However, to produce the characteristic signature for all paths through the subroutine, justifying signatures are embedded after each RETURN instead of reference signatures, except the RETURN that determines the characteristic signature, which has no embedded signature.

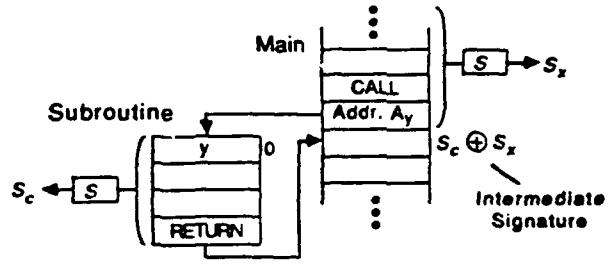


Figure 7: *Minimum-Overhead Subroutine Signaturing.*

To generate the run-time signature, the monitor must use a signature stack. The run-time signature is pushed onto the stack after the CALL is executed. The monitor then sets the run-time signature equal to the fixed value and calculates the subroutine's run-time signature. When a RETURN is executed, the monitor pops the signature off the stack and XORs it with the subroutine's run-time signature to obtain the return location's run-time intermediate signature. Signature errors that occur in the subroutine and are undetected after execution of a RETURN, propagate to the return address where they can be detected. If the signature on the stack contains an undetected error, detection is delayed until subroutine execution completes, which causes average latency to increase.

A Refined Subroutine-Signaturing Technique

A refinement to the preceding technique for signaturing subroutines uses *Branch Address Hashing* (BAH) [16] to eliminate the signature stack and the long latency it can cause. A BAH compiler replaces a branch address by the branch address XORed with the intermediate signature of the location containing the branch address. During execution, the monitor uses the run-time intermediate signature to unhash the branch address for the processor.

Figure 8 illustrates this refinement. All subroutine addresses are hashed, and the intermediate signature at the return location is the subroutine's characteristic signature XORed with the return location's address.

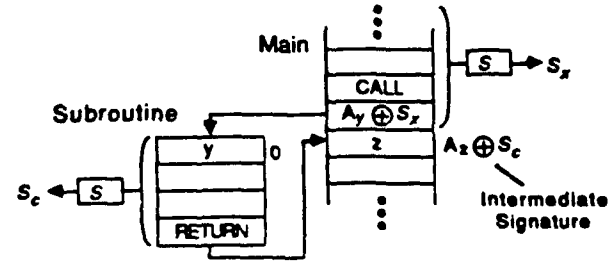


Figure 8: Hashed CALL Addresses.

A run-time signature error at a CALL causes an *induced control-flow error*, which is a branch by the processor to an arbitrary location, caused by the incorrect subroutine address the monitor produces when it unhashes the subroutine address using an incorrect run-time signature. A small fraction of the induced control-flow errors will land at the beginning of an incorrect subroutine, which has the same intermediate signature as the correct destination. These errors will be detected with probability $1\text{-}2^{-w}$ when the subroutine completes, because the characteristic signature will be incorrect following a RETURN. For the remaining induced control-flow errors, the run-time signature does not match the intermediate signature at the error destination with probability $1\text{-}2^{-w}$, and the error is detected after the short CSM latency. This compares with a latency that equals the subroutine's execution time for subroutine addresses that are not hashed.

Hashing subroutine addresses reduces coverage because following an induced control-flow error the run-time signature matches the intermediate signature at the error destination with probability 2^{-w} and the error is not detected. Because only a small fraction of all errors cause induced control-flow errors, the reduction in average detection coverage is small. Moreover, the CSM detection hierarchy discussed in Section 4 can detect many of the control-flow errors that match intermediate signatures.

Signature errors that occur inside the subroutine and are not detected when the subroutine completes must propagate to the calling program for detection. This technique propagates the errors by including the subroutine's characteristic signature in the return location's intermediate signature. However, if this intermediate signature were simply equal to the characteristic signature, all return locations for this subroutine would have the same intermediate signature, which reduces control-flow error coverage. The return address can be XORed with the subroutine's characteristic signature to uncorrelate that location's intermediate signature, as shown in Figure 8. The monitor can access the return address on the data lines when the processor pops it from its return-address stack.

Reduced Horizontal Memory Overhead

At run-time, the monitor must be able to locate embedded signatures. The basic technique does this using indicator bits in an extra memory column. A technique is proposed that allows the monitor to locate signatures without an extra memory column, thus reducing horizontal memory overhead. This technique uses a single type of embedded signature. The CSM signaturing procedure only embeds reference signatures following a program exit, all other embedded signatures are justifying signatures. A program characteristic signature, determined by a path from the entry node to a selected exit, can be used to eliminate the reference signatures. The reference signature at the selected exit is removed. The reference signature at all other exits is replaced by the justifying signature that causes the final signature to equal the program's characteristic signature. The program's final run-time signature can be compared with the characteristic signature using processor software, e.g. the program's characteristic signature can be stored in the program's process control block (PCB), and the comparison can be done by the operating system.

The CSM signaturing procedure places each justifying signature after a branch instruction. For each location j that is a branch instruction followed by a justifying signature, the compiler sets an indicator bit i_j equal to 1, otherwise i_j is set equal to 0. Each indicator bit is XORed with each horizontal signature bit to form x, the *hashed indicator bit*: $x_j = i_j \oplus (P_j \oplus H^*_j)$. The hashed indicator bit is stored in the existing parity column as illustrated in Figure 9.

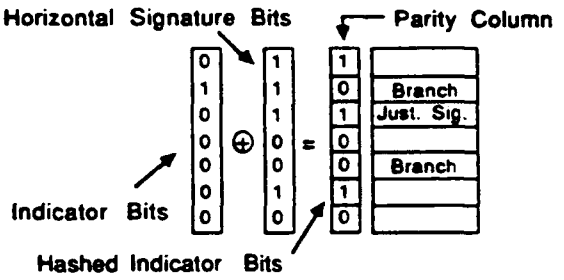


Figure 9: Hashed Indicator Bits.

The monitor uses the run-time horizontal signature bit to unhash the indicator bit from the stored bit. If the run-time signature bit differs from its compile-time value, the run-time indicator bit becomes the complement of its compile-time value. For non-branch locations, the complemented run-time indicator bit equals 1, changed from its compile-time value of 0. This state is illegal because it implies that a justifying signature follows a non-branch location. The monitor can use the unhashed indicator bit and the opcode field to detect this illegal state. For locations that contain branch instructions, an incorrect run-time signature bit complements the run-time indicator bit to become 0 or 1 from its compile-time value of 1 or 0, respectively. Because both values are possible for a branch instruction, the monitor cannot detect the error.

Hashed indicator bits increase error-detection latency because an incorrect run-time signature bit can only be detected at non-branch locations. After an error, the probability is 1/2 that the signature bit is incorrect at the first and each following location. If the fraction of non-branch locations is n and the types of instructions are randomly distributed, then the probability of detecting the error at each location is $n/2$. Substituting $n/2$ for 2^{-h} in equation (1), the detection latency l is:

$$l \approx [(2-n)/n]^2 \qquad (2)$$

If all blocks ended with branches, the fraction of non-branch locations would be 3/4 to 9/10 for the typical average-block-sizes (4 to 10 words). However, many blocks do not end with branches. Some blocks end at non-branch locations that precede a branch destination. Also, some blocks end at a CALL, which

is classified as a non-branch because it is never followed by a justifying signature. Thus, the fraction of non-branch locations will be higher than 3/4 to 9/10. Using the example value n = 0.9, from equation (2), detection latency is 1.5 program memory cycles. This compares with 1 program memory cycle for a one-bit horizontal signature that is not hashed with the indicator bit. For most signature-monitoring applications, this average-latency increase is minor compared with the eliminated cost of the extra memory column.

4. Increased Error-Detection Coverage

This section shows that the CSM approach increases error-detection coverage by detecting more types of errors and by detecting control-flow errors with a higher probability than existing signature-monitoring techniques.

Detection of Additional Error Types

The CSM approach detects the following types of errors not detected by the existing techniques:

False Loops. A previous report noted that if a processor's program counter (PC) is stuck at an address, the existing techniques cannot detect the error unless that address contains a reference signature [21]. In general, any error-created loop that contain no reference signature is not detected by the existing techniques. Sosnowski [19] analyzed error-created *false loops* and showed that the probability that a control-flow error creates a false loop can be as high as 0.1 for some processors. To detect a stuck PC and other false loops containing no reference signature, a watchdog timer [18] must be used with the existing techniques. A watchdog timer increases monitor complexity and adds memory overhead for timer-reset commands, which also decreases processor performance.

The CSM technique can detect false loops without augmentation. At non-branch locations in the loop, the monitor detects the error when the unhashed indicator bit equals 1. A branch instruction that has a compile-time indicator bit equal to 1 must be followed by a justifying signature (a non-branch), which has an indicator bit equal to 0. The monitor can also detect the error when the unhashed indicator bit equals 1 at a branch instruction and equals 1 at the next location.

Program-Bounds Violations. Without increasing cost, the existing techniques do not detect control-flow errors that cause instruction execution from the data space, a program-bounds violation. One existing technique augments its monitor with bounds checking hardware to detect these errors [16].

The CSM approach can detect program-bounds violations using its standard monitor. Following a control-flow error that lands in the data space, the monitor assumes that the fetched parity bit is a hashed indicator bit, and that the fetched data is an instruction. The unhashed "indicator bit" and the "instruction's" opcode field are examined, and the monitor declares an error if the state of these bits is illegal. Assuming the data values are random, the detection probability at each location is determined by the fraction of the architecture's opcodes that are non-branch opcodes. Substituting this fraction for n, equation (2) gives an estimate for the expected detection latency for these program-bounds violations.

Stuck-Incrementing PC. Errors caused by a PC that is stuck incrementing through memory, *i.e.* PC = PC+1, are not detected by some existing techniques. Following this error, program execution will increment from one interval to the next contiguous interval. If the intervals are not connected in the program graph, this constitutes a control-flow error. The techniques that begin each interval with a fixed value [13, 16, 20] cannot detect this error, because the monitor only insures that the beginning of any interval follows the end of the current interval. This error can be detected by placing a reference signature between contiguous intervals that are not connected in the program graph, however this increases memory overhead.

The CSM approach can detect a stuck-incrementing PC without alteration. When execution increments from one interval to a contiguous interval not connected to it in the program graph, a signature error occurs with probability $1\text{-}2^{-w}$, because the first intermediate signature of the contiguous interval is not correlated with the first intermediate signature of the succeeding interval in the program graph.

Control-Flow Error Detection

The subroutine and program characteristic signatures increase coverage of control-flow errors because they provide a hierarchy of detection. A control-flow error is detected wherever it lands with probability $1\text{-}2^{-w}$, because the run-time intermediate signature is incorrect. If the error lands with the correct intermediate signature and lands inside a subroutine, the error is detected with probability $1\text{-}2^{-w}$, because the subroutine's run-time characteristic signature will be incorrect following a RETURN. In a multi-program system, if the error lands inside a different program with the correct intermediate and subroutine characteristic signatures, the error is detected after the program exit with probability $1\text{-}2^{-w}$, because the program's run-time characteristic signature will be incorrect.

The control-flow-error coverage of this detection hierarchy can be estimated. Assume that the error lands at a random location in program memory. The fraction of program memory the executing program occupies is p. The fraction of program memory not occupied by subroutines is s. Because detection is nearly independent at each level in the hierarchy, the probability that the error is not detected at all levels is approximately $ps2^{-w}$, and coverage of control-flow errors is $1\text{-}ps2^{-w}$. This improves upon the best previous result of $1\text{-}2^{-w}$ [21].

A New Coverage Technique

Control-flow errors that land at a location that has the same intermediate signature as the correct location are undetected. A

technique is presented that reduces the number of locations that share intermediate signatures to below that of the signature function's random distribution, thereby increasing coverage of control-flow errors.

The order of the instructions in a block that produces the desired program result is not always unique. The possible orderings are determined by the data dependence relations between instructions [8]. Also, a branch instruction has a fixed position at the end of the block. Within these limitations, the compiler can re-order a block's instructions to reduce the locations that share intermediate signatures. The compiler creates a hash table for storing each location's intermediate signature when it is determined. As each block is signatured, its intermediate signatures are compared with those in the hash table. If one or more of the block's intermediate signatures collides with an entry in the hash table, the block's instructions are re-ordered. For each possible ordering, all collisions can be given a weight that is a function of d, the number of entries in the hash table that share that intermediate signature. Based on the coverage analysis reported in [21], the appropriate weighting function is d^2+d. The ordering of the block's instructions with the least total weight is selected, and the corresponding intermediate signatures are entered into the hash table.

Collisions among subroutine or program characteristic signatures can also be reduced. If more than one exit exists, the characteristic signature can be determined by an exit that does not cause a collision. If no such exit exists, the instructions of each exit block can be re-ordered, and an ordering of a block that has minimum intermediate-signature collision-weight and eliminates the characteristic-signature collision can be selected.

5. Performance and Monitor Complexity

This section discusses the reductions the CSM approach makes in processor performance-loss and monitor complexity.

A Less Complex Monitor

Concurrent detection must be used to detect transient monitor errors. Existing techniques detect such errors using duplication [12]. A recursive application of behavior-based error detection can be used to concurrently detect most of the CSM monitor's transient errors with little increase in monitor complexity. Thus, monitor complexity is reduced by nearly half. The monitor is divided into two parts, as illustrated in Figure 10. The major part, the decoder, inputs the address and the signatured instruction, which includes the hashed indicator bit. The decoder outputs the unsignatured instruction and the unhashed indicator bit. The decoder contains the hardware for: run-time signature generation, parity generation, indicator bit unhashing, opcode decoding and CALL detection, subroutine address unhashing, NOP generation for justifying signatures, and branch detection. The smaller part, the sub-monitor, inputs the unsignatured instruction's opcode field and the unhashed indicator bit.

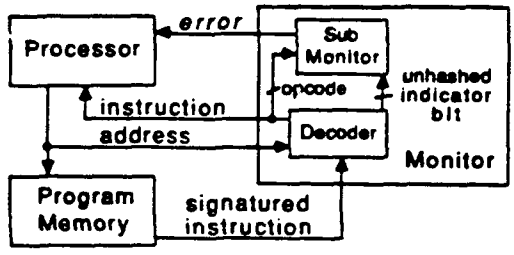


Figure 10: CSM Monitor-Processor Organization.

Using these inputs, the sub-monitor can observe the mutual behavior of the processor coupled with the decoder, and can detect illegal behavior. Because the sub-monitor consists of a small number of gates, errors it produces can be detected using duplication with little increase in monitor complexity.

The CSM monitor's complexity is lower for other reasons. If the system is designed to tolerate transient errors, the CSM approach significantly reduces the size of the recovery buffers, because the detection latency is much shorter. Also, the CSM monitor does not need to add hardware to detect bounds violations, or add a watchdog timer to detect false loops, as shown in Section 4. Horizontal signature checking adds little hardware because the horizontal signature is taken directly from the vertical signature generator, as proposed in Section 2. The monitor's parity generator can replace the system's existing parity generator, and thus does increase system cost. The vertical signature generator is typically a linear feedback shift register (LFSR) [18], which is a module that is also used by BIST techniques. The monitor's LFSR can be used for non-concurrent testing, replacing an existing BIST LFSR, if any.

Subroutine characteristic signatures and subroutine address hashing eliminate the subroutine signature stack, as shown in Section 3. A second stack has been proposed for storing the run-time signature when an exception occurs [16]. This stack can also be eliminated by adding a justifying signature after each exit of the exception handling routine so that its characteristic signature is 0. When the exception occurs, the signature of the handling routine is included in the run-time signature. During normal execution, the run-time signature when the exception completes is the same as when it occurred.

Reduced Processor Performance-Loss

Performance is lost when the processor executes a NOP at a location that contains a justifying signature. A rough performance-loss approximation is made by assuming that execution is equally likely at all locations and that all program memory cycles have the same duration. With these assumptions, the fraction of lost performance is equal to the fraction of vertical memory overhead. The processor executes each signature word (i.e., a NOP) in the minimum program-memory-cycle time. The program memory cycles of some instructions are longer than the minimum. The ratio of the weighted-average program-memory-cycle to the minimum program-memory-cycle can be determined, and then divided into the

fraction of vertical memory overhead, to obtain an improved (and lower) estimate of performance loss. Because the CSM approach achieves minimum vertical memory overhead, it also achieves minimum performance loss.

Performance loss is further reduced for processors that pipeline or pre-fetch instructions, because a signature that follows a branch that is taken is flushed and not executed by the processor [21]. A signature that follows an unconditional branch is always flushed, and a signature that follows a conditional branch is flushed a fraction of the time. Unlike existing techniques, the CSM signaturing procedure places each signature after a branch instruction and guarantees that the maximum number of signatures follow a unconditional branch. Therefore, the CSM approach also achieves minimum performance loss for processors that pipeline or pre-fetch instructions.

Some performance may be lost because CSM subroutine addresses are hashed. To unhash the address, an XOR gate must be placed in the path between the processor and memory, which adds two primitive-gate delays to each memory access.

Namjoo [14] proposed eliminating performance loss by storing signatures in memory that is local to the monitor. This technique can be used with the CSM approach by moving all CSM justifying signatures into monitor memory, and then adding links between the signatures to allow the monitor to find the correct signatures as the program executes. The hashed indicator bits stored in the parity column of program memory are retained, and are unhashed and used by the monitor as before. This combination preserves all the advantages of the CSM approach: a short latency, minimum signature overhead, high coverage, and concurrent detection of monitor errors. To this Namjoo's technique adds the benefit of performance-loss elimination, but also adds the liabilities of increased memory overhead for links between the signatures, and increased monitor complexity for traversing the links. If the monitor is on-chip, which is required for a processor with an on-chip instruction cache or an instruction pre-fetch queue, Namjoo's technique is less viable because the signatures generally cannot be located on-chip.

6. Comparison with Existing Techniques

In this section a method is presented that allows a numerical comparison to be made among different signature-monitoring techniques' memory overhead, latency, performance loss, and control-flow-error coverage. This is an extension of the method used in [21]. The techniques compared are the basic technique, generalized *Path Signature Analysis* (PSA) proposed by Namjoo [13], *Signatured Instruction Streams* (SIS) proposed by Schuette and Shen [16], and the CSM technique.

This method assumes that the signatured program is compiled from a structured high-level language (HLL). The HLL control-flow constructs are identified. A typical list is: IF, IF-ELSE, SWITCH, FOR, WHILE, DO, CALL, and RETURN. The number of blocks each control-flow construct creates is shown in Table 1. (SWITCH is assumed to have four cases.) Figure 2 is an instance of IF-ELSE, which creates 3 blocks.

IF	IF-ELSE	SWITCH	FOR	WHILE	DO	CALL	RETURN
2	3	7	3	3	2	1	1

Table 1: Blocks per HLL Control-Flow Construct.

The vertical overhead used by each technique for each control-flow construct is listed in Table 2. Table 3 shows a relative control-flow-construct usage for programs studied in [1], which is used in making the numerical comparisons. Usage statistics from other programs could also be used.

	Basic	PSA	SIS	CSM
IF	2	1	2	1
IF-ELSE	3	1	2	1
SWITCH	7	3	2	3
FOR	3	2	2	1
WHILE	3	2	2	1
DO	2	2	2	1
CALL	1	1	0	0
RETURN	1	1	1	0

Table 2: Words of Vertical Overhead.

IF	IF-ELSE	SWITCH	FOR	WHILE	DO	CALL	RETURN
0.23	0.14	0.02	0.08	0.05	0.00	0.38	0.10

Table 3: *Relative Control-Flow-Construct Usage.*

Memory-Overhead Comparison

Each technique's vertical overhead can be estimated using these data. The matrix product of Table 3 and each column of Table 2 yields each technique's weighted-average overhead per control-flow construct. The matrix product of Table 3 and the transpose of Table 1 is the weighted-average blocks per construct. Multiplying this by the typical average-block-sizes (4 to 10), cited earlier, yields the weighted-average words per construct. Dividing this into each technique's weighted-average overhead per control-flow construct yields the fraction of vertical overhead, which is shown in the first row of Table 4.

	Basic	PSA	SIS	CSM
Vertical	10-25%	6-15%	6-15%	3-7%
Horizontal	3%	6%	0%	0%
Total	13-28%	12-21%	6-15%	3-7%

Table 4: Estimated Memory Overhead.

The second row of Table 4 lists the horizontal overhead used by each technique, assuming a 32-bit word. The basic technique uses one memory column for an indicator bit. The PSA technique uses two columns, one to indicate the beginning of a path and the other to indicate the path's end [13]. The SIS technique

uses no horizontal overhead because a special opcode word (included in the vertical overhead) precedes each signature to indicate its location [16]. The CSM technique adds no horizontal overhead because the existing parity column is used to store the hashed indicator bit. The third row of Table 4 shows each technique's estimated total overhead. Comparing the midpoint of each range, the CSM technique is seen to reduce total memory overhead by as much as 4 times.

Latency Comparison

The fraction of non-branch instructions n, which determines the CSM technique's detection latency, can be estimated. Table 5 shows the branch instructions per control-flow construct that use an indicator bit. From Tables 1, 3, and 5, the weighted-average branch instructions per block is 0.5. Dividing this by the typical average-block-sizes (4 to 10 words) yields the fraction of branch instructions, 0.05 to 0.12. Thus, the fraction of non-branch instructions n ranges from 0.88 to 0.95. From equation (2), the estimated latency using hashed indicator bits is 1.2 to 1.6 program memory cycles.

IF	IF-ELSE	SWITCH	FOR	WHILE	DO	CALL	RETURN
1	2	6	2	2	1	0	1

Table 5: Branch Instructions per Control-Flow Construct.

	IF	IF-ELSE	SWITCH	FOR	WHILE	DO	CALL	RETURN
PSA	0	0	0	1	1	1	1	1
SIS	1	1	1	1	1	1	0	1

Table 6: Reference Signatures per Control-Flow Construct.

The average detection latencies for the SIS and PSA techniques can also estimated. Table 6 shows the number of reference signatures used by the SIS and PSA techniques for each control-flow construct. Using Tables 1, 3, and 6, the weighted-average reference-signatures per block is 0.32 and 0.33 for the PSA and SIS techniques, respectively. Thus, both techniques have 3 times fewer reference signatures than the basic technique. Vertical reference-signature density and latency are inversely related. Therefore, latency increases from the basic technique's 2 to 5 program memory cycles to an estimated 6 to 15 cycles for the PSA and SIS techniques. Table 7 summarizes the estimated average detection-latency for each technique. Comparing the midpoint of each latency range, the CSM technique decreases latency by as much as 8 times.

Basic	PSA	SIS	CSM
2-5	6-15	6-15	1.2-1.6

Table 7: Average Detection-Latency in P.M. Cycles.

Performance-Loss Comparison

High-performance processors typically pipeline or pre-fetch instructions. Each technique's performance loss can be estimated for such processors. A signature that does not follow a branch will be executed 100% of the time that its block is executed. A signature that follows an unconditional branch will be flushed and is executed 0% of the time. Various studies, e.g. [4], have shown that conditional branches are typically taken 50% of the time. Thus, signatures that follow a conditional branch will typically be executed 50% of the time. For each technique, multiplying these percentages by the number of signatures of that type used in each control-flow construct, and summing the products yields the performance overhead for that construct, which is shown in Table 8.

	Basic	PSA	SIS	CSM
IF	1.5	1	2	0.5
IF-ELSE	1.5	1	2	0
SWITCH	2	3	2	0
FOR	1.5	2	2	0
WHILE	1.5	2	2	0
DO	1.5	2	2	0.5
CALL	0	1	0	0
RETURN	0	1	0	0

Table 8: Performance Overhead.

The matrix product of Table 3 and each column of Table 8 yields each technique's weighted-average performance overhead per control-flow construct. Tables 1 and 3, and the typical average-block-sizes can be used to produce the weighted-average words per construct. Dividing this into each technique's weighted-average performance overhead per control-flow construct yields the fraction of performance lost, which is shown in Table 9. The CSM approach is seen to reduce performance loss by as much as 10 times.

Basic	PSA	SIS	CSM
4-11%	6-15%	6-14%	0.6-1.5%

Table 9: Estimated Performance Loss.

Error-Detection Coverage Comparison

A comparison can be made among each technique's control-flow-error coverage using the analysis reported in [21]. Techniques such as CSM that have a random intermediate signature distribution have control-flow-error coverage of $1\text{-}2^{-w}$, which is higher than 99.99% for a 16 or 32-bit signature. The control-flow-error coverage of the SIS technique was shown to be 84 to 96% [21]. For the basic and PSA techniques, the analysis reported in [21] shows that control-flow-error coverage is less than $1 - (r^2)$, where r is the density of reference signatures. For the basic technique $r = 1/(b+1)$, where b is the block size. For the typical average-block-sizes (4 to 10 words), the control-flow-error coverage of the basic technique is 96 to 99%. Using Tables 3 and 6, and the typical average-block-size, r is 3 to 7% for the PSA technique, which results in an estimated control-flow-error coverage of 99.5 to 99.9%. These results are shown in Table 10. The CSM technique seen to leave orders of magnitude fewer control-flow errors undetected.

Basic	PSA	SIS	CSM
96-99%	99.5-99.9%	84-96%	99.99+%

Table 10: Control-Flow-Error Coverage.

Each of these signature-monitoring techniques can detect all errors that convert a program-memory word into a random value, and can detect all double and triple single bit errors within a specified interval by using the signaturing function proposed by Carter [2]. However, the CSM approach can detect more multiple error patterns because its latency is shorter and detection of the initial error(s) is probable before a following error possibly cancels the error in the run-time signature.

7. Summary and Future Work

This paper presents a new approach to signature monitoring that significantly improves each signature-monitoring property. Signature monitoring, which can be viewed as concurrent signature analysis, is useful for detecting processor control errors at all levels of control: assembly-code, microcode, and hardware control. At the assembly level, coverage of program-memory errors is markedly higher compared with adding an equal number of parity bits. Continuous Signature Monitoring lowers the cost of adding signature monitoring to a system by reducing memory overhead to a theoretical minimum and by decreasing the complexity of the monitor. The short CSM latency can facilitate recovery from transient errors and can prevent the spread of error contamination. By increasing coverage of control-flow errors and multiple program-memory errors, the CSM approach improves signature monitoring's effectiveness. Continuous Signature Monitoring is a viable approach for concurrently detecting processor control errors that is much less expensive than duplication.

In the future, a CSM monitor for a specific processor will be designed. This will provide a better quantitative understanding of monitor complexity and allow experiments to be conducted to confirm the analysis presented in this paper. The CSM monitor can also be used for off-line test. Methods will be explored for generating signatured test programs (i.e. test vectors) so that the CSM monitor can be used to provide good off-line coverage of permanent processor faults.

Acknowledgment

This work was supported by the Office of Naval Research (ONR) under contract N00014-86-K-0507.

References

[1] Alexander, W. G. & D. Wortman, Static and Dynamic Characteristics of XPL Programs, *IEEE Computer 8*, 11 (November 1975), 41-46.

[2] Carter, W., *Improved Parallel Signature Checkers/Analyzers*, pp. 416-421, Proc. 16th FTCS, IEEE, (1986).

[3] Daniels, S., *A Concurrent Test Technique for Standard Microprocessors*, pp. 389-394, Dig. of Papers Compcon Spring 83, IEEE, (1983).

[4] DeRosa, J. & H. Levy, *An Evaluation of Branch Architectures*, pp. 10-16, Proc. 14th Comp. Arch., (1987).

[5] Eifert, J & J. Shen, *Processor Monitoring Using Asynchronous Signatured Instruction Streams*, pp. 394-399, Proc. 14th FTCS, IEEE, (1984).

[6] Frohwerk, R., Signature Analysis: A New Digital Field Service Method, *Hewlett Packard Journal 5* (May 1977), 2-8.

[7] Halbert, M. & S. Bose, *Design Approach for a VLSI Self-Checking MIL-STD-1750A Microprocessor*, pp. 254-259, Proc. 14th FTCS, IEEE, (1984).

[8] Kuck, D., *The Structure of Computers and Computation*, (Wiley, 1978).

[9] Lu, D., Watchdog Processors and Structural Integrity Checking, *IEEE Transactions on Computers C-31*, 7 (July 1982), 681-685.

[10] Mahmood, A. & E. McCluskey, *Concurrent Fault Detection Using a Watchdog Processor and Assertions*, pp. 622-628, Proc. 13th ITC, IEEE, (1983).

[11] Mahmood, A. & E. McCluskey, *Watchdog Processors: Error Coverage and Overhead*, pp. 214-219, Proc. 15th FTCS, IEEE, (1985).

[12] Mahmood, A. & E. McCluskey, Concurrent Error Detection Using Watchdog Processors - A Survey, *IEEE Transactions on Computers 37*, 2 (February 1988), 160-174.

[13] Namjoo, M., *Techniques for Testing of VLSI Processor Operation*, pp. 461-468, Proc. 12th ITC, IEEE, (1982).

[14] Namjoo, M., *Cerberus-16: An Architecture For a General Purpose Watchdog Processor*, pp. 216-219, Proc. 13th FTCS, IEEE, (1983).

[15] Schmid, M., R. Trapp, A. Davidoff & G. Masson, *Upset Exposure by Means of Abstraction Verification*, pp. 237-244, Proc. 12th FTCS, IEEE, (1982).

[16] Schuette, M. & J. Shen, Processor Control Flow Monitoring Using Signatured Instruction Streams, *IEEE Transactions on Computers C-36*, 3 (March 1987), 264-276.

[17] Shen, J. & S. Tomas, A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems, *Microprocessing and Microprogramming 20*, 4 & 5 (May 1987), 249-269.

[18] Siewiorek, D. & R. Swarz, *The Theory and Practice of Reliable System Design*, (Digital Press, 1982).

[19] Sosnowski, J., *Evaluation of Transient Hazards in Microprocessor Controllers*, pp. 364-369, Proc. 16th FTCS, IEEE, (1986).

[20] Sridhar, T. & S. Thatte, *Concurrent Checking of Program Flow in VLSI Processors*, pp. 191-199, Proc 12th ITC, (1982).

[21] Wilken, K. & J. Shen, *Embedded Signature Monitoring: Analysis and Technique*, pp. 324-333, Proc. 17th ITC, (1987).

Concurrent Error Detection Using Signature Monitoring and Encryption

Kent Wilken and John Paul Shen

Center for Dependable Systems
Department of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213 U.S.A.

Abstract -- *This paper presents an efficient approach to concurrent detection of program execution errors that combines signature monitoring with program encryption. Sources of detectable errors include permanent and transient hardware faults, software and hardware design faults, and computer viruses. Errors are detected by a simple monitor that uses signatures embedded in a compatibly encrypted program. The monitor concurrently decrypts the program using the processor control-bit sequences that are included in the signatures. Computer virus attacks are difficult because details of the processor's internal operation are needed to attach compatibly encrypted code. Encryption and a small signature cache added to the monitor allow the lowest memory overhead of any proposed signature-monitoring technique. Encryption and the program memory's error correction/detection code are combined to reduce signature-error detection latency by more than 60 times, while maintaining memory error correction/detection.*

1. Introduction

Complete computer dependability requires detection of errors from all sources. Since the earliest computers, much attention has been focused on detecting errors caused by hardware faults. As system complexity increased, detection of errors caused by software and hardware design faults became important. Although faults are often assumed to be inadvertent, deliberate faults (e.g. computer viruses) cause errors that must be detected. The potential for deliberate faults becomes greater as computer use and computer communication increases. Detection of errors caused by deliberate faults, a problem traditionally considered by computer security researchers, is emerging as a fault-tolerant computing research topic [10].

This paper proposes an efficient behavior-based approach to detecting errors caused by certain hardware, design, and deliberate faults. In the behavior-based approach, a program's behavior is abstracted and the abstraction is monitored for run-time violations. No fault model is assumed, any fault (hardware, design, or deliberate) that causes incorrect program behavior is potentially detectable. To be efficient, the selected abstraction must provide high error-detection coverage at a low cost. Schmid, et al., [15] studied several program abstractions and found that program control flow offers the most error detection potential. Researchers [9, 13, 14, 16, 18, 22] have proposed techniques that monitor program control flow using signatured programs and a simple hardware monitor, a general approach we call *signature monitoring*. This paper proposes a new approach to signature monitoring that increases its efficiency and effectiveness.

1.1 Signature Monitoring

To provide efficient error detection, signature monitoring exploits a common program redundancy: few instructions alter control flow. This redundancy allows program segments containing many instructions to be coded and later checked, as a unit. Figure 1 shows an elementary signature-monitoring technique. The signature compiler divides the assembly code into *basic blocks* [1] and computes a *reference signature* for each block using the function V. The compiler embeds the reference signature at the end of the block, and sets the *indicator bit* in an added memory column at the corresponding location.

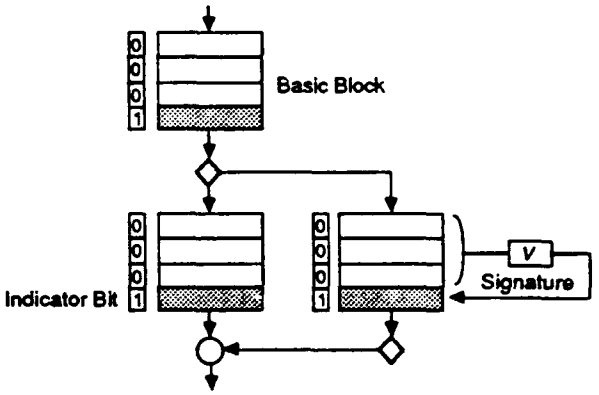


Figure 1: Elementary Signature-Monitoring Technique.

During execution, the monitor generates the block's run-time signature as it observes the executed instructions. At the set indicator bit the monitor compares the run-time signature with the reference signature and declares an error if they differ. Detectable errors include *control-bit errors* and *control-flow errors*. A control-flow error occurs when the instruction execution sequence is incorrect. Control-bit errors result when instructions are executed in the correct order but one or more of the signatured control bits is incorrect. The signature can include control bits from assembly code, microcode, and hardware control lines.

Presented at the 1st International Working Conference on Dependable Computers in Critical Applications

Signature monitoring techniques have been proposed that reduce memory overhead for storing signatures by allowing a *path* containing more than one basic block to be encoded into each signature [13, 16, 22]. The technique proposed by Wilken and Shen [22], *Continuous Signature Monitoring* (CSM), was shown to reduce memory overhead to a theoretical lower bound by partitioning the program into the minimum number of paths. This result is based on the assumptions that the monitor contains no memory other than a register for accumulating the run-time signature, and that control-flow error detection coverage is $1\text{-}2^{-w}$ for a w-bit signature [22]. In Section 2 of this paper, these assumptions are relaxed and a technique is proposed that uses encryption and a small signature cache added to the monitor to further lower memory overhead without impacting coverage.

The CSM technique proposed in [22] significantly reduces the latency for detecting signature errors by using the program memory's parity column to store an encoded bit that allows detection of parity and signature errors at each program location. In Section 3, an encryption-based technique is proposed for use with single-error-correcting/double-error-detecting (SEC/DED) program memory. This technique exploits the SEC/DED code to produce a dramatic reduction in signature-error detection latency, while preserving the code's error correction/detection capability.

1.2 Computer Virus Detection

Recently, the computing community has experienced numerous computer virus attacks [5]. Cohen [3] showed that computer viruses can be created with modest skill and effort, can spread rapidly, and pose a significant security threat. As computers proliferate, the number and severity of computer virus attacks is likely to increase. Effective and efficient virus-detection techniques are needed.

Joseph and Avizienis [9] propose extending signature monitoring to include concurrent virus detection. Signature monitoring can detect a virus, unless the virus is properly signatured. Proper signaturing of a virus may be easy for earlier techniques because they use a single signature function that the attacker might know or easily deduce [9]. Joseph and Avizienis propose using multiple signature functions, one of which is randomly selected by the signature compiler for each program. Using a technique proposed by Namjoo [14], signatures are linked to form a graph that is isomorphic to the program flow graph. While the processor executes the program, the monitor traverses the graph and checks the signatures [14]. Joseph and Avizienis proposed encrypting the signature graph and a vector that represents the function. The decryption key is securely stored, and later delivered to the monitor when the program is loaded. The monitor decrypts the graph and the function's vector, and stores the *plaintext* [2] in its local memory, which is not readable externally. Attacks are averted because a virus cannot easily attach segments to the program that conform to the existing signature graph, or easily alter the program and the encrypted signature graph, without detection.

Although innovative, the encrypted signature-graph approach has limitations. The decryption overhead precludes this approach if process context switches are frequent [9]. For systems that use virtual memory, the monitor's memory is large because it must contain the entire signature graph, even though only a fraction of the program may reside in the processor's real memory. Moreover, the signature graph is large because it contains the signatures plus the links that form the graph. For microprocessors that use an on-chip cache, the monitor must be located on-chip to observe the program's behavior [22]. For this approach, an on-chip monitor requires a separate address and data bus (and possibly its own cache) for accessing the signature graph, to avoid reducing processor performance, and to ensure the privacy of the graph's plaintext.

In Section 4, an alternative approach to concurrent virus detection is proposed that uses signatures embedded in the program. This approach provides significant resistance to virus attacks, and avoids the limitations of the encrypted signature-graph approach. Section 5 summarizes the paper's contributions.

2. Basic Encryption and Signature Caching

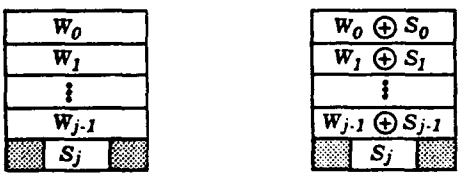
This section introduces the basic approach to combining signature monitoring and program encryption. The improvements provided by this approach include reductions in both memory overhead and error detection latency.

2.1 Basic Encryption

Figure 2a shows a path that is signatured using the conventional approach. A path's signature is the result of a series of intermediate calculations performed at each word in the path:

$$S_k = V(S_{k-1}, C_{k-1}) \qquad (1)$$

where k ranges from 1 to j, V is the signature function, S_0 is a specified initial value, and C_{k-1} represents the values of the monitored control bits during execution of word W_{k-1}. Location k is associated with *intermediate signature* [21] S_k, which for $k>0$ is the signature of the sub-path $[0, k\text{-}1]$. The last intermediate signature, S_j, is the path's signature.



(a) Conventional Approach (b) Instruction Hashing

Figure 2: Program Signaturing.

The conventional approach's only alteration to the assembly code is the embedded reference signatures. For the proposed approach illustrated in Figure 2b, reference signatures are embedded as before, but the signature compiler also encrypts each word using that location's intermediate signature as the key. Figure 2b shows an efficient encryption function, the exclusive-

or (XOR) operator. The monitor generates the run-time intermediate signature, decrypts the word, and delivers the result to the processor for execution.

Schuette and Shen [16] proposed a related technique called *branch address hashing* (BAH) that eliminates reference signatures that follow branch instructions. Each branch address is replaced by the branch address hashed (XORed) with its intermediate signature. Following a signature error, the unhashed branch address becomes a pseudo-random value, and a jump is taken to an arbitrary location, where the error may be detected. The basic encryption approach proposed here can be viewed as a generalization of BAH because all instruction words are hashed, not just branch addresses. Thus, this approach is termed *instruction hashing*.

Instruction hashing provides several improvements to signature monitoring's efficiency. A computer system that uses signature monitoring will generally include numerous other hardware and software mechanisms to detect such errors as illegal opcodes, address or capability violations, etc. Experiments show that such mechanisms can detect a large fraction of processor errors [16]. Following an error that produces an incorrect signature, the intermediate signatures and hence the unhashed instructions are pseudo-random. Execution of pseudo-random instructions will trigger numerous error detection mechanisms, resulting in reduced error-detection latency. Moreover, these mechanisms provide a redundant (and diverse) means for detecting signature errors should the monitor fail in a mode that prevents it from detecting or reporting errors.

Instruction hashing also provides improvements where BAH would otherwise be applied. Signatures that follow branch instructions can be removed using instruction hashing, as with BAH. However, BAH must expand a branch instruction that contains a short branch address into a branch instruction followed by a full-word address [16]. Instruction hashing reduces memory overhead because address expansion is not needed. Also, instruction hashing's unhashing circuit is less complex and adds less delay to instruction fetches because all words are unhashed, compared with BAH's selective unhashing.

An advantage can result from the pseudo-random instruction distribution that instruction hashing produces following a signature error. The CSM technique proposed in [22] uses BAH to eliminate the signature that would otherwise follow each subroutine CALL. After an error, if a CALL is executed before the error is detected, and if the arbitrary destination address (produced by unhashing the branch address) is the beginning of any subroutine, detection of the error is delayed until that subroutine completes [22]. Combining instruction hashing with CSM decreases the probability of these long-latency events, because the fraction of CALLs in a pseudo-random distribution of instructions is generally significantly less than the fraction of CALLs in a program.

In [22], CSM was shown to be more efficient than previous signature-monitoring techniques because it has less memory overhead, lower error-detection latency, higher error-detection coverage, a less complex monitor, and low processor-performance loss. Adding instruction hashing to this technique further increases its efficiency because CSM benefits from all of the aforementioned instruction-hashing improvements. Moreover, CSM combined with instruction hashing is the basis for the signature-caching technique proposed later in this section, the latency reduction technique proposed in Section 3, and the virus-resistant technique proposed in Section 4.

2.2 Justifying Signatures

This subsection reviews *justifying signatures* [13] as background for the signature-caching technique proposed in the next subsection. A justifying signature is a word embedded in a path that sets (justifies) the path's signature to a particular value. Namjoo's [13] *Path Signature Analysis* (PSA) technique uses justifying signatures to reduce memory overhead. In Figure 3, a simple program is represented by a program graph, a directed graph that represents each basic block by a node and each possible transition between basic blocks by an arc. PSA constructs sets of paths that cover all legal sequences of nodes in the graph. All paths in a set start at the same node, and share a common reference signature, which is embedded at the beginning of the starting node. PSA adds justifying signatures to selected nodes so that all paths in a set produce the same signature. The path sets for Figure 3 are {ABD, ABC} and {BD, BC}. Reference signatures are embedded at the beginning of nodes A and B. A justifying signature is embedded in node C or D so that these nodes (and hence the paths in the two path sets) produce the same signature. Thus, PSA uses three signatures for this program, compared with four used by the basic technique.

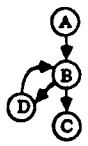


Figure 3: Example Program Graph.

The CSM technique [22] further reduces memory overhead by using justifying signatures to create a program that produces the same signature along any route from entry to exit. CSM could be viewed as a generalization of PSA that needs only one path set and one reference signature to cover an entire program. CSM partitions a program graph into *maximal paths* [22]. The first maximal path is formed starting at the program's entry node. Each remaining maximal path begins at the "branch taken" arc of a conditional-branch node that is already included in a maximal path. A maximal path is formed by adding a node (from a contiguous location if possible) and the connecting arc until the maximal path merges with another maximal path or itself, or a program exit is reached. A justifying signature is embedded in the maximal path so that the path's signature equals the merge location's intermediate signature. The program's reference signature is the signature of one maximal path that ends

at a program exit. A justifying signature is embedded in other maximal paths that end at a program exit so that the path signatures equal the reference signature. CSM requires only two signatures for the program graph in Figure 3: the program reference signature from maximal path ABC and a justifying signature embedded in maximal path D.

2.3 Signature Caching

In [22], CSM was shown to achieve a theoretical lower bound for the number of signatures that must be added to a program, assuming that the monitor contains no memory other than a register for accumulating the run-time signature, and that control-flow error detection coverage is $1\text{-}2^{-w}$ for a w-bit signature. Memory overhead can be further reduced by relaxing these assumptions and adding a small signature cache to the monitor. Within each program loop, a CSM justifying signature provides the monitor with the correct intermediate signature when the processor returns to the loop's first location. However, the monitor previously calculated and discarded this intermediate signature. The proposed monitor stores each calculated intermediate signature and its corresponding address in a small cache. For simplicity, a direct-mapped cache [19] can be used. When a branch instruction is executed, and the instruction is not followed by a justifying signature, the monitor compares the branch's destination address with the addresses in its cache, and copies the corresponding intermediate signature into its signature register when a match occurs. Given the signature-cache size, the signature compiler can determine which of the CSM *justifying signatures can be removed from program loops*. For the program in Figure 3, a cache larger than node D allows node D's justifying signature to be eliminated. Thus, signature caching uses only one signature for this program, the reference signature from maximal path ABC.

To avoid reduced error detection coverage, signature caching must be used with instruction hashing. Without instruction hashing, an error that occurs in a loop where the justifying signature was removed, and that is not detected at the loop's end, is undetectable. This occurs because the error-free intermediate signature of the loop's first location is copied from the cache after the branch instruction at the loop's end is executed. Using instruction hashing, a valid intermediate signature is copied from the cache following an error only if an executed pseudo-random instruction is a branch to a location contained in the cache. Because the cache is small, only a few instructions from the vast instruction space (e.g. 2^{32}) cause such an event. Thus, the decrease in error-detection coverage is negligible.

A technique's memory overhead can be estimated using high-level language control-flow constructs, by determining the number of signatures required for each construct type and determining each type's average size [22]. CSM requires one justifying signature for each construct that contains a conditional branch: IF, IF-ELSE, SWITCH, FOR, WHILE, and DO [22]. Signature caching can remove the signature from those constructs that contain a loop: FOR, WHILE, and DO. For the program statistics used in [22], FOR, WHILE, and DO contain 22% of the total signatures. Thus, signature caching can reduce CSM memory overhead from a range of 3 to 7% [22], to as low as 2.3 to 5.5%.

The optimum cache size depends on a few factors. A larger cache allows the signature compiler to eliminate more justifying signatures, but increases the monitor's cost. Context-switch time increases with cache size because the signature cache is part of the process state that must be saved when a context switch occurs.

3. Short Error-Detection Latency

Computers used in critical applications generally contain single-error-correcting/double-error-detecting (SEC/DED) memory. This section presents an instruction hashing technique that dramatically reduces signature-error detection latency by exploiting the SEC/DED code.

3.1 A New SEC/DED Code

Figure 4 illustrates an SEC/DED *code word* [11] $c_k(X)$ at location k that consists of the w-bit instruction word W_k and $2+\log_2 w = m$ check bits. The code word $c_k(X)$ is the matrix product of W_k and the code's *generator matrix* [11]. The SEC/DED *Hamming code* [11] is well suited for bit-serial communication because the decoder can use a simple linear-feedback shift register (LFSR). For computer memory, Hsiao's [8] SEC/DED code is widely used because it optimizes parallel decoding.

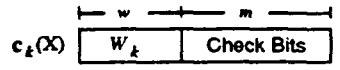


Figure 4: SEC/DED Code Word.

A new code is proposed that uses instruction hashing and an SEC/DED code to reduce error-detection latency. An even *weight* [11] $w+m$ bit hashing vector $s_k(X)$ is formed using the intermediate signature S_k. For example, the vector can be formed by dividing the w-bit intermediate signature into m groups, calculating an even parity bit for each group, and then appending the m parity bits to the intermediate signature. Any even-weight vector is a multiple of $(1+X)$ [11]. Thus:

$$s_k(X) = (1+X)\, q_k(X) \tag{2}$$

The new code word, $C_k(X)$, is the SEC/DED code word XORed with the hashing vector:

$$C_k(X) = c_k(X) + s_k(X) \tag{3}$$

Figure 5 illustrates the decoding organization of the monitor, memory, SEC/DED decoder, and instruction execution unit. During program execution, the monitor generates the run-time intermediate signature S_k', and forms the even-weight unhashing vector $s_k'(X)$. Using $s_k'(X)$, the monitor unhashes the vector read from memory, $C_k'(X)$.

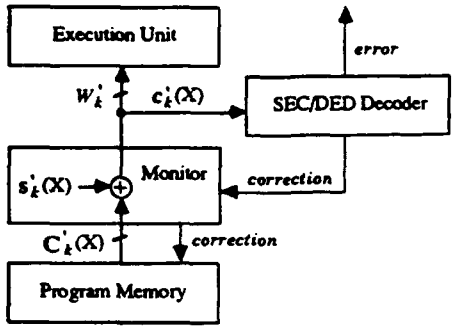


Figure 5: Decoding Organization.

The vector $c_k'(X)$ is received by the SEC/DED decoder:

$$c_k'(X) = C_k'(X) + s_k'(X) \quad (4)$$

3.2 New Code's Performance

The correction/detection performance of this code is analyzed assuming a single fault. If a memory error $e_k(X)$ occurs at location k, $C_k'(X) = c_k(X) + s_k(X) + e_k(X)$. From the single fault assumption, the intermediate signature is error-free because it is derived from previous instructions, i.e. $s_k'(X) = s_k(X)$. Substituting these expressions into equation (4) produces the vector received by the SEC/DED decoder:

$$c_k'(X) = [c_k(X) + s_k(X) + e_k(X)] + s_k(X)$$
$$c_k'(X) = c_k(X) + e_k(X)$$

This same vector would be received by the decoder if the memory error occurred without instruction hashing. Thus, the SEC/DED capability still exists for this new code.

Using the new code, the SEC/DED decoder also detects signature errors. If the run-time intermediate signature contains an error at location k, that error is included in the even weight vector formed by the monitor. The unhashing vector $s_k'(X)$ contains an error of the form:

$$s_k'(X) = (1+X)\, q_k'(X)$$
$$s_k'(X) = (1+X)\, [q_k(X) + e_k(X)]$$

Expanding the terms, and using equation (2):

$$s_k'(X) = s_k(X) + (1+X)\, e_k(X)$$

From the single fault assumption, the memory at location k is error-free, thus $C_k'(X) = C_k(X)$. The vector received by the SEC/DED decoder, $c_k'(X)$, can be determined by substituting these expressions for $s_k'(X)$ and $C_k'(X)$ into equation (4):

$$c_k'(X) = C_k(X) + s_k(X) + (1+X)\, e_k(X)$$

Substituting the expression from equation (3) for $C_k(X)$:

$$c_k'(X) = c_k(X) + s_k(X) + s_k(X) + (1+X)\, e_k(X)$$
$$c_k'(X) = c_k(X) + (1+X)\, e_k(X) \quad (5)$$

A Hamming code word is a multiple of $(1+X)p(X)$, where $p(X)$ is a *primitive polynomial* of degree m-1 [11]. A Hamming SEC/DED decoder divides the received vector by (1+X) and by $p(X)$ to produce the remainders (*syndromes*) r_1 and r_2, respectively [11]. For the new code, both terms of the received vector in equation (5) are multiples of (1+X), thus $r_1 = 0$. The SEC/DED code word $c_k(X)$ is a multiple of $p(X)$. Because $(1+X)e_k(X)$ is a random value with respect to $p(X)$, the syndrome r_2 is uniformly distributed over the 2^{m-1} possible values. With probability 2^{m-1}, $r_2 = 0$ and the vector is assumed to be error-free. With probability $1\text{-}2^{m-1}$, $r_2 \neq 0$ and the decoder reports an uncorrectable error [11].

For Hsiao's code, each column of the *parity check matrix* [11] has odd weight [8]. For the even-weight error vector from equation (5), Hsiao's decoder produces an even weight syndrome, because the syndrome is the sum of an even number of odd weight columns. All but one of the 2^{m-1} even weight syndromes correspond to an uncorrectable error. The remaining even weight syndrome, the all zeros syndrome, indicates no error. Thus, when used with the new code, Hsiao's decoder detects signature errors with the same probability as the Hamming code.

For a 32-bit processor, an SEC/DED decoder detects the signature error with probability 63/64 before the first program word is executed following an error, and if the error remains undetected, with probability 63/64 before subsequent words are executed. The average error detection latency for this geometric series is $(1\text{-}63/64)/(63/64) = 0.016$ program memory cycles. This average latency is more than 60 times shorter than the best existing latency-reduction technique [22]. Moreover, this short latency is achieved without increasing memory overhead.

The new code facilitates recovery from transient errors. When the decoder reports an uncorrectable error, the processor assumes that the error is a transient signature error, and invokes a rollback procedure (e.g. [20]) that restores the processor and monitor to a previous state. The error is deemed uncorrectable only if the rollback fails. The new code's short latency significantly reduces the size and complexity of the rollback buffers. For example, saving a single state for a 32-bit processor allows >98% of transient signature errors to be tolerated. The short, predictable recovery time is well matched to the needs of real-time systems. In addition to transient processor errors, transient monitor errors can be detected and corrected, without duplicating the monitor as required by previous techniques [12].

This approach requires the memory bus width to increase to $w+m$ bits from w bits so that the encoded check bits can be written and read along with the encoded instruction. In addition to the approach's aforementioned benefits, single-bit bus errors are correctable, even-weight bus errors are detectable immediately, and other bus errors are detectable with high probability after a short latency.

4. Computer Virus Resistance

This section proposes extensions to instruction hashing that provide significant resistance to computer virus attacks.

4.1 Modifications to CSM

Instruction hashing combined with CSM provides some resistance to computer virus attacks because a program's assembly code is encrypted using CSM's pseudo-random intermediate signatures. However, this combination contains weaknesses that must be eliminated for the virus resistance to be significant.

First, the typical CSM signature function, a cyclic-redundancy check (CRC) polynomial, allows the program's plaintext to be readily deduced from the hashed program using well known methods [2]. Instead, this function must be replaced by a cryptographic function. The external structure of the cryptographic function is illustrated in Figure 6. The inputs to the function V are the instruction word W_k, the other monitored control bits c_k that occur during the execution of W_k, the function's previous output S_k, and a program key K_p that the signature compiler selects at random for each program. The resulting intermediate signature S_{k+1} is used for hashing and unhashing W_{k+1}.

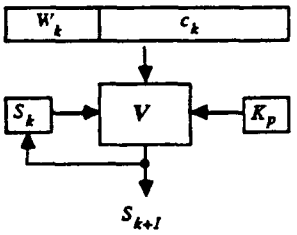


Figure 6: Cryptographic Signaturing Function.

To avoid reducing processor performance, a hardware signature generator should execute the function in real-time, i.e. the result S_{k+1} should be available when the memory fetch of W_{k+1} completes. Also, the key should be difficult to deduce by examining the function's outputs for a known set of inputs, a *known plaintext attack* [2]. In practice these requirements are likely to conflict, with performance taking precedence for most applications. The performance requirement may restrict the possible functions to those for which the program is not *theoretically secure* [2] against a general cryptanalytic attack. However, practical security against a virus attack can still exist. A virus has limited resources: its size and hence its ability to attack are constricted; its computation facility is limited to the host computer; excessive execution time can make the virus conspicuous.

Second, the CSM technique (summarized in Section 2.2) derives all intermediate signatures from the program's instructions and from the initial intermediate signature assigned to the program entry-node's first location [22]. As proposed in [22], the program's initial intermediate signature is the same for all programs, e.g. 0. For virus resistance, the signature compiler selects each program's initial intermediate signature at random.

Third, the CSM technique uses BAH to eliminate justifying signatures following CALLs [22]. The intermediate signature of each subroutine's first location is a fixed value, e.g. 0. For virus resistance, these constant intermediate signatures must be eliminated. This can be done by reinstating a justifying signature after some CALLs, using the technique illustrated in Figure 7. A subroutine is signatured as a part of a maximal path that contains a CALL to the subroutine. A justifying signature is not needed after this CALL. A justifying signature is embedded after other CALLs to the subroutine so that the path's signature equals the intermediate signature of the subroutine's first location. A stack is added to the monitor to save the intermediate signature at a CALL's justifying signature, which is used to derive the intermediate signature at the return (next) location.

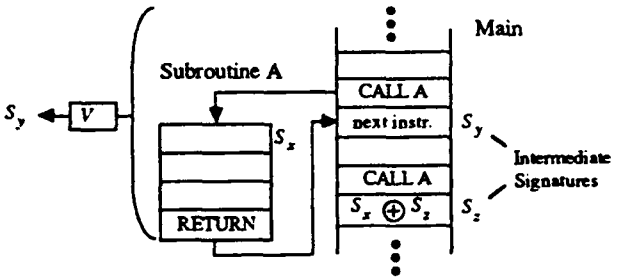


Figure 7: Signaturing Subroutines.

Fourth, the program's initial intermediate signature, reference signature, and key must be securely stored, and then be available to the monitor when the program executes. Also, when a context switch occurs, the contents of the signature register, signature cache, and stack must be securely stored, and then restored when the process resumes. In both cases, these data can be encrypted and decrypted using a cryptographic function contained in the monitor. This function need not execute in real time and might be unrelated to the real-time cryptographic function. This function's key can be loaded during system generation and stored in non-volatile memory that is not readable by the processor. The trusted and encrypted signature compiler writes to the monitor the plaintext of the program's key, initial intermediate signature, and reference signature. The ciphertext of these data is read by the compiler and stored with the encrypted program. When the program executes, the processor delivers to the monitor the ciphertext, which is decrypted and used internally. Similarly, when a context switch occurs, the ciphertext of the contents of the signature register, cache, and stack are read and stored by the processor, and later decrypted by the monitor. The processor cannot read the plaintext of the various signatures or the key stored inside the monitor.

With these changes, instruction hashing poses a significant barrier to virus attacks. Additional virus resistance can be had by exploiting the processor's complex internal behavior.

4.2 Monitor-Assisted Signature Compilation

Previous signature-monitoring papers suggest using a software

signature compiler to generate a program's signatures. Although the signature compiler's complexity and execution time are important, they have not been considered in the literature. Error detection coverage can be increased significantly by including the processor's internal control sequences (from microcode and hardware control lines) in the signature along with the assembly code [16]. However, generating such signatures using software requires that the signature compiler include a model of the processor's control section. Using this model, signature generation is equivalent to simulating the response of the processor's control section to each program instruction. Besides adding significant complexity to the signature compiler, compiler throughput is significantly reduced.

Minor modifications to the monitor and processor can create a new instruction-execution mode that allows the monitor's hardware signature-generator to assist with signature compilation. ENTER MODE and EXIT MODE instructions are added to the instruction set. While the new mode is enabled the following occurs: (1) the monitor does not unhash instructions, (2) instruction output (store to register or memory) is blocked, and (3) the program counter (PC) always increments, i.e. PC = PC + 1. The compiler constructs a code segment that includes the instruction sequence that is the target of signature generation, bracketed by delimiters as illustrated in Figure 5. The leading delimiter is a hashed ENTER MODE followed by a justifying signature that equals the intermediate signature of the sequence's first instruction. The trailing delimiter is EXIT MODE.

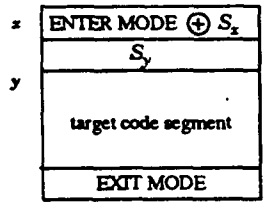


Figure 8: Signature-Generation Code Segment.

After the segment is constructed, ENTER MODE is unhashed and executed, and then the monitor copies the justifying signature into its signature register. Next the target sequence is executed. During execution, each intermediate is calculated and saved in the monitor's cache. After EXIT MODE is executed, the plaintext intermediate signatures are read by the compiler and hashed with the corresponding instructions in the target sequence to form the encrypted instructions.

Using monitor-assisted signature compilation, signatures that include a large number of the internal control sequences are feasible. For maximum error detection coverage, all possible control sequences are included in the signatures. These signatures can capture a significant portion of the processor's complex internal behavior. This complexity provides an instruction-hashed program with significant additional virus resistance.

The proposed instruction-execution mode must be secured so that it can only be enabled by the trusted and encrypted signature compiler during program installation. Thus, a virus cannot receive monitor assistance when attempting to decrypt a program or encrypt itself. The virus must carry its own software decryption/encryption mechanism. Because the complexity of the processor's internal operation must be reflected in this mechanism, it is likely to be large (which makes the virus conspicuous), to be difficult to construct, and to execute slowly.

4.3 Comparison with Previous Approaches

Instruction hashing has many advantages compared with previous approaches to computer virus detection. Various techniques have been proposed that use cryptographic checksums to ensure program integrity when a program is loaded, e.g. [4]. Unlike these techniques, instruction hashing also ensures integrity during program execution. For example, instruction hashing could detect (and preclude) the "fingerd" attack used by the Internet Worm [17]. Cryptographic checksums must be securely stored, otherwise a virus can attach to a program and substitute a new checksum that reflects its changes. Instruction hashing requires no secure storage outside the monitor.

Contrasted with the encrypted-graph approach [9] (summarized in Section 1.2), instruction hashing can be applied where context switching is frequent, it works in a memory hierarchy without complication, and it requires less memory overhead. Using instruction hashing, a monitor failure does not make the system insecure because the failure can be detected by other processor detection mechanisms. The encrypted-graph approach must duplicate the monitor to create a fail-safe system. Unlike the encrypted-graph approach, instruction hashing provides an additional element of security: privacy. Instruction hashing is a deterrent against unauthorized examination or use of a program.

Techniques have been proposed that use program encryption and managed key distribution to prevent software piracy, e.g. [7]. By employing an error-detection code, these techniques can detect modifications to the assembly code, including those caused by computer viruses. Unlike these techniques, instruction hashing can detect control-flow errors, and control-bit errors from all levels in the control hierarchy, including assembly-code modification. By adding managed key distribution, instruction hashing could also be used to deter software pirates.

Unlike the aforementioned approaches, instruction hashing's cryptographic function must execute in real-time. This may limit instruction hashing's effectiveness against a general cryptanalytic attack. The other approaches can be based on non real-time functions that have established effectiveness, e.g. DES [6]. Instruction hashing's security may be derived in part from the concealed details of the processor's internal implementation, which could be exposed. Also, code that is encrypted using internal control sequences is implementation specific and will not execute on different implementations of the same instruction set architecture. This can be a limitation for heterogeneous multiprocessor/multi-computer systems.

5. Summary

The paper presents a new approach to concurrent error detection that combines signature monitoring with encryption. The new approach, called instruction hashing, is shown to be robust because it allows several signature-monitoring advances. Following a signature error, instruction hashing produces pseudo-random instructions, which can trigger various processor error-detection mechanisms. This provides a redundant and diverse means for detecting errors should the monitor fail in a mode that prevents it from detecting or reporting errors. A signature cache is proposed that reduces memory overhead to below that of the best existing technique by eliminating justifying signatures from loops. Instruction hashing prevents signature caching from reducing error-detection coverage. Computers used in critical applications generally use SEC/DED memory. A new code is proposed that uses instruction hashing to exploit the SEC/DED code. The new code retains memory error correction/detection capability, and provides an average signature-error detection latency of 0.016 program memory cycles, 60 times shorter than the best existing technique. This short latency facilitates low-cost recovery from transient hardware faults. Basic instruction hashing provides some resistance to computer virus attacks because the program is hashed with CSM's pseudo-random intermediate signatures. Minor modifications to basic instruction hashing significantly increase virus resistance. Monitor-assisted signature compilation is proposed, which allows numerous internal control sequences to be included in a program's signatures. Hashing instructions with these signatures further increases virus resistance because the virus requires inferred details of the processor's internal operation. Instruction hashing is a low-cost approach that is well suited for applications that must tolerate transient hardware faults, detect permanent hardware faults, and resist attacks by computer viruses.

Acknowledgement

This work was supported by the Office of Naval Research (ONR) under contract N00014-86-K-0507.

References

[1] Aho, A., R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, (Addison-Wesley, 1985).

[2] Beker, H., and F. Piper, *Cipher Systems: The Protection of Communications*, (John Wiley, 1982).

[3] Cohen, F., *Computer Viruses: Theory and Experiments*, pp. 240-263, 7th National Computer Security Conf., , (Sept. 1984).

[4] Cohen, F., A Cryptographic Checksum for Integrity Protection, *Computers & Security 6*, 6 (Dec. 1987), 505-510.

[5] Denning, P., Computer Viruses, *American Scientist 76*, (May-June 1988), 236-238.

[6] National Bureau of Standards, *Data Encryption Standard*, FIPS Publication 46, U.S. Department of Commerce, (1977).

[7] Herzberg, A. & S. Pinter, Public Protection of Software, *ACM Transactions on Computer Systems 5*, 4 (November 1987), 371-393.

[8] Hsiao, M., A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes, *IBM Journal of Research & Development 14*, 4 (July 1970), 395-401.

[9] Joseph, M., & A. Avizienis, *A Fault Tolerance Approach to Computer Viruses*, pp. 52-58, Proc. Symp. on Security and Privacy, IEEE, (1988).

[10] Joseph, M., *Architectural Issues in Fault-Tolerant, Secure Computing Systems*, Ph.D. Dissertation, T.R. #CSD-880047, UCLA Computer Science Dept., (1988).

[11] Lin, S., *An Introduction to Error-Correcting Codes*, (Prentice Hall, 1970).

[12] Mahmood, A. & E. McCluskey, Concurrent Error Detection Using Watchdog Processors - A Survey, *IEEE Transactions on Computers 37*, 2 (February 1988), 160-174.

[13] Namjoo, M., *Techniques for Testing of VLSI Processor Operation*, pp. 461-468, Proc. 12th ITC, IEEE, (1982).

[14] Namjoo, M., *Cerberus-16: An Architecture For a General Purpose Watchdog Processor*, pp. 216-219, Proc. 13th FTCS, IEEE, (1983).

[15] Schmid, M., R. Trapp, A. Davidoff & G. Masson, *Upset Exposure by Means of Abstraction Verification*, pp. 237-244, Proc. 12th FTCS, IEEE, (1982).

[16] Schuette, M. & J. Shen, Processor Control Flow Monitoring Using Signatured Instruction Streams, *IEEE Transactions on Computers C-36*, 3 (March 1987), 264-276.

[17] Spafford, E., The Internet Worm: Crisis and Aftermath, *Communications of the ACM 32*, 6 (June 1989), 678-687.

[18] Sridhar, T. & S. Thatte, *Concurrent Checking of Program Flow in VLSI Processors*, pp. 191-199, Proc 12th ITC, IEEE, (1982).

[19] Stone, H., *High-Performance Computer Architecture*, (Addison-Wesley, 1987).

[20] Tamir, Y., M. Tremblay & D. Rennels, *The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems*, pp. 234-239, Proc. 18th FTCS, IEEE, (1988).

[21] Wilken, K. & J. Shen, *Embedded Signature Monitoring: Analysis and Technique*, pp. 324-333, Proc. 17th ITC, IEEE, (1987).

[22] Wilken, K. & J. Shen, *Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors*, pp. 914-925, Proc. 18th ITC, IEEE, (1988).

Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors

KENT WILKEN, MEMBER, IEEE, AND JOHN PAUL SHEN, MEMBER, IEEE

Abstract—**This paper presents a low-cost approach to concurrent detection of processor control errors that uses a simple hardware monitor and signatures embedded into the executing program. Existing signature-monitoring techniques detect a large portion of processor control errors at a fraction of the cost of duplication. Analytical methods developed in this paper show that the new approach, continuous signature monitoring (CSM), makes major advances beyond existing techniques. CSM reduces the fraction of undetected control-flow errors by orders of magnitude, to less than 10^{-6}. The number of signatures reaches a theoretical minimum, lowered by as much as 3 times to a range of 4–11%. Signature cost is reduced by placing CSM signatures at locations that minimize performance loss and (for some architectures) memory overhead. CSM exploits the program memory's SEC/DED code to decrease error-detection latency by as much as 1000 times, to 0.016 program memory cycles, without increasing memory overhead. This short latency allows transient faults to be tolerated.**

I. INTRODUCTION

CONCURRENT error detection is necessary to ensure reliable computer operation. Although permanent hardware faults can be detected using built-in self-test (BIST) or an external tester, concurrent detection must be used to detect errors caused by transient faults. Transient faults are increasing as device size decreases because the energy difference between logic levels is lower and because the higher possible speeds reduce timing margins [26]. At the same time, the number of devices per computer is growing, more computers are subjected to noisy environments, and computer reliability requirements are more stringent.

Traditional approaches to concurrent error detection add redundancy based on a computer's structure. The most common approach is structural duplication, comparing the output of two identical modules. Although effective, duplication is too expensive for all but a few applications. Redundancy can also be added by decomposing a computer into smaller structures, and then applying the most efficient error-detection technique to each substructure. Redundancy is less than 100% if any substructure is checked using a technique that is more efficient than duplication. A self-checking processor with 40–60% redundancy was implemented using this approach [8].

Manuscript received November 8, 1988; revised August 16, 1989. This work was supported by the Office of Naval Research under Contract N00014-86-K-0507. This paper was recommended by Associate Editor V. K. Agarwal.

The authors are with the Center for Dependable Systems, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA.

IEEE Log Number 9034772.

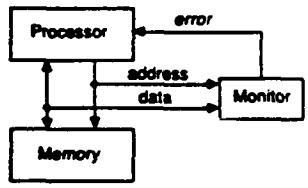


Fig. 1. Typical processor-monitor organization.

Recently, researchers have proposed concurrent error detection using a behavioral abstraction of the executing program that is monitored for runtime violations [11], [14], [15], [17], [18], [19], [20], [22], [23], [29], [34]. Abstractions can be formed using various aspects of program behavior, including control flow, memory access, input–output, and object type or range. Fig. 1 shows an organization of a processor, the monitor, and memory. An advantage behavior-based error detection has over the structure-based approach is that errors from any source are potentially detectable, including software and hardware design faults, as well as permanent and transient hardware faults.

Behavior-based error detection may prove to be more cost effective than the structure-based approach for most applications. A cost-effective abstraction must allow high error-detection coverage using a simple monitor. Experimental comparison of various abstractions shows that control flow offers the most error-detection potential [22]. Several researchers have proposed techniques that detect control-flow errors using a simple monitor and signatured programs [5], [11], [18], [19], [23], [25], [28], [29], [31], [33], [34], [35], an approach we call *signature monitoring*. A prototype system demonstrates that signature monitoring provides significant error-detection coverage at a fraction of the cost of duplication [23].

Signature monitoring can be viewed as concurrent *signature analysis* [6], with the analyzer and the reference signatures included in the system, and the executing program used as the test stimulus. Fig. 2 shows an assembly code segment that is signatured using the *basic technique*. The signature compiler divides the code into *basic blocks* [1], computes each block's reference signature, and then embeds a *signature instruction* at the block's end. The signature instruction has a field that contains an identifying opcode, and a field that contains the signature. The opcode could be a coprocessor opcode already included in the processor's instruction set, or it could be a specific addition to the instruction set.

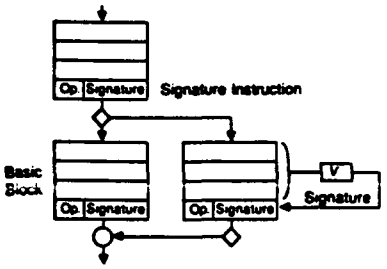


Fig. 2. Basic signature-monitoring technique.

During execution, the monitor observes the executed instructions and generates each basic block's runtime signature using dedicated hardware. At each signature instruction, the processor executes a NOP while the monitor compares the runtime and reference signatures, declaring an error if they differ. The signature function V is typically a cyclic-redundancy check (CRC) polynomial so that the runtime signature can be generated using a parallel-input linear feedback shift-register (PLFSR).

The signature can include deterministic bit sequences from any level in the control hierarchy: assembly code, microcode, or hardware control lines. Generally, data sequences cannot be signatured because they are not deterministic. For maximum error-detection coverage, control bits from all levels can be included in the reference signatures that are embedded in the assembly code. Without loss of generality, the remainder of this paper will consider assembly code signatures that could also incorporate lower-level control bits.

A signature-monitoring technique's effectiveness can be characterized by five properties: (1) error-detection coverage, (2) memory overhead, (3) processor-performance loss, (4) error-detection latency, and (5) monitor complexity. Existing signature-monitoring techniques improve upon the basic technique in one or more of these properties. Namjoo [18] proposes a technique that encodes into each reference signature a *path* of instructions that can include multiple basic blocks. This technique reduces memory overhead and performance loss because not every block requires an embedded signature. Namjoo [19] also proposes eliminating performance loss by storing signatures in memory that is local to the monitor. Schuette and Shen [23] propose a technique that eliminates the signature storage location that normally follows a branch instruction, thus reducing memory overhead and performance loss.

However, all of the proposed improvements degrade one or more of the other signature-monitoring properties. The techniques that reduce memory overhead also increase detection latency because the distance between reference signatures expands. Namjoo's proposal to eliminate performance loss increases memory overhead because links must be added between reference signatures to allow the monitor to find the correct reference signatures as the program executes.

Continuous signature monitoring (CSM), the new approach proposed in this paper, makes major improvements in all signature-monitoring properties. The following four sections present the techniques that provide the improvements, and develop analytical methods for quantifying the properties. In Section VI, these analytical methods are used for a quantitative comparison of the CSM approach and existing signature-monitoring techniques. The final section summarizes the results and outlines plans for future work.

II. Error Detection Coverage

Signature monitoring detects two types of errors [29], *control-bit errors* and *control-flow errors*. Control-bit errors occur when a program is executed in correct order but one or more of the monitored control bits is altered. A control-flow error occurs when the instruction execution sequence is incorrect. A good understanding of control-bit error detection coverage comes from prior research. Mahmood and McCluskey [16] show that signature monitoring can detect all single-word and most memory-column errors. Work on PLFSR aliasing for signature analysis also applies to control-bit error coverage for signature monitoring. Williams *et al.* [36] show that a PLFSR based on a *primitive polynomial* [21] in general produces less aliasing than one based on a nonprimitive polynomial. Iwasaki [10] shows that multiple bit errors can be detected by using a *Reed-Solomon code* [21] as the signature function.

Control-flow error detection coverage has not been thoroughly studied. The subject's only treatment [16] proposes that control-flow errors can be modeled as control-bit errors, and concludes that control-bit coverage analysis applies to both error types. However, this section shows that control-flow error coverage is a distinct problem, and develops a method for analyzing control-flow error coverage.

2.1. Control-Flow Error Detection Coverage

The signature of a path of length j is the result of a series of intermediate calculations performed at each word in the path:

$$S_k = V(S_{k-1}, W_{k-1}) \qquad (1)$$

where k ranges from 1 to j, V is the signature function, S_0 is a specified initial value, and W_{k-1} is the word at location $k-1$. Location k is associated with *intermediate signature* S_k, which for $k > 0$ is the signature of the subpath $[0, k-1]$. The last intermediate signature, S_j, is the path's signature.

Intermediate signatures determine if a control-flow error is detectable. Normally, a program transitions from the current location C to a next location D. A control-flow error causes a transition to a different location D*. If $S_{D^*} = S_D$, the signature calculation continuing from D* will produce a runtime signature that matches the reference signature of the path that contains D*, and the error is undetectable. If $S_{D^*} \neq S_D$, the runtime and reference signatures will differ, and the error is detectable.

The fraction of undetected control-flow errors can be estimated using intermediate signatures. Each location's intermediate signature is determined, and all locations

with intermediate signature i are grouped in D_i. All locations with a destination in D_i are grouped in C_i. A control-flow error is assumed to emanate from any program location with equal probability and errantly transition to any program location with equal probability. Let d_i denote the size of D_i, c_i denote the size of C_i, and m denote the number of program locations. The fraction of undetected control-flow errors originating from a specific location in C_i is $(d_i - 1)/(m - 1) \approx (d_i - 1)/m$ for $m >> 1$. The fraction of control-flow errors that originate from all locations in C_i is c_i/m. Thus the total fraction of undetected control-flow errors is e:

$$e \approx \sum_i (d_i - 1)c_i/m^2. \quad (2)$$

2.2. Correlated Intermediate Signatures

It follows from the preceding analysis that undetected control-flow errors are a minimum if intermediate signatures are uniformly distributed. Correlation among intermediate signatures increases the size of certain groups, and hence, decreases control-flow error coverage. Existing signature-monitoring techniques use a fixed value for S_0, the initial intermediate signature of each path. Thus the first location of each path is in D_{S_0}, and the last location is in C_{S_0}. The intermediate signature of a path's second location, S_1, is a function of S_0 and the path's first word, W_0. Correlated W_0 values cause S_1 values to be correlated. Similarly, if paths begin with correlated sets of words, sets of intermediate signatures will be correspondingly correlated. Kobayashi [12] shows a strong correlation exists among entire basic blocks, an artifact of common language constructs, macro expansion, etc.

A lower bound on the fraction of undetected control-flow errors is set by considering only undetected errors due to S_0. For average path length p, the fraction of locations in D_{S_0} is $1/p$, and the fraction in C_{S_0} is $1/p$, thus:

$$e_{S_0} \approx p^{-2}. \quad (3)$$

Several studies report average block sizes range from 4 to 10 words [4], [12], [16], [18], [23]. Thus the basic technique's average path size (one block plus one signature) is 5–11 words. From (3), a lower bound on undetected control-flow errors for the basic technique is 1–4%, which is coverage of 96–99%. For a v-bit signature, control-flow error coverage for randomly distributed intermediate signatures would be $1 - 2^{-v}$, 99.9985% for a 16-b signature. Thus undetected control-flow errors are largely the product of correlated intermediate signatures.

2.3. Coverage of Induced Control-Flow Errors

Schuette and Shen [23] propose a technique called *branch address hashing* (BAH) that eliminates reference signatures that follow branch instructions, reducing the number of signatures by as much as 50% [24] compared with the basic technique. The signature compiler replaces each branch address by the branch address hashed (XORed) with its intermediate signature. During execution, the monitor unhashes the address for the processor using the runtime intermediate signature. Following a signature error, the unhashed branch address is pseudorandom, and a jump is taken to an arbtitrary location, where the error is possibly detectable. This event is termed an *induced control-flow error.*

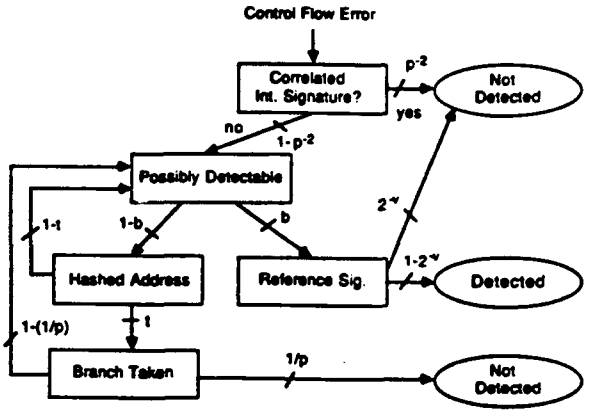


Fig. 3. Model of BAH control-flow error detection.

Induced control-flow errors cause many signature errors to be undetectable. When a branch is taken, the BAH monitor sets the runtime intermediate signature to S_0. Thus an induced control-flow error that lands at the beginning of a path is undetectable. Fig. 3 shows a Markov model of BAH control-flow error detection. The original control-flow error is not detected with probability p^{-2}, as determined by (3). The remaining errors are "possibly detectable," and are detected with probability $1 - 2^{-v}$ in the fraction b of blocks that end with a reference signature, or are not detected in those blocks because the runtime and reference signatures randomly match. The remaining $1 - b$ blocks end with a branch instruction that has a hashed branch address. The branch is not taken with probability $1 - t$ and the error is possibly detectable in the contiguous block. The remaining branches are taken, resulting in an induced control-flow error. With probability $1/p$, the induced control-flow error lands at the beginning of a path and is undetectable. Otherwise, the error is possibly detectable in the error-destination block. Of all possibly-detectable errors, $b(1 - 2^{-v})$ are absorbed in the detected state and $(1 - b)t/p + b2^{-v}$ are absorbed in the not-detected state. Thus the estimated fraction of undetected BAH control-flow errors is:

$$e_{BAH} \approx p^{-2} + (1 - p^{-2})\big((1 - b)t/p + b2^{-v}\big)/\big((1 - b)t/p + b\big). \quad (4)$$

Example values for b, p, and t derived from program statistics presented in Section VI (where various signature-monitoring techniques are compared) will be used to illustrate the significant decrease in control-flow error coverage caused by BAH.

III. Minimum Signatures

As noted in Section II, control-flow error detection coverage can be high $(1 - 2^{-v})$ if intermediate signatures are randomly distributed. This section presents a theorem that sets a lower bound on the number of signatures needed to achieve coverage of $1 - 2^{-v}$ for a random intermediate-signature distribution. A new signaturing technique is

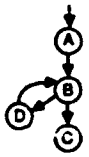


Fig. 4. Example program graph.

presented that uncorrelates intermediate signatures and meets this minimum-signature bound.

3.1. Minimum-Signature Theorem

A program can be represented by a *program graph*, a directed graph that represents each block by a node and all legal transitions between blocks by an arc. Fig. 4 shows an example program graph. The program graph representation is used to develop a minimum-signature theorem and a new signaturing technique.

Definition 1: A *maximal path* is a path in a program graph that is formed starting at any node or arc that is not included in a formed path, and continuing along any route in the program graph until a node contained in a formed path or a program exit node is reached.

The following theorem sets a lower bound on the number of signatures that must be embedded into a program, assuming:

1) branches are either one-way (unconditional) or two-way (conditional) (where necessary, multiway branches can be decomposed into two-way branches);
2) the program graph can be determined at compile-time and does not change during execution;
3) the program has a single entry node;
4) the monitor is memoryless except for the signature register.

Theorem 1: If a program has n conditional branches and its v-bit intermediate signatures are randomly distributed, then $(n + 1)v$ bits must be embedded in the program for control-flow error coverage of $1 - 2^{-v}$.

Proof: This theorem is proved by first showing that a program graph contains $n + 1$ maximal paths, and then showing that each maximal path requires v embedded bits. It follows from Definition 1 that a maximal path that contains a conditional branch node can contain only one of the node's outgoing arcs, unless the maximal path starts at one of these arcs. The outgoing arc not contained in this maximal path starts another maximal path. Thus a maximal path starts at an outgoing arc of each of n conditional branch nodes, and one starts at the program's entry node, for a total of $n + 1$.

A maximal path either ends at a program exit node, or ends at the arc where it merges with another maximal path or itself. The random intermediate-signature distribution implies that the program's control-flow error coverage is at most $1 - 2^{-v}$. For each maximal path that ends at an exit node, if the runtime signature is not checked at the path's end, arriving control-flow errors are undetected because the program terminates, and coverage becomes less than $1 - 2^{-v}$. Thus v bits must be embedded for a signature check for each of these maximal paths. For each maximal path that ends by merging with another maximal path or itself, the monitor must resolve the merge-location intermediate signature to continue. This requires that v bits be embedded for each of these maximal paths because the intermediate signature is a v-bit random value. □

3.2. Minimum-Signature Technique

Namjoo's [18] *path signature analysis* (PSA) technique uses *justifying signatures* to reduce the number of signatures. A justifying signature is a word embedded in a path that sets (justifies) the path's signature to a particular value. PSA constructs path sets that cover all legal sequences of nodes in a program graph. All paths in a set start at the same node, and share a common reference signature, embedded at the beginning of the starting node. PSA embeds justifying signatures into selected nodes so that all paths in a set produce the same signature. The path sets for the program graph in Fig. 4 are {ABD, ABC} and {BD, BC}. Reference signatures are embedded at the beginning of nodes A and B. A justifying signature is embedded in node C or D so that these nodes (and hence, the paths in the two path sets) produce the same signature. Thus PSA uses three signatures for this program, compared with four used by the basic technique. PSA's intermediate signatures are correlated because each path has the same initial intermediate signature.

A new technique uses justifying signatures to uncorrelate intermediate signatures and to lower the number of signatures to the bound established by Theorem 1. These are the technique's basic rules.

1) A program is partitioned into a set of maximal paths that start at the program's entry node and at one outgoing arc of each conditional branch node.
2) The initial intermediate signature of the maximal path that starts at the entry node is a fixed value, e.g., 0.
3) The initial intermediate signature of each maximal path that starts at an outgoing arc of a conditional branch node equals the signature of the path ending at that node.
4) When a maximal path's initial intermediate signature is resolved, the maximal path's intermediate signatures are calculated.
5) A justifying-signature instruction is embedded into each maximal path that merges with another maximal path or itself such that the path's signature equals the intermediate signature of the merge location. For a maximal path that merges with itself, the justifying signature must be embedded between the merge location and the path's end.
6) A justifying-signature instruction is embedded into each maximal path that ends at a program exit such that the path's signature equals the fixed termination signature S_t, e.g., 0.

The new technique's justifying signatures cause the same signature S_t to be produced along any route through a program from entry to exit. Thus this technique could be viewed as a generalization of PSA that uses only one

path set and one (fixed) reference signature to cover an entire program. Error detection latency is generally long using the new technique because signatures can be only checked at program exits. This long latency is eliminated by the companion latency reduction approach presented in Section V.

For illustration, this technique is applied to the program graph in Fig. 4. This graph can be partitioned into one of three maximal-path sets: {ABD, C}, {ABC, D}, or {DBC, A}. For the set {ABD, C}, the intermediate signatures of maximal path ABD are calculated based on the path's fixed initial intermediate signature. A justifying signature is then embedded into node B or D so that the signature of maximal path ABD equals the intermediate signature of node B's first location. Next, the intermediate signatures of maximal path C are calculated based on its initial intermediate signature, the signature of path AB. A justifying signature is embedded into node C so that the signature of maximal path C equals S_t. The new technique uses only two signatures for this program. In Section VI, program statistics are used to show that this technique provides significant memory-overhead reduction compared with existing techniques.

3.3. Additional Signaturing Rules

The new signaturing technique has some additional rules beyond the basic rules listed above. A maximal path may consist of a starting arc and zero nodes, because the starting arc may merge at a node contained in another maximal path. However, each maximal path must contain a location for its justifying-signature instruction. The following rule accommodates zero-node maximal paths: a justifying signature instruction can be located after a conditional branch instruction, and can be included in the maximal path that contains that node, or can be included in the maximal path that starts at an outgoing arc of that node. This signature-instruction placement is compatible with instruction-set architectures that use *delayed branching* [30]. This rule can be implemented by adding a 1-b field to the signature instruction that indicates whether the justifying signature is contained in the path that includes the branch-taken arc, or the path that includes the branch-not-taken arc. Alternatively, for architectures that employ *squashing*, e.g., [3], the branch instructions' squash bit can be used to indicate which of these paths contains the justifying signature.

The initial intermediate signature of all maximal paths but one is the signature of a preceding path. Because the signature of each preceding path is a pseudorandom value, the new technique largely uncorrelates intermediate signatures. However, two sources of correlation remain. First, all paths that merge at a node have the same path signature. If two or more of the merging paths end with a sequence of one or more identical instructions, the intermediate signatures of these sequences will match. For example, this occurs where multiple paths merge at node x, and more than one of these paths end with the instruction JUMP ⟨absolute address of node x⟩. Interestingly, this correlation does not reduce control-flow error coverage because a fault that causes one of these sequences to be executed rather than another does not cause a program error. These sequences and their intermediate signatures are redundant with respect to control-flow faults.

Second, the two locations that follow a conditional branch node have the same intermediate signature. A sequence of words starting at one location that matches a sequence starting at the other causes additional intermediate-signature correlation. This correlation is eliminated by an additional rule: two signatures must be produced for each path that ends at a conditional branch node, one used for the intermediate signature of the branch not-taken location, and the other for the intermediate signature of the branch-taken location. Two signatures are produced directly if the signature includes the processor control line that indicates the result of the branch taken/not taken decision. Otherwise, the second signature can be derived from the path signature, e.g., by complementing one or more of its bits. With this provision, the new technique uncorrelates all nonredundant intermediate signatures and thus provides control-flow error coverage of $1 - 2^{-r}$. Section VI shows that undetected control-flow errors are reduced by orders of magnitude compared with existing techniques.

The initial intermediate signature of a maximal path that starts at an arc that is in a loop is not directly resolvable. For example, for the maximal-path set {DBC, A} of Fig. 4, the initial intermediate signature of maximal path DBC depends on the signature of path DB, which depends on the initial intermediate signature of maximal path DBC. Such a cycle is broken by labeling the initial intermediate signature x and then calculating the maximal path's intermediate signatures as a function of x. The value of x is resolved by equating a known intermediate signature to that intermediate signature's value as a function of x, and then solving for x. For example, the signature of maximal path DBC as a function of x is equated to S_t to resolve x. This topology requires an added rule: for a maximal path that starts at an arc that is in a loop, the justifying signature must be embedded into the maximal path within the loop. For example, the justifying signature of maximal path DBC must be embedded into node D or B.

Exception handlers are signatured as separate programs using the preceding technique. When an exception occurs, the current signature is saved on a signature stack and the signature register is set to its fixed initial value. When exception handling is complete, the current signature is restored from the stack.

3.4. Minimum Signatures For Subroutines

The preceding technique and theorem are based on the assumption that branches are either one-way or two-way. A RETURN from subroutine instruction is in general a multi-way branch and therefore requires separate consideration. This subsection develops a subroutine signaturing technique that provides control-flow error detection coverage of $1 - 2^{-r}$ using a minimum number of signatures.

A CALL node is followed in the program graph by the subroutine's entry node as shown in Fig. 5. One maximal path that contains a CALL to a specific subroutine also

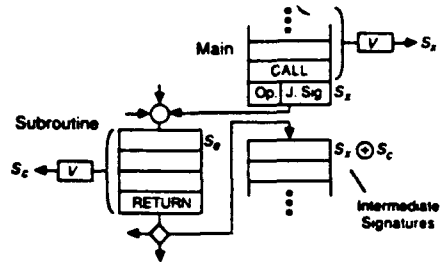


Fig. 5. Subroutine signaturing technique.

contains the subroutine's entry node. This maximal path determines the entry location's intermediate signature, S_e. A justifying signature is embedded into each of the other maximal paths containing a CALL to the subroutine so that each maximal path's signature is S_e. For k CALLs to a subroutine, $k - 1$ signatures are used to justify the signature at the entry node.

Following the technique proposed in Section III-3.2, a subroutine that has n conditional branches is partitioned into $n + 1$ maximal paths. This includes the maximal path that contains the entry node, which is only partly contained in the subroutine. The signature of one maximal path that ends at an exit (RETURN) is designated as the subroutine's *characteristic signature*, S_c. One justifying signature can be embedded into each of the other n maximal paths so that any path through the subroutine from entry to exit produces the same signature, S_c. Thus, $n + k - 1$ justifying signatures are used for the subroutine and its CALLs.

Each subroutine return location starts a maximal path that requires an uncorrelated initial intermediate signature. As illustrated in Fig. 5, the return location's intermediate signature can be the subroutine's characteristic signature S_c XORed with the intermediate signature from the last location of the contiguous CALL node. The intermediate signatures of return locations are correlated if paths that contain CALLs to the same subroutine end with a correlated word, because these paths have the same signature, S_e. These intermediate signatures can be uncorrelated by placing the respective maximal path's justifying signature instruction at the path's last location, *i.e.*, at the location after the CALL. At runtime, when a CALL is executed, the monitor saves on a signature stack the intermediate signature from the path's last location. When a RETURN is executed, the contents of the signature register (S_c for error-free operation) are XORed with the signature retrieved from the stack to produce the return location's intermediate signature.

IV. Reduced Signature Cost

The cost of adding signatures to a program is processor performance loss and memory overhead. The new signaturing technique presented in Section III reduces the number of signatures and hence reduces cost. This section presents a method for reducing the cost of the remaining signatures.

4.1. Signature Placement

Within the stated restrictions for loops, the new signaturing technique allows a maximal path's justifying signature to be placed at any maximal-path location. A justifying signature's cost at certain locations can be less than at others. Low-cost locations include (but are not limited to) the following types: (1) Some architectures require NOPs to be inserted into delay slots that the compiler cannot fill, or require NOPs to resolve pipeline interlocks that the compiler cannot avoid [30]. For example, programs for the MIPS-X architecture contain 16–18% NOPs [3]. A justifying-signature instruction that replaces a NOP incurs zero cost. (2) Branch delay slots are sometimes filled with a replica of the branch destination's instruction, and the branch is altered so that destination = destination + 1. Embedding a justifying-signature instruction into such a delay slot precludes this performance optimization, but does not increase memory overhead. This location's cost can be lower than the cost of a location that reduces performance and increase memory overhead. (3) Some program locations are executed more frequently than others. Execution frequency can be estimated by profiling the program. Except for a location that contains a NOP, a justifying-signature instruction placed at a location that is executed less frequently will incur less performance loss.

Each location's cost can be determined as a function of the memory overhead and the performance loss for placing a justifying-signature instruction there. Each node's signature cost is the cost of its lowest-cost location. Each maximal path's signature cost is the cost of its lowest-cost node. The maximal path's justifying signature is placed at the lowest-cost location in that node. The total signature cost is the sum of all maximal-path signature costs.

4.2. Cost-Reduction Algorithm

Program graphs can generally be partitioned into many maximal-path sets. A program with n conditional branches can be partitioned into at least 2^n distinct maximal-path sets, because the maximal path that starts at an outgoing arc of each conditional branch node can start at either of the two outgoing arcs. The number of maximal-path sets is generally greater than 2^n because for a given set of starting arcs, a node with >1 incoming arcs (a *merge code*) can be contained in any one of the maximal paths that contains an incoming arc. The partitioning at this node is determined by the order of maximal-path formation. Different maximal-path sets can have different signature costs. Partitioning the program graph to achieve minimum signature cost is a *combinatorial optimization problem* [7] that is stated as follows.

Partitioning Problem: Given a program graph and each node's signature cost, partition the graph into a maximal-path set with the minimum total signature cost.

An iterative improvement algorithm is proposed that produces a low-cost maximal-path set. The algorithm starts with an initial maximal-path set, and then iteratively applies a series of local transformations that creates

new maximal-path sets. If a transformation produces a lower cost maximal-path set, the new maximal-path set replaces the original, otherwise it is rejected. The two local transformations used by the algorithm are *swap_outgoing_arcs* and *swap_incoming_arcs*.

Swap_outgoing_arcs operates on the maximal path that contains the conditional branch node i and on the maximal path b_i that starts at an outgoing arc of node i. The path starting at the outgoing arc not contained in b_i becomes a new maximal path. Maximal path b_i is combined with the path that ends at node i to form a second new maximal path. The new maximal paths cover the same nodes and arcs covered by the current maximal paths. If the combined cost of the new maximal paths is lower than that of the current maximal paths, they replace the current maximal paths, and the program's signature cost is reduced.

Swap_incoming_arcs operates on the maximal path m_i containing merge node i and on the k maximal paths that merge at node i. The path from the beginning of maximal path m_i to the arc that merges with node i becomes a new maximal path. Of the k maximal paths, one with the highest signature cost is combined with the path that begins at node i to form a second new maximal path. The new maximal paths replace the two current maximal paths if their combined signature cost is lower. The complete algorithm is described below in pseudocode. Section VI shows that the new technique's signature cost can be significantly reduced using this algorithm.

Signature Cost Reduction Algorithm

```
reduce_signature_cost ()
begin
  path_set = any maximal-path set;
  reduced = TRUE;
  while reduced = TRUE do
    begin
      reduced = FALSE;
      for i:=1 to n do
        begin
          lowered = FALSE;
          swap_outgoing_arcs (cond. branch node i);
          if lowered = TRUE then
            path_set := new_path_set;
        end
      for i:=1 to # of merge nodes do
        begin
          lowered = FALSE,
          swap_incoming_arcs (merge node i);
          if lowered = TRUE then
            path_set := new_path_set;
        end
    end
end.
```

V. Short Error-Detection Latency

This section presents a new signature-monitoring approach that significantly decreases error-detection latency, without increasing memory overhead. A short error-detection latency allows transient faults to be tolerated and prevents error contamination from spreading.

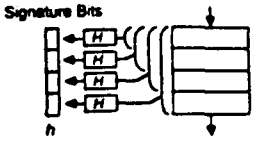


Fig. 6. Horizontal signatures.

5.1. Vertical Signatures

Existing signature monitoring techniques encode an instruction sequence by embedding signatures in the vertical direction as illustrated in Fig. 2. Error-detection latency can be long using *vertical signatures* because detection is delayed until the signature is checked at the path's end. Detection latency is measured in *program memory cycles*, the period from the start of one program-memory access until the start of the next. One program memory cycle may consist of multiple clock cycles for processing complex instructions, operand access, etc. The average detection latency for an error that occurs in a path of length p is $(p-1)/2$ program memory cycles, if errors are equally likely at all locations. The basic technique's latency is 2–5 program memory cycles for its typical average-path-sizes, shown in Section II-2.2 to be 5–11 words.

Detection latency is inversely proportional to the fraction of memory used for vertical reference signatures. Existing signature monitoring techniques [18], [23] reduce memory overhead by using one vertical reference signature to cover multiple basic blocks and thus increase detection latency. The new signaturing technique proposed in Section III fully extends this concept by using one (fixed) reference signature to cover an entire program. Vertical reference signatures can be added to any of these techniques to reduce latency; however, memory overhead increases and processor performance is reduced. For existing techniques that use a fixed initial intermediate signature, adding vertical reference signatures decreases coverage as determined by (3) because average path size decreases.

5.2. Horizontal Signatures

A new signature monitoring approach uses *horizontal signatures* to reduce detection latency. Fig. 6 shows the h bits added horizontally to each word for storing a horizontal reference signature. The function H generates the horizontal signature for word j by operating on the instruction sequence from the path's beginning through word j. Horizontal signatures reduce detection latency because the monitor checks a signature at each program location. There are 2^h possible horizontal reference signatures, thus a random horizontal signature produced by an error differs from the current location's reference signature with probability $1 - 2^{-h}$, and the error is detected.

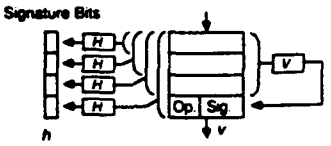


Fig. 7. Combining horizontal and vertical signatures.

An undetected error is detected with probability $1 - 2^{-h}$ at each following location in the path because the error causes the function H to produce a pseudorandom runtime signature for each location. The average latency l can be estimated by assuming an infinitely long path:

$$l = \sum_{j=1}^{\infty} j2^{-jh}(1 - 2^{-h}) = [2^{-h}/(1 - 2^{-h})]. \quad (5)$$

Replacing vertical signatures by horizontal signatures significantly reduces detection latency if memory overhead is constant. For constant memory overhead, $h = v/p$. For the minimum size of h (1 b), $p = v$ and the average vertical-signature latency is $v/2$ program memory cycles, compared with a 1 cycle average horizontal-signature latency. As horizontal bits are added, the decrease in horizontal-signature latency is roughly exponential. For an equal vertical signature increase, vertical-signature latency decreases only linearly. Moreover, horizontal signatures cause no performance loss because the signatures are fetched in parallel with the program code.

Horizontal signatures have the drawback that they provide lower error-detection coverage than vertical signatures for constant memory overhead. Horizontal-signature coverage varies significantly with location in the path and with path length. A path's first word is included in each of the path's horizontal signatures, as shown in Fig. 6. A random signature error at the first location in a path of length p is detected by horizontal signatures with probability $1 - 2^{-ph}$. The path's last word is only included in the last horizontal signature. An error at the last location is detected with probability $1 - 2^{-h}$. In contrast, a v-bit vertical signature can provide coverage of $1 - 2^{-v}$ at any location. If horizontal overhead equals vertical overhead, then $h = v/p$ or $v = ph$. Thus for equal overhead, vertical signatures provide the same high coverage ($1 - 2^{ph}$) at all locations as is provided by horizontal signatures only at each path's first location.

5.3. Two-Dimensional Signatures

Horizontal and vertical signatures can be combined so that a short error-detection latency is ensured by the horizontal signatures while high error-detection coverage is provided by the vertical signatures. Fig. 7 shows a path encoded with signatures in two dimensions. The signature compiler first generates the vertical reference signature using the function V, and then generates a horizontal reference signature for each location (including the vertical signature's location) using the function H. During execution, the monitor regenerates both runtime signatures, and compares them with their respective reference signatures.

Although horizontal signatures can be combined with any existing vertical-signaturing technique, the new approach proposed in Sections III and IV provides the highest control-flow error detection coverage and the lowest signature cost. Adding horizontal signatures to this approach results in a combination that provides the shortest latency, the highest coverage, and the lowest signature cost. This approach is termed *continuous signature monitoring* (CSM) because the horizontal signatures are checked continuously, and because the justifying signatures maintain vertical signature continuity between maximal paths.

5.4. Combining CSM With Parity

A computer's main memory generally contains either a parity code or an error correction/detection code. CSM can be combined with either of these codes to reduce error detection latency without adding horizontal memory overhead. This subsection presents a technique that combines CSM with a parity code.

For this technique, the parity bit normally stored at each location is replaced by a hashed parity bit: $H^* \oplus P$, where H^* is a function that generates a 1-b horizontal signature based on the instruction sequence from the path's beginning through word $j - 1$, and P generates the parity of word j. The function H^* is selected to equal one of the bits from the vertical intermediate signature generated by the function V. This selection reduces signature compilation time and requires less hardware for runtime signature generation than if the functions V and H^* were independent. Moreover, with this selection each maximal path's initial horizontal signature is defined, and a separate mechanism is not required for justifying the horizontal signature where maximal paths merge or end.

The monitor unhashes the parity bit using the runtime signature bit $H^{*\prime}$, and passes the result to the parity decoder. Assuming a single error, the parity decoder detects odd-bit memory errors or signature errors at each location. If $H^{*\prime}$ is error free, the bits received by the parity decoder are the same as if a standard parity bit were used, thus odd-bit memory errors are detectable. If $H^{*\prime}$ contains an error, the single error assumption implies that the corresponding memory location is error free. The parity decoder detects the signature error because it receives an error-free word with a complemented parity bit. Thus this new code adds signature-error detection capability equal to one horizontal bit, without adding horizontal memory overhead. From (5), error detection latency is 1 program memory cycle. Section VI shows that this detection latency is significantly shorter than the detection latency of existing techniques.

5.5. Combining CSM With SEC/DED Codes

A second technique is proposed for use with computers that contain single-error-correcting/double-error-detecting (SEC/DED) memory. An SEC/DED *code word* [13]

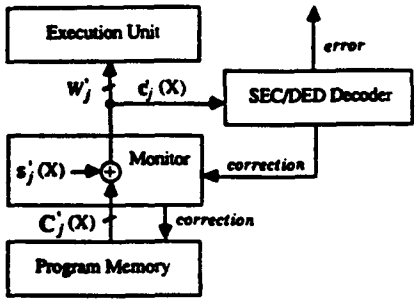


Fig. 8. CSM-SEC/DED decoding organization.

$\mathbf{c}_j(X)$ at location j consists of the w-bit instruction word W_j and $2 + \log_2 w = m$ check bits. The SEC/DED *Hamming code* [13] is well suited for bit-serial communication because the decoder can use a simple linear-feedback shift register (LFSR). For computer memory, Hsiao's [9] SEC/DED code is widely used because it allows fast parallel decoding.

A new code combines CSM with an SEC/DED code to reduce error-detection latency. An even-*weight* [13] $w + m$ bit hashing vector $\mathbf{s}_j(X)$ is formed using the intermediate signature S_j. The vector consists of w zeros at the bit locations that correspond to the instruction, and $m - 1$ b from S_j and an even parity bit corresponding to the check-bit locations. Any even-weight vector is a multiple of $(1 \oplus X)$ [13]. Thus:

$$\mathbf{s}_j(X) = (1 \oplus X)\,\mathbf{q}_j(X). \tag{6}$$

The new code word, $\mathbf{C}_j(X)$, is the SEC/DED code word XORed with the hashing vector:

$$\mathbf{C}_j(X) = \mathbf{c}_j(X) \oplus \mathbf{s}_j(X). \tag{7}$$

Fig. 8 illustrates the decoding organization of the monitor, memory, SEC/DED decoder, and instruction execution unit. During program execution, the monitor generates the runtime intermediate signature S_j', and forms the even-weight unhashing vector $\mathbf{s}_j'(X)$. Using $\mathbf{s}_j'(X)$, the monitor unhashes the vector read from memory, $\mathbf{C}_j'(X)$. The vector $\mathbf{c}_j'(X)$ is received by the SEC/DED decoder:

$$\mathbf{c}_j'(X) = \mathbf{C}_j'(X) \oplus \mathbf{s}_j'(X). \tag{8}$$

5.6. *New Code's Performance*

The correction/detection performance of this code is analyzed assuming a single error. If a memory error $\mathbf{e}_j(X)$ occurs at location j, $\mathbf{C}_j'(X) = \mathbf{c}_j(X) \oplus \mathbf{s}_j(X) \oplus \mathbf{e}_j(X)$. From the single error assumption, the intermediate signature is error-free because it is derived from previous instructions, i.e., $\mathbf{s}_j'(X) = \mathbf{s}_j(X)$. Substituting these expressions into (8) produces the vector received by the SEC/DED decoder:

$$\mathbf{c}_j'(X) = [\mathbf{c}_j(X) \oplus \mathbf{s}_j(X) \oplus \mathbf{e}_j(X)] \oplus \mathbf{s}_j(X)$$

$$\mathbf{c}_j'(X) = \mathbf{c}_j(X) \oplus \mathbf{e}_j(X).$$

This same vector would be received by the decoder if the memory error occurred using a standard SEC/DED code. Thus the SEC/DED capability is maintained for this new code.

Using the new code, the SEC/DED decoder also detects signature errors. If the runtime intermediate signature contains an error at location j, that error is included in the even-weight vector produced by the monitor. The unhashing vector $\mathbf{s}_j'(X)$ contains an error of the form:

$$\mathbf{s}_j'(X) = (1 \oplus X)\,\mathbf{q}_j'(X)$$

$$\mathbf{s}_j'(X) = (1 \oplus X)\,[\mathbf{q}_j(X) \oplus \mathbf{e}_j(X)].$$

Expanding the terms, and using (6):

$$\mathbf{s}_j'(X) = \mathbf{s}_j(X) \oplus (1 \oplus X)\,\mathbf{e}_j(X).$$

From the single error assumption, the memory at location j is error-free, thus $\mathbf{C}_j'(X) = \mathbf{C}_j(X)$. The vector received by the SEC/DED decoder, $\mathbf{c}_j'(X)$, can be determined by substituting these expressions for $\mathbf{s}_j'(X)$ and $\mathbf{C}_j'(X)$ into (8):

$$\mathbf{c}_j'(X) = \mathbf{C}_j(X) \oplus \mathbf{s}_j(X) \oplus (1 \oplus X)\,\mathbf{e}_j(X).$$

Substituting the expression from (7) for $\mathbf{C}_j(X)$:

$$\mathbf{c}_j'(X) = \mathbf{c}_j(X) \oplus \mathbf{s}_j(X) \oplus \mathbf{s}_j(X) \oplus (1 \oplus X)\mathbf{e}_j(X)$$

$$\mathbf{c}_j'(X) = \mathbf{c}_j(X) \oplus (1 \oplus X)\,\mathbf{e}_j(X). \tag{9}$$

A Hamming code word is a multiple of $(1 \oplus X)\mathbf{p}(X)$, where $\mathbf{p}(X)$ is a primitive polynomial of degree $m - 1$ [13]. A Hamming SEC/DED decoder divides the received vector by $(1 \oplus X)$ and $\mathbf{p}(X)$ to produce the *syndromes* s_1 and s_2, respectively [13]. For the new code, both terms of the received vector in (9) are multiples of $(1 \oplus X)$, thus $s_1 = 0$. The SEC/DED code word $\mathbf{c}_j(X)$ is a multiple of $\mathbf{p}(X)$. Because $(1 \oplus X)\,\mathbf{e}_j(X)$ is random with respect to $\mathbf{p}(X)$, the syndrome s_2 is uniformly distributed over the 2^{m-1} possible values. With probability 2^{m-1}, $s_2 = 0$ and the vector is assumed to be error-free. With probability $1 - 2^{m-1}$, $s_2 \neq 0$ and the decoder reports an uncorrectable error [13]. Thus the new code provides detection capability equivalent to $m - 1$ horizontal signature bits, without adding horizontal memory overhead. For a 32-b processor, $m - 1 = 6$ and the average error detection latency from (5) is 0.016 program memory cycles. Section VI shows that this detection latency is orders of magnitude lower than that of existing techniques.

For Hsiao's code, each column of the *parity check matrix* [13] has odd weight [9]. The even-weight error vector in (9) produces an even-weight syndrome, because the syndrome is the GF(2) [13] sum of an even number of odd weight columns. All but one of the 2^{m-1} even-weight syndromes indicate an uncorrectable error. The remaining even weight syndrome and the all zeros syndrome indicates no error. Thus when used with the new code, Hsiao's code provides the same detection capability as the Hamming code.

5.7. *Transient Fault Tolerance*

Previous signature monitoring work considers error detection, but not transient-fault tolerance. CSM allows significant transient-fault tolerance. Most processors use a pipeline with several stages to improve performance. Although an instruction's horizontal signature is checked at the pipeline's first stage, the instruction's effect does not occur for k stages. An errant instruction is detected with probability $1 - 2^{-kh}$ before its effect occurs. A transient fault can be tolerated with this probability by invalidating the instructions that are still in the pipeline when a signature error is detected, and then restarting execution at the leading instruction's address. Because this same process is used for handling other exceptions [3], transient faults are tolerated with little or no added complexity. For a five stage pipeline where results are posted at the last stage, e.g., [3], $k = 4$ and transient faults are tolerated with probabilities $1 - 2^{-24}$ and $1 - 2^{-4}$ for the SEC/DED and the parity techniques, respectively.

5.8. *Other CSM Benefits*

Concurrent detection must be used to detect errors caused by transient monitor faults. Existing techniques detect such errors by duplicating the monitor [17]. For CSM, transient monitor errors are detected by the SEC/DED or parity decoder. Thus duplication is not needed and monitor complexity is halved. CSM uses a recursive application of behavior-based error detection: the processor's behavior is observed by a simple monitor, whose behavior is observed by a simpler decoder (which can be totally self-checking).

The processor's memory bus must be wide enough so that the encoded SEC/DED or parity bits can be written and read along with the instruction. With this provision, bus errors are detectable (or correctable) along with errors from other sources. CSM's encoded parity or SEC/DED bits add to secondary (disk) memory overhead because they must be stored along with the instructions and the vertical signatures. However, certain secondary-memory errors that are otherwise undetected, e.g., loading an incorrect program page, are detected within a short latency by CSM.

VI. Comparison with Existing Techniques

This section quantitatively compares the effectiveness of the basic technique, PSA [18], *signatured instruction streams* (SIS) [23], and CSM. For this comparison, a program is assumed to be compiled from a structured high-level language (HLL). Typical HLL control-flow constructs are: IF, IF-ELSE, SWITCH, FOR, WHILE, DO, CALL, and RETURN. Each control-flow construct creates a fixed number of blocks, e.g., Fig. 2 is an instance of IF-ELSE, which creates 3 blocks. Table I lists the total vertical overhead and the number of reference signatures added by each technique for each control-flow construct. (SWITCH is assumed to have four cases.) For CSM, k is the average number of CALLs to each subroutine. Table II lists relative control-flow construct usage from [2], which is used for the numerical comparisons.

TABLE I
VERTICAL OVERHEAD AND (REFERENCE SIGNATURES).

	Basic	PSA	SIS	CSM
IF	2 (2)	1 (0)	2 (1)	1 (0)
IF-ELSE	3 (3)	1 (0)	2 (1)	1 (0)
SWITCH	7 (7)	3 (0)	2 (1)	3 (0)
FOR	3 (3)	2 (1)	2 (1)	1 (0)
WHILE	3 (3)	2 (1)	2 (1)	1 (0)
DO	2 (2)	2 (1)	2 (1)	1 (0)
CALL	1 (1)	1 (1)	0 (0)	1-1/k (0)
RETURN	1 (1)	1 (1)	1 (1)	0 (0)

TABLE II
RELATIVE CONTROL-FLOW CONSTRUCT USAGE.

IF	IF-ELSE	SWITCH	FOR	WHILE	DO	CALL	RETURN
0.23	0.14	0.02	0.08	0.05	0.00	0.38	0.10

TABLE III
ESTIMATED MEMORY OVERHEAD.

	Basic	PSA	SIS	CSM
Vertical	10-25%	6-15%	6-15%	4-11%
Horizontal	0%	6%	0%	0%

TABLE IV
COMPARISON RESULTS SUMMARY.

	Basic	PSA	SIS	CSM
Total Memory Overhead	10-25%	12-21%	6-15%	4-11%
Latency in PM Cycles	2-5	7-17	7-17	0.016-1.0
Control-Flow Error Coverage	96-99%	99.5-99.9%	85-93%	99.9999%
Control-Bit Error Coverage	99.9999%	100%	85-93%	99.9999%

6.1. *Memory-Overhead*

The fraction of vertical overhead added by each technique can be estimated using these data. The matrix product of Table II and each overhead column of Table I is each technique's weighted-average overhead per control-flow construct. This result for the basic technique times the average words per block (4–10) is the weighted-average words per control-flow construct. This result divided into each technique's weighted-average overhead per control-flow construct is the fraction of vertical overhead, shown in the first row of Table III. For CSM, each subroutine is assumed to have one RETURN, so that k is estimated as CALLs/RETURNs. Table III's second row lists the horizontal overhead added by each technique, as-

suming a 32-b word. Only PSA adds horizontal overhead, two columns to indicate reference signatures and the end of a path. The first row of Table IV lists each technique's estimated total overhead. CSM is seen to reduce total memory overhead by as much as 3 times.

6.2. Error-Detection Latency

In Section V detection latency is estimated to be 2 to 5 program memory cycles for the basic technique, and 1 or 0.016 cycles for CSM, using parity or SEC/DED protected memory, respectively. Detection latencies for PSA and SIS can also be estimated. The operation performed above on Table I's overhead columns is performed on the reference-signature columns to produce the fraction of reference signatures: 3–8% for PSA and for SIS. This is one reference signature for each 14–35 words, including vertical-overhead words. Thus the average detection latency for PSA and SIS is 7–17 cycles. The second row of Table IV lists each technique's detection latency. CSM decreases detection latency by as much as 1000 times.

6.3. Control-Flow Error Coverage

In Section II-2.2 control-flow error coverage is estimated to be 96–99% for the basic technique. Control-flow error coverage can also be estimated for the other techniques. From the above result, PSA's path set size p is 14–35 words. From (3), PSA's control-flow error coverage is 99.5–99.9%. For SIS, a path starts after each CALL and after each reference signature that does not follow a RETURN, thus p is 10–24 words. Various studies show that conditional branches are taken roughly 65% of the time [4]. For this estimate and Table II, the fraction of all branches that are taken t is 84%. The fraction of blocks b that end with a reference signature is 0.33. From (4), SIS's control-flow error coverage is 85–93%. CSM's randomly distributed intermediate signatures provide control-flow error coverage of $1 - 2^{-v}$, which is greater than 99.9999% for a 32-b processor. Table IV's third row lists each technique's control-flow error detection coverage. CSM reduces undetected control-flow errors by orders of magnitude.

These coverage results pertain to single (transient) control-flow errors that land in the program space. Other types of control-flow errors must also be considered.

Stuck-at PC/False Loops. Existing techniques do not detect a processor's program counter (PC) stuck at an address unless that address contains a reference signature, nor do they detect error-created loops that contain no reference signature. The stuck PC has been identified as an error that must be detected in safety critical systems [32]. Sosnowski [27] showed that a control-flow error creates a *false loop* with probability as high as 0.1. Existing techniques must use a watchdog timer [26] to detect a stuck PC and false loops. A watchdog timer increases monitor complexity and adds memory overhead for timer-reset commands, which also decreases processor performance. CSM detects false loops without augmentation because a horizontal reference-signature bit(s) is encoded at each program location.

Program-Bounds Violations. Existing techniques do not detect control-flow errors that cause instruction execution from the data space, a program-bounds violation. These techniques must be augmented with bounds checking hardware [23]. CSM detects program-bounds violations without augmentation. Following a control-flow error that lands in the data space, the monitor unhashes the data's check bits (which should not be unhashed) and with probability $1 - 2^{-h}$ creates an error that is detected by the decoder.

Stuck-Incrementing PC. Existing techniques do not detect control-flow errors caused by a stuck-incrementing PC. For this fault, program execution always increments from the current path's end to the contiguous path. A control-flow error occurs if these paths are not connected in the program graph. Existing techniques do not detect this error because all paths begin with the same initial intermediate signature. CSM detects this error because the intermediate signature of the contiguous path is not correlated with the initial intermediate signature of the succeeding path in the program graph.

6.4. Control-Bit Error Coverage

Each technique's control-bit error coverage can be estimated assuming a fault that causes a single word to become a random value (e.g., a memory-bus timing fault). PSA detects 100% of these errors because it employs a signature with size equal to the word width w. The basic technique and CSM detect these errors with probability $1 - 2^{-v}$ because their signatures are of width v and $v < w$. For SIS, a control-bit error is equivalent to entering the "possibly detectable" state of Fig. 3. SIS's control-bit error coverage can be estimated by deleting the p^{-2} terms from (4) and using the values determined above for b, p, and t in the resulting equation. The result is control-bit coverage of 85–93%. The last row of Table IV lists each techique's control-bit coverage. CSM is seen to have control-bit error coverage that is much higher than SIS, equal to the basic tecnnique, and essentially equal to PSA.

6.5. Reduced Signature Costs

For architectures that use delayed branching, delayed loads, and/or resolve pipeline interlocks using NOPs, the cost reduction algorithm presented in Section IV-4.2 reduces CSM's memory overhead and performance loss by placing justifying signatures at locations with NOPs, at locations with duplicate instructions, or at locations with low execution frequency. The fraction of justifying signatures used by CSM (4–11%) is significantly less than the 16–18% NOPs contained in programs for the MIPS-X architecture. Thus CSM's memory overhead and performance loss can be largely eliminated for MIPS-X or related architectures. Similar cost reduction is not possible for existing techniques because they use reference signatures (3–8% or more) that must be placed at fixed

locations that will generally not correspond to NOPS, and because they have more overhead to eliminate. CSM can be shown to have similar cost reduction advantages for other architectures.

VII. Summary and Future Work

This paper presents continuous signature monitoring (CSM), a new signature-monitoring approach that makes major advances beyond existing techniques in each signature-monitoring property. A new analytical method shows that control-flow error detection coverage is determined by the distribution of intermediate signatures. CSM reduces undetected control-flow errors by orders of magnitude because its intermediate signatures are randomly distributed. A theorem is presented that sets a lower bound on the number of signatures that must be added to a program for a random intermediate-signature distribution. A new signaturing technique is presented that achieves this bound by partitioning a program into the minimum number of paths, and then adding one justifying-signature instruction to each path. A new analytical method is presented that allows techniques to be compared through usage-statistics of HLL control-flow constructs. Using this method, CSM is shown to reduce the number of signatures by as much as 3 times. This paper is the first to recognize that a signature's cost, which is a function of memory overhead and performance loss, varies among program locations. A cost-reduction algorithm is presented that can significantly reduce total signature cost by partitioning a program graph into a low-cost maximal-path set. For architectures that require NOPS in unfilled delay slots, or require NOPS to resolve pipeline interlocks, total signature cost can potentially approach zero.

Error-detection latency is dramatically reduced by using horizontal and vertical signatures. Techniques are presented that combine a horizontal-signature code with the program memory's SEC/DED or parity code. The resulting *product codes* [21] correct/detect memory errors, detect signature errors within a short latency, and do not increase horizontal memory overhead. Error-detection latency is shown to be reduced by orders of magnitude. For processors that use pipelines, the short latency allows recovery from transient faults without the added complexity of rollback/recovery buffers. Complexity is also low because the SEC/DED or parity decoder can detect transient monitor errors, eliminating the need to duplicate the monitor. Because CSM can detect a stuck-at PC and false loops, the complexity of a watchdog timer and the cost of its timer-reset instructions are eliminated.

Future research will focus on two areas. First, the CSM monitor can also be used for at-speed nonconcurrent BIST in the factory or in the field to detect intermittent or permanent faults. Methods will be explored for generating signatured test programs (i.e., test vectors) so that the CSM monitor can be used to provide this capability with high coverage. Used in this manner, the CSM monitor may possibly replace some of the BIST hardware currently added to processors. Second, CSM is effective at detecting errors in deterministic control sequences, but generally cannot detect data errors or errors in nondeterministic control sequences. Fault-injection experiments show that these later errors account for roughly 20% of all processor errors [22], [23]. Low-cost techniques for detecting these errors will be investigated. The ultimate goal is to combine one or more of these techniques with CSM to produce a composite low-cost technique that provides high coverage and concurrent-detection of all processor errors, and provides high tolerance of all transient processor faults.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1985.

[2] W. G. Alexander and D. Wortman, "Static and dynamic characteristics of XPL programs," *Comput.*, vol. 8, pp. 41–46, Nov. 1975.

[3] P. Chow and M. Horowitz, "Architectural tradeoffs in the design of MIPS-X," in *Proc. 14th IEEE Comp. Arch.*, 1987, pp. 300–308.

[4] J. DeRosa and H. Levy, "An evaluation of branch architectures," in *Proc. IEEE 14th Comp. Arch.*, 1987, pp. 10–16.

[5] J. Eifert and J. Shen, "Processor monitoring using asynchronous signatured instruction streams," in *Proc. 14th IEEE FTCS*, 1984, pp. 394–399.

[6] R. Frohwerk, "Signature analysis: a new digital field service method," *Hewlett Packard J.*, vol. 5, pp. 2–8, May 1977.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractibility—A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[8] M. Halbert and S. Bose, "Design approach for a VLSI self-checking MIL-STD-1750A microprocessor," in *Proc. 14th IEEE FTCS*, 1984, pp. 254–259.

[9] M. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM J. Res. Develop.*, vol. 14, pp. 395–401, July 1970.

[10] K. Iwasaki, "Analysis and proposal of signature circuits for LSI testing," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 84–90, Jan. 1988.

[11] J. Joseph and A. Avizienis, "A fault tolerance approach to computer viruses," in *Proc. IEEE Symp. on Security and Privacy*, 1988, pp. 52–58.

[12] M. Kobayashi, "Dynamic profile of instruction sequences for the IBM system/370," *IEEE Trans. Comput.*, vol. C-32, pp. 859–861, Sept. 1983.

[13] S. Lin, *An Introduction to Error-Correcting Codes*. Englewood Cliffs, NJ: Prentice-Hall, 1970.

[14] D. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, pp. 681–685, July 1982.

[15] A. Mahmood and E. McCluskey, "Concurrent fault detection using a watchdog processor and assertions," in *Proc. 13th IEEE ITC*, 1983, pp. 622–628.

[16] A. Mahmood and E. McCluskey, "Watchdog processors: error coverage and overhead," in *Proc. 15th IEEE FTCS*, 1985, pp. 214–219.

[17] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors—A survey," *IEEE Trans. Comput.*, vol. 37, pp. 160–174, Feb. 1988.

[18] M. Namjoo, "Techniques for testing of VLSI processor operation," in *Proc. 12th IEEE ITC*, 1982, pp. 461–468.

[19] M. Namjoo, "Cerberus-16: An architecture for a general purpose watchdog processor," in *Proc. 13th IEEE FTCS*, 1983, pp. 216–219.

[20] K. Oikonomou and R. Kain, "Abstractions for node level passive fault detection in distributed systems," *IEEE Trans. Comput.*, vol. C-32, pp. 543–550, June 1983.

[21] W. Peterson and E. Weldon, Jr., *Error-Correcting Codes*. Cambridge, MA: MIT Press, 1972.

[22] M. Schmid, R. Trapp, A. Davidoff, and G. Masson, "Upset exposure by means of abstraction verification," in *Proc. 12th IEEE FTCS*, 1982, pp. 237–244.

[23] M. Schuette and J. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Trans. Comput.*, vol. C-36, pp. 264–276, March 1987.

[24] J. Shen and M. Schuette, "On-line self-monitoring using signatured instruction streams," in *Proc. 13th IEEE ITC*, 1983, pp. 275–282.

[25] J. Shen and S. Tomas, "A roving monitoring processor for detection of control flow errors in multiple processor systems," *Microprocessing and Microprogramming*, vol. 20, pp. 249–269, May 1987.

[26] D. Siewiorek and R. Swarz, *The Theory and Practice of Reliable System Design*. Digital Press, 1982.
[27] J. Sosnowski, "Evaluation of transient hazards in microprocessor controllers," in *Proc. 16th IEEE FTCS*, 1986, pp. 364-369.
[28] J. Sosnowski, "Detection of control flow errors using signature and checking instructions," in *Proc. 18th IEEE ITC*, 1988, pp. 81-88.
[29] T. Sridhar and S. Thatte, "Concurrent checking of program flow in VLSI processors," in *Proc. 12th IEEE ITC*, 1982, pp. 191-199.
[30] H. Stone, *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1987.
[31] C. Tung and J. Robinson, "On concurrently testable microprogrammed control units," in *Proc. 16th IEEE ITC*, 1986, pp. 895-900.
[32] D. Turner, R. Burns, and H. Hecht, "Designing micro-based systems for fail-safe travel," *IEEE Spectrum*, vol. 24, pp. 58-63, Feb. 1987.
[33] K. Wilken and J. Shen, "Embedded signature monitoring: analysis and technique," in *Proc. 17th IEEE ITC*, 1987, pp. 324-333.
[34] ——, "Continuous signature monitoring: efficient concurrent-detection of processor control errors," in *Proc. 18th IEEE ITC*, 1988, pp. 914-925.
[35] K. Wilken and J. Shen, "Concurrent error detection using signature monitoring and encryption," in *Proc. Int. 1st Working Conference on Dependable Computers in Critical Applications*. New York: Springer-Verlag, to be published.
[36] T. Williams, W. Daehn, M. Gruetzner, and C. Starke, "Bounds and analysis of aliasing errors in linear feedback shift registers," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 75-83, Jan. 1988.

*

Kent Wilken (M'84) received the B.S. and M.S. degrees in electrical engineering (with distinction) from Stanford University, Stanford, CA, in 1977. Since 1986 he has been a Ph.D. candidate in the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA.

From 1977 to 1985 he was with the Hewlett Packard Company where he was involved in disk controller architecture, mass storage performance, I/O architecture, and fault-tolerant computing. In 1987 he served as Local Arrangements Chair for the 17th Fault-Tolerant Computing Symposium. His research interests include computer architecture, fault-tolerant computing, mass storage systems, and computer security.

*

John Paul Shen (M'84) received the B.S. degree from the University of Michigan, Ann Arbor, MI, in 1973, and the M.S. and Ph.D. degrees from the University of Southern California, Los Angeles, CA, in 1975 and 1981, respectively, all in electrical engineering.

From 1973 to 1975, he was with the Hughes Aircraft Company, designing fault detection/isolation and built-in-test circuits for avionic systems. In 1977, he was with TRW Inc. and was involved in the preliminary design of a local area computer network. From 1977 to 1981, he performed research on multicomputer interconnection networks in the Department of Electrical Engineering, University of Southern California. In 1981 he joined the Department of Electrical and Computer Engineering of Carnegie Mellon University, Pittsburgh, PA, where he is currently a Professor and Director of the Research Center for Dependable Systems. He has consulted for the IBM Federal Systems Division, General Electric, and the Aerospace Corporation. During the '88-'89 academic year, he was on sabbatical leave at Stanford University and ESL-TRW. His research interests include concurrent error monitoring in fault tolerant systems and high-performance application-specific processor design.

Dr. Shen is a member of IEEE, ACM, Tau Beta Pi, Eta Kappa Nu, Sigma Xi, and a recipient of the NSF Presidential Young Investigator Award.

Detecting Processor Hardware Errors and Computer Viruses Using Program Encryption and Signature Monitoring

Kent Wilken and John Paul Shen

Center for Dependable Systems
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213 U.S.A.

Telephone: [Wilken] (412) 268-6639
INTERNET: wilken@ece.cmu.edu

Abstract -- *This paper presents a low cost approach to concurrent detection of program execution errors that combines program encryption with signature monitoring. Sources of detectable errors include permanent and transient hardware faults, and computer viruses. Errors are detected using a hardware monitor and signatures the compiler embeds into the program. Using intermediate signatures, a program is encrypted by the signature compiler and decrypted during execution in real time by the monitor. The new approach is more efficient than existing signature-monitoring techniques because it significantly reduces error-detection latency and the number of embedded signatures. Computer virus attacks are made difficult because a virus must deduce a program's key and the monitor's cryptographic signature function to attach compatibly encrypted code. Including the processor's internal control signals in the signature increases coverage of errors caused by hardware faults, and also significantly increases virus resistance because a successful attack must deduce details of the processor's implementation.*

Key Words:
Fault Tolerance and Security
Concurrent Error Detection
Computer-Virus Detection
Program Encryption
Signature Monitoring
Signature Analysis
Control-Flow Checking
Cache Memory

1. Introduction

Complete computer dependability requires detection of errors from all sources. Since the earliest computers, much attention has been focused on detecting errors caused by hardware faults. As system complexity increased, detection of errors caused by software and hardware design faults became important. Although faults are often assumed to be inadvertent, deliberate faults (e.g. computer viruses) cause errors that must be detected. The potential for deliberate faults is becoming greater as computer use and computer communication increases.

This paper proposes a low cost behavior-based approach to detecting errors caused by certain hardware, design, and deliberate faults. In the behavior-based approach, a program's behavior is abstracted and the abstraction is monitored for run-time violations. No fault model is assumed, any fault (hardware, design, or deliberate) that causes incorrect program behavior is potentially detectable. To be efficient, the selected abstraction must provide high error-detection coverage at a low cost. Experiments using various program abstractions find that the program-control-flow abstraction offers the most error detection potential [7, 14]. Researchers [9, 11, 12, 13, 15, 16, 18, 21] have proposed techniques that monitor program control flow using signatured programs and a hardware monitor, a general approach we call *signature monitoring*. This paper proposes a new approach to signature monitoring that significantly increases its efficiency and effectiveness.

1.1 Signature Monitoring

To provide efficient error detection, signature monitoring exploits a common program redundancy: few instructions alter control flow. This redundancy allows program segments containing many instructions to be coded and later checked, as a unit. Figure 1 shows the basic signature-monitoring technique. The signature compiler divides the code into *basic blocks* [1], computes each block's *reference signature*, and then embeds a *signature instruction* at the block's end. The signature instruction has a field that contains an identifying opcode, and a field that contains the reference signature. The opcode could be a coprocessor opcode already included in the processor's instruction set, or it could be a specific addition to the instruction set.

During execution, the monitor observes the executed instructions and generates each block's run-time signature using dedicated hardware. At each signature instruction, the processor executes a NOP while the monitor compares the run-time and reference signatures, declaring an error if they differ. The sig-

nature function V is typically a cyclic-redundancy check (CRC) polynomial so that the run-time signature can be generated using a parallel-input linear feedback shift-register (PLFSR). Detectable errors include *control-flow errors* and *control-bit errors*. A control-flow error occurs when the instruction execution sequence is incorrect. Control-bit errors result when instructions are executed in the correct order but one or more of the signatured control bits is incorrect. The signature can include control bits from assembly code, microcode, and hardware control lines.

Recently researchers have proposed approaches that reduce the latency for detecting signature errors. A short error detection latency is important because it can allow low cost recovery from errors caused by transient hardware faults [21]. Wilken and Shen [21] propose checking one or more *horizontal signature* [21] bits at each program word, thereby providing order of magnitude reduction in error detection latency. Memory overhead does not increase because the horizontal signatures can be combined with the program memory's parity or single-error-correcting/double-error-detecting (SEC/DED) code [21]. However, horizontal signatures cause the instruction width to increase, which causes incompatibility with existing system architectures throughout the memory hierarchy [13]. Saxena and McCluskey [13] propose reducing error detection latency by using extended precision checksums as signatures and by exploiting the naturally occurring opcode redundancy. The extended-precision-checksum approach has the advantage that it does not increase instruction width, however latency reduction is small compared with that provided by horizontal signatures, and significant reduction is limited to faults causing opcode-field errors in multiple instructions. In Section 2 a new approach is proposed that uses program encryption to provide latency reduction comparable to that provided by horizontal signatures, without increasing instruction width.

Signature monitoring techniques have been proposed that reduce the number of embedded signatures by allowing a *path* containing more than one basic block to be encoded into each signature [11, 15, 21]. The technique proposed by Wilken and Shen [21], *Continuous Signature Monitoring* (CSM), was shown to reduce the number of signatures to a theoretical lower bound by partitioning the program into the minimum number of paths. This result is based on the assumptions that the monitor contains no memory other than a register for accumulating the run-time signature, and that control-flow error detection coverage is $1\text{-}2^{-v}$ for a v-bit signature [21]. In Section 3 these assumptions are relaxed and a technique is proposed that uses program encryption and a small signature cache in the monitor to reduce the number of embedded signatures compared with CSM without substantially reducing coverage. This technique significantly reduces performance overhead because the signature that was previously re-

quired inside each loop can be eliminated. In Section 4 the CSM coverage assumption is also relaxed and a technique is proposed that further reduces the number of embedded signatures without substantial coverage reduction by using program encryption to eliminate the signatures previously required for subroutine calls.

The MIPS RISC [10] is used in this paper as an example architecture for illustrating the effectiveness of the proposed techniques. However, these new techniques are general and are applicable to other architectures.

1.2 Computer Virus Detection

Recently, the computing community has experienced numerous computer virus attacks [5]. Cohen [3] showed that computer viruses can be created with modest skill and effort, can spread rapidly, and pose a significant security threat. As computers proliferate, the number and severity of computer virus attacks is likely to increase. Effective and efficient virus-detection techniques are needed.

Various techniques have been proposed that use cryptographic checksums to ensure program integrity when a program is loaded, e.g. [4]. Although effective, these techniques do not ensure the integrity of an executing process. For example, these techniques cannot detect (and preclude) process alterations such as that caused by the Internet Worm's "fingerd" attack [17].

Joseph and Avizienis [9] propose extending signature monitoring to include concurrent virus detection. Signature monitoring can concurrently detect a virus, unless the virus is properly signatured. Proper signaturing of a virus may be easy for earlier techniques because they use a single signature function that the attacker might know or easily deduce [9]. Joseph and Avizienis propose using multiple signature functions, one of which is randomly selected for each program by the signature compiler. Using a technique proposed by Namjoo [12], the signature compiler links the program's signatures to form a graph that the monitor traverses and checks during program execution. Joseph and Avizienis propose that the graph and a vector that represents the signature function are encrypted by the signature compiler, and are decrypted by the monitor when the program is loaded. The *plaintext* [2] is stored in the monitor's local memory, which is not readable externally. Attacks are averted because a virus cannot easily attach segments to the program that conform to the encrypted signature graph, or easily alter the program and the encrypted signature graph in a consistent way, without detection.

Although innovative, the encrypted signature-graph approach has significant limitations. The decryption overhead precludes this approach if process context switches are frequent [9]. For systems that use virtual memory, the monitor's memory is large because it must contain the entire signature graph, even though only a fraction of the program may reside in the processor's real memory. Moreover, the signature graph is large because it contains the signatures plus the links that form the graph's arcs. For processors that use an on-chip cache, the monitor must be located on-chip to observe the program's behavior [20]. However the memory required by the monitor is generally much too large to reside on chip. For the encrypted signature graph approach, an on-chip monitor requires separate address and data buses for accessing the signature graph, which is a significant cost.

In Section 5, a new approach to concurrent virus detection is proposed that uses signatures embedded in an encrypted program. This approach provides significant resistance to virus attacks, and avoids the limitations of the encrypted signature-graph approach. Section 6 summarizes the paper's contributions and discusses future work.

2. Basic Program Encryption

This section introduces the basic approach to combining program encryption with signature monitoring. The basic approach is shown to provide reductions in error detection latency comparable to that provided by horizontal signatures, without causing significant incompatibility with existing system architectures.

2.1 Instruction Hashing

Figure 2a shows a path that is signatured using the conventional approach. A path's signature is the result of a series of intermediate calculations performed at each word in the path:

$$S_k = V(W_{k-1}, S_{k-1}) \tag{1}$$

where k ranges from 1 to j, V is the signature function, W_{k-1} is the word at location k-1, and S_0 is a specified initial value [21]. Location k is associated with *intermediate signature* [21] S_k, which for $k>0$ is the signature of the sub-path [0, k-1]. The last intermediate signature, S_j, is the path's signature.

The conventional signature monitoring approach only alters the assembly code by embedding signatures. The proposed approach, illustrated in Figure 2b, embeds signatures as before, but also encrypts

each program word using its location's intermediate signature as the key. Figure 2b shows a path encrypted by the signature compiler using an efficient encryption function, the exclusive-or (XOR) operator. During execution, the monitor generates each run-time intermediate signature, decrypts the word in real time, and delivers the result to the processor for execution.

Schuette and Shen [15] proposed a technique called *branch address hashing* (BAH) that can eliminate reference signatures that follow branch instructions. The address word of each two-word branch instruction is replaced by the address word hashed (XORed) with its intermediate signature. Following a signature error, the unhashed branch address becomes a pseudo-random value, and a jump is taken to an arbitrary location, where the error may be detected. The basic encryption approach proposed here can be viewed as a generalization of BAH in which all instruction words are hashed, not just branch addresses. Thus, the new approach is termed *instruction hashing*. The remainder of this paper describes how instruction hashing is used to significantly improve signature monitoring's efficiency and effectiveness.

2.2 Short Error-Detection Latency

Computer systems generally contain hardware and software mechanisms for detecting such errors as illegal opcodes, address or capability violations, alignment errors, illegal system calls, etc. For signature monitoring, an error that produces an incorrect signature causes the following run-time intermediate signatures to be pseudo-random. With instruction hashing, the instructions unhashed using these intermediate signatures are also pseudo-random. Executing pseudo-random instructions will trigger the system's built-in error detection mechanisms, resulting in reduced error-detection latency. Additionally, these mechanisms can detect monitor errors because a monitor error will cause the runtime intermediate signatures to be incorrect, causing pseudo-random instructions to be produced.

To illustrate the magnitude of the short error-detection that latency instruction hashing provides, experiments were performed using a MIPS RISC processor. The object code of a small user program (3K words) was repeatedly altered by replacing a valid sequence of instructions with a pseudo-random sequence. Each of 100,000 program alterations was executed. All executed random instructions were recorded and categorized by primary instruction opcode, as well as all instructions that produced detected errors. The results of these experiments are summarized in Figure 3, which shows a bar for each primary opcode indicating the probability that an error is detected when a random instruction with

that opcode is executed. The average detection probability for all opcodes is 0.81, which corresponds to an error detection latency of (1 - 0.81)/0.81 = 0.23 cycles. For the MIPS RISC architecture, instruction hashing's detection latency is less than the 1.0 cycle detection latency for a 1-bit horizontal signature [21], and is comparable to the 0.25 cycle detection latency that would be provided by a 2-bit horizontal signature. A principal advantage of using instruction hashing to reduce error detection latency vs. horizontal signatures is that the instruction width is unchanged and thus greater compatibility is maintained with existing system architectures.

Examination of the types of detected errors in the above experiments shows that the primary detection mechanisms are the illegal opcode, address bounds violation, coprocessor unusable, and data alignment error detectors. Of the 64 primary MIPS RISC opcodes, 24 are illegal (reserved) and always generate an illegal opcode error, accounting for 46% of all detected errors. Nine opcodes always generate a coprocessor unusable error because the corresponding coprocessor is not implemented (coprocessor 2, coprocessor 3) or is not accessible in user mode (coprocessor 0), accounting for 17% of all detected errors. Address bounds violations to locations in the kernel were 13% of all detected errors, and bounds violations to unused locations in the user space represented 12% of all detected errors. Alignment errors were 9% of the total. The remaining 3% were due mainly to illegal (reserved) secondary opcodes and arithmetic exceptions, both floating point and integer.

Detection probability decreases for larger user programs because there are fewer address bounds violations to unused locations in the user space. However the decrease is small even for the largest programs because such errors only account for 12% of the total. Detection probability is less for kernel programs because legal access can be made to used locations in both the kernel and user spaces, in an extreme case reducing the fraction of detected errors by 25%. Also fewer errors are detected in kernel mode because coprocessor 0 is accessible, although at least 2/3 of all random coprocessor-0 instructions are detected because of alignment errors or illegal secondary opcodes. The experiments overstate the average error detection probability by roughly 0.01 because the monitor would be implemented as coprocessor 2 or 3, and a small fraction of the corresponding instructions would no longer generate errors when executed randomly.

Instruction hashing's short detection latency is not unique to the MIPS RISC processor. Short detection latencies were also observed for random instruction execution on a VAX [6] processor.

2.3 Instruction Hashing's Cost

Sections 3, 4, and 5 detail other significant benefits allowed by instruction hashing, including major reductions in the memory and performance overhead caused by embedded signatures, and significant resistance to computer virus attacks. In this subsection instruction hashing's costs are presented.

Instruction hashing's primary cost is the small delay added to each instruction fetch by the XOR gate that is needed for unhashing instructions, as illustrated in Figure 4. The magnitude of this delay can be determined from the ratio of the processor's cycle time to the nominal gate-delay time. For example, the 16.7 MHz MIPS R2000 processor uses a 2 micron CMOS process [8], which has a nominal 1.2 nanosecond gate delay. Thus the R2000's cycle time equals 50 gate-delay times. This suggests that an XOR gate, which has two primitive gate delays, has a delay time that is 4% of the cycle time. Assuming the instruction fetch is a critical path, the instruction-cache RAM must be faster by two gate-delay times (e.g. by 2.4 nanoseconds for the R2000) to avoid reducing processor performance. This faster-RAM requirement will generally increase a computer's cost. A secondary cost for using instruction hashing is the slight increase in the monitor's size caused by the unhashing circuit.

Instruction hashing also causes a decrease in error detection coverage. Following a signature error, instruction hashing can induce a control-flow error by creating an instruction that alters control flow, or by creating an instruction that does not alter control flow where an unconditional control-flow-altering instruction normally exists. An upper bound on this coverage decrease can be determined. Let d be the probability that an error is detected in each cycle, and f be the fraction of undetected errors that are induced control-flow errors. The probability that an induced control-flow error cancels the original signature error is 2^{-v}. Thus, the expected reduction in coverage is:

$$[(1-d) f 2^{-v}] \; / \; [d + [(1-d) f 2^{-v}]]$$

For the above experiments d is 0.81. The fraction f is at most 1. For the MIPS RISC architecture, the 25-bit cofunction field of the coprocessor instruction COPz [10] can be used as the signature field. For these data, the reduction in coverage caused by instruction hashing is at most 0.23×2^{-v}, or 7×10^{-9}, which is negligible.

3. Signature Caching

In [21], the CSM signaturing technique is shown to use the theoretical minimum number of signatures, assuming that the monitor contains no memory except the run-time signature accumulator, and that control-flow error coverage is $1-2^{-v}$ for a v-bit signature [21]. Here these assumptions are relaxed and a new technique is proposed that uses instruction hashing and a small signature cache to reduce the number of signatures below that required by CSM, without substantially reducing control-flow error coverage.

3.1 Previous Signature-Reduction Techniques

This subsection reviews previous signature-reduction techniques as background for the signature-caching technique proposed in the next subsection. Namjoo [11] proposed *Path Signature Analysis* (PSA), a technique that uses *justifying signatures* to reduce the number of embedded signatures. A justifying signature is a word embedded in a path that sets (justifies) the path's signature to a particular value. In Figure 5, a simple program is represented by a program graph, a directed graph that represents each basic block by a node and each possible transition between basic blocks by an arc. PSA constructs sets of paths that cover all legal sequences of nodes in the graph. All paths in a set start at the same node, and share a common reference signature that is embedded at the beginning of the starting node. PSA adds justifying signatures to selected nodes so that all paths in a set produce the same signature. The path sets for Figure 5 are {ABD, ABC} and {BD, BC}. Reference signatures are embedded at the beginning of nodes A and B, and a justifying signature is embedded in node C or D so that these nodes (and hence the paths in the two path sets) produce the same signature. Compared with the basic technique, PSA reduces the number of signatures for this program from 4 to 3, and reduces the number of signatures for substantial programs by approximately 40% [21].

The CSM technique [21] further reduces the number of signatures by using justifying signatures to create a program that produces the same signature for any path from entry to exit. CSM could be viewed as a generalization of PSA that uses a single path set and a fixed-value reference signature (e.g. 0) to cover an entire program. CSM partitions a program graph into disjoint *maximal paths* [21]. One maximal path begins at the program's entry node, and a maximal path begins at one of the outgoing arcs of each conditional-branch node. A justifying signature is embedded into a maximal path that ends by merging with another maximal path or itself, such that the path's signature equals the merge location's intermediate signature. A justifying signature is embedded into a maximal path that ends at a

program exit such that the path's signature equals the fixed-value reference signature, e.g. 0. Horizontal signatures were proposed as the means for detecting errors. Thus, CSM requires only two justifying signatures for the program graph in Figure 5: one embedded in node B or node D of maximal path ABD, and the other embedded in node C of maximal path C. Compared with PSA, CSM reduces the number of signatures for this program from 3 to 2, and reduces the number of signatures for substantial programs by approximately 30% [21].

3.2 Signature Caching Technique

A new technique is proposed that uses instruction hashing and a small signature cache in the monitor to reduce the number of signatures below that required by CSM. Within each program loop, CSM must embed a justifying signature to ensure that the intermediate signature is correct when the program returns to the loop's first location. However, the monitor previously calculated (and then discarded) this intermediate signature. For the proposed technique, illustrated in Figure 6, the signature compiler removes the loop's justifying signature. During execution, the monitor stores each newly-calculated intermediate signature and the corresponding address in a small cache. For simplicity, a direct-mapped cache [19] can be used. When a branch with a negative displacement is taken, the monitor compares the target instruction's address with the addresses in its cache, and copies the corresponding intermediate signature into its signature register if a match occurs. Given the size of the cache, the signature compiler can remove the justifying signature from each loop that is smaller than or equal to the size of the cache. For the program in Figure 5, if the size of node B plus the size of node D is less than or equal to the cache size, the justifying signature from maximal path ABD can be eliminated, leaving only the justifying signature in maximal path C. Thus, compared with CSM, the new technique can reduce the number of signatures for this program from 2 to 1.

To determine the reduction that signature caching allows for substantial programs, data were gathered for five *C* programs compiled for the MIPS RISC architecture: *awk*, *cc*, *csh*, *emacs*, and *latex*. The gathered data includes the (static) fraction of specific instructions and specific instruction types. The combined data for these programs are shown in Table 1. The fraction of signatures that CSM requires for these programs can be determined using these data. CSM requires a justifying signature for each conditional branch, and a justifying signature for all but one of the calls to a subroutine [21]. Assuming that each subroutine has one return, the average fraction of CSM signatures is: Conditional_Branches + Calls - Returns = 13.96%. Relative to CSM, signature caching allows one justifying signature to be

eliminated per Loop End. For these programs, signature caching can reduce the fraction of signatures by as much as 1.69%, to 12.27%, an improvement of 12% compared with CSM. Moreover, the reduction in performance overhead is much larger because signatures within loops, especially small inner loops, cause a disproportionate fraction of the total performance overhead.

The actual fraction of loop signatures that are eliminated depends on both cache size and the distribution of loop sizes. Figure 7 shows the average percentage of loop signatures that can be removed from these five programs for cache sizes that are a power of 2. For these data, a cache of size 256 will eliminate the justifying signatures from 95% of all loops, while a cache of size 32 will eliminate 60%.

Besides reducing the number of signatures and performance overhead, signature caching also allows a significant reduction in error detection latency. If the initial iteration of a loop is error free, and the loop's size is less than or equal to the size of the cache, the cache contains the correct intermediate signature for each of the loop's locations. On the second and each subsequent loop iteration, a cache hit will occur at each loop location and the monitor can compare the current, calculated intermediate signature with the corresponding intermediate signature in the cache. An error is declared if these intermediate signatures differ. Because a large fraction of a program's execution time is spent within loops, a large fraction of transient control errors are detected in this manner, with zero latency.

3.4 Error Detection Coverage

Instruction hashing must accompany signature caching to avoid a large reduction in error detection coverage. Without instruction hashing, any error that occurs within a loop where the justifying signature has been removed, is undetectable if the error is not detected when the loop's end is reached. Such errors are undetectable because the incorrect run-time signature would be overwritten with the correct intermediate signature of the loop's first location, which is copied from the cache. Instruction hashing prevents this occurrence because, following an error, the branch that ends the loop is transformed into a pseudo-random instruction, and the loop is broken.

However, the combination of instruction hashing and signature caching does cause some reduction in error detection coverage. Following an error, if a pseudo-random instruction is a branch to a location contained in the cache, a correct intermediate signature is copied from the cache and the error becomes undetectable. The expected reduction in coverage caused by the signature cache is related to the instruc-

tion set architecture and to the cache size. This reduction can be conservatively estimated for the MIPS RISC architecture. An error is assumed to be either a control error, or a control-flow error with a displacement that is within the 16-offset used by the MIPS RISC branch instructions. For a cache of size c, if a pseudo-random instruction is a taken branch, the error escapes detection with probability $c\ 2^{-16}$. Of the 64 total primary opcodes, the MIPS RISC has 4 primary branch opcodes, and secondary branch opcodes that are equivalent to 1/2 of a primary opcode. Assuming the branch condition is satisfied with probability 1/2, the probability that a pseudo-random instruction is a taken branch is $(4.5/64)/2 \approx 2^{-5}$. Thus the estimated error escape probability is $\approx c\ 2^{-21}$. For the cache sizes suggested by the data in the preceding subsection, i.e. $\leq 2^8$, the coverage reduction caused by signature caching is negligible.

3.5 Monitor Size and Context Switching

The signature cache increases the monitor's size. Each cache entry includes an address tag and a signature field. For a 32-bit processor, the size of the address tag for the direct mapped cache is $\lceil 32-log_2 c \rceil$. For the MIPS RISC architecture, the preceding analysis shows that a signature that is much larger than $\lceil 21 - log_2 c \rceil$ bits will not substantially increase coverage. Based on the data presented in Section 3.3, a 16-bit signature would be an appropriate size. Thus for the MIPS RISC example, each cache entry increases the monitor's size by adding $\lceil 48-log_2 c \rceil$ bits of memory. [Note that unlike a processor cache, a signature cache entry does not require a valid bit because when a program is loaded the cache can be initialized with c valid entries, e.g. the addresses and intermediate signatures of the program's first c locations. Following initialization, cache entries are always valid.]

Signature caching also increases the monitor's size because, unlike existing signature monitoring techniques, the monitor must have access to instruction addresses, and because of the comparison circuitry needed to determine cache hits.

Within loops where the justifying signature has been removed, the cache's contents become part of the process state that must be saved, and later restored, when a context switch occurs. Thus, signature caching increases the time for context switches. However, there is no increase for context switches that are known (scheduled) not to occur inside loops where the justifying signature has been removed, because for these context switches the signature register's contents are the only process state in the monitor that must be saved.

4. Subroutine-Signature Reduction

A new technique is proposed that uses instruction hashing to eliminate the signatures that previous techniques require for subroutine calls. Compared with CSM, the combination of this technique plus signature caching is shown to reduce the number of signatures by 50%, without substantially reducing error detection coverage.

4.1 A New Subroutine-Signaturing Technique

A new technique is proposed that does not require justifying signatures for subroutine calls. For a RISC processor, the CSM signaturing technique places a justifying signature in the delay slot following all but one call to a subroutine [21], e.g. after a Jump and Link (JAL) instruction [10]. The new technique eliminates these justifying signatures as illustrated in Figure 8. The subroutine's first instruction sub_1 is placed in the delay slot following each call to that subroutine. The delay slot's intermediate signature is set to the value of the JAL's address field, which is the address of the subroutine's second instruction, sub_2. Using the CSM signaturing technique, justifying signatures are embedded in the subroutine so that any route through the subroutine produces the same signature S_c [21]. The return location's intermediate signature is set to S_c. Each instruction is hashed with its intermediate signature, which for clarity is not illustrated.

The Jump and Link Register (JALR) instruction is also used for subroutine calls by the MIPS RISC architecture [10]. JALR fetches a subroutine's initial address from a designated register. Assuming that register contents are not visible to the monitor, a separate signaturing method is necessary to accommodate JALR. This is done by modifying the preceding technique for JALR so that the intermediate signature of the JALR delay slot is set to a fixed value, e.g. 0. Thus, a subroutine is called using either JAL or JALR, but not both.

Instruction hashing must be used with this technique to avoid a significant reduction in error detection coverage. Without instruction hashing, any error that is not detected when a subroutine call is reached is undetectable because the incorrect signature would be overwritten with a valid intermediate signature, i.e. the JAL address field or 0 for a JALR. Instruction hashing effectively propagates signature errors until they are detected because with a high probability either the decrypted pseudo-random instruction does not cause the incorrect signature to be overwritten, or it is overwritten with another incorrect value.

Unlike the CSM technique, for the new technique a return location's intermediate signature does not depend on the intermediate signature of the delay slot following the call instruction, and thus a signature link register in the monitor and a signature stack in main memory are not necessary. This can significantly decrease the complexity of the monitor, and reduce performance overhead for nested subroutine calls.

4.2 Signature Reduction

The new technique significantly reduces the number of embedded signatures. As noted in Section 3.3, the fraction of CSM signatures is: Conditional_Branches + Calls - Returns, which averages 13.96% for the five example programs. Relative to CSM, the new subroutine signaturing technique reduces this fraction by (Calls - Returns), which is 5.21% for these programs. This signature reduction corresponds to a significant reduction in performance overhead because useful instructions can be executed in the delay slot following calls, instead of signature instructions. However, memory overhead is not reduced relative to CSM because the number of duplicate subroutine instructions that are added equals the number of justifying signatures that are eliminated.

Including the 1.69% reduction that is allowed by signature caching, the two instruction-hashing based techniques allow the fraction of signatures to be reduced to 7.06% for the five example programs. The MIPS RISC architecture requires that a NOP be placed in a branch or a load delay-slot that the compiler cannot fill with a useful instruction [10]. From Table 1, the fraction of NOPs for these programs is 12.97%, which is 84% greater than the fraction of signatures required using these new techniques. An algorithm is proposed in [21] that partitions a program graph into maximal paths such that a maximum number of paths contain a NOP. If a path contains a NOP, the justifying signature can replace the NOP and no performance or memory overhead is incurred [21]. For MIPS RISC architecture, the proposed techniques potentially eliminate the overhead for justifying signatures because there are significantly more NOPs than justifying signatures, and because the graph can be partitioned so that a maximal path that requires a justifying signature also generally contains a NOP. Thus performance overhead is potentially eliminated, and memory overhead is potentially reduced to the fraction of duplicate subroutine instructions, i.e. (Calls - Returns), which is 5.21% for the five example programs. However, note that for the unsignatured program the compiler may duplicate a subroutine's first instruction to fill the delay slot following a call. Thus the memory overhead (i.e. duplicate subroutine instructions) due to this new signature-monitoring approach is generally less than (Calls - Returns).

4.3 Coverage Analysis

Control-flow error coverage is reduce below $1-2^{-v}$ if intermediate signatures are correlated [21]. The new subroutine signaturing technique causes little reduction in control flow error coverage because few intermediate signatures are correlated. Three potential sources of correlated signatures are: the delay slots following JALs, the delay slots following JALRs, and return locations. The delay slots following JAL calls to the same subroutine have the same intermediate signatures and hence cause intermediate signature correlation. However this correlation does not reduce control-flow error coverage because a fault that causes one of these instructions to be executed instead of another does not cause a program error, because the instructions are the same. These instructions and their intermediate signatures are redundant with respect to control-flow faults. The delay slots following JAL calls to different subroutines necessarily have different intermediate signatures and thus are not correlated.

The intermediate signatures in the delay slots following all JALRs are correlated. A control flow error that causes an instruction from an incorrect JALR delay slot to be fetched is not immediately detectable. However the error is detectable at the next instruction when the control flow effect of the JALR occurs. If the instruction in the incorrect delay slot differs from that in the correct delay slot, the run-time signature at the subroutine's second instruction will be incorrect and the error is detectable. If the instruction in the incorrect delay slot is the same as in the correct delay slot, the instructions are redundant with respect to the control-flow fault, and a program error does not occur.

If the instructions in the JALR delay slots are correlated, the intermediate signatures of the subroutines' following instructions are correlated, which causes reduced control-flow error detection coverage. Using the data in Table 1, this reduction is conservatively estimated by assuming that all JALR delay-slot instructions are identical, that the second instruction of each JALR-called subroutine is different, that there are 5 calls to each subroutine, and that control-flow errors stay within a program's bounds. From Table 1, the probability that a given control-flow error occurs at a JALR delay slot is 0.0009. From Table 1 and the above assumptions, the probability that the control flow error lands at a JALR-called subroutine's second instruction is 0.0002. Thus, the estimated reduction in control-flow error coverage is 2×10^{-7}, which is negligible.

Because return locations for the same subroutine have the same intermediate signature, a control-flow error that lands at one of these locations rather than another is not detected. This coverage reduction can

be conservatively estimated using the data in Table 1 by assuming there are 5 calls to each subroutine and that control-flow errors stay within a program's bounds. From Table 1, the probability that a given control-flow error occurs at a return is 0.0124. From Table 1 and the above assumptions, the probability that the control flow error lands at an incorrect return location for that subroutine is $0.0620/s$, where s is the number of subroutines in the program. Thus, the estimated reduction in control-flow error coverage is $0.0007/s$. The average value of s for the five example programs is 516, implying an estimated coverage reduction of 1.5×10^{-6}, which is also negligible.

References

[1] Aho, A., R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* (Addison-Wesley, 1985).

[2] Beker, H., and F. Piper, *Cipher Systems: The Protection of Communications,* (John Wiley, 1982).

[3] Cohen, F., *Computer Viruses: Theory and Experiments*, pp. 240-263, 7th National Computer Security Conf., , (Sept. 1984).

[4] Cohen, F., A Cryptographic Checksum for Integrity Protection, *Computers & Security 6*, 6 (Dec. 1987), 505-510.

[5] Denning, P., Computer Viruses, *American Scientist 76*, (May-June 1988), 236-238.

[6] Digital Equipment Corp., *VAX Architecture Handbook,* (Digital Press, 1981).

[7] Gunneflo, U., Karlsson, J. & J. Torin, *Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation*, pp. 340-347, Proc. 19th FTCS, IEEE, (1989).

[8] Johnson, Mark G., A Symmetric CMOS NOR Gate for High-Speed Applications, *IEEE Journal of Solid-State Circuits 23*, 5 (October 1988), 1233-1236.

[9] Joseph, M., & A. Avizienis, *A Fault Tolerance Approach to Computer Viruses*, pp. 52-58, Proc. Symp. on Security and Privacy, IEEE, (1988).

[10] Kane, G., *MIPS RISC Architecture,* (Prentice-Hall, 1988).

[11] Namjoo, M., *Techniques for Testing of VLSI Processor Operation*, pp. 461-468, Proc. 12th ITC, IEEE, (1982).

[12] Namjoo, M., *Cerberus-16: An Architecture For a General Purpose Watchdog Processor*, pp. 216-219, Proc. 13th FTCS, IEEE, (1983).

[13] Saxena, N. & E. McCluskey, *Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums*, pp. 428-435, Proc. 19th FTCS, IEEE, (1989).

[14] Schmid, M., R. Trapp, A. Davidoff & G. Masson, *Upset Exposure by Means of Abstraction Verification*, pp. 237-244, Proc. 12th FTCS, IEEE, (1982).

[15] Schuette, M. & J. Shen, Processor Control Flow Monitoring Using Signatured Instruction Streams, *IEEE Transactions on Computers C-36*, 3 (March 1987), 264-276.

[16] Shen, J. & S. Tomas, A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems, *Microprocessing and Microprogramming 20*, 4 & 5 (May 1987), 249-269.

[17] Spafford, E., The Internet Worm: Crisis and Aftermath, *Communications of the ACM 32*, 6 (June 1989), 678-687.

[18] Sridhar, T. & S. Thatte, *Concurrent Checking of Program Flow in VLSI Processors*, pp. 191-199, Proc 12th ITC, IEEE, (1982).

[19] Stone, H., *High-Performance Computer Architecture*, (Addison-Wesley, 1987).

[20] Wilken, K. & J. Shen, *Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors*, pp. 914-925, Proc. 18th ITC, IEEE, (1988).

[21] Wilken, K. & J. Shen, Continuous Signature Monitoring: Low-Cost Concurrent-Detection of Processor Control Errors, *IEEE Transactions on CAD* (June 1990), to be published.

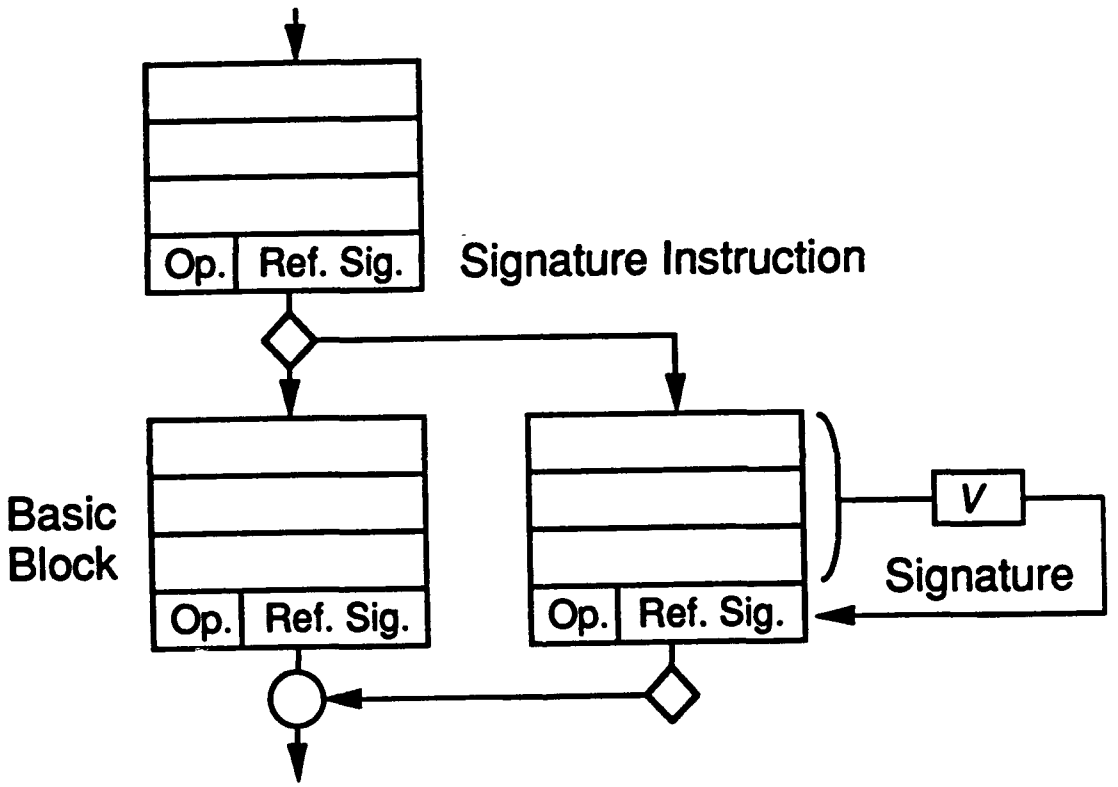
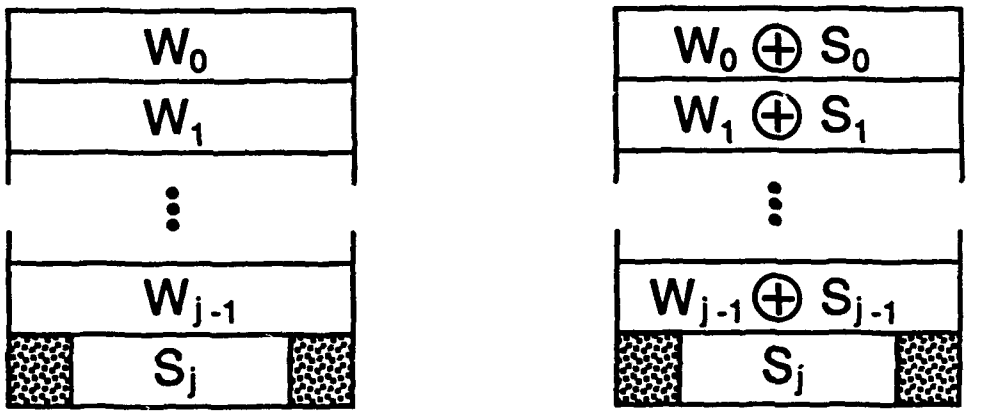


Figure 1: Basic Signature-Monitoring Technique.



(a) Conventional Approach (b) Instruction Hashing

Figure 2: Program Signaturing.

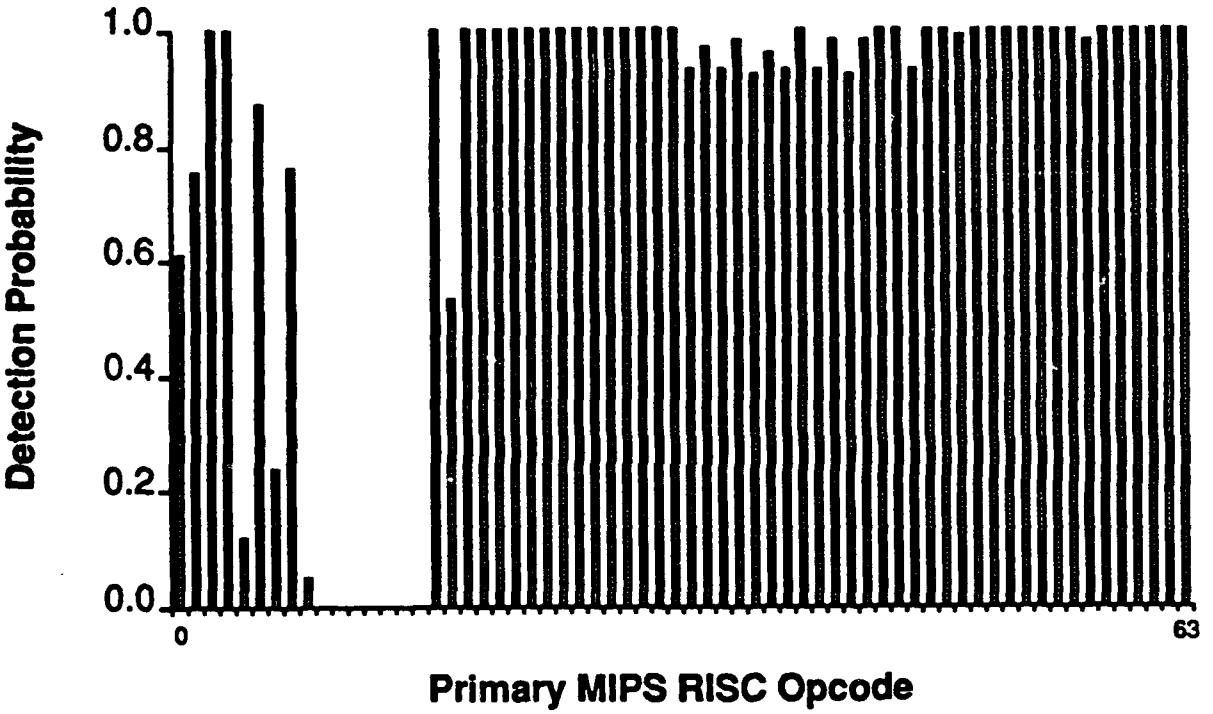


Figure 3: Detection Probability for Random Instruction Execution.

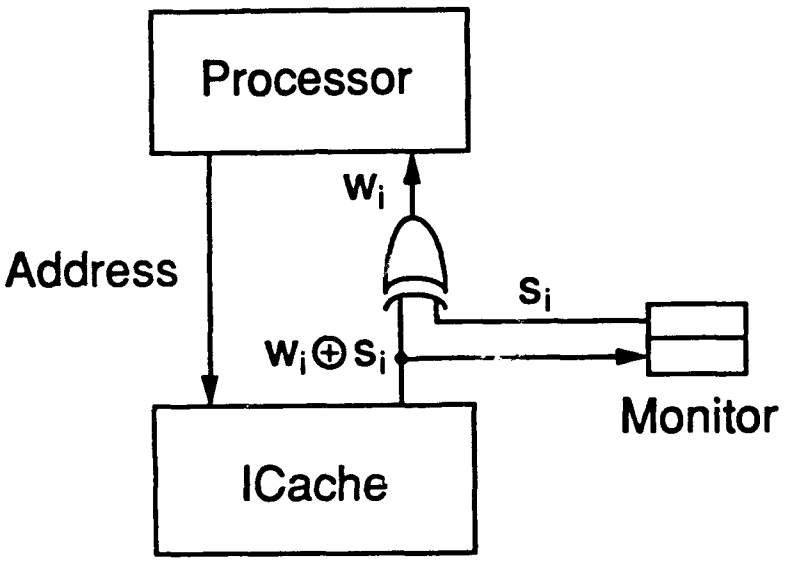


Figure 4: Instruction Fetch Delay.

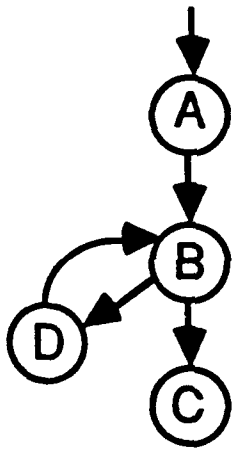


Figure 5: Example Program Graph.

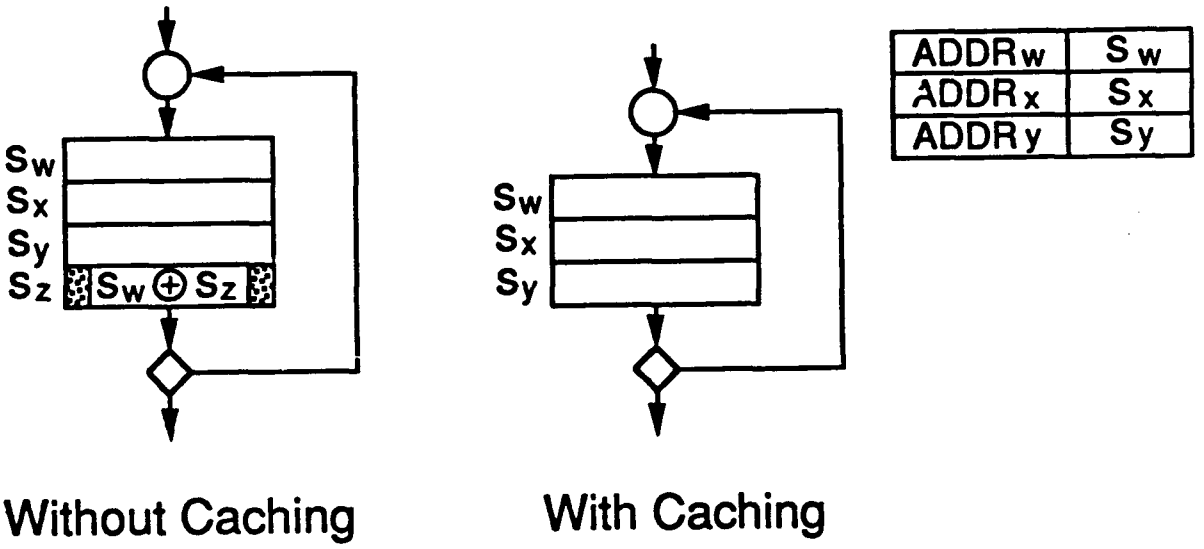


Figure 6: Signature Caching.

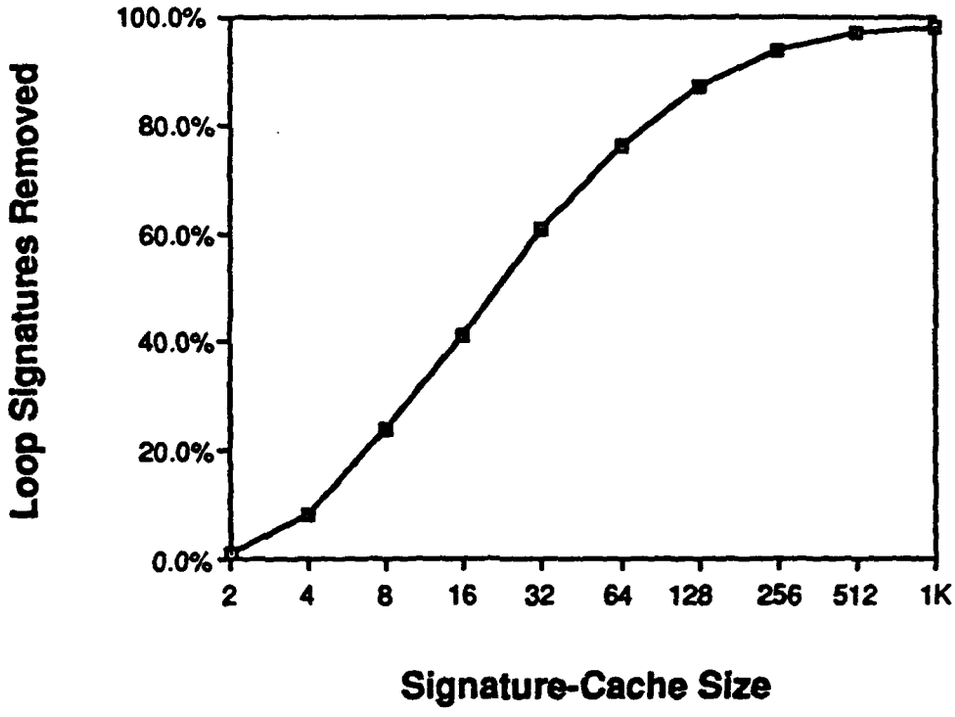


Figure 7: Fraction of Loop Signatures Removed vs. Cache Size

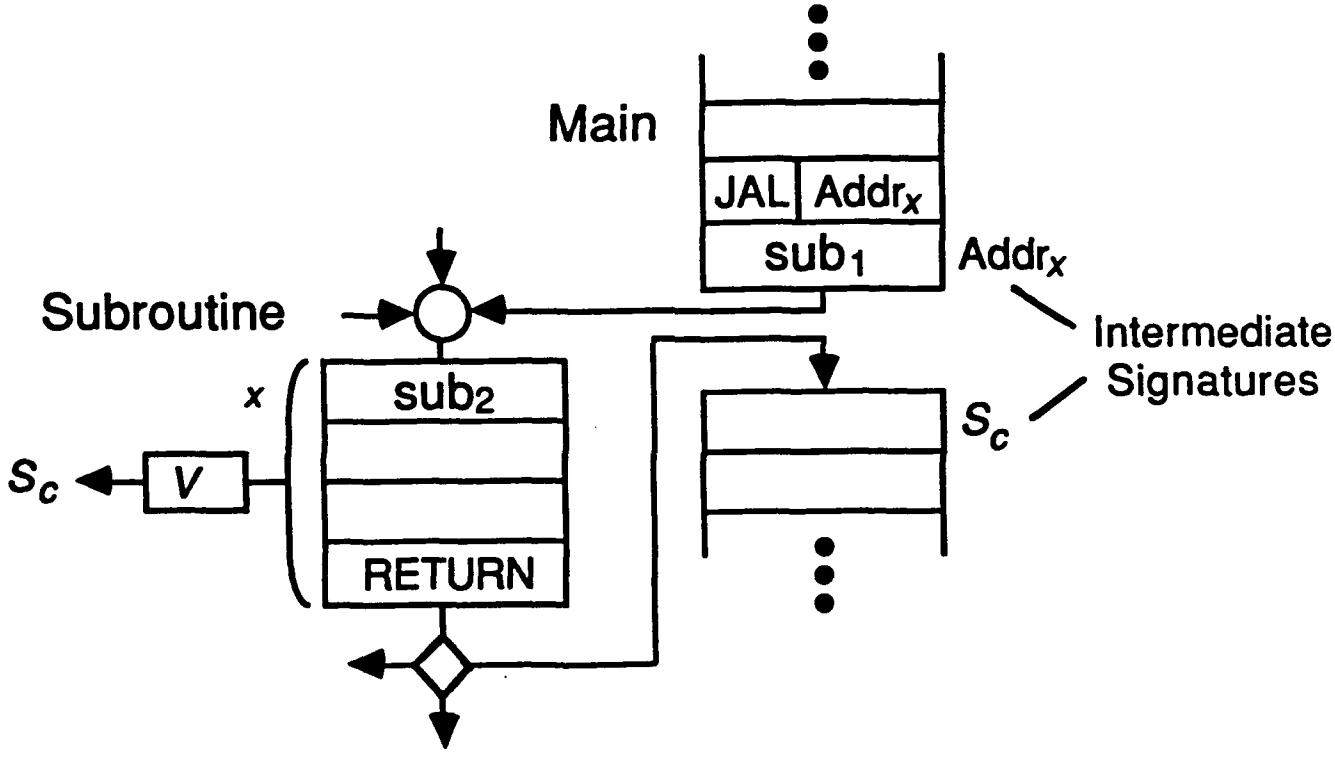


Figure 8: Subroutine Signaturing Technique.

	Range	Average
Instructions	16.3-63.8K	38.9K
NOPs	11.06-16.83%	12.97%
Conditional Branches	7.40-9.80%	8.75%
Calls (JAL)	4.99-7.96%	6.36%
Calls (JALR)	0.01-0.23%	0.09%
Returns	0.77-1.59%	1.24%
Loop Ends	1.23-2.19%	1.69%

Table 1: Data From Example MIPS RISC Program Code.