

Design and Evaluation of Preemptive Control Signature (PECOS) Checking for Distributed Applications

S. Bagchi, Z. Kalbarczyk, R. Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801
E-mail: [bagchi, kalbar, iyer]@crhc.uiuc.edu

Y. Levendel
Corporate Software Technology Center
Motorola Inc.
1303 East Algonquin Rd.
Schaumburg, IL 60196
E-mail: I.Levendel@motorola.com

Abstract

The paper presents the design and evaluation of PECOS, a PreEMptive CONTROL Signature technique for on-line detection of control flow errors. The technique uses assertions that can be embedded in the assembly language code and that are triggered by control flow instructions in the code. PECOS can detect errors in control flow that spans multiple subroutines or source files as well as control-flow, which is determined at runtime. PECOS is evaluated through software-based error injection—both directed control flow injections and random injections into the text segment of the running application. The injected errors model impact of failures in the address and data lines between a processor and memory. The effectiveness of PECOS is illustrated on a real application - DHCP (Dynamic Host Configuration Protocol) server. It is shown that PECOS detects more than 87% of control flow errors and reduces the incidence of fail-silence violations from 3.6% to 0.1% and the cases of process crash from 54.6% to 7.1%. Performance studies show a degradation of 15%-29% with instrumentation of the entire DHCP server, and a 5%-13% degradation with instrumentation of only the critical DHCP protocol engine.

Index terms: Control flow signature, preemptive checking, fail-silence, detection coverage, software implemented fault/error injection.

1 Introduction

In this paper, we focus on control flow errors, i.e., errors that cause a divergence from the sequence of program counter values seen during the error-free execution of the application. These errors can lead to data corruption, process crashes, or fail-silence violations¹. Control flow errors have been demonstrated to account for between 33% [OHL92] and 77% [SCH87] of all errors. Our study shows that about one third of the activated errors in the application code of a non-data-intensive application lead to control flow errors.

We propose a control-flow signature generation and preemptive checking technique called *PECOS* (**Pre**EMptive **C**Ontrol **S**ignatures). The PECOS target error model is any corruption that causes the application to take an incorrect control flow path. This may result from corruptions to the control flow instructions or from any other corruption (such as a register error) that subsequently affects the control flow. The proposed technique is shown to handle both static and dynamic control flow constructs.

Broadly speaking, the-state-of-the-art techniques detect between 15%-30% of the injected errors; the remaining errors are detected by the system detection techniques, such as raising an operating system signal. The unstated premise in providing error detection is that system detection (i.e., process crash) is an acceptable way of detecting an error and that only errors that escape both system detection and the proposed control flow error detection technique constitute a problem. From a recovery point of view it is questionable that process crash is an acceptable form of detection. Data from real systems has shown that while many crashes are benign, severe system failures often result from latent errors causing undetected error propagation, which results in crashes or hangs and severe recovery problems [CHA98, IYE86(a)(b), THA97, TSA83]. Further, application recovery time is higher when it involves spawning a new process (after crash) as compared with creating a new thread (in the case of multithreaded processes). In contrast to previous schemes, PECOS is preemptive in nature and is triggered before the error causes an application process crash. This approach has distinct advantages:

1. Because PECOS is preemptive, it significantly reduces the incidence of process crashes, thus enabling graceful termination of an offending process or thread.
2. PECOS minimizes error detection latency because it is triggered before an error manifests as a failure. Minimizing the error latency is important in reducing error propagation and the chance of other undesirable events such as checkpoint corruption in rollback-recovery schemes. Because it is preemptive, PECOS reduces fail-silence violations for the target application as well.

¹ A fail-silent application process either works correctly, or stops functioning (i.e., becomes silent) if an internal failure occurs [BRA96]. A violation of this premise is termed a fail-silence violation. In distributed applications, fail-silence violations can have potentially catastrophic effects by causing fault propagation.

3. Unlike other techniques, PECOS can also protect control flow instructions whose destinations are determined at run-time (e.g., a jump or a branch based on a register value determined at runtime). Modern applications increasingly contain such dynamic control flow characteristics.
4. PECOS is demonstrated on a real-world, client-server application: the Dynamic Host Configuration Protocol (DHCP) application. Most previous schemes have been demonstrated on simple applications, e.g., Quicksort (an exception is ECCA technique demonstrated on two integer SPEC92 benchmark programs [ALK99]). Although the evaluations are typically performed via error injection, it is not always clear how the experiments were conducted (e.g., whether the checking code itself was injected with faults/errors).
5. While clearly usable if the target source code is available, PECOS is also applicable if only the application executable is available. This is in contrast to several existing schemes (e.g., BSSC [MIR92]), which require high-level source code access.

The evaluation of PECOS was performed using DHCP application running on a Solaris platform. The results show that:

- PECOS detects over 87% of the injected control flow errors and 31% of the injected random errors to the application text segment.
- Process crashes are reduced from 54.6% to 7.1% for control flow errors and from 40% to 31% for random errors.
- The incidences of fail-silence violation are reduced from 3.6% to 0.1% for control flow errors and from 4.8% to 1.4% for random errors.

2 Related Work

The field of control-flow checking has evolved over a period of about 20 years. The first paper, by Yau and Chen [YAU80], outlined the general control flow checking scheme using the program specification language PDL.

2.1 Hardware Watchdog Schemes

Mahmood [MAH88] presents a survey of the various techniques in hardware for detecting control flow errors. Many schemes for checking control flow in hardware have been proposed [NAM82, SCH86, WIL90, MAD91, MIC91, UPA94, MIR95]. A summary of some of these techniques is presented in Table 11 (see Appendix). The basic scheme is to divide the application program into blocks. Each block has a single entry and a single exit point. A golden (or reference) signature is associated with the block that represents an encoding of the correct execution sequence of instructions in the block. Examples of signatures are the XOR function, the output from a Linear Feedback Shift Register (LFSR), and the output from a cyclic code generator. The selected signature is then calculated by a watchdog processor at runtime. The watchdog validates the application program at the end

of a block by comparing the runtime and the golden signatures of the block. The hardware watchdog-based techniques have been evaluated on small applications, and on fairly simple microprocessors, such as the Z80 [MAD91]. Among the hardware schemes, two broad classes of techniques have been defined that access the pre-computed signature in two different ways. Embedded Signature Monitoring (ESM) embeds the signature in the application program itself [WIL90, SCH86], while Autonomous Signature Monitoring (ASM) stores the signature in memory dedicated to the watchdog processor [MIC91]. Upadhyaya *et al.* [UPA94] explore an approach in which no reference signatures need be stored, and the runtime signature is any *m-out-of-n* code word.

Applying hardware schemes to distributed environments suffers from several limitations:

1. Hardware schemes are quite suitable for embedded small processors running single programs, but they do not scale well to complex modern processors. If multiple processes (or threads) execute on the main processor, then the memory access pattern observed by the external watchdog will not correspond to the signature of any single process, and hence an error will be flagged by mistake. This is due to the different possible interleaving among the multiple processes being executed.
2. The underlying assumption is that, in presence of errors, runtime memory accesses observed by the watchdog will differ from the reference signature. Consequently, an error after an instruction has been fetched from memory, e.g., while it resides in the cache of the main processor, will not be caught even if it causes application misbehavior. For current processors with increasingly large cache sizes, such errors are no longer negligible and a watchdog would have to be embedded in the processor itself, which in turn presents new challenges.
3. Hardware watchdog schemes require transmission on the system bus to communicate information to the external watchdog (e.g., the OSLC signature generator communicates the runtime signature to the checker [MAD91]). Transmission errors on the bus (address or data) during those transmission cycles would potentially reduce the coverage of the signature techniques.

2.2 Software Schemes

The software techniques partition the application into blocks, either in the assembly language or in the high-level language, and insert appropriate instrumentation at the beginning and/or end of the blocks. The checking code is inserted in the instruction stream, eliminating the need for a hardware watchdog processor. Representative software-based control flow monitoring schemes are Block Signature Self Checking (BSSC) [MIR92], Control Checking with Assertions (CCA) [KAN96], and Enhanced Control Checking with Assertions (ECCA) [ALK99]. A survey of some of these techniques is presented in Table 11 (see Appendix).

The outstanding issues with the software solutions proposed to date can be summarized as follows:

- To our knowledge, none of the existing software-based techniques is preemptive in nature, i.e., the checking involves executing code from the target address of the control flow instruction². If there is a control flow error, executing instructions from an invalid target location is likely to lead to system detection, causing a process crash. Consequently, for a majority of cases (e.g., see the experimental results in Section 5) system detection is triggered before the specific control signature checking technique, and this results in process crash.
- None of the techniques we are aware of has the capability of handling situations where the control flow behavior is determined by runtime parameters. This is because the reference signatures used, cannot be determined at runtime. As a result, a large class of control flow instructions are left uncovered, including register-indirect jumps, calls to dynamic libraries, and function calls using virtual function tables (as in object-oriented languages like C++). Application programs increasingly make use of dynamic library calls, follow object-oriented class hierarchies with inheritance, and have register-indirect jumps for high-level constructs like the *case* statement, all of which result in control flow structures that are determined at runtime.
- The evaluations of existing signature techniques often do not bring out how sensitive the system is to errors in the checking code itself. It is often not clear from the experimental results if the coverage values are obtained with or without the signature instructions being injected. If the checking code introduces control flow instructions itself (as in BSSC [MIR92] and CCA [KAN96]), then the effectiveness of such a technique depends on protecting the additional instructions.
- How generally applicable is the technique to off-the-shelf applications running on off-the-shelf processors? With the exception of the ECCA technique, which has been evaluated on two significant integer SPEC92 benchmark programs. To date, the evaluation of the existing software control flow techniques have been done on fairly simple applications. For example, BSSC uses programs containing linked list operations, quicksort and matrix transpose, CCA uses matrix multiply and quicksort. Fail-silence violations become critical in distributed environments because of the possibility of error propagation. Therefore, to obtain a thorough understanding of the effectiveness of a technique, it is important to run substantial distributed and client-server type workloads, in addition to the simple mathematical ones.

The identified problems with existing hardware and software control flow checking techniques served as a reference point in designing PECOS. Throughout this paper, we will show how our design and implementation leverages existing experience in control flow checking and addresses the issues raised above.

² A possible exception is the technique called concurrent process monitoring with no reference signatures [UPA94], which has aspects of preemptive checking built into it.

3 PECOS: Principle, Design, and Error Model

3.1 PECOS Preemptive Checking Principle

PECOS monitors the runtime control path taken by an application and compares this with the set of expected control paths to validate the application behavior. The scheme can handle situations in which the control paths are either statically or dynamically (i.e., at runtime) determined. Figure 1 depicts the basic PECOS instrumentation and the resulting change to the application code structure. The application is decomposed into blocks, and a group of PECOS instructions called Assertion Blocks are embedded in the instruction stream of the application to be monitored. Normally, each block is a basic block in the traditional compiler sense of a branch-free interval (i.e., the decomposition of the code is performed at the assembly-code level), and each basic block is terminated by a Control Flow Instruction (CFI), which is used as a “trigger” for PECOS. As shown in Figure 1(b), PECOS Assertion Block (AB) is inserted at trigger points. The assertion block contains: (1) the set of valid target addresses (or address offsets) the application may jump to, which are determined either at compile time or at runtime, and (2) code to determine the runtime target address. The determination of the runtime target address and its comparison against the valid addresses is done *before* the jump to the target address is made. In case of an error, the Assertion Block raises a *divide-by-zero* exception, which is handled by the PECOS signal handler. The signal handler checks, whether the problem was caused by a control flow error³, and if so take a recovery action, e.g., terminate the malfunctioning thread of execution.

3.2 Why Preemptive Detection Is Important

Figure 2 summarizes the problem with non-preemptive schemes and discusses the solution proposed by PECOS. Figure 2(a) shows a non-preemptive scheme, in which the validity of the control path is checked at the end of execution of a block. All existing control flow error detection schemes that we are aware of are non-preemptive in nature and cannot be easily made preemptive as discussed in Section 3.2. There are two fundamental reasons why a preemptive approach is preferable. Both reasons are related to the ease of recovery following detection.

1. Preventing process crash. Experiments with random error injection into the text segment of an application process have shown that with a non-preemptive scheme a significant number of cases lead to process crash before detection by the technique (Figure 2(b)). For example, our experiments using a SPECInt benchmark application instrumented with Enhanced Control Checking with Assertions (ECCA) [ALK99] showed an average of 32.7% of random errors and 57.0% of directed control flow errors resulted in system detections (i.e., process crash). The detailed results are shown in Section 4.9. A crash of the entire application process incurs a higher recovery overhead due to the overhead of process creation (the kernel allocating a new entry in the

³ The PECOS signal handler examines the PC (program counter) from which the signal was raised, and if it corresponds to a PECOS Assertion Block, concludes that a control flow error raised the signal.

process table and updating the structure's *inode* counter, file counter, etc.) and the overhead of reloading the entire process state from a checkpoint. The PECOS approach is to check the target location *before* the CFI is executed (Figure 2(c)). Thus, an error is diagnosed before a crash can occur. The application may then be terminated gracefully, freeing the resources. For example, for a multi-threaded process, only the offending thread may need to be killed. If checkpointing is used, the process's current state can be discarded and the process restarted from a previous checkpoint. It can be argued that a process crash (i.e., system detection) can also be prevented via an error handler, which intercepts system raised signals and terminates gracefully the application process or a thread. In this scenario, however, the handler takes actions after a corrupted instruction is executed and consequently undefined damage to the system would have happened. Data from real systems has shown that not all crashes are benign. Severe system failures often result from latent errors causing error propagation, which results in crashes or hangs and severe recovery problems [TSA83, IYE86(a)(b)].

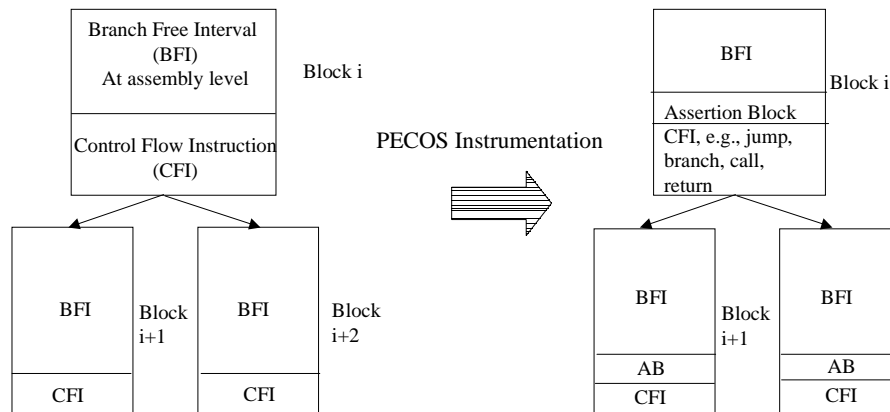


Figure 1: Change in the Code Structure Due to Inserting PECOS Assertion Blocks

CFI = Control Flow Instruction; AB = Assertion Block.

2. *Preventing error propagation.* A second problem with non-preemptive schemes is the possibility of error propagation. Data from bKAN96] shows that the latency of detection is several hundreds of instruction cycles for software-based schemes (approximately 500 instruction cycles for the Control Checking with Assertions (CCA) technique). Data on error latency [CHI89, YOU91, YOU92] has shown that while a very large majority of the errors are detected with a very small latency, the ones that remain undetected for a large period have the potential to cause severe problems. These ranges from checkpoint corruption and thus invalidating retries to sending an incorrect message out to a peer process in a distributed application. Thus error propagation complicates recovery, which in turn critically affects the overall system availability.

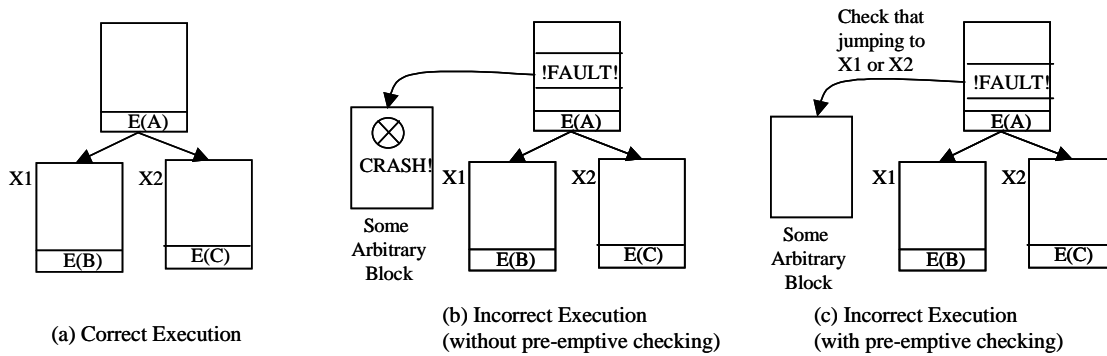


Figure 2: Reason for Preemptive Control Flow Checking

$E(X)$ = Signature block inserted at end of each block and point where checking is done.

$X1, X2$ = Target addresses.

Can existing techniques be made preemptive without substantial effort?

Our investigation suggests that while, in principle, existing techniques could perhaps all be made preemptive, there will be several theoretical and practical hurdles, which could fundamentally change the nature of the technique in question. For example, for a typical hardware scheme presented in [NAM83], the watchdog processor would have to stall the main processor on detecting that the application process has reached the end of a block, fetch the control flow instruction, perform the target address calculation, and verify that the target address points to one of the allowed destination blocks. Consequently, simply allocating a node id to each node will not suffice, but a table-mapping of node ids to the address range of each node will also have to be maintained. The watchdog design thus becomes substantially more complicated in order to handle multiple processes running on the application processor. For a software control-flow detection scheme like ECCA [ALK99], making the scheme preemptive would involve changing the Test Assertion at the end of the block so that it will:

- determine the target address of the control flow instruction,
- jump to the target address and read off the block id of that address, and
- perform a check on the block id to validate that it is one of the allowable blocks to jump to.

This is likely to insert control flow instructions in the Test Assertion itself, which will then become an added source of vulnerability. Another challenge is to determine the block id of the destination block without executing any of its instructions, since executing instructions from an illegal block can cause a process crash. Similar problems would need to be solved to make other techniques such as BSSC, preemptive in nature. Finally, even if it is claimed that a technique can be made preemptive with some effort, this claim needs to be substantiated by applying the technique to a sizeable workload; clearly this has not been shown.

The control flow signaturing technique by Upadhyaya *et al.* [UPA94] is possibly the most conducive to being made preemptive. The checking is performed at the beginning of the block, at the target location of a control flow instruction. However, control is still transferred to the target of the control flow instruction, and

consequently, an incorrect target calculation can still cause the process to crash. Moreover, a fine level of synchronization is involved between the checking hardware and the application processor. The checking circuitry should disable the application processor as soon as the justifying signature at the beginning of the target block is found. It should not let the application processor continue executing instructions from the target block till the check has been performed. Also, it is difficult to quantify the effectiveness of the technique, since no experimental implementation or evaluation is available. Summarizing, while the technique may be a good candidate for preemption, significant new effort is required before this can be achieved.

3.3 PECOS Instrumentation

In order to automate the instrumentation, we developed a PECOS parser, which embeds assertions into the application source code. The current parser is implemented for the SPARC architecture and works in two phases. Phase 1 of PECOS analyzes the control flow graph of the application and generates an assembly file with inserted Assertion Blocks that determine valid branch addresses. The user generates the object files from the assembly files and links them to create the executable. Phase 2 of PECOS inserts the correct address offsets in the executable. Phase 2 executes only if we need to instrument multiple files that constitute the application.

To illustrate the need for phase 2, consider the example in which the application consists of two source files (*foo.c*, *foo1.c*). The corresponding assembly files are *foo.s* and *foo1.s*, with a call being made from *foo.s* to *func1* which is defined in *foo1.s*. The offset of *func1* within *foo1.s* is 0x500, so the Assertion Block that precedes the call to *func1* has 0x500 in it as the valid address. When the executable is generated after linking *foo.o* and *foo1.o*, only then are the relative placements of the individual object files in the executable known. The file *foo1.s* is placed at an offset 0x12000, so that the offset of *func1* becomes 0x12500. The Assertion Block preceding the call to *func1* will need to be changed from 0x500 to 0x12500, which is done by phase 2 of the tool.

As Figure 3 indicates, to instrument an application with PECOS one may start with either the source code or the executable of the application. If the source code is available, then the path shown with solid lines in Figure 3 is to be followed. Otherwise the path shown with dotted lines is followed, where the executable is disassembled to generate the assembly code and then the PECOS instrumentation tool is applied to the generated assembly code⁴.

The PECOS tool has parts that are assembly-language specific and therefore architecture-specific (e.g., the assembly instruction format, the opcodes for the control flow instructions, and the Assertion Block written in assembly language). Other parts of PECOS are architecture-neutral (e.g., phase 2 where the executable is

⁴ Due to some nuances of the disassembler available on SPARC, some post-processing has to be done to convert the assembly code generated by the disassembler to a form that can be parsed by the PECOS tool. For example, the disassembler generates code that uses register *r33*, which is not available in the SPARC version 8 architecture.

patched). In phase 2 of PECOS, one assumes that the executable file is in the Executable and Linkable Format (ELF).

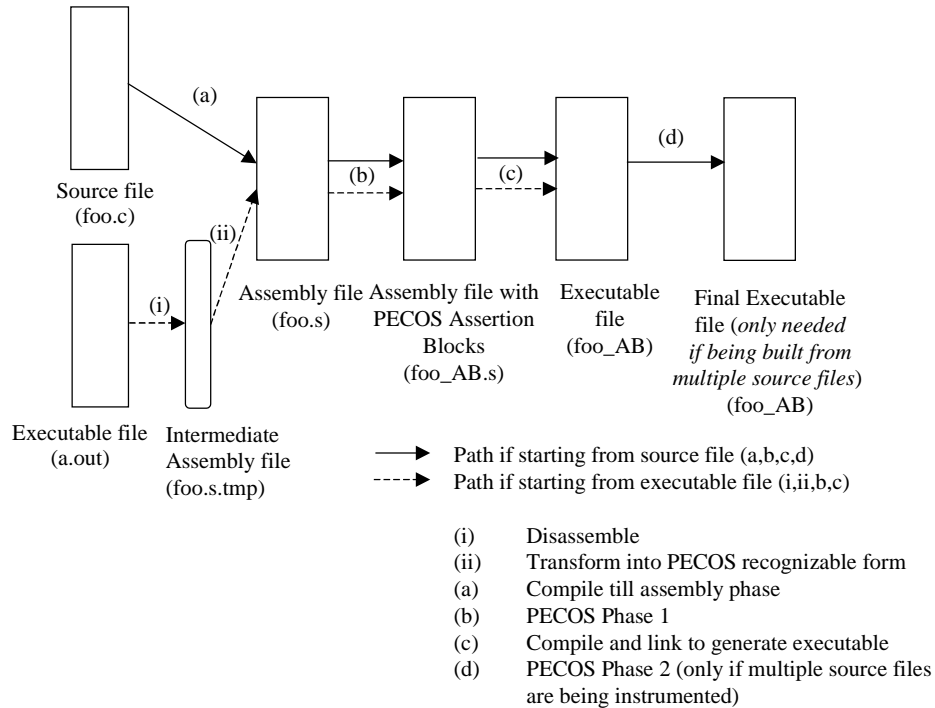


Figure 3: Process of Instrumenting an Application with PECOS

It is important to emphasize that PECOS uses virtual addresses for its Assertion Blocks, therefore relocatable code can be handled. Since PECOS handles assembly files rather than high-level source files (e.g., C++ files), the parsing is substantially less complicated. A tool such as ECCA, which has to embed assertions in C files, needs a more sophisticated parser because it has to handle more possible variations in the source code structure.

To be preemptive in detecting control flow errors PECOS assertion blocks are embedded into the assembly code. Operating on the assembly level enables obtaining the necessary information on valid control flow in advance and thus facilitates preemptive error detection. The same information is not easily obtainable at the high-level source code. Any attempt to do this at high-level would require access to the information available only at the assembly code and hence a high complexity tool (comparable to a compiler with runtime capabilities) would be needed.

3.4 Construction of PECOS Assertion Blocks

As shown in Figure 3, at compile time the PECOS tool instruments the application assembly code with Assertion Blocks placed at the end of each block. Each basic block terminated by a Control Flow Instruction (CFI) is considered a block in PECOS, and each CFI is preceded by an Assertion Block. Note that the Assertion Block itself does not introduce additional CFIs. Since we are trying to protect a CFI, it defeats the purpose to

have the Assertion Block insert further CFI(s). At runtime, the task of the Assertion Block can be broken down into two sub-tasks:

1. Determine the runtime target address of the CFI (referred to as X_{out} in the following discussion). We consider the following situations: (a) the target address of CFI is static, i.e., is a constant embedded in the instruction stream, (b) the target address is determined by runtime calculation, and (c) the target is the address of dynamic library call. We will discuss and illustrate each of the instances.
2. Compare the runtime target address with the valid target addresses determined by a compile time analysis. In general, the number of valid target addresses can be one (jump), two (branch), or many (calls or returns). For two valid target addresses: $X1$ and $X2$, the resulting control decision implemented by the Assertion Block is shown in Figure 4. The comparison is designed so that an impending illegal control flow will cause a divide-by-zero exception in the calculation of the variable ID, indicating an error.

1. Determine the runtime target address [= X_{out}]
2. Extract the list of valid target addresses [= $\{X1, X2\}$]
3. Calculate $ID := X_{out} * 1/P$,
where, $P = ![(X_{out}-X1) * (X_{out}-X2)]$

Figure 4: High-level Control Decision in the Assertion Block

Example: Assertion Block for CFIs with Constant Operand

In the example, PECOS is applied to protect a conditional branch statement. The target address of the CFI is static, i.e. is a constant embedded in the instruction stream. It will be demonstrated how an error in the CFI or in the assertion block instruction is captured.

Figure 5(i) shows a basic block (for the SPARC architecture) terminated with the conditional branch instruction. Figure 5(ii) shows the same block with the assertion block inserted into the object code by the PECOS tool. Instructions (1)-(3) load the runtime instruction, which has the branch opcode and the runtime target address combined, into register $l7$ ⁵. In this computation, SPARC uses Program Counter-relative offsets for addressing. Instructions (4)-(5) load the valid target offset⁶ in bytes into register $l6$. The offsets are determined by the usual compile-time creation and analysis of the application’s control flow graph. Instructions (6)-(8) form the valid instruction by combining the valid offset with the opcode and then load it into register $l6$. Instructions (9)-(12) compare the valid instruction word (in $l6$) with the runtime instruction word (in $l7$) and raise a divide-by-zero floating point exception in case of mismatch.

⁵ The assertion uses local registers $l5$, $l6$, $l7$ which were not used by the applications in our study.

⁶ In this case, there is only one valid target offset. In the general case, it will load the multiple valid offsets and compare the runtime offset against each one of them.

Now consider an error case. In fault/error scenario #1 in Figure 5, an error in the memory word storing the conditional branch instruction causes a corruption of the correct memory word (hex value: 0x02800017) to an incorrect value (hex value: 0xffffffff, say). In the absence of a preemptive control flow error detection scheme like PECOS, execution would have reached the incorrect memory word, and the operating system would have generated the illegal instruction signal which, in the absence of a signal handler, would have caused the application process to crash. Now consider the assembly code segment with the PECOS Assertion Block in place. At the end of instruction (3), the incorrect memory word 0xffffffff is loaded into register *l7*. The correct memory word is loaded into register *l6*, and a subsequent comparison between the two register values causes a floating-point exception signal to be raised. By examining the PC, the PECOS signal handler will determine that the signal was raised because of a control flow error. By design, the exception is raised before the control flow instruction is executed, therefore the error cannot propagate, and the PECOS signal handler can take appropriate recovery action, e.g., in the DHCP case, shut down the offending thread. Similarly, any error that hits the PECOS Assertion Block, e.g., changing the operand of *sethi* in (4) from, e.g., 0x5c to 0xfd (fault/error scenario #2), will also be detected by the comparison and division in instruction (12).

(i) Without PECOS

clr	%fp	} Basic Block (Average Size = 4-10 Assembly instructions)
ld	(%sp + 64), %l0	
add	%sp, 68, %l1	
sub	%sp, 32, %sp	
orcc	%g0, %g1, %g0	
cmp	%o0, -1	
be	.L0y	
nop		

(ii) With PECOS

	clr	%fp	} Basic Block (Average Size = 4-10 Assembly instructions)	
	ld	(%sp + 64), %l0		
	add	%sp, 68, %l1		
	sub	%sp, 32, %sp		
	orcc	%g0, %g1, %g0		
!	Begin Branch Assertion			
.zLL1:	sethi	%hi(.zLL1), %l7		(1) Load runtime control flow instruction (=X _{out}) into register l7
	or	%l7, %lo(.zLL1), %l7		(2)
Fault Scenario #2	ld	[%l7+64], %l7		(3) Load valid target address (in words) (= X1) into register l6
	sethi	%hi(0x5c), %l6		(4)
	or	%l6, %lo(0x5c), %l6	(5)	
	sethi	0xa000, %l5	(6) Form valid instruction word (= opcode + operand) in l6	
	or	%l5, 0, %l5	(7)	
	or	%l5, %l6, %l6	(8)	
	sub	%l7, %l6, %l7	(9)	
	cmp	%g0, %l7	(10) Comparison of valid and runtime instruction words. Will generate exception if in error	
Detection Point	subx	%g0, -1, %l7	(11)	
	sdiv	%l6, %l7, %g0	(12)	
	End Branch Assertion			
Fault Scenario #1	cmp	%o0, -1		
	be	.L0y		
	nop			

Figure 5. Assembly Code for Branch Assertion Block

(i) Application assembly code prior to PECOS instrumentation; (ii) Application assembly code instrumented with PECOS Assertion Block.

PECOS Runtime Extension.

The runtime extension of PECOS allows for protecting control flow instructions whose target addresses are dynamically determined by run-time calculations. An example of such an instruction is the register-indirect jump where the target address calculation uses the value of a register. To handle such cases, phase 1 of the PECOS analysis tool performs a data discovery pass that generates the definition-use pairs (DU pairs) for all such runtime-dependent control flow instructions. The embedded Assertion Block has instructions to read off the runtime value that determines the control flow. In Figure 6(a), the use of the register *r1* in the jump instruction is preceded by the definition of its value in instruction (i). In this simple case, the value of the register depends on its last definition only, and the definition and the use are in a single straight-line path. In this case, PECOS can insert an Assertion Block before the use, i.e., the jump instruction. Thus, instruction (ii) in Figure 6(a) can be protected. Similar procedure can be used to protect the *jump* instruction in the scenario depicted in Figure 6(b). More complex situation is illustrated in Figure 6(c), in which the value of the register (*r1*) depends on the execution path that has been followed by the application prior to the definition of the value in the register (*r1*). In this scenario *definition-use* pairs are in multiple (two in the example) paths and to determine which path has been followed by the application requires additional support, which is being currently investigated.

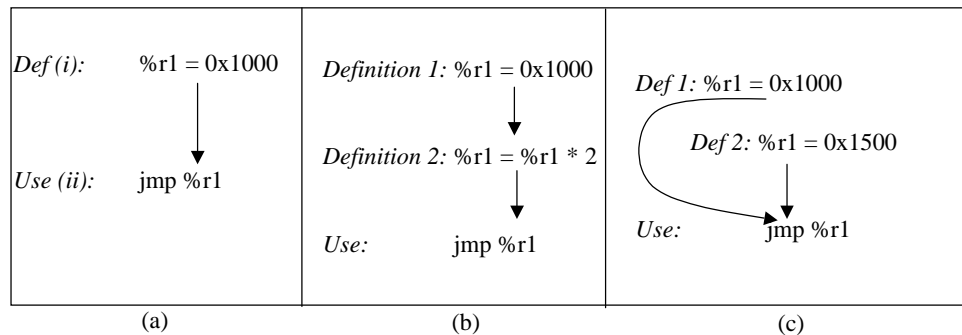


Figure 6: Applicability of PECOS to Runtime Dependent Control-flow Instructions

Another issue to address is how to handle dynamic library calls. Here we use the information provided in the Procedure Linkage Table (PLT). After the loader has loaded the application, the positions of the dynamic library calls in the application are determined and filled up in the PLT. To protect the calls to the dynamic library routines, PECOS considers the PLT as the definition point of the DU pair, with the call instruction being the use point. Then, a similar approach to the one depicted in Figure 6 is followed. The Assertion Block placed before the call asserts that the target of the call will be the location loaded in the PLT.

Example: Assertion Block for CFIs with Register Operand

An example of PECOS instrumentation is provided for a CFI for which the target address is determined by runtime calculation; a register indirect jump. Two error cases—a memory error and a register error—demonstrate how the Assertion Block captures the error. Figure 7(i) shows an unconditional jump instruction in

an application's assembly code (where the location of the jump is determined at runtime by the value in register 17). Figure 7(ii) shows the instruction protected by a PECOS Assertion Block. In this case, to form the Assertion Block, it is not sufficient to assert the contents of a memory location; we must assert on the contents of a register. The example illustrates the ability of PECOS to form reference signatures from runtime quantities. The inserted AB examines the last definition of the register value immediately preceding the jump instruction, where a value is loaded into the register (instructions (1)-(2)). In general, the PECOS tool will have to search further definitions for the corresponding register value.

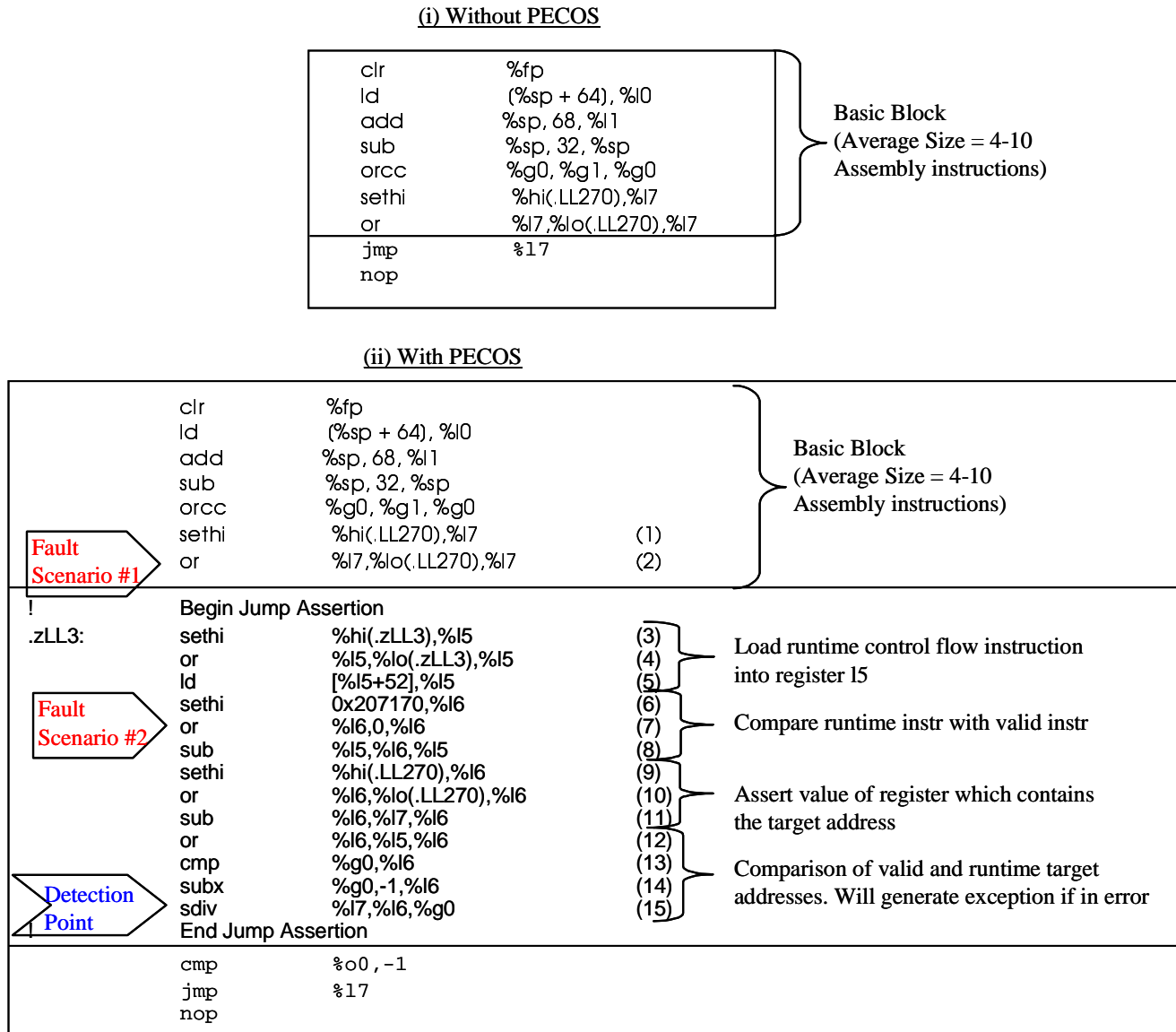


Figure 7. Assembly Code for Jump Assertion Block

(i) Application assembly code prior to PECOS instrumentation; (ii) Application assembly code instrumented with PECOS Assertion Block.

The general structure of the Assertion Block is similar to the branch case. However, in the jump case, both the instruction and the register value must be compared to valid values. This is necessary because an invalid control

flow path may be taken either if the instruction is corrupted (e.g., `jmp %17` becomes `jmp %16`) or if the value of the register is corrupted (e.g., value in register `17` is changed from `.LL270` to `.LL271`). The instruction comparison is performed through instructions (3)-(8), and the register value comparison is performed through instructions (9)-(11).

Now consider an error case. In fault/error scenario #1 in Figure 7, a register error results in the corruption of the value in register `17`. In this simple example, the time during which the execution is being protected by the PECOS Assertion Block is the time between the execution of instructions (1) and (11). In a more general case, there will be a larger number of intervening instructions between the loading of the register value and its use in the jump instruction. In that case, the Assertion Block will be protecting a larger time window of execution. Consider the case in which the register error causes the value of `.LL270 + 0x10` to be loaded into `17` instead of `.LL270`. In the absence of preemptive control flow error detection, execution would have caused a jump to `.LL270 + 0x10` on executing instruction (16). On executing instructions from this invalid target, the process can crash (for example, because of an illegal memory access to an unallocated memory region), or worse still, it may output a wrong value leading to a fail-silence violation. With the PECOS Assertion Block inserted, the comparison in instruction (11) will indicate an error, and instruction (15) will raise a divide-by-zero resulting in the actual detection. Similarly, if an error affects a PECOS instruction, e.g. as in fault/error scenario #2, and corrupts the operand of the `or` instruction in (7) from `0x00` to `0x45`, the comparison in (8) will produce a non-zero result and instruction (15) will raise the exception thereby detecting the error.

In summary, we have discussed two cases: one in which the control-flow is static and another in which the control flow is dynamically determined at run-time. The ability to detect errors in the latter case is unique to PECOS.

3.5 Error Types That PECOS Protects Against

PECOS detects control flow errors. These errors may occur because of various low-level errors or failures. The error models at different levels that can be handled by PECOS and a representative example of each type are discussed below.

1. Single or multiple bit flips in memory—main memory, or cache (on-chip or off-chip), e.g., a burst error corrupts a call instruction in L1 cache.⁷
2. Transmission error during communication between any two levels of the memory hierarchy, e.g., an error on the address line when a control flow instruction is being fetched from memory to the processor.

⁷ When the PECOS Assertion Block accesses the control flow instruction, it is likely that the instruction is already in the instruction cache. For PECOS to detect cache errors, the caching algorithm should be such that it should not fetch the control flow instruction again from memory into the data cache but should read it directly from the instruction cache.

3. Register error, e.g., corruption of a value in the register, which determines the destination address of an unconditional jump instruction.
4. Software bugs impacting the control flow, e.g., the call graph determined from the UML specification (a high-level specification of the software) may show that function *foo1* is called from *foo2*, *foo3*, and *fooa*; because of software bug the implemented code has a call to *foo1* from *foo5*. The automated generation of the assertions from the UML specification and their insertion into the source code are not currently implemented in the PECOS tool.

PECOS cannot capture control flow errors that result from the corruption of a non-control flow instruction into a control flow instruction, since the Assertion Blocks are inserted only before control flow instructions. However, it is observed that the Instruction Set Architectures (ISAs) are such that the Hamming distances of opcodes belonging to the non-control flow category and the control flow category are quite large, e.g., in SPARC. Therefore, it is reasonable to expect that errors that cause a non-control flow instruction to be transformed into a valid control flow instruction are rare in practice. While some signature schemes [ALK99] claim to detect such errors, our experience tells us that if an illegal control flow branch occurs due to corruption of a non-control flow instruction, the application is likely to crash before reaching the trigger for checking. This is illustrated by experiments with non-preemptive technique discussed in Section 4.9.

4 Experimental Evaluation of PECOS

In this section, we describe the evaluation of PECOS applied to a substantial real-world application: the Dynamic Host Configuration Protocol (DHCP) application. DHCP was chosen because it is:

1. widely used in networked wireless/wireline to provide a critical service,
2. a “real-world” application that cannot be simply handcrafted by the developers of the CFI technique and could potentially be a benchmark on which future techniques are evaluated, and
3. an application for which the property of fail-silence is especially important, i.e., sending out an incorrect value can have a significant negative consequence.

Error injection campaigns were conducted into the baseline (uninstrumented) DHCP first and then into DHCP instrumented with PECOS. The goal was to evaluate relative improvements in the dependability metrics, including percentage of system detection, hangs, and fail silence violations. NFTAPE, a software-implemented fault injection⁸ tool, was used for conducting the error injection campaigns [STO00] using a wide range of error models.

⁸ Here we actually inject errors (in accordance with Laprie definition [LAP92]), although the tools are typically referred to as *fault injection* tools.

4.1 The Target Application: DHCP

The Dynamic Host Configuration Protocol (DHCP) application is widely used in mobile and wireless environments for network management. It provides critical services, such as IP address allocation, and can be regarded as representative of a control flow intensive client-server applications. The current study used the DHCP version 2 implementation from the Internet Software Consortium [ISC00].

DHCP [RFC97] implements an Internet Draft Standard Protocol for providing configuration information to hosts on an IP network. This is a client-server protocol used by the client to obtain information about a dynamic network (i.e., a network in which the configuration of the network changes frequently or clients join and leave the network frequently). Importantly, DHCP supports dynamic allocation of IP addresses in which the server allocates an IP address to the client for a limited amount of time (termed as the *lease time*), after which the server can reclaim the address. If the server is unavailable, new hosts cannot be allocated an IP address. Alternately, if the server is behaving incorrectly (i.e., in a non-fail-silent manner), hosts can be denied entry into the network or can be allocated an incorrect or in-use IP address, and be unable to perform any operations in the network thereafter.

The protocol operates in two phases. In Phase 1, the client broadcasts a DHCPDISCOVER message on its local physical subnet. One or more servers on the network respond to the client request by sending a DHCPOFFER message, which contains a tentative offer of an IP address and configuration parameters. In Phase 2, the client collects the DHCPOFFER responses from all the servers that respond, chooses one server to interact via a DHCPREQUEST broadcast message. On receiving the DHCPREQUEST message, the chosen server commits the binding of the IP address of the client to a lease database in stable storage and responds with a DHCPACK message. If the selected server is unable to satisfy the DHCPREQUEST message, then the server responds with a DHCPNAK message. When the client's lease is close to expiration, it tries to renew the lease by re-sending the DHCPREQUEST message.

4.2 Types of Errors Injected

We use error models based on the extensive experiments of Abraham *et al.* [KAN95] and adding random memory errors. These models represent a wide range of transient hardware errors and some software bugs⁹. Several studies indicate that more than 90% of the physical failures in computers are transient [LAL85, SIE98]. Also failures that span many cycles are easily detected, even by relatively simple error detection mechanisms

⁹ For example, a pointer overflow error may be modeled by a hardware bit flip in the data line when the operand of a load instruction is being fetched.

[SCH87, MAD94]. Therefore, transient hardware errors/failures were adopted as the error model for the current experiments¹⁰.

The errors chosen are depicted in Table 1. For example, the DATAOF models an error during transmission over data line, an error in the corresponding memory or cache word, or a software bug corresponding to an uninitialized pointer. All the errors, except ADDIF2, are single-cycle errors, while ADDIF2 is a 2-cycle error. In ADDIF and ADDIF2, the erroneous instructions that are executed are also valid instructions in the instruction stream of the application. The ADDIF2 error may be caused, for example, if the Program Counter (PC) register has a stuck-at error that lasts two cycles. In the DATAInF error model, it is assumed that the data line error may affect either the opcode or the operand of the instruction.

Error Model	Description
ADDIF	Address line error resulting in execution of a different instruction taken from the instruction stream
DATAIF	Data line error when an opcode is fetched
DATAOF	Data line error when an operand is fetched
DATAInF	Data line error when an instruction (whether opcode or operand) is being fetched
ADDIF2	Address line error resulting in execution of two different instructions taken from the instruction stream
ADD OF	Address line error when an operand is fetched
ADD OS	Address line error when an operand is stored
DATA OS	Data line error when an operand is stored

Table 1: Transient Error Models for PECOS Evaluation

The experiments were conducted on the baseline target (without PECOS instrumentation) and to the PECOS-instrumented target. In the PECOS-instrumented case, error injections were divided into two categories: (a) injection of errors randomly in the instruction stream of the application and (b) directed injection of control flow instructions. In the latter, all control flow instructions in the instruction stream (e.g. call, return, branch or jump) were chosen as targets of error injection.

4.3 Classification of Results

For this study, we define a run as a single execution of the target process. For the DHCP workload, each run lasts 30 seconds after which the server and the client are terminated, cleanup is done, and the next run is initiated. Typically, in this period, the client and the server go through five rounds of the protocol described in Section 4.1¹¹. A single error is injected during each run. The outputs of each of the participating processes (the DHCP client and server) are logged and analyzed off-line to categorize the results. The results/outcomes from the runs are classified according to the categories defined and described in Table 2.

¹⁰ Permanent failures are usually covered by dedicated hardware detection mechanisms, including periodic sampling for checking status of hardware components.

¹¹ We experimented with a range of time intervals and found that if the fault is not activated during five (or fewer) rounds of the protocol, it is not activated at all (with high likelihood).

Category	Notation	Description
Error Not Activated	Discarded	The erroneous instruction is not reached in the control flow of the application. These runs are discarded from further analysis.
Error Activated but Not Manifested	NE (No Error)	The erroneous instruction is executed, but it does not manifest as an error, i.e., all the components of the DHCP server exhibit correct behavior.
PECOS Detection	PD	PECOS Assertion Blocks detect the error prior to any other detection technique or any other result.
System Detection	SD	The OS detects the error by raising a signal (e.g., SIGBUS) and as a result the DHCP server crashes.
Application Hang	SH (Server Hang)	The application gets into a deadlock or livelock and does not make any progress.
Program Aborted	SC (Semantic Check Detection)	The application program itself detects an error and exits with an error code.
Fail-silence Violation	FV	The application communicates a wrong value/message to other software components.

Table 2: Categorization of Results from Error Injection

Fail-silence Violation category is considered to have the most debilitating effect on system availability. For our DHCP study, we defined a fail-silence violation as any of the following:

1. The client or the server sends a message that does not follow the protocol's state transition diagram.
2. The server does not offer an IP address to the client (in our setup we run a single client, so there are always available addresses in the pool).
3. The server does not allocate the immediate next IP address available in the address pool to the client.

5 Results

This section presents a summary, detailed results, and discussion of the results from the error injection campaigns. Table 3 and Table 4 present, respectively, the results of the injections directed into control flow instructions and the results of injections into instructions randomly selected from application instruction stream. Each row in the two tables gives a sum of the number of cases from all error injection campaigns. The fourth column gives the improvement due to the PECOS instrumentation. Reduction in system detection, hang, program aborts, and fail-silence violations are considered improvements.

The results in Table 3 characterize the effectiveness of PECOS in detecting errors when an error directly affects a control-flow instruction, (i.e., effectiveness in detecting errors, PECOS was designed to protect against). The major improvements gained by instrumenting the DHCP server with PECOS are:

- PECOS detects more than 87% of all activated errors.
- Fail-silence coverage is improved by about a factor of 36.
- System detection is reduced by a factor of 7.7.
- Application hang cases are reduced by about 3.2 times.

Category	Without PECOS	With PECOS	Improvement Factor {(measured value w/o PECOS) / (measured value with PECOS)}
Error Not Activated	1380	1446	n/a
Error Activated but Not Manifested	426 (38.0%)	46 (4.3%)	n/a
PECOS Detection	n/a	922 (87.5%)	n/a
System Detection	612 (54.6%)	75 (7.1%)	7.7
Application Hang	18 (1.6%)	5 (0.5%)	3.2
Program Aborted	24 (2.2%)	5 (0.5%)	4.4
Fail-silence Violation	40 (3.6%)	1 (0.1%)	36.0
Total	2500	2500	n/a

Table 3: Cumulative Results from Directed Injection to Control Flow Instructions

The results in Table 4 provide an insight into how well PECOS performs when the corrupted instruction is not necessarily a control flow instruction, but it is chosen at random from the instruction stream. Results from random injections give a measure of how often a random error affects a control-flow of the application and is picked up by PECOS. It is possible that randomly corrupted instruction immediately crashes the process before the process executes the PECOS assertion block. Table 4 shows that the proportion of PECOS detection, reduction in fail-silence violations, and system detections are more moderate than for directed control flow injections. From the results, one can draw the following conclusions:

- PECOS detects more than 31% of the activated errors even when the total set of injected errors includes both control and data errors.
- The proportion of fail-silence violations is reduced more than three-fold.
- The largest gain is observed in the case of process hangs which is reduced by more than 14 times.
- System detection is reduced by about 1.3 times.

The above results allow us to estimate the percentage of random errors in the DHCP server that manifest as control flow errors. Assume that the coverage of PECOS for detecting control flow errors is $C\%$ and for random errors to the text segment of the application, PECOS detects $D\%$ of errors. Then, the percentage of random errors in the text segment that become control flow errors is given by the following expression:

$$X = (D / C) * 100 [\%]$$

From the observed results, $C = 87.5\%$ (Table 3 directed injection to control flow instructions), $D = 31.5\%$ (Table 4 random injection to the instruction stream) and therefore, $X = 36.0\%$. This number agrees well with that of Ohlsson *et al.* [OHL92], where simulation of a 32-bit RISC processor showed that 33% of activated errors manifested as control flow errors. This result also indicates that in addition to control flow checking, it is important to have effective data error detection mechanisms to improve overall system reliability.

Category	Without PECOS	With PECOS	Improvement Factor {(measured value w/o PECOS) / (measured value with PECOS)}
Error Not Activated	2486	2476	n/a
Error Activated but Not Manifested	770 (50.9%)	513 (33.7%)	n/a
PECOS Detection	n/a	480 (31.5%)	n/a
System Detection	610 (40.3%)	483 (31.7%)	1.3
Application Hang	22 (1.4%)	2 (0.1%)	14.0
Program borted	40 (2.6%)	24 (1.6%)	1.6
Fail-silence Violation	72 (4.8%)	22 (1.4%)	3.4
Total	4000	4000	n/a

Table 4: Cumulative Results from Injection to Random Instructions from the Instruction Stream

5.1 Downtime Improvement

In the event of system detection, process recovery has to be performed. The time to recover a process involves at least the time for process spawning and reloading the state of the entire process from a checkpoint on disk. The time to recover when PECOS detects the error is the time to gracefully terminate the offending thread¹² or creating a new thread. Conservatively, this is less expensive in terms of time overhead by a factor of 10. Our best case, measurements (taken on SPARC platform running Solaris operating system) show that the time to create a process is of order of 200 ms, while to spawn a new thread takes about 300 μ s, i.e., a difference of three orders of magnitude. Using the assumptions given above, we estimate the reduction in the downtime of the application instrumented with PECOS. Table 5 gives the estimation of the downtime improvement for the midrange factor of 100 (the ratio between the time for process crash recovery and thread recovery) and Figure 8 plots the improvement factor for three data points (10, 100, and 1000). Observe that as the cost of thread-based recovery is reduced, the downtime improvement approaches the reduction in system detection (7.7, Table 3).

	Estimated recovery time for process crash	Estimated recovery time for PECOS detection	Percentage of system detection (i.e., process restart possibly from a checkpoint)	Percentage of PECOS detection (i.e., graceful thread termination)	Estimated downtime	Improvement factor
Baseline case	t_{rec}		54.6%	0.0%	$0.546 * t_{rec}$	7.3 (0.546 / 0.075)
PECOS-instrumented application		$t_{rec} / 100$	7.1%	47.5%	$0.075 * t_{rec}$ ($0.07 * t_{rec} + 0.475 * t_{rec}$)	

Table 5: Estimation of Reduction in Application Downtime

¹² For example, in the multithreaded implementation of DHCP application each execution thread corresponds to a single client request. In this scenario termination of a thread in the case of a detected error is a vital way of recovery. We loose a single request not the entire application.

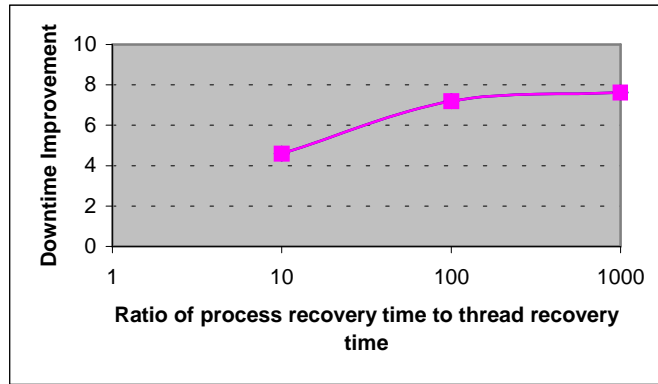


Figure 8: Downtime Improvement Due to Reduction in System Detection for Different Ratios of the Costs of Process Recovery and Thread Recovery

Clearly, the reduction in the incidence of fail-silence violations, program hangs, and program aborts will also reduce the downtime for the PECOS-instrumented server. However, it is difficult to estimate the recovery cost for the fail-silence violation, process hang, and process abort cases. Since fail-silence violations result in error propagation, we can expect the recovery time to be significantly higher than for system-detected errors (i.e., crash errors).

5.2 Detailed Results & Discussion

The results of all error injection campaigns are presented in Table 6¹³. The percentage of runs belonging to each category is mentioned, along with the actual number of runs falling in each category (in parentheses). Recall that the number of injected errors, which were not activated are discarded in our calculations. As a result, the total of the number of runs for each category does not add up to 500, and does not add up to the same figure for the different campaigns. For the PECOS instrumented server, the error activation rate falls between 28.6% (ADDOS) and 49.8% (ADDIF, only CFI injection). For the uninstrumented server, the error activation rate falls between 21.8% (DATAOS) and 50.6% (DATAOF, only CFI injection). Some important observations from the results in Table 6 are:

- The proportion of system detection is reduced for all error injection campaigns except for DATAIF for random instruction injections and ADDOF (see the discussion below).
- The proportion of fail-silence violation is reduced for all the campaigns, including full elimination of fail-silence violation altogether for four error injection campaigns.
- The percentage of activated errors that are detected by PECOS is high —above 87% for all directed injection of control flow instructions.

¹³ Statistical confidence intervals were calculated for cases, where the number of observations is greater than seven. To avoid cluttering up the tables we do not include the confidence intervals. Note that our major conclusions are based on the totality of the observations.

Error Model	Outcome Categories	Injection Target <i>Any Instruction in Instruction Stream</i>		Injection Target <i>Control Flow Instructions</i>	
		Baseline (without PECOS)	With PECOS	Baseline (without PECOS)	With PECOS
ADDIF	NE	49.7% (98)	22.0% (44)	49.8% (106)	8.8% (22)
	PD	N/A	50.5% (101)	N/A	87.6% (218)
	SD	42.2% (83)	25.0% (50)	37.1% (79)	1.2% (3)
	SH	3.6% (7)	0.0% (0)	2.3% (5)	0.4% (1)
	SC	2.0% (4)	1.0% (2)	2.8% (6)	1.6% (4)
	FV	2.5% (5)	1.5% (3)	8.0% (17)	0.4% (1)
DATAIF	NE	52.3% (113)	34.4% (78)	28.6% (62)	4.2% (9)
	PD	N/A	21.6% (49)	N/A	86.4% (185)
	SD	43.5% (94)	43.2% (98)	65.5% (142)	9.4% (20)
	SH	0.9% (2)	0.0% (0)	1.8% (4)	0.0% (0)
	SC	1.4% (3)	0.4% (1)	2.3% (5)	0.0% (0)
	FV	1.9% (4)	0.4% (1)	1.8% (4)	0.0% (0)
DATAOF	NE	52.1% (112)	40.5% (89)	30.3% (77)	1.4% (3)
	PD	N/A	39.1% (86)	N/A	86.7% (189)
	SD	42.8% (92)	19.0% (42)	61.7% (156)	11.4% (25)
	SH	0.9% (2)	0.9% (2)	1.6% (4)	0.5% (1)
	SC	0.9% (2)	0.0% (0)	2.4% (6)	0.0% (0)
	FV	3.3% (7)	0.5% (1)	4.0% (10)	0.0% (0)
DATAInF	NE	56.4% (110)	40.9% (92)	30.5% (68)	1.9% (4)
	PD	N/A	29.4% (66)	N/A	86.5% (180)
	SD	38.5% (75)	28.4% (64)	65.1% (145)	11.1% (23)
	SH	1.5% (3)	0.0% (0)	0.9% (2)	0.0% (0)
	SC	0.5% (1)	0.0% (0)	0.4% (1)	0.5% (1)
	FV	3.1% (6)	1.3% (3)	3.1% (7)	0.0% (0)
ADDIF2	NE	73.1% (147)	65.1% (114)	52.8% (113)	4.9% (8)
	PD	N/A	21.2% (37)	N/A	92.6% (150)
	SD	21.9% (44)	10.9% (19)	42.1% (90)	2.5% (4)
	SH	1.0% (2)	0.0% (0)	1.4% (3)	0.0% (0)
	SC	1.0% (2)	1.7% (3)	2.8% (6)	0.0% (0)
	FV	3.0% (6)	1.1% (2)	0.9% (2)	0.0% (0)
ADDOF	NE	48.5% (100)	32.9% (53)		
	PD	N/A	18.6% (30)	N/A	N/A
	SD	42.7% (88)	42.2% (68)		
	SH	1.5% (3)	0.0% (0)		
	SC	4.4% (9)	4.4% (7)		
	FV	2.9% (6)	1.9% (3)		
ADDOS	NE	50.3% (88)	21.0% (30)		
	PD	N/A	49.0% (70)	N/A	N/A
	SD	28.0% (49)	21.6% (31)		
	SH	1.7% (3)	0.0% (0)		
	SC	7.4% (13)	5.6% (8)		
	FV	12.6% (22)	2.8% (4)		
DATAOS	NE	1.8% (2)	7.5% (13)		
	PD	N/A	23.7% (41)	N/A	N/A
	SD	78.0% (85)	64.2% (111)		
	SH	0.0% (0)	0.0% (0)		
	SC	5.5% (6)	1.7% (3)		
	FV	14.7% (16)	2.9% (5)		

Table 6: Results of Error Injection to DHCP Server

Throughout this paper, we have emphasized the importance of reducing cases of system detection, application hang, and fail-silence violation to minimize the downtime and thus improving availability. An important benefit of fast error detection is ability to reduce likelihood of error propagation. Our measurements show that between tens and thousands of instructions can be executed from the time when the corrupted instruction is executed and when the process crashes. This window of system vulnerability may be enough to corrupt significant system data or impact key system resources or even crash the operating system. The results in Table 6 clearly indicate that PECOS is very efficient in reducing the system vulnerability window.

5.2.1 *Effect of PECOS on System Detection*

The percentage of System Detection is reduced in the instrumented DHCP server for all but two error injection campaigns. For DATAIF (for injections to randomly selected instructions) and ADDOF error models, the percentage of system detections remains about the same for DHCP instrumented and uninstrumented with PECOS. This is due to the fact that the two error types result either in executing an invalid instruction (DATAIF) or in loading an invalid/incorrect operand (ADDOF). In the first case the application will crash before it is able to execute any further instructions (including instructions from the PECOS assertion block). In the second case, the load instructions (recall that the two error models are applied to randomly selected instructions) are not directly protected by PECOS and in most of cases the invalid operand will lead to the application crash. PECOS is designed to detect errors provoked by the two error types only if they affect the application control flow.

For ADDOS and DATAOS error types we observe reduction in the number of system detection. This indicates that there are instances of the application control-flow that are dependent on load and store instructions, e.g., through register-indirect jump instructions. The runtime extension to PECOS (presented in Section 3.4) enables detection of these types of errors.

5.2.2 *Effect of PECOS on Fail-Silence Violations*

The results show that PECOS is able to reduce the incidence of fail-silence violations for all the campaigns, (eliminating it altogether for four campaigns). The remaining cases of fail-silence violations that escape PECOS detection are caused primarily by the following three factors: (1) data corruption (e.g., the lease time that is loaded into a register is zero seconds) and (2) application takes an incorrect rather than an illegal branch.

The fail-silence violation proportions are higher for the error models that target the store instructions (ADDOS, DATAOS). This is intuitive because a corrupted store instruction writes a wrong data value and bypasses most checks, such as semantic checks. This observation points to the necessity of a data protection scheme, such as data audits [HAU85] to reduce fail-silence violations for the ADDOS and DATAOS error models.

The cases of fail-silence violation were manually explored after the automated analysis was over. Some representative examples of the fail-silence violation of the DHCP server include:

1. The server does not get the applicable record for the client host generated after phase 1 of the protocol and therefore sends a DHCPNAK in the second phase.
2. The server gets DHCPDISCOVER but does not respond with DHCPOFFER because it believes erroneously that there are no free leases on the subnet.
3. The server's response to the client causes the client to make a wrong protocol transition. As a result, the client does not try to renew its lease but continues to use the old lease and the server allows the client to do so.

5.2.3 Effect of PECOS on Process Hang and Process Abort

The error injection results show that incidences of process hang and process abort (i.e., semantic check detection) are also reduced by PECOS instrumentation. The detection latency using application specific data checks has been shown to be at least an order of magnitude higher than the detection latency using control signatures [KAN96]. Using PECOS could potentially eliminate the need for this type of checks and, consequently reduce the overall detection latency.

5.2.4 Analysis of No Error Cases (Error Activated – Not Manifested)

We also investigated the fact that for the baseline DHCP server, more than half the activated errors do not cause any problems in application execution. One possible explanation is that for the particular experimental system, the following two conditions hold quite often:

- Faults/errors are injected to don't care bits in the instruction words (e.g., 8 bits in every arithmetic *add* and *sub* instruction in SPARC).
- The application logic is such that the change of condition in a conditional branch instruction does not affect the control flow (e.g., a register has value 5, and the opcode changes from *branch-on-greater-than-zero* to *branch-on-greater-than-equal-to-zero*).

For the DATAOS error model only 1.8% of activated errors do not impact the application. This shows that corruption of data usually results in application crash or fail-silence violation and is another strong indication of the need for efficient protection against data errors.

Finally, Table 6 shows that PECOS reduces the number of *no error* cases (i.e., execution of the erroneous instruction did not cause any error). This indicates that PECOS detects cases where the error would otherwise not impact the application. Since PECOS is a pre-emptive technique, it is expected to suffer from the problem of false alarms. It should be noted that some of non-manifested errors can stay as latent and may be activated at the later time.

5.3 Performance Measurements

The performance measurement for the DHCP application with and without PECOS instrumentation was performed from the server side as well as the client side. The server side measurement gives the time between the server receiving a request from the client and sending out a response. The time on the client side includes this time plus the time spent in the network. The performance measurements are presented in Table 7. The measurements are given separately for the two main phases of the protocol (see Section 4.1 for DHCP details).

Phase 1 <i>DHCPDISCOVER → DHCPOFFER</i>		Phase 2 <i>DHCP REQUEST → DHCPACK/DHCPNAK</i>	
Server overhead	Client overhead	Server overhead	Client overhead
Exhaustive instrumentation of DHCP server			
25.03%	15.34%	29.91%	25.17%
Selective instrumentation of DHCP server (only the core protocol engine)			
5.20%	10.92%	13.80%	18.07%

Table 7: Performance Measurements for DHCP Instrumented with PECOS

Noting that PECOS allows selective instrumentation of the application, performance measurements were made once with the entire DHCP server instrumented, and once with only the core protocol engine instrumented. Server overhead is the percentage overhead seen at the server and the client overhead is that seen from the client side and includes the network overhead. The entire DHCP server consists of 30 source files (written in C) and includes all the support functions, like receive packet from the network, used by the core protocol engine. In terms of lines of source code, the DHCP core protocol engine constitutes 11% of the entire DHCP server code. In terms of size of object code, the DHCP core protocol engine is 7.2%. However, the proportion of time spent in the core protocol engine is much greater than the relative code size of the protocol engine¹⁴. The results show overheads in the range of 5-20% if only the core protocol engine of the server is instrumented.

To get a better understanding of the percentage of the entire code that is instrumented with PECOS Assertion Blocks, statistics about subroutine calls in the application code are presented in Table 8. It may be recalled that the current PECOS implementation cannot protect the calls made to subroutines in dynamic libraries. As a result 64% of the calls in the DHCP server code can be protected. It is found by analysis of the error injection logs that some of the cases of lack of coverage of the PECOS technique are due to this deficiency. From Table 8, it is seen that 81% of the protected calls span two files. This indicates the importance of the ability of PECOS to instrument calls and returns that span files.

Number of calls in DHCP server	1443
Number of protected calls	906 (64%)
Number of calls that span two files	735 (81%)

Table 8: Statistics on Subroutine Calls in DHCP

¹⁴ However, there is no easy way to determine the proportion of time spent in one particular file.

5.4 Code Overhead

If an assertion block is inserted for every CFI, then the memory overhead of PECOS will be fairly high. Memory overhead is defined as the increase in the size of the application text segment. A highly data-intensive application will have larger block sizes than a control-intensive application and, therefore, lower memory overhead. Given that the size of a branch free interval is n assembly instructions and the (average) size of an Assertion Block is a assembly instructions, then the memory overhead is:

$$C = a / n * 100\%$$

A typical value of a is 10 to 15 assembly instructions, and n is 10 to 20 assembly instructions. This gives an overhead of 50%-150%. The value of n is dependent on the type of application, being less for a control-intensive application (like DHCP) and more for a data-centric one. Memory overhead of existing hardware and software control flow checking techniques has been reported to range from 6-135%, though the lower values are almost certainly for clustering of several branch-free intervals in a block, thereby lowering the coverage. For a software scheme, the program storage overhead is likely to be above 100%, since the checking code as well as reference signatures have to be embedded. PECOS allows the application designer the flexibility of instrumenting selective parts of the application. This protects only the required critical portions of the software and incurs only the necessary overhead. It worth noting that some of previous studies on control flow error detection techniques do not provide overhead information; some simply provide a figure; a few provide comprehensive estimates.

5.5 Impact of Errors on Checking Code

A problem with previous evaluations of control flow error detection techniques is that it was not clear whether the checking code itself had been injected with errors. In the current study, in directed control flow injections, PECOS instructions are automatically excluded, since they do not include any control flow instruction. However, for the random injections to the text segment of the application, instructions of the PECOS Assertion Blocks can also be injected.

To determine whether error detection is triggered if the Assertion Block is injected, and whether the PECOS instructions contribute to the fail-silence violation, an additional error injection campaign was performed where only the PECOS Assertion Block instructions were injected. The results are shown in Table 9. The results show that the PECOS Assertion Blocks do not cause any fail-silence violation. Consequently it can be concluded that the PECOS instrumentation does not add any extra vulnerability to the application. Approximately 88% ($51.9 / (51.9 + 6.9 + 0.5)$) of the errors that were injected into the Assertion Blocks and got manifested triggered PECOS detection.

Consequence	Inject CFI without PECOS	Inject Assertion Block
Error Activated but Not Manifested	36.6%	40.7%
PECOS Detection	N/A	51.9%
System Detection	53.9%	6.9%
Server Hang	3.4%	0.5%
Semantic Check Detection	0.9%	0.0%
Fail-Silence Violation	5.2%	0.0%

Table 9: Comparison of Directed Injections without PECOS and into the PECOS Assertion Blocks

5.6 Comparison with Our Implementation of ECCA Technique

Our preliminary investigation of non-preemptive control flow monitoring scheme showed that the percentage of system detection is very high while the checking scheme demonstrates very low coverage. This was because in most of the cases, the application process crashes before the control-flow checking mechanism kicks off. This conclusion was in contrary to the outcomes presented by Alkhalifa *et al.* [ALK99], where they discuss and evaluate software-based, non pre-emptive control signature scheme called ECCA (**E**nhanced **C**ontrol **C**hecking with **A**ssertions).

To resolve this ambiguity we decided to recreate the experiments from [ALK99]. Towards this end, a parser developed in Duke University was used to instrument the application with ECCA assertion blocks¹⁵. The same application, namely the *espresso* from the SPECInt integer benchmark suite is used as the target of the evaluation. Identical error models (as presented in Table 1) are injected (using NFTAPE) to conduct the study.

Table 10 presents error injection results obtained for four error models—ADDIF, DATAIF, DATAOF, and DATAInF. 500 errors from each error model are injected into the target application, once for the baseline case (i.e., the application without ECCA signatures) and once for the ECCA-instrumented application. For each combination of the application and error model, errors are injected (1) randomly in the text segment of the application and (2) directed at the control flow instructions. Thus, a total of 16 ($4 * 2 * 2$) campaigns are defined, and the total number of error injection runs is 8,000. The cases where the error is not activated are discarded. The error activation rate is higher for the uninstrumented code (from 46.2% to 57.0%) than for the ECCA instrumented code (from 30.4% to 31.4%). The possible explanation for this is that the ECCA instrumentation results in expanding the text segment of the application with additional code not all of which is executed during the normal run. To identify the fail-silence violations for the espresso application, the application's output is dumped to a file and compared against a golden output file. A mismatch indicates a fail-silence violation.

¹⁵ The ECCA instrumentation tool was developed by Dongyan Chen working under the supervision of Prof. Kishor Trivedi in the ECE Department of Duke University. The evaluation of ECCA was performed by the students of the ECE442 (advance class on reliable systems and networks) in Fall 2000 at the University of Illinois at Urbana-Champaign taught by one of the co-authors, Prof. R. K. Iyer.

Error Model	Outcome Categories	Injection Target <i>Any Instruction in Instruction Stream</i>		Injection Target <i>Control Flow Instructions</i>	
		Baseline	With ECCA	Baseline	With ECCA
ADDIF	NE	35.1%	39.7%	27.7%	17.5%
	ED	N/A	32.1%	N/A	16.5%
	SD	58.0%	25.6%	66.3%	58.2%
	SH	0.9%	2.6%	0.7%	5.1%
	SC	5.6%	0.0%	4.5%	1.3%
	FV	0.4%	0.0%	0.8%	1.4%
DATAIF	NE	44.0%	35.1%	36.5	24.4%
	ED	N/A	22.1%	N/A	12.8%
	SD	48.8%	41.6%	58.7%	57.7%
	SH	1.5%	1.2%	1.1%	3.8%
	SC	4.9%	0.0%	1.1%	1.3%
	FV	0.8%	0.0%	2.6%	0.0%
DATAOF	NE	55.1%	52.5%	46.9%	18.7%
	ED	N/A	16.7%	N/A	20.0%
	SD	39.8%	29.5%	48.3%	53.8%
	SH	1.2%	1.3%	0.4%	6.2%
	SC	3.5%	0.0%	2.9%	1.3%
	FV	0.4%	0.0%	1.5%	0.0%
DATAInF	NE	53.1%	44.2%	40.4%	22.1%
	ED	N/A	19.0%	N/A	15.6%
	SD	41.3%	34.2%	55.4%	58.4%
	SH	2.4%	1.3%	2.8%	3.9%
	SC	2.0%	1.3%	0.7%	0.0%
	FV	1.2%	0.0%	0.7%	0.0%

Table 10: Results of Evaluation of ECCA Applied to *espresso* Benchmark Program

The important conclusions from our experiments are:

- ECCA detects 22% of the activated errors, with the detection rate coming down (to 16%) for directed control flow injections. The reduction in ECCA detection for control flow errors can be explained by the fact that for random injections, the ECCA checking code is also injected and corruption to the checking code is detected by ECCA.
- For directed injections to control flow instructions, ECCA does not bring down the proportion of system detections, while it reduces system detection cases by about 14% for the random injections.
- The percentage of fail-silence violations is quite small for the baseline application (0.7% for random injections, 1.4% for directed injections). The fail-silence violations are either reduced or eliminated by ECCA.
- The proportion of not manifested errors is reduced by 44% from the baseline to the ECCA-instrumented case. This indicates that ECCA raises false alarms by detecting errors that would not impact the application.

Although we made an attempt to follow the approach presented in Alkhalifa *et al.* [ALK99] (using the same application and the same error models) the results obtained in our study are different from those reported there. In the original study, ECCA detection ranged from 48% to 56% and system detection ranged from 32% to 47%. We speculate that the differences between the published numbers and results from our experiments are because significant number of errors injected in the original study directly hit the ECCA assertion blocks and these errors are always detected by ECCA. Nevertheless, our study shows that large number of application crashes cannot be avoided using non-preemptive control-flow checking.

6 Conclusions

This paper presents a preemptive control flow error detection technique called PECOS. The software-based scheme embeds assertions in the application's assembly code to preemptively detect any illegal or incorrect control flow path and perform a graceful termination of the offending thread(s). The runtime extension of PECOS allows protection of control flow structures that are determined at runtime.

A large-sized client-server application, Dynamic Host Configuration Protocol (DHCP) application is used as a target to evaluate effectiveness of PECOS. The assessment of PECOS coverage and relative improvements due to PECOS was performed through software-based error injections to the baseline and to the instrumented application. The results showed significant reductions in fail-silence violations, system detections, and process hangs due to the instrumentation with PECOS. For specific control flow errors, PECOS was shown to catch about 87% of the activated control flow errors with maximum performance penalty of 29%.

In the current software implementation of PECOS, the Assertion Block contains the valid target addresses embedded in it. It also contains code to calculate the runtime target address. The application of PECOS to hardware is currently being investigated. With hardware support, the determination of the runtime target address is done in hardware, with the valid target addresses still embedded in the Assertion Block. Figure 9 gives three possible points in the execution path where PECOS checking can be implemented. The current support for PECOS is shown in Figure 9(a). Figure 9(b) shows a hardware addition that snoops on the external address bus and data bus and computes the runtime target address. Figure 9(c) shows a hardware addition in which the monitor is built inside the processor and snoops on the internal data and address bus. The closer to the execution unit the signature checking is done, the larger is the error set that can be tolerated by PECOS. The configuration in Figure 9(b) can detect errors during transmission on the external buses, while Figure 9(c) can detect errors internal to the processor.

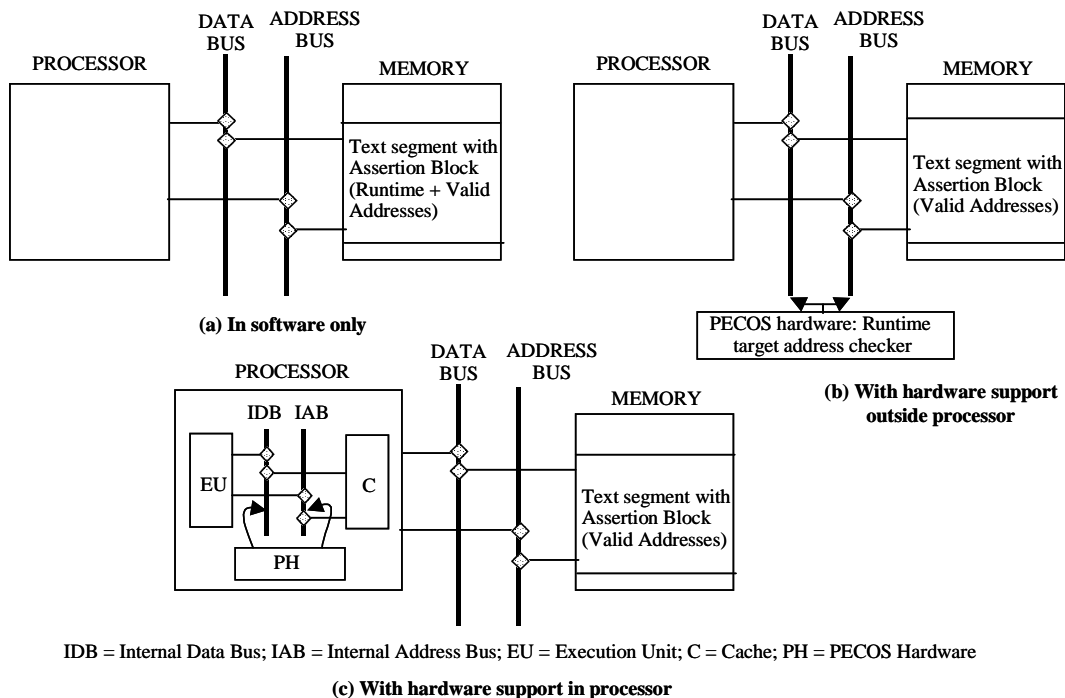


Figure 9: Different Levels of PECOS

Acknowledgements

This work was supported in part by JPL REE program and in part by Motorola Inc. We are also thankful to W. Gu, P. Jones, and S. Narayanaswamy for developing error injectors for different error models used in the evaluation of PECOS and ECCA. We thank the students of ECE 442 of Fall 2000 for the evaluation of ECCA on *espresso*.

References

- [ALK99] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Design and Evaluation of System-Level Checks for on-line Control Flow Error Detection," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 6, pp. 627-641, June 1999.
- [CHA98] S. Chandra, P. M. Chen, "How Fail-stop Are Faulty Programs?," *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (FTCS-28)*, 1998, pp. 240-249.
- [CHI89] R. Chillarege, R. K. Iyer, "An Experimental Study of Memory Fault Latency," *IEEE Transactions on Computers*, Volume: 38 Issue: 6, pp. 869-874, June 1989.
- [HAU85] G. Haugk, F. M. Lax, R. D. Royer, J.R. Williams. *The 5ESS Switching System: Maintenance Capabilities. AT&T Technical Journal*, vol. 64, no. 6, 1985. pp. 1385-1416.
- [ISC00] Internet Software Consortium (ISC) Dynamic Host Configuration Protocol (DHCP), URL: <http://www.isc.org/products/DHCP>.
- [IYE86a] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh, "Measurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Transactions on Computer Systems*, vol.4, no.3, pp.214-237, August, 1986.
- [IYE86b] R. K. Iyer, D. J. Rossetti, "A Measurement-Based Model for Workload Dependence of CPU Errors," *IEEE Trans. on Computers*, vol.C-35, no.6, June 1986.

- [KAN95] G. Kanawati, N. Kanawati, J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. on Computers*, pp. 248-260, February 1995.
- [KAN96] G.A. Kanawati, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Evaluation of Integrated System-Level Checks for on-line Error Detection," *Proc. IEEE Int'l Symp. Parallel and Distributed Systems*, pp. 292-301, Sept. 1996.
- [LAL85] P. K. Lala, "Fault Tolerant and Fault Testable Hardware Design," Prentice Hall International, New York, 1985.
- [LAP92] J.C. Laprie (ed.), *Dependability: Basic Concepts and Terminology*, Dependable Computing and Fault-Tolerant Systems series, Vol.5, Springer-Verlag, 1992.
- [MAD91] H. Madeira, J. G. Silva, "On-Line Signature Learning and Checking," *Proc. 2nd IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-2)*, pp. 170-177, Feb. 1991.
- [MAD94] H. Madeira, J. G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking," *Proc. 24th International Symp. on Fault-Tolerant Computing (FTCS-24)*, pp. 350-359, July, 1994.
- [MAH88] A. Mahmood, E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Trans. on Computers*, Vol. 37, No. 2, pp. 160-174, Feb. 1988.
- [MIC91] T. Michel, R. Leveugle, G. Saucier, "A New Approach to Control Flow Checking without Program Modification," *Proc. 21st International Symp. on Fault-Tolerant Computing (FTCS-21)*, pp. 334-341, 1991.
- [MIR92] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, "Two Software Techniques for On-Line Error Detection," *Proc. 22nd International Symp. on Fault-Tolerant Computing (FTCS-22)*, pp. 328-335, July, 1992.
- [MIR95] G. Miremadi, J. Ohlsson, M. Rimen, J. Karlsson, "Use of Time and Address Signatures for Control Flow Checking," *Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5)*, pp. 113-124, 1995.
- [NAM82] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation," *Digest 1982 Intl. Test Conf.*, pp. 461-468, November 1982.
- [NAM83] M. Namjoo, "CEREBRUS-16: An Architecture for a General Purpose Watchdog Processor," *Proc. 13th Annual Int'l Symp. On Fault-Tolerant Computing (FTCS-13)*, pp. 216-219, June 1983.
- [OHL92] J. Ohlsson, M. Rimen, U. Gunneflo, "A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog," *Proc. 22nd Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-22)*, pp. 316-325, 1991.
- [RFC97] R. Droms, "Dynamic Host Configuration Protocol," *Request for Comments RFC-2131*, Bucknell University, March 1997.
- [SCH86] M.A. Schuette, J.P. Shen, D.P. Siewiorek, Y.X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," *Proc. 16th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-16)*, pp. 138-143, July 1986.
- [SCH87] M.A. Schuette, J.P. Shen, "Processor Control Flow Monitoring Using Signed Instruction Streams," *IEEE Transactions on Computers*, Vol. C-36, No. 3, pp. 264-276, March 1987.
- [SIE98] D. P. Siewiorek, R. S. Swarz, "Reliable Computer Systems: Design and Evaluation," A.K. Peters, Natick, Massachusetts, Chapter 2, pp. 22-77.

- [STO00] D.T. Stott, B. Floering, Z. Kalbarczyk, R.K. Iyer, "Dependability Assessment in Distributed Systems with Lightweight Fault Injectors in NFTAPE," Proc. IEEE Int'l Computer Performance and Dependability Symp. (IPDS'2K), pp.91-100, March 2000.
- [THA97] Anshuman Thakur, "Measurement and Analysis of Failures in Computer Systems", Master's Thesis, Advisor: R.K. Iyer, University of Illinois, UILU-ENG-97, September, 1997.
- [TSA83] M.M. Tsao, D.P. Siewiorek, "Trend Analysis on System Error Files," Proc. 13th Int. Symp. Fault-Tolerant Computing, Italy, June 1983, pp.116-119.
- [UPA94] Shambhu Upadhyaya, Bina Ramamurthy, "Concurrent Process Monitoring with No Reference Signatures," IEEE Transactions on Computers, vol. 43 no. 4, pp. 475-480, April 1994.
- [YAU80] S.S. Yau, F-Ch. Chen, "An Approach to Concurrent Control Flow Checking," IEEE Trans. on Software Engineering, Vol. SE-6, No. 2, pp. 126-137, 1980.
- [YOU92] L.T. Young, R. K. Iyer, K.K. Goswami, C. Alonso, "A Hybrid Monitor Assisted Fault Injection Environment," Proc. Third IFIP Working Conference on Dependable Computing for Critical Applications (DCCA), Italy, 1992.
- [YOU91] L.T. Young, R. K. Iyer, "Error Latency Measurements in Symbolic Architectures," Proc. AIAA Computing in Aerospace 8, Baltimore, Maryland, 1991.
- [WIL90] K. Wilken, J.P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Errors," IEEE Transactions on Computer Aided Design, pp. 629-641, June 1990.

Appendix

Table 11 presents a survey of representative hardware and software techniques for control flow error detection. The first five techniques presented in Table 11 require additional hardware, while the last two are purely software-based approaches. The table mentions the results of evaluations of the available techniques. For the results in Table 11, an attempt has been made to remove the cases in which the detection is done by something other than the technique under consideration, e.g., system detection which crashes the application process, is removed from the coverage number. For the hardware and software resources required, a *HI* next to the entry denotes that the complexity of the particular resource is deemed to be quite high.

Technique	Outline of Scheme	Workload; Performance Metrics (Detection coverage, Memory and Execution Time Overhead); Resources required	Comments ¹⁶
Signed Instruction Stream (SIS) [SCH86]	<p>Operates on assembly language code.</p> <p>The signature is embedded in the application instruction stream.</p> <p>An optimization called <i>Branch Address Hashing</i> reduces memory overhead by eliminating reference signatures at branch points.</p>	<p>Quicksort, String search, Matrix transpose on MC-68000</p> <p>Coverage: 36%⁽¹⁾ Mem overhead⁽²⁾: 6-15%⁽³⁾ Perf overhead: 6-14%</p> <p>Hardware: Watchdog processor with cyclic code signature generator & branch hashing capability</p> <p>Software: Modified assembler and loader</p>	<p>Needs parser to embed signatures.</p> <p>Needs runtime support to rehash branch addresses and synchronize it such that the rehashing happens before the target address calculation.</p> <p>Main contribution of high overall system coverage comes from system detection.</p>
Path Signature Analysis (PSA) [NAM82]	<p>Signatures are computed for paths (sequences of nodes) rather than a single node (a node is a branch free interval).</p> <p>Program graph is decomposed into path sets, with all paths in a path set starting and ending at the same node. All paths in a path set have the same signature.</p> <p>To ensure all paths in a path set have same signature, justifying signatures have to be inserted in some nodes.</p> <p>Combination of reference and justifying signatures optimizes total number of signatures.</p>	<p>50 unspecified programs of different types from data manipulation to I/O control routines on MC-68000</p> <p>Coverage: Not Available Mem overhead: 12-35% Perf overhead: Not Available</p> <p>Hardware: Separate tag memory, Watchdog processor with branch detecting circuit, comparator and control unit for generating control signals to the main processor</p> <p>Software: Modified assembler to identify and match corresponding branch entry and exit points (<i>HI</i>)⁽⁴⁾</p>	<p>Requires complicated parser for finding path sets.</p> <p>Error detection latency can be significant since detection is done at the end of the path.</p> <p>Control flow errors that cause erroneous sequence of paths to be executed are not detected.</p>

¹⁶ Comments are either based on explicit assumptions in the referred papers or express our understanding of a given technique. Consequently there is always a room for slightly different interpretation.

<p>On-line Signature Learning and Checking (OSLC) [MAD91]</p>	<p>Application is decomposed into several sections, each having a certain number (typically 64 or 128) of branch-free intervals (BFI).</p> <p>A signature generator and a watchdog processor are required. The signature generator signals beginning and end of a BFI to the checker and generates the runtime signature of the block.</p> <p>On receiving “Exit-From-Block” signal from the signature generator, the checker checks the runtime signature against signatures of all blocks in the same section.</p>	<p>Pseudo-random number generator, string search, bit manipulation, quick sort, prime number generator on Z-80</p> <p>Coverage: 86.3%⁽⁵⁾ Mem overhead: Not Avail. Perf overhead: Not Avail.</p> <p>Hardware: A Signature Generator with Parallel Linear Feedback Shift Register (PLFSR) tied to main processor (<i>HI</i>), a simple two-stage pipeline Checker</p> <p>Software: Exhaustive tester (<i>HI</i>)</p>	<p>Addresses accessed by the application processor must be visible on the external bus.</p> <p>Mostly control bit errors are detected, few control flow errors are detected⁽⁶⁾.</p> <p>Prior to running, exhaustive path activation to generate golden signatures for each block in a segment is required.</p> <p>Synchronization required between the application processor, signature generator, and checker. (See explanation below)</p>
<p>Time-Time-Address Signature Checking [MIR95]</p>	<p>Program is decomposed into Branch Free Blocks [BFB] (protected branch free intervals) and Partition Blocks [PB] (branch instruction and unprotected branch free intervals). Special instructions are embedded at the beginning and end of each BFB to signal to an external watchdog processor.</p> <p>The watchdog starts a timer on getting notification of beginning of BFB or PB. If notification of end of BFB or PB is not received before timeout expires, an error is detected.</p> <p>Notification of address and block size are made to the watchdog at the beginning of a BFB. At block exit, watchdog checks that exit is made at (start address + size).</p>	<p>Linked list manipulation, Matrix manipulation, Quicksort on MC6809E (8-bit CPU)</p> <p>Coverage: 23.9%⁽⁷⁾ (heavy ion radiation method) 48.6% (power system disturbance method) Mem overhead: 35-39% Perf overhead: 25-30%</p> <p>Hardware: Timer (<i>HI</i>), Watchdog processor</p> <p>Software: Software to decompose application into BFB & PB, and embed signature instructions</p>	<p>Difficult to determine time bounds for the watchdog.</p> <p>Sending frequent signals to the external watchdog may result in performance degradation.</p>
<p>Concurrent Process Monitoring with no Reference Signatures [UPA94]</p>	<p>A known signature function (like modulo-2 sum) is applied to the instruction stream at compilation time. When the accumulated signature forms an m-out-of-n code or a branch point is reached, the instruction is tagged.</p> <p>At runtime, a watchdog verifies that at a tagged instruction, an m-out-of-n code has been accumulated.</p> <p>No reference signatures are required leading to memory overhead reduction.</p>	<p>Possibly no implementation exists</p> <p>Coverage, mem overhead, perf overhead: Not Available</p> <p>Hardware: A signature generator, an m-out-of-n checker, branch detector</p> <p>Software: Software to indicate checkpoints where code word needs to be checked and tag memory</p>	<p>One extra word per branch is required to force the accumulated signature to become an m-out-of-n code.</p> <p>Tagging memory may require allocating an extra memory word.</p> <p>Better at detecting control bit errors, than control flow errors.</p>

Block Signature Self Checking (BSSC) [MIR92]	<p>The program is divided into blocks and each block is assigned a signature, as in the standard approach. The signature is the address of the first instruction in the basic block.</p> <p>A subroutine call inserted at the beginning of the block stores the block signature in a static variable.</p> <p>A call instruction at the end of the block fetches the signature from the variable and compares it to an embedded signature following it. A mismatch signals an error.</p>	<p>Linked list manipulation, Quicksort, Matrix manipulation on MC6809 (8 bit CPU)</p> <p>Coverage: 15-22%⁽⁸⁾ Mem overhead: 23-35% Perf overhead: 88-127%</p> <p>Hardware: None</p> <p>Software: Software to identify blocks, assign calls and signatures</p>	<p>The assertions introduce control flow instructions of their own, which are additional sources of vulnerability.</p> <p>The technique assumes absolute addresses for the start of the basic block. This will preclude using it for relocatable code.</p> <p>May not detect control flow error that causes blocks to be executed in incorrect sequence.</p>
Enhanced Control-flow Checking with Assertions (ECCA) [ALK99]	<p>Branch-free intervals in a high-level language (C for their implementation) are identified. The entry and the exit points of the intervals are fortified through assertions inserted in the instruction stream.</p> <p>Before running, the program is analyzed, and two variables—Branch Free Interval Identifier (BID) and Next—are assigned to each interval.</p> <p>In case of a control flow error, the BID computation will cause a divide-by-zero error leading to detection.</p>	<p>2 SPEC Int92 benchmarks</p> <p>Coverage: 18.4%-37.5% (022.li) 27.8%-73.0% (espresso) (depends on error model) Mem overhead: Not avail. Perf overhead: Not avail.</p> <p>Hardware: None</p> <p>Software: A C-language lexer, filter and parser, signature generator (<i>HI</i>)</p>	<p>A parser for high-level code can be complex because of larger variations in code structure at the high-level language.</p> <p>For large programs overflow of NEXT variable value may occur necessitating a reset to an initial value, which decreases coverage because of aliasing.</p>

Table 11: Survey of Representative Control Flow Error Detection Techniques

The first five techniques are hardware-based, while the last two are software-based.

Index	Explanation
1	Together with application crash, the coverage was 82%.
2	Memory overhead denotes the increase in the text segment size for the application.
3	The memory and performance overhead is in a range because it depends on the application, since the size of the blocks depends on the application.
4	<i>HI</i> denotes that the complexity of the particular resource is considered quite high.
5	Together with errors of duration longer than 1 cycle, the coverage was 93.1%. The system detection cannot be separated from the given results.
6	A control bit error is an error in the instruction word in a block of instructions. A control flow error is an error in the instruction stream that causes blocks to be executed in an incorrect sequence, or instructions within a block to be executed in an incorrect sequence. A control bit error may lead to a control flow error if an instruction like jump is affected. One may protect against control bit errors without protecting against control flow errors because register errors or errors internal to the processor can also give rise to control flow faults.
7	Together with 4 other detection techniques (like a separate watchdog timer to detect timeouts), the coverage was 98%.
8	For only control flow errors, coverage is 78%.

Table 12: Explanation of Terms in Table 11