HIERARCHICAL ERROR DETECTION IN A SOFTWARE IMPLEMENTED FAULT TOLERANCE (SIFT) ENVIRONMENT

BY

SAURABH BAGCHI

B.Tech., Indian Institute of Technology, Kharagpur, 1996
M.S., University of Illinois at Urbana-Champaign, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

Abstract

A key problem besetting distributed applications is how to provide reliability guarantees to them, running on off-the-shelf hardware and software components. Chameleon is a Software Implemented Fault Tolerance (SIFT) middleware capable of providing adaptive fault tolerance in a COTS (components-off-the-shelf) environment with the capability to adapt to changing runtime requirements as well as changing application requirements. The thesis presents the architecture and implementation of a hierarchy of error detection techniques, which can be applied in a distributed SIFT environment. The error detection framework is implemented and demonstrated on the Chameleon testbed, though the principles are of general applicability in a message-passing-based distributed system. The thesis shows how the detection mechanisms are applicable to the components of the SIFT layer and how to extend them to applications executing on such a layer. A flexible mechanism for combining the different levels in the hierarchy to ensure the environment's adaptivity is presented. Next, the thesis presents some novel detection techniques targeted to different kinds of faults – control faults, data faults, message faults, etc. Particularly, a new approach called software signatures, which is used for validating the integrity of software components in Chameleon, is presented. The thesis also presents results from fault injection based assessment of the detection protocols and their performance measures.

Acknowledgements

I would like to thank Prof. Ravi Iyer for starting me on and guiding me in my quest for better error detection which is what this thesis is all about. Without Dr. Zbigniew Kalbarczyk, the thesis wouldn't have been half as thick, or may not even have seen the light of day. Heartfelt thanks go to my research group mates for seeing me through the long nights and short days of research and other interesting matters. Weining, Phillip, Zhenyu and Yiyuan were an invaluable source of help in the PECOS work, Keith in explaining the mysteries of Chameleon and the universe, and Balaji when I was starting off on my work on error detection. A big round of thanks to Shyam at whose laptop the last few pages of the thesis were written in breathless anticipation at New York. Beth and Fran were the greatest help in editing my papers and thesis, and, best of all, in hunting down Ravi when I needed to get hold of him. 9500 miles away in India they may have been, but my parents and my brother Sutirtha were by my side in spirit all through, egging me on. I owe it to all of you for what I am and what I can become.

Table of Contents

List of Tables

List of Figures

1 Introduction

We envision providing a software-implemented fault tolerance (SIFT) layer that executes on a network of heterogeneous nodes that are not inherently fault-tolerant and provides fault-tolerance services. Our current work on Chameleon is an effort at building one such system. In such an environment, it is critical to provide fault-tolerance to the application as well as to make the software infrastructure itself fault-tolerant.

In the networked environment, while the issue of application error detection has been fairly well studied, infrastructure[1] error detection less well understood. The currently employed techniques of timeouts or exceptions raised by the operating system often cannot provide fault containment to a reasonable level. In existing systems, fail silence of nodes and processes is often assumed away, with the exceptions of Delta-4 [POW94], Voltan [SHR92], and Mars [REI94] where hardware solutions were proposed. Field studies have repeatedly shown that in a distributed environment executing on off-the-shelf hardware components, the fail silence assumption is often violated [MAD94, THA97]. Hence, there is a need to explore more sophisticated detection techniques for the SIFT infrastructure. While some of these techniques have been studied in theory, it is as yet unclear what the relative significances of these techniques are in practice, or how they may interact among themselves to provide optimal system-wide error detection.

The purpose of this thesis is to describe the architecture and implementation of a hierarchy of error detection techniques, which can be applied in a distributed SIFT environment. Within this hierarchy, novel techniques for making a process self-checking and a node fail-silent are explored. The levels in the hierarchy are designed to enforce different fault containment boundaries – a process, a node, or a replication group. Within the hierarchy, the interactions between the levels as well as techniques within a level are explored for ways to minimize the overhead and maximize the overall system coverage. The protocols for escalating the error conditions from one level to another to optimize the detection scheme and minimize error propagation are clearly defined. The error detection framework is implemented and demonstrated on the Chameleon testbed, though the principles are of general applicability in a message-passing-based distributed system. The detection techniques in the hierarchy can be applied to the Chameleon infrastructure elements, called ARMORs, as well as to applications running in Chameleon.

The benefits of the detection hierarchy are:

1. Fail silent behavior at the node and the process level
2. Minimal error propagation between interacting processes on different nodes

[1] By infrastructure we mean the software layer that runs on the raw hardware and provides software-implemented fault tolerance to the application executing in the networked environment.

3. No need for hardware redundancy for achieving the above goals

The major new contributions of the thesis can be summarized as follows. First, it shows a way of arranging the error detection techniques in a SIFT environment (example case of Chameleon) in a hierarchy and how the hierarchy can be used to customize the fault tolerance characteristics of the system. Second, it introduces techniques for software-based signature monitoring that can protect applications against a wide variety of faults without taking recourse to replication. Third, it proposes several strategies to optimize the detection overhead and provides a mechanism of enabling the optimizations in a system.

The rest of the thesis is organized as follows. Chapter 2 presents related work in the area of building reliable distributed systems. Chapter 3 presents the hierarchical error detection framework in Chameleon and provides details of the techniques at each level. Chapter 4 presents the performance and coverage measurements from a detailed simulation study of a four-level detection framework. Chapter 5 presents the motivation, design and evaluation of three software signature schemes. Chapter 6 introduces a pre-emptive control flow error detection technique called PECOS. Chapter 7 presents a detailed fault-injection-based evaluation of PECOS applied to a client-server application called Dynamic Host Configuration Protocol (DHCP). Chapter 8 describes an integrated control and data error handling architecture applied to a wireless call processing environment in a digital telephone network controller. Chapter 9 summarizes the work and suggests future contributions.

2 Related Work

There is a large volume of work on theory and practice in reliable distributed systems. Many approaches for providing fault tolerance in a network of unreliable components are based on exploiting distributed groups of cooperating processes. Group communication systems have been proposed as useful building blocks for distributed applications, which help manage the complexity of such applications, as well as provide non-functional properties, such as availability and security.

Group communication systems have two main components:

1. Group membership service
2. Communication service

The Group Membership service enables processes to be considered as part of groups and provides tools for managing the groups. The management infrastructure includes handling group joins, leaves (either intentionally or by failing), and providing consistent group views to operational members of the group.

The Communication service provides various ordering guarantees among the messages delivered by the service. Common examples of guarantees include total ordering, causal ordering or FIFO ordering. ISIS [BIR94], Horus [REN96], Totem [MOS96], Transis [DOL96] provide tools for programming with process groups. By using these tools, a programmer can construct group-based software that provides reliability through explicit replication of code and data. Although reliability may be achieved using these approaches, "fault tolerance," Birman notes [BIR93], "is something of a side effect of the replication approach."

Here we try to summarize some of the needs of distributed applications that are not met or difficult to meet with existing Group Communication (GC) systems.

1. Most of the GC system implementations do not elaborate on the method of detecting failures of processes in the process groups. Implicitly, most of them assume that this can be detected through the lack of a message (like heartbeat response) from the process. This assumes the fail-stop model of failure for the processes. It has been shown in several studies [CHA98, MAD94] that the fail-stop model is violated in 10-46% of cases for off-the-shelf software components running on off-the-shelf hardware platforms. To meet this assumption in a realistic manner, some form of self-checking facility is required within a software component to detect a faulty state transition and stop the component from producing any further outputs. Also, most Group Communication work does not elaborate on the effects of faults on the infrastructure components. Just like applications can fail, so too can the infrastructure software, though maybe in slightly different ways (e.g., software Bohrbugs may be less

likely in the Group Communication stack). So, it is important to consider detection of faulty components in the system stack, and their subsequent recovery.

2. If Group Communication systems are to be used for providing fault-tolerance, a basic model one has to assume is that application-generic replication is the answer. In other words, an operation performed in a different environment at a replica process is not going to suffer the same fault as the original faulty replica process. It has been shown in a detailed study by Chandra *et al* [CHA00] that for three large, open-source applications: the Apache Web Server, the GNOME desktop environment, and the MySQL database, 72-87% of the faults are independent of the operating environment and hence deterministic and non-transient. Thus, recovering from these faults requires application-specific knowledge and simple state-based replication is not sufficient.
3. The underlying protocols that form the basic building blocks for the higher-level multicasting protocols available in GC systems have exposed time windows of vulnerability. For example, the 2-phase commit protocol which is commonly used to implement the multicast protocols is meant to ensure that a message is delivered to all operational processes in a group or none at all. However, there is nothing to prevent a process after it has communicated its decision to commit, to go back on its decision (a typical case of Byzantine behavior). Also, the 2-phase commit ensures that some data is committed, it does not ensure the validity of the data contents. Thus, a multicast primitive built on such building blocks will have a coverage associated with it, and that needs to be carefully analyzed.
4. Self-checking components generally have a smaller error detection latency than remote detection (as in through a voter). It has been shown by Madeira [MAD94] that a low latency error detection mechanism is effective in bringing down the incidence of fail-silence violations. For example, for the Z80 microprocessor, a signature monitoring scheme with an average detection latency of 0.4 msec brings down the fail-silence violation to 1.7%, as opposed to a technique called Error Capturing Instructions which has a latency of 47.8 msec and has 24.5% fail-silence violation. Also, self-checking mechanisms can help prevent process crashes by doing pre-emptive detection and localizing the effect of the fault (e.g., to a thread within the running process). This serves as an important motivation for the work presented in this thesis which explores techniques for building self-checking components with low error detection latency through software means.
5. While Group Communication presents a powerful tool for constructing reliable distributed systems, there is a gap between "building a Group Communication-based system" and "building a fault tolerant distributed system". Many of the fault-tolerant system services, like error detection, error recovery, voting, etc. have to be built in the group communication system. For example, for reintegrating a replica into a process group, there needs to exist tools to save states of existing replicas, freeze the processing by the functional replicas and transfer the state from a functional replica to the recovering

replica. Such recovery tools need to be built for any Group Communication system as for any other distributed system.

6. The configurations of the groups in GC systems are essentially static. For some applications, the communications among processes may follow sharply changing patterns. Thus, one group may want to communicate with another group. To handle such scenarios, most systems constrain that a single large group will need to be formed out of the two smaller groups for the entire lifetime. This has problems of scalability and incurs unnecessary overhead for the long periods of time when the communication is within each (small) group. AQUA [SAB99] is one distributed middleware that begins to address this issue through Connection Groups.
7. Due to changing runtime environments, or changing application needs, distributed systems need to be adaptive. Current GC systems support few hooks for adapting the system. A common mode of adaptation that is supported is to change the replication degree in order to be able to tolerate more faults from the same fault model. However, further levels of adaptivity, such as changing the replication style, are generally missing. AQUA is the only system we are aware of that permits some adaptation.

Most of the systems presented above provide an environment through which a programmer *may* construct a distributed application and provide fault tolerance through replication. While many assume the fail-silent behavior of the system components and in most of the cases do not even assess coverage of this assumption (an exception is Delta-4 [POW94]), Chameleon explicitly provides fault tolerance through a hierarchy of error detection and error recovery mechanisms. To our knowledge, none of the systems available allow a customization of the fault tolerance characteristics of the system to the extent in Chameleon – being able to vary the universe of faults it is able to tolerate, or to adjust the overhead of the detection mechanisms.

Apart from Group Communication, there are two main alternate philosophies of building large scale distributed systems:

1. Self-checking systems: The components in the system are made self-checking through a variety of detection techniques. Hence, any output from a system component is assumed to be correct (with the coverage of the associated self-checking techniques). This obviates the need for replicas and therefore of group communication.

2. Transaction-based systems: The system components support the transactional model which satisfies properties of atomicity (either all operations of the transaction are performed, or none are performed), permanence (the performed operations are permanent despite failures), and serializability (the transactions appear to have been performed in some sequential order).

The above two classes of systems are not antithetical to Group Communication based systems. In fact there have been efforts to combine them in some systems. Chameleon [BAG00] uses replication as well as self-checking, while Schiper et al [SCH96] have shown a path of combining the group-based model with the transactional model. In this thesis, the design and implementation of the self-checking features of Chameleon will be presented, along with the simulation of the replication protocols.

There are two ways of achieving the fail silence property in a distributed environment:

1. Make the system component self-checking, i.e., the component always produces either the proper output or no output [POW91].
2. Allow the system component to produce a wrong output (fail silence violation) and delegate detection and recovery to a receiving entity [BRA96]

In Chameleon, the first approach is adopted as the initial line of defense, and only in case of a coverage miss are distributed detection protocols spanning multiple nodes invoked. It can be argued that distributed systems be designed based on the second approach. However, in our experience delegating detection and recovery to the receiving end can lead to a significant increase in error detection latency and complicate recovery because of fault propagation[2]. Hence, substantial effort has gone into designing protocols that can be embedded into the Chameleon ARMORs or applications to make them self-checking.

There exist examples of systems which explicitly address the issue of fault tolerance and attempt to enforce fail-silence guarantees rather than assume it away. The MARS architecture uses a combination of special-purpose hardware (e.g., comparators) and software (e.g., double execution) approaches to provide fail-silence. Karlsson [KAR96] used three physical fault injection techniques (heavy ion radiation, pin-level injection, and electromagnetic interferences) to assess the coverage of error detection mechanisms in MARS. They found that the hardware and software mechanisms provided 98.7% coverage without duplicated execution. A software injection study for the same system [FUC97] turned up surprisingly different results about the efficiency of the different detection techniques. It showed fail-silence of 85% without application-level detection mechanisms, with the coverage becoming perfect when application specific data consistency and other checks were added. Another system that provides for building fail-silent components is Delta-4 [POW94]. A specialized Network Attachment Controller (NAC) is intended to make the nodes fail-silent. Fault injection results showed a fail-silence coverage of 85-90% [ARL90] and a later study [POW94] reported the improved fail-silence coverage of 99%. A fail-silent node that uses replicated processing with comparison/voting must incorporate mechanisms to keep its replicas synchronized, so as to avoid the states of the replicas from diverging. Synchronization at the level of

[2] Fault-tolerant systems, such as the offerings from Tandem, are substantially based on building fail-fast self-checking components, i.e., the first approach.

processor micro-instructions is logically the most straightforward way to achieve replica synchronism. Such hardware-based designs are increasingly expensive and difficult to implement. Hence there is much interest in providing fail-silent nodes through software. Voltan [SHR92] provides an architecture which uses a number of off-the-shelf processors on which application level processes are replicated to achieve fault-tolerance. However, an evaluation of the system [STO00a] has shown that the overhead incurred even for just duplicated process pairs is 76.6%, in addition to the extra hardware resources required. Chameleon provides an alternative approach to designing fail-silent system components purely through software means, without mandating process replication and at a fraction of the cost.

Many techniques have been proposed to monitor the inter-instruction control flow using signature monitoring. There has been increasing interest in protecting applications from control flow faults because control flow errors have been demonstrated to account for between 33% [OHL92] to 77% [SCH87] of all errors. Additionally control flow errors can lead to data errors, process crashes, or fail-silence violations. In distributed applications, fail-silence violations can have potentially catastrophic effects by causing fault propagation. This thesis presents a technique for control flow error detection that, unlike any existing technique, is pre-emptive in nature and is therefore effective in drastically reducing the incidence of process crash or fault propagation.

The problem of guaranteeing consistent information among multiple replicas is a much-studied problem in the design of reliable distributed systems. This was called the "Byzantine Generals Problem" by Lamport et al in [LAM82] which introduced a completely unconstrained fault model. The problem of locating the faulty processors in a Byzantine protocol is a less-studied problem. Some previous work has been done by Walter [WAL90], Shin and Ramanathan [SHI87], and more recently by Ayeb and Farhat [AYE98]. However, the algorithms proposed suffer from one or more of the following problems: they are off-line, additional rounds of message exchange are required, or there is no deterministic upper bound by when all malicious entities are detected. For our purposes, we have the simplifying assumption of a single failure among the participants in one run of the protocol, and we have a signature mechanism that imposes certain restrictions on faulty ARMORs (such as a faulty ARMOR never being able to generate its correct signature). Using these devices, we propose a simpler algorithm for error detection in the face of Byzantine faults. The issue of replica consistency occurs quite frequently in the literature of reliable distributed systems [MOS96, BIR94]. Our approach augments the data consistency approach with diagnosis performed during the process of consistently distributing data. The messages exchanged during the consensus protocol is used to diagnose errors in any of the participants. Also, we can employ our technique for unreplicated Chameleon processes using a subset of their state information to treat them as pseudo replicas.

To conclude, it has been acknowledged that to build reliable distributed applications, replication and

self-checking components are two complementary technologies. This work shows the path to combine both the technologies in one unified framework. There has been little previous work in providing self-checking mechanisms to components of a distributed system through software means. Part of the motivation for the current work comes from this need to provide tools to make off-the-shelf processes self-checking, capable of detecting faults from a wide fault model – data faults, control flow faults, message faults, etc. Existing work also does not address the issue of creating hierarchical fault containment boundaries so as to detect and repair hardware or software faults as close to the source as possible, and progressively escalating the fault in case of a coverage miss at the tighter fault containment boundary. This is of singular importance in optimizing the cost of error handling. Current work presents a detection hierarchy for structuring the error detection protocols in a distributed system, and provides a strategy for fault escalation.

3 Hierarchical Error Detection

In this section, we motivate the usefulness of structuring the error detection techniques in a distributed system in a hierarchy and present the hierarchy of error detection employed in Chameleon. Since the hierarchy is described as it is applied to Chameleon, a brief description of the Chameleon environment is provided first.

3.1 Chameleon: A Brief Description

Here, only a broad overview of the system is provided. For a comprehensive discussion of the design and architecture of the system, the reader is referred to [KAL99]. [BAG00] presents some of the early work on detection protocols in Chameleon, and [WHI00] discusses the recovery protocols.

Chameleon is able to:

- Support applications with different criticality requirements concurrently in the same networked environment and
- Adapt to runtime changes in availability requirements of the same application.

The properties of adaptivity and reconfigurability are derived from the Chameleon entities called ARMORs (**A**daptive **R**econfigurable **M**obile **O**bjects for **R**eliability). An ARMOR is a process that can be instantiated on any node in a network and send and receive messages from other ARMORs using a unified communication interface. Taken together, a set of ARMORs forms a distributed software infrastructure with the primary goal of providing fault tolerance services (error handling) to applications.

The flexibility of the environment is derived from the fine-grained composability of the ARMORs. The ARMORs effectively serve as the functional modules of Chameleon, and they may be easily combined to provide different fault-tolerant configurations. Furthermore, the ARMORs are location-independent, and designed to be able to execute on a large set of platforms (hardware and operating systems). They can migrate from one node to another to cope with node failure.

Three broad classes of ARMORs are defined:

Managers. Manager ARMORs oversee one or more of the other ARMORs in Chameleon. They are responsible for installing ARMORs, detecting and recovering from their failures, and initiating reconfiguration of subordinate ARMORs. The highest-ranking manager in the environment is the *Fault Tolerance Manager* (FTM), which interfaces with the user to accept the availability requirements, decides on a specific execution configuration (such as primary-backup execution) based on the user specifications, and instantiates appropriate ARMORs to perform the execution. A second class of managers is the *Surrogate Manager* (SM), which is responsible for executing a particular application in

the mode decided upon by the FTM.

Daemons. Daemon ARMORs are installed on every node participating in Chameleon. Daemons act as the gateway of a node for all ARMOR communication, perform message routine between ARMORs, and provide error detection for locally installed ARMORs.

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, overseeing it during execution, and finally communicating its results for further processing. Another example is the Heartbeat ARMOR, which can send heartbeats to nodes as well as to other Chameleon ARMORs.

ARMOR Structure

An ARMOR is composed of a set of basic building blocks, which we call *elements* and *compounds*.

- Elements are the simplest building blocks in Chameleon. They provide specific services in response to messages sent to the ARMOR. These functions are invoked by the ARMOR in response to an incoming message. The element can be looked upon as replaceable units of functionality in the Chameleon environment. Elements also store element-specific data in them, which are not shared with other elements.
- Compounds are composed of one or more elements, and by design, the compound's constituent elements can be substituted during run-time, providing for dynamic reconfigurability. Compounds can be thought of as active entities that provide a unique interface through which an element's functionality is invoked for processing incoming messages. Message subscription and delivery services are provided within the compound layer. These respectively enable elements to subscribe to messages they can process, and the compound to deliver incoming messages to elements that have subscribed to that particular message operation type.

The error detection techniques presented in this thesis are implemented as distinct elements in Chameleon. They are available in the element repository to be included into appropriate ARMORs. Since the elements do not directly communicate with one another (they have to pass through the compound's message delivery functionality), and do not share data, therefore an ARMOR can be reconfigured with different sets of elements if required. The level of indirection in accessing the functionality of the elements within an ARMOR makes the reconfigurability of ARMORs possible. The ARMORs can be reconfigured statically, i.e., a generic ARMOR can be specialized by populating it with specific elements. The ARMORs can also be reconfigured at run-time, e.g., to change the detection features provided by an ARMOR from tolerating just crash failures to tolerating more malicious failures. Hence, the detection functionality in Chameleon does not come *hard-coded* into the environment. Rather, the functionality can

be introduced into an ARMOR incrementally as required and paying the overhead for just the detection functionality required for the particular workload being supported.

ARMOR Messaging

ARMOR processes communicate via message passing. Each incoming message results in a new thread being spawned to process the message. In this thread, one or more elements are invoked to process the message. The thread handling the incoming message terminates when the ARMOR sends out a message, or completes processing the message. Messages are destined to an ARMOR, not to specific elements within an ARMOR. This level of indirection enables Chameleon to reconfigure ARMORs, independent of other ARMORs in the system. The message routing between ARMORs is performed by the daemons resident on each Chameleon node. The functionality of the error detection elements is invoked through Chameleon messages, as in other elements.

3.2 Hierarchical Structure: Principle and Motivation

The different detection techniques employed in a distributed environment are characterized by the following parameters:

i. *Types of errors caught*. An error in an internal data structure of the ARMOR is best detected internally in the ARMOR, while a fault in the operating system of the node on which the ARMOR is executing is best detected by an ARMOR on another node.

ii. *Latency*. Typically, detection close to the site of the error will minimize the latency.

iii. *Cost* (in terms of computation cycles). A local detection mechanism will typically be less expensive than a remote detection that involves message exchange between a distributed group of cooperating processes.

iv. *Coverage*. Different techniques, or rather different combinations of techniques, may give different coverage factors.

The next logical step then is to try to structure the error handling mechanisms in a hierarchy of logical levels where each level is characterized by the above parameters. We decide to use the location of execution of the techniques as the basis for the hierarchy. In Chameleon, we propose a four-level hierarchy for the error detection mechanisms.

Since adaptivity is a key criteria of the Chameleon system, the error handling in Chameleon must also be adaptive with respect to application requirements and run-time environment requirements, e.g., the types of faults being experienced in the system, the reliability characteristic desired of the system, etc. An application that needs a higher throughput and lower availability may want to turn off the higher overhead detection levels, while a more critical application may want to leave such levels on. This imposes two

conditions on the design of the error detection hierarchy. First, the individual levels as well as the individual techniques within a level can be turned on or off, or their invocation conditions suitably changed at runtime. Second, the levels must be designed independently enough so as to allow different compositions of the individual levels. While one level may use the information from another level for optimizing its execution, there should be no mandatory dependency between any of the levels, which would render one without the other useless. The detection hierarchy in Chameleon obeys both these principles. The four-level detection hierarchy in Chameleon is presented in Figure 1. A level is denoted lower than another if it is implemented closer to the ARMOR being monitored. Level 1, the lowest level, consists of detecting errors internally in the ARMOR. Level 2 consists of detection by the Daemon installed on the same node as the ARMOR. The same Daemon is responsible for monitoring all the locally-installed ARMORs. The two levels 1 and 2 interact to provide error containment within the local node. Levels 3 and 4 utilize multiple ARMORs for doing the error detection, which may be on separate nodes. These levels use message passing between distributed processes to perform the detection. The Level 3 message exchanges take place between an ARMOR and its replicas (e.g., the FTM and the backup FTM), while those of Level 4 take place typically between an ARMOR and its replicas, and a manager and its replicas (e.g., an Execution ARMOR, a Surrogate Manager and a replica of the Surrogate Manager).

The different techniques within the levels are implemented in separate elements. Multiple techniques may be incorporated within a single element. For example, the livelock detection and the assertion check handling in level 1 are both provided by the Monitor Element (see Sec. 3.3) while the protocols in levels 3 and 4 are implemented in a single element. As a result of this design, and because the elements can be inserted or removed from ARMORs statically or at runtime, the detection framework in Chameleon is configurable. By inserting or removing the appropriate elements in an ARMOR, the required levels and techniques within a level may be activated or deactivated. Further customization of the invocation conditions of the levels and the techniques within a level is enabled through the Optimizer Element (see Section 3.8).

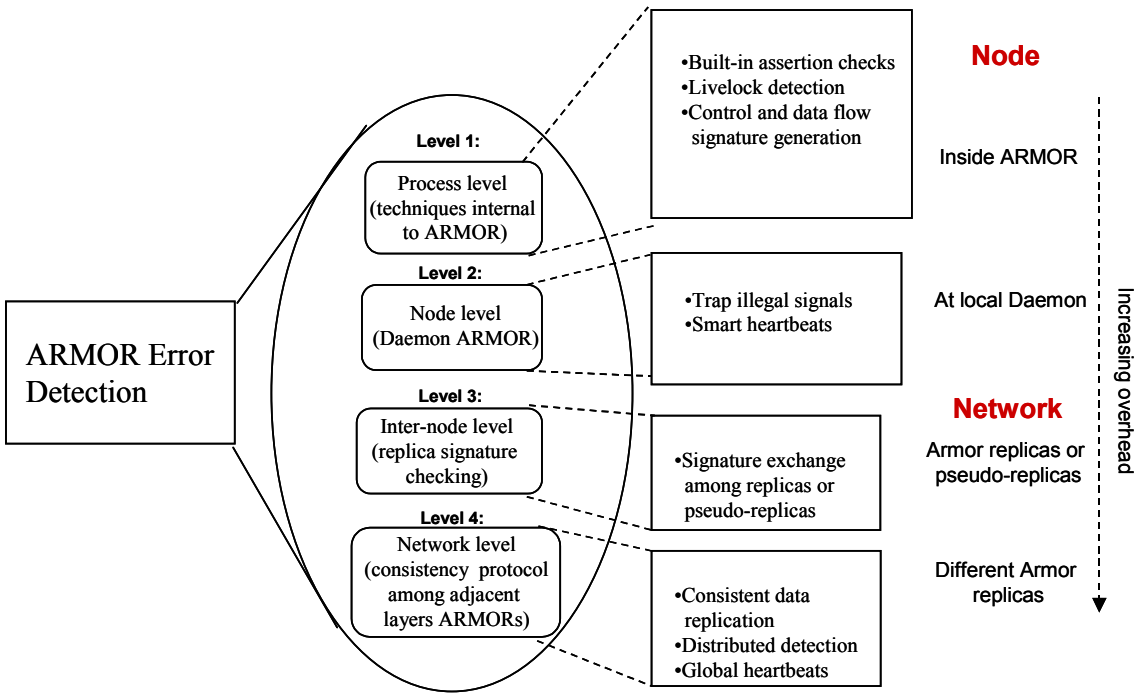


Figure 1. Error Detection Hierarchy in Chameleon

Only the intra-node levels, i.e., levels 1 and 2 in are implemented in the current Chameleon testbed while all the four levels have been simulated. In the rest of this section, we provide an outline of the techniques in each level of the Chameleon hierarchy. ARMOR signatures which are used in all the four levels of the detection hierarchy will be discussed in detail in Chapter 5.

3.3 Level 1: Internal ARMOR Techniques

In level 1 of the hierarchy, the error detection techniques are incorporated within the ARMOR. This level is intended to detect control and data flow errors in the ARMOR, livelocks and some software defects in ARMOR elements.

The techniques in this level can be incorporated into an ARMOR through a specialized element called the *Monitor Element*. The Monitor Element can activate various error detection techniques depending upon the support that is provided by the element being monitored. Table 1 summarizes the techniques available in this level and the extent of support required from the ARMOR element. The different control signatures described in Chapter 5 are also included in this level.

The Monitor Element can be inserted to endow an unprotected ARMOR with various detection facilities. An ARMOR does not need the Monitor Element for its normal functioning and it only needs to be inserted into the ARMOR depending on the criticality requirement of the application. The conditions under which the Monitor Element invokes the different detection mechanisms can also be modified depending on the application requirements using the Optimizer Element [see Sec. 3.8.1], another

specialized element that may be inserted within an ARMOR that has detection techniques enabled. What amount of support must be provided by an ordinary element so that the Monitor Element can enable a detection technique varies widely - from only initialization time support (livelock checking) to runtime support (test message activation).

Detection Technique	*Support required from Element*	*Trigger*	*Description of Technique*
1. Assertion check	Provide hooks for Assertion Handlers (AH)	A hook for AH in the Element code	Monitor Element instructs Element on appropriate type of assertion (possibly using hints from the Element) and attaches appropriate assertion handler at the point where hook is provided. (See Figure 2)
2. Livelock checking	Provide pointer to the lock variable the Element uses to access local state variables	Time i.e. with a certain periodicity	Because of multi-threaded nature of ARMOR architecture (each message processed in a separate thread), Element typically grabs a mutex lock before accessing its local data structure. Monitor Element flags livelock if lock variable continually held for greater than threshold period of time.
3. Element Activation with Test Messages	Element provides routine to handle the test message	Smart heartbeat from Daemon	In response to a smart heartbeat sent to the ARMOR, the Monitor Element activates all the elements with a test message and test pattern taken from its internal dictionary. If expected output is not generated by any element, it is deemed erroneous.
4. Signature Checking	Element registers its golden signature at initialization time	Output inter-ARMOR message is generated	The element generating the message, it's coarse-grained signature is checked for control flow error, i.e., it is checked if the input-output message pair is in the Golden Signature Table. (See Figure 2)

Table 1. Detection Techniques in Level 1

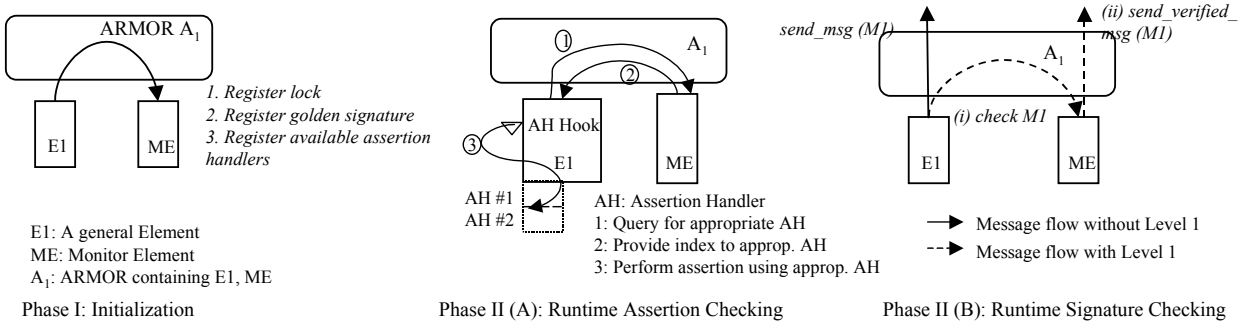


Figure 2. Initialization and Run-time Checking Phases in Level 1

3.4 Level 2: Detection by the Local Daemon

In Level 2 of the hierarchy, the error detection is performed by the local Daemon. Recall that each node has a resident Daemon on it which performs this level of error detection for all the ARMORs installed on this node. This level is expected to capture errors that escape the detection mechanisms internal to the ARMOR or errors that compromise the entire ARMOR. The basic techniques available at this level are summarized in Table 2.

Detection Technique	*Trigger*	*Description of Technique*
1. Smart heartbeat	Synchronous i.e. time triggered, or Asynchronous, i.e. in response to query from ARMOR's manager	A heartbeat polling message sent to the ARMOR in response to which elements are activated with test patterns. Alive response signifies health of the ARMOR, not just liveness.
2. Exception handling	When some illegal operation by the ARMOR raises an OS signal	The exception is captured by the Daemon through the inter-process communication channel established between Daemon and ARMOR.
3. Signature Checking	Inter-node message is generated	Since the Daemon acts as the forwarder for all messages, when some message is generated for a remote ARMOR, the source ARMOR's control flow is checked through the fine-grained signature.

Table 2. Detection Techniques in Level 2

3.5 Level 3: Detection among ARMOR Replicas

The level 3 mechanism involves exchange of signatures – both data and control – among ARMOR replicas. Level 3 is the first level in which detection does not necessarily occur on the local node on which the ARMOR is executing (if we assume that different replicas of an ARMOR execute on different nodes, which is reasonable). Thus, the mechanisms at this level will not be susceptible to malfunctioning of the node.

For the purpose of this protocol, ARMORs need not necessarily be replicated. A distinguishing characteristic of this protocol compared to existing replication-based detection protocols is that we do not require an ARMOR to be replicated in order to take part in it, but instead we can treat different ARMORs as *pseudo replicas* and use some common information to enable this protocol. To understand what we mean by common state, and how the pseudo replicas are chosen to act as participants in the protocol, it is necessary to introduce the concept of ARMOR state.

3.5.1 ARMOR State

The ARMORs in Chameleon are arranged in a tree-like hierarchy according to the manager-subordinate relationship, with the parent being the manager and the child the immediate subordinate ARMOR. A node in the tree may represent a single physical ARMOR, or a logical ARMOR, which means an ARMOR and its replicas[3]. The ARMOR hierarchy is presented in Figure 3. For example, the logical ARMOR consisting of {A(2,2,1), A(2,2,2), A(2,2,3)} is the parent of ARMOR A(3,2,1).

[3] A Chameleon ARMOR may be replicated for increased reliability. The replication degree may differ from ARMOR to ARMOR both within the same rank as well as across ranks. The replicas may be configured as active or passive at any point of their execution.

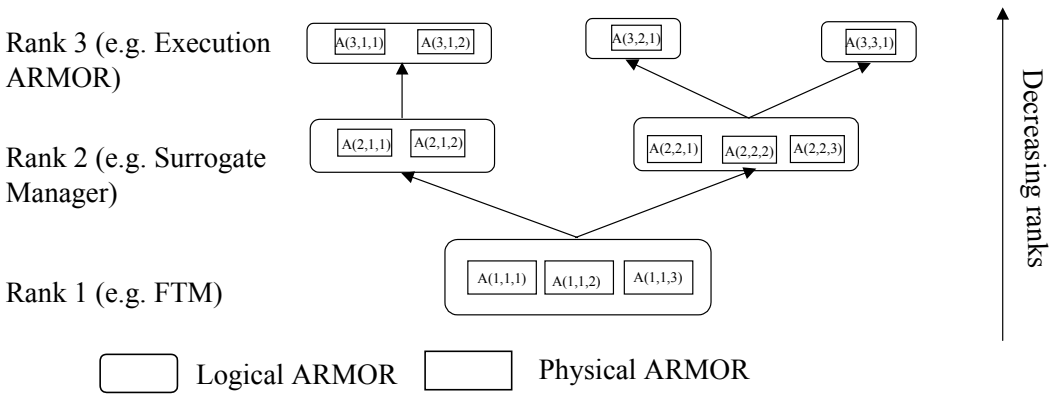


Figure 3. ARMOR Hierarchy showing Parent-Child Relationship

Every Chameleon ARMOR maintains some state for its normal functioning (e.g., FTM has to maintain a list of available nodes in order to be able to allocate nodes for new ARMOR installation), and optionally for error recovery (e.g., the Surrogate Manager maintains state of its subordinate Execution ARMOR so as to be able to recreate a new instance if it fails). At the minimum every ARMOR has the *ARMOR Location Table* (LT) as its state. The LT is the state variable that maintains information about an ARMOR's subordinate ARMORs. Each entry in the LT looks like:

{Subordinate ID, Daemon ID, Manager ID}

where Subordinate ID is the ID of the subordinate ARMOR, Daemon ID is the ID of the Daemon on the node of the subordinate ARMOR, and Manager ID is the ID of the manager of the ARMOR.

Thus, the LT at the ARMOR (2,2,1) will have the following entries:

{(3,2,1), D(3,2,1), (2,2,1)}; {(3,3,1), D(3,3,1), (2,2,1)}

where D(A) denotes the daemon on the node of ARMOR A.

The LT at an ARMOR follows a superset relationship according to its placement in the hierarchy tree, given by:

$$\text{If } (A_1 \in \text{Desc}(A_2)) \text{ Then } S(A_1) \subset S(A_2) \text{ ..... (1)}$$

where, A_1, A_2 are ARMORs, $\text{Desc}(A_2)$ includes all the ARMORs in the sub-tree rooted at A_2, S(A) denotes the state of ARMOR A. Also, the following stronger condition holds:

$$S(A) = \bigcup_{A_i \in \text{Desc}(A)} S(A_i) \text{ ..... (2)}$$

Thus, the LT at the FTM is the union of the LTs at all the other ARMORs in Chameleon. The property of ARMORs sharing state can be used to form the participants for the level 3 detection protocol, which are not exact replicas. ARMORs which share state are considered as pseudo replicas and used in executing the level 3 protocol, utilizing the common state between them to form the data signature.

3.5.2 Level 3 Protocol Description

A simplified pseudo-code for the detection protocol is presented in Figure 4. There are two timeouts associated with this level – one is the periodicity of activation of the protocol (τ) and the other is the timeout for which it waits to receive the signatures from the other replicas (σ). There is a separate element that does the reception and sending the signatures. Since, the data signature is also used in this protocol, it must be made atomic with respect to state changes in the ARMOR, as otherwise the replicas will have inconsistent state and their data signatures will not match. The trigger for the protocol is either the *period_timer* expiring, or a message being received from a replica signifying that it has started a round of the protocol. In response to the trigger, the ARMOR concatenates its control signature and data signature and sends it to each of its replicas. Then, it waits to receive the signatures from all its replicas. Any signature not received within timeout is flagged as a missing value. After the timeout, a comparison is made between the golden and the received signatures for each replica and a mismatch is flagged as error.

```
Protocol has periodicity τ
Self.ID = (M,N,P)
Set of replicas = (M,N,x), x=1,…,R
Current round = p;

do {
Set Period_Timer = τ;
Start timer;
When (Period_Timer expires OR Signature message of round p is received from a replica) {
   Lock element to ensure protocol atomic w.r.t. state change;
   For each ARMOR (M,N,x), x=1,…,R, x≠P {
     Send(Sig_Self = CSig_Self + DSig_Self: p);
  }
  Unlock element;
}

Set Timeout_Timer = σ;
Start Timeout_Timer;
Timed_Wait(σ, Sig(M,N,x):p, ∀ x=1,…,R, x≠P);
For each ARMOR (M,N,x), x=1,…,R, x≠P {
  If (Sig(M,N,x) == SIG(M,N,x)) Then
                OKAY!
  Else
                Notify Manager of (M,N,x) of fault;
}
Current round = p+1;
} while (TRUE);


CSig(x,y,z) & DSig(x,y,z) : Control and Data Signature respectively of ARMOR (x,y,z)
SIG(x,y,z) : Golden signature of ARMOR (x,y,z)
```

Figure 4. Detection Protocol for Level 3

The skew between two replicas can never become greater than the time for message transmission from one replica to another, if we assume no message loss. In case of message loss, the skew will be bounded by (τ+skew between replicas in sending R-1 signature messages). The receiving ARMOR does a comparison between the received signature and the golden signature of each of its replicas. Any mismatch or a missing value is flagged as error to its (and its replica's) manager. Under the assumption of a single

ARMOR failure, the manager will receive error notification from all of the other (R-1) replicas.

3.6 Level 4: Detection Among Adjacent Ranking ARMORs

This is the highest level of error detection in Chameleon. This level incurs the most overhead among the four levels, and is also least frequently invoked. The Level 4 error detection involves the participation of ARMORs from at least two adjacent ranks in the ARMOR hierarchy. The goal of keeping consistent state among ARMORs coupled with checking integrity of the replicas is sought to be achieved through the Level 4 protocol. The two goals are mapped to two logical phases of the protocol. A distinguishing characteristic of this protocol compared to other Byzantine detection protocols is that it utilizes the messages exchanged during the consistency phase to do the diagnosis as well. This results in online diagnosis and cuts down on message traffic in the system.

3.6.1 Protocol Requirements

The Level 4 protocol (L4) needs to satisfy the following requirements:

- **Consistency requirement.** The protocol causes the correctly functioning ARMORs which participate in the protocol to agree upon a consistent update information. This is required to maintain the superset relationship in ARMOR state according to its position in the hierarchy, mentioned in Sec 3.1, which is required so that the manager can recover from its subordinate ARMOR's failure.

- **Diagnosis requirement.** Assuming that at most one of the participating ARMORs is in error, the protocol is able to diagnose the faulty ARMOR. This is required for the protocol to satisfy its basic responsibility for error detection in level 4 of the detection framework.

3.6.2 Protocol Participants

The protocol is modeled as a Byzantine agreement protocol augmented with error detection capabilities. The Authenticated Messages variant of the protocol is used. The theoretical results show that to tolerate *m* Byzantine faults, there need to be at least (*m+2*) participants. Therefore, to tolerate a single fault, there need to be at least 3 participating ARMORs. Generally, L4 is executed among a parent and its replicas and one of its children and its replicas. In Figure 5, A(1,1,1), A(1,1,2), A(1,1,3), A(2,1,1), A(2,1,2) could participate in one run of the protocol. The choice of the participants is done at runtime, but the decision as to who participates in the protocol is based solely on the position of the ARMOR in the ARMOR hierarchy. A common example of a set of participants is the Surrogate Manager, FTM and Backup FTM. The state information that is exchanged in the protocol is the common state that is shared between the protocol participants. By the state relationship of (1), the subordinate ARMOR's state will be a subset of the parent ARMOR's state.

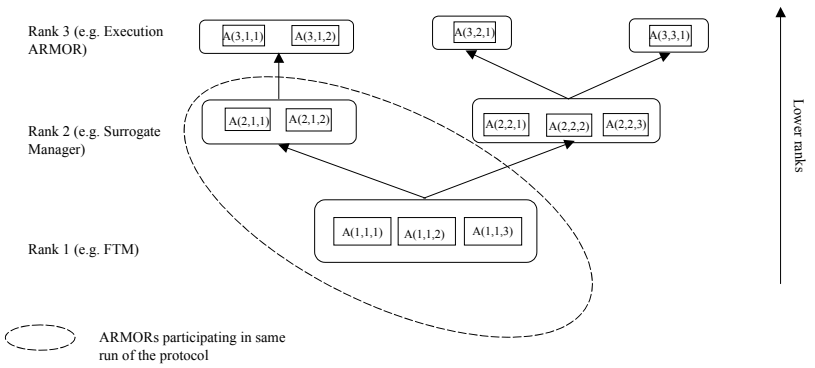


Figure 5: Example of Participating ARMORs in a Level 4 Protocol

3.6.3 Protocol Triggers

The Level 4 protocol is triggered in the following cases:

- **State change in an ARMOR**. The participants on such a trigger are the ARMOR in which the state change has occurred, its replicas and its parent and its replicas (all of whom to which the state change must be communicated). A state change takes place, for example, when the ARMOR installs or uninstalls another ARMOR. It is to be noted that the protocol is triggered before the actual state change has occurred. For example, if an ARMOR is to uninstall one of its subordinate ARMORs, it will execute this protocol before the operation.

- **Error escalation**. In addition, if a Level 3 error detection mechanism indicates an error, but is unable to diagnose the errant ARMOR, then we invoke phase 2 of our Level 4 protocol. This would happen in the case of 2 replicas, each of which would report to the manager that the other is in error.

3.6.4 Protocol Phases

The Level 4 protocol has two logical phases: (1) consistency phase and (2) diagnosis phase. During any particular run, either of the two phases may be omitted depending on the invoking condition.

In the **consistency phase** the participants exchange state update messages in an effort to reach a consistent state. In the **diagnosis phase** the participants determine the erroneous entity, if any, among them by examining the messages exchanged in the first phase or by an exchange of the signatures among the participants. The detailed algorithm used can be found in [WHI98].

3.6.5 Global Heartbeat

Another technique available at Level 4 of the error detection hierarchy is the global heartbeat mechanism, which is used to

- heartbeat the nodes in the environment (the heartbeat sent out to the node is echoed back by the daemon)

- heartbeat specific ARMORs. This corresponds to the Smart Heartbeat discussed earlier in Sec. 3.3. A

response to the heartbeat is not just indicative of the aliveness of the ARMOR but also of the health of elements that constitute the ARMOR.

3.7 Fault Model

A fault model for ARMOR failures is presented in Table 3. It is organized in a four-level hierarchy to draw a correspondence with the detection technique that is most likely to detect it first. Some of the ARMOR faults may be manifested in multiple levels. For example, an ARMOR crash may be detected by the timeouts in levels 3 or 4, but the most direct detection is by the lack of response to a heartbeat query message sent out by the Daemon in level 2, or a raised exception. The hardware faults can also be mapped to the fault model hierarchy. Node crash is detected by the absence of global heartbeat (of level 4) from the Daemon as well as from all locally resident ARMORs. The Daemon faults [A]-[D] are detected like for any other ARMOR. Daemon fault [E] is detected when smart heartbeat query is sent to the Daemon by its manager. Daemon fault [F] is detected through the above smart heartbeat query.

Level	*Fault*	*Detected By*
Level 1	[A] Message delivery functionality of ARMOR experiences transient fault	Assertion check in the Element asserting the received message (since a message not subscribed to is delivered)
	[B] Element's internal state inconsistent (w.r.t. an assertion, e.g., table size greater than some pre-asserted value)	Assertion check in the Element asserting on the state variable
	[C] Element is hung (e.g., in livelock)	Monitor Element's livelock checking
Level 2	[D] Element died	Smart heartbeat sent by the Daemon to the ARMOR
	[E] ARMOR as a whole in error (maybe its message delivery routine is faulty)	Smart heartbeat sent by the Daemon to the ARMOR
	[F] ARMOR's internal state inconsistent (not w.r.t. a replica, but just a local inconsistency, like some element being dropped from its subscription table)	Smart heartbeat from the Daemon or the Fine-grained Signature Checking
Level 3	[G] ARMOR data flow error	Level 3 Signature Checking
	[H] ARMOR's state inconsistent (w.r.t. its replicas)	Level 3 Signature Checking
Level 4	[I] ARMOR's state inconsistent (w.r.t. its manager or subordinates or replicas)	Level 4 Consistency Checking
	[J] Byzantine Fault	Signature Checking or Consistency Checking
	[K] Daemon crash	Global Heartbeat

Table 3. ARMOR fault model

3.8 Optimizations

There are three different optimizations proposed for the Chameleon detection framework:

1. Buddy Optimization
2. Probabilistic Escalation
3. Speculative Execution

The different detection levels impose differing overheads on the execution of the Chameleon environment. Since each detection technique is implemented in a separate element, depending on the needs of the application, some detection elements may not be used to populate the ARMOR at all. However, it is useful to have a more fine-tuned control over the detection levels as well. This can take one of the following forms:

1. Modifying the trigger conditions for invocation of techniques within a level and
2. Specifying how the execution of one level affects the invocation condition of another level.

These two motivations give rise to the first two optimization strategies – Buddy Optimization and Probabilistic Escalation, respectively, while the third strategy – Speculative Execution - concerns removing the checking overhead from the critical execution path.

3.8.1 Buddy Optimization

Buddy Optimization is an intra-level optimization technique. It is specified quantitatively by a *Domain of Influence Formula* (DIF) and is enforced using an *Optimizer Element*. The DIF of any technique T1 with respect to another technique T2 represents the extent over which T1's execution will affect the invocation condition of T2. The extent of influence is specifiable in terms of control blocks, messages, or time. For example, a mathematical specification of the DIF of T1 w.r.t. T2, given in terms of control block, is

$$\text{DIF}(T2|T1) = \begin{cases} 0, & CB(0) \\ 1, & CB(1,\ldots) \end{cases}$$

The DIF is the scaling factor with which the usual number of invocations (i.e., in the unoptimized case) of T2 will have to be multiplied to arrive at the actual number of invocations because of the execution of T1. The parameter within parentheses [0 in CB(0)] refers to the offset. In the above example, if T1 has been executed in a specific control block, then T2 will not be invoked in the same control block, i.e., block whose offset is 0 from the current control block. But T2 will be invoked as usual in other CBs outside the current one, wherever it was supposed to be invoked. Currently, this optimization is designed to support only the scaling factors of 0 and 1 (i.e., either no invocation at all, or the usual number of invocations, respectively).

Optimizer Element. The Optimizer Element can optionally be included in any ARMOR. The Optimizer Element was designed with two motivations.

1. Optimize the number of invocations of the detection techniques
2. Reconfigure the fault tolerance characteristic of the system by appropriately activating or deactivating any of the techniques

The Optimizer Element has two broad functions:

1. Act as the repository of the DIF specifications provided by the ARMOR's manager. The DIF specifications are provided at the time of installing and configuring the ARMOR and also in response to any changes in the runtime condition that necessitates a change of the detection strategy.

Keep log of activations of techniques within a detection level and enable or disable a technique depending on the log and the DIFs. A snapshot of an example repository of DIFs at an Optimizer Element along with the explanation is given in

Figure 6. The basic algorithm implemented at the Optimizer Element is presented in

Figure 7. This algorithm can be made more efficient by having the DIF being cached locally by the Monitor Element, which is executing the actual detection technique, instead of it having to ask the Optimizer Element every time. When there is a modification to the DIF repository at the Optimizer Element, it updates the cache at the Monitor Element.

Technique	*Technique ID (Level#,ID)*
Assertion Check	(1,1)
Coarse-grained Signature Check	(1,2)

DIF Specification {a1,a2} where a1's invocation condition is being modified due to a2	Explanation
1. DIF{(1,2)|(1,2)} = 0, Msg(m) 1, otherwise	Do the Coarse-grained Signature Check every *m* messages
2. DIF{(1,2)|(1,1)} = 0, Msg(1) 1, otherwise	If Assertion Check done while processing a message, don't check Coarse-grained Signature on that message

Figure 6. DIF Specification in an Optimizer Element

1. Technique T1 to be activated by Monitor Element (ME).
2. ME sends a message to Optimizer Element (OE) to ascertain if activation condition is met.
3. OE selects the DIFs from the repository with T1 as the first tuple.
4. OE does an AND-ing of the constraints from all such DIFs.
5. If result is 0, it means activation condition is not met. If result is 1, it means the technique can be activated. OE logs the activation and sends message to ME.

Running Example
T1 = (1,2), m=2
Activation log: [Message #, Technique Executed]
[1, (1,1)], [2, (1,1)], [3, (1,2)], [4,(1,1)]
Decision to be taken by OE: Whether [5, (1,2)]?
Step 3: Both DIFs selected
Step 4: DIF#1 returns TRUE, DIF#2 returns TRUE.
Composite result TRUE.
Step 5: OE tells ME to activate (1,2) and logs [5, (1,2)]

Figure 7. Algorithm in an Optimizer Element

3.8.2 Probabilistic Escalation

Probabilistic Escalation is an inter-level optimization technique. It is also specified in terms of a DIF specification, but this time the two tuples belong to different levels and the domain is specified in terms of time. The basic premise of this optimization technique is that one might avoid the invocation of detection techniques in one level if one can place sufficient confidence in the diagnosis performed by another level. The confidence placed will naturally degrade with time, and this is captured by a confidence curve. If the decision whether to invoke a particular level is being taken at time t_i, the value of the confidence is read off from the curve for the time difference between t_i and time of invocation of previous level. If it falls below the confidence threshold, then the level is invoked, otherwise it is not. This is shown schematically in Figure 8. The confidence curve and the threshold confidence are system parameters that are specified

to the Optimizer Element at the time of installing the ARMOR or in response to any change in the runtime environment that necessitates a change of the detection strategy.

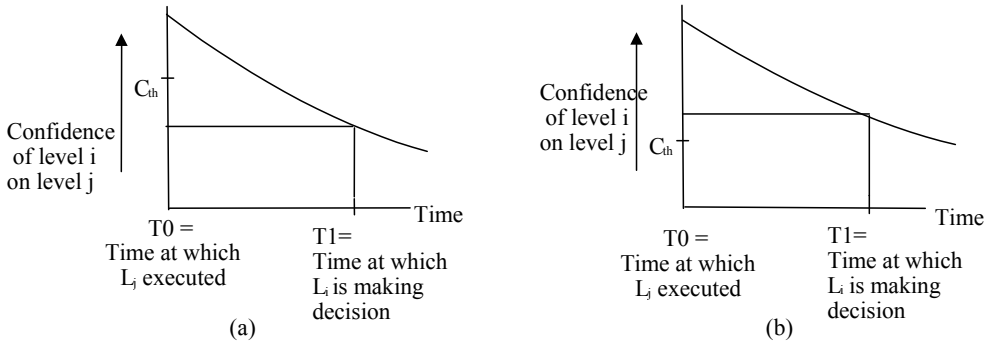


Figure 8. Probabilistic Escalation from *level j* to *level I*
(a) Decision: Li will be invoked; (b) Decision: Li will not be invoked

3.8.3 Speculative Execution

The basic premise of this optimization is to defer the checking and to speculatively do message send and execution. Typically, when an inter-node message is generated by an ARMOR, the Daemon does a fine-grained signature check of the source ARMOR, and only then does it forward the message. Under this optimization, it would send out the message before the checking and send a follow-up message after the checking is complete, either validating or invalidating the speculative message. The speculative message may trigger further message sends by the receiving ARMOR. A limit on the propagation of a speculative message is obtained by attaching a hop count to such messages. The hop count is decremented at every destination ARMOR, and when it reaches 0 it does not send out any more speculative messages but waits for the follow-up message. The schematic diagram for the optimization is presented in Figure 9. It shows an element E1 speculatively sending out message M1 to element E2. E2 does the normal processing on the speculative message, but all state updates are made to a temporary cache. On receipt of a follow-up message, the cache is committed to the stable state variables. It shows the error-free case only. Various failure scenarios and a quantitative analysis of the performance benefit of this optimization is part of the ongoing work on Chameleon.

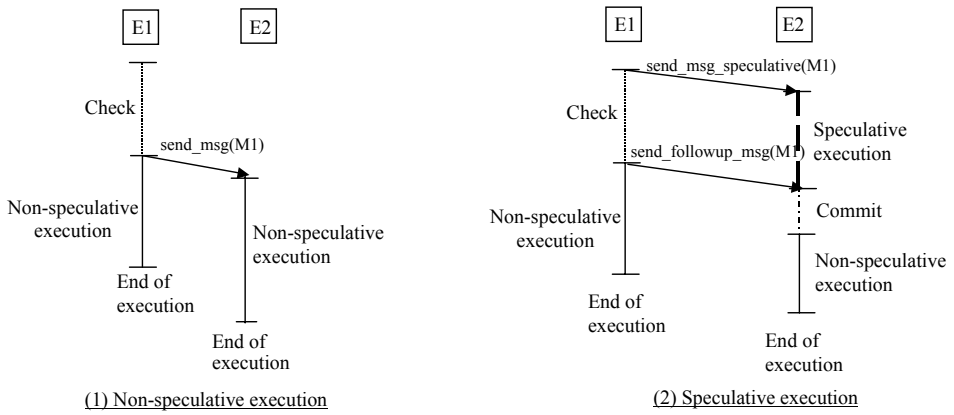


Figure 9. Speculative Execution Technique

This scheme has serious ramifications on the issue of ARMOR recovery because it allows for a certain degree of fault propagation. However, the topic of recovery in ARMORs requires a detailed treatment in itself and is going to be the subject of a forthcoming publication that demonstrates a recovery hierarchy in Chameleon and provides algorithms for recovering from ARMOR (both replicated and unreplicated) failures using checkpointing and message logging techniques.

4 Simulation of Detection Hierarchy

In this section, we describe the simulation study performed to analyze the error detection protocols in Chameleon. The simulation not only allows us to study all the four levels of the detection hierarchy, but also the proposed optimizations. Further, it also permitted the evaluation of the various performance-availability tradeoffs not possible in the implementation. DEPEND, a simulation environment for analysis and modeling of dependable systems [GOS97], was used.

4.1 Motivation

The specific objectives of the simulation study were three-fold:

1. To measure overhead, error latency and coverage while using different combinations of techniques, some of which are not available in the existing implementation (e.g., the remote detection levels 3 & 4).
2. To enable us to fix the system parameters (like, heartbeat interval) to obtain a desired coverage or availability for an allowable overhead.
3. To measure the effectiveness of the proposed optimization techniques – Speculative Execution, and Probabilistic Escalation.

The simulation results show that:

- Availability of 0.9999 (four Nines) can be achieved for an error rate of about once every three days and the measured overhead was 1.2% of the application execution time [Figures 16(a) & (c)].
- The time overhead due to the detection levels was between 1% and 1.5% for the given set of parameters and trigger conditions [Figure 16(c)].
- The optimizations – Speculative Execution and Probabilistic Escalation – lowered the measured overhead for a small loss in coverage or an increase in error latency [Figures 16(g)-(j)].

The simulation model consisted of a set of processors (servers) connected by an Ethernet-like link. Each processor (also referred to as a node or a host) could host the different ARMORs and was a part of the Chameleon system. Nodes also ran background workloads. The underlying model of a node consisted of a server (the processor) executing the ARMOR processes, and other applications. Each node ran a Daemon, both to monitor the ARMOR processes and for inter-ARMOR communication. The processor itself serviced the different jobs in its queue in a time-sharing manner. In this setup, the four error detection levels were modeled.

4.2 Simulation Model

In this model, we consider a typical scenario where an FTM operates in primary-backup mode.

Applications to be run under the Chameleon environment arrive according to an exponential distribution and are assigned randomly to the processors. The application spawns off multiple tasks of varying time durations. While the complexity of the tasks is not modeled in this simulation, in practice, they can be simple standalone tasks, or distributed ones that communicate amongst themselves. With each application task, we associate a Surrogate Manager and an Execution ARMOR. Upon termination of the application, the Surrogate Manager and the Execution ARMOR are uninstalled. The simulation terminates when a designated number of application tasks are completed. Faults are injected into the environment according to an exponential distribution with a specified mean and can invoke detection at any of the four levels of the detection hierarchy.

Error Detection Techniques

In the simulation model, we include the following error detection techniques:

1. *Assertion Checks,* modeled as a fixed percentage of overhead in the execution times of the elements.
2. *Checking by the Monitor Element,* which wakes up periodically and checks for livelocks.
3. *A periodic smart heartbeat generated by the Daemon,* in response to which all elements execute a self-test. On detecting a faulty ARMOR, the Daemon restarts the ARMOR.
4. *Signature checking when a message is being sent out of the node.* This involves signature generation within the ARMOR and signature checking by the Daemon.
5. *Level 3 signature checking protocol* between ARMOR replicas at a particular frequency.
6. *Level 4 protocol* between ARMORs of adjacent ranks.

We also associate a coverage with each error detection technique, that determines whether an error is detected or not by a particular technique. A heuristic value of coverage was assumed for the different levels, with the expected property that the coverage of the higher levels (3 and 4) is higher than those of Level 1 or 2. This is because an error not captured at a lower level has a high probability of being detected at the next higher level. We allow each level to be turned on or off so that we can simulate different combinations of the detection levels. In the simulations, we consider three combinations of detection levels in the system.

- Levels 1, 2, 3 and 4 present in the system (**Configuration A)**
- Only levels 1 and 2 are present (**Configuration B**)
- Only levels 3 and 4 are present (**Configuration C**)

We also simulated the two proposed optimization techniques – *Speculative execution* and *Probabilistic escalation* which were discussed earlier in Section 3.8.

The parameters that were used for the simulation are presented in

Table 4. For most of the parameters, the values are obtained from the measurements from the prototype implementation. Examples are: Execution time of an element, Message generation frequency, Times for techniques in levels 1 and 2. To study the different simulation scenarios we varied the following parameters:

Parameter	Value Range
Number of processors	6
Monitor frequency	1 in 5secs mean
Message generation frequency	1 in 20 secs mean
Application task inter-arrival time	300 secs mean
Execution time of application task	1 – 5 hours
Time for undiagnosed error to show up	10 mins
Execution time of Element	10 – 100 μsec
Number of Elements in ARMOR	3 –10
Threshold for probabilistic execution	0.5
Probability of ARMOR crash	0.01 of level 2 errors
Coverages: Node level, Level 3, Level 4	0.9, 0.95, 0.99
Daemon send time	100 – 200 μsec
Daemon heartbeat frequency	1 in 10 secs globally
Level 3 frequency	45 – 60 secs
Level 3 timeout	5 secs mean
Level 4 timeout	5 secs mean

Table 4. Simulation Parameters

- *The fault injection rate.* This is a global fault rate for all types of faults in the environment. The fault rates used varied from 2/min to 0.001/hour[4].

- *The composition of the faults.* We also varied the composition of the faults injected into the simulated system. The probability that an injected fault was a Level 1 fault (within the ARMOR process), a Level 2 fault (within the node), or a Level 3 and 4 fault (between nodes) is defined by a tuple <p1,p2,p3> where p1 + p2 + p3 = 1. Three sets of tuples were selected to reflect a reasonable variation in the fault probabilities: <0.4,0.3,0.3>, <0.5,0.3,0.2> and <0.6,0.3,0.1>. The default composition for the experiments when other parameters were varied was chosen to be: <0.5,0.3,0.2>. Level 1 or Level 2 faults can also be detected by Levels 3 and 4. However, the FTM is the only replicated ARMOR in the simulated configuration. Therefore, it is the only ARMOR that can execute both the Level 3 and the Level 4 protocols. Any other ARMOR can execute the Level 4 protocol only during ARMOR installation or uninstallation, which is not frequent (once every 1-5 hours). The above two reasons explain why the availability or coverage measure obtained in the configuration with only Levels 3 and 4 is rather poor (Figures 16(a),(d)).

- *Checking_time to Sending_time ratio:* To characterize the effectiveness of speculative message send, we simulated the effect of optimization on overhead and latency, for various ratios of the signature checking time to the message sending time.

- *Confidence distribution:* The confidence distribution function for Probabilistic escalation was chosen to be a negative exponential function [exp(- λ * (current time – last checking time)] and the parameter λ was varied.

4.3 Results

The main output parameters obtained from the simulation are the following. These parameters are obtained with 99% confidence interval for 30 simulation runs.

- *Overhead in invoking the different detection levels.* This parameter measures the additional time required to execute the various detection levels. It is calculated by adding the time taken for each of the steps in the detection techniques over the time taken for the execution of the application without the detection techniques, and is generally expressed as a percentage.

- *Error detection latency.* This parameter gives the average error detection latency per error. The error latency is the difference between the time the error was injected to the time the error was detected by any of the error detection levels, or an arbitrarily large time interval if the error was not detected at all. This arbitrary time interval denotes the case when a manual system reboot or some other manual intervention removes any latent errors.

- *Coverage.* The coverage is the ratio of the number of detected errors to the total number of errors injected into the environment.

- *Availability.* The availability measure is with respect to the Chameleon infrastructure. The availability is calculated from a simple Markov model which can be looked upon as having two states – a correct state and an incorrect state. The transition from the correct to the incorrect state is given by the error rate, which is an input parameter for the experiments. The transition from the incorrect to the correct state is given by the recovery rate and the latency of detection, which is obtained from the simulation experiments. The steady-state availability is the probability of being in the correct state.

The simulation experiments provided us with availability, coverage, overhead, and latency measures, which are presented in Figure 16. The results from the plots are summarized below:

1. Figure 10(a) and Figure 10(b) show the variation of availability with error rate. Figure 16(a) shows that the availability does not degrade substantially with increasing error rate if all the four detection levels are active. As expected, using all four detection levels gives the highest availability, but comparable

[4] The entire range of variation was not present for all the experiments. The range for particular experiments can be seen from the corresponding plot in Figure 16.

measures are obtained by using just levels 1 and 2, for the error composition assumed in these simulations. The errors, which can be captured only by levels 3 and 4, are Byzantine errors which are not expected to occur as frequently as errors detected by levels 1 and 2. Table 5 shows that the availability reaches 0.9999 (four Nines) for fault rates of one every three days or lower (with 5% Byzantine faults). Compared to a real environment, this may still be a somewhat high occurrence of Byzantine faults.

Error Rate (per hour)	Availability (5% Byzantine Faults)	Availability (20% Byzantine Faults)
0.001	0.9999	0.9991
0.0125	0.9999	0.9987
0.025	0.9997	0.9980
0.05	0.9995	0.9860
0.1	0.9989	0.9801
0.5	0.9942	0.9720

Table 5. Variation of Availability with Error Rate

2. Figure 10(c) plots the variation of overhead with invocation frequency of the Level 3 detection technique. For this experiment, the configuration had all the four detection levels present, and the invocation frequency of Level 3 was varied. From Figure 10(a) and (c) it can be observed that a 0.9999 (four Nines) availability can be reached with an overhead of approximately 1.2% (frequency of 45-60 seconds in (c)), where the overhead is expressed as a percentage of the application execution time. Simulations conducted to measure the change of overhead with error rate indicated variation of only between 1% to 1.2% for error rates up to two per minute for Level 3 invocation frequency of 45-60 seconds (not shown in Figure 10).

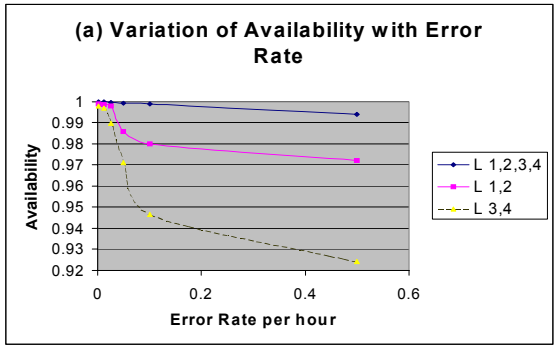
3. Figure 10(d) and Figure 10(e) show the variation of the detection coverage with different combinations of detection levels, fault compositions or error rates. Figure 10(d) plots the variation of the detection coverage with respect to the various compositions of injected faults. The configuration with all four levels present in the system gave the maximum coverage, and the configuration with only Levels 3 and 4 the minimum. This was seen even when the percentage of Byzantine faults was relatively high (30%). This indicates that for most scenarios (where crash faults dominate over Byzantine faults), it is imperative to have levels 1 and 2 of the detection framework. Figure 10(e) plots the variation of coverage with error rate. It is observed that, as expected, for error rates up to two per minute, the coverage does not vary substantially with all four levels.

4. Figure 10(f) shows the variation of error detection latency against error rates. Here, we see that the latencies for errors captured in Levels 1 and 2 are about 10 seconds, but the latencies for errors captured by the higher levels are an order of magnitude higher. The latter is because of the low periodicity of Level 3 protocol (45 – 60 seconds). On the other hand, techniques internal to the node are invoked with a higher frequency. For example, the monitor element checks for livelocks every five seconds and the signature

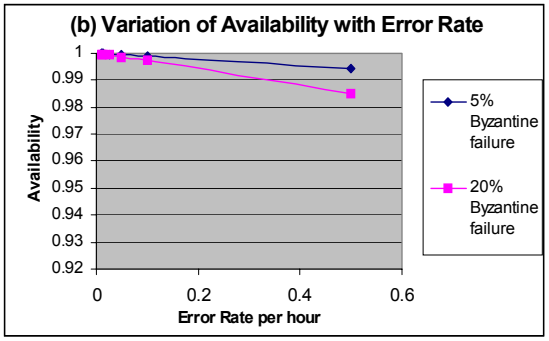
checking can be done at interval of a few seconds, depending on the message I/O activity of the ARMOR. We can also deduce from this graph that the error latency does not increase much with higher error rates.

Speculative Execution (Figure 10(g) and (h)) reduces the overhead, with a marginal increase in error latency (due to delayed checking for errors). Optimization yields a lower overhead when the value of the ratio of the time to do the checking to the time to send the additional follow-up message is more than 5. Our implementation has yielded a value of 12. Therefore, for the current environment, speculative execution appears to be a valuable technique.

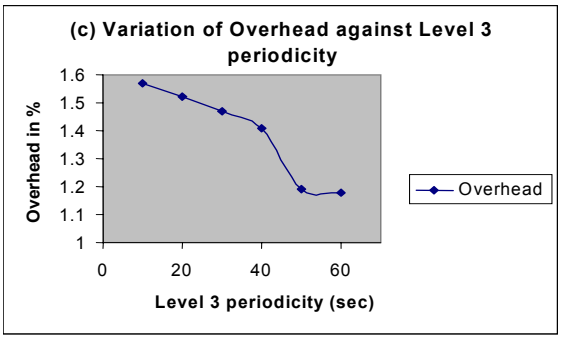
6. Figure 10(i) and Figure 10(j) plot how Probabilistic Escalation influences the error detection latency and the coverage for the environment. The Probabilistic Escalation is used between the Levels 3 and 4. The confidence distribution used for Probabilistic Escalation is an exponential distribution and the parameter λ of that distribution is being varied. A low value of λ implies that the confidence placed on another level's diagnosis is high, and hence the latency is higher and the coverage lower. A spectrum can be seen, one end with low coverage and high latency, and the other end with low latency and high coverage. A higher value of λ will decrease the latency and increase the coverage at the cost of frequent invocation of the higher overhead levels 3 and 4, thereby increasing the overhead of the Chameleon environment as a whole. From the plots, the user should be able to specify, say, a minimum coverage and/or a maximum latency and derive a particular confidence distribution that he can use. A distinct knee can be observed in the curve for a λ value of 0.1, which indicates that for the selected parameter values and error rate, that is a suitable confidence distribution.



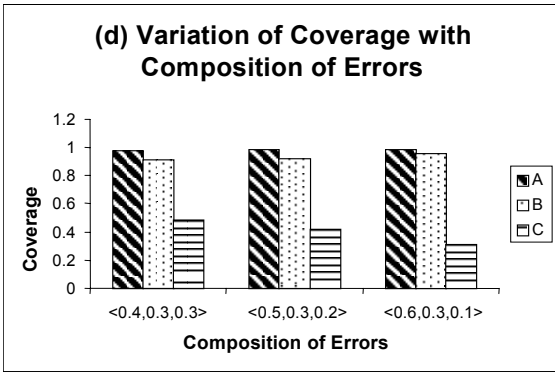
5% Byzantine Faults
Error Composition: (0.50,0.45,0.05)



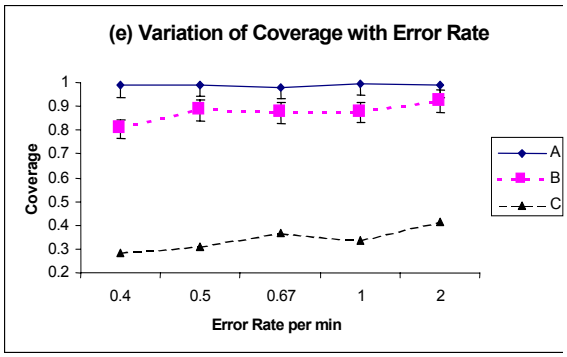
First plot same as plot with L1,2,3,4 in (a)



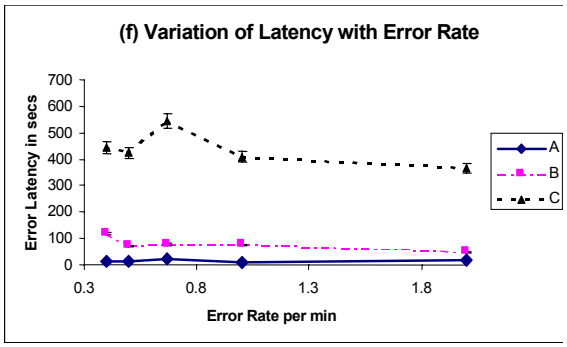
Error rate: 0.1/hour



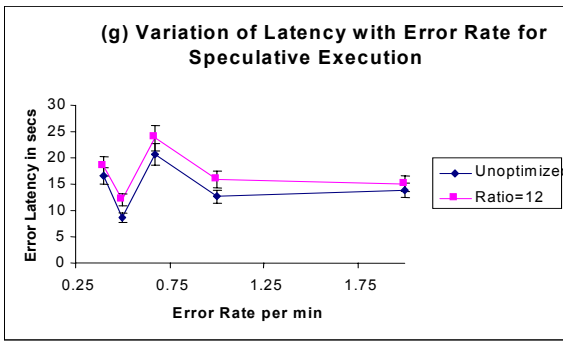
Error rate: 1/min



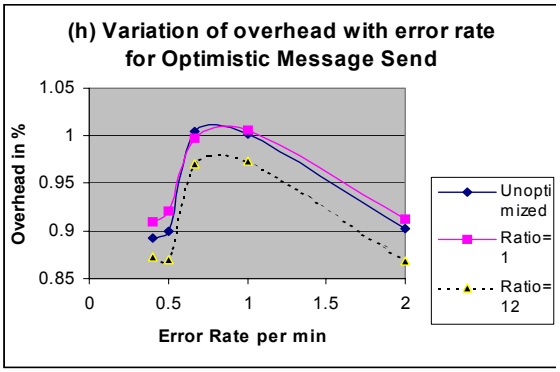
Error composition: (0.5, 0.3, 0.2)



A: L1,2,3,4 B: L1,2 C: L3,4



Ratio = Message Checking time/Message Sending time



Ratio = Message Checking time/Message Sending time

Figure 10. Plots of Output Parameters obtained from Simulation

5 ARMOR Signatures

Signaturing is a technique used at several levels of the proposed detection hierarchy to validate the integrity of ARMORs and the nodes on which the ARMORs execute. The concept of ARMOR signatures and the mechanism of generating them are described in this chapter.

An ARMOR's signature is defined to consist of:

- A control signature and
- A data signature.

There are four types of control signatures:

- Coarse-grained: An I/O-based low-overhead, non-intrusive signature that is formed from the messages that are generated as part of the normal message flow in the system.
- Fine-grained: A higher-overhead, more intrusive signature that contains information about the control flow of the ARMOR.
- PECOS: A pre-emptive control flow checking scheme that embeds checking code in the application's assembly code. It provides the highest level of diagnosability among all the signature schemes. PECOS is described in Chapter 6.
- Text segment: An encoding of the text segment pages of an application that provides protection against memory corruption.

In the remainder of the section, the design, implementation and evaluation of each of the detection techniques is presented. While the control and data signatures are discussed here in the context of a Chameleon ARMOR, it will be clear that the techniques are more generally applicable to non-Chameleon environments as well.

5.1 Coarse-grained I/O Signature

5.1.1 Design

The coarse-grained signature (CGS) of an ARMOR is formed from the valid input-output message types that are handled by its constituent elements. In the Chameleon architecture, during element initialization, every element subscribes with the compound layer, the messages it wishes to receive and also registers the valid output message type(s) that it would generate after processing the input. This information from all the constituent elements forms the golden signature of the compound. The golden signature of the ARMOR is the union of the golden signatures of all its constituent elements and compounds.

The compound layer is responsible for delivering messages to the appropriate elements. Every incoming message from outside the ARMOR causes a new thread to be spawned in Chameleon. All intra-ARMOR messages are handled in a single thread, and this may involve processing by several elements. Suppose a

message of type M1 comes into the ARMOR and is handled by the elements E1, E2, E3, … , En in that order, i.e., E1 takes M1 as its input, processes it, and generates M2 as its output, which serves as the input message for E2, and so on. Generally speaking, element Ei takes message Mi as its input and generates M(i+1) as its output. Finally, the message M(n+1) is sent out of the ARMOR by the element En. In this scenario, the coarse-grained signature generated at runtime is defined as follows:

$$\{ (M1, M2, M3, \dots , M(n+1) \}$$

If multiple threads are currently active in the ARMOR, there will be multiple runtime signatures yet to be validated. For two messages being currently processed, the signature will have the following form:

$$\{ (I_{11}, I_{12}, \dots , I_{1n}, O_1), (I_{21}, I_{22}, \dots , I_{2n}, O_2)$$

This indicates that currently there are two messages that have been processed in the ARMOR but have not yet been checked. Each entry keeps the history of processing that one of the messages has undergone. For example, the first entry corresponds to an input message type I_{11} which resulted in the generation of intermediate messages I_{12}, … , I_{1n} , and finally an inter-ARMOR output message O_1 being generated.

During actual processing, all incoming and outgoing messages pass through the compound layer where the runtime signature is formed. This runtime signature is compared with the golden signature formed earlier, and a mismatch indicates that the element is in error.

Depending upon the overhead that can be tolerated for the CGS, the history of input-output messages may be truncated. In the current implementation, only a one-step history is kept. The signatures corresponding to the correct behavior of the elements are of the form:

$$\{ (I1,O1), (I2,O2), (I2,O3), (I3,\_) \}$$

where O1, O2, O3 are the output messages that were emitted in response to the processing of the messages I1, I2, I2 respectively. If the constituent element that subscribed to message type I1 generates an output message of type O5 because of a control flow error, it will be caught by a mismatch with the golden signature. Note that no (message I3), one (message I1) or more (message I2) output messages may be generated in response to the processing of one input message.

The proposed coarse-grained signature is different from typical validity checks performed by many operating systems, since here the output is validated not only by checking whether it is a valid output message type, but also whether it is an allowable output type for the specific history of input messages. Importantly, the single or multi-step history allows us to localize the error to a specific element. This is attractive in a message-passing based system because, unlike the other signatures, it is not intrusive, in that it does not require any code insertion or additional overhead for signature generation.

The coarse-grained signature is not specific to Chameleon but is generally applicable in most message-passing based system, since most such systems have a message transmission and delivery component and a phase where message recipients subscribe to messages. The requirements are that the checking component

interacts with the delivery and transmission components, and that during subscription of input messages, expected behavior with respect to output messages, be communicated to the checking component.

The coarse-grained control signature will capture control errors that cause any of the elements to generate either an invalid output message type or a valid but incorrect output message type. It will not capture errors in which the element accepts a valid input message and generates a correct output message type but the intermediate processing is wrong (e.g., an intermediate state update is missed by the element). For handling this class of errors, the fine-grained control signature is proposed.

5.1.2 Evaluation

The evaluation of the coarse-grained signature technique was performed using fault injection. Directed message control flow faults were injected to Chameleon messages running scientific workloads. The two chosen workloads for this experiment were:

1. *Fast Fourier Transform* (FFT). This is an algorithm widely used in several scientific applications such as image processing. The implementation is based on the algorithm in [PRE92]. For the experiments, a 4096 element input array was used.
2. *Radix Sort.* This is a linear time integer sorting algorithm. A 10000 element array of integers was used for the experiments.

Both the applications were run under Chameleon as ARMORs, with only the coarse-grained signature detection enabled in level 1. For both FFT and Radix sort, the applications were parallelized into two MPI tasks denoted as Master and Slave. The configuration in which the applications were run is shown in Figure 11.

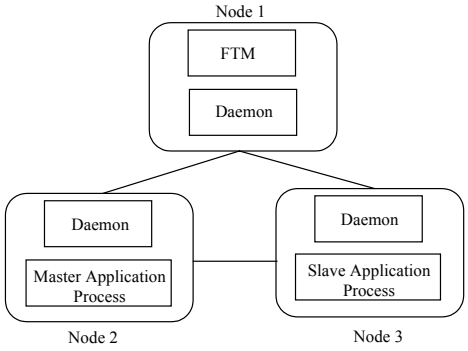


Figure 11. Experimental Configuration for running FFT and Radix Sort in Chameleon

The fault injection was targeted to the control fields of messages generated by the master application process. The injections were performed after the message had been generated by the application element but before the coarse-grained signature element had checked the message. To support this fault model, a fault injection element was written in Chameleon and inserted into the application ARMOR. This type of fault injection cannot be time-triggered exactly, rather a hybrid of message-rate-based and time-based triggering was used. The fault injection element attempted to inject a fault once per second; if no message was present, the fault injector would set a flag to inject the next message(s).

The results from the fault injection campaign are shown in Table 6. A run is classified as "No Error Manifestation" if the application generates the output file and it matches with the golden output file. If the application does not terminate within 30 seconds (a normal execution takes less than 10 seconds) and no detection mechanism flags an error, it is considered a case of application process hang. If the application generates the output file and it does not match with the golden output file, it is considered a case of fail-silence violation. The rationale behind this categorization is that the application generates an output file for use by another module and if the output file is erroneous, then the fault is being propagated to the module that is going to use the output.

The results show that the coarse-grained signature is effective in all but one case of message corruption. An important observation is that in 35-60% of cases, coarse-grained signature detects the error before the error causes a process crash. The single case of non-fail-silent behavior observed with FFT resulted from the corruption of the pointer field of one element of the linked list of message operations. As a result, the FFT application skipped an operation on one element of the input data, and the output result differed from the golden run for one data point.

Result	**FFT**	**Radix Sort**
No Error Manifestation	0	0
Application crash	15 (50.0%)	10 (33.3%)
Coarse-grained signature detection	11 (36.7%)	18 (60.0%)
Application Hang	3 (10.0%)	2 (6.7%)
Fail-silence violation	1 (3.3%)	0 (0.0%)
Total	30	30

Table 6. Results from Directed Message Control Field Injections to FFT and Radix Sort

5.2 Fine-grained Control Signature

5.2.1 Design

The coarse-grained signature can capture errors that manifest themselves in the external interactions of the faulty element. In contrast, the fine-grained control signature is intended to capture control flow errors within an element. There are two main steps involved in the validation process.

- Generation of a golden signature, which specifies the correct execution sequence(s) of execution blocks[5] within an element. This signature is specified through regular expressions registered by the element with the ARMOR level at the time of initialization of the ARMOR. Golden signatures of all constituent elements are stored in a Golden Signature Table (GST) at each ARMOR.
- Generation of a runtime signature, which indicates the control path actually taken by the element. This is generated through special instructions embedded at the end of each execution block in an element. This

[5] An execution block is a group of high-level language instructions executed by an element, e.g., a basic block which has a single entry and exit point.

signature is processed at the ARMOR level by (a) combining it with the signatures generated by other elements (to track control flows across elements), and (b) then comparing with the golden signature. A mismatch indicates that the particular element is in error. The runtime signature of each constituent element is stored in a separate frame[6], and the frames of all the elements of an ARMOR are stored in a Runtime Signature Table (RST). It should be emphasized that, unlike conventional control flow signatures [SCH86], the runtime signature is generated after a particular execution block has executed, not while it is fetched from memory. Another distinguishing characteristic of the fine-grained signature technique from other software control flow detection techniques [ALK99] is that the granularity of signature generation and that of checking need not be the same. For example, one may generate a signature for every block of instructions, but may check the runtime signature only when the ARMOR is going to communicate with some other component. This is a useful configuration for systems where fault containment is desired.

Fine-grained signature generation is illustrated in Figure 12, which shows the control flow graph of a generic element. Each rectangle denotes an execution block with a signature-generation instruction – *Emit* - embedded at the end. The valid control flows for the element are specified by the regular expression A.(B.(C.D|E))*, and the actual control flow that the element follows at runtime is given by the sequence of *Emit* statements executed by the element. In Chameleon, the points of message transmission or reception, and the state update operations are the crucial points for tracking the control flow since these operations can cause fault propagation from a malfunctioning element. Hence, at these points, an execution block is terminated, an Emit is inserted, and fine-grained signature checking is forced.

Currently, the element designer needs to insert these Emit calls in the element code. It is conceivable that this process might be automated, since there are well-defined patterns in the element code where the insertions need to be done.

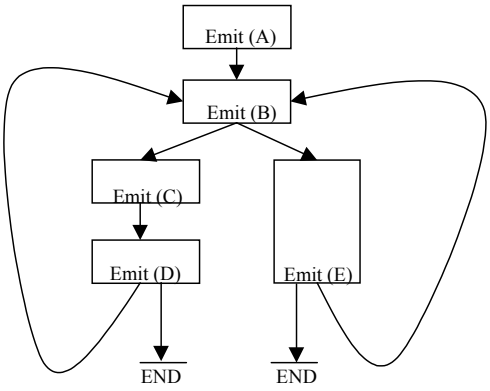


Figure 12. Sample Control-Flow Graph of an Element

[6] A frame is a logical unit that serves as the boundary between signatures of different elements or ARMORs.

5.2.2 Fine-grained Signature Generation and Propagation

Figure 13 shows a configuration with an ARMOR A1, with elements E1 and E2, and the local Daemon D1. In the example, the signature is generated for element E1, which is checked by the Daemon D1.

Initialization.

During initialization, E1 registers the following information with the ARMOR A1:

- The set of valid input-output messages for E1. In this example, it is (I1,O1), (I2,O2).
- The valid fine-grained signatures for E1. In this example, it is given by the regular expression

$$[\ S1\ .\ (\ (\ S2\ .\ S3\ )\ |\ S4\ )\ ]\ *$$

This registration information from all the constituent elements forms the *Registration Information Table* (RIT) at the ARMOR. The RIT is then sent to the local Daemon to form the GST for this particular ARMOR.

Runtime

During the execution of an ARMOR, its runtime signature is constructed incrementally in a data structure called the *Runtime Signature Table (RST)*. The RST consists of (a) the runtime coarse-grained signature which is generated whenever an output message is generated by an element, and (b) the fine-grained signature which is generated by the *Emit* function calls within each element. The RST is communicated to the local Daemon either synchronously (e.g., when the Daemon checks for ARMOR signature with a certain periodicity) or asynchronously (e.g., the case in which the signature is checked on activation of a detection protocol). In the example, the element E1 was activated by an input message I1 and generated an output message O1. It has executed the blocks whose signatures are S1,S2,S3. Observe that at the current instant, the control flow signature (both coarse and fine-grained) of E1 is valid. If the signature of every element in the ARMOR is valid, then the ARMOR will be certified as functioning correctly by the local Daemon.

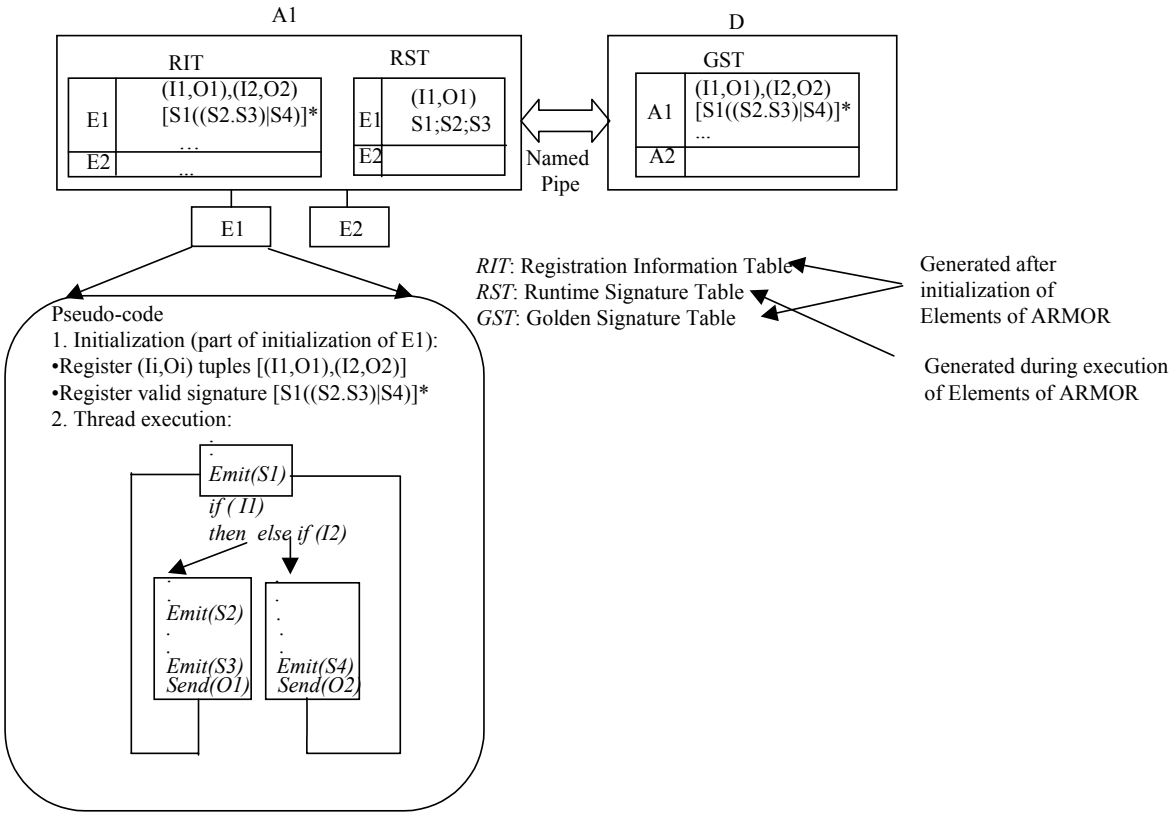


Figure 13. Control Flow Signature Generation and Propagation

Inter-element Dependencies

The fine-grained signatures are designed to track dependencies that span multiple elements or ARMORs. When multiple elements sequentially execute within the same thread, it becomes important to track dependencies across the elements in order to capture any invalid interactions among them. An example of an invalid interaction is an element that is supposed to handle an incoming message by itself but sends an intra-ARMOR message to another element. Similarly, in order for the Daemon to capture invalid interactions among multiple ARMOR processes, dependencies across multiple ARMORs have to be tracked.

At the ARMOR level, each element's signature is maintained in a separate frame in the RST. As shown in Figure 14, when two elements E1 and E2 execute in the same thread and execution passes from one element to the other in the same thread, this is noted in the RST by having a pointer from the runtime signature frame of element E1 to that of E2. The frame of E1 in the GST indicates that it can invoke another element after executing its first two execution blocks and hence this is a valid execution. However, if this was not indicated in the GST, the existence of the pointer will lead to detection of the error type mentioned above as an invalid interaction. This will not capture an error in which execution passes from E1 to another element E3, but it will capture an error in which there was not supposed to be a jump in execution out of E1 (i.e., there is no *X* in the GST).

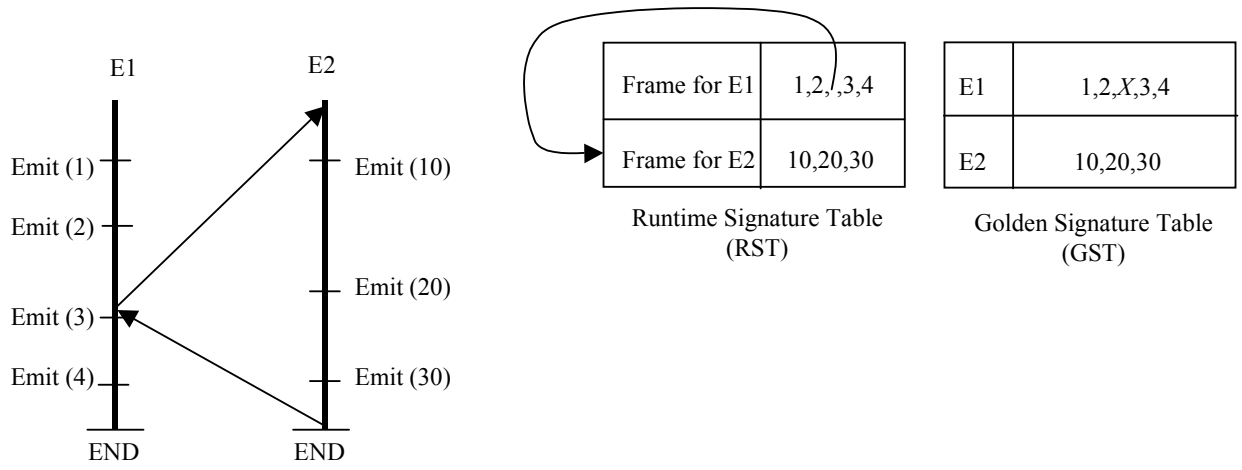


Figure 14. Inter-Element Dependencies in the Fine-grained Signature

5.2.3 Evaluation

Evaluation of the fine-grained signature detection scheme is done by fault injection into the Radix Sort workload running under Chameleon as a Chameleon ARMOR. Directed control flow faults are injected into the application through the software fault injection tool called NFTAPE [STO00]. The faults are injected into the operands of the assembly level control flow instructions of the application. These faults are considered likely to cause control flow errors which is the fault model that the fine-grained signature was designed to protect against.

The application is run both with fine-grained signature enabled and disabled. In each campaign, 400 runs were performed, with one fault being injected per run. Radix sort is run in a configuration identical to the one used for the experiments for the coarse-grained signature evaluation (Figure 11). The results of the evaluation are presented in Table 7. The interpretation of the categories – No Error Manifestation, Application Hang, and Fail-silence Violation – are as in the evaluation of coarse-grained signature. Chameleon has some semantic reasonableness checks built into its code as part of the standard error checking. For example, the return code from a Chameleon function in the correct case is 0 and when the caller finds a non-zero return code, it flags an error. This can also be looked upon as a form of detection. Following the terminology of the four-level detection hierarchy (Figure 1), this form of detection is called Level 0 detection. Similar to this, the application has some semantic checks built into it and when a check detects an error the application is aborted. Such cases constitute "Application Abort". The cases where the error is not manifested can be because the faulty instruction was not executed at all, or the faulty instruction was executed but still had no effect. From the experimental setup, it is not possible to separate these two cases. The first case is dependent on factors like how much dead code (i.e., code not activated in the normal scenarios) there is in the application program, and how selectively within the application, the target of the injection is chosen. Therefore, the runs in this category were removed for calculating the percentages for the remaining categories.

Result	Baseline (Without Fine-grained Signature)	With Fine-grained Signature
No Error Manifestation	172	121
Application Crash	167 (73.2%)	227 (81.4%)
Fine-grained signature detection	---	20 (7.2%)
Chameleon Level 0 detection	21 (9.2%)	9 (3.2%)
Application Hang	30 (13.2%)	10 (3.6%)
Application Abort	3 (1.3%)	3 (1.0%)
Fail-silence Violation	7 (3.1%)	10 (3.6%)
Total	400	400

Table 7. Results from Directed Control Flow Fault Injection to Radix Sort

The results show that for a data intensive program like Radix Sort, fine-grained signature is not effective in reducing fail-silence violations or process crash. Its value lies in reducing the incidence of application hang. Process hang typically has a large detection latency (through heartbeat timeouts) and therefore adversely impacts the system availability. Therefore, the fine-grained signature detection technique should increase system availability. It is observed that in the application instrumented with fine-grained signatures, for 3 cases, there is a fail-silence violation even though there is signature detection (1 case) or system detection , i.e., process crash (2 cases). This indicates that a detection technique should have a low latency to eliminate fail-silence violations. The trigger for checking the fine-grained signatures is tunable in Chameleon. For the current experiments, the trigger was set as an inter-ARMOR message send. This causes a higher detection latency than the case when every message send (inter-ARMOR or intra-ARMOR) is a trigger. It will be useful to conduct further experiments with other triggers to evaluate the coverage provided by the technique. The number of cases of Chameleon level 0 detection came down for the instrumented application. This implies that the fine-grained signature is picking up some error cases which would ultimately have been detected by Chameleon's semantic checks. Therefore, if an application is being run in an environment where such semantic checks are not available, fine-grained signatures will be valuable. Also, it has been shown that application-specific semantic checks typically have a large detection latency [KAN96]. Hence, pre-empting such detection should yield gains in availability. The results show that the proportion of system detection, i.e., application crashes is not reduced by the fine-grained signature technique. Since process crashes are undesirable in many applications (e.g., real-time control applications), this serves as a powerful motivation to explore other control flow error detection techniques. The work on PECOS resulted in part from this motivation.

5.3 Text Segment Signature

5.3.1 Motivation

A problem with the earlier signatures is that the trigger for checking them is a message send. Therefore, if the application is following an illegal control flow, then the problem will not be detected till the next message send by an element in the concerned ARMOR. Our initial fault-injection experiments into the text segment demonstrate that in 40% of the cases, the application crashes because of the illegal control flow before a trigger for the checking of the two above signatures is reached. This happens in the case of a branch to a location in the data segment, or a branch to a text segment location in another stack frame. Control flow errors can be caused by faults in:

- the memory – e.g., corruption of the target of a branch or a jump, the opcode for the branch or the jump, and the label of a branch or a jump.
- or, the processor – e.g., failures in the instruction register, program counter, address register.

To protect against the memory errors, it is necessary to protect the memory of the text segment of the program with some code.

5.3.2 Design

We examine three coding algorithms from the points of view of their performance degradation and the coverage offered. The coding algorithm is executed first at the beginning of execution of the program when the text segment is loaded into memory. A page of the text segment is considered as the unit for this signature. The initial computation forms the golden signature table with the number of entries being the number of pages in the text segment. At runtime, during execution of the application (the ARMOR being the application we are trying to protect here), the coding algorithm is re-executed and the code generated is compared against the golden signature table. The trigger for the re-computation is provided by a timer. For performance reasons, at the trigger point, the code may be recomputed only for the current page of the text segment and not for the entire segment. Memory errors of the first two types (target and opcode corruption) will be detected by a mismatch of the golden and runtime signatures. The third type of memory error leading to control flow error (label corruption) can also be detected if the target of the branch or jump lies in the same page. This is likely as branch targets in about 90% of the branches in Spec benchmark programs were found to be less than seven instructions away [HEN96]. Of course, the detection is conditional on the probability of aliasing (a corrupted page yielding the same code as a correct one) due to the coding algorithm.

The three codes examined are:

1. *Checksum.* A page of the text segment is partitioned as 16-bit words and a 16-bit single precision checksum with carry wrap-around is computed on it.

2. *Checksum with sequencing information.* The 16-bit words on which the above checksum was done are augmented with information about the sequence of the instructions. The checksum is then computed on these augmented words. The scheme is shown in Figure 15. The figure shows sequencing information that has a look-ahead of one instruction, but in the general scheme the sequencing information can have a higher look-ahead. The function chosen for the current study is even parity. This scheme detects two additional classes of errors than the simple checksumming technique: (a) a change in the sequencing of the instructions (i1,i3,i2 instead of the correct sequence of i1,i2,i3); (b) even number of errors in the same column position in opposite directions.

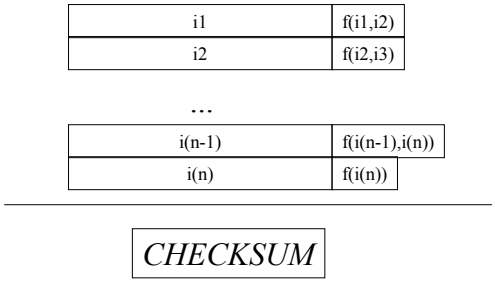


Figure 15. A coding scheme augmenting simple checksumming with sequencing information

3. *Counting runs of Ones-and-Zeroes*. In this scheme, unbroken runs of zeroes or ones are represented as a single field. For example, a pattern in the text segment:

   11100001 11000011 is coded as 3 1 ! 4 0 ! 3 1 ! 4 0 ! 2 1

   This is typically used as a data compression technique, but the compressed data is used here as the signature of the text segment. The memory overhead of this technique is dependent on the lengths of unbroken runs in the text segment, increasing as the run lengths decrease. Our analysis of a sample text segment in the Sparc architecture has shown that between 10% and 20% of runs of zeroes have lengths of 10 or more.

5.3.3 Evaluation

Comparison of 3 Coding Schemes

Comparative evaluation of the three coding schemes which are used to create the text segment signatures:

1. Single-precision checksum (C1)
2. Checksum augmented with sequencing information (C2)
3. Counting runs of zeroes and ones (C3)

The overhead in time to compute each of the signatures on a 4KB page of the text segment is given in Table 8. C2 costs approximately 10 times C1 and C3 costs 3 times C2. The additional cost in C2 is parity computation, which in software is computed using shifting and possibly inverting the running parity bit. This could be done using a parity generation circuit in hardware, if available. This will bring down the cost of C2 relative to C1 substantially.

Single-precision checksum (μsec)	Checksum with sequencing (μsec)	Counting 0/1 runs (μsec)
(290.244812, 302.778442)	(3127.270996, 3454.456055)	(7505.892578, 7766.016602)

Table 8. Time overhead to compute the code on one 4KB page of the text segment.
(95% confidence intervals are given for 40 readings)

Fault injection experiments are run with injection being made to the text segment of a computationally intensive matrix multiplication application. Faults injected are random bit flips at random byte offsets within the text segment of the application. The coding algorithm is run once at the beginning and then at runtime it is recomputed and checked with a default periodicity of 2 seconds. Note that this is different from the proposed scheme of triggering off the checking by a branch-out statement. The effects of the fault injection in the base case, i.e., the application without any of the coding techniques are shown in Table 9.

Effect	**Frequency**
Crash	40%
Incorrect Result	40%
Hang	10%
Correct Result	10%

Table 9. Effects of fault injection into the text segment of a computationally intensive workload application

The coverage of detection by each of the coding schemes for each of the above error cases is shown in Table 10. It is seen that C2 gives additional coverage over C1 for the crash and the hang case, but performs identically for the incorrect result case. Hence, if the only concern is to guarantee fail silence, then C1 is quite sufficient. C3 never performs any better than C2 and hence the additional overhead of C3 is quite unnecessary.

Error Case	Coverage of C1	Coverage of C2	Coverage of C3
Crash	8.33%	25.00%	25.00%
Incorrect Result	83.33%	83.33%	83.33%
Hang	0.00%	100.00%	100.00%

Table 10. Coverage of error detection for the three coding schemes for different classes of errors

Fail-Silence Coverage Evaluation

A set of fault injection campaigns directed to the FFT application is performed. Random single bit flips were injected into the text segment, which was protected through the text segment signature following the single precision checksum algorithm. The results of these injections are shown in Table 11. One set of campaigns is performed to the baseline FFT application (i.e., FFT without text segment signature), and a second set of campaigns is performed to the signatured FFT application. All the elements in the Chameleon element repository (about 40) were statically linked to each ARMOR, though the ARMOR may use just a few elements from the repository (e.g., FFT only uses three elements including the core FFT element). In the experiments, Rand. refers to injections made anywhere in the text segment, to any of the elements. Elem. Means injections

were performed to the text segments of the three elements that were used. FFT means injections were performed only to the core FFT element.

Result	**Without Text Signature**				**With Text Signature**			
	Rand.	Elem.	FFT	Total	Rand.	Elem.	FFT	Total
No Error Manifestation	16	5	7	28	0	0	0	0
Application Crash	13	21	20	54	6	9	7	22
Text segment signature detection	---	---	---	---	24	20	22	66
Application Hang	0	0	0	0	0	1	1	2
Fail-silence violation	1	4	3	8	0	0	0	0
Total	30	30	30	90	30	30	30	90

Table 11. Results from Injection to Text Segment of FFT

When randomly injecting faults over the entire text segment, the text segment signature check always detected the error unless the fault caused the program to abort first. Because there was such a large amount of dead code compiled into the text segment, targeting injections to code that was more likely to execute provides a better idea of how effective the signature check is. To test this, faults were selected from the text segment's pages that contained elements being used and into those of the application element. The results from these injections, shown in Table 11 (columns Elem. and FFT), demonstrate that the text segment signature was able to capture most of the faults and again not only prevent fail-silence violations but also capture the error before a process crash. It can be seen that text segment signature resulted in a substantial number of false positives, i.e., it signaled an error when it detected a bit flip in the text segment, though that fault may not otherwise have caused any error. As a result, there were no runs that ran to completion with the text segment signature turned on. To reduce the number of false positives, a modification of the text signature technique is required so that it flags an error only when the fault is in a page of the text segment that is being used or that is going to be used shortly.

6 Pre-Emptive Control Flow Checking: PECOS

6.1 Motivation

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 chapter, we focus on control faults as they can lead to data errors, process crashes, or fail-silence violations[7]. Control flow errors have been demonstrated to account for between 33% [OHL92] and 77% [SCH87] of all errors. In this paper it is shown that about a third of the activated faults in the application code of a non data-intensive application lead to control flow errors. In distributed applications, fail-silence violations can have potentially catastrophic effects by causing fault propagation.

In this chapter, we propose a control-flow signature generation and pre-emptive checking technique, called *PECOS* (**Pre**emptive **Co**ntrol **S**ignatures). The PECOS target error model is any corruption that causes the application to take an incorrect control flow path. This may result from corruptions to the control flow instructions or from any other corruption (such as a register error) that subsequently affects the control flow. The proposed technique is shown to handle both static and dynamic control flow constructs.

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

1. Because PECOS is preemptive, it significantly reduces the incidence of process crashes, thus enabling graceful termination of an offending process or thread.
2. PECOS minimizes error detection latency because it is triggered before an error manifests as a failure. Minimizing the error latency is important in reducing error propagation and the chance of other undesirable

[7] A fail-silent application process either works correctly, or stops functioning (i.e., becomes silent) if an internal failure occurs [BRA96].

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

One of the workloads on which PECOS was evaluated was 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.
- 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 second workload on which the evaluation was done was a wireless call processing (WCP) application running in a digital mobile network controller. The environment includes a database subsystem containing configuration parameters and resource usage status, which is used by the call processing application for managing active calls. The results from its evaluation show that:

- for control flow errors in the call processing clients (a) in the absence of any detection in the client, 14.0% of injected faults propagate to the database, (b) there are no fail-silence violations and no client hangs, and (c) the incidence of client process crash are reduced from 48.0% to 18.5%.
- PECOS eliminates the incidence of hangs and fail-silence violations of the call processing client.
- The incidence of process crash is brought down from 48.0% to 18.5% for control flow errors.
- In the absence of any detection in the client, 14.0% of the injected faults propagate to the database.

6.2 Related Work

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

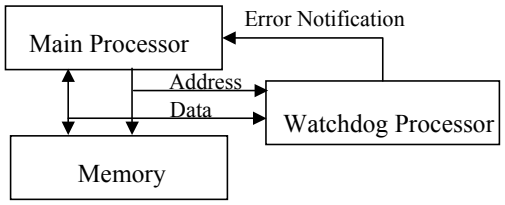


Figure 16. Typical organization of watchdog processor, main processor and memory for hardware-based control flow error detection schemes [From [MAH88]]

6.2.1 Hardware Watchdog Schemes

Mahmood [MAH88] presents a survey of the various techniques in hardware for detecting control flow errors. Many schemes for checking control flow in hardware have been proposed [NAM82, SCH86, WIL90, MAD91, MIC91, UPA94, MIR95]. A summary of some of these techniques is presented in Table 12. The basic scheme is to divide the application program into blocks. Each block has a single entry and a single exit point. A golden (or reference) signature is associated with the block that represents an encoding of the correct execution sequence of instructions in the block. Examples of signatures are the XOR function, the output from a Linear Feedback Shift Register (LFSR), and the output from a cyclic code generator. The selected signature is then calculated by a watchdog processor at runtime. The watchdog validates the application program at the end of a block by comparing the runtime and the golden signatures of the block. Figure 16 depicts the typical arrangement of the watchdog processor with respect to the main processor and memory. The hardware watchdog-based techniques have been evaluated on small applications, and on fairly simple microprocessors, such as the Z80 [MAD91]. Among the hardware schemes, two broad classes of techniques have been defined that access the pre-computed signature in two different ways. Embedded Signature Monitoring (ESM) embeds the signature in the application program itself [WIL90, SCH86], while Autonomous Signature Monitoring (ASM) stores the signature in memory dedicated to the watchdog processor [MIC91]. Upadhyaya *et al.* [UPA94] explore an approach in which no reference signatures need be stored, and the runtime signature is any *m-out-of-n* code word.

Applying hardware schemes to distributed environments suffers from several limitations:

1. Hardware schemes are quite suitable for embedded small processors running single programs, but they do not scale well to complex modern processors. If multiple processes (or threads) execute on the main processor, then the memory access pattern observed by the external watchdog will not correspond to the signature of any single process, and hence an error will be flagged by mistake. This is due to the different possible interleaving among the multiple processes being executed.

2. The underlying assumption is that, in presence of errors, runtime memory accesses observed by the watchdog will differ from the reference signature. Consequently, an error after an instruction has been fetched from memory, e.g., while it resides in the cache of the main processor, will not be caught even if it causes application misbehavior. For current processors with increasingly large cache sizes, such errors are no longer negligible and a watchdog would have to be embedded in the processor itself, which in turn presents new challenges.
3. Hardware watchdog schemes require transmission on the system bus to communicate information to the external watchdog (e.g., the OSLC signature generator communicates the runtime signature to the checker [MAD91]). Transmission errors on the bus (address or data) during those transmission cycles would potentially reduce the coverage of the signature techniques.

6.2.2 Software Schemes

The software techniques partition the application into blocks, either in the assembly language or in the high-level language, and insert appropriate instrumentation at the beginning and/or end of the blocks. The checking code is inserted in the instruction stream, eliminating the need for a hardware watchdog processor. Representative software-based control flow monitoring schemes are Block Signature Self Checking (BSSC) [MIR92], Control Checking with Assertions (CCA) [KAN96], and Enhanced Control Checking with Assertions (ECCA) [ALK99]. A survey of some of these techniques is presented in Table 12.

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

- To our knowledge, none of the existing software-based techniques is preemptive in nature, i.e., the checking involves executing code from the target address of the control flow instruction[8]. If there is a control flow error, executing instructions from an invalid target location is likely to lead to system detection, causing a process crash. Consequently, for a majority of cases (e.g., see the experimental results in Section 5) system detection is triggered before the specific control signature checking technique, and this results in process crash.
- None of the techniques we are aware of has the capability of handling situations where the control flow behavior is determined by runtime parameters. This is because the reference signatures used, cannot be determined at runtime. As a result, a large class of control flow instructions are left uncovered, including register-indirect jumps, calls to dynamic libraries, and function calls using virtual function tables (as in object-oriented languages like C++). Application programs increasingly make use of dynamic library calls, follow object-oriented class hierarchies with inheritance, and have register-indirect jumps for high-level

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

constructs like the *case* statement, all of which result in control flow structures that are determined at runtime.

- The evaluations of existing signature techniques often do not bring out how sensitive the system is to errors in the checking code itself. It is often not clear from the experimental results if the coverage values are obtained with or without the signature instructions being injected. If the checking code introduces control flow instructions itself (as in BSSC [MIR92] and CCA [KAN96]), then the effectiveness of such a technique depends on protecting the additional instructions.
- How generally applicable is the technique to off-the-shelf applications running on off-the-shelf processors? With the exception of the ECCA technique, which has been evaluated on two significant integer SPEC92 benchmark programs. To date, the evaluation of the existing software control flow techniques have been done on fairly simple applications. For example, BSSC uses programs containing linked list operations, quicksort and matrix transpose, CCA uses matrix multiply and quicksort. Fail-silence violations become critical in distributed environments because of the possibility of error propagation. Therefore, to obtain a thorough understanding of the effectiveness of a technique, it is important to run substantial distributed and client-server type workloads, in addition to the simple mathematical ones.

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

Table 12 presents a survey of representative hardware and software techniques for control flow error detection. The first five techniques presented require additional hardware, while the last two are purely software-based approaches. The table mentions the results of evaluations of the available techniques. For the results, an attempt has been made to remove the cases in which the detection is done by something other than the technique under consideration, e.g., system detection which crashes the application process, is removed from the coverage number. For the hardware and software resources required, a *HI* next to the entry denotes that the complexity of the particular resource is deemed to be quite high.

Technique	Outline of Scheme	Workload; Performance Metrics (Detection coverage, Memory and Execution Time Overhead); Resources required	Comments[9]
Signatured Instruction Stream (SIS) [SCH86]	Operates on assembly language code. The signature is embedded in the application instruction stream. 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%(1) Mem overhead(2): 6-15%(3) Perf overhead: 6-14% Hardware: Watchdog processor with cyclic code signature generator & branch hashing capability Software: Modified assembler and loader	Needs parser to embed signatures. Needs runtime support to rehash branch addresses and synchronize it such that the rehashing happens before the target address calculation. Main contribution of high overall system coverage comes from system detection.
Path Signature Analysis (PSA) [NAM82]	Signatures are computed for paths (sequences of nodes) rather than a single node (a node is a branch free interval). 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. To ensure all paths in a path set have same signature, justifying signatures have to be inserted in some nodes. Combination of reference and justifying signatures optimizes total number of signatures.	50 unspecified programs of different types from data manipulation to I/O control routines on MC-68000 Coverage: Not Available Mem overhead: 12-35% Perf overhead: Not Available Hardware: Separate tag memory, Watchdog processor with branch detecting circuit, comparator and control unit for generating control signals to the main processor Software: Modified assembler to identify and match corresponding branch entry and exit points *(HI)*(4)	Requires complicated parser for finding path sets. Error detection latency can be significant since detection is done at the end of the path. Control flow errors that cause erroneous sequence of paths to be executed are not detected.
On-line Signature Learning and Checking (OSLC) [MAD91]	Application is decomposed into several sections, each having a certain number (typically 64 or 128) of branch-free intervals (BFI). A signature generator and a watchdog processor are required. The signature generator signals beginning and end of a BFI to the checker and generates the runtime signature of the block. 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%(5) Mem overhead: Not Avail. Perf overhead: Not Avail. Hardware: A Signature Generator with Parallel Linear Feedback Shift Register (PLFSR) tied to main processor *(HI)*, a simple two-stage pipeline Checker Software: Exhaustive tester *(HI)*	Addresses accessed by the application processor must be visible on the external bus. Mostly control bit errors are detected, few control flow errors are detected(6). Prior to running, exhaustive path activation to generate golden signatures for each block in a segment is required. Synchronization required between the application processor, signature generator, and checker. (See explanation below)

Table 12. Survey of Representative Control Flow Error Detection Techniques
The first five techniques are hardware-based, while the last two are software-based.

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

Time-Time-Address Signature Checking [MIR95]	Program is decomposed into Branch Free Blocks [BFB] (protected branch free intervals) and Partition Blocks [PB] (branch instruction and unprotected branch free intervals). Special instructions are embedded at the beginning and end of each BFB to signal to an external watchdog processor. The watchdog starts a timer on getting notification of beginning of BFB or PB. If notification of end of BFB or PB is not received before timeout expires, an error is detected. Notification of address and block size are made to the watchdog at the beginning of a BFB. At block exit, watchdog checks that exit is made at (start address + size).	Linked list manipulation, Matrix manipulation, Quicksort on MC6809E (8-bit CPU) Coverage: 23.9%[7] (heavy ion radiation method) 48.6% (power system disturbance method) Mem overhead: 35-39% Perf overhead: 25-30% Hardware: Timer *(HI)*, Watchdog processor Software: Software to decompose application into BFB & PB, 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 (like modulo-2 sum) is applied to the instruction stream at compilation time. When the accumulated signature forms an m-out-of-n code or a branch point is reached, the instruction is tagged. At runtime, a watchdog verifies that at a tagged instruction, an m-out-of-n code has been accumulated. No reference signatures are required leading to memory overhead reduction.	Possibly no implementation exists Coverage, mem overhead, perf overhead: Not Available Hardware: A signature generator, an m-out-of-n checker, branch detector 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. Tagging memory may require allocating an extra memory word. Better at detecting control bit errors, than control flow errors.
Block Signature Self Checking (BSSC) [MIR92]	The program is divided into blocks and each block is assigned a signature, as in the standard approach. The signature is the address of the first instruction in the basic block. A subroutine call inserted at the beginning of the block stores the block signature in a static variable. 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 (8 bit CPU) Coverage: 15-22%[8] Mem overhead: 23-35% Perf overhead: 88-127% Hardware: None Software: Software to identify blocks, assign calls and signatures	The assertions introduce control flow instructions of their own, which are additional sources of vulnerability. The technique assumes absolute addresses for the start of the basic block. This will preclude using it for relocatable code. May not detect control flow error that causes blocks to be executed in incorrect sequence.

Table 12 (cont). Survey of Representative Control Flow Error Detection Techniques
The first five techniques are hardware-based, while the last two are software-based.

Enhanced Control-flow Checking with Assertions (ECCA) [ALK99]	Branch-free intervals in a high-level language (C for their implementation) are identified. The entry and the exit points of the intervals are fortified through assertions inserted in the instruction stream. Before running, the program is analyzed, and two variables—Branch Free Interval Identifier (BID) and Next —are assigned to each interval. In case of a control flow error, the BID computation will cause a divide-by-zero error leading to detection.	2 SPEC Int92 benchmarks Coverage: 18.4%-37.5% (022.li) 27.8%-73.0% (espresso) (depends on error model) Mem overhead: Not avail. Perf overhead: Not avail. Hardware: None Software: A C-language lexer, filter and parser, signature generator *(HI)*	A parser for high-level code can be complex because of larger variations in code structure at the high-level language. For large programs overflow of NEXT variable value may occur necessitating a reset to an initial value, which decreases coverage because of aliasing.

Table 12 (cont). Survey of Representative Control Flow Error Detection Techniques
The first five techniques are hardware-based, while the last two are software-based.

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

Table 13. Explanation of Terms in Table 12

6.3 PECOS: Principle, Design and Fault Model

6.3.1 PECOS Pre-emptive Checking Principle

PECOS monitors the runtime control path taken by an application and compares this with the set of expected control paths to validate the application behavior. The scheme can handle situations in which the control paths are either statically or dynamically (i.e., at runtime) determined. Figure 17 depicts the basic PECOS instrumentation and the resulting change to the application code structure. The application is decomposed into blocks, and a group of PECOS instructions called Assertion Blocks are embedded in the instruction stream of the application to be monitored. Normally, each block is a basic block in the traditional compiler sense of a branch-free interval (i.e., the decomposition of the code is performed at the assembly-code level), and each basic block is terminated by a Control Flow Instruction (CFI), which is used as a "trigger" for PECOS. As shown in

Figure 17(b), PECOS Assertion Block (AB) is inserted at trigger points. The assertion block contains: (1) the set of valid target addresses (or address offsets) the application may jump to, which are determined either at compile time or at runtime, and (2) code to determine the runtime target address. The determination of the runtime target address and its comparison against the valid addresses is done *before* the jump to the target address is made. In case of an error, the Assertion Block raises a *divide-by-zero* exception, which is handled by the PECOS signal handler. The signal handler checks, whether the problem was caused by a control flow error[10], and if so take a recovery action, e.g., terminate the malfunctioning thread of execution.

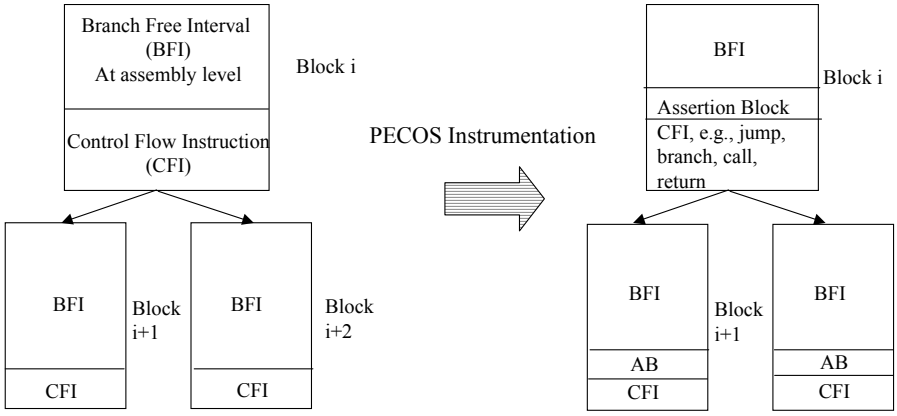


Figure 17. Change in the Code Structure Due to Inserting PECOS Assertion Blocks
CFI = Control Flow Instruction; AB = Assertion Block.

6.4 Why Preemptive Detection Is Important

Figure 18 summarizes the problem with non-preemptive schemes and the solution proposed by PECOS. Figure 18(a) shows a non-preemptive scheme, in which the validity of the control path is checked at the end of execution of a block. All existing control flow error detection schemes that we are aware of are non-preemptive in nature and cannot be easily made preemptive as discussed in Section 3.2. There are two fundamental reasons why a preemptive approach is preferable. Both reasons are related to the ease of recovery following detection.

1. Preventing process crash. Experiments with random error injection into the text segment of an application process have shown that with a non-preemptive scheme a significant number of cases lead to process crash before detection by the technique. For example, our experiments using a SPECInt benchmark application instrumented with Enhanced Control Checking with Assertions (ECCA) [ALK99] showed an average of 32.7% of random errors and 57.0% of directed control flow errors resulted in system detections (i.e., process crash). The detailed results are shown in Section 7.9. A crash of the entire application process incurs a higher recovery overhead due to the overhead of process creation (the kernel allocating a new entry in the 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

[10] 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.

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

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

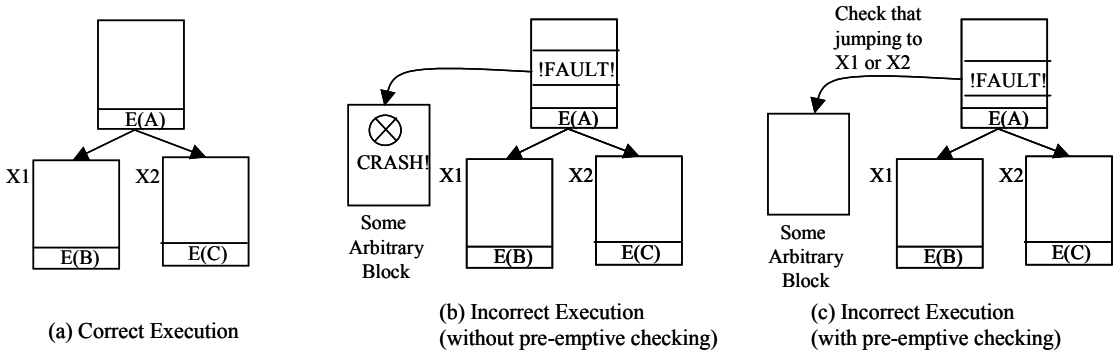


Figure 18. Reason for Preemptive Control Flow Checking
E(X) = Signature block inserted at end of each block and point where checking is done.
X1, X2 = Target addresses.

Can existing techniques be made preemptive without substantial effort?

Our investigation suggests that while, in principle, existing techniques could perhaps all be made preemptive, there will be several theoretical and practical hurdles, which could fundamentally change the nature of the technique in question. For example, for a typical hardware scheme presented in [NAM83], the watchdog processor would have to stall the main processor on detecting that the application process has reached the end of a block, fetch the control flow instruction, perform the target address calculation, and verify that the target address points to one of the allowed destination blocks. Consequently, simply allocating a node id to each node will not suffice, but a table-mapping of node ids to the address range of each node will also have to be

maintained. The watchdog design thus becomes substantially more complicated in order to handle multiple processes running on the application processor. For a software control-flow detection scheme like ECCA [ALK99], making the scheme preemptive would involve changing the Test Assertion at the end of the block so that it will:

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

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

The control flow signaturing technique by Upadhyaya *et al.* [UPA94] is possibly the most conducive to being made preemptive. The checking is performed at the beginning of the block, at the target location of a control flow instruction. However, control is still transferred to the target of the control flow instruction, and consequently, an incorrect target calculation can still cause the process to crash. Moreover, a fine level of synchronization is involved between the checking hardware and the application processor. The checking circuitry should disable the application processor as soon as the justifying signature at the beginning of the target block is found. It should not let the application processor continue executing instructions from the target block till the check has been performed. Also, it is difficult to quantify the effectiveness of the technique, since no experimental implementation or evaluation is available. Summarizing, while the technique may be a good candidate for preemption, significant new effort is required before this can be achieved.

6.5 PECOS Instrumentation

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

To illustrate the need for phase 2, consider the example in which the application consists of two source files (*foo.c*, *foo1.c*). The corresponding assembly files are *foo.s* and *foo1.s*, with a call being made from *foo.s* to *func1* which is defined in *foo1.s*. The offset of *func1* within *foo1.s* is 0x500, so the Assertion Block that precedes the call to *func1* has 0x500 in it as the valid address. When the executable is generated after linking *foo.o* and

foo1.o, only then are the relative placements of the individual object files in the executable known. The file *foo1.s* is placed at an offset 0x12000, so that the offset of *func1* becomes 0x12500. The Assertion Block preceding the call to *func1* will need to be changed from 0x500 to 0x12500, which is done by phase 2 of the tool.

As Figure 19 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 19 is to be followed. Otherwise the path shown with dotted lines is followed, where the executable is disassembled to generate the assembly code and then the PECOS instrumentation tool is applied to the generated assembly code[11].

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

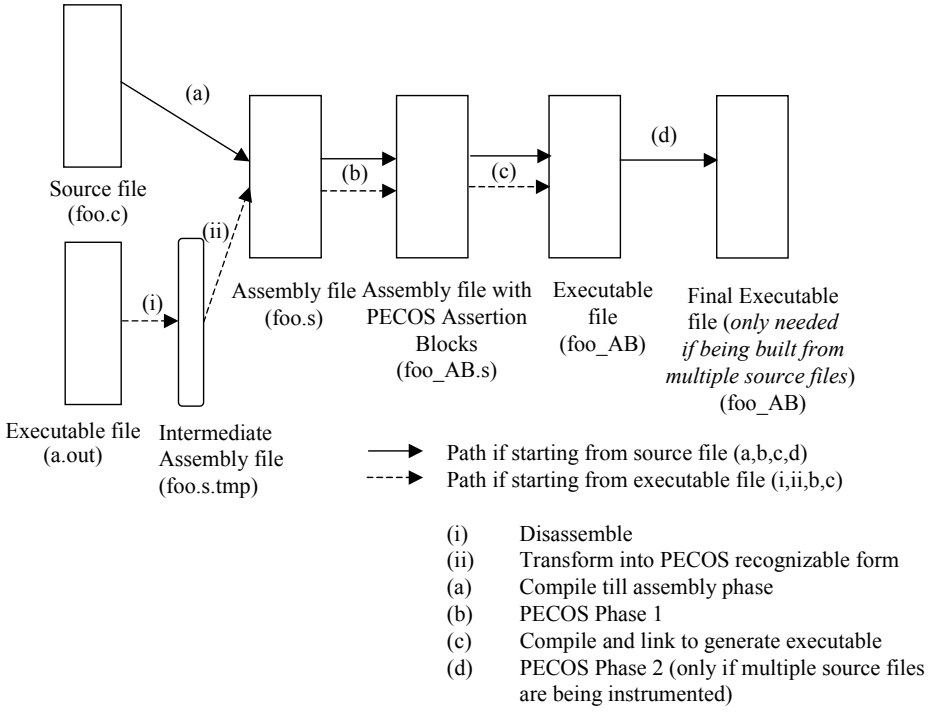


Figure 19. Process of Instrumenting an Application with PECOS

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),

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

the parsing is substantially less complicated. A tool such as ECCA, which has to embed assertions in C files, needs a more sophisticated parser because it has to handle more possible variations in the source code structure.

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

6.6 Construction of PECOS Assertion Blocks

As shown in Figure 17, at compile time the PECOS tool instruments the application assembly code with Assertion Blocks placed at the end of each block. Each basic block terminated by a Control Flow Instruction (CFI) is considered a block in PECOS, and each CFI is preceded by an Assertion Block. Note that the Assertion Block itself does not introduce additional CFIs. Since we are trying to protect a CFI, it defeats the purpose to have the Assertion Block insert further CFI(s). At runtime, the task of the Assertion Block can be broken down into two sub-tasks:

1. Determine the runtime target address of the CFI (referred to as X_{out} in the following discussion). We consider the following situations: (a) the target address of CFI is static, i.e., is a constant embedded in the instruction stream, (b) the target address is determined by runtime calculation, and (c) the target is the address of dynamic library call. We will discuss and illustrate each of the instances.
2. Compare the runtime target address with the valid target addresses determined by a compile time analysis. In general, the number of valid target addresses can be one (jump), two (branch), or many (calls or returns). For two valid target addresses: *X1* and *X2*, the resulting control decision implemented by the Assertion Block is shown in Figure 20. 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 20. High-level Control Decision in the Assertion Block

Example: Assertion Block for CFIs with Constant Operand

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

Figure 21(i) shows a basic block (for the SPARC architecture) terminated with the conditional branch instruction. Figure 21(ii) shows the same block with the assertion block inserted into the object code by the PECOS tool. Instructions (1)-(3) load the runtime instruction, which has the branch opcode and the runtime target address combined, into register *l7*[12]. In this computation, SPARC uses Program Counter-relative offsets for addressing. Instructions (4)-(5) load the valid target offset[13] in bytes into register *l6*. The offsets are determined by the usual compile-time creation and analysis of the application's control flow graph. Instructions (6)-(8) form the valid instruction by combining the valid offset with the opcode and then load it into register *l6*. Instructions (9)-(12) compare the valid instruction word (in *l6*) with the runtime instruction word (in *l7*) and raise a divide-by-zero floating point exception in case of mismatch.

Now consider an error case. In fault/error scenario #1 in Figure 21, an error in the memory word storing the conditional branch instruction causes a corruption of the correct memory word (hex value: 0x02800017) to an incorrect value (hex value: 0xffffffff, say). In the absence of a preemptive control flow error detection scheme like PECOS, execution would have reached the incorrect memory word, and the operating system would have generated the illegal instruction signal which, in the absence of a signal handler, would have caused the application process to crash. Now consider the assembly code segment with the PECOS Assertion Block in place. At the end of instruction (3), the incorrect memory word 0xffffffff is loaded into register *l7*. The correct memory word is loaded into register *l6*, and a subsequent comparison between the two register values causes a floating-point exception signal to be raised. By examining the PC, the PECOS signal handler will determine that the signal was raised because of a control flow error. By design, the exception is raised before the control flow instruction is executed, therefore the error cannot propagate, and the PECOS signal handler can take appropriate recovery action, e.g., in the DHCP case, shut down the offending thread. Similarly, any error that hits the PECOS Assertion Block, e.g., changing the operand of *sethi* in (4) from, e.g., 0x5c to 0xfd (fault/error scenario #2), will also be detected by the comparison and division in instruction (12).

[12] The assertion uses local registers *l5*, *l6*, *l7* which were not used by the applications in our study.

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

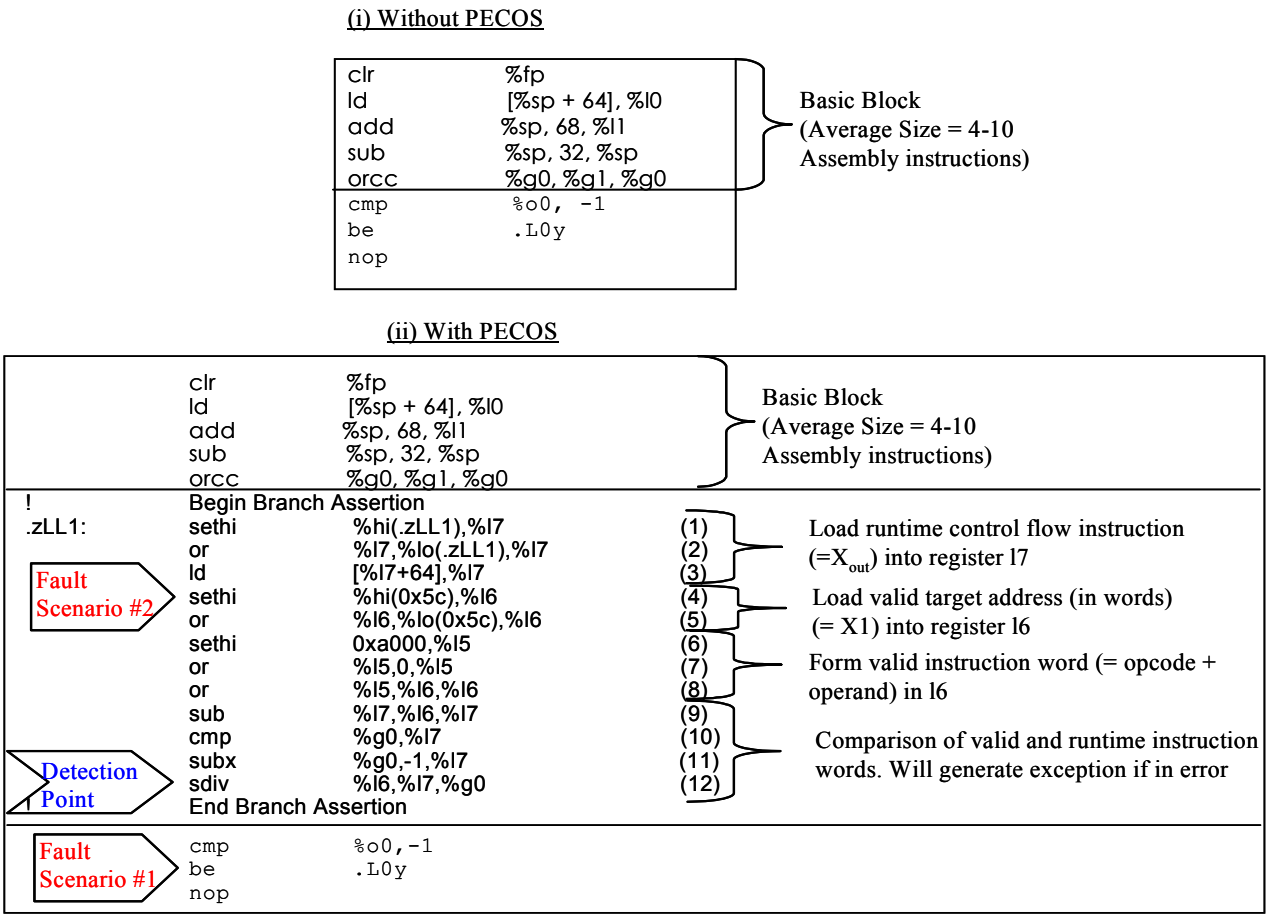


Figure 21. Assembly Code for Branch Assertion Block

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

6.7 PECOS Runtime Extension

The runtime extension of PECOS allows for protecting control flow instructions whose target addresses are dynamically determined by run-time calculations. An example of such an instruction is the register-indirect jump where the target address calculation uses the value of a register. To handle such cases, phase 1 of the PECOS analysis tool performs a data discovery pass that generates the definition-use pairs (DU pairs) for all such runtime-dependent control flow instructions. The embedded Assertion Block has instructions to read off the runtime value that determines the control flow. In Figure 22(a), the use of the register *r1* in the jump instruction is preceded by the definition of its value in instruction (i). In this simple case, the value of the register depends on its last definition only, and the definition and the use are in a single straight-line path. In this case, PECOS can insert an Assertion Block before the use, i.e., the jump instruction. Thus, instruction (ii) in Figure 22 (a) can be protected. Similar procedure can be used to protect the *jump* instruction in the scenario depicted in Figure 22 (b). A more complex situation is illustrated in Figure 22 (c), in which the value of the register (*r1*) depends on the execution path that has been followed by the application prior to the definition of the value in the register (*r1*). In this scenario *definition-use* pairs are in multiple (two in the example) paths and

to determine which path has been followed by the application requires additional support, which is being currently investigated.

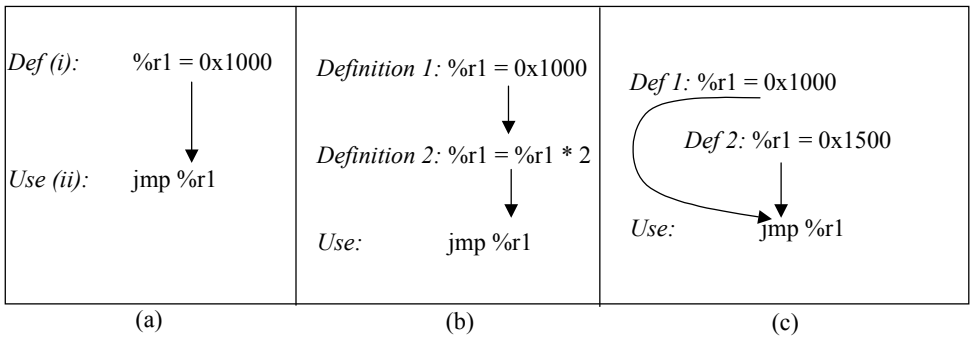


Figure 22. Applicability of PECOS to Runtime Dependent Control-flow Instructions

Another issue to address is how to handle dynamic library calls. Here we use the information provided in the Procedure Linkage Table (PLT). After the loader has loaded the application, the positions of the dynamic library calls in the application are determined and filled up in the PLT. To protect the calls to the dynamic library routines, PECOS considers the PLT as the definition point of the DU pair, with the call instruction being the use point. Then, a similar approach to the one depicted in Figure 22 is followed. The Assertion Block placed before the call asserts that the target of the call will be the location loaded in the PLT.

Example: Assertion Block for CFIs with Register Operand

An example of PECOS instrumentation is provided for a CFI for which the target address is determined by runtime calculation; a register indirect jump. Two error cases—a memory error and a register error—demonstrate how the Assertion Block captures the error. Figure 23(i) shows an unconditional jump instruction in an application's assembly code (where the location of the jump is determined at runtime by the value in register *l7*). Figure 23(ii) shows the instruction protected by a PECOS Assertion Block. In this case, to form the Assertion Block, it is not sufficient to assert the contents of a memory location; we must assert on the contents of a register. The example illustrates the ability of PECOS to form reference signatures from runtime quantities. The inserted AB examines the last definition of the register value immediately preceding the jump instruction, where a value is loaded into the register (instructions (1)-(2)). In general, the PECOS tool will have to search further definitions for the corresponding register value.

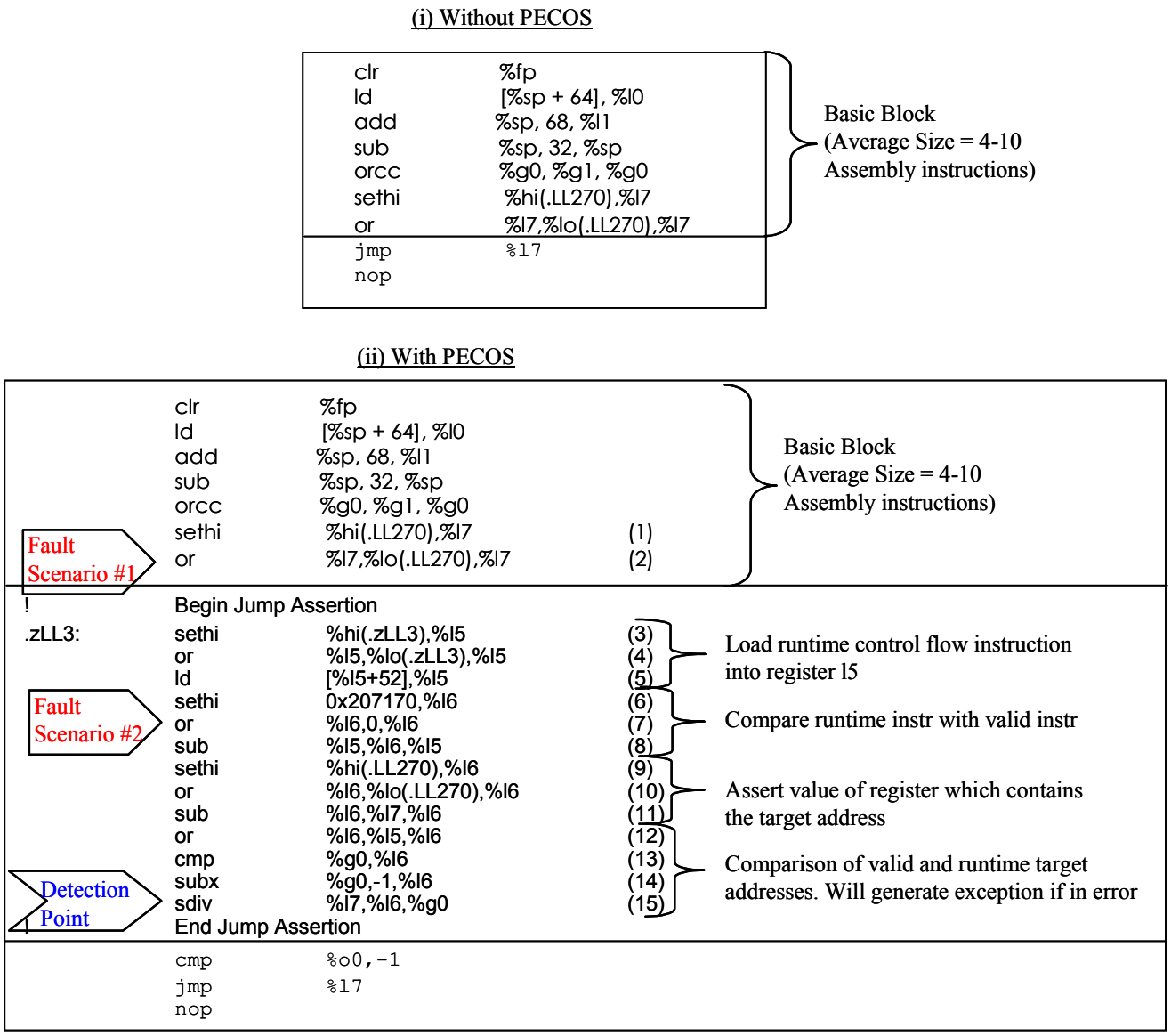


Figure 23. Assembly Code for Jump Assertion Block

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

The general structure of the Assertion Block is similar to the branch case. However, in the jump case, both the instruction and the register value must be compared to valid values. This is necessary because an invalid control flow path may be taken either if the instruction is corrupted (e.g., `jmp %l7` becomes `jmp %l6`) or if the value of the register is corrupted (e.g., value in register *l7* is changed from *.LL270* to *.LL271*). The instruction comparison is performed through instructions (3)-(8), and the register value comparison is performed through instructions (9)-(11).

Now consider an error case. In fault/error scenario #1 in Figure 23, a register error results in the corruption of the value in register *l7*. In this simple example, the time during which the execution is being protected by the PECOS Assertion Block is the time between the execution of instructions (1) and (11). In a more general case,

there will be a larger number of intervening instructions between the loading of the register value and its use in the jump instruction. In that case, the Assertion Block will be protecting a larger time window of execution. Consider the case in which the register error causes the value of .LL270 + 0x10 to be loaded into *l7* instead of .LL270. In the absence of preemptive control flow error detection, execution would have caused a jump to .LL270 + 0x10 on executing instruction (16). On executing instructions from this invalid target, the process can crash (for example, because of an illegal memory access to an unallocated memory region), or worse still, it may output a wrong value leading to a fail-silence violation. With the PECOS Assertion Block inserted, the comparison in instruction (11) will indicate an error, and instruction (15) will raise a divide-by-zero resulting in the actual detection. Similarly, if an error affects a PECOS instruction, e.g. as in fault/error scenario #2, and corrupts the operand of the *or* instruction in (7) from 0x00 to 0x45, the comparison in (8) will produce a non-zero result and instruction (15) will raise the exception thereby detecting the error.

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

6.8 Error Types that PECOS Protects Against

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

1. Single or multiple bit flips in memory—main memory, or cache (on-chip or off-chip), e.g., a burst error corrupts a call instruction in L1 cache.[14]
2. Transmission error during communication between any two levels of the memory hierarchy, e.g., an error on the address line when a control flow instruction is being fetched from memory to the processor.
3. Register error, e.g., corruption of a value in the register, which determines the destination address of an unconditional jump instruction.
4. Software bugs impacting the control flow, e.g., the call graph determined from the UML specification (a high-level specification of the software) may show that function *foo1* is called from *foo2*, *foo3*, and *fooa;* because of software bug the implemented code has a call to *foo1* from *foo5*. The automated generation of the assertions from the UML specification and their insertion into the source code are not currently implemented in the PECOS tool.

PECOS cannot capture control flow errors that result from the corruption of a non-control flow instruction into a control flow instruction, since the Assertion Blocks are inserted only before control flow instructions.

[14] 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.

However, it is observed that the Instruction Set Architectures (ISAs) are such that the Hamming distances of opcodes belonging to the non-control flow category and the control flow category are quite large, e.g., in SPARC. Therefore, it is reasonable to expect that errors that cause a non-control flow instruction to be transformed into a valid control flow instruction are rare in practice. While some signature schemes [ALK99] claim to detect such errors, our experience tells us that if an illegal control flow branch occurs due to corruption of a non-control flow instruction, the application is likely to crash before reaching the trigger for checking. This is illustrated by experiments with non-preemptive technique discussed in Section 4.9.

6.9 Optimizations

If an assertion block is inserted for every CFI, then the memory overhead of PECOS will be fairly high. Here, memory overhead is defined as the increase in the size of the application text segment. Suppose, 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:

$$C = a / n * 100\% \quad \ldots \text{(I)}$$

A typical value of *a* is 15 assembly instructions, and *n* is 10 assembly instructions. Then, the overhead is 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 (like WCP). Therefore, the overhead 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. Memory overhead of techniques in literature has been shown 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.

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. The drawback of this optimization is a reduction in coverage because control flows within a super-block are no longer monitored.

It is possible that not all the modules in the software application are equally critical. For example, for the DHCP server, the core protocol engine that decodes the request from the DHCP client and decides on the appropriate response is the most critical part of the software. PECOS allows the application designer flexibility to instrument selective parts of the application thereby protecting only the required critical portions of the software and incurring only the necessary overhead.

6.10 Extensions

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. However, the general principle of PECOS can also be applied to hardware. With hardware support, the determination of the runtime target address is done in hardware, with the valid target addresses still embedded in the Assertion Block. Figure 24 gives three possible points in the execution path where PECOS checking can be implemented. The current support for PECOS is shown in Figure 24(a). Figure 24(b) shows a hardware addition that snoops on the external address bus and data bus and computes the runtime target address. Figure 24(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.

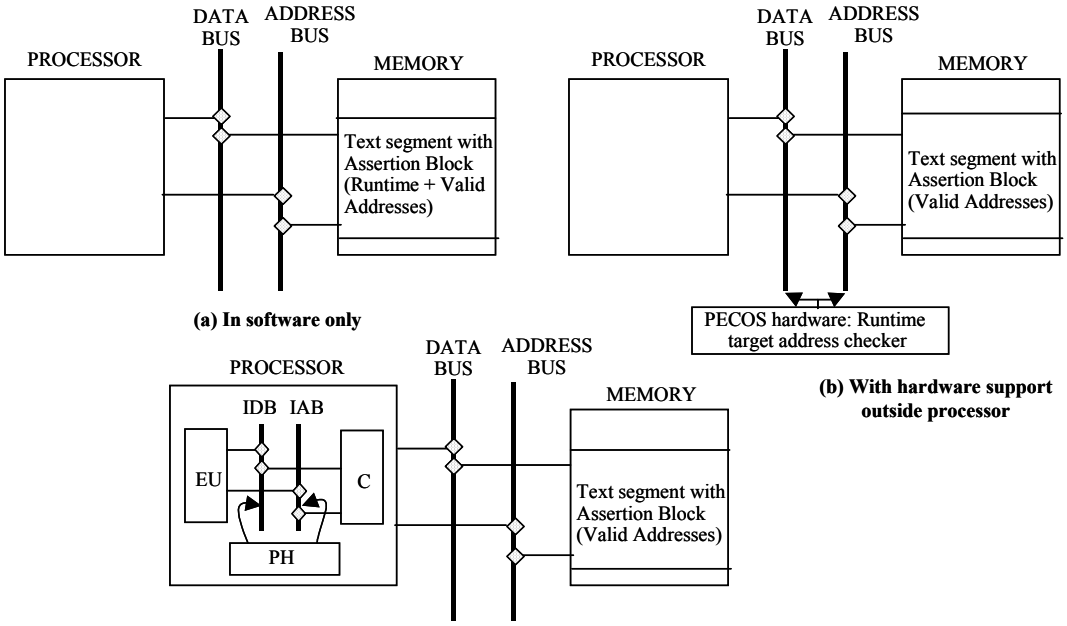


Figure 24. Different levels of PECOS.

7 Evaluation of PECOS on DHCP

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

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

Error injection campaigns were conducted into the baseline (uninstrumented) DHCP server first and then into the DHCP server instrumented with PECOS. The goal was to evaluate relative improvements in the dependability metrics, including percentage of system detection, hangs, and fail silence violations. NFTAPE, a software-implemented fault injection[15] tool, was used for conducting the error injection campaigns [STO00] using a wide range of error models.

7.1 The Target Application: DHCP

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

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

The protocol operates in two phases. In Phase 1, the client broadcasts a DHCPDISCOVER message on its local physical subnet. One or more servers on the network respond to the client request by sending a DHCPOFFER message, which contains a tentative offer of an IP address and configuration parameters. In Phase

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

2, the client collects the DHCPOFFER responses from all the servers that respond, chooses one server to interact via a DHCPREQUEST broadcast message. On receiving the DHCPREQUEST message, the chosen server commits the binding of the IP address of the client to a lease database in stable storage and responds with a DHCPACK message. If the selected server is unable to satisfy the DHCPREQUEST message, then the server responds with a DHCPNAK message. When the client's lease is close to expiration, it tries to renew the lease by re-sending the DHCPREQUEST message.

DHCP can be regarded as a representative example of a control flow intensive client-server application. It is considered control flow intensive because the lease database it has to access is kept fairly small for the experiments and most of the processing is in the protocol engine to determine the appropriate response to the client.

Version 2 of the implementation of DHCP from the Internet Software Consortium [ISC00] was 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). The configuration is shown in Figure 25.

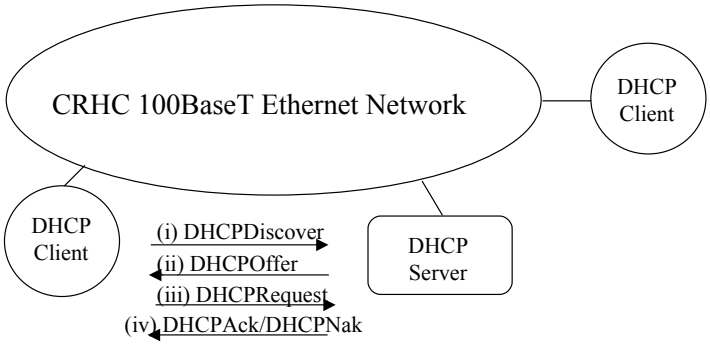


Figure 25. DHCP Client-Server configuration with protocol interactions
Messages (i),(ii) form phase 1 of the protocol, messages (iii),(iv) form phase 2 of the protocol. Phase 1 is performed once at startup, phase 2 is repeated when the leases expires after the lease period.

7.2 Types of Errors Injected

We use error models based on the extensive experiments of Abraham *et al.* [KAN95] and adding random memory errors to it. These models represent a wide range of transient hardware errors and some software bugs[16]. 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[17].

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

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

The errors chosen are depicted in Table 14. Each of these faults model a fairly wide set of actual faults that can occur in real-world systems. For example, the DATAOF models an error during transmission over data line, an error in the corresponding memory or cache word, or a software bug corresponding to an uninitialized pointer. All the errors, except ADDIF2, are single-cycle errors, while ADDIF2 is a 2-cycle error. In ADDIF and ADDIF2, the erroneous instructions that are executed are also valid instructions in the instruction stream of the application. The ADDIF2 error may be caused, for example, if the Program Counter (PC) register has a stuck-at error that lasts two cycles. In the DATAInF error model, it is assumed that the data line error may affect either the opcode or the operand of the instruction, and this fault type models random memory errors.

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 14. Transient Error Models for PECOS Evaluation

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

7.3 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 7.1[18]. 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 15.

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

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

Table 15. Categorization of Results from Error Injection

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

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

7.4 Results

This section presents a summary, detailed results, and discussion of the results from the error injection campaigns. Table 16 and Table 17 present, respectively, the results of the injections directed into control flow instructions and the results of injections into instructions randomly selected from application instruction stream. Each row in the two tables gives a sum of the number of cases from all error injection campaigns. The fourth column gives the improvement due to the PECOS instrumentation. Reduction in system detection, hang, program aborts, and fail-silence violations are considered improvements.

The results in Table 16 characterize the effectiveness of PECOS in detecting errors when an error directly affects a control-flow instruction, i.e., the effectiveness in detecting errors, PECOS was designed to protect against. The major improvements gained by instrumenting the DHCP server with PECOS are:

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

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

Table 16. Cumulative Results from Directed Injection to Control Flow Instructions

The results in Table 17 provide an insight into how well PECOS performs when the corrupted instruction is not necessarily a control flow instruction, but 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 the randomly corrupted instruction immediately crashes the process before the process executes the PECOS assertion block. Table 17 shows that the proportion of PECOS detection, reduction in fail-silence violations, and system detections are more moderate than for directed control flow injections. From the results, one can draw the following conclusions:

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

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

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

From the observed results, C = 87.5% (Table 16: directed injection to control flow instructions), D = 31.5% (Table 17: random injection to the instruction stream) and therefore, X = 36.0%. This number agrees well with that of Ohlsson *et al.* [OHL92], where simulation of a 32-bit RISC processor showed that 33% of activated errors manifested as control flow errors. This result also indicates that in addition to control flow checking, it is important to have effective data error detection mechanisms to improve overall system reliability.

Category	Without PECOS	With PECOS	Improvement Factor {(measured value w/o PECOS) / (measured value with PECOS)}
Error Not Activated	2486	2476	n/a
Error Activated but Not Manifested	770 (50.9%)	513 (33.7%)	n/a
PECOS Detection	n/a	480 (31.5%)	n/a
System Detection	610 (40.3%)	483 (31.7%)	1.3
Application Hang	22 (1.4%)	2 (0.1%)	14.0
Program 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 17. Cumulative Results from Injection to Random Instructions from the Instruction Stream

7.5 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[19] or creating a new thread. Conservatively, this is less expensive in terms of time overhead by a factor of 10. Our best case measurements (taken on the SPARC platform running the Solaris operating system) show that the time to create a process takes of 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 18 gives the estimation of the downtime improvement for the midrange factor of 100 (the ratio between the time for process crash recovery and thread recovery) and Figure 26 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 16).

	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%	0.071 * t_{rec} 0.475 * t_{rec}/100	

Table 18. Estimate of Reduction in Application Downtime

[19] For example, in a 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 desirable way of recovery since we lose a single request not the entire application.

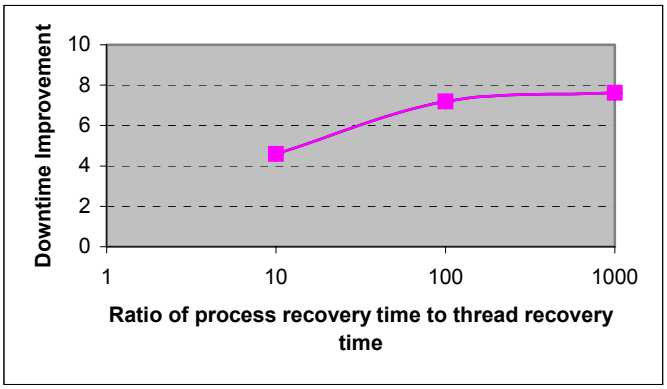


Figure 26. Downtime Improvement Due to Reduction in System Detection for Different Ratios of the Costs of Process Recovery and Thread Recovery

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

7.6 Detailed Results & Discussion

The results of all error injection campaigns are presented in Table 19. The percentage of runs belonging to each category is mentioned, along with the actual number of runs falling in each category (in parentheses). Recall that the number of injected errors, which were not activated are discarded in our calculations. As a result, the total of the number of runs for each category does not add up to 500, and does not add up to the same figure for the different campaigns. For the PECOS instrumented server, the error activation rate falls between 28.6% (ADDOS) and 49.8% (ADDIF, only CFI injection). For the uninstrumented server, the error activation rate falls between 21.8% (DATAOS) and 50.6% (DATAOF, only CFI injection). Some important observations from the results in Table 19 are:

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

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

Table 19. Results of Error Injection to DHCP Server

Throughout this paper, we have emphasized the importance of reducing cases of system detection, application hang, and fail-silence violation to minimize the downtime and thus improving availability. An important benefit of fast error detection is the ability to reduce the likelihood of error propagation. Our measurements show that between tens and thousands of instructions can be executed from the time when the corrupted instruction is executed and when the process crashes. This window of system vulnerability may be enough to corrupt significant system data or impact key system resources or even crash the operating system. The results in Table 19 clearly indicate that PECOS is very efficient in reducing the system vulnerability window.

7.6.1 Effect of PECOS on System Detection

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

For the ADDOS and DATAOS error types we observe a 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 runtime extension to PECOS presented in Section 6.7 enables detection of these types of errors.

7.6.2 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), or (2) the application taking an incorrect rather than an illegal branch that is not detected by the current implementation of PECOS' runtime checking ability.

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

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

1. The server does not get the applicable record for the client host generated after phase 1 of the protocol and therefore sends a DHCPNAK in the second phase.

2. The server gets DHCPDISCOVER but does not respond with DHCPOFFER because it believes erroneously that there are no free leases on the subnet.

3. The server's response to the client causes the client to make a wrong protocol transition. As a result, the client does not try to renew its lease but continues to use the old lease and the server allows the client to do so.

7.6.3 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 reduce the need for this type of checks and, consequently, reduce the overall detection latency.

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

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

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

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

Finally, Table 19 shows that PECOS reduces the number of *no error* cases (i.e., execution of the erroneous instruction did not cause any error). This indicates that PECOS detects cases where the error would otherwise not have impacted the application. Since PECOS is a pre-emptive technique, it is expected to suffer from the problem of false alarms. It should be noted however that some of the non-manifested errors in the experiments could have stayed latent and may have been activated at a later time and therefore should be detected.

7.7 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 20. The measurements are given separately for the two main phases of the protocol.

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 20. 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. The entire DHCP server consists of 30 source files (written in C) and includes all the support functions, like function to receive a 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[20]. The results show overheads in the range of 5-20% if only the core protocol engine of the server is instrumented.

To get a better understanding of the percentage of the entire code that is instrumented with PECOS Assertion Blocks, statistics about subroutine calls in the application code are presented in Table 21. It may be recalled that the current PECOS implementation cannot protect the calls made to subroutines in dynamic libraries. As a result 64% of the calls in the DHCP server code can be protected. It is found by analysis of the error injection logs that some of the cases of lack of coverage of the PECOS technique are due to this deficiency. From Table 21, it is seen that 81% of the protected calls span two files. This indicates the importance of the ability of PECOS to instrument calls and returns that span multiple files.

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

Table 21. Statistics on Subroutine Calls in DHCP

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

7.8 Impact of Errors on Checking Code

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

To determine whether error detection is triggered if the Assertion Block is injected, and whether the PECOS instructions contribute to the fail-silence violation, an additional error injection campaign was performed where only the PECOS Assertion Block instructions were injected. The results are shown in Table 22. The results show that the PECOS Assertion Blocks do not cause any fail-silence violation. Consequently it can be concluded that the PECOS instrumentation does not add any extra vulnerability to the application. Approximately 88% (51.9 / (51.9 + 6.9 + 0.5)) of the errors that were injected into the Assertion Blocks and got manifested triggered PECOS detection.

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

Table 22. Comparison of Directed Injections without PECOS and into the PECOS Assertion Blocks

7.9 Comparison with ECCA

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

To resolve this ambiguity we decided to recreate the experiments from [ALK99]. Towards this end, a parser developed in Duke University was used to instrument the application with ECCA assertion blocks[21]. The same

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

application, namely *espresso* from the SPECInt integer benchmark suite is used as the target of the evaluation. A subset of the error models as presented in Table 14 is injected using NFTAPE to conduct the study.

Table 23 presents error injection results obtained for four error models—ADDIF, DATAIF, DATAOF, and DATAInF. 500 errors from each error model are injected into the target application, once for the baseline case (i.e., the application without ECCA signatures) and once for the ECCA-instrumented application. For each combination of the application and error model, errors are injected (1) randomly in the text segment of the application and (2) directed at the control flow instructions. Thus, a total of 16 (4 * 2 * 2) campaigns are defined, and the total number of error injection runs is 8,000. The cases where the error is not activated are discarded. The error activation rate is higher for the uninstrumented code (from 46.2% to 57.0%) than for the ECCA instrumented code (from 30.4% to 31.4%). The possible explanation for this is that the ECCA instrumentation results in expanding the text segment of the application with additional code not all of which is executed during the normal run. To identify the fail-silence violations for the espresso application, the application's output is dumped to a file and compared against a golden output file. A mismatch indicates a fail-silence violation. In Table 23, ED indicates detection by ECCA.

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

Table 23. Results of Evaluation of ECCA applied to the *espresso* Benchmark Program

The important conclusions from our experiments are:

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

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

8 Evaluation of Data Audit and PECOS on WCP

8.1 Introduction

Application availability can be compromised because of errors that affect application execution flow or due to corruption of data used during the execution. In this chapter, we present the design and implementation of a wireless call processing (WCP) environment for a digital mobile telephone network controller whose availability is dependant on both control and data integrity. The environment includes a database subsystem containing configuration parameters and resource usage status and call processing clients for setting up, managing, and tearing down individual calls. Both subsystems are vulnerable to errors, which can manifest as serious system outages affecting hundreds or thousand of customers. Consequently, we need efficient mechanisms to provide detection and recovery from both control and data errors.

PECOS is applied to detect control flow errors in the call processing application threads. However, since integrity of the database is also critical to the overall system availability, an integrated approach combining control and data protection needs to be followed. Towards this end, we investigate and evaluate the efficiency (in terms of coverage) of data audits and control flow checking in protecting respectively the database and the call processing clients of the mobile network controller. The data audit subsystem was designed by Yiyuan Liu and is presented in detail in [LIU00]. Here, we present the main aspects of the subsystem, which are important in understanding its role in the overall dependable WCP environment. The database contains both static system configuration parameters and dynamic system state information and provides basic services such as read, write, and search operations to the system and application processes. As the data stored in the database is subject to corruption due to a variety of hardware and software errors, database audit plays an important role in maintaining data integrity to effectively support overall system function.

The integrated call processing environment with both data and client protection is implemented using the ARMOR-based framework of Chameleon [KAL99].

The main contributions of the work described in this chapter can be summarized as follows:

1. Design and implementation of a generic, extendible, ARMOR-based framework for providing data audits and control flow checking to applications. In this framework new detection and recovery techniques can be integrated into the system with minimum or no changes to the application.
2. Detailed evaluation of the proposed framework being used to provide integrated control and data error detection and recovery in a wireless call processing environment. The software fault injection based evaluation quantifies:
    - combined coverage provided by data audit and control flow checking in protecting the call processing environment
    - chances of error propagation between the database and the client (and vice-versa).

The evaluation of the environment with both control and data protection shows that:

1. For corruptions to the database subsystem
    a. Data audit detects 84.8% of the errors, and
    b. The incidence of escaped faults is reduced from 62.8% to 13.4%;
2. For control flow errors in the call processing clients
    a. In the absence of any detection in the client, 14.0% of injected faults propagate to the database,
    b. There are no fail-silence violations and no client hangs, and
    c. The incidence of client process crash is reduced from 48.0% to 18.5%.

8.2 Overview of the Target System Software and Database Architecture

The target system is a small-scale digital wireless telephone network controller that integrates many functions in standard wireless telephone network components (including call switching, packet routing, and mobility management). The key components of application software running on the controller are: call processing application (sets and terminates client calls), and database (supports information about system resources)[22].

8.2.1 Call processing client

Call processing is the key component that provides customer-visible functionality. This process dynamically creates a call processing thread to handle activities associated with a voice/data connection. Specifically, the thread is responsible for subscriber authentication, hardware resource allocation and de-allocation, mobility management, and supporting basic and supplementary telephony features. The failure of the call processing task alone could render the entire system unavailable from a user's point of view. It is therefore necessary to monitor its status during system operation. Since call processing requires database access, we refer to the application as a database client.

8.2.2 Database Subsystem

The database subsystem is a critical component as it contains data (static system configuration data and run-time data that indicate resource usage and process activities) necessary to support the operation of application processes, including call processing. As most of the controller operations require database access, any data error or data structure corruption in the database could have a major impact on system performance and availability.

Organization. To satisfy the real time constraints, the entire database is loaded from disk into memory at startup time and resides completely in memory during system operation. The memory region that contains the database is contiguous and is shared among all processes that require database access. To remove the possibility of

[22] Other application components include system maintenance, network management, and system monitor.

memory leak and ensure continuous system operation, the space for all tables in the database is pre-allocated, and no dynamic memory allocation is used in the database. Therefore, during normal operation the size of the memory-based database stays constant.

The database consists of various tables with a pre-defined size that occupy the memory space one after another. Although the entire space is statically allocated, some tables are dynamic in nature while others are static. Static data in tables usually refer to system configuration (e.g., the number of CPUs in the system) that stay constant during operation, whereas dynamic data is often updated, e.g., on every incoming call. Note that each table usually contains a mixture of static and dynamic data.

The database subsystem exports an API for other processes to access the database. Each API primitive performs the requested operation such as write a record or move a record. Some API functions along with brief explanations are shown in Table 24.

DBinit	Initialize client connection to the database
DBclose	Close client connection to the database
DBread_rec	Read a record (row) in a table
DBread_fld	Read a field in a table record
DBwrite_rec	Write a record (row) in a table
DBwrite_fld	Write a field in a table record
DBmove	Move a record to another logical group

Table 24. Examples of Database API

8.2.3 Errors in Database

The database is subject to corruption from a number of sources: human (operator) error, software bugs or faults, and hardware faults. Static system configuration information is typically entered by operators through a user interface. Human error, such as misunderstanding of procedure or interface functions, could result in inconsistent data or data structures. Software bugs uncaught during testing phase could also cause program to write incorrect data or write into incorrect memory locations. Run-time software failure is another cause for data errors. For example, if a process that is in the middle of updating a set of database records crashes, the set of data would likely be in an inconsistent state. Memory hardware or environmental factors can also trigger memory mutilation, although basic hardware mechanisms such as ECC mask some of these faults.

The effects from these types of errors appear in several ways. The most serious consequence occurs if errors are introduced into the system catalog (metadata) of the database. The system catalog contains information used by the database API to access records and consists of several database tables that are referenced on each database operation. Errors in the system catalog can cause all database operations to fail, thus bringing down the whole controller. Reduced system availability is another consequence of data errors. If any data indicating resource status is corrupted, the particular resource in question could falsely appear as “busy” to the rest of the controller (“resource leak”), leading to a reduced system capacity. If this type of error is allowed to accumulate, system availability will be reduced. Other errors may have a local effect only. For example, a damaged call record related to a particular connection will most likely bring down that connection prematurely without

affecting other active connections. It is also possible for errors to have no effect on the system if the errors occur at memory locations that are not used or if they are overwritten.

To improve system availability, it is important not only to prevent errors from occurring, but also to detect and recover from errors quickly. Simplification and standardization help reduce operator and programmer errors, but the possibility of errors due to run-time software or hardware anomaly remains. While exact figures are not publicly available, data in telecommunication industry has repeatedly shown that not performing database audits significantly reduces telephone switching system availability [LEV99].

8.2.4 Errors in Call Processing Threads

The call processing application threads can have control flow errors caused by corruptions to the control flow instructions or from any other corruption (such as a register error) that subsequently affects the control flow. The low-level faults causing control flow errors were presented in Section 6.8. The application does not contain any internal data. All the data that it processes is read off from the database. Hence, we do not consider data errors in the call processing threads. However, there can be errors in the instructions that manipulate the data, and these are considered in the fault injection experiments that were conducted.

8.3 Audit Subsystem Architecture

This section presents the data audit techniques for protecting the database in the call processing environment. The techniques were designed and implemented by Yiyuan Liu and were first presented in [LIU00]. Here, mention is made of the techniques that were used during the experimental evaluation of the environment. The audit techniques are implemented using ARMOR technology that permits the audits to be extensible and added incrementally as the system evolves.

The overall design of the database audit process and its interactions with other system components are shown in Figure 27. The right half of the figure shows the database, a client process, and the database API that is used by client processes to access data. A communication channel (the message queue) is added between the database API and the audit process to transmit events from client activities. The modified database API supports and maintains the communication channel and other data required by the audit process (discussed later). The audit process consists of a custom-designed ARMOR and a dedicated thread (the main thread) acting as the interface to the other components. The function of the main thread is to translate information from the other entities, e.g., the database API, into ARMOR messages via the ARMOR API. The main thread allows other processes to communicate with the audit ARMOR using standard inter-process communication mechanisms. The audit ARMOR is used as the vehicle to provide audit functionality and consists of the top-layer shell and the individual elements that implement specific audit triggering, error detection and recovery techniques. The elements depicted in Figure 27 include: heartbeat (*HB*), progress indicator (*Prog. Ind.*), audit (*Audit Elem.*) which encapsulates a set of error detection and recovery techniques, periodic audit (*Per. audit*) and event

triggered audit (*EvTrig audit*) which support, respectively, periodic and event triggered invocation of different audit techniques.

To reduce contention with database clients, the audit elements access the database directly instead of through the database API. Bypassing the locking and access control mechanisms managed by the API reduces performance penalty, but it requires the audit process to detect database access conflicts between clients and itself to ensure the validity of audit results.

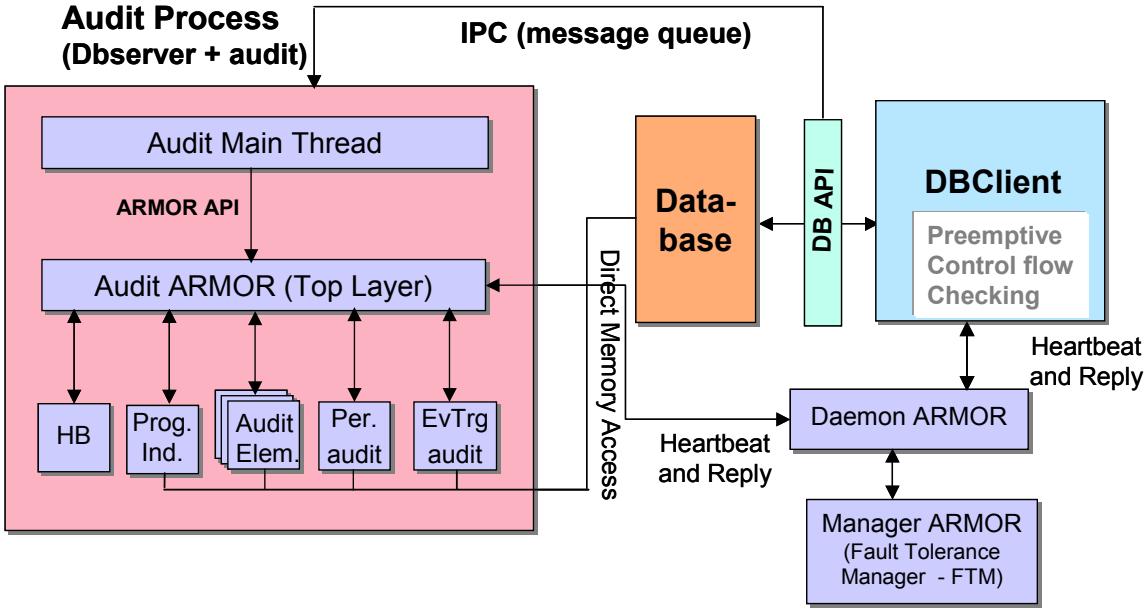


Figure 27. Target System with Embedded Audit and Control Flow Checking

The ARMOR-based framework provides high modularity and transparency allowing for easy extensibility of the audit subsystem. New error detection and recovery techniques can be implemented in new elements and added into the audit ARMOR. The different audit elements can be quite independent of each other, which allows for easy customizability of the audit subsystem to handle different fault models. For a new technique to be incorporated into the system, the element simply needs to communicate the messages that it accepts, to the Audit ARMOR and a new function to invoke the newly added element needs to be defined.

8.3.1 The Heartbeat Element

Detection - The heartbeat element and the ARMOR daemon together implement a heartbeat protocol that allows the daemon to detect audit process failure. Periodically, the daemon process sends a heartbeat message to the heartbeat element in the audit ARMOR and waits for a reply. If the entire audit process has crashed or hung, or if there is scheduling anomaly on the controller system that prevents the audit process from running, the daemon will time out while waiting for the heartbeat reply.

Recovery - When the daemon times out in waiting for heartbeat reply, it assumes that the audit process has failed and notifies the manager ARMOR, allowing the manager to restart the audit process.

8.3.2 The Progress Indicator Element

The progress indicator element is used to detect deadlock in the controller database to ensure uninterrupted system operation. The database API maintains and manipulates locks transparently to the client processes to synchronize their access to the database. If a client process terminates prematurely without fully committing its transaction, the locks left behind by this process would prevent other client processes from accessing the database (or portions of it). The database API is a passive entity and is not capable of detecting and resolving deadlocks, so it is important to have deadlock detection as part of the audit process.

Detection - A standard POSIX IPC message queue is added between the database API and the audit process (see Figure 27). The database API is modified to send a message to the audit process whenever any API function is called. The message contains the client process ID information and the database location being accessed. These messages are used to increment the counter in the progress indicator element as they indicate ongoing database activity. If the audit process receives no message, leaving the counter value unchanged for an extended period of time, the progress indicator element will time out and trigger recovery.

Recovery - The progress indicator element terminates the client process holding the lock for greater than a predetermined threshold duration thereby releasing the lock. While the threshold for a client to hold a lock is typically small (e.g., 100 milliseconds in the current implementation), the progress indicator timeout value is much larger (e.g., 100 seconds in the current implementation) in order to reduce run-time overhead.

8.3.3 The Audit Elements

Specific audit techniques are implemented as separate audit elements (audit elm. in Figure 27) in the ARMOR. The invocation of the audit elements can be either by a periodic trigger, or by an event trigger. The periodic trigger is based on a fixed time period. The event trigger is provided by some specific database operations, e.g., database write in the current implementation. The database API is modified to send a message to the audit ARMOR after each database update. The message contains the ID of the initiating client process and the database location being accessed. The periodic audit element uses as its basis the periodic heartbeat query discussed earlier as trigger to perform the following specific audits: static data integrity check, dynamic data range check, structural audit, and referential integrity audit. The audit subsystem can be configured to use different events to trigger the audit elements, e.g., when the system enters a critically low available resource state. If there is an intervening update to a record being accessed by an audit element, while the audit is in progress, the result of the audit is invalidated and the procedure is run at a later time.

8.3.4 Static & Dynamic Data Check

Detection & recovery - System configuration parameters and some database metadata (e.g., table ID/size/offset) seldom change and are considered static during run-time. Since the values of the static data stay constant, the audit element detects corruption in static data region by computing a “golden” *checksum* of all static data at startup and comparing it with a periodically computed checksum (32-bit Cyclic Redundancy Code). The

recovery for static data corruption involves reloading the affected portion from permanent storage (e.g., the disk).

To make audit on dynamic data possible in the target database, the range of allowable values for database fields are stored in the database system catalog. This information allows the audit program to do a "*range check*" on the dynamic fields in the database. Obviously, this technique cannot detect all corruptions in dynamic data as the exact correct value is unknown to the audit program. The semantic audit, discussed later detects data errors with higher certainty. If the audit detects an error, the field is reset to its default value, which is also specified in the system catalog. In addition, if the table where the error occurred is dynamic, the record is freed as a preemptive measure to stop error propagation.

8.3.5 Structural Check

The structure of the database in the controller system is established by header fields, that precede the data portion in every record of each table. These header fields contain record identifiers and indexes of logically adjacent records.

Detection – The structural audit element calculates the offset of each record header from the beginning of the database based on record sizes stored in system tables (all record sizes are fixed and known). The database structure, in particular the alignment of each record and table within the database, is checked by comparing all header fields at computed offsets with expected values.

Recovery – A single error in record identifier is correctable because the correct record ID can be inferred from the offset within the database. However, multiple consecutive corruptions in header fields are considered to be a strong indication that tables or records within the database may be misaligned, and the entire database is then reloaded from the disk to recover from the structural damage[23].

8.3.6 Semantic Referential Integrity Check

To verify data semantic, the audit process performs semantic referential integrity checking which traces logical relationships among records in different tables. *Referential integrity* requires that the corresponding primary and foreign key attributes in related records match. Corruption of key attributes leads to "lost" records, as some records participating in semantic relationships "disappear" without being properly updated. We refer to this phenomenon as *resource leak*, which is similar to memory leak found in many software systems. Since database records are limited resource themselves, they must be freed properly from semantic relationships to ensure their continuous availability.

---

[23] The use of doubly linked list as the data structure for logical groups within the database allows single pointer corruption to be detected and corrected using robust data structure techniques (e.g., traversing the list of table records in both directions and making proper pointer adjustments) [SET85]. These techniques are not currently implemented, as they require changing the database structure (creating a double link list) and impose unacceptable database down time from client perspective (locking of the linked lists).

Theory of Operation – If tables are considered as sets, and records as elements of these sets, then the semantic constraints of the application can usually be expressed as 1-to-1 or 1-to-N correspondences among these sets. Since a 1-to-N relationship is equivalent to N 1-to-1 relationships, we can construct many 1-to-1 correspondences among records of related tables. These correspondences logically form a "chain" among each set of related records. Furthermore, such a "chain" may be turned into a "closed loop," thereby making it 1-detectable, by connecting the head and tail with an extra correspondence. Referential integrity audit follows semantic linkages among related records in different tables to verify the consistency of these logical "loops" and detect invalid data that are impossible to find when records are examined independent of each other.

As an example, consider the data structure established when servicing a voice connection. A process (or thread) has to be spawned to manage the connection; the process needs to allocate hardware resources and record information related to the call (e.g., the IDs of the parties involved) into the connection table. Thus, a new record needs to be written into each of the following three tables:

Process Table (Process ID, Name, Connection ID, Status, ...),
Connection Table (Connection ID, Channel ID, Caller ID, ...),
Resource Table (Channel ID, Process ID, Status ...)[24]

Clearly, the three new records form a semantic loop because the process record refers to the connection record via the "Connection ID" attribute, the connection record refers to the resource record via the "Channel ID" field, and the resource table "closes" the loop by pointing back to the process record via "Process ID." Thus, the audit program can follow these dependency loops for each active record in each of the three tables and detect violations of semantic constraints.

Detection - The semantic audit component of the periodic audit element checks referential integrity based on a set of predetermined inter-table relationships. It examines all active records in logically related tables to make sure that inter-table record references are consistent, as described in the above example.

Recovery - The recovery actions include *freeing "zombie" records* and *preemptively terminating the processes/threads* that are using these records. Preemptive process termination is desirable because it keeps system resources available even though an active connection may be dropped. The termination is made possible by modifying the database API to maintain, along with each database record, the ID of the client process that last accessed the record. The redundant data structure associated with the database records also includes the time of last access and counters that maintain database access frequencies.

Commercial off-the-shelf database systems from Oracle [ORACLE], Sybase [SYBASE], and more recently, in-memory database from TimesTen [TIMESTEN], all include utilities to perform consistency check of database integrity. These utility programs (e.g., OdBit from Oracle, and DBCC from Sybase) are typically invoked on the command line interface by database administrators to perform physical and logical data structure integrity

[24] Underlined attributes are the *keys* of their respective table.

checks. For example in relational databases such as Oracle the core database engine supports set of rules for identifying relations/dependencies between tables or records in the database. These rules can be used for detecting structural and semantic errors in the database by performing referential integrity checking or structural checking [COS00]. However, the lack of a fault-tolerant infrastructure that ties together the database fault detection and recovery elements is a major limitation in the existing systems, especially when continuous availability and integrity of the database are required. Since no timestamps or process ID for the last access to a record are maintained, automatic recovery action cannot be taken. It must however be noted that some of the recovery actions by the audit subsystem are dependant on domain specific knowledge about the application using the database, in this case the call processing clients. For example, dropping a record as part of recovery is considered tolerable because in our environment, it implies dropping one active call. For a generic database, like Oracle, such recovery action may not be possible in the absence of domain knowledge about the application using the database. Having said that, we should emphasize that in this study, the data audit subsystem has been applied to a fairly large database in a commercial environment and its effectivenes substantiated through a detailed evaluation.

8.4 Evaluation of Audit Effectiveness

To demonstrate the usefulness of the audit process in protecting database clients from errors in database, a program that emulates actual call processing activities is used. The program uses multiple threads to concurrently handle incoming calls. The steps followed in each call processing thread include: authentication, resource allocation, and other phases in a typical call setup, as shown in Figure 28.

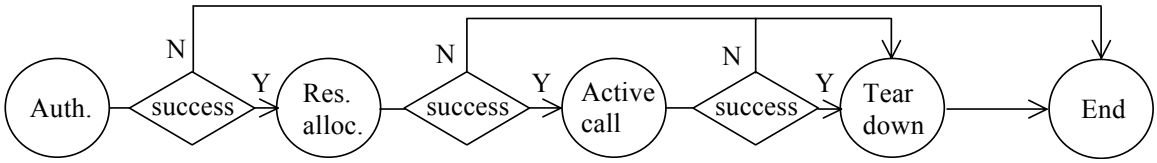


Figure 28. Call Processing Phases Emulated in the Client Program

A run of the experiment lasted 2000 seconds, during which approximately 1000 calls were processed by the multi-threaded client program depicted in Figure 28. Random bit errors were inserted into the database at various rates, and the number of errors that escaped from the audits as well as the average call setup time were recorded. The goal of the experiment was to determine the reduction in the number of escaped faults (i.e., faults that impact the database client program) when audits are used. Table 25 shows the parameters used in the experiments.

Duration of each run	2000 seconds
Number of client threads	16
Call duration	20 ~ 30 seconds
Average call inter-arrival time	10 seconds
Fault inter-arrival time	2,4,6,8,10,12,14,16,18,20 seconds
Interval of periodic audit	10 seconds

Table 25. Experiment Parameters for Evaluation of Audit Effectiveness

Table 26 shows the results of running the call-processing client program with and without database audit at a fixed fault rate of 1 fault every 20 seconds, using data from 30 runs of the experiment. Without database audits, 1884 out of the 3000 injected faults (63%) affected the application process, compared with only 402 (13%) when audits are running. Timing of audit invocation with respect to fault occurrence and accuracy of constraints (e.g., range limits) are the two major factors in determining the number of escaped faults. As the audits are invoked periodically in the experiment, every piece of data that is corrupted and then used by the client application between two consecutive audit invocations leads to an escaped fault. In addition, the audit does not always have enforceable rules to detect all errors. The number of faults having no immediate effect, i.e., latent faults, is also greatly reduced (from 1116 to 55) when audits are running, because the entire database is checked for errors periodically. Due to the processing time required by the audits, the average call setup time in the client process changes from 160 milliseconds to 270 milliseconds, i.e., a 69% increase, as measured on a Sun UltraSPARC-2 system.

Total number of injected faults = 3000	Without Audits	With Audits
Number of faults escaped from audits and affecting application	1884 (62.8%)	402 (13.4%)
Number of faults caught by audits	N/A	2543 (84.8%)
Other (number of faults escaped from audits but having no effect on application)	1116 (37.2%)	55 (1.8%)
Average call setup time (msec)	160	270

Table 26. Comparison of Running Client Process with and without Audits using a 20-second Fault Inter-Arrival Time

Table 27 shows a more detailed breakdown of the data presented in the last column of Table 26. The faults are classified based on their types, and faults that are not induced by the random bit-flip process (i.e., process failure and deadlock) are not shown. The results show that structural audit and static data audit are very effective in detecting and removing errors and both achieve coverage of 100%. On the other hand, dynamic data audit has a lower coverage and is able to detect and remove a total of 79% (45% + 34%) of all errors in dynamic fields through range check and referential integrity check. 4% of the faults escaped detection because of the lack of enforceable rules available to dynamic data audit. Suitable constraints need to be added to the database in order to improve audit coverage in this category. Another 14% of faults escaped because the erroneous data was used by the application process before the audit could detect them. More frequent invocation of audit is needed to reduce the number of such faults escaped due to timing.

Fault types	Structural		Static Data		Dynamic Data				
	Detected	Escaped	Detected	Escaped	Detected		Escaped		No Effect
					By Range Check	By Semantic Check	Due to timing	Due to lack of rule	
Number of faults	194	0	623	0	975	751	313	89	55
Percentage	100%	0%	100%	0%	45%	34%	14%	4%	3%

Table 27. Breakdown of Inserted and Detected Faults

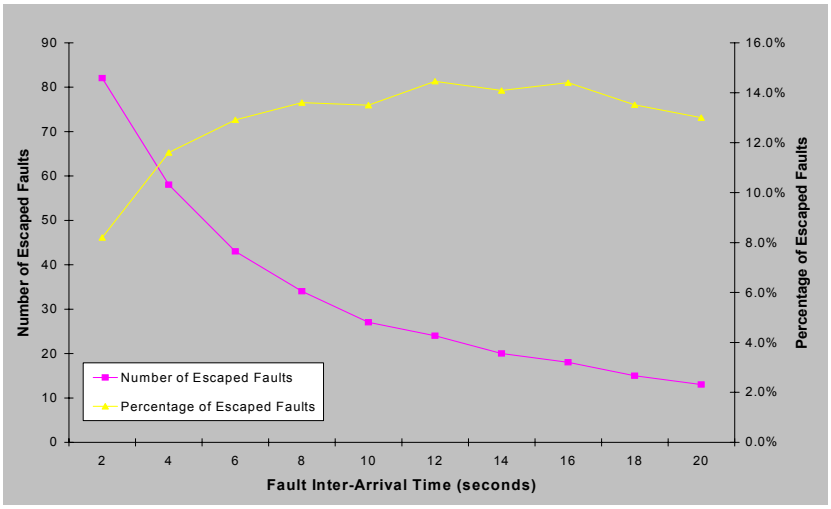


Figure 29. Number of Escaped Faults under Different Fault Rates

To further investigate the relationships among audit coverage, audit invocation frequency, and fault rate, the inter-arrival time of random faults is varied from 2 seconds to 20 seconds. All other parameters remain the same as shown in Table 25. Figure 29 shows that the number of escaped faults increases as the fault inter-arrival time decreases (fault rate increases). More importantly, as the fault inter-arrival time drops below 10 seconds (the audit period used in the experiments), the increase in the number of escaped faults speeds up as the audits start to become overwhelmed by the burst of errors. The curve shows gradual change instead of an abrupt jump as the fault inter-arrival time crosses the audit period, which indicates continuity in the performance of audit. The percentage of escaped faults in all injected faults remains relatively constant and ranges from 8% to 14%. In fact, as the fault rate increases above the audit frequency, the percentage of escaped faults increases slowly. These results show that the database audit is useful in removing data errors and preventing error propagation under different fault rates and does not break down even when the fault rate is high.

The results also show that even though the audits are effective in fault detection and recovery, there is no guarantee that they can remove all data errors and in 13.4% of cases the fault is propagated to the client. Therefore, it is important to build checking at the database client to prevent failures of the entire call processing application due to database corruptions.

8.5 Evaluation of PECOS on WCP

To evaluate the effectiveness of PECOS preemptive control flow checking in protecting the database client a set of fault injection experiments is performed. The system configuration is analogous to the one presented in Figure 27 with PECOS embedded into the client code. The database client runs for 200 seconds. Typically, in this period, about 40 calls are simulated by the call processing client. Corresponding to the four possible states of activation for the two techniques, we define four sets of campaigns: (1) without PECOS, without Audit, (2) without PECOS, with Audit, (3) with PECOS, without Audit, and (4) with PECOS, with Audit. For each campaign the call processing client is the target of fault injection. Four of the eight fault models used in the DHCP evaluation (Table 14) are chosen for this study. For one set of campaigns, injections are done in the entire text segment of the application, and for the second set, injections are performed only to the control flow instructions in the text segment. Thus, a total of 4 * 4 * 2 = 32 fault injection campaigns are defined. For each campaign, 200 runs are performed, thus giving a total of 32 * 200 = 6,400 runs for the call processing workload.

A breakpoint is used as the trigger for the fault injection and in each run only one fault is injected. However, interestingly, a single fault being injected may affect multiple client threads. Since call processing is a multi-threaded application, even if a fault is injected into a single instruction, it is possible that another thread may execute the same faulty instruction. In the fault injection methodology followed, once a thread reaches a breakpoint, the fault is injected, the thread is made to execute the faulty instruction, and then the fault is removed. But, in the time interval between the breakpoint being reached and the correct instruction being restored, other thread(s) may come and execute the faulty instruction as well. Therefore, cases of multiple faults being activated are observed.

8.6 Error Classification

The classification of outcomes from fault injection experiments is similar to the one used in the DHCP evaluation and is repeated here in Table 28. A run is considered to be in the "Fault Activated but not Manifested" category if the faulty instruction is executed, none of the detection schemes flags an error, the comparison of the database records by the client prior to termination of the thread (step (5) in Figure 30) does not detect a mismatch and the client prints the message indicating it completed successfully. If neither PECOS detection nor system detection flags an error, the final comparison of golden and runtime records by the client does not detect a mismatch, and the client does not print the message that it completed successfully, it is taken to be a case of "Application Hang". If the comparison by the client of the records in the database and the records that it wrote to the database detects a mismatch, it is taken as a case of "Fail-silence violation". The rationale behind this is that an error in the client process has resulted in the writing of a corrupt record to the database. Since this is a common database, which is shared among all the call processing threads, the writing of a corrupt record to the database can lead to fault propagation.

Category	Notation
Fault Not Activated	*Discarded*
Fault Activated but not Manifested	*NE* (No Error)
PECOS Detection	*PD*
Audit Detection	*AD*
System Detection	*SD*
Client Hang	*CH*
Fail-silence Violation	*FV*

Table 28. Notation for the Results Categories of Fault Injection Runs

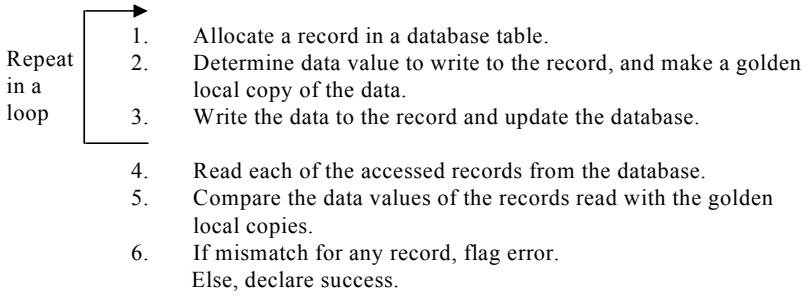


Figure 30. Control Structure of Database Client

8.7 Cumulative Results

This section presents results from error injection experiments conducted to evaluate the join effectiveness of data checking and control flow checking applied to the call processing environment. The results are shown in Table 29 and Table 30, respectively for directed control flow injections and random injections to the instruction stream. In this section, the results from the different fault models have been aggregated. The next section presents the detailed results with a breakdown of the different fault models. The values in the two tables are percentages of runs resulting in each possible outcome listed in the first column, followed by the 95% confidence interval in parentheses. The confidence intervals are calculated using the exact binomial distribution, not the Normal approximation. The last row gives the number of runs used in the error injection campaigns. The number of runs per error injection campaign varies (i.e., is not always 800) because we ignore experiments for which the application does not even start to execute because of an error. Consequently we cannot classify these cases. Note that confidence intervals are not provided for the outcome categories with a small number of observations (less than 10) since they are not considered statistically significant. For these outcome categories, we give the raw number of observed cases.

The key results from the error injection campaigns are:

- Combined use of audits and PECOS is effective in reducing the proportion of system detection (66% to 39% for random injections, and 52% to 19% for directed injections). Recall that for the call processing environment, system detection implies a crash of the call processing application. Preventing the application crash allows for graceful termination of offending thread(s) without affecting other calls being currently processed and thus guarantees high availability of the overall system.

Category	Without PECOS Without Audit	Without PECOS With Audit	With PECOS Without Audit	With PECOS With Audit
Errors Not Activated[1]	58%	53%	50%	50%
Errors Activated but not Manifested[2]	46% (40, 51) [%]	53% (48, 59) [%]	2% (1, 4) [%]	4% (2, 6) [%]
PECOS Detection	N/A	N/A	83% (79, 87) [%]	77% (73, 81) [%]
Audit Detection	N/A	8% (6, 12) [%]	N/A	1
System Detection	52% (47, 58) [%]	37% (32, 42) [%]	14% (11, 18) [%]	19% (15, 23) [%]
Client Hang	6	4	0	0
Fail-silence violation	1	1	1	0
Total Number of Injected Errors	777	738	800	787

Table 29. Cumulative Results from Directed Injection to Control Flow Instructions

(1) The percentage is with respect to the total number of runs

(2) The percentage for this and subsequent categories is with respect to the number of runs where the fault was activated.

Category	Without PECOS Without Audit	Without PECOS With Audit	With PECOS Without Audit	With PECOS With Audit
Errors Not Activated[1]	64%	73%	52%	53%
Errors Activated but not Manifested[2]	28% (23, 34) [%]	26% (20, 32) [%]	12% (9, 15) [%]	7% (5, 10) [%]
PECOS Detection	N/A	N/A	45% (40, 50) [%]	49% (44, 54) [%]
Audit Detection	N/A	7% (4, 12) [%]	N/A	2% (1, 4) [%]
System Detection	66% (60, 71) [%]	61% (54, 68) [%]	41% (36, 46) [%]	39% (34, 44) [%]
Client Hang	4	5	2	2
Fail-silence violation	5% (3, 8) [%]	3% (1, 7) [%]	2% (1, 4) [%]	2% (1, 4) [%]
Total Number of Injected Errors	745	774	800	800

Table 30. Cumulative Results from Random Injection to the Instruction Stream

(1) The percentage is with respect to the total number of runs

(2) The percentage for this and subsequent categories is with respect to the number of runs where the fault was activated.

- Although the number of observed client hangs is small, the results indicate that audits and PECOS are effective in detecting these failures. Usually mechanisms for detecting hangs use a timeout to determine whether a process/thread is still operational or is making progress. Static assumptions of a timeout value do not work well in highly dynamic environments, such as the call processing environment under study. More complex and difficult to implement dynamic schemes capable of on-line adjusting of the timeout value based on current system activity are required. Our techniques allow for detection of client hangs with a low latency and consequently enable fast recovery.
- The proportion of fail-silence violations is quite negligible (one case out of 328 activated errors in the second column in Table 29). This indicates that in a majority of the cases, direct corruption of control flow instructions in the client does not manifest in corruption of the database. This is because of the structure of the client where there is a little scope for an incorrect control flow to cause the client to write

a data different from the local data value, which is the only way a fail-silence violation can occur in the experimental setup.

The situation is different when we perform random injection to the client instruction stream. The results in Table 30 indicate a larger number of fail-silence violations, 13 cases (5%) out of 267 activated errors, (second column in Table 30). Random injections can impact any instruction (not necessarily a control flow instruction) in the instruction stream of the client and there is a higher chance that the client will write incorrect data to the database. The proposed data and control checking mechanisms successfully reduce the fail-silence violations to about 2% (seven cases out of 372 activated errors, last column in Table 30).

Fault Propagation from Client to Database.

The above results allow us to estimate the proportion of error propagation from the client to the database while the client has an error. For injections to the application *without PECOS with audit* (see Table 30), it is seen that for random injections to the text segment of the call processing client, the data audits pick up 7% of errors. Considering the efficiency of audits of detecting actual database errors to be 85% (Table 26), the proportion of the error propagation is 7*(100/85) = 8%. This indicates that error propagation due to client vulnerability can lead to database corruption. Importantly, we have shown above that our detection techniques can significantly improve the fail-silence coverage.

8.8 Detailed Results

This section presents results of the error injection campaigns broken down by the 4 fault models according to which the errors are injected. Table 31 shows the detailed results of all the 32 fault injections campaigns to the call processing clients. The percentage figures are with respect to the total number of activated faults, i.e., the runs in which the fault was not activated have been discarded. The 95% confidence interval for the observations in each error category are given in parentheses. For the cases where there are 5 or fewer observations, the number of observations is not considered statistically significant and therefore, only the number of number of observations in the particular error category are mentioned.

One can conclude from the results that for injections to the call processing application (as opposed to injections to the database itself) that the data audits are not very effective. This is shown by campaign set 4 where both PECOS and audits are present, and PECOS detection completely dominates audit detection. This can be explained by the fact that single bit injections performed in the experiments do not cause enough divergence from expected values to be picked up by the audits. Additionally, since single faults are being injected in a run, the chance of multiple faults accumulating and triggering the audit detection does not exist.

Error Model	Outcome Categories	Injection Target *Any Instruction in Instruction Stream*				Injection Target *Control Flow Instructions*			
		w/o PECOS, w/o Audit	w/o PECOS, with Audit	With PECOS, w/o Audit	with PECOS, With Audit	w/o PECOS, w/o Audit	w/o PECOS, with Audit	With PECOS, w/o Audit	With PECOS, With Audit
ADDIF	NE	15% (10-25%)	18% (10-28%)	10% (5-18%)	4	21% (39-62%)	52% (40-65%)	1	7% (3-14%)
	PD	N/A	N/A	46% (36-56%)	54% (43-64%)	N/A	N/A	85% (76-92%)	75% (65-83%)
	AD	N/A	5	N/A	0	N/A	4	N/A	0
	SD	82% (72-90%)	73% (61-82%)	42% (32-52)%	41% (31-52%)	75% (65-89%)	42% (30-55%)	14% (8-23%)	18% (11-28%)
	CH	1	0	0	0	2	0	0	0
	FV	1	3	2	1	0	0	0	0
DATAIF	NE	20% (11-31%)	24% (11-40%)	7% (2-14%)	5	48% (38-58%)	57% (46-67%)	0	4
	PD	N/A	N/A	36% (26-46%)	32% (23-42%)	N/A	N/A	85% (77-92%)	78% (68-85%)
	AD	N/A	2	N/A	3	N/A	5	N/A	0
	SD	79% (67-88%)	3	56% (45-66%)	56% (46-57%)	52% (42-63%)	34% (25-45%)	14% (8-22%)	18% (11-28%)
	CH	0	0	1	1	0	2	0	0
	FV	1	2	1	2	0	1	1	0
DATAOF	NE	43% (29-58%)	46% (29-65%)	22% (14-31%)	15% (8-23%)	52% (41-63%)	56% (45-67%)	2	2
	PD	N/A	N/A	48% (38-58%)	57% (47-67%)	N/A	N/A	87% (78-93%)	79% (69-86%)
	AD	N/A	1	N/A	4	N/A	15% (8-24%)	N/A	0
	SD	41% (28-56%)	38% (21-56%)	27% (19-37%)	20% (12-29%)	44% (34-55%)	29% (20-39%)	12% (6-20%)	19% (12-29%)
	CH	1	4	0	1	3	0	0	0
	FV	14% (6-26%)	0	3	3	0	0	0	0
DATAInF	NE	39% (27-51%)	28% (17-41%)	8% (3-15%)	4	50% (39-60%)	49% (38-59%)	5	2
	PD	N/A	N/A	50% (39-61%)	53% (42-64%)	N/A	N/A	77% (68-85%)	79% (70-86%)
	AD	N/A	12% (5-23%)	N/A	1	N/A	6% (2-13%)	N/A	1
	SD	53% (41-65%)	55% (42-68%)	41% (31-52%)	39% (29-50%)	49% (38-59%)	43% (34-54%)	18% (11-27%)	18% (11-27%)
	CH	2	1	1	0	1	2	0	0
	FV	4	2	0	2	1	0	0	0

Table 31. Results of Fault Injection into the Call Processing Client

8.9 Combined System Evaluation

In the previous sections, the evaluation results for injections separately into the client and the database have been shown. A reasonable question to ask at this point is overall in the system, if there are both client and database faults, which of the two available techniques (PECOS & Data Audits) will be more valuable. Intuitively, it is obvious the answer will depend on the relative proportion of the faults. Table 32 summarizes the

coverage due to the two techniques for the two individual fault targets (rows 2 & 3), and then uses these measured numbers to estimate the coverage for "joint" faults. The proportion of faults is taken to be 50%-50%. Coverage is given by 100- (System Detection + Fail Silence Violation + Hang)%.

Fault Target	No Audit – no PECOS	Only Data Audits	Only PECOS	PECOS + Data Audits
Client	28.1%	33.2%	56.7%	60.5%
Database	37.2%	84.8%	37.2%	84.8%
Client + Database (50%-50% fault mix)	32.7%	59.0%	46.9%	72.6%

Table 32. System-wide Coverage for Database or Client Errors

For the estimation it is assumed that the fault detection effectiveness of PECOS and audits remains approximately equal with fault rate for the range of operation of the experiment. For the client injections that were performed (Table 31), one fault was injected per run of the client, which equates to one fault every 20-30 seconds. For the database results (Table 26), 1 fault was injected every 20 seconds. For PECOS, since detections of individual faults are independent, it is immaterial to the detection coverage how many instructions are faulty. For data audits, it has been shown that the efficiency reaches a saturation in the operating region used here (Figure 29). From the result we conclude that for database faults, PECOS is ineffective, while for a 50%-50% mix of data and client faults, the data audits still are more valuable. Expectedly for client faults, PECOS is more effective.

9 Conclusion

The thesis presented the design of a hierarchical error detection framework in a SIFT environment. The framework is designed to be able to adapt to changing application needs with respect to the throughput and criticality level desired. A mechanism for quantifying the interactions between levels as well as between techniques within a level is introduced. This is used to enable optimizations in their invocations, which can then optimize the ratio of cost of detection and the coverage afforded, for various different error scenarios. Techniques for incorporating the detection framework in a substantially non-intrusive and incremental manner (through the Monitor Element), and of customizing the detection infrastructure (through the Optimizer Element) are demonstrated.

Several control flow detection techniques for implementing fail-silent processes through pure software means were presented. Experimental evaluations of the techniques were presented running a wide variety of workloads – regular scientific applications (FFT and Radix Sort), a large-scale, control-intensive application (DHCP), and a commercial data-centric application (WCP). Different types of fault injection campaigns were used for the evaluations – random memory bit flips, directed message injections, directed control flow injections, and injections modeling faults in address and data buses. The injections were chosen to model real-world physical faults. Evaluations brought out the improvements in availability that are possible through the proposed detection techniques. For example, the pre-emptive control flow monitoring technique, PECOS, reduced the incidence of fail-silence violations by a factor of 40 and process crashes by a factor of 8 for control flow errors. Further studies are planned to evaluate the relative effectiveness of the techniques when they are all present in a system.

The inter-node levels of the detection hierarchy were simulated. The simulation results provided an idea of the relative overheads associated with each of the detection levels and their impact on error latency and the resultant application availability. The impact of the optimization schemes proposed in the thesis was also demonstrated through the simulation results. For example, optimistic message send with delayed checking outside the critical path reduced the execution time overhead by 5%, and the selective PECOS instrumentation of the DHCP application reduced the overhead from 30% to 13%.

There are several avenues for future work. One is to explore the area of control flow error detection to see if it can be augmented to protect more dynamic control flow structures, such as calls to virtual functions through pointers. It is also obvious through our fault injection studies that some form of data signature is required along with the control flow detection techniques to make the process as a whole fail-silent. Hence, a combined process signature incorporating both data and control is likely to be a valuable tool. A second area that needs to be explored is the customizability of the detection framework. We need to know how an application QoS requirement maps to a particular configuration of the detection framework. We also need to know how to design a manager entity that performs runtime reconfigurations without allowing semantic conflicts between multiple protocols to arise. A broad question that needs to be answered is which model of constructing reliable

distributed systems – group communication, self-checking, or transaction-based – fits which class of applications. We believe that the current work provides the tools with which one can start to explore this question. A necessary precondition for this enterprise is to implement and evaluate the inter-node detection protocols, subjecting them to different kinds of faults, both the ones used here for the intra-node techniques and others, such as Byzantine faults.

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.

[ARL90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, D. Powell, “Experimental Evaluation of the Fault Tolerance of an Atomic Multicast System,” IEEE Transactions on Reliability, Vol. 39, No. 4, pp. 455-467, Oct. 1990.

[AYE98] B. Ayeb, A. Farhat. "The Byzantine Generals Problemn: Identifying the Traitors," Fast-Abs, FTCS-28, 1998.

[BAG00] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, R. K. Iyer, “Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment,” IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No. 2, March/April 2000.

[BIR93] K. P. Birman, "The Process Group Approach to Reliable Distributed Computing," Communications of the ACM, vol.36, No.12, 1993, pp. 37-53.

[BIR94] K. P. Birman, R. van Renesse, "Reliable Distributed Computing with the Isis Toolkit," IEEE Computer Society Press, Los Alamitos, California, 1994.

[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.

[CHA98] S. Chandra, P. M. Chen, “How fail-stop are faulty programs?,” Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (FTCS-28), 1998, pp. 240-249.

[CHA00] S. Chandra, P. M. Chen, “Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software,” International Conf. On Dependable Systems and Networks (DSN-2000), June 2000, pp. 97-106.

[CHI89] R. Chillarege, R. K. Iyer, “An Experimental Study of Memory Fault Latency,” IEEE Transactions on Computers, Volume: 38 Issue: 6, pp. 869-874, June 1989.

[COS00] D. Costa, T. Rilho, H. Madeira, “Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection,” in Proc. Int. Conference on Dependable Systems and Networks, 2000, pp. 251-260.

[DOL96] D. Dolev, D. Malkhi, "The Transis Approach to High Availability Cluster Communication," Communications of the ACM, vol. 39, No. 4, 1996, pp.64-70.

[FUC97] E. Fuchs, “An Evaluation of the Error Detection Mechanisms in MARS Using Software-Implemented Fault Injection,” European Dependable Computing Conference (EDCC) 1996, pp. 73-90.

[GOS97] K. Goswami, R. K. Iyer, “DEPEND: A Simulation Based Environment forSystem Level Dependability Analysis,” IEEE Transactions on Computers, vol. 46, no. 1, 1997, pp.60-74.

[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.

[HEN96] J. L. Hennessy, D.A. Patterson, “Computer Architecture: A Quantitative Approach,”, Second Edition. Morgan Kaufmann, pp.83.

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

[IYE86a] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh, "Measurement and Modeling of Computer Reliability as Affected by System Activity," ACM Transactions on Computer Systems, vol.4, no.3, pp.214-237, August, 1986.

[IYE86b] R. K. Iyer, D. J. Rossetti, "A Measurement-Based Model for Workload Dependence of CPU Errors," IEEE Trans. on Computers, vol.C-35, no.6, June 1986.

[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.

[KAN95] G. Kanawati, N. Kanawati, J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," IEEE Trans. on Computers, February 1995.

[KAN96] G.A. Kanawati, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Evaluation of Integrated System-Level Checks for on-line Error Detection," Proc. IEEE Int'l Symp. Parallel and Distributed Systems, pp. 292-301, Sept. 1996.

[KAR96] J. Karlsson, P. Folkesson, J. Arlat, G. Leber, "Application of three Physical Fault Injection Techniquesto Experimental Assessment of the MARS Architecture," in Proc. Of the 5th IFIP Working Conf. On Dependable Computing for Critical Applications (DCCA-5), pp. 267-287, March 1996.

[KOP90] H. Kopetz, H. Kantz, G. Gruensteidl, P. Puschner, J. Reisinger, "Tolerating Transient Faults in MARS," Proc. 20th International Symposium on Fault Tolerant Computing, pp. 466-473, June 1990.

[LAL85] P. K. Lala, "Fault Tolerant and Fault Testable Hardware Design," Prentice Hall International, New York, 1985.

[LAM82] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem," ACM Trans. on Programming Languages and Systems, vol. 4, no. 3, 1982, pp.382-401.

[LEV99] H. Levendel. Private communication. May 1999.

[LIU00] Y. Liu, "Database Error Detection and Recovery in a Wireless Network Controller," M.S. Thesis (Advisor: R. K. Iyer), Center for Reliable and High-Performance Computing, Univ. of Illinois, December 2000.

[MAD91] H. Madeira, J. G. Silva, "On-Line Signature Learning and Checking," Proc. 2nd IFIP Working Conf, on Dependable Computing for Critical Applications (DCCA-2), pp. 170-177, Feb. 1991.

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

[MAH88] A. Mahmood, E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors – A Survey," IEEE Trans. on Computers, Vol. 37, No. 2, pp. 160-174, Feb. 1988.

[MIC91] T. Michel, R. Leveugle, G. Saucier, "A New Approach to Control Flow Checking without Program Modification," Proc. 21st International Symp. on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[MIR92] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, "Two Software Techniques for On-Line Error Detection," Proc. 22nd International Symp. on Fault-Tolerant Computing (FTCS-22), pp. 328-335, July, 1992.

[MIR95] G. Miremadi, J. Ohlsson, M. Rimen, J. Karlsson, "Use of Time and Address Signatures for Control Flow Checking," Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5), pp. 113-124, 1995.

[MOS96] L. E. Moser., P.M.Melliar-Smith, D.A.Agarwal, R.K.Budhia, C.A.Lingley-Papadopoulos. "Totem: A Fault-Tolerant Multicast Group Communication System," Comm. of the ACM, vol. 39, No. 4, 1996, pp.54-63.

[NAM82] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation," Digest 1982 Intl. Test Conf., pp. 461-468, November 1982.

[NAM83] M. Namjoo, "CEREBRUS-16: An Architecture for a General Purpose Watchdog Processor," Proc. 13th Annual Int'l Symp. On Fault-Tolerant Computing (FTCS-13), pp. 216-219, June 1983.

[OHL92] J. Ohlsson, M. Rimen, U. Gunneflo, "A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog," Proc. 22nd Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-22), pp. 316-325, 1991.

[ORACLE] Oracle8 Server Utilities, Release 8.0, Chapter 10, http://pundit.bus.utexas.edu/oradocs/server803/A54652_01/toc.ht

[POW91] D. Powell, ed., "DELTA-4: A Generic Architecture for Dependable Distributed Systems," Springer-Verlag, October 1991.

[POW94] D. Powell, "Lessons Learned from Delta-4," IEEE Micro, vol.14, No. 4, 1994, pp.36-47.

[PRE92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. O. Flannery, "Numerical Recipes in C: The Art of Scientific Computing," Cambridge University Press, 2nd Edition, 1992.

[REI94] J. Reisinger, A. Steininger, "The Design of a Fail-Silent Processing Node for MARS," Distributed Systems Engineering Journal, 1994.

[REN96] R. van Renesse, K. Birman, and S. Maffeis. Horus: a flexible group communication system. Communications of the ACM, 39(4), pp. 76-83, April 1996.

[RFC97] R. Droms, "Dynamic Host Configuration Protocol," Request for Comments RFC-2131, Bucknell University, March 1997.

[SAB99] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, "Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AquA," Dependable Computing for Critical Applications 7, vol. 12, pp. 149-168, 1999.

[SCH86] M.A. Schuette, J.P. Shen, D.P. Siewiorek, Y.X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," Proc. 16th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-16), pp. 138-143, July 1986.

[SCH87] M.A. Schuette, J.P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," IEEE Transactions on Computers, Vol. C-36, No. 3, pp. 264-276, March 1987.

[SCH96] A. Schiper, M. Raynal, "From Group Communication to Transactions in Distributed Systems," Communications of the ACM, Vol. 39 No. 4, April 1996, pp. 84-87.

[SHE83] J.P. Shen, M.A. Schuette, "On-Line Monitoring Using Signatured Instruction Streams," Proc. 13th International Test Conference, pp. 275-282, Oct. 1983.

[SHI87] K. G. Shin, P. Ramanathan. "Diagnosis of Processors with Byzantine Faults in a Distributed Computing System," FTCS-17, 1987, pp.55-60.

[SHR92] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, A. Tully, "Principal Features of the VOLTAN Family of Reliable Node Architectures for Distributed Systems," IEEE Transactions on Computers, Vol. 41, No. 5, May 1992.

[SIE98] D. P. Siewiorek, R. S. Swarz, "Reliable Computer Systems: Design and Evaluation," A.K. Peters, Natick, Massachusetts, Chapter 2, pp. 22-77.

[STO00] D.T. Stott, B. Floering, Z. Kalbarczyk, R.K. Iyer, "Dependability Assessment in Distributed Systems with Lightweight Fault Injectors in NFTAPE," Proc. IEEE Int'l Computer Performance and Dependability Symp. (IPDS'2K), pp.91-100, March 2000.

[STO00a] D. T. Stott, "Automated Fault-Injection Based Dependability Analysis of Distributed Computer Systems," Ph.D. Thesis, Advisor: R. K. Iyer, Center for Reliable and High-Performance Computing, Univ. of Illinois, December 2000.

[SYBASE] Sybase Adaptive Server System Administration Guide, Chapter 25 http://manuals.sybase.com:80/onlinebooks/group-as/asg1200e/asesag

[THA97] Anshuman Thakur, "Measurement and Analysis of Failures in Computer Systems", Master's Thesis, Advisor: R. K. Iyer, University of Illinois, UILU-ENG-97, September, 1997.

[TIMESTEN] TimesTen 4.0 Datasheet, http://www.timesten.com/products/ttdatasheet.html

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

[UPA94] Shambhu Upadhyaya, Bina Ramamurthy, "Concurrent Process Monitoring with No Reference Signatures," IEEE Transactions on Computers, vol. 43 no. 4, pp. 475-480, April 1994.

[WAL90] C. J. Walter, "Identifying the Cause of Detected Errors," FTCS-20, 1990, pp.48-55.

[WHI98] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, R.K. Iyer, "Incorporating Reconfigurability, Error Detection and Recovery into the Chameleon ARMOR Architecture," Technical Report CRHC-98-13, Coordinated Science Laboratory, University of Illinois, 1998.

[WHI00] K. Whisnant, Z. Kalbarczyk, R.K. Iyer, "Micro-checkpointing: A Checkpointing for Multithreaded Applications," 6th 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.

[YOU91] L.T. Young, R. K. Iyer, "Error Latency Measurements in Symbolic Architectures," Proc. AIAA Computing in Aerospace 8, Baltimore, Maryland, 1991.

[YOU92] L.T. Young, R. K. Iyer, K.K. Goswami, C. Alonso, "A Hybrid Monitor Assissted Fault Injection Environment," Proc. Third IFIP Working Conference on Dependable Computing for Critical Applications (DCCA), Italy, 1992.

[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.

[STO00a] D. T. Stott, "Automated Fault-Injection Based Dependability Analysis of Distributed Computer Systems," Ph.D. Thesis, Advisor: R. K. Iyer, Center for Reliable and High-Performance Computing, Univ. of Illinois, December 2000.

[THA97] Anshuman Thakur, "Measurement and Analysis of Failures in Computer Systems", Master's Thesis, Advisor: R. K. Iyer, University of Illinois, UILU-ENG-97, September, 1997.

[UPA94] Shambhu Upadhyaya, Bina Ramamurthy, "Concurrent Process Monitoring with No Reference Signatures," IEEE Transactions on Computers, vol. 43 no. 4, pp. 475-480, April 1994.

[WHI00] K. Whisnant, Z. Kalbarczyk, R.K. Iyer, "Micro-checkpointing: A Checkpointing for Multithreaded Applications," 6th 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.

Vita

Saurabh Bagchi received his B. Tech. in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur in 1996, and his MS in Computer Science from the University of Illinois at Urbana-Champaign in 1998. His research interests are in software-based fault tolerance and distributed systems. He has published 7 articles in different conferences and journals. He is a student member of the IEEE. He is currently a Research Staff Member at the IBM Thomas J. Watson Research Center at Hawthorne, New York.