

November 2000

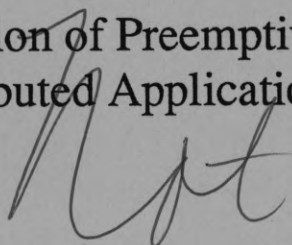
UILU-ENG-00-2213  
CRHC-00-04

---

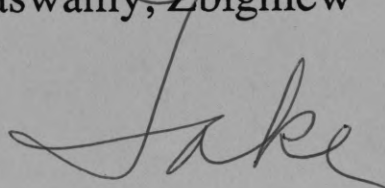
University of Illinois at Urbana-Champaign



Design and Evaluation of Preemptive Control Signature  
Checking for Distributed Applications

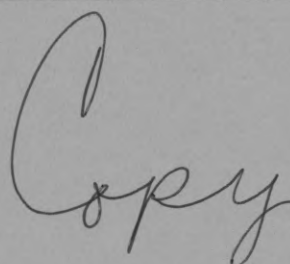
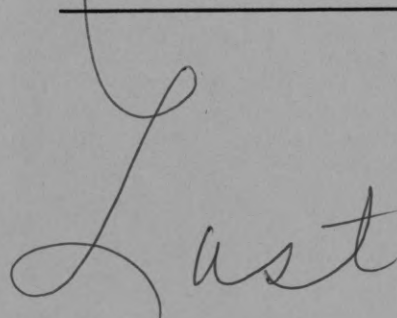


Saurabh Bagchi, Swetha Narayanaswamy, Zbigniew  
Kalbarczyk, and Ravishankar Iyer



Coordinated Science Laboratory  
1308 West Main Street, Urbana, IL 61801

---



REPORT DOCUMENTATION PAGE			Form Approved OMB NO. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comment regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 2000	3. REPORT TYPE AND DATES COVERED		
4. TITLE AND SUBTITLE Design and Evaluation of Preemptive Control Signature Checking for Distributed Applications			5. FUNDING NUMBERS	
6. AUTHOR(S) S. Bagchi, S. Narayanaswamy, Z. Kalbarczyk, and R. K. Iyer				
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES) Coordinated Science Laboratory University of Illinois 1308 W. Main Street Urbana, IL 61801			8. PERFORMING ORGANIZATION REPORT NUMBER UIIU-ENG-00-2213 CRHC-00-04	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT  Approved for public release; distribution unlimited.			12 b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)  The paper presents the design of a pre-emptive control flow signature (PECOS) technique used for on-line detection of control flow errors. The technique uses assertions that can be embedded in the assembly language code and which are triggered by control flow instructions in the code. PECOS is evaluated through software-based fault injection - both directed control flow injections and random injections into the text segment of the running application. For a distributed workload consisting of the Dynamic Host Configuration Protocol (DHCP), application of the technique reduces the incidence of fail-silence violations (from 7.6% to 0.4%), reduces the cases of process crash (from 49.1% to 13.6%) and eliminates the need for program semantic based data checks which require program knowledge and are intrusive in nature. Performance studies conducted show marginal degradation (about 1%) because of the signatures. However, a higher initial overhead is incurred (about 15%) at protocol startup. Finally, the signature is embedded in a Software Implemented Fault Tolerance (SIFT) middleware called Chameleon and is used to provide detection to applications running on the middleware. Results are provided from this environment running DHCP.				
14. SUBJECT TERMS control flow signature, preemptive checking, fail-silence, detection coverage, software fault injection			15. NUMBER OF PAGES 24	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OR REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	



# Design and Evaluation of Preemptive Control Signature Checking for Distributed Applications

Saurabh Bagchi, Swetha Narayanaswamy, Zbigniew Kalbarczyk, Ravishankar K. Iyer

Center for Reliable and High-Performance Computing

University of Illinois at Urbana-Champaign

1308 W. Main St., Urbana, IL 61801, USA

[bagchi, swetha\_n, kalbar, iyer]@crhc.uiuc.edu

## Abstract

*The paper presents the design and evaluation of PECOS, a preemptive control signature (PECOS) technique to be used for on-line detection of control flow errors. The technique uses assertions that can be embedded in the assembly language code and which are triggered by control flow instructions in the code. PECOS is evaluated through software-based fault injection – both directed control flow injections and random injections into the text segment of the running application. For a distributed workload consisting of the Dynamic Host Configuration Protocol (DHCP), application of the technique reduces the incidence of fail-silence violations (from 5.2% to 0.4%), reduces the cases of process crash (from 53.9% to 13.6%) and eliminates the need for program semantic based data checks which require program knowledge and are intrusive in nature. Performance studies show marginal degradation (about 1%, seen from the client side) because of the signatures. However, a higher initial overhead is incurred (about 15%, execution at the server side) at protocol startup. Finally, the signature is embedded in a Software Implemented Fault Tolerance (SIFT) middleware called Chameleon and is used to provide detection to applications running on the middleware. Results are provided from this environment running DHCP.*

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

# 1 Introduction

The faults seen by an application can be classified broadly as data faults and control faults. Data faults are faults that affect the values of data variables, registers, or memory locations used by the application. Control faults are faults that change the control flow of the application and can be defined to be any fault that causes a divergence from the sequence of program counter values seen during the fault-free execution of the application. In this paper, we focus on control faults as they can lead to data errors, process crashes, or fail-silence violations<sup>1</sup>. The reason for the interest is that control flow errors have been demonstrated to account for between 33% [OHL92] and 77% [SCH87] of all errors, depending on the fault assumption (e.g., register bit-flip fault or pin-level stuck-at fault). Moreover, results from the fault injection experiments presented in this paper indicate that for a control-intensive application, about a third of the activated faults in the text segment of the application cause control-flow errors. In distributed applications, e.g., our target application, the Dynamic Host Configuration Protocol (DHCP) application, fail-silence violations can have catastrophic effects by causing fault propagation. Hence, it is imperative to provide protection against control flow faults in order to build highly available applications or middleware.

In this paper, we propose a control-flow signature generation and checking technique, called *PECOS* (Preemptive Control Signatures) that provides detection against control flow faults occurring in the memory or during transmission from memory to the processor. At a higher level, the fault model being targeted is corruption to the control flow instructions of a program, or a software bug that causes a divergence from the correct (in the sense of a correct specification) control flow of the program. PECOS does not require access to application source code and is currently implemented in software, but is extensible to hardware and a hybrid of hardware-software.

Previous behavior-based control flow signature schemes [KAN96, ALK99] have relied heavily on system detection to achieve fail-silence. System detection is problematic because it causes the process to crash, and the application recovery time is correspondingly higher. Contrary to such schemes, PECOS is preemptive in nature and can therefore be triggered before the application process crashes due to system detection. The main benefits of PECOS are three-fold:

1. It reduces the incidence of process crashes and enables a graceful termination of a faulty process or thread.
2. It improves the fail-silence property of an application. A fail-silent application process either works correctly, or stops functioning (i.e., becomes silent) if an internal failure occurs [BRA96].

---

<sup>1</sup> An entity is said to be fail-silent if it either operates correctly or stops operating. A violation of this condition is called a fail-silence violation.



3. It minimizes error detection latency because it is triggered before the error manifests itself. If latency is considered as the lag between the error manifestation and the detection, then the latency is zero for PECOS.

Importantly, PECOS is demonstrated by applying it to a sizable workload: the Dynamic Host Configuration Protocol (DHCP) application running on the SPARC platform. Evaluations of previous schemes have been done on relatively simple, standalone applications running on simple processors [MAD91,MIR92]. In contrast, DHCP is a real-world application. It is a control-centric, distributed, client-server application widely used in networks with mobile clients who use it to request and be assigned IP addresses. PECOS is applied to a standard DHCP server as well as a DHCP server running under a distributed, reliable middleware called Chameleon [KAL99]. Software-based fault injection using NFTAPE [STO00] is used to evaluate the effectiveness of PECOS under both configurations. Both directed control flow faults and random faults to the text segment of the application process are injected. We believe that the evaluation of the unmodified DHCP application is valuable in itself because it gives insight into properties such as fail-silence and crash-proneness of a fairly representative client-server application. The results show that PECOS provides approximately a four-fold reduction in the incidence of process crash (from 53.9% to 13.6%) and a twenty-fold reduction in fail-silence violation (from 5.2% to 0.4%). It is also observed that PECOS does not throw any significant number of false alarms, i.e., it does not flag faults which would not manifest themselves in the baseline case, as errors.

This paper studies the software implementation of PECOS, but the same principle can be extended to hardware and the signature unit can be inserted at various points in the path from the memory to the processor core, for example, between the main memory and an off-chip L2 cache or between an off-chip L2 cache and an on-chip L1 cache. With the hardware extension, PECOS can tolerate faults from a large fault class – faults in the memory or cache or during transmission between different levels of the memory hierarchy.

The rest of the paper is organized as follows. Section 2 presents related work in the area of control flow error detection. Section 3 presents the principle, design, and assumptions of PECOS. Section 4 provides details of the SIFT middleware, the fault injection environment and the workload. Section 5 describes the experiments and presents and analyzes the results. Section 6 concludes the paper and provides ideas for future work.

## 2 Related Work

The field of control-flow checking has evolved over a period of more than 15 years. In most of related literature, various control-flow checking techniques have been proposed in hardware to detect processor faults. The techniques employ a watchdog (WD) processor to compute run-time signatures from the instruction stream and compare them with pre-computed golden signatures. A WD is a simple processor that monitors the memory accesses by snooping on the external address and data bus between the processor and memory. Mahmood [MAH88] presents a survey of the various techniques in hardware for detecting control flow errors. 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], while Autonomous signature monitoring (ASM) stores the signature in memory dedicated to the watchdog processor [MIC91].

Table 1 presents a survey of representative hardware-based techniques for control flow error detection. The techniques have been evaluated on small-sized standalone applications, and on fairly simple microprocessors, such as the Z80 [MAD91].

Name	Required Resource	Evaluation Method	Workload	Detection Coverage <sup>2</sup>	Overhead
Signed Instruction Stream (SIS) [SCH86]	WD; Parser to embed signatures and hash branch addresses	HW fault injector injecting external processor buses	Quicksort, String search, Matrix transpose	36% <sup>(i)</sup>	NA
Time-Time-Address Signature (TTA) [MIR95]	WD; SW to signal WD processor. This scheme requires information about time to execute application blocks	Heavy ion radiation (HIR); Power system disturbances (PSD)	Linked list operations, Quicksort, Matrix transpose	23.9%(HIR) <sup>(ii)</sup> 48.6%(PSD)	35-39% memory 25-30% execution time
Online Signature Learning & Checking (OSLC) [MAD91]	WD; Signature generator circuit attached to main processor. Requires exhaustive testing that sensitizes all paths.	Pin-level fault injection to Address/Data/Control pins	Random number generator, String search, Quicksort	86.3% <sup>(iii)</sup>	NA

**Table 1. Survey of representative hardware-based control flow error detection techniques.**

(i) Together with application crash, the coverage was 82%,

(ii) Together with 4 other detection techniques, the coverage was 98% (with 4 other detection techniques),

(iii) Together with faults of duration longer than 1 cycle, the coverage was 93.1%

Applying the hardware schemes to distributed environments suffers from two major limitations:

1. If multiple processes (or threads) execute on the main processor, then the memory access pattern observed by the WD will not correspond to the signature of any single process, and hence an error will be flagged in mistake. This is because of the different possible interleavings between the multiple processes being executed. The number of such patterns is difficult to enumerate.
2. The underlying assumption is that, under faults, runtime memory accesses observed by the WD will be different from the reference signature. Consequently, an error while an instruction resides in the cache of the main processor, for example, will not be caught even if it causes an application misbehavior. Since processors have increasingly larger cache sizes, such errors are no longer negligible.

To solve the problems with the hardware schemes, several control flow monitoring schemes have been proposed in software. The software techniques embed the checking code in the instruction stream and do away with a hardware WD processor. Miremadi *et al.* [MIR92] have proposed a scheme called Block Signature Self

<sup>2</sup> The coverage number is obtained after removing the effects of any other detection, including system detection and data checks.



Checking (BSSC). In BSSC, the program is divided into blocks and each block is assigned a signature, as in the standard approach. A subroutine call instruction is embedded before the basic block, which stores the runtime signature at a pre-determined memory location. The golden signature and another call instruction which compares the runtime and golden signatures are inserted at the end of the basic block. Kanawati *et al.* [KAN96] have developed a high-level, Control-flow Checking Approach (CCA) using assertions. In CCA, branch-free intervals in a high-level language (C for their implementation) are identified and the entry and exit points of the intervals are fortified through assertions inserted in the instruction stream. Before starting, the program is analyzed, and two variables – Branch Free Interval Identifier (BID) and Control Flow Identifier (CFID) – are assigned to each interval. The control flow checking is done by setting and checking the variables at the entry and exit of the branch-free intervals respectively. An improvement to CCA was proposed by Alkhalifa *et al.* [ALK99] in a scheme called Enhanced Control-flow Checking using Assertions (ECCA). It reduces the storage requirement from two variables per block to one. Also, contrary to CCA, the assertions do not contain any uncovered branches of their own.

The key outstanding issues with the software solutions proposed till date can be summarized as follows.

- None of the schemes do pre-emptive checking of the control path. In other words, the checking instruction is reached after the control block has been executed. Consequently, for a majority of cases (e.g., 58% in [KAN96]), system detection is triggered before the specific control signature checking technique and results in process crash. The assumption for such techniques is that process crash is desirable at best, and benign at worst and hence their evaluations have included the system detection in the error detection coverage. However, practical system designers typically do not view crash as a benign outcome and prefer to perform detection prior to the application process crash. For a large class of applications, e.g., real-time control applications, crash is an undesirable condition because it increases the recovery time.
- The evaluation often does not bring out how sensitive the dependability of the system is to errors in the checking code. It is sometimes not clear from the experimental results if faults were injected into the checking code. If the checking code introduces control flow instructions itself (as in BSSC and CCA), then the effectiveness of such a technique is dependent on protecting the additional instructions.
- An important metric for a detection technique is the case of false alarms, i.e., the application ran in the baseline mode and completed correctly in  $n$  cases, but when the detection technique was included, it flagged alarms, falsely, in  $m$  of those  $n$  cases.
- How generally applicable is the technique to off-the-shelf applications running on off-the-shelf processors? To date, the evaluation of the existing software control flow techniques have been done on fairly simple standalone applications. BSSC uses programs containing linked list operations, quicksort and matrix transpose, CCA uses matrix multiply and quicksort, while ECCA uses 2 SPECINT benchmarks. Fail-silence

violations become critical in distributed environments because of the possibility of fault propagation. To obtain a thorough understanding of the effectiveness of a technique, it is important to run substantial distributed workloads, both data-centric and control-centric.

In PECOS, the above four issues have been addressed. PECOS being a pre-emptive checking technique reduces the incidence of process crash. The evaluation of PECOS includes results from injections into the assertion blocks themselves. The evaluation quantifies the incidence of false alarms for PECOS-embedded applications. The evaluation is done on a fairly substantial application. Our workload application is DHCP, an application written by an unrelated organization (Internet Software Consortium), distributed in nature, fairly sizeable (approximately 20,000 lines of source code), and is considered a representative client-server application

### 3 PECOS: Principle, Design and Fault Model

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. In PECOS, the application is decomposed into blocks and signature instructions are embedded in the instruction stream of the application to be monitored. The blocks may be basic blocks in the traditional compiler sense of branch-free intervals, or a collection of several basic blocks grouped for optimization reasons (see Section 3.5). The decomposition is done at the level of assembly code. Each basic block is terminated by a *Control Flow Instruction* (CFI), which acts as a trigger. At a trigger point, an *Assertion Block* (AB) containing the signature instructions is inserted. The AB contains the set of valid target addresses the application may jump to, and 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. This is the key to pre-empting the system detection (through signals raised by the operating system, e.g., SIGBUS), and instead to have PECOS catch the problem, if any. The change to the application code due to inserting the PECOS instructions is depicted in Figure 1.

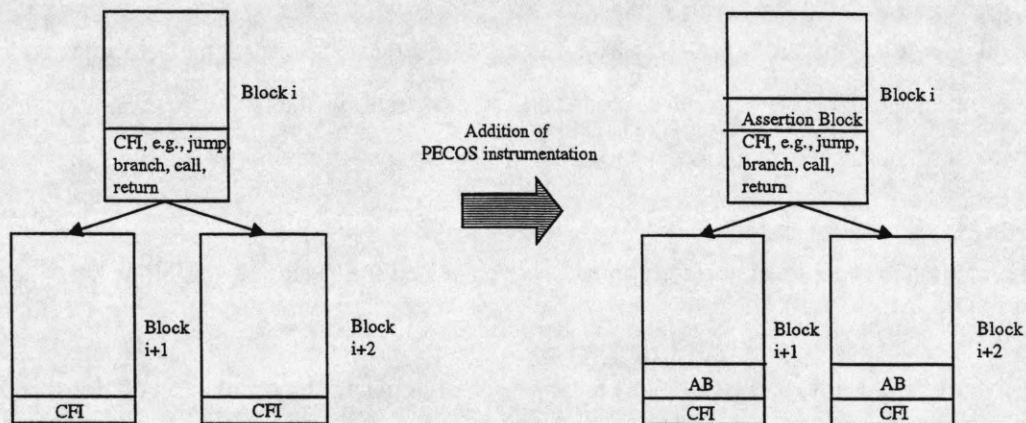
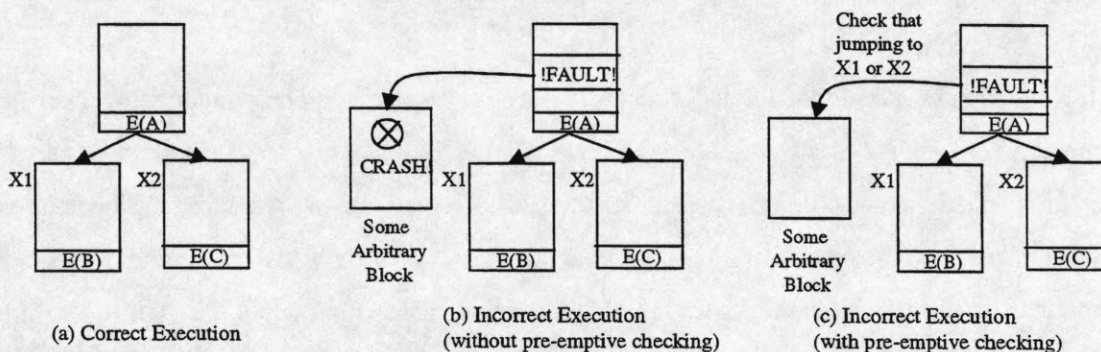


Figure 1. Change in code structure due to inserting PECOS assertion blocks (CFI = control flow instruction; AB = assertion block)



The problem with non-preemptive schemes and the solution proposed by PECOS are schematically represented in Figure 2. Figure 2(a) shows that in a non-preemptive scheme [KAN96,SHE83], the validity of the control path is checked at the end of execution of a block. Random fault injection into the text segment of the application process with such a non-preemptive scheme has shown that in a significant number of cases, this approach allows the process to crash [Figure 2(b)]. While crash failure is better than non-fail-silent behavior, it is unacceptable in several situations, e.g., real-time control applications. The approach in PECOS is to check the target location *before* the CFI is executed [Figure 2(c)]. Not allowing the process to crash is advantageous from the point of view of application recovery time. If the fault is diagnosed before the crash, then the application may be terminated gracefully, freeing up resources, for example. Also, if checkpointing is being used for rollback-recovery, the process's current state can be discarded and the process restarted from a previous checkpoint. In contrast, if the process has crashed, a new process needs to be created and the checkpoint loaded from stable storage to the memory of the new process. Process creation incurs the overhead of the kernel allocating a new entry in the process table and updating the structure's *inode* counter, file counter, etc.



**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.

### 3.1 PECOS: Software & Hardware Implementations

The current implementation of PECOS is completely in software. In the software implementation, the assertion block contains the valid target addresses embedded in it. It also contains code to calculate the runtime target address. The general principle of PECOS can be applied to hardware, also. With hardware support, the determination of the runtime target address is done in hardware, with the valid target addresses still embedded in the AB. Figure 3 gives three possible points in the execution path where PECOS checking can be implemented. The current support for PECOS is shown in Figure 3(a). Figure 3(b) shows a hardware addition that snoops on the external address bus and data bus and computes the runtime target address. Figure 3(c) shows a hardware addition where the monitor is built inside the processor and snoops on the internal data and address bus. The closer to the execution unit the signature verification is done, the larger is the fault set that can be tolerated by PECOS. The next section discusses the fault model tolerated by PECOS.

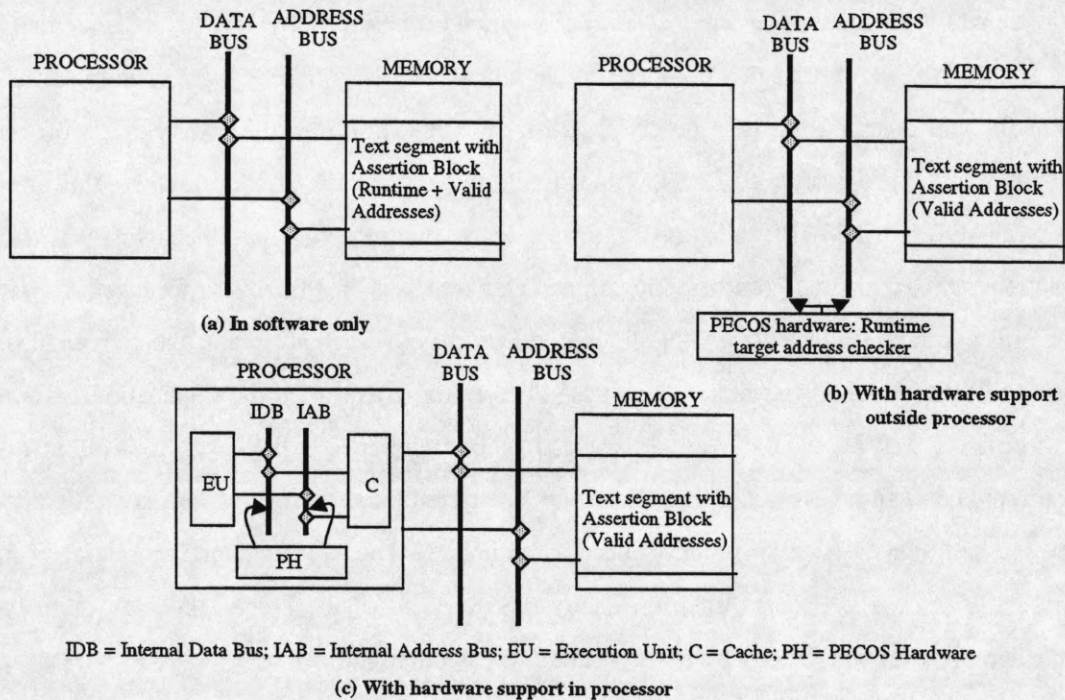


Figure 3. Different levels of PECOS.

### 3.2 PECOS Fault Model

The high-level fault model that PECOS tolerates is control flow faults. These faults may occur because of bit flips in memory, or because of software bugs. The control fault may be caused by flips at any level of the memory hierarchy (disk, main memory or cache) or during transmission between any two levels of the memory hierarchy. If the golden signature is generated from a high-level specification of the software, then PECOS can also detect software bugs. As mentioned before, assertion blocks contain information about the valid target addresses of a control flow instruction (CFI). It may be possible to generate the set of valid target addresses of an assembly-level CFI from the high-level specification of the software, knowing the mapping from the high-level control constructs (e.g., *if* and *while*) to the assembly level CFI structure. In this case, a bug in the software that implements the given specification (assumed correct) can also be caught by PECOS. The low-level bit flip fault model that can be tolerated by PECOS depends on where PECOS is implemented in the transmission path of the memory words in the instruction stream. For example, if it is implemented on the external bus between the processor and the memory, it can detect bit flips that affect the main memory or disk (generally, any component lower down in the memory hierarchy) or transmission errors between memory and processor or between disk and memory. Table 2 gives the different fault models and the support PECOS provides in its current implementation and in its future extensions in software and hardware.



Fault being protected against	Currently supported?	Can be supported in PECOS in software?	Can be supported in PECOS in hardware?
Software bug	NO	YES	YES
Memory fault	YES	YES	YES
External bus fault	NO	NO	YES
In-processor cache fault	NO	YES <sup>3</sup>	YES
Internal bus fault	NO	NO	YES

**Table 2. Fault model supported by PECOS – current implementation and future extensions.**

### 3.3 Design of Assertion Block

The assertion block (AB) that is inserted in the instruction stream for protecting the CFI must be designed carefully. It is desirable that the AB have the following properties:

1. The AB does not have any CFI of its own. Since we are trying to protect a CFI, it defeats the purpose to have the AB insert CFI(s) of its own. A naïve implementation of the AB would either implement the control decision of Figure 4 inline or make calls to a routine that implements the control decision. It is obvious that both will add CFIs in the assembly code – for extracting the valid addresses and making the if-then decision.

1. Determine the runtime target address [= X].
2. Extract the list of valid target addresses [= {X1,X2}].
3. *if (X != X1) && (X != X2) then*  
Flag Error

**Figure 4. High-level control decision in the assertion block.**

A naïve implementation of this structure introduces control flow instructions of its own and is undesirable.

2. Different CFIs have different numbers of possible valid addresses. A jump and a call each have one, a branch has two, while a return has multiple, equal to the number of places from which the subroutine may be called. The AB should be able to match with any of these numbers of valid target addresses.

The assertion block for PECOS does not introduce any additional CFIs. The technique for this is similar to one used by Alkhalifa *et al.* [ALK99]. A mathematical expression is formed, which is a function of X (the runtime target address) and X1, X2, ... (the valid target addresses determined through offline analysis) such that the expression evaluates to zero if X does not match either X1 or X2 or any of the other valid addresses. Then a division is done with this expression in the denominator. Thus, if there has been a mismatch, the division operation raises a Divide-By-Zero signal which is caught and handled. The handling involves looking up the valid target addresses for the current PC and checking if the actual runtime address matches any of those. If it does not, then it is concluded that the Divide-By-Zero exception was raised because of the control flow fault (as opposed to an application fault), and PECOS flags a control flow error detection. Example of AB for a call instruction that

<sup>3</sup> It can be tolerated in software at the expense of performance, by reducing the time the instruction word spends in the cache by flushing the cache and bringing the cache line in from the memory a few cycles before it is accessed.

can handle multiple valid target addresses is given in Figure 5. The AB is for the SPARC platform. Obviously, the size of the AB increases with the number of target addresses to be checked.

```

! Begin Call Assertion
.Lfind_lease56:
    sethi %hi(.Lfind_lease56),%l7          (1)
    or %l7,%lo(.Lfind_lease56),%l7       (2)
! %l7 = Address(start assert) + main
    ld [%l7+52],%l7                       (3)
! %l7 = call dummy
    sethi 0xffff,%l6                      (4)
    or %l6,1023,%l6                      (5)
! %l6 = 0x3ffffff
    and %l7,%l6,%l7                      (6)
! %l7 = dummy = Xout (in words)
    sll %l7,2,%l7                        (7)
! %l7 = Xout (in bytes)
    sethi %hi(0x5e74),%l6                (8)
    or %l6,%lo(0x5e74),%l6              (9)
! %l6 = X1
    sub %l7,%l6,%l7                      (10)
! %l7 = Xout - X1
    cmp %g0,%l7                          (11)
    subx %g0,-1,%l7                      (12)
! %l7 = !(Xout - X1)
    sdiv %l6,%l7,%g0                     (13)
! will raise SIGFPE if mismatch
! End Call Assertion
.L_find_lease_R_1:
    call find_lease,0                     (14)

```

**Figure 5. Assertion Block for a call instruction.**

The subroutine being called is *find\_lease*. (1)-(3) load the CFI from memory, (4)-(7) extract the displacement in the CFI and place it in *Xout*, (8)-(9) load the valid target address in *X1*, the address being generated by a static analysis of assembly code, (10)-(12) form the expression  $!(Xout - X1)$ , (13) performs the division in order to raise DIVBYZERO signal in case of mismatch.

The ABs are inserted automatically into the disassembled application code by a 4-pass tool called *Insert\_Assert*. The tool is specific to the architecture of the machine for which the machine code is being generated. Currently, the tool has been developed for the SPARC architecture. The work done by the tool in its four phases is outlined below: (Let us say the baseline application code (without PECOS) is *dhcp\_gold.s* and the application code with the PECOS assertion blocks embedded is *dhcp.s*. We are using file names from the DHCP application, which served as the workload for our experiments.)

1. In phase 1, *dhcp\_gold.s* is taken as the input, and the uninstantiated call table and the uninstantiated label table are produced as output. The call table consists of entries with the following fields: the name of the subroutine, the number of callers of the subroutine, the addresses of the points from which the subroutine may return to the caller, the addresses of the points from where the subroutine may be called. The label table



consists of entries with the following fields: the label string (like, L123) and the address offset where the label appears. In phase 1, only the names of the subroutines and the number of callers of each one are filled in in the call table and the label strings are filled in in the label table.

2. Phase 2 takes *dhcp\_gold.s* and the uninstantiated call and label tables as input. It puts the ABs in the code. However, the ABs are as yet uninstantiated, i.e., the address offsets have not been filled in and there are placeholder labels instead. The intermediate file with the uninstantiated ABs is called *dhcp.s.out*, say.
3. Phase 3 takes *dhcp.s.out*, processes it to generate the address offsets for every instruction in the file, and uses this to instantiate the call and label tables. Instantiating the tables implies filling up the address fields in them. All addresses are offsets from the main function of the application and the absolute addresses are only generated at runtime. As a consequence, PECOS is able to handle code that is dynamically linked and loaded.
4. Phase 4 takes *dhcp.s.out* and the instantiated call and label tables and instantiates the ABs to generate the final augmented assembly file *dhcp.s*. This is then compiled to generate the final executable.

The steps to be followed in applying PECOS to an application are shown in Figure 6. Note that the application can be a black-box application, so long as it can be disassembled to generate the assembly code representation. The CFIs are augmented with the assertion blocks to protect against the control flow faults.

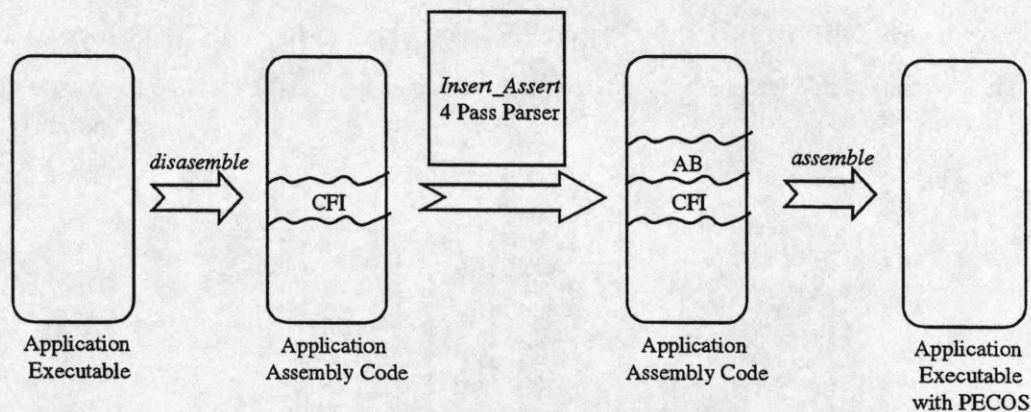


Figure 6. Steps to be followed for applying PECOS to an application

### 3.4 Assumptions

Applying PECOS to an application requires making the following assumptions:

1. The application can be disassembled to its assembly code representation. Our experiences with a range of applications (quicksort, DHCP) have shown that this was not a problem on the SPARC architecture. However, CISC architectures, such as x86, with data embedding in the instruction stream and no fixed boundaries between opcode and operand, may pose problems in the disassembly phase.
2. The valid target addresses can be discovered at compile time. This is obviously not true when functions are accessed through pointers or when the target of a jump is dependent on a variable's value. This points

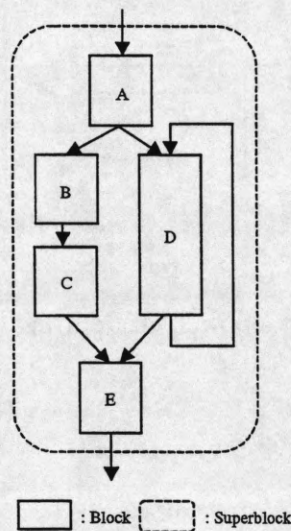
to a fundamental limitation of control flow signature techniques: they cannot detect if an incorrect but valid control path is taken (e.g., a taken path is taken instead of the not-taken path). For this, the control flow monitoring must be coupled with data monitoring.

3. The AB uses some local registers<sup>4</sup> for its computation. In SPARC, there are 8 local registers, and the AB uses 2 of these, assuming that the application is not using them. If the application is currently using any of these 2 local registers, then the AB will have to store them in memory and reload them at the end. However, for the call and return ABs, a transition is being made from one subroutine to another. Therefore, the local registers of the first subroutine may safely be overwritten by a PECOS instruction.

### 3.5 Optimizations

If an assertion block is inserted for every CFI, then the memory overhead of PECOS will be fairly high. The overhead obviously depends on the structure of the application. A highly data-intensive application will have larger block sizes than a control-intensive application, and therefore, lower memory overhead.

An optimization that can be applied to reduce the memory overhead is to cluster several blocks into a super-block and use an AB for each super-block. The super-block has the property that there is a single entry point to it and a single exit point from it, though there may be jumps in between. Figure 7 shows how four blocks are clustered to form one super-block for the purpose of embedding ABs of PECOS. The drawback of this optimization is a reduction in coverage because control flow within a super-block is no longer being monitored.



**Figure 7. Optimization: Combining multiple blocks into a superblock for inserting the Assertion Block**

---

<sup>4</sup> *Local register* means the register is meant for usage locally within the same subroutine.



## 4 System Description: Chameleon, NFTAPE and DHCP

The evaluation of PECOS is done using the Dynamic Host Configuration Protocol (DHCP) application as workload. The application is run both separately and as an application in the Chameleon environment, a Software Implemented Fault Tolerance (SIFT) middleware. Once PECOS flags an error, recovery needs to be initiated. Since Chameleon provides a low overhead recovery mechanism [WHI00], the application executions in Chameleon have the benefit of recovery. A software implemented fault injection tool called the Networked Fault Tolerance and Performance Evaluator (NFTAPE) is used for evaluating PECOS. Chameleon, DHCP and NFTAPE are described in the following sections.

### 4.1 Chameleon: SIFT Middleware

The Chameleon environment [KAL99] provides a means for constructing reliable distributed applications around ARMOR (Adaptive Reconfigurable Mobile Object for Reliability) processes executing on a network of heterogeneous, non-fault-tolerant nodes. ARMORs offers: (a) architecture, mechanisms, and an API to encapsulate a wide range of detection and recovery techniques that can be efficiently used by applications and (b) a management framework, which controls the use of these techniques for constructing highly configurable fault tolerance services.

ARMORs communicate through message passing and are built from replaceable components called *elements* and *compounds*. Elements constitute the most basic functional unit of the ARMOR and can be replaced during runtime, thus allowing the ARMOR process to adapt to changing application requirements or changing runtime environments. Elements are passive objects that are invoked by messages to perform specific operations (such as taking a checkpoint). Each incoming message spawns a new thread of execution within the ARMOR process. A compound is a collection of elements combined to perform some common task. A compound provides the message subscription and delivery function for the elements. An ARMOR is a compound that can exist independently and can migrate to remote nodes, be installed there, and provide fault tolerance services to an application.

ARMORs fall into three broad classes:

1. *Managers*: Manager ARMORs oversee other ARMORs. They are responsible for installing ARMORs, detecting and recovering from their failures, and initiating reconfiguration of subordinate ARMORs. The highest-ranking manager is the {it Fault Tolerance Manager} (FTM), which interfaces with the user to accept the reliability requirements, decides on a specific execution configuration (such as primary-backup execution) based on the requirements, and instantiates appropriate ARMORs to perform the execution.
2. *Daemons*: Daemon ARMORs are installed on every node participating in Chameleon. A daemons acts as the gateway of a node for all ARMOR communication and provide error detection for the locally installed ARMORs.

3. *Common ARMORs*: Common ARMORs implement specific techniques for providing application-required dependability. Examples of common ARMORs include the Execution ARMOR which is responsible for installing the application and overseeing it during execution.

An application can take advantage of Chameleon services (such as error detection and recovery) through the concept of an *Embedded ARMOR*. The core element-compound structure of the ARMOR is linked to the user application process. The application code is lightly instrumented with embedded ARMOR API to invoke services of the underlying element-compound structure. In this configuration, the embedded ARMOR process appears as a full-fledged ARMOR to the Chameleon environment and as a native application process to other processes outside Chameleon.

## 4.2 DHCP: Workload

For this study, we wished to use an application that is:

1. A real-world application so that the detection coverage of PECOS could be demonstrated for a workload that hadn't been handcrafted by the developers of the technique,
2. A distributed application for which the property of fail-silence is especially important,
3. Open source since modifications to the application source code are required to make it run within Chameleon, and
4. Widely used in real-world environments to provide some critical service.

The Dynamic Host Configuration Protocol (DHCP) application has each of the above properties. The application, written by the Internet Software Consortium [ISC00], was used as the workload for the current study. It is widely used in mobile and wireless environments for network management, and the DHCP server provides critical services like IP address allocation in such environments.

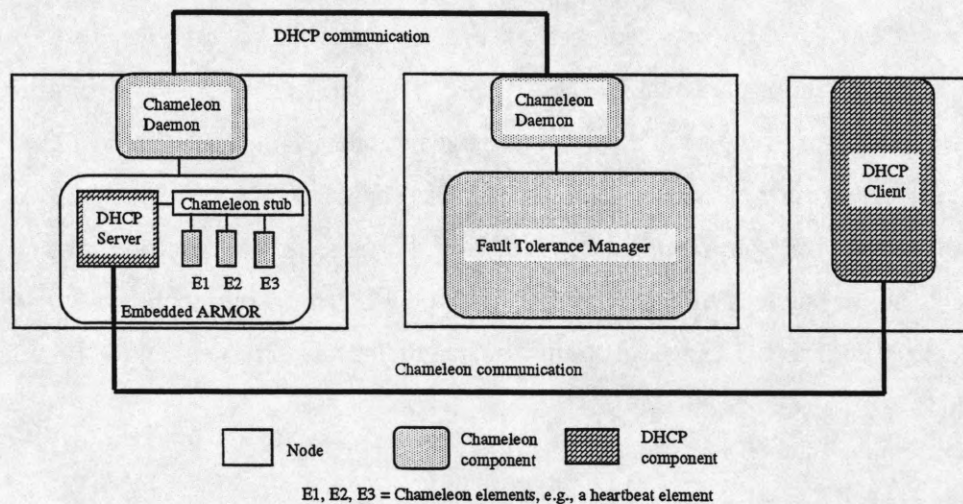
DHCP [RFC97] is an Internet Draft Standard Protocol for providing configuration information to hosts on an IP network. It is a client-server protocol used by the client to obtain information about a dynamic network. By dynamic we mean that either the configuration of the network is changing frequently or clients are joining and leaving the network frequently. The latter is a common case where there are mobile or wireless clients. The most important information needed by the client is an IP address that can be assigned to it and which it can use for further communication on the network. In addition, it can also use the protocol to discover network parameters such as the Maximum Transmission Unit (MTU) of the network. In networks using DHCP, the availability and correctness of the DHCP server are critical when hosts join the network. 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 may be allocated an incorrect or already used IP address.



DHCP supports dynamic allocation of IP addresses. In dynamic allocation, the server allocates an IP address to the client for a limited amount of time (termed as the *lease time*), after which the address can be reclaimed by the server. This is the mode in which the application is run for the experiments. The basic protocol exchanges that take place between the client and the server are described here. 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 additionally contains a tentative offer of an IP address and configuration parameters. The client collects the DHCPOFFER responses, chooses one server to interact with further, and broadcasts a DHCPREQUEST message with the identification of the chosen server. The chosen server on receiving the DHCPREQUEST message commits the binding of the IP address of the client to a leases database in stable storage and responds with a DHCPACK message containing the IP address and the configuration parameters for the requesting client. If the selected server is unable to satisfy the DHCPREQUEST message (e.g., the requested network address has been allocated), then the server responds with a DHCPNAK message.

Version 2 of the implementation of DHCP from the Internet Software Consortium [ISC00] is used for the experiments. The distribution includes the DHCP server, the DHCP client, and the DHCP relay agent (to pass messages between clients and servers not on the same physical subnet). For the experiments, the DHCP server is run in two ways: as an unmodified DHCP server augmented with PECOS instructions (which we call the standalone DHCP server) and as an embedded ARMOR, whereby it runs in the same process as an ARMOR. The server is instrumented with some Chameleon API calls through which it interacts with the ARMOR part of the process (e.g., to receive heartbeat query messages and respond to them). The configuration is shown in Figure 8.

The motivation behind running the DHCP server as an embedded ARMOR is to demonstrate how an off-the-shelf application can make use of Chameleon's management, detection, and recovery facilities. For the purpose of this study, however, only PECOS detection is turned on.



**Figure 8. DHCP Server running as an Embedded ARMOR in Chameleon.**

The Embedded ARMOR has two parts – the application part and the ARMOR part. The server interacts with the client through messages outside of Chameleon.

### 4.3 NFTAPE: Fault Injection Tool

NFTAPE [STO00] is a software-implemented fault injection tool that can be used in a networked environment for injecting a wide variety of faults. It provides an API for writing simple fault injectors, called light-weight fault injectors (LWFI). The fault injectors can be light-weight because most of the weighty tasks such as logging the results of the injection campaigns or setting triggers for the fault injection, are provided as separate, easily decomposable modules within NFTAPE. So the programmer can concentrate on developing the platform-specific LWFI and leverage off the infrastructure provided by NFTAPE for the platform-independent services.

For the experimental evaluation of PECOS, a control flow fault injector is developed as a LWFI in NFTAPE. The fault model used throughout the experiments is memory bit flip. The LWFI is developed for the Solaris platform. It uses the *proc* file system on Solaris to obtain access to the memory image of the running application. The LWFI is run as a parent process and the application is run as its child process so that the injector can corrupt any location in the text segment of the application. In each run, a single fault is injected. This is so that a direct correlation can be done between the injected fault and the observed manifestation of the fault. For injecting a fault, the instruction to be faulted is chosen at random, it is corrupted, and a breakpoint is set after the instruction. If the instruction is reached in the execution path of the application, then the breakpoint raises the trap. At this point, the instruction is reset to the correct value and the breakpoint removed. This ensures that even if the instruction is part of a loop, the faulty instruction will be executed only once.

## 5 Experiments and Results

### 5.1 Fault Injection Campaigns

To evaluate the coverage of PECOS, the application was run respectively without and with PECOS assertion blocks embedded. Two sets of fault injection campaigns were performed. In the first set of campaigns, bit flips were injected into the CFIs of the application. The CFIs in the application were identified, a random CFI was chosen, and a random bit in the particular CFI was flipped. This can be looked upon as directed fault injection to trigger control flow errors. In the second set of campaigns, random fault injections to the text segment of the process were done. An instruction, not necessarily a CFI, was chosen at random from the instruction stream and a random bit was selected for the injection. Depending on the target of the injection, and whether the vanilla application code or the augmented application code (with PECOS assertion blocks) was injected, the six campaigns presented in Table 3 can be defined. Campaigns 0-3 are directed control flow injections, 4-5 are random injections.



Campaign No.	Fault Injection Target
0	CFI, without PECOS
1	CFI & AB, with PECOS
2	CFI, with PECOS
3	AB, with PECOS
4	Anywhere in instruction stream, without PECOS
5	Anywhere in instruction stream, with PECOS

**Table 3. Different fault injection campaigns for evaluating PECOS.**  
(CFI = Control Flow Instruction, AB = Assertion Block)

For the DHCP application used, the breakdown of the different types of CFI and the number of assembly level instructions in the corresponding AB are shown in Table 4.

Type of CFI	% Occurrence	# instructions per AB
BRANCH	87.6%	13
CALL	3.8%	13
JUMP/RETURN <sup>5</sup>	8.6%	16 + ( <i>num_callers</i> )*4

**Table 4. Types of control flow instructions in DHCP and corresponding assertion block size**

In each campaign, 1000 runs were done. In each run, one fault of the corresponding type was injected and the application was allowed to run for 30 seconds, which gave the application time to go through approximately 5 rounds of the complete DHCP transitions. The fault was injected into the randomly chosen instruction before the DHCP server started execution. A consequence of this method of fault injection is that the latency of the fault cannot be measured meaningfully since the fault latency depends solely on when the faulty instruction is reached in the execution path.

For each of the campaigns, the results are categorized as follows.

1. *Fault Not Activated*: The faulty instruction was not executed because the execution path of the application did not touch that instruction. These cases were discarded from further analysis.
2. *Fault Activated But Not Manifested*: The faulty instruction was executed, but no error manifested, i.e., the client and the server went through the correct protocol exchanges.
3. *PECOS Detection*: The PECOS assertion block caught the fault.

---

<sup>5</sup> In SPARC, a return is implemented as a register indirect jump. *num\_callers* refers to the number of places from which the subroutine may be called, which gives the number of valid target addresses to assert for a return.

4. *System Detection*: The system detected the fault, raised an illegal signal, and caused the application process to crash.
5. *DHCP Server Program Check*: This category consists of semantic checks and reasonableness checks built into the DHCP server. For example, if the server detects that a client has duplicate leases, it flags an error and terminates.
6. *Hang*: The DHCP server hung.
7. *Fail-Silence Violation*: A process is said to be fail-silent if it either works correctly or stops functioning (i.e., becomes silent) if an internal failure occurs [BRA96]. A process failure that violates this specification is categorized as non-fail-silent behavior. This is the most insidious and damaging failure case. Here, the server does not crash, it is not detected by any of the techniques, and it sends out messages.

The DHCP server and client outputs were logged during each run. The log also contains the instruction being injected. After the set of runs completed, an offline analysis tool examined all the logs and categorized each run into one of the above seven categories. Some runs were discarded because the fault injector did not even start up the application (DHCP server). The number of runs where no fault was activated was noted, and then removed for computing the percentage of runs which fell in categories 2-7.

## 5.2 Evaluation of Standard DHCP Server

In this section, the results of the injection into the standard DHCP server, i.e., the server not running as an application within Chameleon, are discussed. Table 5 gives the results of the directed fault injection (campaigns 0-3), and Table 6 gives the results of the random fault injection (campaigns 4 and 5).

Consequence	Campaign 0: CFI without PECOS	Campaign 1: CFI + AB with PECOS	Campaign 2: CFI with PECOS	Campaign 3: AB with PECOS
Fault Activated But Not Manifested	36.6%	38.4%	14.6%	40.7%
PECOS Detection	—	46.9%	68.2%	51.9%
System Detection	53.9%	13.6%	10.0%	6.9%
DHCP Server Program Check	0.9%	0.0%	0.0%	0.0%
Hang	3.4%	0.6%	2.9%	0.5%
Fail-Silence Violation	5.2%	0.4%	4.2%	0.0%

**Table 5. Results of directed control flow fault injection into DHCP server.**

The injection target is given in the heading with the campaign number.

(CFI = Control Flow Instruction, AB = Assertion Block)



Consequence	Campaign 4: Everywhere without PECOS	Campaign 5: Everywhere with PECOS
Fault Activated But Not Manifested	32.6%	31.3%
PECOS Detection	—	45.0%
System Detection	41.7%	21.7%
DHCP Server Program Check	0.0%	0.0%
Hang	19.9%	1.7%
Fail-Silence Violation	5.9%	0.5%

**Table 6. Results of random fault injection into DHCP server.**

The results show that PECOS reduced the incidence of fail-silence violations from 5.2% in the baseline case (Table 5 Campaign 0) to 0.4% with the PECOS assertion blocks (Table 5 Campaign 1), a 92.3% reduction. However, when the CFIs were selectively injected, the reduction was only 19.2% (Table 5 Campaigns 0 & 2). For the directed control flow injection, PECOS brought down the incidence of process crash from 53.9% to 13.6% (Table 5 Campaigns 0 & 1), approximately a four-fold decrease. Even with random instruction stream injections, a two-fold decrease in process crash was observed (Table 6).

The injection into the CFI and AB showed that the number of cases of fault activated but not manifested (first category) remained statistically equal (Table 5 Campaigns 0 & 1). This indicates that the technique did not generate false alarms and flag faults that would never be manifested as errors in the baseline case. Previous studies have seldom made this distinction [KAN96,ALK99]. Directed injections to the assertion blocks shows no fail-silence violation. This is an important validation of the PECOS assertion blocks because it shows that they were not sources of additional vulnerabilities. The percentage of cases of the faulty instruction not being reached in the execution of the program was about 50% for the directed injections and between 30-45% for the random injections. This indicates that it would have been misleading if we took the results from all the runs without filtering out the cases in which the fault was not activated at all because the percentage of success cases would have dominated and the rest of the categories would have shown smaller variations between the baseline and signaturred case. An interesting sidelight is that PECOS caught the two cases of DHCP program check errors. For the baseline application, the DHCP application would detect an error through a semantic check and recover from it by some application-specific recovery routine. But, for the augmented PECOS case, the semantic checks were not triggered at all. Rather, the AB detected the problem. Since semantic-based checks are more difficult to write and require application knowledge, if PECOS can catch the errors that would otherwise be caught by such checks, then PECOS can replace them. (However, the sample set is too small to draw a definite conclusion.) Moreover, since the detection latency using data checks has been shown to be at least an order of magnitude higher than the

detection latency using control signatures [KAN96], PECOS detection is valuable because it reduces the detection latency.

### Causes of Fail-Silence Violations

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 are given below:

1. The server is not getting the applicable record for the client host and therefore it sends a DHCPNAK. A DHCPNAK is a negative acknowledgement sent by the server to the client in the second phase of the transactions when it cannot allocate an IP address to the client for any reason.
2. The server is getting 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 any further but continues to use the old lease and the server allows it to do so.

The fail-silence violations with the PECOS inserted occur for the following reasons:

1. Some CFIs cannot be asserted due to limitations of the current tool *Insert\_Assert*. For example, a subroutine call to a subroutine in another file cannot be asserted.
2. The target address because of the corruption is still valid, but incorrect, i.e., an incorrect but valid control path is taken.
3. The opcode of the CFI is affected. The AB currently asserts only on the target address, not on the opcode of the CFI.
4. The type of the instruction is changed, e.g., from a move to a branch. Since the AB is inserted based on a CFI as a trigger, this kind of error is not protected by an AB. However, in the current fault injection experiments, only single bit flips are injected. Hamming distances between control flow and non-control flow instructions is more than one. So this kind of error is impossible in this setting.

### Performance Measurements

It is important to evaluate the performance degradation caused by any software detection technique. The performance measurement for the DHCP application was done 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 given separately for the two main phases of the protocol (see Section 4.1 for DHCP details).

- Phase 1 consists of the exchange of DHCPDISCOVER and DHCPOFFER messages from the client to the server, and from the server to the client, respectively.



- Phase 2 consists of the exchange of DHCPREQUEST and DHCPACK/DHCPNAK messages, from the client to the server, and from the server to the client, respectively.

Phase 1		Phase 2	
Server overhead	Client overhead	Server overhead	Client overhead
15.72%	1.30%	0.95%	1.81%

**Table 7. Performance measurements for PECOS.**

DHCP Server is run with and without signature and the percentage overhead measured. Server overhead is the percentage overhead seen at the server, client overhead is that seen from the client side and includes the network overhead.

Phase 1 is the startup phase, which is executed by the client once when it joins the network. Phase 2 is done when the lease needs to be renewed. (It is actually done after half the time of the lease has expired.) So it can be argued that though the overhead on the server side is relatively high for phase 1, this can be tolerated as it is a startup and not a recurring cost. Also the overhead from the client side is about 1% for both the phases. The reason for the client side overhead being low in phase 1, in spite of a higher server side overhead, is that the network overhead is added in both the baseline and signed case. Therefore, in the percentage figure, the overhead becomes smaller.

### 5.3 Evaluation of DHCP as a Chameleon Application

This section presents results from injection into the DHCP server when it is run as an Embedded ARMOR within Chameleon. As mentioned before, running the DHCP server within Chameleon enables process recovery, which is provided by Chameleon [WHI00]. However, in this paper, we focus only on the evaluation of the control flow signature technique.

Consequence	Campaign 0: CFI without PECOS	Campaign 1: CFI + AB with PECOS	Campaign 2: CFI with PECOS	Campaign 3: AB with PECOS
Fault Activated But Not Manifested	45.5%	46.2%	18.6%	36.2%
PECOS Detection	—	31.6%	58.5%	44.7%
System Detection	46.8%	21.2%	19.1%	17.6%
DHCP Server Program Check	0.0%	0.0%	0.0%	0.0%
Hang	1.3%	0.9%	0.0%	0.0%
Fail-Silence Violation	6.5%	0.0%	3.8%	1.5%

**Table 8. Results of directed control flow fault injection into DHCP server running within Chameleon. The injection target is given in the heading with the campaign number.**

(CFI = Control Flow Instruction, AB = Assertion Block)

Table 8 gives the results of the directed fault injection (campaigns 0-3), and Table 9 gives the results of the random fault injection. The percentage figures are arrived at from 1000 runs in each campaign, after eliminating the cases in which the fault was not activated because the faulty instruction was not reached.

Consequence	Campaign 4: Everywhere without PECOS	Campaign 5: Everywhere with PECOS
Fault Activated But Not Manifested	47.3%	37.5%
PECOS Detection	—	32.4%
System Detection	50.0%	29.4%
DHCP Server Program Check	0.0%	0.0%
Hang	0.4%	0.0%
Fail-Silence Violation	2.3%	0.7%

**Table 9. Results of random fault injection into DHCP server running as Chameleon application.**

The directed injection shows a startling drop in the fail-silence violations. With directed injection only into the CFI, PECOS detects more than half the faults (Table 8 Campaign 2), while including the AB brings this down to less than one-third (Table 8 Campaign 1). Since the AB is longer than the CFI block, random injections would target the AB instructions more frequently. This result is expected because the technique is supposed to detect errors in the CFI, not in the block itself. However, an encouraging result is that injecting in a directed manner to AB instructions does not provoke too many fail-silence violations (1.5%, from Table 8 Campaign 3).

For the random injection anywhere in the text segment, PECOS is still able to reduce the fail-silence violations by about 70% (Table 9 Campaigns 4 & 5). About one-third of the faults are detected by PECOS (32.4%, from Table 9 Campaign 5). This indicates that for random injections to the text segment, a control flow error occurs at least one-third of the time. This is an important observation because it places in perspective the importance of control flow faults in the domain of a random memory bit flip fault model, at least for the class of applications represented by DHCP.

## 6 Conclusions and Future Work

The paper presents a control flow error detection technique called PECOS that is preemptive in nature and performs the checking between the golden and runtime control signatures before the execution of the control flow instruction. Due to the preemptive nature, the technique has no latency between the error and the detection. PECOS is shown to bring down the incidence of process crash and fail-silence violations for a representative client-server application, the DHCP application. Preliminary measurements of performance overhead done for the unoptimized case of the PECOS blocks embedded into the DHCP server shows that performance degradation



varies between 1% to 15% depending on the phase of the protocol and whether the client or the server side is monitored.

A detailed fault injection study has been performed to analyze the vanilla DHCP Server's failure characteristics (i.e., what types of failures occur – crash, hang, fail-silence violations, etc.). Then, the injection experiments are performed to evaluate the improvement in dependability (improvement in the fail-silence coverage in particular) due to PECOS. Importantly, fault injection campaigns are also performed to evaluate the robustness of the checking code that is inserted in the PECOS assertion blocks. Random injections to the instruction stream of the application shows that control flow errors constitute a significant fraction of errors in a control-centric application like DHCP.

PECOS has high memory overhead in its current unoptimized implementation. Current work is exploring means of reducing the memory overhead. PECOS is currently unable to handle control flow instructions that cause a branch to an instruction in a function in another file. The call instruction frequently causes this kind of control change. Hence, such control constructs also need to be handled. Finally, PECOS needs to be evaluated for a variety of workloads – data centric, distributed applications with more coarse-grained communication, or with shared state.

An important avenue for future work is to bring the concepts of data signature and control signature together. A control flow based approach alone will not be able to detect errors that cause the application to take incorrect but valid control paths, which is observed to be the main cause of fail-silence violations. For this, a hybrid control and data signature will have to be used. This can be explored in the context of Chameleon where the idea of data signature has been introduced for ARMORs.

## 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.
- [BRA96] F.V. Brasileiro, P.D. Ezhilchevan, S.K. Shrivastava, N.A. Speirs, S. Tao, "Implementing Fail-Silent Nodes for Distributed Systems," *IEEE Transactions on Computers*, Vol. 45, No. 11, pp. 1226-1238, November 1996.
- [ISC00] Internet Software Consortium (ISC) Dynamic Host Configuration Protocol (DHCP), URL: <http://www.isc.org/products/DHCP>.
- [KAL99] Z. Kalbarczyk, R.K. Iyer, S. Bagchi, K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 10, No. 6, pp. 560-579, June 1999.
- [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.

- [MAD91] H. Madeira, J.G. Silva, "On-Line Signature Learning and Checking," Proc. 2<sup>nd</sup> IFIP Working Conf, on Dependable Computing for Critical Applications (DCCA-2), pp. 170-177, Feb. 1991.
- [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.
- [MIR92] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, "Two Software Techniques for On-Line Error Detection," Proc. 22<sup>nd</sup> 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. 5<sup>th</sup> IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5), pp. 113-124, 1995.
- [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. 22<sup>nd</sup> 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. 16<sup>th</sup> 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.
- [SHE83] J.P. Shen, M.A. Schuette, "On-Line Monitoring Using Signed Instruction Streams," Proc. 13<sup>th</sup> International Test Conference, pp. 275-282, Oct. 1983.
- [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.
- [WHI00] K. Whisnant, Z. Kalbarczyk, R.K. Iyer, "Micro-checkpointing: A Checkpointing for Multithreaded Applications," To Appear at the 6<sup>th</sup> IEEE On-Line Testing Workshop, Mallorca, Spain, July 2000.
- [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.