Submission for IEEE Trans. *on Computers – regular paper*

Design and Evaluation of Preemptive Control Signature (PECOS) Checking

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
Fax: (217) 244 5686; Tel: (217) 244 7110
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. The PECOS target error model is any corruption that causes the application to take an incorrect control flow path. This includes corrupted control flow instructions as well as any other corruption that subsequently affects the control flow. The proposed technique is shown to handle both static and dynamic control flow constructs. 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 the impact of failures in the address and data lines between a processor and memory. The effectiveness of PECOS is illustrated on a real application: the Dynamic Host Configuration Protocol (DHCP) server. It is shown that PECOS detects more than 87% of control flow errors, reducing the incidence of fail-silence violations from 3.6% to 0.1% and of process crashes from 54.6% to 7.1%. Performance studies show a degradation of 15-29% with instrumentation of the entire DHCP server, and a degradation of 5-13% 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.

* Contact author.

Design and Evaluation of Preemptive Control Signature (PECOS) Checking

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. The PECOS target error model is any corruption that causes the application to take an incorrect control flow path. This includes corrupted control flow instructions as well as any other corruption that subsequently affects the control flow. The proposed technique is shown to handle both static and dynamic control flow constructs. 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 the impact of failures in the address and data lines between a processor and memory. The effectiveness of PECOS is illustrated on a real application: the Dynamic Host Configuration Protocol (DHCP) server. It is shown that PECOS detects more than 87% of control flow errors, reducing the incidence of fail-silence violations from 3.6% to 0.1% and of process crashes from 54.6% to 7.1%. Performance studies show a degradation of 15-29% with instrumentation of the entire DHCP server, and a degradation of 5-13% 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*, which are 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 violations A fail-silent application process either works correctly, or it 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. 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 (PreEmptive COntrol Signatures). In this context, *preemptive checking* means that the detection technique is activated before the error causes an incorrect control flow path to be taken. The PECOS target error model is any corruption that causes the application to take an incorrect control flow path. This includes corrupted control flow instructions as well as 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. PECOS was first presented in [BAG01], where it was applied to protect call-processing clients in a wireless telephone network controller.

Broadly speaking, state-of-the-art techniques detect 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 has been that system detection (i.e., process crash) is an acceptable way of detecting an error—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 the crash), 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 three distinct advantages:

1. PECOS 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. PECOS reduces fail-silence violations for the target application as well.

3. Unlike other techniques, PECOS can also protect control flow instructions whose destinations are determined at run-time, such as a jump or a branch based on a register value determined at runtime. Modern applications increasingly contain such dynamic control flow characteristics.

The applicability of PECOS also makes it notable in the class of control flow error detection techniques. Here 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 the 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). While PECOS is clearly usable if the target source code is available, it 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.

PECOS is evaluated in this paper using the 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.

The remainder of the paper is organized as follows. Section 2 presents related work in the field. Section 3 introduces the design principle of PECOS, the implementation of the tool for instrumenting the application and the error types that PECOS protects against. Section 4 and Section 5 describe, respectively, the experimental evaluation of PECOS applied to the Dynamic Host Configuration Protocol (DHCP) and the results of that evaluation. Section 6 concludes the paper.

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]. The basic scheme is to divide the application program into blocks, each block having 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 at runtime by a watchdog processor. 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; each accesses the precomputed signature in a different way. Embedded Signature Monitoring (ESM) embeds the signature in the application program itself [WIL90, SCH86]. 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 are stored and the runtime signature is any *m*-out-of-*n* code word.

Applying hardware schemes to distributed environments suffers from limitations:

- Hardware schemes are quite suitable for small, embedded 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 of the multiple processes being executed.
- The underlying assumption of hardware schemes 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.
- 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 signature-based techniques.

2.2 Software Schemes

Software techniques also 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].

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

1. 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 instruction.[1] If there is a control flow error, executing instructions from an invalid target location is likely to trigger system detection before the specific control signature checking technique, and this results in process crash.
2. None of the techniques we are aware of can handle situations in which the control flow behavior is determined by runtime parameters, such as register-indirect jumps and calls to dynamic libraries.
3. Evaluations of existing signature techniques often do not bring out how sensitive the system is to errors in the checking code itself.
4. There is a question as to how generally applicable software approaches are 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, the existing software control flow techniques have been evaluated on fairly simple applications; for example, BSSC uses programs containing linked list operations, quicksort, and matrix transpose.

Table 1 presents a summary of representative hardware and software techniques for control flow error detection. The first five techniques presented require additional hardware, while the last two are software-based approaches. The table shows the results of evaluation of the available techniques. For the results, an attempt has been made to remove the cases in which the detection is done by a technique other than the one under consideration (e.g., system detection of application process crash is removed from the coverage number).

Finally, detection of control flow errors was one of the primary objectives in designing the so-called *coded microprocessor,* an idea introduced in the late sixties to detect errors in arithmetic units of processors [RAO74]. In recent years, in the area of safety-critical applications/systems (e.g., railway control systems), there was an effort to develop coded microprocessors based on information redundancy to handle a variety of data processing errors including control flow errors. The information is encoded employing: (1) an arithmetic code to detect operation errors (i.e., the incorrect execution of an instruction) and data store and transfer errors, and (2) a signature technique to detect operator errors (e.g., an addition operator replaced by a multiplication operator), operands errors, control flow errors, and errors in information refreshing (e.g., using incorrectly updated variable during successive iterations of a loop). The coded processor operates on encoded data employing

---

[1] A possible exception is the technique called *concurrent process monitoring with no reference signatures* [UPA94], which incorporates aspects of preemptive checking.

specifically designed coded operators. Dedicated software encodes the data, calculates the signatures, and validates the signature's correctness before the program can execute.

Observe that: (1) the coded processor performs error checks after an operation/instruction is executed, i.e., not preemptively, and (2) the approach is suitable for embedded processors executing simple (single-process/thread) applications with strictly deterministic behavior. The Vital Coded Processor was originally designed to protect trains in the Paris SACEM System [FOR89], [HEN93].

Technique	Outline of Scheme	Workload, Coverage, Resources	Comments
Signatured Instruction Stream (SIS) [Sch86]	The signature is embedded in the application instruction stream at the assembly code level. An optimization called *branch address hashing* reduces memory overhead by eliminating reference signatures at branch points.	Quicksort, string search, matrix transpose on MC-68000 *Coverage:* 36% *Hardware:* Watchdog processor with cyclic code signature generator *Software:* Modified assembler and loader	Needs runtime support to rehash branch addresses before target address calculation.
Path Signature Analysis (PSA) [Nam82]	Signatures are computed for paths, i.e., sequences of branch-free intervals (or nodes). 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.	Fifty unspecified programs from data manipulation to I/O control routines on MC-68000 *Coverage:* Not Available *Hardware:* Watchdog processor *Software:* Modified assembler	Requires complicated parser for finding path sets. Error detection latency can be significant, since detection is done at the end of the path.
On-line Signature Learning and Checking (OSLC) [Mad91]	Application is decomposed into sections, each having a number of branch-free intervals. 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.	Pseudo-random number generator, string search, bit manipulation, quick sort, prime number generator on Z-80 *Coverage:* 86.3% *Hardware:* Watchdog processor *Software:* Exhaustive tester	Requires exhaustive path activation to generate golden signatures. Synchronization required between the application processor, signature generator, and the checker.
Time-Time-Address Signature Checking [Mir95]	Program is decomposed into branch-free blocks (BFB). The watchdog starts a timer on getting notification of beginning of BFB. If notification of end of BFB is not received before timeout expires, an error is detected. At block exit, watchdog checks that exit is made at (start address + size).	Linked list manipulation, matrix manipulation, quicksort on MC6809E (8-bit CPU) *Coverage:* 23.9% (heavy ion radiation), 48.6% (power system disturbance) *Hardware:* Timer, watchdog processor *Software:* Software to decompose application into BFB and embed signature instructions	Difficult to determine time bounds for the watchdog. Sending frequent signals to the external watchdog may result in performance degradation.
Concurrent Process Monitoring with No Reference Signatures [Upa94]	A known signature function 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. At runtime, a watchdog verifies that, at a tagged instruction, an *m*-out-of-*n* code has been accumulated. No reference signatures are required.	Possibly no implementation exists *Coverage:* Not Available *Hardware:* Watchdog processor *Software:* Software to indicate checkpoints where code word needs to be checked and tag memory	One extra word per branch is required to force the accumulated signature to become an *m*-out-of-*n* code. Better at detecting control bit errors, than control flow errors.
Block Signature Self Checking (BSSC) [Mir92]	The program is divided into basic blocks, and each block is assigned a signature (the address of the first instruction in the basic block). 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.	Linked list manipulation, quicksort, matrix manipulation on MC6809 *Coverage:* 15-22% *Hardware:* None *Software:* Software to identify blocks, assign calls and signatures	The assertions introduce control flow instructions of their own. The technique assumes absolute addresses for the start of the basic block. This will preclude using it for relocatable code.
Enhanced Control-flow Checking with Assertions (ECCA) [Alk99]	The entry and the exit points of the branch-free intervals in a high-level language are fortified through assertions inserted in the instruction stream. In case of a control flow error, the BID (Branch Free Interval Identifier) computation will cause a divide-by-zero error.	Two SPEC Int92 benchmarks *Coverage:* 18.4-37.5% (022.li), 27.8-73.0% (*espresso*) *Hardware:* None *Software:* C-language lexer, filter and parser, signature generator	A parser for high-level code can be complex because of larger variations in code structure in the high-level language.

Table 1: Survey of Representative Control Flow Error Detection Techniques

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. 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. A PECOS assertion block is inserted at each trigger point. The assertion block contains: (1) the set of valid target addresses (i.e., reference signatures or golden values) 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 error,[2] and if it was, takes a recovery action, e.g., terminates the malfunctioning thread of execution.

At this point, it is important to clarify the distinction between *incorrect* and *illegal* control flow instructions. Suppose that a conditional branch can take one of the two possible execution paths, T1 or T2, and that correct runtime execution dictates that path T1 be taken. If the execution takes execution path T2, we say that an *incorrect but legal* branch has been taken. If the execution is transferred to a random location, T3, (i.e., the program does not follow any of the two legal execution paths, T1 or T2) we say that an *illegal (and clearly incorrect)* branch has been taken. PECOS can detect all illegal control flow transfers and a subset of incorrect but legal ones.

3.2 Why Preemptive Detection Is Important

Figure 1 summarizes the problem with non-preemptive schemes and discusses the solution proposed by PECOS approach. Figure 1(a) shows a non-preemptive scheme, in which the validity of the control path is checked at the time of arrival at the next execution block. All existing control flow error detection schemes that we are aware of are non-preemptive in nature and cannot easily be made preemptive, as discussed later in this section. The two fundamental reasons why a preemptive approach is preferable are examined next. Both reasons are related to the ease of recovery following detection.

---

[2] 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.

1. Process crashes are prevented. 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 a process crash before detection by the technique (Figure 1(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 a process crash (i.e., system detection). 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 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 1(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 multithreaded 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 can also be prevented via an error handler, which intercepts system-raised signals and gracefully terminates 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 my have occurred. 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)].

2. Error propagation is prevented. A second problem with non-preemptive schemes is the possibility of error propagation. Data from [KAN96] shows that the latency of detection is several hundreds of instruction cycles for software-based schemes (approximately 500 instruction cycles for the **C**ontrol **C**hecking with **A**ssertions (CCA) technique). Data on error latency [CHI89, YOU91, YOU92] has shown that, while the great majority of the errors are detected with a very small latency, errors that remain undetected for a long period have the potential to cause severe problems. These problems range from checkpoint corruption that invalidates retries to the sending of incorrect messages to peer processes in a distributed application. Thus, error propagation complicates recovery, which in turn critically affects overall system availability.

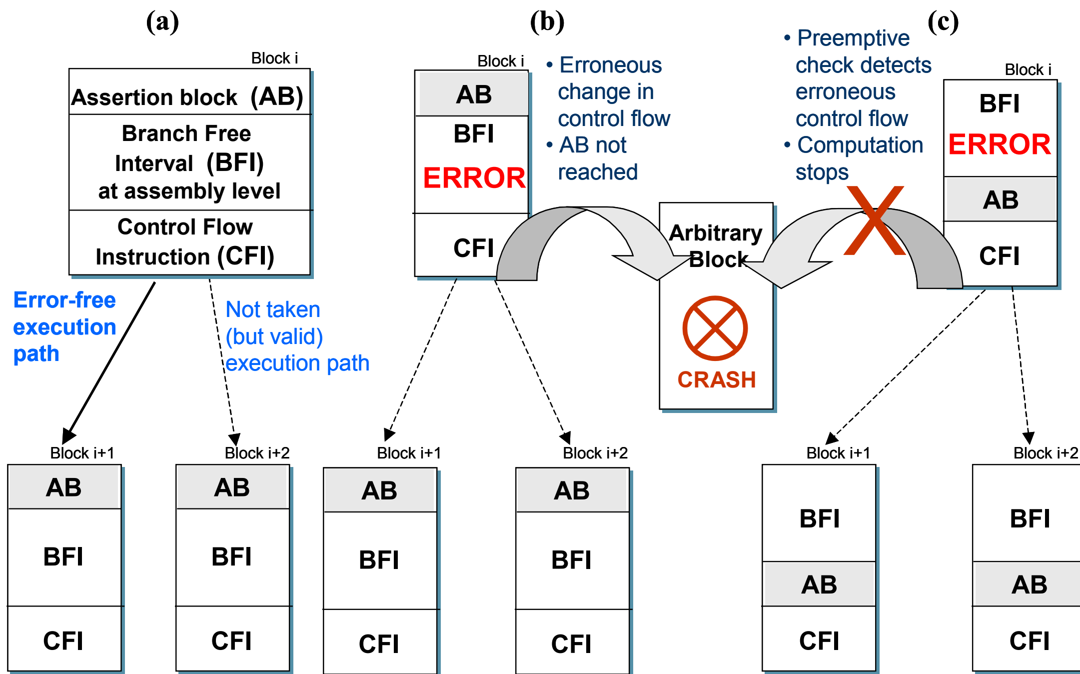

Figure 1: Reason for Preemptive Control Flow Checking: (a) Correct Execution, (b) Incorrect Execution (*without preemptive checking*), (c) Incorrect Execution (*with preemptive checking*)

Can existing techniques be made preemptive without substantial effort?

Our investigation suggests that while, in principle, existing techniques could perhaps all be made preemptive, several theoretical and practical hurdles involved 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; 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 do the following:

- Determine the target address of the control flow instruction.
- Jump to the target address and read off the block id of that address.
- 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. Finally, any claim that a technique can be made preemptive needs to be substantiated by applying the technique to a sizeable workload; clearly this has not been done.

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 until 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 to achieve this.

3.3 PECOS Instrumentation

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. The PECOS parser analyzes the control flow graph of the application and generates an assembly file with inserted assertion blocks. The assembly code is compiled and linked to create the final executable. The overhead in terms of person-hours for instrumenting applications with PECOS is minimal, as it involves only running the tool with the application code as input. It is to be noted that once the tool is available for a particular platform, there is very little manual intervention required to add PECOS, and therefore, the benefits of the technique can be made available to a large class of applications.

The tool was originally developed for the Sparc architecture and is structurally divided into an architecture-specific and an architecture-neutral component. Porting the architecture-specific component involves rewriting the assertion blocks in the assembly language of the target architecture and understanding the memory layout of the new platform. Since the tool is based on the widely used ELF executable format, supporting executables on a different platform is a non-issue. The tool has subsequently been ported to the PowerPC platform [YAN02].

As Figure 2 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 2 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 code.[3]

---

[3] Due to some nuances of the disassembler available on SPARC, some post-processing is required 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.

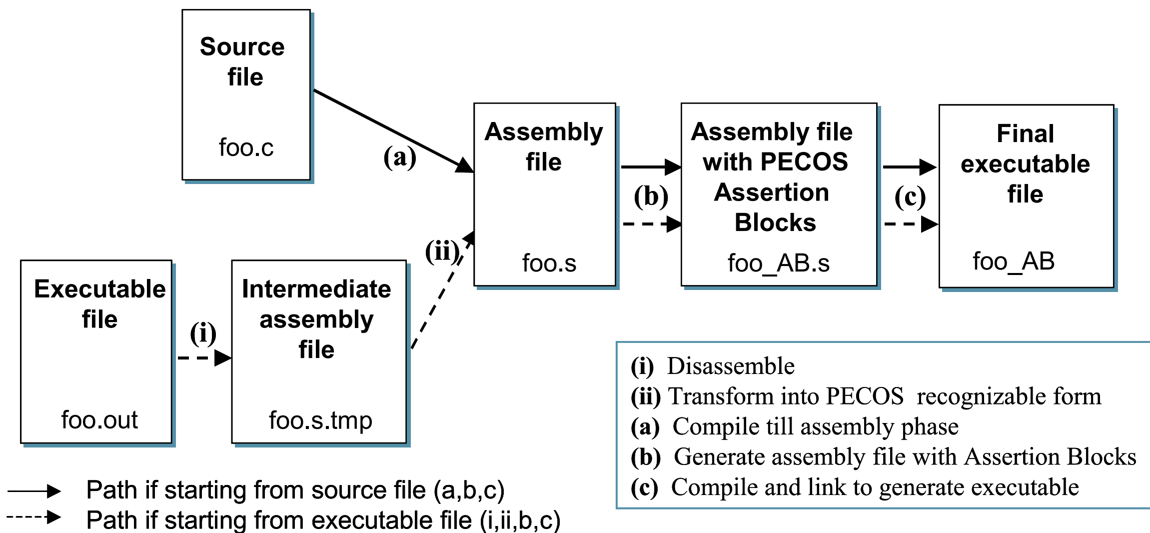

Figure 2: Process of Instrumenting an Application with PECOS

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. Moreover, 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 a high complexity tool comparable to a compiler with runtime capabilities.

3.4 Construction of PECOS Assertion Blocks

As shown in Figure 2, 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., it 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 these 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 3. 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 3: High-level Control Decision in the Assertion Block

Example assertion block for CFIs with constant operand. Figure 4 shows an assertion block for protecting a control flow instruction with a constant operand, in this case, a branch instruction. The assembly code is divided into three sections. The first and the third sections are taken from a vanilla application: Section I depict a sample *basic block* with no control flow instructions, and Section III depicts the *compare* and *branch* instructions. These two sections are augmented with Section II, the PECOS assertion block, which is placed between them. Instructions (1) and (2) load the runtime control flow instruction into the register *r7*[4]. Instruction (3) loads the valid target address offset into register *r6*. The PECOS parser discovers the valid address by a static analysis of the control flow graph of the application.[5] Instructions (4) and (5) add the opcode for the branch instruction to the address and form the valid instruction word in register r6. Instructions (6) through (8) check whether the valid instruction word and the runtime instruction word match and raise a divide-by-zero signal in case of a mismatch.

Now consider an error case. An error in the memory word storing the conditional branch instruction causes a corruption from the correct memory value of 0x02800017 to 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 (2) (see Figure 4) the incorrect memory word 0xffffffff is loaded into register *r7*. The correct memory word is loaded into register *r6*, and a subsequent comparison between the two register values causes a floating-point exception signal to be raised at instruction (8). By examining the program counter, the PECOS signal handler will find that the exception was raised by the assertion block and thus detect 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 application case (discussed later), this means terminating the offending thread. Similarly, an error that hits the PECOS assertion block, e.g.,

---

[4] The assertion uses local registers *r5*, *r6*, and *r7*, which were not used by the application in our study. If these registers are being used, then either some special-purpose unused registers can be used or the memory can be used as a scratch pad.

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

changing the valid address offset in instruction (3), will be detected by causing a floating-point exception. Further discussion of impact of errors in the assertion code is presented in Section 5.5.

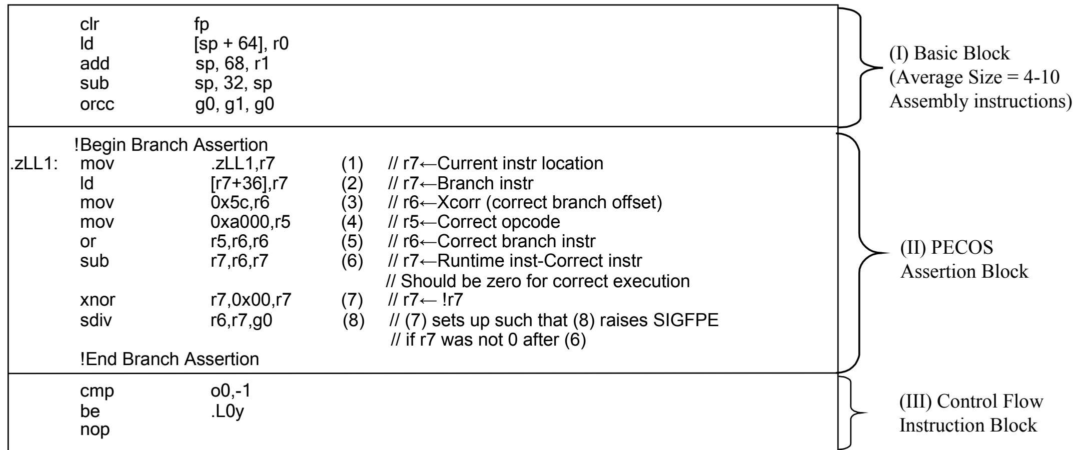

Figure 4: PECOS Assertion Block for Protecting Control Flow Instruction with a Constant Operand

3.5 PECOS Extension

The previous section presents an example of a PECOS assertion block for protecting control flow instructions that transfer control to an address that is a constant offset away. The extension presented in this section deals with control flow instructions that transfer control to an address determined by a runtime calculation. The discussion focuses on two cases: (1) indirect addressing modes and (2) dynamic library calls.

Construction of assertion blocks for indirect addressing mode. In this case the target address of the control flow instruction is determined by a runtime calculation, e.g., the register-indirect jump. To handle such cases, the PECOS parser can leverage compiler techniques used for data-flow analysis. An optimizing compiler needs information on variable *definition* and *use* between basic blocks for improving register usage, eliminating redundant computation, or eliminating dead-code. Usually, this information is stored as *use-definition chains (UD-chains)* and *definition-use chains (DU-chains),* which are created during the program compilation phase and represented as a list of program statement addresses (pointers to statements) [ASU96]. The PECOS parser can use this information to conduct a data discovery pass for extracting the addresses/locations of program statements that define the contents of registers supplying a runtime operand for control flow instructions, e.g., a destination address for an indirect branch. The program can be instrumented adding (after each statement/instruction identified in the data discovery phase) an extra code to preserve the register contents in a separate storage, which can be a register not used by the program, or a memory location if no such register is available). The PECOS assertion block inserted before the control flow instruction reads the value of the register at runtime prior to the execution of the instruction and compares it to the valid value stored at the time of the last definition of the register content. Figure 5 and Figure 6 depict, respectively, example scenarios of DU-chains for

a register indirect jump instruction[6] and an example of an assertion block for protecting a control flow instruction with indirect addressing.

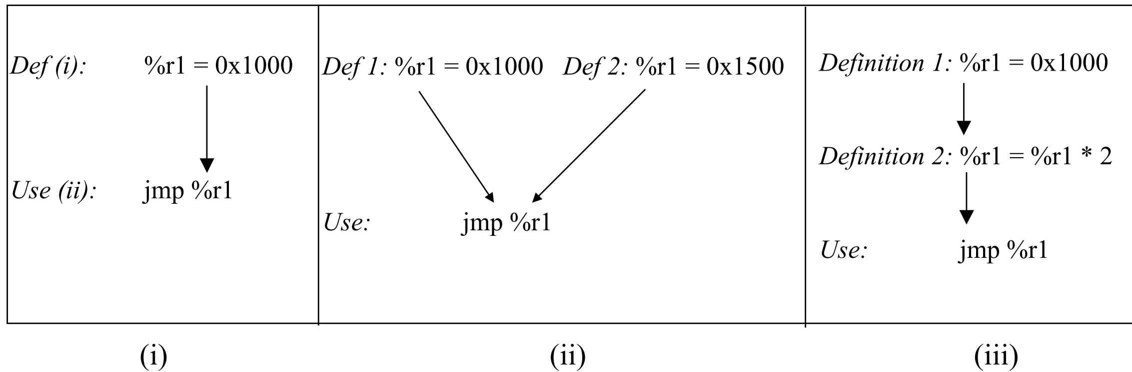

Figure 5: DU-Pairs For Control Flow Instructions with Register Indirect Addressing

Example assertion block for CFI with register operands. The assembly code in Figure 6 is divided into five sections. All but the fourth section are taken from a vanilla application. Section I depicts a sample basic block with no control flow instructions, which defines the target of the jump to follow later. Section II depicts the intervening basic blocks between the definition and the use of the register. Section III depicts the basic block in the use block, and Section V the *compare* and the *jump* instructions. These sections are augmented with Section IV, the PECOS assertion block placed just before the use of the register that holds the target address of the control flow instruction. Figure 6 depicts the general case in which the definition and the use of the register are not in the same basic block. Our initial study with SPEC2000 integer benchmarks running on an emulation engine for Alpha architecture indicated that, for several applications from the benchmark suite, a significant fraction of the indirect jumps (e.g., 40% and 30% for *gzip* and *twolf*, respectively) the definition and the use of the register are separated by 100 or more instructions.[7]

Instruction (1) loads the target of the jump instruction into a register. This represents the definition instruction of the D-U chain. Instruction (2) (embedded by the PECOS parser) creates a copy of the target address. Instructions (3) and (4) load the runtime control flow instruction into register *r5*. The code corresponding to the valid instruction (*jmp r7*) is 0x207170. Instructions (5) and (6) compare the instruction opcode to check that it is the valid instruction. Instructions (7) and (8) compare the runtime value of the register with the golden/stored

---

[6] Note that to enable protecting control flow instructions whose target addresses are determined at runtime may require analysis of the execution path that has been followed by the application prior to reaching the definition of a target address for a given control flow instruction. In the software testing community, there is a significant volume of work on branch testing, which requires: (1) generating a set of paths that will cover every arc in the program flow graph (called a path cover), and (2) finding a set of program inputs that will execute every path in the path cover, e.g., [BER94]. Program path analysis was also used to guide fault-injection-based (path-based injection) testing of applications e.g., [TSAI99], and to support fault injection for formal testing of fault tolerance algorithms and mechanisms e.g., [AVR92]. All these approaches are based on pre-analysis of the control and data flow of the program and, as such, are not applicable for discovering runtime-determined target addresses for control flow instructions, as required by PECOS.

[7] This study was conducted by J. Strutz, a student of the ECC442 class in the Fall 2001 at the University of Illinois at Urbana-Champaign taught by Dr. Z. Kalbarczyk.

value, which is available from *r4* stored in instruction (2). The assumption is that if the instruction is correct (i.e., it is indeed a jump to the location contained in register *r7*) and the value of the register *r7* has not been corrupted since its last definition, the target of the control flow instruction is valid. In the error free case, register *r5* should be zero (correct opcode) and register *r6* should be zero (correct jump target). Instructions (7) through (10) check that this holds and raise a divide-by-zero exception if it does not. Note that the larger the separation between the last definition and the use, the greater is the window of vulnerability that the PECOS assertion block protects against. Again, the study of the D-U instruction separations leads us to believe that PECOS is valuable in protecting corruptions to control flow instructions with the indirect addressing mode.

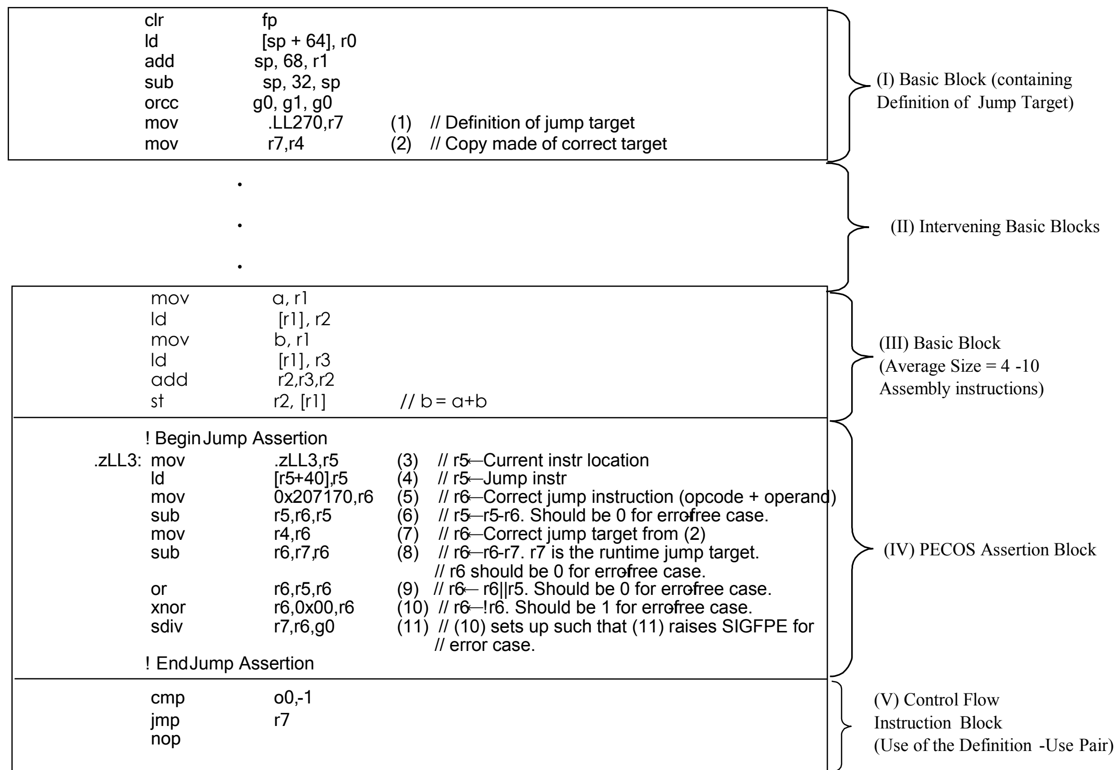

Figure 6: PECOS Assertion Block for Protecting CFI with Register Indirect Addressing

Construction of assertion blocks for dynamic library calls. For dynamic library calls, the targets of the call instructions are not resolved at compile time. Instead, the targets point to appropriate offsets in the Program Linkage Table (PLT). At the time the program is loaded into memory, the positions of the dynamic libraries are instantiated in the PLT. The assertion block that is used to protect the call instruction to the dynamic library reads the position of the library from the PLT, uses that as the golden value, and compares it against the target address of the call instruction. In effect, it makes a redundant copy of the address of the dynamic library at the time of program loading and flags an error if the two copies differ at the time of making the call.

Handling exotic addressing modes. In Sparc, all the addressing modes calculate addresses through three basic mechanisms: (1) register + register, (2) register + immediate, or (3) immediate. In the first two cases, the

mechanism shown in Figure 6 for register indirect jumps will hold. The calculation of the correct target address will have to include the arithmetic operation in the addressing mode. The last case is described in Section 3.4 in the example of the branch instruction.

3.6 Error Types That PECOS Protects Against

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

1. Single or multiple bit flips in memory, including main memory or cache (on-chip or off-chip), e.g., a burst error that corrupts a call instruction in L1 cache.[8]
2. Transmission errors 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.
3. Register errors, e.g., corruption of a register value that determines the destination address of an unconditional jump instruction.

PECOS cannot capture control flow errors that result from the corruption of a non-control-flow instruction in 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 distance of opcodes belonging to the non-control-flow category and to the control flow category is 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.

Software control flow detection tools cannot in general detect errors in the processor's internal structures. For this, support from in-processor hardware modules is needed. A current extension to incorporate the PECOS technique into a reliability engine that sits inside the processor and monitors instructions after the fetching has been studied in [PAT01]. With this extension, errors in internal processor structures such as program counters can be detected. With the software-implemented technique as described in the current paper, a small subset of program counter errors can be detected, namely, those that cause some key instruction within the assertion block not to be executed. This causes the register not to be set, and then, when the division instruction is executed, the signal is raised indicating an error.

[8] 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.

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). This application was chosen for three reasons:

1. DHCP is widely used in networked wireless/wireline systems to provide a critical service.
2. As a "real-world" application, DHCP cannot be simply handcrafted by the developers of the CFI technique and can potentially serve as a benchmark on which future techniques are evaluated.
3. The property of fail-silence is especially important in DHCP; sending out an incorrect value can have significant negative consequences.

Error injection campaigns were conducted into the baseline (uninstrumented) DHCP first and then into DHCP instrumented with PECOS. For the dependability analysis, the entire DHCP application was instrumented. Performance results are shown for exhaustive instrumentation and selective instrumentation of the core protocol engine. 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-injection[9] tool, was used for conducting the error injection campaigns [STO00] involving a wide range of error models.

4.1 The Target Application: DHCP

The Dynamic Host Configuration Protocol (DHCP) application is widely used in mobile and wireless environments for network management. DHCP is a client-server protocol used by a client to obtain configuration information, most importantly a client IP address, in a dynamic network, i.e., a network in which the configuration of the network changes frequently or clients join and leave the network frequently. 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 current study used the DHCP Version 2 implementation from the Internet Software Consortium [ISC00].

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 and chooses one server to interact with 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

---

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

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], adding random memory errors. These models represent a wide range of transient hardware errors and some software bugs (e.g., 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). 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 [SCH87, MAD94]. Therefore, transient hardware errors/failures were adopted as the error model for the current experiments.[10]

The errors chosen are depicted in Table 2. For example, the DATAOF models an error during transmission over a 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 occur, 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
ADDOF	Address line error when an operand is fetched
ADDOS	Address line error when an operand is stored
DATAOS	Data line error when an operand is stored

Table 2: Transient Error Models for PECOS Evaluation

Though the errors were injected using memory bit flips, they model a larger class of errors, including:

- Transmission errors on the bus between the memory and the process, or between the on-chip cache and the processor execution core.
- Errors in disk, main memory, or on-chip cache.

[10] Permanent failures are usually covered by dedicated hardware detection mechanisms, including periodic sampling to check the status of hardware components.

- Software errors. Work by Madeira [MAD00] has analyzed what class of software faults can be mimicked by memory bit flips. The study showed that, of the six classes of software faults identified by Chillarege *et al.* in their Orthogonal Defect Classification work [CHI95], assignment errors (values assigned incorrectly or not assigned at all) and checking errors (missing or incorrect validation of data or incorrect loop or conditional statements) can be modeled by memory bit flips.

The experiments were conducted on the baseline target (without PECOS instrumentation) and on the PECOS-instrumented target. In the PECOS-instrumented case, error injections were divided into two categories: (1) injection of errors randomly in the instruction stream of the application and (2) 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. The following method (provided by the debugger-based fault injector in NFTAPE) was used to inject the errors:

1. Attach the fault injector to the target process and set a breakpoint at the target instruction.
2. Corrupt the target instruction when the breakpoint is reached. (If the target instruction is not reached, the fault is not activated.)
3. Execute the corrupted instruction.
4. Replace the corrupted instruction with the original instruction.
5. Continue executing the target process.

It should be noted that PECOS is capable of detecting multiple bit errors as well. Any error that causes a change of control flow to an illegal control flow can be detected. A subset of errors that cause a change to an incorrect but still legal control flow can be caught by the technique (see Section 3.5).

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 4.1.[11] 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 3.

[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 highly likely that it will not be activated at all.

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 3: Categorization of Results from Error Injection

The fail-silence violation category (FV) is considered to have the most debilitating effect on system availability. For our DHCP study, we define a fail-silence violation as any of the following three cases:

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 immediately next IP address available in the address pool to the client.

5 Results

This section discusses the results from the error injection campaigns. Table 4 and Table 5 present, respectively, the results of the injections into control flow instructions and the results of injections into instructions randomly selected from the 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 4 characterize the effectiveness of PECOS in detecting errors when an error directly affects a control flow instruction, (i.e., effectiveness in detecting errors that 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 4: Cumulative Results from Directed Injection to Control Flow Instructions

The results in Table 5 provide an insight into how well PECOS performs when the corrupted instruction is not necessarily a control flow instruction but rather 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 a randomly corrupted instruction immediately crashes the process before the process executes the PECOS assertion block. Table 5 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 four conclusions:

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

Percentage of random errors in the DHCP server that manifest as control flow errors. 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 C% and that its coverage for detecting random errors in the text segment of the application is D%. 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 4 directed injections to control flow instructions), D = 31.5% (Table 5 random injections to the instruction stream); 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, effective data error detection mechanisms are important 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 Aborted	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 5: Cumulative Results from Injection to Random Instructions from the Instruction Stream

5.1 Discussion

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 improve availability. An important benefit of fast error detection is the ability to reduce the likelihood of error propagation. Our measurements show that between tens and thousands of instructions can be executed between the time at which the corrupted instruction is executed and the time at which the process crashes. This *window of system vulnerability* may be enough to corrupt significant system data, impact key system resources, or even crash the operating system. The results in Table 4 and Table 5 clearly indicate that PECOS is very efficient in reducing the system vulnerability window. The remainder of this section discusses PECOS's contribution to improving detection coverage of individual error categories. In the following discussion, some of the conclusions are based on detailed error injection results not shown in Table 4 and Table 5. The reader is referred to [BAG00] for these results.

5.1.1 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 these 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 these 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 detections. 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 PECOS extension (presented in Section 3.5) enables detection of these types of errors.

5.1.2 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 factors: (1) data corruption (e.g., the lease time that is loaded into a register is zero seconds) and (2) the application's taking 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. The following are three representative examples of the fail-silence violation of the DHCP server:

1. The server does not get the applicable record for the client host generated after phase 1 of the protocol, and therefore it 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. Since the server does not then stay silent forever but responds intermittently, this is an example of omission failure and is considered a fail-silence violation.
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 it to do so.

5.1.3 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 check and, consequently, reduce the overall detection latency.

5.1.4 Analysis of No Error Cases (Error Activated but 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:

1. 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).
2. 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 an application crash or fail-silence violation and is another strong indication of the need for efficient protection against data errors.

Finally, PECOS reduces the number of *no error* cases (i.e., execution of the erroneous instruction does not cause any error). This indicates that PECOS detects cases where the error would otherwise not impact the application. Since PECOS is a preemptive 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 later.

5.2 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 thread[12] or creating a new thread. Conservatively, the latter recovery process is less expensive in terms of time overhead by a factor of 10. Our best case, (taken on SPARC platform running Solaris operating system) measurements show that the time to create a process is on the order of 200 ms, while to spawn a new thread takes about 300 µs, 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 6 gives estimates of the downtime improvement for the midrange factor of 100 (the ratio between the time for process crash recovery and thread recovery), and Figure 7 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, in Table 4).

---

[12] For example, in the multithreaded implementation of the 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 to recover: we loose a single request, not the entire application.

	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%	$0.075 * t_{rec}$ ($0.07 * t_{rec} + 0.0047 * t_{rec}$)	

Table 6: Estimation of Reduction in Application Downtime

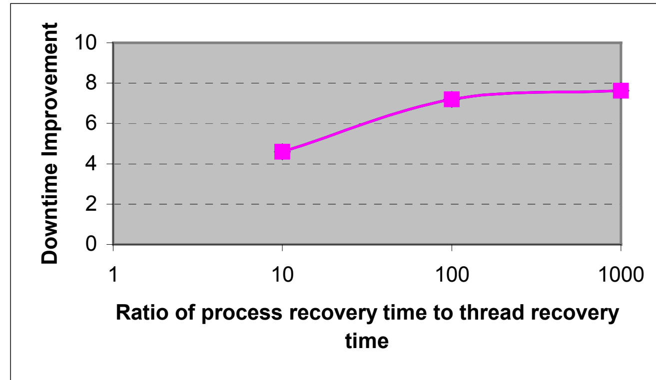

Figure 7: Downtime Improvement Due to Reduction in System Detection for Different Ratios of the Costs of Process Recovery and Thread Recovery

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.

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 *DHCPDISCOVER → DHCPOFFER*		**Phase 2** *DHCP REQUEST → DHCPACK/DHCPNAK*	
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, such as 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 engine. The results show overheads in the range of 5-20% if only the core protocol engine of the server is instrumented.

It may be recalled that the current PECOS implementation cannot protect the calls made to subroutines in dynamic libraries. As a result, 36% of the calls in the DHCP server code cannot be protected. It is found by analysis of the error injection logs that some of the cases of lack of coverage by PECOS are due to this deficiency.

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 previous studies on control flow error detection techniques do not provide overhead information; some simply provide a figure; only 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 instructions. 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 fail-silence violations, an additional error injection campaign was performed in which only the PECOS assertion block instructions were injected. The results are shown in Table 8. The results show that the PECOS assertion blocks do not cause any fail-silence violations. Consequently, it can be concluded that the PECOS instrumentation does not add any significant 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. The limited impact of corruption in the checking code can be attributed to the following two factors:

1. A single-bit error cannot change a non-control-flow instruction into a valid control flow instruction; this is due to the way in which the instruction set is encoded.
2. By design, the assertion block does not introduce additional control flow instructions and manipulates values in registers, and consequently, most of the errors result in corrupting the address or data of the destination or the source registers. Such a corruption is equivalent to a mismatch between the runtime target address of a CFI and the valid target addresses obtained from the program pre-analysis. Each such mismatch is detected by raising a floating-point exception.

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 8: Comparison of Directed Injections without PECOS and into PECOS Assertion Blocks

6 Conclusions

This paper presents a preemptive control flow error detection technique called PECOS. This software-based scheme embeds assertions in the application's assembly code to preemptively detect any illegal or incorrect control flow paths 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, client-server application, Dynamic Host Configuration Protocol (DHCP), was used as a target to evaluate the 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 the instrumented application. Results showed significant reductions in fail-silence violations, system detections, and process hangs in the

instrumented 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%.

Our conclusions have been reinforced by a follow-up study, which focused on the design and implementation of PECOS using the executable editing technique [YAN02]. The effectiveness of the technique was evaluated on selected benchmarks from the SPEC 2000 benchmark suite: *gzip*, *espresso*, *mcf*, *twolf*, and *vpr*. The measured average detection rate by PECOS was 41.1% (for manifested random errors). The performance overhead ranged from 53.5% (*mcf*) to 99.5% (*gzip*). Higher overhead than with DHCP can be attributed to the fact that the benchmark programs are rather computation-intensive, not client-server applications like DHCP.

Acknowledgments

This work was supported in part by NSF grant CCR-9902026, in part by a grant from Motorola Inc. as part of the Motorola Center for Communications, and in part by JPL under grant NASAJPL-961345. We thank P. Jones, J. Xu, W. Gu, and S. Narayanaswamy for developing error injectors used in the experiments.

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 Trans. on Parallel and Distributed Systems, vol. 10, no. 6, pp. 627-641, June 1999.

[ASU96] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principle, Techniques and Tools*, Addison-Wesley.

[AVR92] D. Avresky, et al., "Fault Injection for the Formal Testing of Fault Tolerance," Proc. 22nd Int'l Symp. on Fault-Tolerant Computing (FTCS-22), 1992, pp. 345-354.

[BAG00] S. Bagchi, "Hierarchical Error Detection in a SIFT Environment," Ph.D. Thesis, Univ. of Illinois, 2000.

[BAG01] S. Bagchi, Y. Liu, Z. Kalbarczyk, R.K. Iyer, Y. Levendel, L. Votta, "A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller," Proc. Int'l Conf. on Dependable Systems and Networks, 2001 (DSN 2001), July 2001.

[BER94] A. Bertolino, M. Marre, "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs," IEEE Trans. on Software Engineering, vol. 20, no. 12, 1994, pp. 885-899.

[CHA98] S. Chandra, P. M. Chen, "How Fail-stop Are Faulty Programs?," Proc. 28th Int'l Symp. on Fault-Tolerant Computing (FTCS-28), 1998, pp. 240-249.

[CHI89] R. Chillarege, R. K. Iyer, "An Experimental Study of Memory Fault Latency," IEEE Trans. on Computers, vol. 38, no. 6, pp. 869-874, June 1989.

[CHI95] R. Chillarege, "Orthogonal Defect Classification," Chapter in *Handbook of Software Reliability Engineering*, Ed. Michael R. Lyu, McGraw-Hill, 1995.

[FOR89] P. Forin, “Vital Coded Microprocessor Principles and Application for Various Transit Systems,” Proc. IFAC Conf. on Control, Computers, Communications in Transportation (CCCT’89), 1989, pp.137-142.

[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.

[HEN93] C. Hennebert, G. Guiho, “SACAM: A Fault Tolerant System for Train Speed Control,” Proc. 23rd Int’l Symp. on Fault-Tolerant Computing (FTCS-23), 1993, pp. 624-628.

[ISC00] Internet Software Consortium (ISC) Dynamic Host Configuration Protocol (DHCP), URL: http://www.isc.org/products/DHCP.

[IYE86a] R. K. Iyer, D. J. Rossetti, M. C. Hsueh, “Measurement and Modeling of Computer Reliability as Affected by System Activity,” ACM Trans. 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. on Parallel and Distributed Systems, pp. 292-301, September 1996.

[LAL85] P.K. Lala, *Fault Tolerant and Fault Testable Hardware Design*, Prentice Hall Int’l, New York, 1985.

[LAP92] J.C. Laprie (ed.), *Dependability: Basic Concepts and Terminology*, Dependable Computing and Fault-Tolerant Systems Series, Vol.5, Spinger-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, February 1991.

[MAD94] H. Madeira, J. G. Silva, “Experimental Evaluation of the Fail-Silent Behavior in Computers without Error Masking, ” Proc. 24th Int’l Symp. on Fault-Tolerant Computing (FTCS-24), pp. 350-359, July 1994.

[MAD00] H. Madeira, D. Costa, M. Vieira, “On the emulation of software faults by software fault injection,” Proc. of the Int’l Conf. on Dependable Systems and Networks, pages 417--426, New York, June 2000

[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, February 1988.

[MIC91] T. Michel, R. Leveugle, G. Saucier, “A New Approach to Control Flow Checking without Program Modification,” Proc. 21st Int’l 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 Int'l 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 Int'l Test Conf., pp. 461-468, November 1982.

[NAM83] M. Namjoo, "CEREBRUS-16: An Architecture for a General Purpose Watchdog Processor," Proc. 13th 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 Int'l Symp. on Fault-Tolerant Computing (FTCS-22), pp. 316-325, 1992.

[PAT01] S.J. Patel, Z. Kalbarczyk, R.K. Iyer, W. Magda, N. Nakka "A Processor-Level Framework for High-Performance and High-Dependability," Workshop on Evaluating and Architecting Systems for Dependability, 2001.

[RAO74] T.R.N. Rao, *Error Coding for Arithmetic Processors*, Academic Press, 1974.

[SCH86] M.A. Schuette, J.P. Shen, D.P. Siewiorek, Y.X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," Proc. 16th 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 Signatured Instruction Streams," IEEE Trans. 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, MA, 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] A. Thakur, "Measurement and Analysis of Failures in Computer Systems," M.S. Thesis, University of Illinois, UILU-ENG-97, September 1997.

[TSAI99] T. Tsai, M-Ch. Hsueh, H. Zhao, Z. Kalbarczyk, R. K. Iyer, "Stress-based and Path-based Fault Injection," IEEE Trans. on Computers, vol. 8, no. 11, November 1999, pp.1183-1201.

[TSA83] M.M. Tsao, D.P. Siewiorek, "Trend Analysis on System Error Files," Proc. 13th Int'l Symp. on Fault-Tolerant Computing, Italy, June 1983, pp.116-119.

[UPA94] S. Upadhyaya, B. Ramamurthy, “Concurrent Process Monitoring with No Reference Signatures,” IEEE Trans. on Computers, vol. 43 no. 4, pp. 475-480, April 1994.

[YAN02] Z. Yang, “Implementation of Preemptive Control Flow Checking via Editing of Program Executables,” M.S. Thesis, University of Illinois, 2002.

[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 Assissted Fault Injection Environment,” Proc. 3rd IFIP Working Conf. 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 Trans. on Computer-Aided Design, pp. 629-641, June 1990.