

Comparing Fail-Silence Provided by Process Duplication versus Internal Error Detection for DHCP Server

David T. Stott, Neil A. Speirs[†], Zbigniew Kalbarczyk, Josh Scheid, Jun Xu, Ravishankar K. Iyer

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801-2307
E-mail: {dstott,kalbar,scheid,junxu,iyer}@crhc.uiuc.edu

[†]Department of Computing Science
University of Newcastle upon Tyne
United Kingdom, NE1 7RU
E-mail: Neil.Speirs@ncl.ac.uk

Abstract

This paper uses fault injection to compare the ability of two fault-tolerant software architectures to protect an application from faults. These two architectures are Voltan, which uses process replication, and Chameleon ARMORs, which use self-checking. The target application is a Dynamic Host Configuration Protocol (DHCP) server, a widely used utility for managing IP addresses. NFTAPE is used to inject three classes of faults into each software architecture and into baseline Solaris and Linux versions. These campaigns use control-flow faults, high-level target-specific faults, and random memory bit-flips. The analysis found that Voltan provided 100% fail-silence coverage against fault types injected and Chameleon achieved coverage of 99.84%. However Chameleon had a much lower overhead in terms of resources and response times (about 1.7% compared to 76.6% in Voltan).

Keywords: Fault Injection, Dependability, Software-Implemented Fault Tolerance, Fail-Silent, Distributed Computing

1 Introduction

A common assumption made in software-implemented fault tolerance mechanisms, such as message logging, checkpointing, and process replication, is that the processing elements will suffer only crash failures, i.e., a processing element will either perform correct state transitions or it will cease to function and become silent. To meet this assumption in a realistic manner, some form of self-checking facility is required within an element to detect a faulty state transition and stop the element from producing any further outputs. Field studies have also shown that in a distributed environment executing on off-the-shelf hardware components, the fail-silence assumption can be violated [20]. There are two ways of achieving the fail silence property in a distributed environment:

1. make the system component (processing element) self-checking, i.e., the component always produces either the proper output or no output [14].
2. allow the system component to produce a wrong output (fail silence violation) and delegate detection and recovery to a receiving end [5].

In this study we assume the first approach ¹ and we analyze two software-based middleware architectures for providing fail silence: Voltan [5] and Chameleon ARMORs (Adaptive Reconfigurable Mobile Objects for Reliability) [8]. Voltan uses replicated processes and a voting algorithm between processes to provide fail-silence. The Chameleon design supports a range of execution modes including replication (e.g., results from TMR execution in an earlier Chameleon implementation are provided in [8]) and a variety of error detection techniques to provide node and process fail-silence. Chameleon ARMORs possess a set of internal detection mechanisms to achieve the self-checking property without using duplication. This study uses fail-silence as a metric because (1) it is a commonly used metric for distributed systems and (2) both target systems claim to provide fail-silence. The goal of this study is to compare the fail-silence coverage provided by the internal error detection techniques of Chameleon ARMORs with Voltan’s full duplication approach. The target application is Dynamic Host Configuration Protocol (DHCP) [6, 1], a nontrivial application for dynamic allocation of IP addresses to clients in a network.

To analyze the fail-silence of each system, we conduct a set of fault injection campaigns employing NF-TAPE [19], a tool for composing and executing fault injection experiments in a distributed environment. Besides using traditional random memory bit-flips, we also focus on injecting control-flow faults and high-level, target-specific faults. Control-flow faults affect the execution-flow of a process, e.g., a corrupt branch instruction. An example of a high-level, target-specific fault is a corrupt message in a specific message queue.

The experimental results allow us to derive quantitative measures, which characterize:

1. Fail-silence coverage achieved by process duplication (in Voltan) and by embedding the error detection techniques into the application (supported by Chameleon ARMORs).
2. The cost of each approach in terms of performance and resource overhead.

The derived measures and experimental data provide insights into each system and serve as a basis for characterizing the fault behavior of DHCP. As the results are obtained from a single study, caution is necessary in generalizing them to other applications and systems.

The rest of the paper is organized as follows. Section 2 describes related research on fail-silent nodes and their validation. Section 3 provides background information on Voltan, Chameleon, DHCP, and NF-TAPE. Section 4 describes our validation methodology. Section 5 presents the experimental results. Section 6 provides some

¹It can be argued that distributed systems should be designed based on approach '2' (i.e., the receiving end detects an invalid message and prevents the message processing) as opposed to approach '1' (i.e., making the components self-checking). Selecting the right approach is of course a design decision. However, fault-tolerant systems such as Tandem are substantially based on approach (1) ("fail-fast self-checking" components) to achieve fast error detection and rapid recovery. In our experience delegating detection and recovery to a receiving end can contribute to a significant increase in error detection latency and can complicate the recovery process (e.g., an error can impact more than one node).

insight into the dependability attained by Voltan and Chameleon and performance overhead. Section 7 concludes the paper.

2 Related Work

The fail-silence property of several experimental prototypes has been studied, mostly using fault injection campaigns. Pin-level injection used on the Draper Fault Tolerant Multi-Processor [12] found that the system correctly handled all 21,000 faults that were injected. MESSALINE [2] was used to assess the DELTA-4 architecture [14], in particular the self-checking mechanisms of the Network Attachment Controllers intended to support the fail-silent property of nodes in the network. Fault injection results [3] showed that the network controller covered 85-90% of the injected faults and the later study (on the improved network controller) reported the fail-silence coverage of 99% [15].

The MARS architecture [16, 11] uses a combination of special-purpose hardware (e.g., comparators) and software approaches (e.g., double execution) to provide fault tolerance. Karlsson *et al.* [9] used three physical fault injection techniques (heavy-ion radiation, pin-level injection, and electromagnetic interferences) to assess the coverage of error detection mechanisms in this system. They found that the hardware and system software mechanisms provided 97.6% fail-silence coverage. Adding application-level checksums on messages improved the coverage to 98.7%. A software injection study for the same system [7] showed fail-silence coverage of 85% without application-level detection mechanisms, with the coverage becoming perfect when application-specific checks, e.g., data consistency checks were added. A direct comparison of the two studies of MARS system is difficult since they seem to focus on different sets of faults. Nevertheless, they illustrate the fact that it is important to develop a common basis for comparing systems and results from fault injection.

A few studies have compared the dependability of different systems on a common basis. Koopman [10] compared the robustness of 13 different POSIX operating systems by testing their system calls and C library calls. Tsai [21] compared the reliability of three versions of Tandem's TMR-based prototype machines. Madeira *et al.* [13] used physical fault injection to test the built-in detection techniques of two processors: Z80 and MC68K.

No previous study, however, has employed fault injection to compare different software-implemented middlewares for providing fail-silent processes. This requires attempts to establish a sound experimental approach using substantial target applications.

3 System Descriptions

This section first describes the fundamentals of the two target systems. Detailed descriptions of Voltan and Chameleon can be found in [5] and [8], respectively. Next, it describes the target application, the Dynamic Host Configuration Protocol (DHCP).

3.1 Voltan

Voltan assumes that a failed process can exhibit Byzantine behavior but that each nonfaulty process can sign a message it sends by affixing the message with a message-dependent, unforgeable signature. A nonfaulty process is assumed to be able to authenticate any message it receives. The computation performed by a process is assumed to be deterministic. Voltan processes are distributed and communicate via passing messages. So, if nonfaulty replicas have same initial states, then they will produce identical output messages, provided (a) all the nonfaulty replicas of a process receive identical input messages and (b) all the nonfaulty replicas process the messages in an identical order. A fail-silent Voltan process will output either correct messages or detectably incorrect messages.

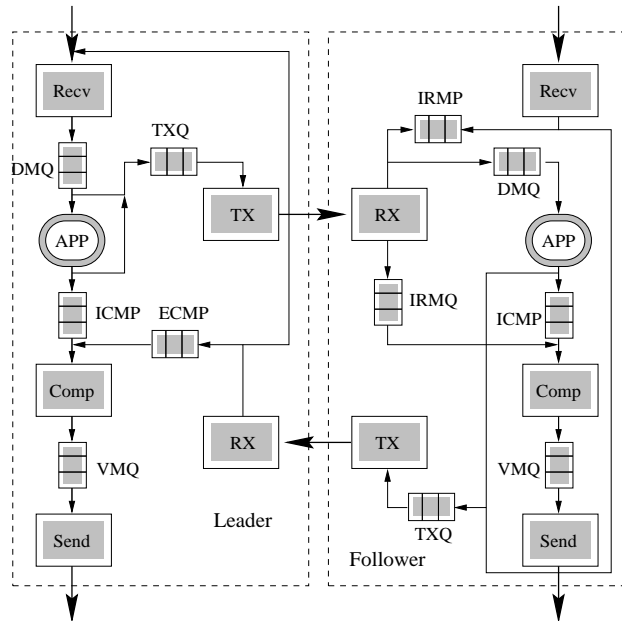


Figure 1. The Structure of a Fail-Silent Process in Voltan

The method of operation for a fail-silent process pair in Voltan is shown in Figure 1. The process labeled Nizam is used to instantiate new Voltan processes; it does not play a role after the application starts running. The ‘Voltan system’ consists of five cooperating threads on each node (Recv., Comp., Send, RX, and TX). The processes in the process pair are called the *Leader* and the *Follower*. The leader’s receiver thread (Recv) accepts authentic, doubly

signed messages and places them into the application's Delivered Message Queue (DMQ) while sending a copy to the follower. The application thread selects a message from DMQ, processes the message, and form an output message. A copy of this message is signed and transmitted to the other replica. The unsigned message is stored locally in the Internal Candidate Message Pool (ICMP). When the Reception thread (RX) receives a singly signed message, it places it in the External Candidate Message Pool (ECMP). The Comparison thread (Comp) compares messages from ICMP and ECMP. If the comparison succeeds, the message from the ECMP is signed again and the doubly signed message is placed into the Voted Message Queue (VMQ); otherwise the replica terminates to prevent any error from propagating. Finally, the Send thread picks up messages from the VMQ and dispatches them to their destinations.

The follower process also contains a mechanism to ensure that if a correctly functioning leader misses receiving a valid message for processing but the follower does receive that message then the message gets ordered and processed by the pair. In summary, Voltan claims to be able to protect a distributed system from any arbitrary failure on a single process.

3.2 Chameleon ARMORs

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

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

1. *Managers*: Manager ARMORs oversee other ARMORs and recover from failures in their subordinates.
2. *Daemons*: Daemon ARMORs are installed on every node participating in Chameleon. A daemon acts as

the gateway of a node for all ARMOR communication and provides error detection for the locally installed ARMORs.

3. *Common ARMORs*: Common ARMORs implement specific techniques for providing application-required dependability.

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

Chameleon provides a four-level hierarchy of error detection techniques for protecting against different error classes [4]. Level 1 consists of detecting errors within the ARMOR to provide error containment within the local node (i.e., without replication). Level 2 consists of detection by the daemon installed on the same node. Levels 3 and 4 (not used in this study) rely on replication of ARMORs to detect errors that escape levels 1 and 2, they provide protection for data and detect Byzantine errors. Table 1 lists the error detection techniques used in this study.

Table 1. Error Detection Mechanisms in Chameleon ARMORs

Detection Mechanism	Level	Description
Coarse-grained I/O Signature	1	Checks pattern of I/O message types against prescribed set of valid sequences
Preemptive Control-Flow Checking	1	Checks against corrupt control-flow branches in runtime
Text-Segment Signature	1	Periodically checks signature of text segment pages
Crash Failure Detection	2	Detects abnormal process termination

Using these error detection mechanisms, Chameleon claims that it will detect the following fault scenarios: (1) any control-flow fault, (2) any single bit-flip and most multi-bit flips to the text segment (program code), (3) any bit-flip to the control part of Chameleon messages, and (4) any thread hung while holding a mutex lock.

3.3 Target Application: DHCP

The target application for this study is version 2.0 of the Dynamic Host Configuration Protocol (DHCP) application written by the Internet Software Consortium. The decision to use the DHCP server was based on the fact that it is:

1. a real-world third party application developed independently of the target systems,
2. a distributed application for which the property of fail-silence is especially important,

3. an open source application so that the source code can be modified to run in either system, and
4. an application that is widely used in real-world environments to provide critical services.

3.3.1 Description of DHCP

Dynamic Host Configuration Protocol (DHCP) [6, 1] is an Internet Draft Standard Protocol for providing configuration information to hosts on an IP network. It is widely used by Internet service providers and network administrators (particularly in dial-up or mobile environments) for network management. A commonly used DHCP service is IP address allocation. It can also be used to discover network parameters such as the Maximum Transmission Unit (MTU) of the network. In networks using DHCP, the availability and correctness of the DHCP server are critical. When joining the network, a client cannot connect to other services (e.g., file server and mail server) until after it obtains an IP address from the DHCP server.

In dynamic allocation, the DHCP server allocates an IP address to the client for a limited amount of time (termed as its lease time), after which the address can be reclaimed by the server. The basic protocol exchanges that take place between the client and the server are as follows. The client broadcasts a DHCPDISCOVER message on its local subnet. DHCP servers on the network respond to the client's request by sending a DHCPOFFER message, which contains a tentative offer of an IP address and appropriate configuration parameters. The client collects the DHCPOFFER responses, chooses one server to interact with further, and broadcasts a DHCPREQUEST message with the identification of the chosen server. 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 containing the configuration parameters for the requesting client. If the selected server is unable to satisfy the DHCPREQUEST message (e.g., the requested network address has been allocated), then the server responds with a DHCPNAK message.

3.3.2 DHCP Modifications to Run in Chameleon and Voltan

To provide services to DHCP, either system, Chameleon or Voltan, requires source-level changes to the original server code. This section describes the modifications we made to DHCP to enable it to execute in the Chameleon and Voltan environments. In each case, fail-silence was provided only to the server and not to the clients.

To receive services from Voltan, applications must use Voltan's communication layer to communicate between the duplicated Voltan processes. We added a pair of simple relay processes to forward messages to and from the DHCP server process pair. In order to communicate with unmodified DHCP clients, a fanin/fanout process (referred to as a splitter) outside of Voltan was also developed to provide a single point of contact for the clients.

The splitter process accepts incoming messages from DHCP clients, resends them to both relay processes within Voltan, and forwards the outgoing messages from relays to clients. Figure 2 illustrates the layout of the processes and the communications that occur in Voltan. Fewer than 1000 lines of code were needed to create the splitter and relay programs and to modify the DHCP server code to be used by Voltan.

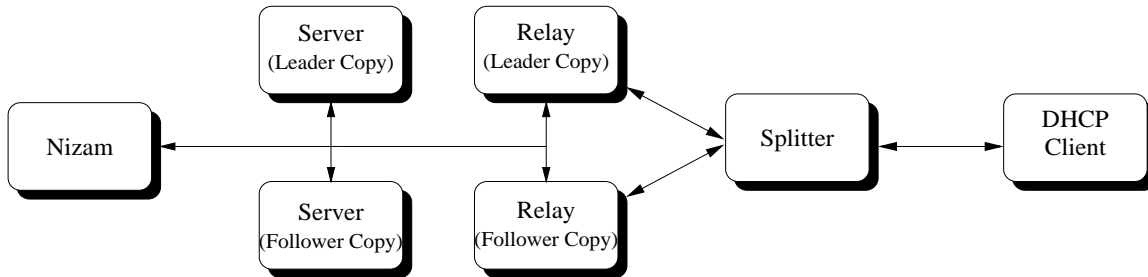


Figure 2. Processes for DHCP Application in Voltan

For Chameleon, the DHCP server is run as an embedded ARMOR, i.e., it runs in the same process as an ARMOR. The server is instrumented with Chameleon API calls, which are processed by a Chameleon stub and which send messages to the ARMOR part of the process. For example, an API call is made by the DHCP server to receive heartbeat query messages and respond to them. Aside from the preemptive control-flow checks (which are automatically added at the assembly code level), about 25 lines of code were added to the DHCP server source code. This configuration is shown in Figure 3.

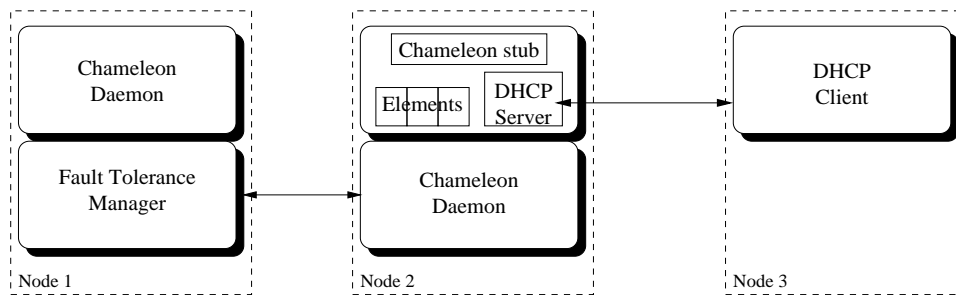


Figure 3. Processes for DHCP Application in Chameleon

4 Dependability Analysis: Objectives, Approach, and Specifics of the Experimental Setup

The objective of this study is to characterize and compare fail-silence guarantees provided to a particular application by Voltan and by Chameleon ARMORs. We use a DHCP server as the target application to obtain insights into the following issues:

1. What fail-silence coverage can be achieved using full duplication of the hardware and the application (supported by Voltan) versus using the error detection techniques embedded into the application (supported by

Chameleon ARMORs)?²

2. What is the cost of each approach in terms of (1) performance overhead and (2) memory overhead?
3. What are the unique application characteristics of DHCP that may be used in selecting an approach to achieving application fail-silence?

We employ software implemented fault injection (SWIFI) to characterize the application behavior under variety of failure scenarios. To perform efficient fault injection experiments, we must determine:

1. *What* faults to inject (our fault model), e.g., control-flow faults,
2. *Which* components to inject faults into, e.g., the DHCP server process,
3. *When* (or under what conditions) to inject faults (our fault trigger), e.g., when the program executes a specific instruction, and
4. *How* to categorize and interpret the results e.g., determine if a run is fail-silent.

Later in this section we discuss the above aspects of our fault injection experiments.

4.1 Fault Model

Transient faults and software bugs are the major causes of system failures. Due to the system and application complexity and the large input space, it is not feasible to study the system under all possible failure scenarios. In any system faults can impact the control flow of the software, can corrupt messages exchanged between communicating processes in a distributed environment, and corrupt data in memory (including stack and heap). We therefore selected several types of transient faults, all of which can be injected using software methods and which correspond to the type of corruption outlined above. Although it is difficult to emulate software bugs (due to lack of a representative fault models), many of software faults manifest in a way similar to transient hardware faults (e.g., use of uninitialized pointers). Therefore, injection of transients can provide useful insight into how good a system is in handling software bugs. The fault models used in this study include:

1. High-level, target-specific faults, e.g., message corruption, message drop, message duplication, and message reordering,
2. Control-flow faults (i.e., targeted memory bit-flips that affect a program's control-flow), and
3. Random memory bit flips into application memory space (text, heap, and stack).

²Note that Chameleon ARMORs support a range of execution modes including replication and a variety of error detection techniques to provide node and process fail-silence. In this study, however, we investigate the fail-silence achieved by using embedded error detection techniques alone.

The selected fault types allow us to provide useful insight into failure behavior of the target application and of the middleware (Voltan and Chameleon ARMORs). It could be argued that we should also inject CPU register faults into the memory space of the operating system. These two fault categories are important and might affect the fail-silence behavior of the target application. Moreover, many of register faults manifest in a way similar to transients injected to the application memory space and therefore, might be redundant to the memory injections. In order to limit the scope of this study, we made a decision that the selected fault classes provide valid basis for a meaningful characterization of fail-silence due to errors that affect the target application and the middleware (i.e., Chameleon and Voltan).

4.1.1 High-Level, Target-Specific Faults

Some important faults that affect the system cannot be easily created with random memory bit-flips. A software bug in a linked list class, for example, may corrupt the list by creating a cycle resulting in repeated scanning of the same entries in the list. For such faults, an alternative to random bit-flips is to use target-specific information (such as the addresses of specific data structures or tables) to guide fault injection. Because the target has knowledge of the data structures, modifying the target program by adding functions to inject such faults at the source-code level is straightforward. The disadvantage of this approach is that the programmer needs to include a way to trigger the fault and log information about the injection parameters. In our experimental setup, this is simplified by using NFTAPE, which provides suitable API calls for the application to incorporate the necessary functionality.

Voltan-Specific Faults: The message queues in Voltan are vulnerable components because they account for a large part of Voltan’s structure and its voting protocol makes heavy use of them. In particular, the DMQ and the VMQ queue (shown in Figure 1) are the most vulnerable, since they send messages directly to the application or to the other node. Thus, a reasonable target-specific fault model is to corrupt a selected message queue by reordering, duplicating, deleting, or corrupting messages in that queue.

To support these fault models, a target-specific fault injector operating in the Voltan environment was created, as shown in Figure 4. The injector requires a few simple modifications to the Voltan application library, such as adding a function to reorder two adjacent messages in a queue (fault functions in Figure 4) and providing a generic facility to open a named pipe (e.g., a UNIX socket) to wait for trigger events from the control part of the fault injection environment (NFTAPE). In addition, an interface process was provided by NFTAPE for receiving and forwarding trigger events from NFTAPE to the fault injection function (or target-specific fault injector) executing in Voltan.

Chameleon-Specific Faults: In Chameleon, the ARMOR processes communicate data and control through mes-

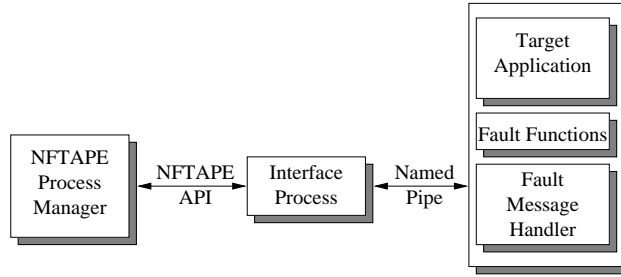


Figure 4. Target-Specific Fault Injector Example for Voltan

sages. Hence, the fail-silence property of the system depends on the integrity of the messages being exchanged. We use message corruption of Chameleon messages just before the messages are sent to the client as fault models for exercising the course-grain control-flow signature check on Chameleon messages. Since this check is only available in the full-fledged ARMOR configuration, the target program we used is slightly different from those used in other Chameleon campaigns.

To support these fault models, a fault injection element, written as an ARMOR, was added to the application. Since the element executes as a thread within the same address space as the target application, it has direct access to the application space, including message buffers.

The fault injection element can be triggered either by an internal timer or by sending a message from an external source on a named pipe using the named pipe interface and the interface program, just like the one for Voltan.

4.1.2 Control-Flow Faults

Control-flow faults cause a divergence of the program counter values of the application from those seen during a fault-free run. For this study, specific types of control-flow faults (e.g., execution of wrong instruction) are emulated by flipping single bits at random in all the control-flow instructions, including various conditional branches, unconditional jumps, subroutine calls, and subroutine returns. The instructions that form the injection target are from the functions in the main protocol engine of the DHCP server. The protocol engine takes the incoming message from the DHCP client as input, does the state transition indicated by the protocol, and does the corresponding processing on the input message to generate the final output message to the client. This is a critical part of the DHCP server, as faults here can cause the server to make incorrect state transitions and send out incorrect messages to the client, or can cause the server to reach a state from which it cannot escape, resulting in a hang. In the Chameleon environment, the functions in the protocol engine are modified to include preemptive control-flow checking to detect the control-flow faults. For consistency, the same set of functions is used in the Voltan case and the baseline cases.

4.1.3 Random Memory Bit-Flips

Random memory bit-flips target a region in the virtual address space Of the target process and randomly select addresses within that region. Because each section has different uses (e.g., executable code is stored in the text segment, program data are stored in the heap segment, and data saved during function calls are stored on the stack), the analysis is separated for each section.

For random memory bit-flips, we use two types of lightweight fault injectors (LWFIs): debugger-based and driver-based. Both LWFIs dynamically find the range of the target memory section, select an address and bit position at random, and inject the fault when triggered.

In the experimental setup, all the fault injections to the Solaris-based platform (including those to Chameleon environment) and all the baseline injections to the Linux platform are performed using debugger-based LWFI A driver-based fault injector is used to insert faults into an application running in Voltan because Voltan failed when running under a debugger. NFTAPE [19], the fault injection environment used to control these LWFIs is described in Section 4.5.

4.2 Target Components

In conducting efficient fault injection experiments, it is critical to decide which components in the target system to inject faults into. In this study, we focus on a DHCP server process, and fault injection campaigns target the memory space of that process. We also injected faults into selected Voltan and Chameleon processes, which impact the fail-silence. We list the key components of the two target systems and discuss a rationale for selecting or not selecting them as fault injection targets:

- **DHCP Server:** The DHCP Server process is the main component that processes the DHCP protocol. Most of the fault injection campaigns are targeted at this process.
- **Voltan Relay and Splitter:** The Relay is a Voltan process responsible for sending data to the outside world. The Voltan Splitter is a process outside Voltan which serves a gateway between the DHCP client and the Voltan Relay. Additional fault injection campaigns target these components.
- **DHCP Client:** The client can be provided by a third party (which will probably not use either middleware). Although client robustness and security can be a concern, they are beyond the scope of this paper. We do not inject faults to the client.
- **Nizam, FTM (fault-tolerance manager), and Daemon:** Voltan’s Nizam and Chameleon’s FTM and Daemon are specific processes that are active only during the initialization and termination of the application. We do not inject faults to these processes.
- **Communications Internal to Middleware:** We do not target communication between the components in each middleware because we assume reliable delivery (from TCP/IP).

- **Communications External to Server:** Though faults to the messages sent between the client and server are likely to produce fail-silence violations, the reliability of IP networks is outside the scope of this paper.

4.3 Fault Trigger

A fault trigger is defined as the criterion for when a fault is to be injected. The campaigns in this study use two types of triggers: breakpoints and timers.

The control-flow fault campaigns set a breakpoint at a target address to determine when to inject each fault. Before each run, a target address is randomly selected as described in Section 4.1.2. The fault injector inserts a breakpoint at the target address when the program begins to execute. The first time the code reaches the breakpoint, the fault injector (1) corrupts the instruction, (2) lets the program single-step over the corrupt instruction, (3) removes the fault so that it will only activate once in the run, and (4) lets the program resume normal execution.

The remaining campaigns (high-level target-specific faults and memory bit-flips) use a timer to trigger faults. The fault rate is set to one fault every two seconds (i.e., up to 15 faults per single run of the experiment). This particular rate was selected to cause a fault activation rate of close to one fault per run. Our previous studies showed that a low fault rate results in a small probability of fault activation and significantly reduces the effectiveness of fault injection experiments (i.e., many fault injection runs do not have any impact on the system behavior). A high fault rate may result in activation of multiple faults during the same experimental run, and as a result, the interpretation of results could be misleading. A detailed analysis of sampled logs from our fault injection experiments confirmed that in all cases the observed application behavior was provoked by a single fault activation.

4.4 Error Detection and Classification

For this study, we define a *run* as a single execution of each of the target processes. The client program runs for 30 seconds unless the fault causes it to terminate early. If no detection occurs during this time, the fault is declared inactive. We then define a *campaign* as a set of runs using the same target system and fault parameters (e.g., control-flow faults to the Chameleon version of DHCP).

Table 2 describes the possible outcomes for each run. If there are multiple outcomes, the more severe outcome is selected. For example, if a run sends incorrect output to the client and then terminates due to a segmentation fault, the run is considered a *Fail-silence Violation* because that is more severe than the abnormal termination of the process due to an error flagged by the operating system. Note that the run will be considered *Correct Behavior* if the server prints an error message but otherwise operates correctly.

In order to determine if the client receives an invalid message, each process's output is stored in a log file and processed offline using an acceptance test. The test reads the logs and flags any message that violates the DHCP

protocol.

Table 2. Possible Outcomes for Each Experimental Run

Outcome	Description
Fail-silence Violation	The client receives an invalid message.
OS Detection	A program terminates abnormally from a signal detected by the operating system (e.g., segmentation error).
Program Aborted	Program detects an error and exits.
Server Hang	The server does not provide its service in a timely fashion.
Middleware Detection	The middleware system detects an error and halts the application.
No Fault Injected	The trigger criterion does not occur (i.e., the address for a control-flow fault is not reached).
Correct Behavior	All programs run correctly.

4.5 NFTAPE

NFTAPE [19], a tool for conducting automated fault injection based experiments, was used to run the campaigns in this study. It provides an API for writing simple fault injectors called lightweight fault injectors (LWFI). LWFIs differ from traditional fault injection tools because they do not need to include code for triggering faults or for logging results, as these services are provided by NFTAPE. Simple triggers wait for events and then use API calls to send a trigger event to a LWFI or to another trigger (to support a cascade of triggers). NFTAPE can compose new fault injection experiments by interchanging LWFIs and triggers that use the NFTAPE API.

NFTAPE contains two main components: the Control Host and the Target Nodes. The Control Host, which generally resides on a safe node, processes a file called a Campaign Script. This file provides information about the global sequencing of events in a campaign, such as, what processes need to be executed, what parameters they take, and when to run or terminate the processes. These processes can be target applications, monitors, triggers, fault injectors, acceptance tests, or other processes on the node. The Control Host also logs the results of the experiments for off-line analysis.

Four important features of NFTAPE facilitated these experiments:

- **LWFI API:** New fault injectors, such as the target-specific fault injectors, are easy to write.
- **Process manager:** NFTAPE can start all the processes used in the experiment and clean them up if the program terminates abnormally.
- **Logging:** The outputs from each process, including fault injector, trigger, and application, are logged using the same format.
- **Automated experiment sequencing:** The campaign script sets the order of events, such as the order in which processes run.

5 Experiments

This study uses several fault injection experiments to answer the questions posed in Section 4 as they apply to the DHCP application. These questions include: (a) What is the fail-silence coverage achieved by process duplication and by self-checking ARMORs? (b) What is the cost of each approach? and (c) What are other characteristics particular to each system that may be important in deciding which one to apply in practice? This section presents results from experiments and comments on individual fault injection campaigns. Discussion of how the results answer the above questions is saved for Section 6.

5.1 Baseline Experiments

The first set of experiments use the original, unmodified version of the DHCP software executing on the Linux and Solaris platforms. These experiments form a baseline case to measure improvement in the fail-silence coverage achieved by process duplication versus local detection alone. The results from these experiments are given in Tables 3 and 4.

Table 3. Results from Fault Injection to Baseline Linux DHCP Server

Outcome	Random memory bit-flips						Control flow[%]	
	Heap[%]		Stack[%]		Text[%]			
Fail-Silence Violation	0.2	(0.005, 1.1)	0.0	(0.0, 0.7)	2.6	(1.4, 4.4)	1.1	(0.6, 2.0)
OS Detection	13.3	(10.5, 16.7)	9.2	(6.8, 12.1)	63.0	(58.2, 66.9)	19.0	(16.6, 21.6)
Program Aborted	0.0	(0.0, 0.7)	4.8	(3.1, 7.1)	1.4	(0.6, 2.9)	0.0	(0.6, 2.9)
Server Hang	0.2	(0.005, 1.1)	0.0	(0.0, 0.7)	0.0	(0.0, 0.7)	0.1	(0.003, 0.6)
Middleware Detection	n.a.		n.a.		n.a.		n.a.	
Correct Behavior	86.3	(82.9, 89.1)	86.0	(82.6, 88.9)	33.0	(29.3, 37.7)	26.5	(23.8, 29.4)
No Fault Injected	n.a.		n.a.		n.a.		53.3	(50.2, 56.4)
Number of Runs	500		500		500		1000	

Table 4. Results from Fault Injection to Baseline Solaris DHCP Server

Outcome	Random memory bit-flips						Control flow[%]	
	Heap[%]		Stack[%]		Text[%]			
Fail-Silence Violation	0.6	(0.1, 1.7)	0.2	(0.005, 1.1)	4.4	(2.8, 6.6)	1.4	(0.6, 2.9)
OS Detection	29.4	(25.4, 33.6)	18.2	(14.9, 21.9)	47.0	(42.6, 51.5)	29.0	(25.1, 33.2)
Program Aborted	0.0	(0, 0.7)	1.8	(0.8, 3.4)	1.0	(0.3, 2.3)	0.4	(0.05, 1.4)
Server Hang	0.2	(0.005, 1.1)	0.2	(0.005, 1.1)	0.8	(0.2, 2.0)	0.2	(0.005, 1.1)
Middleware Detection	n.a.		n.a.		n.a.		n.a.	
Correct Behavior	69.8	(65.6, 73.8)	79.6	(75.8, 83.0)	46.8	(42.4, 51.3)	13.6	(10.7, 16.9)
No Fault Injected	n.a.		n.a.		n.a.		55.4	(50.9, 59.8)
Number of Runs	500		500		500		500	

Each column in these tables represents one campaign. The values in the column are the percentage of runs resulting in each possible outcome (listed in the first column) followed by 95% confidence interval on the percent-

age of runs (in parentheses). For example, in Table 3, 2.6% of the runs (i.e., 13 of 500 runs) in the fault injection campaign that targeted the application text-segment caused fail-silence violations, and the 95% confidence interval is (1.4%, 4.4%). The confidence intervals are calculated assuming a binomial distribution. The last row gives the number of runs used in the campaign³. For the baseline experiments, there are no explicit error detection mechanisms, and consequently the Middleware Detection category is always zero.

The results indicate that a significant percentage of faults (which manifested as errors) are detected by the operating system (e.g., 19.0% and 18.3% of control flow errors are detected by Linux and Solaris OSs, respectively). Random injections to memory show that the target application is particularly sensitive to the text segment injections, where the largest percentage of fail-silence violation is observed (2.6% and 4.4% for Linux and Solaris platforms, respectively). The DHCP executing on Linux platform seems to be more fail-silent than when it runs on Solaris platform. It is difficult to say whether this is due to intrinsic differences between the two operating systems or to differences in the Pentium and Sparc architectures.

5.2 Voltan Experiments

Fault injection experiments in Voltan target the server, the relay, and the splitter processes (see Tables 5-7 respectively).⁴ Previous experiments to determine how differently the Leader and Follower processes tolerate faults showed that Voltan was able to mask certain faults to the Follower that it could not mask for the Leader, but otherwise they behaved equally [18]. Based on this result, we injected faults only to the Leader node.

We observe that the primary mode of error detection remains operating system detection (i.e., application crash), e.g., the operating system detects 63.8%, 36.0%, and 54.0% faults injected to the text segment of the server, the relay, and the splitter, respectively. The results from the relay process are similar to those of the server process. In neither case do we observe fail-silence violation. The major difference in the results for the DHCP server and the relay is in the number of faults that lead to application misbehavior. This percentage is significantly lower for the relay process. This result is expected because the relay is much less complex than the server, performs less work, and maintains very little state information.

Table 6 is included to show that faults to the relay process behave similarly to those injected to the server

³In each of the cases we performed the fault injection experiments until the distribution of outcomes stabilized, e.g., 4 (and the associated 95% confidence intervals for individual outcomes were within acceptable limits). Observe that the ratio between the percentages corresponding to individual outcomes within a given fault injection campaign is large enough for us to identify major failure categories and their significance for the system operation. In this context, 500 runs per fault injection campaign was sufficient to draw meaningful conclusions.

⁴In the current implementation of Voltan, faults were injected using a driver-based fault injector, as Voltan currently does not run properly under a debugger (consequently we could not use the debugger-based fault injector). The driver-based fault injector cannot determine precisely whether a specific fault was exercised, as the breakpoint trigger does.

process. Because the relay process is much simpler than the server, the fault activation rate is lower for the relay process.

Table 5. Results from Fault Injection to Voltan DHCP Server

Outcome	Random memory bit-flips						Control flow[%]	Message faults[%]
	Heap[%]		Stack[%]		Text[%]			
Fail-Silence Violation	0.0	(0.0, 0.7)	0.0	(0.0, 0.7)	0.0	(0.0, 0.7)	0.0	(0.0, 0.7)
OS Detection	19.2	(15.8, 22.9)	6.2	(4.2, 8.7)	63.8	(59.4, 68.0)	25.2	(21.5, 29.2)
Program Aborted	0.0	(0.0, 0.7)	0.0	(0.0, 0.7)	0.0	(0.0, 0.7)	0.0	(0.0, 0.7)
Server Hang	6.0	(4.1, 8.4)	7.2	(5.1, 9.8)	6.4	(4.4, 8.9)	4.4	(2.8, 6.6)
Middleware Detection	1.0	(0.3, 2.3)	0.2	(0.005, 1.1)	3.0	(1.7, 4.9)	3.8	(2.3, 5.9)
Correct Behavior	73.8	(69.7, 77.6)	86.4	(83.1, 89.3)	26.8	(23.0, 30.9)	66.6	(62.3, 70.7)
No Fault Injected	n.a.		n.a.		n.a.		n.a.	
Number of Runs	500		500		500		500	

Table 6. Results from Fault Injection to Voltan DHCP Relay Process

Outcome	Random memory bit-flips					
	Heap[%]		Stack[%]		Text[%]	
Fail-Silence Violation	0.0	(0.0, 1.8)	0.0	(0.0, 1.8)	0.0	(0.0, 1.8)
OS Detection	10.5	(6.6, 15.6)	7.5	(4.3, 12.1)	36.0	(29.4, 43.1)
Program Aborted	0.0	(0.0, 1.8)	0.0	(0.0, 1.8)	0.0	(0.0, 1.8)
Server Hang	1.5	(0.3, 4.3)	0.0	(0.0, 1.8)	4.0	(1.7, 7.7)
Middleware Detection	0.0	(0.0, 1.8)	0.0	(0.0, 1.8)	0.0	(0.0, 1.8)
Correct Behavior	88.0	(82.7, 92.2)	92.5	(87.9, 95.7)	60.0	(52.9, 66.8)
No Fault Injected	n.a.		n.a.		n.a.	
Number of Runs	200		200		200	

Table 7 shows that fault injections to the splitter lead to fail-silence violations. Although the splitter is technically outside Voltan (and as such is not duplicated), it is essential (since it provides a voting gateway to the client) for running Voltan in the mode in which a server is duplicated but not the client. The results illustrate that this single process may become a fail-silence bottleneck and, regardless of its simplicity, can impact the fail-silence guarantees provided by the overall system. In other words, care must be taken to ensure that this single entity does not compromise the system.

5.3 Chameleon ARMOR Experiments

The results from fault injection experiments performed with Chameleon are shown in Table 8.⁵ We observe a significant reduction in the percentage of error detections by the operating system (i.e., reduction of application

⁵The number of runs per fault injection campaign (given in the last row of the Table 8) varies because we ignore experiments for which the application, because of a fault, does not even start to execute, and consequently we cannot classify these cases as operating system detection or program abort.

Table 7. Results from Fault Injection to Voltan DHCP Splitter Process

Outcome	Random memory bit-flips					
	Heap[%]		Stack[%]		Text[%]	
Fail-Silence Violation	0.0	(0.0, 1.4)	0.4	(0.1, 2.2)	0.8	(0.1, 2.8)
OS Detection	0.0	(0.0, 1.4)	1.2	(0.3, 3.5)	54.0	(48.0, 60.3)
Program Aborted	0.0	(0.0, 1.4)	0.8	(0.1, 2.8)	22.0	(17.0, 27.7)
Server Hang	5.2	(2.8, 8.7)	0.0	(0.0, 1.4)	21.6	(16.7, 27.2)
Middleware Detection	n.a.		n.a.		n.a.	
Correct Behavior	94.8	(91.3, 97.2)	97.6	(94.8, 99.1)	1.6	(0.4, 4.0)
No Fault Injected	n.a.		n.a.		n.a.	
Number of Runs	250		250		250	

crashes). For example, for control-flow errors, the percentage of OS detection is reduced from 29.0% (for the DHCP baseline) to 4.6% when running in Chameleon. The results indicate that Chameleon middleware provides high coverage and detects a significant percentage of errors. The text segment signature checker detects text segment faults in almost every run, including runs which would have completed without error otherwise. It also eliminates the fail-silence violations caused by faults in the text segment.

For the control-flow fault injection campaign, the text segment signature is disabled. In this case, we observe that 36.0% of faults injected into the application text segment are detected by preemptive control-flow checking. More importantly, fail-silence violation cases are fully eliminated when running in Chameleon with enabled preemptive control flow checking.

In random injection to the heap and stack, we still have a significant percentage of faults that do not lead to application misbehavior. This is because the heap memory allocated in the Chameleon version of DHCP has a size of 866KB, only a small percentage of which is actively used by the application (in the baseline case the heap had a size of only 84KB). Consequently most faults injected into the heap affected the Chameleon library (and possibly affected elements that were unused) rather than the application part of the embedded ARMOR process; this results in a low fault activation rate. To compare, the heap memory allocations for standalone DHCP running on Linux and for DHCP executing in Voltan were 56KB and 100KB, respectively.

Table 8. Results from Fault Injection to Chameleon DHCP Server

Outcome	Random memory bit-flips						Control flow[%]	Message faults[%]		
	Heap[%]		Stack[%]		Text[%]					
Fail-Silence Violation	0.0	(0.0, 0.8)	0.6	(0.1, 1.8)	0.0	(0.0, 0.8)	0.0	(0.0, 0.6)	18.4	(15.1, 22.1)
OS Detection	8.8	(6.4, 11.8)	9.4	(6.9, 12.3)	1.3	(0.5, 2.8)	4.6	(2.9, 6.8)	0.0	(0.0, 0.7)
Program Aborted	0.2	(0.006, 1.2)	0.0	(0.0, 0.8)	0.0	(0.0, 0.8)	0.0	(0.0, 0.6)	0.0	(0.0, 0.7)
Server Hang	0.0	(0.0, 0.8)	0.2	(0.005, 1.2)	0.0	(0.0, 0.8)	0.2	(0.005, 1.1)	0.0	(0.0, 0.7)
Middleware Detection	0.0	(0.0, 0.8)	0.0	(0.0, 0.8)	98.7	(97.2, 99.5)	36.0	(31.8, 40.4)	7.2	(5.1, 9.8)
Correct Behavior	91.0	(88.0, 93.4)	89.8	(86.7, 92.4)	0.0	(0.0, 0.8)	0.8	(0.2, 2.0)	74.4	(70.3, 78.2)
No Fault Injected	n.a.		n.a.		n.a.		58.4	(53.9, 62.8)	n.a.	
Number of Runs	464		471		470		500		500	

The purpose of the high-level, target-specific message faults is to exercise the coarse-grained signature in the full-fledge ARMOR version of DHCP application. Faults are injected into the first 64 bytes of the message header. In this setup, the coarse-grained signature checks for correct message types. The message type constitutes 4 out of the 64 bytes in the message header. As a result, a significant percentage of errors escaped; 18.4% of the errors caused a fail-silence violation (see Table 8). It is worth noting that designing efficient mechanisms for detecting data errors (i.e., data resulting from a computation and exchanged among processes) requires taking into account specifics of the application. Consequently it is difficult to generalize these error detection techniques across different applications and systems.

5.4 Performance Evaluation

An important consideration is how much overhead is being paid to achieve fault tolerance. For computationally intensive applications, overheads are generally measured by comparing the time it takes the program to complete with and without the fault tolerance techniques. With server-based applications, overhead can be measured as the amount of time it takes the server to complete a request or as the maximum rate transaction completion rate (transactions per time). For this study, we measured the time it takes the server to complete a transaction from the client’s perspective, i.e., the time between the client’s request and the client’s receipt of a response.

The client in this case makes two types of requests. The first request, DHCPDISCOVER, is used to locate the server when the client first starts up. The second request, DHCPREQUEST, is used to request or renew an IP address. The overhead that we measured includes the time the request packet is on the network, the time the server spends processing the request, and the time OS spends before scheduling the processes. For the DHCPREQUEST packet, the server also needs to write the lease file to disk.

The performance overhead is calculated as follows:

$$\left\{ PerfOverhead = \frac{\text{average time to complete a client request using the middleware (Voltan or Chameleon)}}{\text{average time to complete a client request in the baseline case}} \right\}$$

Table 9 shows the results of measurements taken on each target system. For each message type and each target middleware, the table gives the mean and standard deviation (in milliseconds) of the measured response times over more than 100 messages (user requests to DHCP server). The results show that the Chameleon overhead is very small, while the Voltan overhead is close to 80%. A major contributor to the overhead in Voltan is the time needed to synchronize processes.

Table 9 also shows significant difference in the execution time of the two protocol phases—DHCPDISCOVER (4.77ms) and DHCPREQUEST (52.4ms; involves writing to disk)—on the Solaris platform. On the Linux platform, these two protocol phases require approximately the same amount of time to complete. The observed performance

differences are due to the different behavior of the Solaris and Linux operating systems when accessing the NFS. When a program writes to NFS in Solaris, the write call blocks until the data has been saved to disk, but Linux allows the write to return immediately.

Table 9. Round-trip Transaction Times (in milliseconds)

	Baseline Linux	Voltan	Baseline Solaris	Chameleon
DHCPDISCOVER-Message Phase [ms]				
Mean Time	29.5	52.9	4.77	4.84
Stand. Dev.	6.89	12.9	3.51	2.90
Overhead	79.3%		1.4%	
DHCPCREQUEST-Message Phase [ms]				
Mean Time	28.5	49.5	52.4	53.3
Stand. Dev.	6.87	7.53	13.7	9.04
Overhead	73.7%		1.7%	

6 Discussion

In this section, we discuss the lessons learned from this study. The experiments provide useful insights into both the fail-silent behavior of the application and its resource usage.

6.1 Claims of Fail-Silence

Voltan case. One of the goals of this study was to assess how well each of the target systems defends its claims of fail-silence. Voltan claims to provide fail-silence when a single node experiences any arbitrary failure mode. Voltan succeeded in this objective for all 2600 runs (Tables 5 and 6). The only exception is the splitter process, which although simple and highly reliable can be a single point of failure (as shown in Table 7). In about 4.0% to 7.0% of the runs, the server hung (see Table 5). This means that although the fail silence is not violated, Voltan processes must be restarted to enable further services from the DHCP server, which in turn will impact the availability metric of the target application.

Chameleon case. Chameleon succeeded in providing fail silence to the application in 99.84% of 1905 runs (Table 8). With respect to the specific techniques used:

The text segment signature detection was shown to be very affective at detecting faults. In 98.7% of the runs with text segment faults, the signature detected a fault. In the remaining 1.3%, the system exited on an illegal memory access (bus error).

For control-flow faults, Chameleon was able to detect 86.5% of the activated faults (column 5 in Table 8)⁶. The remaining 13.5% of control-flow faults crash the server process (OS Detection), hang the server process (Server Hang), or do not impact the application (Correct Behavior). Observe that in all these cases the fail silence of the application is not violated.

The percentage of runs that completed successfully (Correct Behavior) without a detected error was reduced from 13.6% to 0.8% (Tables 4 and 8). This could suggest that the assertion halts programs which probably would complete correctly. It is also possible that the injected fault remains dormant for a time longer than a single run of the experiment and can impact the application in a long run. In this scenario preemptive control-flow checking reduces error detection latency and prevents possible error propagation in the future. Comparing the control-flow fault injection campaign for Chameleon with the one for baseline Solaris, the cases of fail-silence violation are eliminated and the number of uncaught operating system signals is reduced by a factor of 6 (from 29% for the baseline case to 4.6% in Chameleon; Tables 4 and 8).

6.2 Summary of Comparison between Voltan and Chameleon

In Table 10 the two target middleware systems are compared by assessing the fail-silence coverage, performance overhead, and resource overhead. Control-flow faults and the random memory faults injections to the DHCP server constitute a basis for the comparison. The DHCP server process is a common entity present in all studied configurations (baseline cases, Voltan, and Chameleon). Also control-flow faults and random memory faults are injected to study each of the DHCP configurations. In other words the extent to which faults directly impact the DHCP server process creates the basis for our comparison⁷. In our analysis the fail-silence coverage (given in Table 10) is calculated as follows: $\left\{ 1 - \frac{\text{number of fail-silence violation runs}}{\text{total number of runs}} \right\}$.

The overhead in Chameleon includes the time to execute the control-flow assertion blocks. The overhead in Voltan appears to be the time to synchronize the Voltan processes and to exchanged messages between them. Voltan requires several logical processes—each process needs to be duplicated doubling the aggregate CPU load. To compare the resource overhead of each system, we look at the number of processes needed and the factor of utilization overhead due to replication. These are given as *Number of Processes* and *Utilization Overhead* in Table 10.

⁶Recall that not activated fault (“No Fault Injected” outcome category) means that the fault is injected however the part of the code containing the corrupted instruction is not exercised during the execution.

⁷Faults injected to messages are not considered since they use different fault models for each system. Message faults could be a cause for concern in the case of Chameleon as we observe fail silence violations in 18% of the runs. However, this is not the deficiency of the design, rather inefficiency of an early implementation of the Chameleon (e.g., at the time of experiments Chameleon could not handle duplicate messages). More importantly even if we include the escapes, the fail silence coverage (over all runs) is reduced slightly (to 97%). This should not be compared with the baseline case, because the baseline case does not include fault injection to messages.

Table 10. Summary of Comparisons

	Voltan		Chameleon	
Fail-silence Coverage				
Random memory bit-flips	100%		99.79%	
Control-flow faults	100%		100%	
Weighted Average	100%		99.84%	
Average Performance Overhead (Table 9)				
Performance Overhead	76.6%		1.70%	
Memory Considerations				
	Voltan	Baseline Linux	Chameleon	Baseline Solaris
Text size (KB)	2484	319	1791	1018
Text overhead	7.79		1.76	
Heap size (KB)	100	56	866	84
Heap overhead	1.79		10.3	
CPU Resources				
Number of Processes	6	1	4	1
Utilization Overhead	2		1	

6.3 Detection Latency

An important metric is the detection latency for each system. For Chameleon, the control-flow faults were detected within a few cycles of when they were injected, and the random text segment faults may have been latent for up to 5 seconds before being detected (the check executes every 5 seconds). For Voltan, faults were only detected when the server sent a message and after detecting a potential fault; the voter waits indefinitely for the faulty node to send a correct message. Though the logs NFTAPE produces include the times the fault was injected and when it was detected, these results are not presented in this paper due to space limitations.

6.4 Comparison of DHCP Application and Scientific Applications

The experiments shed light on specific areas of the target application’s behavior. The results show that the fault activation rate for the DHCP server is much lower than that of scientific applications or simple algorithms, such as quick sort, used in earlier studies [17, 9].

To better understand how generalizable our observations were, we performed similar study using *scientific applications* - radix sort (a linear time integer sorting algorithm) and FFT (Fast Fourier Transform) - as fault injection targets. Due to space limitations, Table 11 outlines the key findings. From injecting random memory faults and target specific faults we observe an average 99.2% and 90.2% fail silence coverage for Voltan and Chameleon, respectively (rows 3 and 4 in Table 11). The immediate conclusion is that the type of workload impacts the measured fail silence coverage. Radix sort and FFT applications are more computation intensive than DHCP. Both applications actively use the heap memory to store large data sets and consequently there is

a high chance that any fault to the heap will affect the results produced by the applications. Chameleon does not provide protection to data stored in dynamically allocated memory (i.e., the heap) and consequently the fail silence coverage drops (row 3, columns 4 and 5 in Table 11) compared with DHCP (where Chameleon self-checking mechanisms achieved 99.84% fail silence coverage). Also, Voltan does not protect all data residing on the heap, however, due to the nature of the duplication, Voltan still protects well data exchanged between the nodes (the Leader and the Follower). Consequently the fail silence coverage does not drop that significantly compared with DHCP (where full duplication in Voltan covered all injected faults).

Table 11. Summary of Results from Fault Injection to FFT and radix sort

	Fail-silence Coverage			
	Voltan		Chameleon	
	FFT	rsort	FFT	rsort
Random memory bit-flips (stack, heap, text)	97.6%	98.4%	78.9%	83.3%
Target-specific faults (e.g., message corruption, message duplication)	100%	100%	98.9%	100%

Our experiments with *DHCP application* show that a small percentage of the heap and stack data is critical for service-based applications such as DHCP or that a small percentage of the allocated space is actually used. In DHCP application, the heap allocates space for several tables where only a few entries are in use. For example, one of the tables allocates entries to hold data for each subnet the server manages, but only one subnet was in use in our experimental setup. Consequently, only a portion of this table was accessed.

Likewise, the stack also appears to be less sensitive to faults in service-based applications than in scientific applications. This is for temporal reasons. In a service-based application (such as DHCP), the program spends most of its time waiting for an event. While the program is waiting (i.e., in the `dispatch()` function in the DHCP case), the stack contains variables and registers from functions such as `main()` that called `dispatch()`. The data from these functions are dead (i.e., they will never be used again) because `dispatch()` does not return. The activation rate would increase by using a breakpoint to trigger a stack fault while the server is processing a request or by targeting the data at the bottom of the stack, which is more likely to be used.

6.5 Issues for Comparison Studies

As one of the first studies to compare two software-based approaches to providing fail-silence guarantees to applications, this study uncovered some important issues for future studies.

- Although we tried to inject the same faults into each version/configuration of the target application, subtle differences between the versions/configurations made it difficult to guarantee inserting of the same faults.

- A typical assumption that well understood fault model, such as random bit-flip to memory, would be comparable between the versions of the target application is not entirely true. Memory regions have different sizes in each version. For example, the size of the heap region was 56KB for Linux, 84KB for Solaris, 100KB for Voltan, and 866KB for Chameleon. To compensate for these differences, it may make sense to adjust the fault rate to hold the fault rate per bit of memory constant.
- The DHCP application executes as a single process/thread in the baseline cases and as a multithreaded process in Voltan and in Chameleon. Multithreaded programs use several stacks within the same memory region. Consequently, it is difficult to draw direct comparisons in the stack fault injection campaigns.

We overcome some of these uncertainties by using a more severe fault model than the one used in most other studies, i.e., we include faults the target systems were not explicitly designed to tolerate. Our results agree with those reported, for example, in [17]), given the differences between the applications noted in Section 6.4. We believe that uncertainties in our experiments towards the conservative side, i.e., we do not overestimate the fail-silence property of the systems.

7 Conclusion

In this paper, we study two software-based approaches to providing fail silence to applications: (1) process duplication, as supported by Voltan, and (2) internal error detection, as provided by Chameleon ARMORs. The key objective is to assess the coverage and the cost (in terms of performance and memory overhead) of the two approaches. As a target for this study, we use Dynamic Host Configuration Protocol (DHCP) application. A set of fault injection experiments (using NFTAPE) is performed to characterize fail-silent behavior of the application. Several conclusions can be drawn from this study:

1. Process duplication can provide very high fail silence coverage (in our experiments Voltan covered all injected faults to the Voltan processes) with the price of significant performance (76.6% in Voltan) and resource (e.g., memory, the number of needed processes) overhead.
2. Software-based self-checking mechanisms offer high (but not perfect) fail silence coverage, (Chameleon ARMORs achieved 99.84% fail silence coverage) with relatively small performance (1.7% using Chameleon ARMORs) and resource overhead. It should be noted that these overheads depend on the application type (service-based or control-centric) used in this study and may differ for more CPU-intensive applications.
3. The major reason for imperfect coverage provided by ARMORs is a lack of efficient mechanisms for detecting data errors (i.e., data resulting from a computation and exchanged among processes). Designing such mechanisms requires taking into account the specifics of the application. Consequently, it is difficult to generalize these error detection techniques across different applications and systems. For example, ARMORs support coarse-grained signature checks for a message type, which constitutes about 6% of the message header; the remaining portion of the header is not protected.
4. A user perspective on the services provided by the application (in terms of requirements on service availability, timeliness, security, and performance) should be considered as a criterion for selecting an appropriate

approach to achieving fail silence. In making this decision, it is critical to understand application failure modes and failure frequencies and to analyze whether, for example, self-checking alone can provide sufficiently high fail-silence coverage and at what cost (in terms of performance and resource overhead).

References

- [1] S. Alexander and R. Droms, "DHCP options and BOOTP vendor extensions," Request for Comment RFC 2132, Silicon Graphics, Inc., Mar. 1997.
- [2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 166–182, 1990.
- [3] J. Arlat, M. Aguera, Y. Crouzet, J.-C. Fabre, E. Martins, and D. Powell, "Experimental evaluation of the fault tolerance of an atomic multicast system," *IEEE Trans. Reliability*, vol. 39, no. 4, pp. 455–467, 1990.
- [4] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, "Hierarchical error detection in a software implemented fault tolerance (SIFT) environment," *To Appear in IEEE Trans. on Knowledge and Data Eng.*, Apr. 2000.
- [5] F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao, "Implementing fail-silent nodes for distributed systems," *IEEE Trans. Computers*, vol. 45, pp. 1226–1238, Nov. 1996.
- [6] R. Droms, "Dynamic host configuration protocol," Request for Comment RFC 2131, Bucknell University, Mar. 1997.
- [7] E. Fuchs, "Validating the fail-silence of the MARS architecture," in *Proc. of the 6th IFIP Int'l Working Conf. Dependable Computing for Critical Applications (DCCA-6)*, Mar. 1997.
- [8] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Trans. Parallel and Distributed Systems*, June 1999.
- [9] J. Karlsson, P. Folkesson, J. Arlat, and G. Leber, "Application of three physical fault injection techniques to experimental assessment of the MARS architecture," in *Proc. of the 5th IFIP Int'l Working Conf. Dependable Computing for Critical Applications (DCCA-5)*, pp. 267–287, Mar. 1996.
- [10] P. Koopman and J. DeVale, "Comparing the robustness of posix operating systems," in *Proc. of the 29th Int'l Symp. on Fault-Tolerant Computing (FTCS-29)*, pp. 30–39, June 1999.
- [11] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating transient faults in MARS," in *Proc. of the 20th Int'l Symp. on Fault-Tolerant Computing (FTCS-20)*, pp. 466–473, June 1990.
- [12] J. H. Lala, "Fault detection, isolation, and reconfiguration in FTMP: Methods and experimental results," in *Proc. 5th AIAA/IEEE Digital Avionics Systems Conf. (DASC)*, pp. 130–146, 1983.
- [13] H. Madeira and J. G. Silva, "Experimental evaluation of the fail-silent behavior in computers without error masking," in *Proc. of the 24th Int'l Symp. on Fault-Tolerant Computing (FTCS-24)*, pp. 350–359, June 1994.
- [14] D. Powell, ed., *DELTA-4: A Generic Architecture for Dependable Distributed Systems*. Springer-Verlag, Oct. 1991.
- [15] D. Powell, "Lessons learned from delta-4," *IEEE Micro*, vol. 14, no. 4, pp. 36–47, 1994.
- [16] J. Reisinger and A. Steiniger, "The design of a fail-silent processing node for the predictable hard real-time system mars," *Distributed System Eng. Journal*, vol. 1, no. 2, pp. 104–111, 1993.
- [17] M. Z. Rela, H. Madeira, and J. G. Silva, "Experimental evaluation of the fail-silent behavior in program with consistency checks," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing (FTCS-26)*, pp. 394–403, July 1996.
- [18] D. T. Stott *et al.*, "Fault-injection-based assessment of fail-silence provided by process duplication versus internal error detection in scientific-based applications," Tech. Rep. CRHC-00-03, University of Illinois at Urbana-Champaign, 2000.
- [19] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer, "Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE," in *IEEE Int'l Computer Performance and Dependability Symp. (IPDS'2K)*, pp. 91–100, Mar. 2000.
- [20] A. Thakur, "Measurement and analysis of failures in computer systems," Master's thesis, University of Illinois at Urbana-Champaign, 1997. Advisor R. K. Iyer.
- [21] T. K. Tsai and R. K. Iyer, "An approach to benchmarking of fault-tolerant commercial systems," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing (FTCS-26)*, pp. 314–323, June 1996.