Tags: Augmenting Microkernel Messages with Lightweight Metadata

by

Ahmad Saif Ur Rehman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this work, we propose Tags, an efficient mechanism that augments microkernel interprocess messages with lightweight metadata to enable the development of new, systemwide functionality without requiring the modification of application source code. Therefore, the technology is well suited for systems with a large legacy code base and for third-party applications such as phone and tablet applications.

As examples, we detailed use cases in areas consisting of mandatory security and runtime verification of process interactions. In the area of mandatory security, we use tagging to assess the feasibility of implementing a mandatory integrity propagation model in the microkernel. The process interaction verification use case shows the utility of tagging to track and verify interaction history among system components.

To demonstrate that tagging is technically feasible and practical, we implemented it in a commercial microkernel and executed multiple sets of standard benchmarks on two different computing architectures. The results clearly demonstrate that tagging has only negligible overhead and strong potential for many applications.

Acknowledgements

I would like to thank all the little people who made this possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In todays world, embedded systems are becoming omnipresent, controlling technologies in applications ranging from handheld devices to safety-critical medical devices. Embedded systems are specialized systems that are designed to implement a particular task. Most embedded systems work in real time; i.e., temporal requirements, along with the logical result, define the correctness of the outcome.

Real-time embedded systems can be broadly categorized into two classes: hard real-time systems and soft real-time systems. In hard real-time embedded systems, strict temporal requirements drive the correctness of outputs. A missed deadline leads to a catastrophic failure on such systems. Hard real-time embedded systems are usually deployed in safety-critical systems such as those used in avionics. In soft real-time embedded systems, violations of a temporal requirement affect the usefulness of the outcome but do not lead to a catastrophic failure. Less safety-critical systems such as household systems are examples of soft real-time embedded systems.

Embedded systems are proliferating with increasing speed. With an increase in demand, embedded systems have become rich in features. Several components, at different software layers, provide support to implement these features. Generally, an embedded system categorizes components into three layers: the application layer, the system software layer, and the hardware support layer. To implement a particular task, components interact with other components of either the same or a different layer. A multi-component environment improves the performance, modularity, and reusability of the system, but several interacting components can add to the complexity of the system.

Building complex systems—especially those with a high level of interaction between several components at several abstraction layers—is a difficult task. Development of system-

wide functionalities on such systems requires between 30 and 50 percent of the total development cost in software systems [24, 51]. In such complex systems, component interaction information can aid the development of system-wide features that are orthogonal to the application functionality. Profiling, tracing, interaction verification, and mandatory security are examples of such functionalities that only require runtime access to components' execution flow.

Tracking and extraction of a component's interaction flow are challenges for the development of systemwide features, specifically in real-time systems. In real-time systems, the overhead of tracking component information should be minimal so that applications can meet the timing constraints. Instrumenting the source code of an application can help in extracting the interaction pattern, but retrofitting a large code base will still be time consuming. In the case of real-time systems, code instrumentation will degrade the application's performance to the extent that it may violate its timing constraints. Furthermore, instrumentation, at the source level, will not be applicable to closed-source applications.

Dedicated embedded systems often run everything as a single application when they aim to avoid the complexities of a multi-component environment. In contrast, rich-featured embedded systems must run an operating system to support a multitude of applications. Many operating systems are specifically tailored for real-time embedded systems. Modern operating systems play an important role in helping the developer to build systemwide features because operating systems provide common abstractions for system services and hardware, manage resources, and provide basic functionality as a part of libraries and system calls. Any modern operating system should strive to provide a rich set of functionality to allow developers to rapidly implement systemwide features. For example, a versatile and reusable interprocess communication infrastructure will potentially speed up development if it keeps the programmer from reimplementing the same functionality. To aid the development of systemwide functionality, an operating system may contain the infrastructure for tracing and extracting the interaction between system components. Tracking interaction among system components at the operating system level does not require access to the source code of the application. This makes it instantly reusable and applicable for closed-source applications.

In the operating system, one way to track and extract the program's interaction information is to attach information with programs and propagate this information as the program communicates with other system components. Past approaches [43] have used this mechanism for information flow control (IFC). IFC is a mechanism to track data flow between components. Other approaches [104] have proposed a labelling technique for profiling and debugging purposes. These approaches track information flow among different components in the system at the granularity of memory byte, function call, and process

communication. These information-tracking mechanisms introduce great overhead in the system, which limits these approaches to the testing of systems only.

In this study, we introduce the notion of Tags. A Tag represents lightweight metadata that the system attaches with threads and propagates with passed messages. The tagging mechanism has the following properties:

- It provides a generic infrastructure to track and extract components interaction patterns.
- It snoops the communication layer to track interaction patterns with minimal overhead and is thus applicable to real-time systems.
- The operating system implements tagging and thus makes the tagging mechanism application-agnostic and applicable for closed-source applications.

The application-agnostic behaviour of tagging can help in realizing some important systemwide use cases that are orthogonal to an application's functionality. These use cases include, but are not limited to, mandatory security, verification of process interaction, and profiling. Each of these use cases is briefly described in the following paragraphs.

μMIP: Mandatory Security for Microkernel-based Systems μMIP assesses the feasibility of realizing mandatory security in microkernel-based systems. The μMIP infrastructure is an adaptation of a recent approach called the Usable Mandatory Integrity Protection Model (UMIP) [88]. Like UMIP, μMIP assigns integrity levels to the processes that define the capabilities of a process in the system. The integrity level of a process disseminates in the system as the process communicates with other processes. The tagging infrastructure implements the mechanism to define and propagate integrity levels. Chapter 3 discusses the μMIP model in detail.

Intersert: Assertions on Process Interaction Section The *Intersert* framework uses the tagging mechanism to verify the process interaction patterns. intersert() demonstrates the utility of assertions on the interaction history, among system components. In the proposed framework, tagging solves the challenges of efficiently maintaining interaction data. intersert() provides an expressive interface for developers to program assertions on the interaction history of threads. The assertions contain Linear Temporal Logic (LTL)

statements placed on the interaction history. Chapter 4 discusses the intersert() framework in detail.

Profiling/Debugging To aid the system designer in understanding the interaction between system components, tagging enables comprehensive tracing of those interactions. If a thread creates a tag, it will be passed on with each message it sends and, eventually, all components in the system that it interacts with (directly or indirectly) will also have received that tag. Lifeline implementation also offers additional information for profiling. The *lifeline*, as described in Section 1.1.9, for a particular tag shows the complete flow of the tag through different threads in the system, helping the developer to identify how much time is spent in each component, the number of involved threads, the order of execution, and the termination of the flow (either expected or unexpected).

1.1 System Model and Terminology

For the benefit of the reader, this section briefly describes the tagging model and associated concepts. This section also provides a brief description of standard operating system services, which aid the development of the tagging infrastructure.

1.1.1 Microkernel and Monolithic Kernel

Microkernel and monolithic kernel are two well-known operating system architectures. A monolithic kernelbased operating system provides most of the services as part of the kernel. These services run as a part of system process in privileged mode. In a microkernel architecture, services are strictly categorized as essential and optional. A microkernel system implements the essential services, which include scheduling, synchronization, memory management and Inter Process Communication (IPC). All other optional services, such as device drivers and web servers, run as external processes in the user space.

1.1.2 Multiprocess and Multithread

Processes and threads are the key entities of the operating system. Process represents a container that contains all of the resources, whereas thread is the executional unit. Process provides all of the resources for the threads to execute. A multiprocess operating system has the ability to host multiple processes. In a multithreaded operating system, a process can contain more than one thread.

1.1.3 Inter Process Communication

All the services communicate through Inter Process Communication (IPC) mechanism, which the microkernel provides. Modern microkernels support both synchronous and asynchronous communication among services. Services use *messages* for synchronous communication. Pulses and signals implement asynchronous communication. Following paragraphs provide brief description of different forms of supported IPC.

Messages: In a microkernel, the services usually communicate through a messaging layer. A message contains the sender identification, the receiver identification and the payload. The microkernel uses sender and receiver information to deliver the message. The payload represents the data that the sender wants to exchange with the receiver. The protocol between the sender and the receiver defines the structure of the payload. The system contains two types of messages: requests and replies. *Request messages* initiate communication between two services. A *reply message* is a response to a request message. Modern microkernels also implement transparent distributed messaging.

Pulses and Signals: Pulses and signals are short asynchronous messages. These short non-blocking messages are used to notify other services about events in the system.

Shared Memory: Shared memory provides high-performance IPC among services. Services communicate through shared memory by directly reading and writing to the shared memory region. Access to a shared memory is unsynchronized and services should agree on a synchronization mechanism to prevent data inconsistency.

1.1.4 The Concept of of a Tag

The key abstraction for the tagging model is the notion of a *tag*. A tag is an abstract entity similar to a label, which users or programs can attach to threads. We extend the concept of IPC, as described in Section 1.1.3, to include the propagation of tags. In the tagging infrastructure, all communications between sending and receiving threads contain an additional field, a *tag*. When it is received, the receiving thread will acquire the tags the sender had at the time of the transmission. All future communication initiated by the receiver will carry its current tags unless the developer deliberately chooses to change this behaviour.

1.1.5 Tag Propagation

The well-defined message-passing mechanism of microkernels serves as the medium for tag propagation among threads. The tagging infrastructure adds a tag which is passed with the message. Once a thread has a tag, the thread propagates the tag to the receiving thread via *request messages*, as described in section 1.1.3. Tags do not propagate with *reply messages*. We found this propagation mechanism to be intuitive and sufficient for a large variety of use cases.

In addition to the request messages, the tagging mechanism also propagates tags through pulses, signals and shared memory.

1.1.6 Distributed Tagging

Tagging also works for distributed systems in which all participating nodes run a compatible microkernel. In a microkernel architecture, messages can transparently pass through an interconnect from one node to another. The implementation of this mechanism is more complicated: however, the underlying concept and system remains the same.

1.1.7 Tag Propagation Modes

One can implement different semantics for tag propagation such as *tag duplication* or *baton passing* as entities interact. *Tag duplication* mode refers to the concept of "copying" tags to other entities on interaction as opposed to baton passing, where the tag passes from one entity to another.

In *baton passing* mode of propagation, tags are propagated *without* duplication. As an example, consider two threads, A and B. Thread A currently holds a tag and interacts with Thread B. Thread B will receive Thread A's tag, while Thread A will lose its tag. *Tag duplication* propagation mode copies the tag from source to the receiver while retaining the tag at the source. For example, in the previous example, Thread A will retain it's tag even after the reception of tag at Thread B.

Example 1 (Abstract Example) *The example shown in figure 1.1 illustrates the tag propagation mechanism with duplication mode. Large circles represents the processes P1, P2 and P3. Each of these processes contains two threads t1 and t2, shown as small circles in the diagram.For simplification we will be referring to these threads as* txpy *where* tx

represents the thread x in the process y. Horizontal line shading shows the tag τ_a whereas the vertical line indicates the presence of tag τ_b. Lines represent a message pass from one thread to the other with arrow indicating the direction of the message pass.

The system starts with 3 processes. At the system start-up, the user assigns tag τ_a and tag τ_b to thread t1p1 *and* t2p1*, respectively. As soon as these threads communicate with any other thread in the system they will propagate their respective tags. The figure shows the dissemination of tag τ_a and tag τ_b by shading threads with horizontal and vertical lines, respectively. As shown in the figure 1.1, thread* t1p1 *propagates the tag τ_a to* t1p2 *as soon as it communicates with* t1p2*. The kernel further propagates the tag τ_a to* t2p3 *as a result of the message pass from* t1p2 *to* t2p3*. Propagation of tag τ_a stops at* t2p2 *as* t2p2 *does not communicate further with other threads in the system. As a result of not receiving any messages from other processes, thread* t1p3 *will not have any tag.*

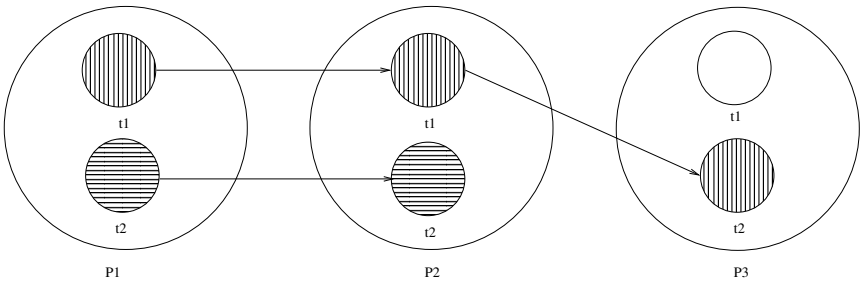

Figure 1.1: Example of tag propagation

1.1.8 Controlling Tag Propagation

In a microkernel nearly every action results in a message pass between two or several threads. As a tag propagates with messages, occasionally the large number of messages may result in uncontrolled dissemination of this tag through the system. Referring to figure 1.1, thread *t2p2* will propagate tag τ_b with all the future outgoing messages. Oftentimes a guided and limited propagation of a particular tag is desirable to extract only the information of interest. We provide several mechanisms to control or limit the dissemination of any particular tag in the system. These mechanism include tag propagation modes, time to live (TTL) counters, tag terminators and system tags. Figure 1.2 highlights these mechanisms by showing the tag propagation in different scenarios. Figure 1.2 shows the processes as large circles. The small circles, within the large circles, represent the threads.

Vertical and horizontal line shadings of threads represent the presence of tag τ_a and tag τ_b, respectively. The arrows in the figure indicates the message pass where direction of the arrow indicates the message flow. Process P5 is a system process and processes P1 to P4 represent user processes.

Time to live (TTL) is a concept that involves limiting the timespan of a packet or data on a computer or network. The tagging mechanism implements the TTL as counter. The TTL value puts an upper threshold on tag propagation. Every time a tag propagates to another thread, the kernel increments the TTL count of the tag. As soon as the TTL count of the tag reaches the predefined TTL threshold, the kernel will not allow further propagation of the tag. Figure 1.2 explains the TTL mechanism by associating TTL value of 3 with tag τ_a. As shown in the figure 1.2, tag τ_a propagates from thread *t1p1* to thread *t1p2* as a result of message pass between two threads. Upon reception at thread *t1p2*, the TTL value of tag τ_a will increment to 2. As figure 1.2 indicates, the τ_a tag will propagate to thread *t1p3* as soon as thread *t2p1* communicates with thread *t1p3*. As a result of this propagation of tag τ_a the TTL value will reach it's threshold value i.e., 3. Now the TTL value of tag τ_a will prevent the further propagation of tag τ_a. As depicted in the figure 1.2, the message pass from *t1p3* to *t1p4* will not propagate tag τ_a to *t1p4*.

Tag terminator is an attribute of a thread in the tagging infrastructure. A thread can have a tag terminator for one or several tags. Presence of a tag terminator prevents the tag from propagating further in the system. In other words, any particular tag will not propagate beyond a thread that defines a tag terminator for that tag. In figure 1.2, thread *t2p2* has the tag terminator for tag τ_b. The presence of this tag terminator will prevent further propagation of tag τ_b from *t2p2* to any other thread in the system. Consider figure 1.2, thread *t2p2* will mark the end of propagation for tag τ_b and will not propagate it to *t2p3* with the message.

System threads frequently communicate with all components of the system. During our experiments the instant propagation of the system thread's tag, to almost all of the components, became evident. Tagging mechanism distinguishes the system level thread by means of system tags. All system level threads carry system tags. The unpassable property differentiates systems tags from user tags i.e., system tags do not propagate with the message. Furthermore, threads with system tags do not receive tags from other threads. In figure 1.2, process P5 is a system process with threads *t1p5* and *t2p5*. As shown in figure 1.2, the thread *t1p5* will not receive or propagate tags to any other thread in the system.

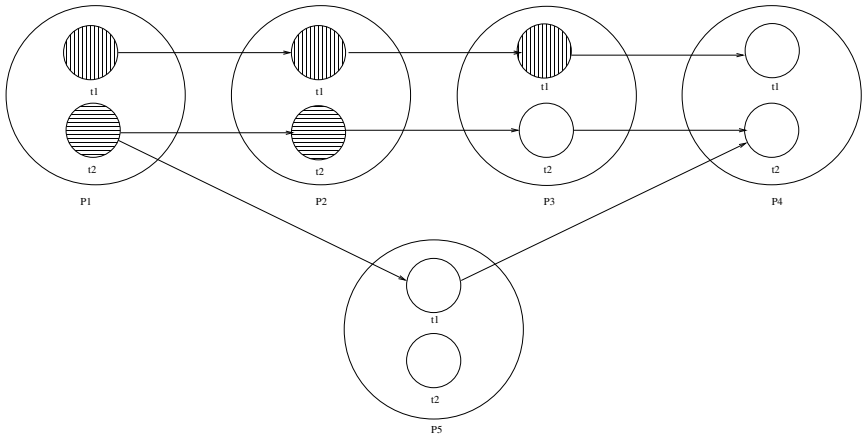

Figure 1.2: Limiting tag propagation

1.1.9 Lifeline

The lifeline mechanism offers additional information for several purposes. The lifeline for a particular tag shows the complete flow of the tag through different threads in the system. This aids the developer in the identification of how much time is spent in each component, the number of involved threads, order of execution, and the termination of flow (either expected or unexpected). Example 2 illustrates the lifeline mechanism along with other tagging features.

Example 2 (Tracing File Writes) *This example uses our tagging mechanism to determine what processes are involved in the* mmap() *system call. The* mmap() *function maps a file or memory in the process address space. The example consists of three threads running in a microkernel such as QNX Neutrino: the parent process thread, the child process thread, and the filesystem resource manager thread. The parent process spawns the resource manager thread and the child process. The resource manager thread handles all file operations whereas the child process calls file functions. Using tags, we can discover how the child process interacts with the resource manager thread to perform its file operations.*

We setup three tags to track the activity of each thread in the system. We use CreateTagsfield() *to create the tags τ_p, τ_c and τ_r and assign them to the parent thread, the child thread, and the resource manager thread, respectively.*

After execution of the program, τ_p and τ_r are still only associated with their original threads. On the other hand τ_c is spread throughout the system, as a printout of the list of threads containing τ_c shows:

```
Process ID 253980 Thread ID 0 Time Fri Apr 23 12:39:25:901577382
Process ID 249877 Thread ID 1 Time Fri Apr 23 12:39:25:913575546
Process ID 249877 Thread ID 1 Time Fri Apr 23 12:39:25:914575393
Process ID 249877 Thread ID 1 Time Fri Apr 23 12:39:25:914575393
Process ID 8200   Thread ID 6 Time Fri Apr 23 12:39:25:915575240
Process ID 249877 Thread ID 1 Time Fri Apr 23 12:39:25:915575240
Process ID 249877 Thread ID 1 Time Fri Apr 23 12:39:25:915575240
Process ID 249877 Thread ID 1 Time Fri Apr 23 12:39:25:916575087
Process ID 249877 Thread ID 1 Time Fri Apr 23 12:39:25:916575087
```

Process ID 253980 Thread ID 0 is the child thread, Process ID 249877 Thread ID 1 is the resource manager thread, and Process ID 8200 Thread ID 6 is the devb-eide thread, which manages the IDE bus.

The user can employ the information above to understand the interaction among the threads in the system. For example, the propagation of τ_c to the resource manager indicates the request of the child process thread to the resource manager thread for functions calls like stat(), unlink() and write(). The resource manager thread has also propagated τ_c to the devb-eide thread for example through the unlink() operation.

The tag thread list for τ_c also gives the user an idea of the execution flow of mapping a file initiated by the child process. Dissemination of τ_c can help in identifying the number of threads used to complete the file mapping operation. In this example, τ_c was propagated to three threads, i.e., the child process thread, the resource manager thread and the devb-eide thread.

If τ_a were a system tag, then the tag list would have shown more entries. The system thread that manages the memory also handles requests from mmap() issued by the client. However, since it is a system thread and τ_a in our example is only a user tag, this system thread is absent in the tag list for τ_a.

1.2 Layout

The remainder of the thesis is structured as follows: Chapter 2 highlights the past work on information tracking. Chapter 2 also describes the approaches that relate to the tagging mechanism in terms of the proposed use cases. Chapter 3 and 4 describe the mandatory security and process interaction verification use cases of the tagging infrastructure, respectively. Chapter 5 discusses the implementation details. Chapter 6 presents the evaluation methodology and results. Chapter 7 concludes the thesis and discusses the future work.

Chapter 2

Literature Review

The use cases explored in Chapters 3 and 4 demonstrate the versatility of tags; tags are more than a simple message logging or a profiling mechanism. Tag creation, deletion and propagation is completely dynamic and distributed. Furthermore, applications can act upon the presence of a tag at runtime, not only in after-the-fact trace analysis. It is, therefore, difficult to compare tagging with related approaches since we feel not many similar works are as versatile; we will, then, make a series comparisons by use case, highlighting why tagging generally has a versatility edge.

2.1 Labelling Techniques

Asbestos [43] presents the idea of attaching labels to processes for controlling and tracking information flow. In Asbestos, each process contains two labels: a clearance label and a tracking label. The tracking label contains the level of all the information the process has seen whereas the clearance label represents the information level the process is allowed to see. A process can send a message to another process, if the tracking label of the sender process is less than or equal to the clearance label of the receiver process. If the receiver is cleared, its tracking label will be updated to represent the different level of information the process has viewed. In our proposed mechanism the propagation of tags could be used for similar means, by having the kernel stop messages that broke the clearance relationship. Also, our approach can easily be used to implement the tracking of information flow with minimum overhead.

The concept of badges in sel4 [117] allows the server to provide multiple interfaces to the clients through same endpoint. Badges propagates with the messages like the tags, in

our tagging mechanism. Tag differs from badges in the area of application as badges are used to identify the capability of the badged endpoint or thread.

strace [49] is a tool used for profiling system calls made by a process in Linux. It logs all the system calls made by a process and the signals it receives. strace is useful for tracing the activities beyond the user space boundary into the kernel as both levels communicate through signals and system calls. Tracing of system calls is supported by Tags through the logging of interaction between user and system threads. Our implementation of Tags was done in the kernel, allowing it to trace different kinds of activities initiated by a process either from the user space or the kernel space. Furthermore, tagging is not limited to system calls, as it also profiles the interaction between threads at different system layers. Furthermore, applications can dynamically act upon the presence or absence of tags, a feature that is absent from systems that focus exclusively on tracing.

The Data Tomography [104] system proposes tracking data flow across multiple layers of abstraction by tagging the data in the system. The data tomography technique consists of inserting tags at the application, the network and the instruction level. It creates a tag map for each byte in the physical memory. The tag map of every byte stored in physical memory either the instruction or the data, points to some format of the tag. The format can vary from a simple collection of numbers to any other complex format. In contrast, our approach is to attach tags to threads rather than the physical memory in the system. Our mechanism incurs less overhead than the data tomography by avoiding the approach of tagging all the physical memory. Overhead reduction makes our tagging mechanism deployable in a production system rather than using it just for instrumenting purposes.

TaintDroid [45] is an extension to the Android [15] operating system that uses message-based taint tracking to detect the leak of sensitive information in mobile devices. While TaintDroid's approach is similar to ours (attaching metadata to IPC messages), Tagging is a more general mechanic aimed at enabling a wide range of use-cases. Furthermore, because it is based on a real microkernel, Tagging is capable of tracking interactions between system services and user applications; the same functionality would require modifications to the underlying native services in the case of Android. On the other, TaintDroid's is capable of tracking taints at the variable level while tagging only tracks them at the thread level.

The labeling approaches like HiStar [147] and LoStar [148] are based on the Asbestos labeling technique. The HiStar defines new kernel architecture with focus on the system security. The LoStar is an extension of the HiStar and uses tagged memory architecture.

Different kinds of related work in the past addressed the issue of dynamically tracing and debugging operating systems. This includes for example the Linux Trace tool [145], dynamic probes [101], kernel probes [77] and DTrace [30]. All of them provide mechanisms

for inserting probes, sensors, and monitors into the system, with the objective of capturing data or the system state for tracing purposes. The tagging mechanism gives the user the provision of tracing the system at the granularity of the threads. The user can utilize the tagging without the deep understanding of the system and access to the source code. DTrace's D scripts are powerful, and may conceivably be used to achieve functionality similar to Tags, but they offer no direct support for distributed systems.

2.2 μUMIP: Mandatory Security for Microkernel-based Systems

The μUMIP model relates to the past work along two dimensions: Integrity Models, and security.

Integrity Models: The Clark-Wilson model [34] attaches integrity with data in terms of constrained data items and unconstrained data items. Transformation procedures are allowed to change constrained data items. The system certifies each transformation procedure by assigning the list of CDI to the transformation Procedure.

Usable integrity propagation model (UMIP) [88] is the most recent work on integrity propagation and closely related to our work. Like our model, UMIP propagates and tracks the integrity levels among processes in the system. UMIP also associates and updates the integrity level of the files. UMIP model trusts most of the components of the operating system like kernel modules, device drivers and filesystems.

Microsoft Windows vista [11] introduces Mandatory Integrity Control (MIC). The MIC associates the mandatory label with each securable object i.e., processes, files etc. Each object also has a security identifier that represents the integrity level of the object. The operating system performs a mandatory access control check based on the integrity level of the requesting process and the mandatory label of the object being accessed. MIC enforces different policies like no write up, no read up and no execute up. These policies define integrity access rules. For example, no write up policy prevents lower integrity level processes from writing to objects at higher integrity level.

Other works on mandatory access control includes Trusted Solaris and 1X [98] and PACL [143]. Trusted Solaris provides multi-level security through mandatory access control mechanism. PACL focuses on data integrity and attaches integrity with the object. It binds a list of programs, allowed to change the file, with a file.

Security: AppArmor [67] provides system protection by creating system profiles for programs. A security profile list all the system operations and files, a process is allowed

to access. AppArmor does not attach integrity with the processes and files in the system. Furthermore AppArmor does not guarantee the security in the scenarios where user downloads and executes a malicious program.

Securelevel [72] uses securelevel indicator to reflect the security state of the system. The positive securelevel restricts all the processes from certain tasks. The super user process is allowed to raise the securelevel and only the init process is allowed to lower it. Securelevel is very restrictive in terms of usability, and protecting a system with it is difficult.

SELinux [7] provides the mandatory access control for the Linux operating system. SELinux requires extensive configuration that includes manual labelling of all the files in the system, definition of the MAC privileges of the users, definition of different complex policies and updating the policies with the installation of the new application. All these configurations can be error prone and difficult to understand by a system administrator.

The μMIP model is the first model to provide a mandatory integrity protection mechanism for microkernels. Our model takes advantages over all previously proposed integrity propagation model by ensuring the system integrity with minimal configuration and impact on usability, negligible impact on performance, reducing the trusted code base and dynamic assignment and tracking of processes and files contamination.

2.3 Intersert: Assertions on Process Interaction Section

Several works propose the idea of using LTL to verify applications at run time, or to use more sophisticated assertions in programming languages. Some approaches attempt to verify LTL properties statically [64, 70] while most apply runtime verification at run time. Our work relates specifically to the work that provide assert based verification (ABV) at software level.

Partial translation verification [132] provides a mechanism for the verification of the correctness of code generated by code generators. The work demonstrates the effectiveness of its proposed strategy by showing its applicability to model-based development. When using modelling languages like Simulink, LTL properties can capture the requirements of the model before code generation. The proposed infrastructure translates these LTL properties into C assertions. The C assertions, based on a set of LTL properties, verify the correctness the generated C code for a particular model.

The Trace Analyzer (TaZ) [52] translates LTL properties to finite-state automata called observers. These LTL-based observers permit verification of the traces of a running programs. The observer checks whether the current running process conforms the supplied LTL formula. The TaZ has been integrated into the Java language.

Different existing approaches use code annotation techniques to verify the safety properties of the code. Denny proposed a code annotation mechanism [39] to specify safety properties. A proof checker verifies these safety properties expressed as annotations. Frama-C [29] is also based on the idea of annotating code with LTL properties and verifying the annotations using an external checker. Necula and Lee proposed a compiler [106] that verifies the memory-safety properties of assembler using code annotations.

sPSL [32], a subset of the Property Specification Language (PSL), proposes assertion-based verification (ABV) of C programs using LTL properties. sPSL provides developers with the provision of writing specifications of C programs, captured as LTL properties, in PSL. sPSL checks the specified properties during the execution of the program.

Several published tools have presented the idea of runtime monitoring by translating LTL properties into executable code. The executable code is merged with the target program and performs monitoring during execution. Temporal Rover and DBRover [41, 42] are examples of such tools. They both support LTL properties written within code comments. A parser converts the LTL formulae contained within the comments into assertions.

ASAP [38] proposes the idea of detecting faults at run time by using assertions and pre-processors. ASAP creates C assertion statements from first-order logic and partial functions. These assertions are placed into the application's code and are responsible for checking the system at run time.

Java-Mac [74], uses runtime monitors to check the executing program against the defined formal specification. Java-Mac defines events and their desired relations by using Primitive Event Definition Language (PEDL). Java-Mac can only be used to verify Java programs and is oblivious to process interactions.

The *Intersert* framework differs from past work as it concentrates on the interaction behaviour of applications. Unlike the approaches mentioned above, *Intersert* validates the interaction history of threads.

Chapter 3

Use Case: Mandatory Access Control in Microkernel Based Operating System

Microkernels provide protection against attacks and faults by isolating components into address spaces. Components in microkernel-based systems communicate through Inter Process Communication (IPC). Spatial protection, using address space, does not guarantee security as an attacker can exploit IPC mechanisms to gain control of trusted components. For instance, in case of remote attacks, restricting network-facing applications is not sufficient to thwart security attacks on systems. As successful attacks against network-facing applications can cascade into local attacks against other software (to achieve a privilege escalation, for example), simply marking network-facing software as a potential intruder is insufficient. For example, if a webserver becomes compromised and then performs IPC with other software, any faults in the participating processes could conceivably be exploited. It is important, therefore, to mark participating processes as potential intruders too. As the marked processes communicate, the security model needs to propagate the threat flag to further participating process. The tagging propagation model provides the grounds for tracking such attacks in the system.

This chapter assesses the feasibility of the tagging mechanism for the implementation of mandatory-security propagation models in microkernel-based systems. The particular focus is on our design of the tagging infrastructure for the propagation of what we (and Li et al [88]) call an *integrity level*. An integrity level defines a process' capabilities and the trust given to it. The mandatory security model, presented in this chapter, exploits the tagging propagation mechanism to define and track the integrity level of processes.

The structure of the chapter is as follows: The first Section 3.1 provides an overview and motivation for the work. In the next section, we discuss UMIP. In Section 3.3 we describe our adaptation of UMIP to microkernel-based systems. In Section 3.3.1 we discuss the design of μMIP The chapter concludes with Section 3.5, in which we also discuss future work.

3.1 Overview

The security administration of enterprise operating and file systems can be a significant challenge. With the number, size and complexity of programs that are run, it is difficult for a security administrator to have some sense of control over his systems. The operating system and the security administrator seem powerless in the presence of bugs in user programs such as buffer overflows [1] and format-string vulnerabilities [3] that impact security and show no sign of abatement.

The situation seems no different in microkernel-based systems. Unlike monolithic operating systems, the kernel in a microkernel-based operating system is small, and provides only the most basic services. Components such as device drivers and file system managers reside outside the kernel, in userspace. Microkernel-based systems are the operating systems of choice for emergent embedded- and application-centric devices that have started to proliferate [2, 4, 5]. The modern enterprise comprises not only traditional computer systems that run monolithic operating systems, but also scores of devices that run microkernel-based operating systems.

It has been argued [137] that microkernel-based systems are inherently more secure than monolithic systems. However, the discovery of security vulnerabilities in deployed microkernel based systems suggests otherwise. Indeed, a perusal of the security vulnerabilities that have been discovered in deployed QNX systems suggests that such systems are as susceptible as monolithic systems to user programs that are buggy [8]. QNX Neutrino [10] is a widely-deployed [9] microkernel-based operating system, and the system on which we have carried out our proof-of-concept implementation.

In our work, we focus on a particular approach for mitigating the impact of buggy programs on the security of a system. The approach is mandatory security protection. With mandatory security, the system imposes rules that individual users are unable to influence as the system runs. Mandatory security is typically contrasted with discretionary approaches, in which users may change security rules at runtime. Mandatory security has been explored extensively for monolithic systems (see, for example, [6, 7, 16, 54]). Our intent is to assess its feasibility for microkernel-based systems.

The usefulness of mandatory security (in particular, access control), as a complement to other approaches that are typically deployed in enterprise settings has been recognized since the work on Multics [68]. SE Linux [7] is a more recent example of the realization of mandatory security. SE Linux complements traditional discretionary protections in Linux by providing configurable mandatory protection. The objective of SE Linux is [7]:

> ... provides a mechanism to enforce the separation of information based on confidentiality and integrity requirements. This allows threats of tampering and bypassing of application security mechanisms to be addressed and enables the confinement of damage that can be caused by malicious or flawed applications.

It is our premise that mandatory security, as a complement to other approaches such as discretionary and role-based [126], can greatly ease security administration. Consider the passage on SE Linux from above. Rather than relying on users that may not be fully trustworthy to exercise good judgement, which is the case with discretionary security, with mandatory security, an administrator can configure a system to ensure that certain kinds of operations are not possible. For example, a somewhat draconian policy such as, "no process that has ever communicated over the network is allowed to read `/etc/passwd`," can be effected with mandatory security, notwithstanding the actions of particular users.

To establish the premise that mandatory protection can be meaningfully realized in realistic, commercial microkernel-based systems, we have chosen to focus on and realize Usable Mandatory Integrity Protection (UMIP) [88] on QNX Neutrino. UMIP has been explored in past work for monolithic systems, and, as mentioned before, QNX Neutrino is widely deployed in realistic settings. Our choice is motivated by UMIP's newness, and the philosophy that underlies its design. In this context, "usable" refers to ease of administration. An intent behind UMIP is to mitigate the somewhat high administrative impact that approaches prior to it have imposed on a security administrator from the standpoint of configuration. We discuss our choice of UMIP in more detail in Section 3.2. We argue however (see Section 3.3) that there is nothing inherent to our work that is limited to UMIP. We reemphasize that our goal is to demonstrate that mandatory protection, whether UMIP or some other approach, can be meaningfully and effectively realized in microkernel-based systems.

Novelty and Contributions The novelty of our work is in the demonstration that mandatory protection can be realized effectively in realistic, commercial microkernel-based systems. To our knowledge, there is no prior work that establishes this. In our section on related work (Section 2.2 of Chapter 2), we discuss relevant prior work in more detail. Here,

we summarize by asserting that prior work can be broadly dichotomized into: (1) work on monolithic systems to realize mandatory protection, and, (2) work on microkernel-based systems that is related to security, but not mandatory protection for realistic, commercial systems.

As contributions, we point to our concrete instantiation of an effective approach to mandatory protection from the literature that has been touted as easy to administer. The instantiation, as it turns out, is not simply a matter of reimplementing the UMIP approach in a microkernel. As we discuss in Section 3.3.1, we have had to make a number of design choices in doing so, and the adaptation to microkernel-based systems is not straightforward. Indeed, we were unsure when we started this work that it would even be possible to meaningfully realize an approach such as UMIP in microkernel-based systems. We provide a comprehensive empirical assessment of the approach, and a clear articulation of the trade-offs we encountered in our work.

Motivation We assert above that microkernel-based systems, in practice, have shown themselves to be susceptible to the same kinds of security issues as monolithic systems. This may come as somewhat of a surprise, and therefore may be worth reemphasizing.

A common way for a remote attacker to compromise a monolithic system is to first find a buggy program that he can access that runs with elevated privileges. Once he compromises such a program, for example, via a buffer overflow, he installs a rootkit. A rootkit grants a privileged access to an attacker. To our knowledge, microkernel-based systems (that run QNX Neutrino, for example), do not have a well-known rootkit available for them.

However, a rootkit can certainly be developed for such systems. The nature of bugs with security consequence that have been discovered for such systems [8] is similar to those for monolithic systems, and lend themselves well to the installation of a rootkit. The kinds of problems include buffer overflows that can result in elevated privileges for a remote attacker, misconfigurations in systems that have been shipped such as incorrect permission settings on sensitive files, format-string vulnerabilities, and the provision of sensitive services such as the qconn utility for remote administration [122] which an attacker could potentially compromise. Indeed, we have been able to simulate a breach in the QNX Neutrino microkernel and have used the qconn utility as the equivalent of a rootkit. In our attack, we managed to read and then remove the /etc/shadow file that is used for authenticating users.

In summary, we assert that microkernel-based systems, in practice, appear to suffer from several of the problems that monolithic systems have. As mandatory protection is

seen as a possible approach to mitigating the vulnerability of monolithic systems, it is natural to ask whether mandatory protection is feasible for microkernel-based systems as well. This is exactly the question we address.

3.2 UMIP

UMIP [88] is a recent approach to mandatory protection in monolithic systems. Underlying UMIP is a number of design choices that distinguishes it from approaches prior to it. We refer the reader to the original work on UMIP [88] for a comprehensive discussion on their design rationale. We give a limited discussion here.

One of the goals of UMIP, as its name suggests, is to be usable. What usable means in this context is easy to administer. Prior approaches to mandatory protection such as SELinux [7] have imposed a significant burden on the administrator from the standpoint of configuration. The designers of UMIP have also made other appealing design choices, such as providing policy and not just mechanism, and articulating a clear security objective. The security objective of UMIP, that we adopt as well, is to protect the integrity of a system from network-based attacks. That is, an adversary is someone that is remote to the system, and attempts to compromise the system over the network. The (remote) adversary may attempt to do this in several steps, and via indirection. For example, he may first exploit a buffer overflow, then install a rootkit, and then overwrite system files, or compromise the kernel, for example, by dynamically loading modules. In the security model of UMIP, the kernel of the operating system is fully trusted, and therefore it is certainly necessary to preclude any contamination of the kernel. Once that can be assured, the mechanisms related to mandatory protection can be part of the kernel.

Integrity, in this context, means the following. We recognize and reconcile ourselves to the reality that many userspace programs have bugs such as buffer overflows and format-string vulnerabilities that lend themselves to security compromises. These bugs may be exploited by an adversary. However, we are then able to confine the (further) damage to the system that the adversary can effect. In particular, he is unable to "take over" the system by, for example, installing a rootkit. He is also unable to compromise the system by some indirection, such as overwriting the /etc/passwd file so he can later login as a legitimate user.

We recognize that the above characterization of integrity is informal and not precise. However, we argue that this is necessary to realize solutions such as UMIP that are realistic, and in practice, are able to deter attacks. ("Good enough security" is another design goal

of UMIP.) UMIP associates an integrity level with every process. The integrity level is one bit; if it is set, the process is said to be of low integrity, and high integrity otherwise. In keeping with our discussions on the semantics of integrity from above, a process with low integrity can be thought of as tainted, and therefore, its privileges are restricted.

In the case of UMIP, as the threat is from the network, a process that has directly or indirectly interacted with the network or otherwise received low integrity data (e.g., from a file that is deemed to be of low integrity) is susceptible to be downgraded to the low integrity level. In this context, "directly communicated with the network" means that the process read from a network socket. "Indirectly" means that the process received some Inter-Process Communication (IPC) from another process of low integrity. We say that such a process is only susceptible to the lower integrity level, because such conditions are not always sufficient for lowering its integrity level. Other conditions, such as exceptions, may be applied in the determination as well. We defer a discussion of those details to the next section on our adaptation of UMIP to microkernel-based systems.

From the standpoint of mechanism, UMIP works as follows. Every process is associated with an integrity level as we discuss above. This bit is maintained as part of the Process Control Block (PCB) by the kernel. A process may be created as a low or high integrity process. Once a process reaches low integrity, it cannot be upgraded to high integrity. A process can change from high to low integrity in one of three ways: (1) receive network traffic, (2) receive IPC from a low integrity process, and, (3) read a low integrity file. In Figure 3.1, we reproduce the figure from the work on UMIP that illustrates this somewhat simple state-machine.

A process that is of low integrity is restricted in various ways. For example, it can write to only those files that are world-writable (i.e., writable by any process according to the discretionary access control policy), and it can only read files that are world-readable, or not owned by certain privileged users such as root. There can be exceptions to these rules, and a process can be excepted explicitly via a configuration by the administrator.

3.3 Our Adaptation of UMIP — μMIP

As we mention in Section 4.1, our objective is to demonstrate that realistic mandatory protection can be effectively realized for microkernel-based systems. Our approach has been to adapt and realize UMIP for the QNX Neutrino microkernel. We call our version μMIP. In this section, we discuss why we have chosen to focus on UMIP. We discuss also details of our adaptation, which reflects the difficulties we encountered, and the trade-offs

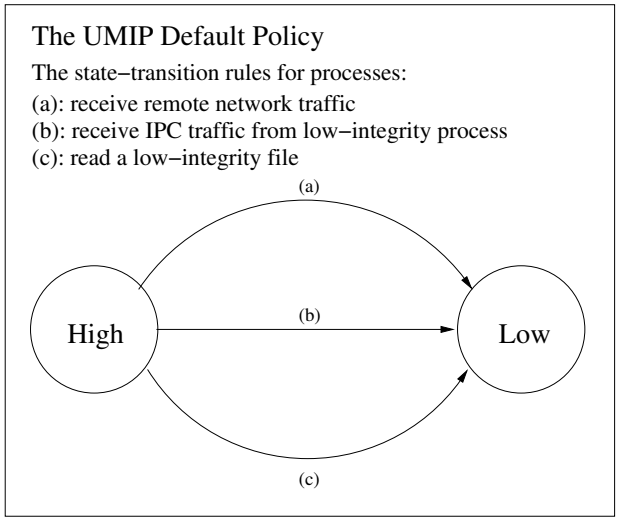

Figure 3.1: The UMIP state-machine for integrity levels, which we adopt for μMIP as well.

we have made. The discussion in this section is mostly about our design. In Section 5.2, we discuss implementation details related to QNX Neutrino.

Our choice of UMIP There are three reasons we chose to focus on UMIP. One is its newness. To our knowledge, it is the state-of-the-art in practical mandatory protection for realistic, albeit monolithic systems. We call it practical because it has been implemented for the Linux operating system, and therefore holds promise for real-world microkernel-based systems. A second reason is the focus on what UMIP calls usability. As we mention in the previous section, usability in this context refers to ease of administration. This is an appealing design philosophy to us. Indeed, it has been observed that administration can be a significant challenge in security [88].

UMIP's other design choices, such as the exception policy and not just mechanism are also appealing to us. We argue that the validity of the principles based on which UMIP is designed holds for microkernel-based systems as well. Our final reason is that UMIP has all the elements of a mandatory protection mechanism. Consequently, we argue that a demonstration that UMIP can be realized effectively for microkernel-based systems can be used to infer that other approaches to mandatory protection are also feasible. The reason is that the underlying mechanisms are the same, and one should be able to meaningfully reason about potential trade-offs with other approaches based on our observations in this

work.

Challenges that we faced Having justified our choice of UMIP in the above discussion, we now discuss the challenges in adapting UMIP to microkernel-based systems to realize μMIP. We faced two challenges. We discuss each challenge here, and the manner in which we address these challenges in the next two sections.

One challenge we faced regards an aspect that is customarily touted as an advantage, from a security standpoint, that microkernels have over monolithic systems. The filesystem and device drivers, are not part of the kernel. Consequently, unlike in UMIP, we cannot simply consult and trust attributes of files and devices as managed by a filesystem and device drivers. UMIP uses the discretionary access control settings (i.e., the Unix file permission bits) to determine whether a file should be deemed to be high or low integrity. We cannot do the same in μMIP.

The second challenge regards the interposition of our mandatory protection mechanism. As much as possible, we want the kernel to be the only entity that we trust. Consequently, the most natural location for our mechanism is as part of the kernel. However, we still need to clarify where exactly in the kernel we locate our mechanism. Or more specifically, at what points in the working of the system our mechanism kicks in. The location of our mechanism in the kernel raises other issues as well.

One is that the code-base of the kernel is now larger. This can be seen as a trade-off with the increased security from mandatory protection. However, an excessive increase may be deemed to be unacceptable. Also, our mechanism introduces overhead, quantified as delay, in the working of the kernel. This can also be seen as a trade-off for increased security. However, in this aspect as well, excessive overhead is unacceptable. Consequently, our challenge was to realize the mandatory protection mechanism is a lightweight manner, both in terms of the size of the code and the overhead it introduces to the working of the kernel.

Having discussed why we have chosen UMIP, and the specific challenges that we had to address in realizing μMIP, in the following sections, we discuss our design of μMIP and rationalize it. Section 5.2 of Chapter 3 discusses the specific implementation aspects of μMIP in QNX Neutrino.

3.3.1 Design Aspects of μMIP

A microkernel implements only basic system functions such as IPC and memory management; all other services such as filesystems or device drivers are executed as different

processes, each with its own address space. As we discuss in Section 4.1, microkernel-based systems are nonetheless susceptible to network-based attacks. μMIP aims to maintain the integrity of a microkernel system by restricting the effect of a successful attack.

μMIP accomplishes this by associating integrity level, with all processes, tracking low-integrity operations throughout the system and limiting the privileges of all low-integrity processes. μMIP uses the tagging infrastructure to define and track integrity levels. In μMIP model, a tag represents an integrity level. Since μMIP uses tagging mechanism that performs all operation tracking from within the microkernel, no modification in the source code of user-space binaries (which includes applications, drivers, filesystems, network stacks, etc.) is necessary. Also, it is easy to configure μMIP.

Integrity Levels

As in UMIP, μMIP defines two tags that represent integrity levels: high and low. These integrity levels define a process' capabilities and the trust given to it. A low-integrity flags a process as being potentially compromised. The propagation of integrity levels is a key mechanism in μMIP: a process' integrity value will drop from high to low whenever it performs a potentially insecure operation (see Figure 3.1). When a process' integrity value is low or is dropped to low, the process is prohibited from performing sensitive operations.

μMIP's uses *duplication mode* (Section 1.1.7 of Chapter 1) of tag propagation model to disseminate integrity level as the process communicates with other processes in the system.. While MAC models for monolithic kernel architectures must trust services such as filesystems or network stacks, because they are linked against (and essentially indistinguishable from) the rest of the kernel, we can perform integrity tracking on an individual basis for each of these services.

It is also important for μMIP to protect files, as they are also an entry vector for exploits. Malicious code can modify files to exploit vulnerabilities in any program that reads them, in a manner similar to a network-based attack. To prevent such attacks through file I/O, μMIP extends the tagging model to associate tags with files. A file tag represents the integrity level of the file. μMIP assigns an integrity level to files based on the access permissions and the process that creates the file. If a process of low integrity creates a file, then the file is of low integrity. I/O requests on files update the integrity level of the requesting process depending on the flow of information.

In the following two sections, we discuss how μMIP handles tracking of the integrity levels of processes and the files handled by them.

Process Integrity

Microkernels use message passing for virtually all functionality. Whenever a process wants to manipulate a file, it sends a request to the process responsible for handling that file, which we term a *server*. Therefore, each request to an file passes through a server process. A web browser, for instance, sends requests to the filesystem process to manipulate files, and the filesystem sends requests to the disk driver to manipulate logical blocks. In a microkernel, each of these components executes as a separate process, and μMIP performs integrity tracking and protection at this level.

The μMIP model divides servers into two categories as follows.

Boot-time Servers Boot-time servers are the servers responsible for initializing the system at boot-time such as initial disk management. Boot-time servers terminate after completing the initialization. The μMIP model trusts boot-time servers as they are part of the boot process and must be included in the immutable system image. For example, the boot-time filesystem server is responsible for mounting disks and loading any files required for the boot process. The boot-time filesystem server assigns an integrity level to each file according to its access permissions. After the boot process finishes, this boot-time filesystem server terminates. Doing so protects the system against attacks where, for instance, the boot-time filesystem server is compromised and is used to compromise the initialization files. We assume that at boot-time, the system is not susceptible to network-based attacks. This can be ensured by simply disabling networking functionality till after the booting process.

Run-time Servers After the boot process finishes, dynamic integrity tracking and protection becomes active. All run-time servers, except for those that implement network stacks, are of high integrity by default. It is not until they receive data from the network or another low-integrity source that their integrity is lowered.

One problem specific to filesystem servers is that if all of them have their integrity levels lowered it will become impossible to write to any high-integrity file. To overcome this limitation, μMIP allows any process to instantiate servers to mediate access to files. Low-integrity processes are restricted to initializing only low-integrity servers.

If at any given time the system has multiple servers with different integrity levels, the kernel redirects requests to the appropriate server based on the integrity level of the file being requested. Thus, the kernel protects high-integrity servers from having their integrity constantly lowered by serving requests from low-integrity processes. This can be seen as

an exception to the rule expressed by the state-machine in Figure 3.1. Reception of IPC from a low integrity process lowers the receiver's integrity level.

File Integrity Tracking

In a microkernel architecture, each server, including the filesystem server, enforces discretionary access control (DAC) on the objects it handles. Henceforth, we focus on files as filesystem server objects. We point out, however, that the concept behind file integrity tracking applies to other server/object relationships as well, because like filesystems, all servers handle requests to a certain mountpoint through standard client APIs. For example, a serial port may be accessed by opening /dev/ser1 through the standard open() call.

μMIP generates integrity information based on DAC permissions. This integrity mapping is only done for system files, however; user files are all treated as low-integrity from boot time. This is to avoid situations where a user becomes unable to write to her own files because all of his applications use the network at some point during their execution.

Table 3.1 summarizes the mapping of DAC permissions to integrity levels for system files. As shown in Table 3.1, μMIP marks a file as low integrity only if the file is world-writable. In μMIP, all high-integrity files have limited DAC permissions, i.e., read and write access is not allowed to everyone. This predefinition of integrity levels takes the burden of manually setting them off the system administrator.

Table 3.2 indicates the different relationships between processes and files depending on their respective integrity levels. The first three columns describe the integrity of the requesting process at the time of the request, the current integrity of the file, and the requested operation. The second to last column shows whether μMIP permits the operation. The last column shows the resulting integrity level of the requesting process.

μMIP ensures the integrity of files even when their server is compromised. μMIP maintains integrity information within the kernel, so a compromised server cannot access it. The kernel is responsible for controlling file access based on integrity level. If all filesystem servers are compromised, the kernel would prevent the attacker from writing to any high-integrity file, because their integrity level of the filesystem servers would have dropped as per the integrity propagation rules listed in Table 3.2. This approach may seem somewhat heavy-weight, and at odds with the mindset of a microkernel. Somewhat surprisingly, however, we are able to do this without significant overhead (see Chapter 6).

To make μMIP realistic, we introduce two exceptions to the integrity propagation rules. These exceptions are ambivalent operations as they leave the integrity of the requesting

DAC Permission	File Integrity
Limited Read	High Integrity
Limited Write	High Integrity
World Writeable	Low Integrity
World Readable	Low Integrity

Table 3.1: Relationship between DAC permissions and file integrity for administrator-owned files

process intact regardless whether it reads a high or low-integrity file. The first exception is a read request for a high-integrity file from a low-integrity process. With μMIP, we redirect such a *read* request to the high-integrity filesystem server. The high-integrity filesystem server retains its high integrity level even after receiving the IO_READ IPC from a low-integrity client. This is because only a high-integrity filesystem should have access to high-integrity file metadata, and reading the high-integrity file does not compromise the filesystem.

The second exception is a write request to a low-integrity file from a high-integrity process. In this case, μMIP does not lower the integrity level of the high-integrity process as a consequence of the IO_WRITE IPC message to the low-integrity filesystem server. This exception is needed because writing to a file does not compromise the process that does the write.

Process Integrity	File Integrity	Operation	μMIP Access	Post Process Integrity
High	High	Read	Allowed	High
		Write	Allowed	High
	Low	Read	Allowed	Low
		Write	Allowed	High
Low	High	Read	Allowed	Low
		Write	Not allowed	Low
	Low	Read	Allowed	Low
		Write	Allowed	Low

Table 3.2: μMIP file access permissions and integrity

To summarize, the kernel performs the following integrity checks/changes with each file operation, depending on the integrity relation:

- **Equal integrity level:** If the process and the file have the same integrity level, then the kernel simply mediates the request and allows the client process to operate on the file.

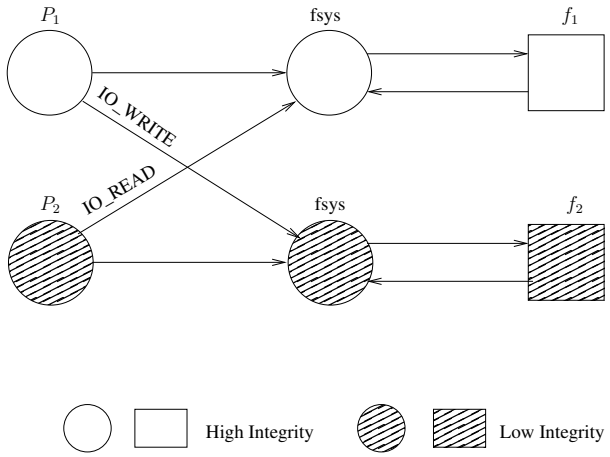

Figure 3.2: Tracking integrity propagation to files

- **Low-integrity process accessing a high-integrity file:** In this scenario, the kernel does not permit the process to modify the file unless the operation is a read.
- **High-integrity process accessing a low-integrity file:** In this case, the high-integrity process can write to a low-integrity file without lowering its integrity level. For all other operations, the kernel lowers the integrity of the process from high to low.
- **Ambivalent operations:** μMIP does not restrict a low integrity process from reading a high integrity file. A high-integrity process can also write to low-integrity files without having its integrity level lowered.

Example 3 (Tracking integrity propagation to files) *Figure 3.2 shows the integrity level propagation mechanism to and from a file. Processes P_1 and P_2 are shown as circles, Files f_1 and f_2 are shown as rectangles. Arrows indicate the flow of information and shading indicates the low-integrity level.*

As soon as process P_1 reads from the low-integrity file f_2, μMIP drops P_1's integrity level to low. The low-integrity process P_2 cannot write to the high-integrity file f_1.

Figure 3.2 also illustrates integrity ambivalent operations. An IO_READ operation by process P_2, a low-integrity process, to file f_1, a high-integrity file, does not affect the integrity levels of either the process or the file. This ambivalence also applies to the IO_WRITE operation from process P_1, a high-integrity process, to the file f_2, a low-integrity file.

Even if an attacker tries to exploit a vulnerability in P_1 through f_2, μMIP protects the system's integrity by lowering process P_1's integrity as soon as the process reads from f_2 and thereafter restricts the privileges of process P_1. This mechanism identifies possibly compromised files and the processes whose faults could be exploited by these files.

Integrity Propagation Rules

To summarize, the μMIP integrity propagation rules are the following:

- The kernel, as the only trusted entity, propagates integrity from a process to process along with IPC;
- When a process requests to open a file, the kernel redirects the request to the appropriate server according to the file's integrity level;
- When a process receives data from a low-integrity source—be it a file, the network or another process—its integrity is lowered.

For example, consider two high-integrity processes bash and adduser and a low-integrity file userlist.txt. Bash executes a script that opens the user list then feeds it into adduser. The kernel will lower bash's integrity when it reads from the file and bash will, in turn, spawn low-integrity adduser processes. These propagation rules also contribute to making μMIP more usable, as they require no administrator input.

Restrictions on Low-integrity Processes

Once integrity values are in place, μMIP restricts the actions of low-integrity processes at runtime. In a microkernel, every operation results in a message pass from one process to another through the kernel. μMIP relies on message passing, since it limits the effect of a successful attack by restricting the type of messages a low-integrity process can send.

Table 3.3 shows a list of some of the sensitive operations that μMIP restricts low-integrity processes from performing. Whenever one of these forbidden operations happens, an error code is returned to the calling process, allowing developers to detect and correct their program's behaviour. The kernel differently handles each of these operations depending on their effect on the system's integrity.

Take for example, the chmod() libc function. This call is used to change a file's permissions. In QNX Neutrino, a commercial microkernel [63], when a process calls the chmod()

function, it sends a message of type IO_CHMOD to the filesystem. Since the μMIP model prevents a low-integrity process from changing the permissions of a high-integrity file, the kernel will check the integrity of the requesting process. If the process is marked as low integrity, the message will simply be dropped within the kernel and a log entry will be generated. In the case of the IO_LOCK message, a similar rationale applies. However, both high and low-integrity processes are allowed to use IO_SPAWN. μMIP will only force the spawned process to inherit the integrity of the caller.

Operation	Message Type
Spawn a new process	PROC_SPAWN
Write to a file/device	IO_WRITE
Lock a file/device	IO_LOCK
Configure the path of a file/device	IO_PATHCONF
Change permissions of a file/device	IO_CHMOD

Table 3.3: Sensitive operations in QNX Neutrino and their message types

μMIP Example

To illustrate the full functionality of μMIP, Figure 3.3 shows a sequence of operations, their handling by the kernel and their consequences on processes' integrity levels. Again, processes are shown as circles, files are shown as rectangles, shading represents a low integrity value and arrows indicate the flow of information. The timeline of the example is as follows:

Example 4 (Integrity with Process Pooling) *Take the example given in the introduction, where a faulty tftp implementation is exploited into writing over* /etc/shadow*. In μMIP, all of tftp's requests to open, read and write to shadow are subject to approval by the kernel. In this example, the filesystem server R_1 has registered to handle all IO requests under the* /etc/ *directory. It has three instances: server processes RP_1 through RP_3. As a result of the* open("/etc/shadow") *request from compromised tftp, the kernel will check the integrity value of that file. Since it has a high integrity value, the kernel forwards the request to the one of the high-integrity servers RP_2 or RP_3. Any read calls from tftp will also be forwarded to that process. If tftp were to write to that file, however, the request would be dropped, protecting the high-integrity file from corruption.*

If an attacker gains control of tftp*, μMIP protects the system in two ways: First, the mechanism of resource manager assignment prevents the contamination of all high-integrity resource managers. This also prevents the system from entering a denial-of-service*

state where high-integrity objects become inaccessible. Second, the μMIP model prevents tftp *from writing to system-sensitive files marked as high integrity. In this way, μMIP guarantees the integrity of the data.*

Consider also the adduser process as shown in Figure 3.3. If it were to open the shadow file, it would be forwarded by the kernel to one of the high-integrity servers, exactly like tftp was. However, when it writes to the high-integrity file, its request would be allowed since it is also of high integrity. If it were to open and read from the low-integrity file /etc/profile*, however, its requests would be forwarded to RP1 and its integrity value would be lowered.*

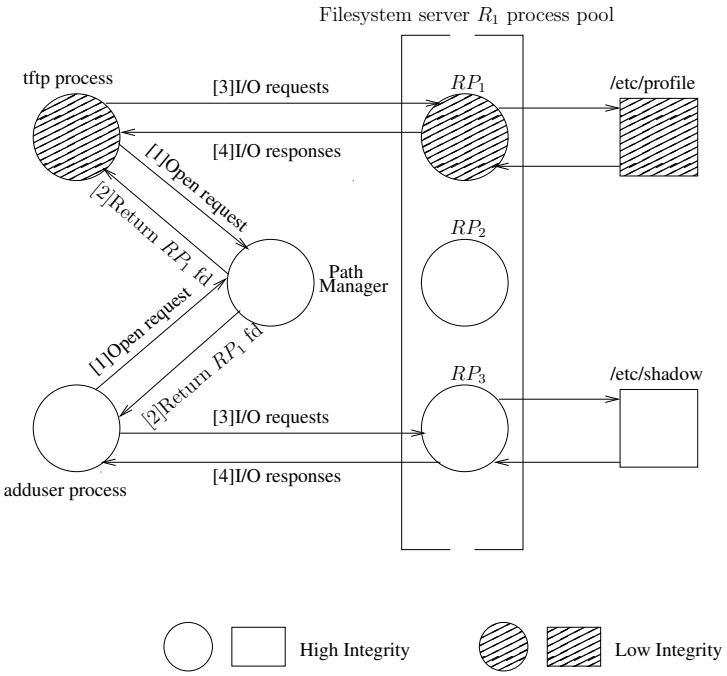

Figure 3.3: μMIP example

It is very important to notice that integrity checks happen in addition to, not instead of, normal DAC that the filesystems implement. In the example above, if tftp was executed under a non-root user, it would be kept from reading */etc/shadow* regardless of integrity value. μMIP's integrity protection relates only to operations that would already normally be allowed by ordinary filesystem access control.

3.4 Discussion

Extension of microkernel architecture: The design of μMIP applies to the microkernel based systems. Filesystem manager in a microkernel system runs as an external process and is responsible for enforcing access rules. In our proposed integrity protection model, the kernel is responsible for enforcing the access control mechanism based on the integrity levels of the file and the process as discussed in section 3.3.1. The μMIP model extends the microkernel to maintain the integrity information of all the files in the kernel as describe in Section 5.2 of Chapter 3.

μMIP restricts the operations of a low integrity process based on the message types. In order to enforce these restrictions, μMIP based kernel should have the information of specific message types. An example of such restriction is a low integrity process is not allowed to change the permission of any file in the system. The kernel enforces this rule by checking the integrity level of requesting process on all messages of IO_CHMOD type. Section 3.3.1 discusses these scenarios in detail.

Extension in the tagging infrastructure: In μMIP model a tag represents an integrity level which μMIP model assigns to the processes (rather than thread). μMIP extends the tagging model to support the assignment of tags to processes. μMIP model also extends the basic tagging infrastructure to associate tags with files.

Multiple Levels of Integrity: Currently μMIP provides two different labels i.e., high integrity and low integrity. The binary choice of labels adds to the simplicity and performance of the μMIP model. μMIP implementation can easily be extended to contain multiple levels of integrity. We can use multiple bits, in the tag bitmap, to represent an integrity level. However, manipulation of multiple integrity levels on every message pass might incur performance overhead. Multiple integrity levels will also add complexity to the μMIP model.

3.5 Summary

We have discussed our design of μMIP, our adaptation of a particular approach from prior work called UMIP, for mandatory security. Our focus is microkernel-based systems. We have discussed our implementation of μMIP in the widely-deployed QNX Neutrino commercial microkernel-based system. Our work is strongly motivated by the observation that such microkernel-based systems have been shown to be susceptible to the same kinds of vulnerabilities as their monolithic counterparts. We have discussed the trade-offs inherent

to μMIP, and the increase in size in the microkernel's code-base that results from it. We have also presented empirical results for the overhead imposed by μMIP across three well-established benchmarks for POSIX-compliant systems. We observe that μMIP offers its protection with only a very small runtime overhead.

There is tremendous scope for future work. One aspect is to investigate approaches other than UMIP for microkernels. As we assert in this work, it is likely that other approaches will also be feasible as the underlying mechanisms to realize them are the same as for UMIP. We also seek to refine our trust assumptions regarding servers, particularly the high integrity filesystem servers. It is quite possible that we can combine our approach with an approach such as privilege separation [120] for a more robust system. Yet another avenue for future work is a long-term study from deployments of μMIP in QNX Neutrino. Only such a study can fully validate that the approach is useful, and does not significantly impact usability in realistic settings. To carry out such work, we will have to build meaningful probes that coexist with μMIP to collect data.

Chapter 4

Use Case: Assertions on Process Interaction Sessions

In a modern system, an application can be abstracted as a number of interacting components. In such systems, the correct operation of individual components does not guarantee the correctness of the system as a whole. A system's correctness also depends on the pattern of its component's interaction. An unexpected interaction among components might lead to a incorrect results, a system failure or a security threat. Testing and verification of a system, in the presence of complex interactions among custom and third party components, can be difficult.

The need is clear for a mechanism to ensure that component interaction patterns comply with developer's intent at run-time. This chapter demonstrates the utility of tagging for the tracking and verification of a component's interaction pattern at run-time. The tagging infrastructure provides a *lifeline* mechanism for the efficient maintenance and manipulation of component's interaction history. The proposed framework builds upon the *lifeline* mechanism and supports assertions on interaction history among system components. Subsequently, our tool chain enables developers to program assertions on interaction history written in Linear Temporal Logic (LTL). The LTL statements can incorporate inter-component interaction behaviour.

The chapter is structured as follows: Section 4.1 provides an overview of assertions and our proposed assertion mechanism. Section 4.2 provides a guiding example and demonstrates the utility of intersert(). Section 4.3 provides an overview of our infrastructure from the developers' perspective. Section 4.5 details the runtime support used for intersert(). Section 4.6 applies the *Intersert* infrastructure to a commonly found use case in a

safety-critical systems. Sections 7.1 and 7.2 discusses the framework and summarizes the chapter.

4.1 Overview

Assertions are a widely used method for increasing program reliability and enhancing debugging as they permit checking program state against a specified statement at run time. Developers use assertions to check whether their assumptions about the state of a program are true at the moment an assertion is executed. This concept is useful for achieving a variety of goals [80] including testing software correctness, detecting software defects, and isolating fault.

Assertions operate on program state information, which is commonly encoded in global and local variables. Developers write assertions based on these variables to verify that the application is following its expected behaviour. For example, an assertion can check that the size of an input buffer is sufficient to contain data required by an application.

An important type of information that traditional assertions do not operate on is the history of interaction between threads (or single-threaded processes). Interaction properties a developer might want to assert include, for example, event ordering such as whether Process A communicated with Process B before opening a file or that a pair of redundant sensors have both been read before their values are used in calculations.

In contrast to assertions on program state, assertions on interaction history require supporting infrastructure. Often it is left to the developer to create and manage this infrastructure, and to developing a state-based checking mechanism for the specific interaction properties to be asserted. This additional development effort incurs cost and introduces another source for defects.

Adaptations of traditional assertions to operate on interaction history are already used in modern software. The Canadian Darlington nuclear power station uses interaction history to verify at run time whether a particular set of actions has occurred, and whether it has occurred in a particular sequence [113]. The Apache Portable Runtime (APR) uses assertions to prevent threads in thread-pools from interacting with task abstractions which have already been taken by other threads. In fact, any standard concurrency problem such as producer-consumer, reader-writer, or the barber shop problem can utilize assertions on interaction history. For example, in the barber shop problem [136], interaction history assertions could ensure that customers only interact with chairs which are empty, or that the barber only interacts with customers which are in chairs. Finally, applications such

as the GNU C Compiler (GCC) and the Linux kernel use assertions on specific program interaction with manually coded infrastructure to support them. For example, GCC has 41 assert statements that the authors are aware of which check whether a particular action was completed prior to executing the current action.

We propose that, instead of requiring the programmer to implement infrastructure to record interaction history on a case-by-case basis, the operating system should provide support for recording these interactions and verifying assertions about them. Leaving developers with the task of developing and maintaining this infrastructure counteracts their original intention, which is to increase reliability and provide debugging support. This is because a hand-coded support infrastructure, like all hand-coded software, is prone to defects [116]. However, it is unclear how the operating system should solve the problem of maintaining the interaction history, in operating system, with negligible runtime overhead, how to use it in assertions, and how to minimize the runtime overhead.

This work introduces *Intersert*, which is an infrastructure for programming assertions on interaction history of threads. The assertions contain Linear Temporal Logic (LTL) statements [118] placed the interaction history. This work shows that the *lifeline* (as described in Section 1.1.9 of Chapter 1) mechanism can provide interaction history at run time with negligible overhead and that this information is particularly useful in combination with assertions. In the context of the use we will refer to *lifeline* as *interaction history*. The contributions of this work are demonstrating: (1) the utilization of *lifeline* to expose interaction history for placing assertions, (2) the applicability of LTL for checking properties of the interaction history at the programming level, and (3) the feasibility of providing this functionality with negligible overhead at run time in a fully working tool chain on a commercial platform used for safety-critical applications.

4.2 Assertions on Interaction History

To illustrate the use of assertions on interaction history, consider the example application shown in Figure 4.1. This application, composed of 6 threads, periodically takes sensor readings in Thread A. Each sensor reading is processed by a branched pipeline of threads until, eventually, Thread E returns the final validated reading. The branch in Thread B can route the readings through either Thread C or C' based, for instance, on the reading quality. Thread D can return the reading to Thread B for further processing, or forward it to Thread E for finalization.

The developer intends the application to obey the following rules: (1) any interaction must be initiated by Thread A, (2) if a reading passes through Threads C, then the reading

is validated and Thread C must pass it on to Thread D which must forward it immediately to Thread E, and (3) if a reading passes through Thread C', then Thread D must return the data to Thread B for further processing. To verify the proper operation of the system, the developer can use LTL assertions on interaction history. As the data passes through the threads, the system records the interaction history. Let the propositional variables A through E be true, if their corresponding threads are present in the interaction history. The following assertions can then be used to capture the desired behaviour of the system:

- **Check that Thread A initiates all interactions.** With this assertion, the developer ensures that Thread A initiates all interaction chains. This assertion could be checked in any thread. To check this in Thread E, the assertion statement can state this LTL property as "A R E". R is the LTL "release" operator. It ensures that, if Thread A is absent from the interaction history prior to Thread E receiving the data, then the assertion will fail.

- **Check that a reading which reaches Thread C' has not yet passed through Thread C.** With this assertion, the developer ensures that a validated reading is not returned for further processing. This assertion is checked in Thread C' by "¬C". This statement means that if Thread C is present in the interaction history when Thread C' receives the reading, then this assertion will fail, alerting the developer of an unintended sequence of events.

- **Check that Thread D only initiates interaction with Threads B or E.** With this assertion, the developer ensures that Thread D only starts interaction as intended. This assertion is checked in Thread E by "D $\rightarrow$ XB $\vee$ XE". X is the LTL "next" operator. This assertion states that if Thread D participates in the interaction, then the only threads that can follow it are Threads B and E.

Consider the following interaction history in sequence: [A, B, C, D, E]. All the assertions described earlier would pass when applied to this history. The same is true for the following sequence: [A, B, C', D, B, C, D, E]. Another history sequence containing [A, B, C, D, B, C'] will fail the second assertion when checked in C' as the propositional variable for C will resolve to true while the assertion ¬C will resolve to false.

This simple example demonstrates the utility of placing assertions on interaction history. The challenge is how to realize this functionality in a user-friendly way, that hides as much complexity from the user as possible, and at the same time has low overhead at run time. Our framework, called *Intersert*, demonstrates how these goals can be achieved.

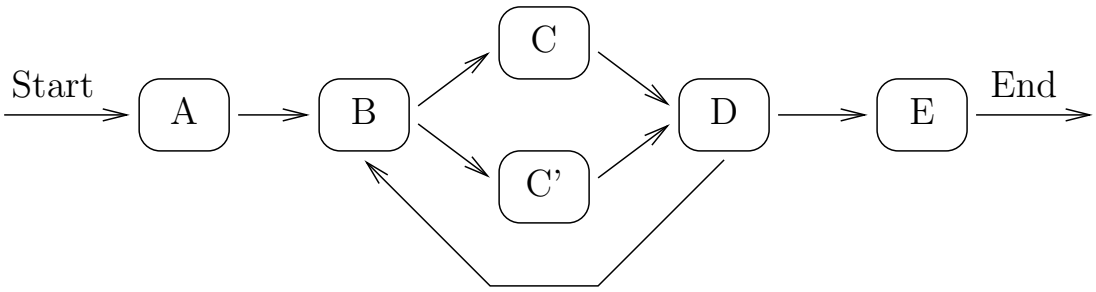

Figure 4.1: Interaction diagram of the example application.

4.3 Intersert

Intersert provides a variation on the standard C assert function, referred to as intersert(). Developers can use intersert() statements to verify properties in the form of an LTL statement on the history of interaction among threads (and single-threaded processes). *Intersert* uses LTL syntax in intersert() statements, because LTL has been proven to be a good choice to specify properties to be checked on sequences of events [18, 118].

Currently, the *Intersert* framework supports assertions on the interaction history between threads (and processes for single-threaded applications). An interaction simply indicates inter-process (or inter-thread) message passing.

Lifeline mechanism, as described in Section 1.1.9 of Chapter 1, supports the recording of interaction history in the operating system. An interaction history is a list of threads in the sequence in which they communicated in one continuous interaction. For example, consider a system with three threads, T_1, T_2, and T_3. If T_1 interacts with T_3, which subsequently interacts with T_2, then the interaction history will consist of the following entries: [T_1, T_3, T_2]. The interaction history is similar to a list of participating threads in a UML sequence diagram [112].

The *Intersert* framework uses standard LTL syntax in intersert() statements. Propositional variables in these statements represent the presence of a thread in the interaction history. For example, intersert("T_1 -> X T_3") will verify that T_3 directly follows T_1 in the interaction history; in other words, T_1 will directly interact with T_3. The violation of an LTL property results in the termination of the program similarly to standard C assertions.

4.3.1 The Developers' Perspective

From a developer's perspective, the use of intersert() statements is very similar to regular assert() statements. The developer writes intersert() statements inside C functions. Our tool chain processes the intersert() statements during the compilation stage and automatically generates the necessary infrastructure for checking whether they hold. However, there are three differences between intersert() and regular assert() statements: (1) to be available as propositional variables, threads must register themselves in a lookup directory, (2) intersert() statements use LTL syntax, and (3) intersert() statements operate on a finite *interaction session* (interaction sessions are detailed in Section 4.3.2).

The lookup directory enables developers to refer to specific threads in the propositional logic portion of their intersert() statements. For example, to give a specific identifier to the threads in Figure 4.1, one would use the fill_id() function. When fill_id("A") is called at run time, the calling thread will become associated with the label "A". Thus, each time this thread participates in an interaction, the system will record an "A" in the interaction history. Note that our framework currently prohibits delisting a thread or reusing a propositional variable, which is a minor limitation for safety-critical systems.

Listing 4.1 shows sections of code for the example presented in Figure 4.1. The example excludes the code for Threads C and D. Line 31 shows the first assertion of our example application: intersert("A R E"). When checking this assertion, Thread E will be present in the interaction history, as the intersert() happens immediately after receiving a message. If Thread A is present in the interaction history, the assertion will pass; otherwise, the assertion will fail and the program will be halted. Line 23 shows the second assertion, that verifies that a reading is not returned through Thread C′: intersert("!C"). This assertion will hold only if Thread C is absent in the interaction history prior to calling Thread C′.

```
1 void A () {
    fill_id("A");
3   start_session();
    data = read_sensor();
5   msg_send(B, data);
  }
7
  void B () {
9   fill_id("B");
    ...
11  msg_receive(&data);

13  if(is_valid(data))
        msg_send(C, data);
15  else
        msg_send(Cprime, data);
17 }
```

```
19 void Cprime () {
     fill_id("Cprime");
21   msg_receive(&data);
     ...
23   intersert("!C");
     ...
25   msg_send(D, data)
   }
27
   void E () {
29   fill_id("E");
     msg_receive(&data);
31   intersert("A R E");
     intersert("D --> X B | X E");
33   ...
     end_session();
35 }
```

Listing 4.1: Example usage of the intersert() statement.

4.3.2 Interaction Sessions

Interaction sessions are periods of execution during which interaction history is recorded. When a session is started, a new *baton* with a unique identifier symbolizes the new session. This baton is passed along with every interaction between threads. Upon receiving the baton, a thread automatically adds itself to the interaction history. Calling end_session() will destroy the specified baton with its associated interaction history. This mechanism precludes the need for branching in interaction history, guaranteeing linearity in the sequence of threads that take part in it. *Duplication* propagation mode can be used for branching in the interaction history. The branching in the interaction history will introduce complexity and other challenges. On the other hand, *baton* propagation mode simplifies the expression of properties, while still allowing a wide range of properties to be verified.

Interaction sessions and their batons allow the developer to (1) define the boundaries for the history sequence on which properties are checked, and (2) define multiple, concurrent interaction histories to be maintained independently. Limiting history is important for practical concerns; as the time for verifying properties depends on the length of history being analyzed, it is of interest to the developer to keep history as short as possible, while still permitting to verify the properties of interest. Permitting concurrent sessions is important as periodic interactions (such as the ones initiated by Thread A in Figure 4.1) require properties to be verifiable on each separate interaction history.

In the example presented in Listing 4.1, a new session starts each time Thread A takes a reading from the sensor (Line 3). Intuitively, this means that multiple sensor readings pass through the system at any given time, and that properties are checked on a per-reading basis on separate interaction history sessions. After Thread E processed a reading, it will end the session (Line 34).

It is worth noting that, while a session must start at some point to start recording history information, it need not end; in this case, the system will continue recording more interaction history, and any intersert() call will verify properties over the entire history. This is useful in systems that enter a steady-state with no clear session boundaries, but where invariants such as "T_x is never present" are of interest. Infinite interaction history is obviously impossible to implement. Section 5.4 of Chapter 5, provides details on the circular buffer (*lifeline*) used to record history and our framework leaves it to the developer to ensure the proper configuration of the buffer size to guarantee proper and correct operation at run time.

4.4 Code Transformation

As a proof-of-concept, we have implemented a fully functioning *Intersert* framework comprising a pre-processor for intersert() statements and runtime support. The pre-processor phase consist of a tool chain that translates a C program with intersert() statements into ANSI-compliant C code. The tool chain performs the conversion in several phases that includes extraction of LTL properties, mapping of LTL propositions to list operations and generation of TGBA. The Section 5.3 of Chapter 5 provides the details of these phases.

4.5 Runtime Support

The runtime support for intersert() consists of two components: a history recording mechanism, and a property verification mechanism. The first component is responsible for the collection of the interaction sessions as described in Section 4.3.2. The tagging mechanism provides this support. The second component are the TGBA's produced by the code generator.

Recording Interaction Sessions: In a microkernel architecture, interaction between threads manifests as messages passed between them. To record interaction history for

intersert() statements, our runtime system snoops these messages and records their associated metadata. Snooping these interactions while keeping overhead low enough for use in embedded systems is a challenge. intersert() system uses baton mode of the tagging mechanism to snoop the interaction between system entities and consequently to record interaction history sessions.

To create a session, an application must call the start_session() function. Creating a session instantiates a new tag, uniquely identifying that session. The system then uses the *lifeline* mechanism(Section 1.1.9 of Chapter 1) of the tagging infrastructure to record that tag propagation through the system.

To build the history within the session, *lifeline* records the identity of threads as they pass the baton. *Lifeline* of a tag results in a comprehensive history of interactions from the beginning of the session. Due to practical memory limitations, *lifeline* is implemented as circular buffer with fixed length. The length of this buffer is configurable, and it should be sufficiently large to correctly verify all properties of interest. Additional refinements could provide a mechanism for handling session overflows (i.e., the history buffer is too small). Our current implementation assumes that the developer specifies an adequate buffer size.

Finally, sessions also have a termination point. A session will be terminated when the end_session() function is called. The number of concurrent sessions is also limited by number of supported tags, however, this can easily extended as the need arises. Similar to length of a *lifeline*, we assume that the developer specifies adequate limits.

Processing intersert() Statements: At run time, a call to intersert() retrieves the interaction history of the current session for a given process, passes it on to the appropriate TGBA for that assertion, and returns the truth value determined by the automaton. The verification mechanism processes the interaction history from the start, processing entries in the interaction history while traversing the automaton. The state of the TGBA after processing the last interaction entry dictates whether the assertion passes (true accepting state), fails (false accepting state), or is undecided (non-accepting state).

Our evaluation of the LTL properties specified in intersert() statements follows the approach presented in [18]. Since interaction history may not lead to a final decision whether the property holds or not, the authors suggest an additional third, undecided, state labeled as *'?'*. In this state, it is unknown whether the property will hold and thus no verdict is made. If an unknown state is left when a given session's list is traversed, then a warning will be issued indicating that the assertion is potentially unverifiable. For example, the "A $\rightarrow$ XB" property is unverifiable if the only entry in the interaction history is [A].

4.6 Case Study

We demonstrate the applicability of the *Intersert* framework by applying it to a real-world data acquisition and control case study. A common need in data acquisition for safety-critical systems is to eliminate potentially spurious data from sensors to prevent incorrect decisions. The typical approach to solve this problem is to use cross validation of measurements combined with filtering before making decisions. Note that this problem differs from the fault masking of sensors for which triple-modular redundancy would be more appropriate.

Our application is a temperature-based safety shutdown system. The purpose of this trigger is to obtain input from temperature sensors installed in two different locations within a host system. The temperature data serves as the basis for determining, whether the host system must be shut down or whether it may continue running. The events fed to the temperature trigger are the inputs from the two different temperature sensors. Each sensor generates events at a given frequency. The system first temporally orders the input events, filters them, cross-validates them, and the finally makes its decision. Usually the system uses redundancy and diversity in the computation platform to process the data in two parallel streams.

Figure 4.2 shows an abstract model for this application. Each node represents a process, and arrows between them indicate message passing. Processes E_1 and E_2 produce new measurements at arbitrary times, and communicate them to D_1 and D_2. D_1 and D_2 agree cooperatively upon the order in which the events occur and identify correlated events. Once in agreement, D_1 and D_2 separately pass on the data to the filter processes S_1 and S_2. Note that D_1 passes on the measurements originating in E_1 while D_2 passes on the measurements from E_2. S_1 and S_2 filter the data to remove spurious events and performs other data transformations. The V_1 and V_2 processes cross validate the data and confirm with each other that the results were calculated across the same set of input events. When V_1 and V_2 have confirmed matching event sets, they pass on their results to O_1 which decides on the course of action.

The following contains a set of example intersert() statements useful for this application:

1: $E_1 \wedge XD_1 \rightarrow XD_2$ 2: $E_2 \wedge XD_2 \rightarrow XD_1$
3: $V_1 \wedge V_2 \rightarrow XO_1$ 4: $S_1 \rightarrow XV_1$
5: $D_1 \wedge XD_2 \rightarrow XS_1$ 6: $D_2 \wedge XD_1 \rightarrow XS_2$
7: $E_1 \rightarrow XD_1 \vee XD_2$ 8: $E_2 \rightarrow XD_1 \vee XD_2$

intersert$_1$ in S_1 and S_2 checks that if data is produced by E_1 and D_1 accesses this data, then D_1 will communicate this data to D_2 to agree on correlated events. Similarly, intersert$_2$

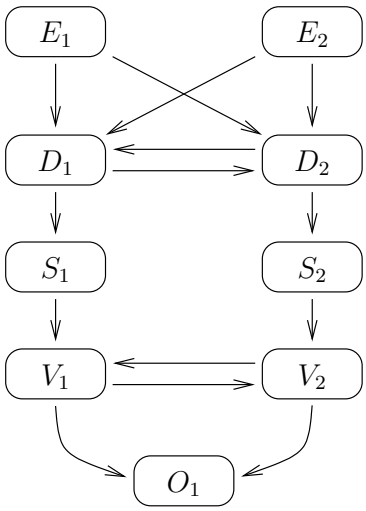

Figure 4.2: Process interaction in the case study.

checks the same property with E_2 as the data source. intersert$_3$ in O_1 checks that when V_1 and V_2 agree (i.e., the session continues), then O_1 will be the next process receiving data. intersert$_4$ in V_1 checks the integrity of the interaction between S_1 and V_1. intersert$_5$ in S_1 checks that if D_1 receives the data first and D_2 agreed, then S_1 will filter the data. intersert$_6$ in S_2 checks a similar interaction to intersert$_4$, in the case D_2 receives the data first. intersert$_7$ and intersert$_8$ in S_1 and S_2 check that data can be received by either D_1 or D_2.

4.7 Discussion

This work lead to several interesting observations about checking interaction history, assertions, and the intersert() infrastructure. Furthermore, this section also discusses potential mistakenly perceived limitations of intersert().

The current implementation only supports a limited number of concurrent sessions and a finite number of entries in the interaction history. The *Intersert* infrastructure uses tags from the *tagging* framework to represent its interaction sessions. In this framework, tags are encoded as bit-field metadata in the thread control block. The width of the bit field bounds the number of concurrently active tags and thus the number of concurrent sessions. This, however, is merely a perceived limitation, as the developer can trivially widen the bit field and thus increase the number of concurrent sessions. The same holds

for the maximum length of the interaction history per session. To increase the length, the developer just needs to configure the size of the circular buffer storing the interaction history.

At the moment, the *Intersert* framework relies on the developer to specify the bounds on the number of concurrent sessions and the maximal history length. Automatic configuration of these parameters based on a high-level specification might be an interesting avenue for future work. Yet to this date our case studies and example have not resulted in this specific need as safety-critical systems are usually well understood prior to implementation and only a couple of kilobytes of memory already drastically increase the number of sessions and length of the history. In our implementation of *Intersert*, an additional tag incurs the overhead of 64x bytes, where x is the length of the history session. An additional entry in the session consumes 64 bytes of memory.

The concepts of the *Intersert* framework and intersert() statements are also applicable to single-threaded programs. Our implementation of intersert() statements only uses interaction between processes and threads, however this can easily be extended. For instance, it is possible to use intersert() statements to check interaction history of messages passed in an object system such as Smalltalk's [53]. Furthermore, using aspect-oriented programming, a developer could weave a runtime support system similar to the one in the *Intersert* framework into regular applications. This would permit the developer to specify intersert() statements in single-threaded programs.

Currently our the *Intersert* framework does not permit assertions on resource interaction. Adding resource interaction, such as file access, to interaction history would allow for finer grained control over resource access. Each time a thread holding a session accessed a resource it would add that resource to the interaction history. Assertions on file access could be used for the shutdown of threads which attempt to access certain files.

Intersert only verifies the *safety properties*. Safety property ensures that 'something bad never happens". In contrast to the safety properties, *Intersert* framework does not support *liveness properties*, which states that 'something good eventually happens". Verifying only safety properties makes the system simpler and easier. Furthermore, most of the properties can be expressed a safety properties.

4.8 Summary

Program assertions are a common means for adding runtime checks to applications. This work explores the idea of using interaction history and session information in program

assertions. To accomplish this, this work presents the *Intersert* infrastructure, which supports LTL statements for checking interaction behaviour between processes and threads at run time.

The work resulted in a number of surprising insights and results: (1) placing assertions and interaction history is useful, (2) exposing interaction history can be achieved with negligible overhead, and (3) it is possible to push the complexity of this work into a toolchain that makes it easy for the developer to program such assertions.

The work's results open up several possible and interesting avenues for further work and Section 7.1 already highlights some of these. Others include extending the amount of history information used beyond interaction on the local host as well as synthesizing the interaction assertions from high-level specifications.

Chapter 5

Implementation

We have implemented our tagging mechanism in QNX operating system. QNX is a commercially available microkernel based operating system. Our choice of QNX is influenced by its true microkernel architecture where user and system level services interact through well defined message passing interface.

5.1 Basic Implementation

A global variable holds the a tag field which contains the set of all the tags present in the system. Each bit in the global tag field represents a single tag. The length of the global tag field puts an upper bound on the number of tags in the system. Currently the tagging mechanism supports 32 tags but it can be easily extended or reduced by changing the size of the global tag field. In addition to the global field, a global set of the "passable" tags defines the tag passable property. A set bit in passable tag set, marks the associated tag as passable.

Each thread in the system has a tag field similar to the global tag field. The thread specific tag field, defined in the thread control block, contains all the tags that the thread has received since its creation. In addition to the tag field, thread control blocks also define an additional bit to mark the active tag. An active tag represents the most recent tag received by the thread. The active tag is set automatically every time the thread receives a tag. Users can also explicitly change the active tag of a thread through an API.

A thread passes a message to another thread by using message passing routines. Depending on the structure of the message, QNX provides several message passing routines.

The most common routines are MessageSend(), MessageSendv() and MessageSendPulse(). These message routines pass a message payload to the receiving thread. The tagging module modifies these routines such that they pass both their original payload and the tag of the sending thread to the receiving thread. The basic thread passing mechanism is simply a logical OR operation between the active tag of the sending thread and the tag field of the receiving thread. Listing 1 shows the pseudocode of the tag passing routine.

Algorithm 1 Tag Passing Routine

```
if recv_thread == sys_thread then
  return
else
  if tag_ttl_val ≥ threshhold_ttl then
    return
  else if sender_tag & sender_tagterminator then
    return
  else
    if PROPAGATION_MODE == BATON then
      reciever_tag = sender_tag
      sender_tag = 0
    else
      reciever_tag ∨ sender_tag
    end if
    tag_ttl = tag_ttl + 1
  end if
end if
```

As shown in listing 1, the first condition prevents the propagation of the tag to the system thread. This condition simply checks if the receiving thread is a system thread by checking it's parent process ID (QNX identifies system threads by assigning a unique global ID of 1 to the parent system process). In case of a receiving system thread, the routine does not pass the tag and simply returns. The next condition imposes the TTL threshold. This condition will prevent further propagation of the tag if the TTL value has exceeded the TTL threshold. The next condition, as shown in listing 1, enforces the tag terminator mechanism. If the tag terminator field of the sending thread contains the current tag of the sending thread than this condition will not pass on the tag. At the end, depending on the propagation mode, the tag of the sending thread will propagate to the receiving thread. After the assignment of the sender's tag to the receiver, baton mode clears the sender's tag field by assigning it 0. Duplication differs only in that it does not clear the

sender's tag field. In case of successful propagation, the tag passing routine terminates by incrementing the TTL value.

5.2 μUMIP: Mandatory Security for Microkernel-based Systems

This section describes the implementation of mandatory security model, μmip. μmip described in detail in Chapter 3. The tagging model provides the basis for the implementation of μmip model. The implementation of μMIP was a matter of creating a high-integrity tag, whose presence in a thread indicates that it should be treated as high integrity as per μMIP rules; its absence indicates low integrity.

During system startup, μmip assigns high integrity tags to all the processes. An exception is the network stack (i.e., io-pkt-v4 in QNX), which starts with the low integrity. Because integrity value initialization and propagation is performed from within the kernel, it is entirely transparent to all software (such as drivers, filesystems and other operating system components) but the microkernel and therefore using μMIP requires no modification of application source code.

The basic tag propagation mechanism provides mandatory access control functionality. Tagging high and low-integrity processes in this manner provides the functionality defined in Section 3.3.1 of Chapter 3, but there tagging the files handled by those servers is also required. To accomplish this in QNX Neutrino, we modified the original path manager, which is also located inside the kernel.

Since all device drivers run as servers in user space, they are decoupled from the kernel. In a manner similar to Linux, device drivers use the kernel's path manager to create special files that allow clients to communicate with them. A serial port driver, for example, may ask the path manager to create a file called /dev/ser1. When an application or some other server needs to use that serial port, it does so by opening, reading and writing to this file. It is the path manager's responsibility to forward all operations made on the file to the appropriate server.

In our implementation of the μMIP, we modified Neutrino's path manager to perform the following operations:

Integrity Check: As soon as the kernel gets an IO_OPEN request from a process, it will look up the requested file in an internal integrity table. The file integrity table is a bitmap where each bit represents the integrity level of a particular file on the disk. High

integrity is indicated by a bit value of 1 and low as a value of 0. After lookup, the rules of Section 3.3.1 are applied. If the rules mandate it, the operation will be dropped. In the case of a dropped operation, the kernel will reply back to the client with an error. Algorithm 2 shows the pseudocode for the integrity check imposed by the path manager.

An integrity table is part of the path manager component of the kernel. We have implemented the internal integrity table as a hash table, solving collisions with linked lists. The worst case complexity of the lookup and insert operations is, therefore, O(n); however, it is much better in the average case.

Algorithm 2 Path Manager Integrity Check

```
if msg_type == IO_OPEN then
  {Get file integrity level from hash table}
  file_integrity ← get_file_integrity(filepath);
  if file_integrity == requesting_process_integrity then
    {Get the relevant server with same integrity as of file else returns NULL}
    rel_server ← lookupserver(filepath);
    if rel_server ≡ NULL then
      return ERROR
    else
      return rel_server
    end if
  else if file_integrity ≠ requesting_process_integrity then
    if file_integrity == LOW_INTEGRITY then
      {Lower the integrity level of the process}
      requesting_process_integrity ← LOW_INTEGRITY;
    else
      {File has higher integrity than requesting process}
      return ERROR
    end if
  else
    return ERROR
  end if
else
  return ERROR
end if
```

Resource Manager Instantiation: After the integrity check, if μMIP allows the file IO operation, the kernel will redirect the request to the appropriate resource manager.

This involves checking the integrity of the file, and choosing between servers of different integrity values if multiple exist. At this point, the kernel may still drop the request even if the integrity rules permit it, on the basis of the integrity level of the available servers. For example, the kernel will deny a low-integrity process from reading a low-integrity file if there are only high-integrity filesystem servers available. To work around this case, as discussed in Section 3.3.1, the client process can use the resmgr_attach() call to initialize a new resource manager of the appropriate integrity level.

Cryptography: After the client has been connected to the appropriate server, the kernel mediates all the I/O requests between them. To prevent unauthorized tampering by the compromised filesystem server, μMIP encodes the metadata of the file. The encoded metadata includes information like the address of the file, file name and amount of space on the storage media.

To elaborate more on the effectiveness of the cryptography consider a disk driver server. The disk driver provides the interface to the disk and does not require any mediator to write to and read from the disk. A compromised disk driver means that the attacker has complete control over the disk. To protect the disk content from the compromised disk driver μMIP uses cryptography, storing all information about files on the disk in encoded form. The attacker cannot get the meaningful data without obtaining the file encryption key from the kernel. However, cryptography does not prevent an attacker from writing garbage data or blindly deleting the contents of the disk.

We have implemented and tested μmipwith QNX Neutrino's embedded transaction filesystem (ETFS) [121]. We have used the AES encryption to encode the metadata of high-integrity files. Other works on filesystem encoding like VPFS [142] and I3FS [115] are complementary to μMIP. VPFS and I3FS can be used with μMIP to encode the entire contents of the disk.

5.3 Intersert: Assertions on Process Interaction Sessions

intersert() approach builds upon the tagging infrastructure. The *Intersert* framework uses *baton passing* semantics for tags. In this mode of operation, tags are propagated *without* duplication. The implementation of intersert() framework consist of two phases: a tool chain to convert intersert() functions into ANSI C code and runtime system to record, retrieve and verify interactions among components.

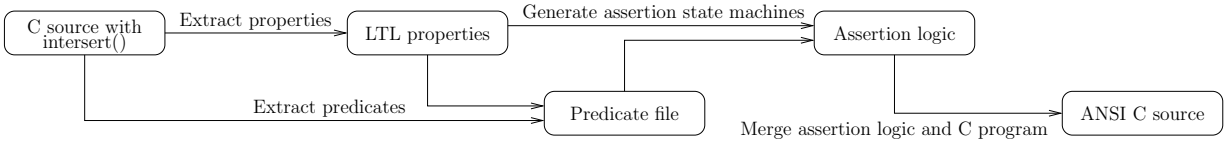

Figure 5.1: Converting C programs with an intersert() call into regular C programs.

5.3.1 Toolchain

Figure 5.1 shows the internals of our tool chain which converts intersert() statements into ANSI-compliant C code. Each node represents artifacts (e.g., the code and LTL data structures) at different stages in the tool chain. The transitions describe the associated transformation process, with arrows indicating the direction of the transformation.

LTL Operators	List Operation	Flow Assert Syntax
A ∧ B	Both tags for A and B exist in the list	intersert("A & B")
A ∨ B	Either A or B exists in the list	intersert("A — B")
A → XB	The node next to A must be B	intersert("A –¿ XB")
∼A	A does not exist in the list	intersert("∼A")
A R B	A exists before B in the list	intersert("A R B")

Table 5.1: Mapping of LTL operators to intersert() statements.

Table 5.1 summarizes the mapping of different LTL properties to intersert() statements. The LTL formulae enclosed by the intersert() statements can use any syntax accepted by the Spot library [91]. The Spot library supports various syntaxes for expressing LTL statements. The second column of the table, shows the semantics of the list operation for the given LTL property.

Each propositional variable in an LTL statement that occurs in an intersert() statement represents a thread. These symbols are linked to threads through fill_id() statements. Whether or not a symbol evaluates to true is determined through the use of list operations. If the symbol's associated thread is present in the current interaction session, then the symbol will evaluate to true.

Developers insert intersert() statements in their C source code and compile these programs with our *Intersert* tool chain. The first stage in the tool chain extracts the LTL properties from the source code and writes them into a property file. A simple script parses the source to extract statements containing intersert() calls. The tool chain then extracts

all propositional terms (i.e., the identifiers registered with fill_id()) used in the LTL expressions, saving them to a predicate file. The tool chain uses the spot library [91] parser for the extraction of propositional terms. This phase also checks the LTL properties for any syntax errors. After the extraction, this phase maps the LTL propositions to the list operation. An example of this mapping is shown in table 5.3.1.

A code generator module synthesizes state machines to check the intersert() LTL statements in C. Our approach uses a code generator based on LTL_3 tools [18] to generate the state machines. The code generator uses the files containing the predicates and LTL properties as input, and translates the properties into Transition-based Generalized Büchi Automata (TGBA). Each LTL property results in the generation of a separate TGBA. The TGBAs, which provide the support for assertion evaluation, are essentially a series of if statements and C data structures. The C data structure represent the states and the if statements mimics the transitions in the TGBA.

The resulting C code containing the TGBA logic is then merged into the original program, replacing the intersert() statement with a C assert() statements. This generated C assert() passes the current interaction history as a parameter to the appropriate TGBA, which evaluates the LTL property based on that history. Section 4.5 of Chapter 4 discusses the details of the runtime mechanism for processing intersert() statements.

The output from the various stages of the process of transforming intersert() statements is shown below. The output from the code generation phase is omitted for brevity reasons, and is instead summarized in a state transition diagram shown in Figure 5.2. The figure only shows all states that return *true* for the intersert() statement. All other transitions and states will return *false*. Note that the *Intersert* framework uses LTL_3 and Section 4.5 of Chapter 4 provides details why partially evaluated statements also return *true*.

Toolchain Stage	**Output**
LTL Statement	A → XB
intersert() statement	intersert("A → XB")
LTL Parser Output	a: `curnode = searchnode(A)` b: `isequal (curnode->next,B)`
TGBA Generator Output	See Figure 5.2

5.3.2 Runtime System

The basic tagging implementation, described in Section 5.1, provides tracking of interaction among components at runtime. The tag lifeline, as described in Section 5.4, maintains the

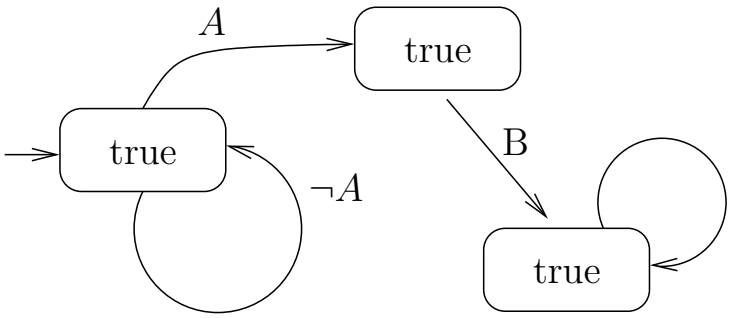

Figure 5.2: Code Generator output for A→XB. All other transitions lead to a state returning *false*.

interaction history. The tagging library implements the following functions to support the *Intersert* runtime system.

start_session() A developer calls start_session(), that creates a tag in baton propagation mode via CreateTagField() function call. The new tag uniquely identifies a session. As soon as start_session() creates a tag, the tag starts propagating while recording the receiving components in the lifeline.

end_session() A developer can call end_session() to end the tag propagation and stop recording the interaction history in the lifeline. Terminating a session removes the tag and associated lifeline history from the system by calling DeleteTagField().

fill_id() fill_id() maps the LTL predicates to the thread identifiers. fill_id() writes the mapping of LTL predicates to the thread identifier into the file. intersert() retrieves the predicate-to-thread mapping before running generated TGBA on the given LTL property.

intersert() The intersert() implementation is the core of the *Intersert* framework. intersert() retrieves and manipulates the lifeline of a tag. The lifeline of a tag provides the interaction history of components for a single session. After retrieval, intersert() translates the propositions of the given LTL property to the list operations as shown in table 5.1. The tagging library provides APIs to simplify the proposition translation to lifeline operations i.e., searchnode() and isequal(). Table 5.3.1 shows an example for such translation for the LTL property A → XB. Finally, intersert() runs the TGBA on a translated LTL property.

As described earlier, the TGBA is a sequence of if statements, representing transitions in the TGBA.

5.4 Lifeline

Our implementation of the tagging mechanism allows the user to log and timestamp the flow of tags through different threads in the system. We call this mechanism the *lifeline* of the tag [48]. The lifeline mechanism is built on the top of a tag's thread list. The presence of a thread in a tag's thread list, in addition to the associated timestamps, indicate the reception of that tag by that thread at that particular time. The user can access this information through a system call. Our current implementation registers the local wall time at the time of propagation, but for the distributed case a logical clock, vector clock, or matrix clock [37] would be required.

A circular buffer of predefined length implements the lifeline. Each entry of the circular buffer can contain IDs for the source and receiver with the time of reception. The length of the circular buffer and contents are configurable. When circular buffer overflows it starts overwriting the old entries, starting from the initial index.

5.5 Tagging Library

The user can create and control the behaviour of tag via command line utilities. The developer might want to access the tagging features from the application source to control and retrieve the tag information at runtime. At application level, tagging library provides an interface to the tagging module. The developer can use the following functions through tagging library:

- GetTagsField() gets the tag field for the calling thread.
- SetTagsField(char *tag) sets the calling thread's tag to tag.
- UnSetTagsField(char *tag) clears the tag tag from the calling thread.
- CreateTagsField(char *tag) creates tag as a new tag.
- DeleteTagsField(char *tag) deletes tag from the tag list.

- SetTagFieldtoPass(char *tag) marks tag as a passable tag.
- UnSetTagFieldtoPass(char *tag) marks tag as an impassable tag.
- SetActiveTagField(char *tag) marks tag as an active tag of the calling thread.
- LookupTag(char *tag) searches for tag in the tag list of the calling thread and returns the tag number associated with it.
- GetThreadsForTag(char *tag,struct tag_lifeline_node *temp) gets a list of all threads which own the tag tag.

All these functions use Neutrino's ThreadCtl() kernel call. It allows the user to access Neutrino-specific thread settings. The commands for each of the functions above are defined and passed to ThreadCtl() as its parameter. ThreadCtl() resolves the command and calls the kernel-level functions to manipulate tags at the kernel level.

Chapter 6

Performance Evaluation

6.1 Goal

Since the propagation of tags involves adding instructions to every message pass, it is imperative that the incurred overhead is minimal. The goal of the evaluation study is to compare the modified QNX operating system (with tagging) with the QNX baseline operating system under different system workload. The baseline operating system is the original operating system without tagging support.

6.2 Services and Outcomes

The evaluation study compares the performance of the QNX microkernel with and without tagging feature. QNX microkernel implements message passing services along with the core POSIX features listed below. QNX microkernel is a POSIX compliant operating system, so most of its services and outcomes of services should comply with the POSIX standards. Below is the brief description of the outcome of core services in QNX microkernel.

- Threads Management: The outcome of thread management should result in correct POSIX based thread operations that includes thread creation, thread scheduling etc.
- Inter Process Communication: The outcome of inter process communication should result in reliable information interchange between processes and threads.
  - Signals: A signal of supported length and type should be delivered from source thread to the receiver thread.

  - Pulses: A pulse of supported length and type should be delivered from source thread to the receiver thread.
  - Shared memory: The shared memory of one process should be accessible to other processes upon request.

- Clocks: All the clocks, maintained by the kernel, should output the correct time i.e., time of the day.
- Timers: All the timers, maintained by the kernel, should output the correct timing information i.e, real time counters etc.
- Interrupt handlers: The kernel should provide support for interrupt handlers according to the underlying hardware.and react to the interrupt request Proper registered interrupt service routine should be called upon hardware signal.
- Synchronization: As a result of synchronization mechanism, the kernel should control the access to particular objects depending on the synchronization technique.
  - Semaphores: The operating system should provide the correct semaphore functionality according to the POSIX document.
  - Mutual exclusion locks (Mutexes): The operating system should provide the correct mutex functionality according to the POSIX document.
  - Condition variables (condvars): The operating system should provide the correct condvar functionality according to the POSIX document.
  - Barriers: The operating system should provide the correct barrier functionality according to the POSIX document.

6.3 Performance Metric

Since tag propagation incurs overhead to every message pass and in a microkernel architecture even the simplest of libc calls causes a message to be emitted. To show that the tagging overhead is negligible, we choose a broad set of performance metrics focusing on the performance of system calls, libc calls, file operations and user-level applications. Following text briefly highlights and justifies the inclusion of all the performance metrics, for the evaluation of tagging system.

Execution speed of standard System Calls: Almost all the system calls in QNX operating system results in a message pass between a user thread and the system thread. Since the purpose of this evaluation is to study the overhead that tagging adds to each message pass, we have chosen the execution speed of system calls, under different workloads, as one of the performance metric.

Execution speed/Performance of library C calls: Like the system calls, most of the C library functions exchange messages with system thread to acquire/release different OS level resources and functionalities. Since the purpose of this evaluation is to study the overhead that tagging adds to each message pass, we have chosen the execution speed of library C function, under different workload, as one of the performance metric.

Execution speed/Performance of file operations: QNX implements filesystems in a server/client manner. The server process is the process responsible for specific paths in the filesystem. Any request, from the client process, to access the file will result in messages exchange between the server process and client process. Eventually, all file operations result in a message pass from client to server. Since the purpose of this evaluation is to study the overhead that tagging adds to each message pass, we have chosen the execution speed of standard file operations, under different workload, as one of the performance metric.

Execution speed/Performance of standard user application: Message passing is the core for most of the operations in QNX operating system. Since the purpose of this evaluation is to study the overhead that tagging adds to each message pass, we were interested to study the impact of tagging on some real world user level applications. The execution speed of the user application will help us understanding the overhead of tagging on standard applications.

6.4 Parameters

For the evaluation of the tagging mechanism, we ran several benchmark suites as describe in Section 6.7. All the benchmark suites provide a set of parameters to configure and control the behaviour of different benchmark tests. These parameters impact the performance metrics. We have characterized all the parameters as *System Parameters* and *Workload Parameters*.

Workload Parameters: Workload parameters define the load of the system at the time of evaluation. To ensure that the tagging overhead is minimal under different workloads, following parameters stress system calls, C library function calls and file operations under different workload. In addition to the measurement of performance, workload parameters also helps in proving the integrity of the results. For example, one can verify the integrity of results by verifying that the performance of the system decreases as the workload increases.

- Amount of Memory: Different memory intensive benchmarks in various benchmark suites uses the amount of memory as parameter e.g., malloc() benchmark, in libmicro benchmark suite, uses the amount of memory parameter to direct the size of the allocated memory.

The amount of memory has the huge impact on most of the benchmarks. Usually the large amount of memory degrades the performance.

- File size: Different file intensive benchmarks in various benchmark suites uses the size of file as parameter e.g., read() benchmark, in IOZone benchmark suite, uses the size of file as the parameter. The size of the file has the huge impact on the performance of the file operations. Usually the large size of file results in performance degradation.

- Sample size: The sample size represents the number of times we want to run a particular benchmark. The benchmark suites use the sample size parameter in there own contexts. For example, libmicro takes sample size as parameter that determines the number of runs for each benchmark in the suite. The sample size impacts the credibility of the measured data. The large sample size makes the data more reliable.

- Number of threads: Few of the benchmark use number of thread as parameter. This parameter directs the number of threads that will be used to execute the benchmark. For instance, libmicro uses number of threads as parameter to all the benchmarks included in the suite.In a multithreaded environment, number of threads can impact the performance results by running the different tasks concurrently. On the other hand, a bad multi thread design might lead to a worst performance.

- Number of processes: Few of the benchmarks use number of processes as parameter. This parameter directs the number of processes that will be used to execute the benchmark. For instance, libmicro uses number of processes as parameter to all the benchmarks included in the suite. Like threads, In a mutiprocess environment, number of processes can impact the performance results by running the different tasks concurrently.

- Number of synchronization objects: Different benchmarks stress the synch operation by increasing the number of synchronization objects. For example, libmicro increases the number of mutexes for mutual execution benchmark.

- Number of blocks: This parameter quantifies the number of blocks that file operation benchmarks use to measure the performance of the file operations e.g., IOZone uses this parameter for benchmarks focusing on file operations. Usually the number of file blocks has an inverse impact on the performance of the file operations i.e, larger number of blocks usually results in low performance.

- Size of buffer: This parameter represents the size of file buffers that different file operation benchmarks use to buffer file data. e.g., IOZone uses this parameter for benchmarks that measure the performance of the file operations. The size of file buffer can be effectively manage to improve the performance.

- Number of File descriptors: The performance of few file operations are sensitive to the number of file descriptors e.g., select operation on file.

- Record Length: Varying the record length introduces the variation in performance of filesystem operations. Due to its direct impact on performance of file operations we have included record length in the parameter list.

System Parameters System parameters direct the hardware and software resources. The following system parameters describe the resources and scheduling discipline included in the evaluation study of the tagging mechanism.

- Operating System: We ran all the benchmarks on QNX Neutrino 6.5 operating system with tagging and without tagging.
- Hardware: The underlying hardware has a deep impact on the performance evaluation. Faster hardware will yield better performance compare to the slower hardware. We ran all the benchmarks on a 1.8GHz and 3.2GHz Pentium 4 with 1GB of RAM.
- Scheduling Technique: The scheduling technique in operating system affects the execution time of different threads or processes. We have used priority based scheduling to schedule threads during the execution of benchmarks.
- Filesystem: The filesystem parameter indicate the type of filesystem used for the evaluation. The possible values for this parameter can be memory based filesystem, network based file system and disk based file system. The filesystem impacts the performance numbers as some filesystem performs file operations faster than the other e.g., memory based filesystem is faster than disk based filesystem.
- Compiler: Compiler translates the source code into machine level byte code. Different options supported by compilers can directly influence the performance of the generated machine code e.g., time optimization, space optimization etc. We have compiled the source of all the benchmarks and operating system kernel using the GNU C compiler with basic options set that excludes the optimization.

6.5 Factors

Most of the parameters, explained in the above Section 6.4, are variables that can take on different values. Since we want to measure the performance of the tagging mechanism under different workload, we have obtained the performance numbers by varying values of such parameters. We term the parameters with varied values during evaluation as *factors*. The number of possible values for the factors are called *levels*. Table 6.1 shows the list of all the factors with respective levels and possible values. For each factor, table 6.1 also indicates the respective benchmark test and benchmark suite. Table 6.1 only shows the varying parameters, all the parameters with constant values are not shown in the table.

Benchmark	Factor/Parameter	Benchmark Suite	Level	Values
memset	Amount of memory	Libmicro	5	1k,4k,10k,1m,10m
malloc	Amount of memory	Libmicro	5	1k,4k,10k,1m,10m
memcpy	Amount of memory	Libmicro	5	10,1k,10k,1m,10m
strcpy	Amount of memory	Libmicro	2	10,1k
strlen	Amount of memory	Libmicro	2	10,1k
strchr	Amount of memory	Libmicro	2	10,1k
scasecmp	Amount of memory	Libmicro	2	10,1k
read	Amount of memory	Libmicro	3	1k,10k,100k
write	Amount of memory	Libmicro	3	1k,10k,100k
pwrite	Amount of memory	Libmicro	3	1k,10k,100k
mmap	Amount of memory	Libmicro	2	8k,128k
unmmap	Amount of memory	Libmicro	2	8k,128k
mprot	Amount of memory	Libmicro	2	8k,128k
mutex	Number of threads	Libmicro	2	1,2
malloc	Number of threads	Libmicro	2	1,2
pthread	Number of threads	Libmicro	4	8,32,128,512
memset	Number of Processes	Libmicro	2	1,2
fork	Number of Processes	Libmicro	3	10,100,1000
File Copy	File buffer size	Unixbench	2	256,1024
File Copy	Number of blocks	Unixbench	2	500,2000
Select	Number of file descriptors	Lmbench	2	100,250
IOZone file operations	Record length	IOZone	12	4,8,16,32 8192
All benchmarks	Operating system	All suites	2	Tag, No tag
IOZone file operations	Filesystem	IOZone	2	ETFS,QNET

Table 6.1: Factors with their levels and corresponding values

6.6 Evaluation Technique

For the evaluation and comparison of the two techniques i.e., operating system with and without tagging, we have both the systems fully implemented. Having the luxury of implemented systems, we have used measurement technique to collect the performance data. Measuring technique generally provides the valid results.

6.7 Workload

To measure the overhead in terms of performance metrics, we have conducted six sets of benchmarks. Each benchmark suite focuses on one or more performance metrics. For example, IOZone focuses on performance of file operations whereas libmicro focuses on the performance of C library functions. Following is the brief description of each benchmark suite.

Standard OS Benchmark: OS benchmark programs measure the performance of the QNX kernel and its closest components including the C Library, the process manager and the path manager.

MiBench: MiBench suite [59] is an application-level benchmark, which serves to illustrate the effects that tagging has on the performance of real world applications. MiBench has been widely used in academia to evaluate the performance of processors and other software systems [27].

IOZone: IOZone is a filesystem benchmark with focus on file I/O operations. We configured IOZone [109] benchmark to measure the overhead that propagating tags over the network adds to QNET. QNET in QNX allows two QNX nodes to communicate transparently over network. Another run of IOZone is configured with memory based file system, *ETFS*, to gauge the filesystem overhead specifically. These operations range from simple reads to random reads, to mmap calls, etc. Put together, these different benchmark suites allow us to confidently evaluate the overhead of μMIP as implemented in QNX Neutrino.

lmbench and unixbench: The fourth and fifth sets of benchmarks are lmbench [84] and unixbench [95]. Both suites are designed to tax the most frequent operations in a POSIX system. lmbench and unixbench benchmark suites comprise different microbenchmarks, each focused on stressing a particular part of the system. These microbenchmarks might, for example, stress the memory read and write, creating/deleting files or forking processes.

Libmicro: libmicro [130] benchmark suite focuses on the C library operations. The libMicro benchmark set focuses on stressing system and library calls, and was created to compare the performance between Solaris and Linux. Since our implementation of intersert() is based on the QNX microkernel, every system call (such as `malloc()` or `write()`) results in the submission of a message from the application to the kernel, stressing our interaction tracking mechanism.

Pipebench: In addition to above benchmark suites, we collected data from the execution of pipebench [60] on both the original and the modified kernel. Pipebench stresses the message passing aspect of the system by generating fread() and fwrite() calls in a tight loop, while also collecting a measure of useful work, i.e., throughput.

6.8 Design Experiments

To measure the overhead caused by Tags, we executed all benchmarks with and without tagging enabled in the kernel, and also with the extension for *lifelines*. In every test shown here, the tag vector width is set to 32-bits, the word size for the architectures used. We ran Mibench, IOZone, lmbench, unixbench on QNX Neutrino 6.5, running on a 1.8GHz Pentium 4 with 1GB of RAM. Pentium 4 platform with 3.2Ghz Pentium 4 with 1GB of RAM was used for libmicro and pipebench. For Mibench and OS benchmarks, all the code (benchmark and kernel) was compiled without GCC optimizations to eliminate compiler interference on the results; indeed, compiler optimizations made the tagged benchmarks execute faster than their untagged counterparts. The unoptimized numbers shown here are, therefore, the worst case overhead we observed. We ran the all the benchmarks, except standard OS benchmark, libmicro and pipebench, without lifeline support to avoid modification of its original source code, and also because the internal benchmarks conducted prior to them showed no evidence of significant slowdown.

To measure the execution time of each run of the OS benchmarks, we used the ClockCycles() libc function. For all the other benchmarks, we used the clock_gettime() libc function, which allows measurements as precise as the system's free running counter. After each run of each benchmark, we recorded the execution time in a file for further processing. We analyzed the data using R 2.10.1.

To evaluate the overhead that distributed tagging incurs on QNET (QNX network protocol) communication, we executed the IOZone benchmark between two Pentium 4 machines, one mounting a remote directory exported by the other through QNET. IOZone is a filesystem benchmark with focus on file I/O operations. These operations include simple reads, random reads, strides, record rewrites, file rewrites, etc. Every file operation performed by IOZone in that setup lead to the propagation of tags between the two participating nodes, stressing message passing over the network. We collected statistics on a 10MB file with record sizes ranging from 4 to 8192 bytes resulting in more than 2.6 million individual measurements. Since tagging mechanism extends the file I/O operations to propagate and track the tags to the files, we also configured the IOZone benchmark suites to perform all file I/O operations on a particular mount point where we mounted our modified memory-based filesystem. Two instances of the memory-based filesystem server ran during the benchmark tests.

In case of libmicro benchmark suite, to measure reliable results, we collect for each microbenchmark approximately 1000 samples, each containing between 5 and 50 000 system calls.

The number of system calls depends on their execution time; system calls with a short execution time (e.g., getpid()) will have a high call count while others (e.g., memcpy() with 10Mb jobs) have lower call counts. To remove outliers, libMicro ignores any samples that are more than three standard deviations from the mean. Prior to analysis, some consistency checks were performed on the data, ensuring that the coefficients of variation were acceptable for all benchmarks in both kernel versions.

Pipebench measures the data throughput of a pipe between two processes. Our experiment uses pipebench by using the shell pipe operator and pushing 800MB of data through that pipe to pipebench. Pipebench then reports the throughput in MB/s, the execution time, and the amount data received.

We executed pipebench on the same platform as libMicro. For the experiment, we collected thirty samples by executing the following line:

```
dd if=/dev/zero bs=80k count=5k 2>/dev/null | \
   ./pipebench -q -Sresults.dat > /dev/null
```

Each benchmark suite uses its own data collection and data processing mechanisms. To permit future comparison with our work, we report the raw values are produced by lmbench, unixbench, libmicro, and iozone. For example, lmbench collects measurements internally before aggregating the results, unixbench has sample size of of three and reruns three times before reporting the results, likewise IOZone collects ten measurements and libmicro collects the data depending on the sample size parameter. Appendix A shows the sample size for each benchmark in libmicro.

6.9 Results and Analysis of Output Data

MiBench All programs have an execution time distribution that differs from the normal distribution. We established this using the Shapiro-Wilk test for normality and visual inspection. Figure 6.1 shows histograms of the execution time for the MiBench *lame* program on the Pentium platform. The x-axis show the execution time in seconds and the y-axis show the frequency of a particular execution time occurring in the sample. The distribution of execution times can be justified by several timing properties such as instruction scheduling anomalies. Figure 6.2 shows the individual results for the MiBench *tiff2rgba* program on the Pentium platform. The x-axis shows the grouped results for the baseline and for tagging. The y-axis shows the execution time in seconds. Using visual inspection, both figures confirm the results of the Shapiro-Wilk test for normality. We therefore rely on robust statistics using for example the median and rank-based testing mechanism in the subsequent analysis.

Table 6.2 shows all results for the MiBench benchmark on the Pentium platform. None of the execution times significantly differ for any of the benchmark programs. The first column

indicates the name of the benchmark program. The second column indicates whether the our basic tagging mechanics are enabled. The third column shows the median of the execution time. And finally, the last column shows the median absolute deviation for the runs. We used the Kruskal-Wallis rank sum test to check for significant slowdown when using tagging. The analysis showed no significant slowdown for any of the benchmark programs. The program *tiffmedian* has the largest difference in the median, however, it is still insignificant with a $p = 0.0458$ given a Bonferroni correction of seven tests on the data. Even if were significant, it would only be a negligible slowdown of a factor of 1.005 (0.5%).

	Name	**Tag**	**Median**	**MAD**
1	jpeg	N	0.163	0.003
2		Y	0.165	0.001
3	lame	N	2.365	0.007
4		Y	2.368	0.009
5	mad	N	0.642	0.004
6		Y	0.645	0.006
7	tiff2bw	N	0.658	0.012
8		Y	0.656	0.013
9	tiff2rgba	N	0.913	0.048
10		Y	0.910	0.044
11	tiffdither	N	0.630	0.001
12		Y	0.630	0.001
13	tiffmedian	N	0.923	0.013
14		Y	0.918	0.015

Table 6.2: Performance summary for MiBench

System Calls The distribution of the execution time for system calls also differs from a normal distribution. Similarly to the MiBench, we confirmed this using a statistical test and visual inspection. Figure 6.3 and Figure 6.4 also confirm this. We again use robust statistics instead of average and mean errors.

Table 6.3 shows all results of our comparison with the unmodified kernel on the Pentium platform. All raw speed measurements are in CPU clock cycles. The first column lists the name of the system call tested in this row. The second column shows the median of the execution times for the baseline (i.e., the unmodified kernel). The third column shows the median absolute deviation of the baseline. The next two pairs of columns provide the same data for the modified version of the kernel with tagging and with lifelines. The last two columns show the ratio between the baseline and the tagging and the lifeline extension.

Figure 6.1: Density plot of the execution time of the MiBench *lame* program.

Although some results show a statistically significant difference, the overall differences are negligible and just a few clock cycles. The function most affected by tagging is msgpass and the results show no increase in the median. The reason is that (1) the best case, our extension adds eleven instructions and (2) in the worst case, our extension adds 58 instructions. Given the regular interference from the computer architecture resulting from pipeline stalls, cache misses, page alignments, and out-of-order execution, it is expected that the measurements show nearly identical values.

Figure 6.2: Individual results for MiBench *tiff2rgba* program.

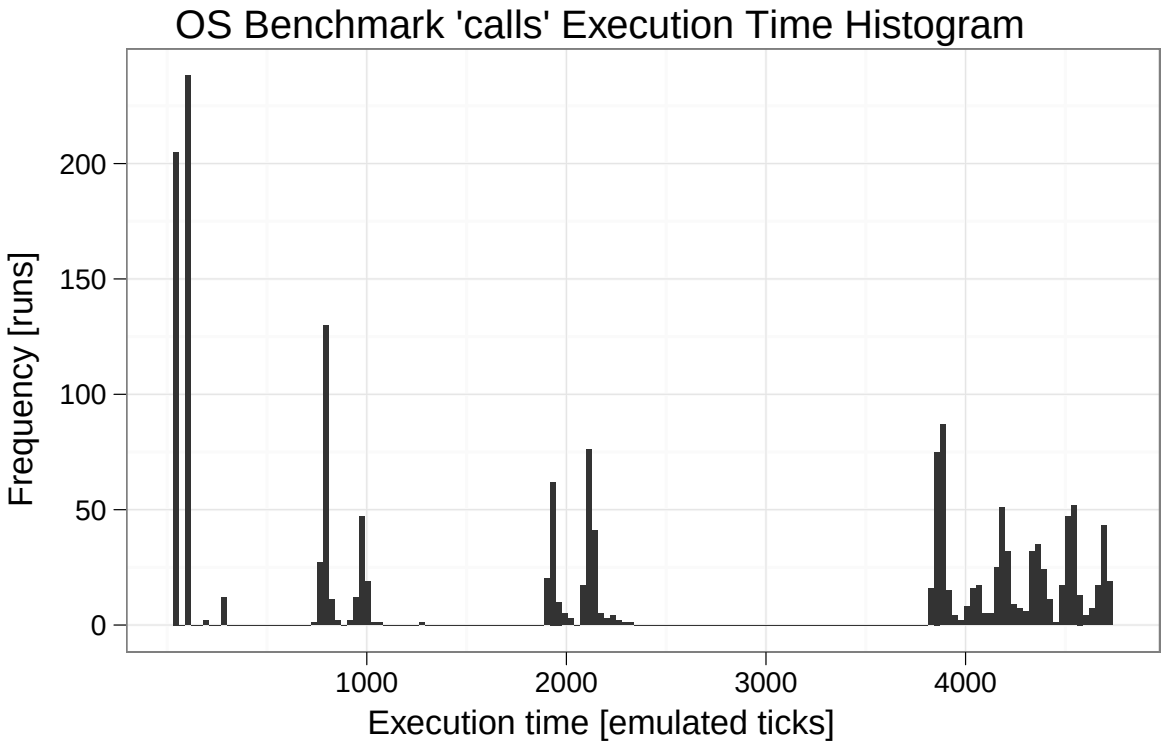

Figure 6.3: Histogram for the *calls* benchmark program.

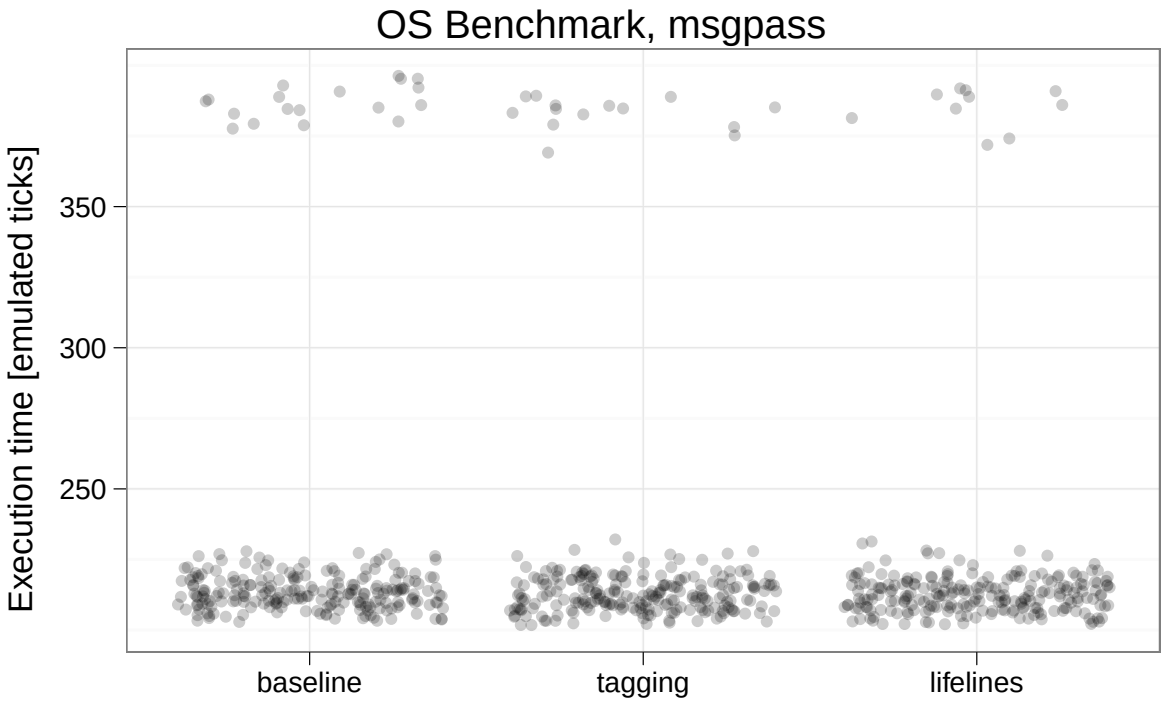

Figure 6.4: Individual results of the OS benchmark on the *msgpass* program.

	Name	Baseline		Tagging		Lifeline		Ratio	
		Median	MAD	Median	MAD	Median	MAD	Tagging	Lifeline
1	calls	2106	2972.613	2103	2969.648	2128	3005.230	0.999	1.010
2	channel	90	2.965	89	2.965	86	2.965	0.989	0.956
3	devnull	1072	65.234	1073	60.787	1080	250.559	1.001	1.007
4	devnullr	839	38.548	850	59.304	833	26.687	1.013	0.993
5	kill	80	2.965	80	2.965	76	1.483	1.000	0.950
6	malloc	109	2.965	108	2.965	106	2.965	0.991	0.972
7	msgpass	213	5.930	213	7.413	213	5.930	1.000	1.000
8	mutex	35	1.483	34	1.483	35	1.483	0.971	1.000
9	mutex_alloc	159	1.483	158	1.483	155	1.483	0.994	0.975
10	process	39934	169.016	39927	180.877	39991	140.847	1.000	1.001
11	sbrk	4123	54.856	4119	54.856	4124	35.582	0.999	1.000
12	signal	34829	1123.811	34953	1323.962	34735	853.978	1.004	0.997
13	syscall	729	382.511	736	357.307	736	367.685	1.010	1.010
14	thread	5841	40.030	5832	42.995	5867	34.100	0.998	1.004
15	timer	111	2.965	112	4.448	113	4.448	1.009	1.018
16	yield	534	10.378	537	8.896	540	8.896	1.006	1.011

Table 6.3: Slowdown for system calls in emulated clock ticks.

Reclen	Mean	SEM	CI
4	0.992	0.004	0.008
8	1.001	0.009	0.018
16	1.003	0.008	0.015
32	0.997	0.000	0.001
64	0.980	0.018	0.036
128	0.999	0.000	0.001
256	0.999	0.000	0.000
512	0.990	0.021	0.041
1024	1.027	0.026	0.052
2048	1.018	0.036	0.071
4096	1.030	0.052	0.103
8192	0.979	0.042	0.082
Overall	1.001	0.018	0.035

Table 6.4: IOZone overhead summary results

Distributed Overhead Since the addition of tags to QNET represents an extra 32-bits per message on the network, we expected the overhead to be very low. The results shown in Table 6.4 confirm this. The column titled 'Mean' shows the ratio between the results of IOZone over QNET with Tags enabled and Tags disabled. The column titled 'SEM' describes the standard error of the mean, and the column titled 'CI' shows the 95% confidence interval. Each row is the summary of a single record length and subsumes the results on the individual micro-benchmarks of IOZone like random read and block rewrite.

The table clearly shows that tagging, even in the distributed version with our modification of QNET at least for message sizes between 4 and 8192 bytes causes no significant overhead.

Libmicro Table 6.5 shows the results for libmicro benchmarks. We ran individual benchmarks with different factor levels. While Welch's t-tests conducted at the 99% confidence level showed statistically different means for the majority of benchmarks, libMicro shows negligible differences between the original kernel and the kernel supporting tags. Table 6.5 shows the ten (out of 138) microbenchmarks with the largest ratio between means, i.e., the ones where tagging overhead is the highest. The first column shows the benchmark name, the next three columns show the results for the original kernel with the number of samples, the mean, and the standard deviation. Next, the table shows the same metrics for the modified kernel that supports tagging. The last column shows the ratio computed by dividing the tag-based kernel's mean by the original kernel's one. This data confirms that, even in the worst case, the overhead is negligible. Appendix A shows the complete set of results for libmicro.

	Original			Tagging-kernel			
Name	**Samples**	**Mean**	**Std. Dev**	**Samples**	**Mean**	**Std. Dev.**	**Ratio**
open_zero	894	14.00	0.00	1001	15.19	0.39	1.08
write_t1k	887	14.24	0.65	996	15.32	0.95	1.08
write_u1k	998	14.92	1.00	909	16.00	0.00	1.07
write_u10k	988	26.30	0.71	947	27.98	1.07	1.06
mktimeT2	1002	27.60	0.84	1001	28.90	1.01	1.05
read_t100k	1001	16.43	0.82	1002	17.18	0.98	1.05
read_t10k	909	14.00	0.00	1002	14.60	0.92	1.04
write_t10k	961	26.03	0.70	967	27.15	0.99	1.04
writev_t1k	897	26.00	0.00	991	26.99	1.00	1.04
pthread_128	102	26.65	3.86	102	27.49	3.92	1.03

Table 6.5: The ten microbenchmarks of the libMicro suite with the worst overhead results. Mean and std. dev. are reported in [us] and values less than 0.004 show as 0.00.

		Original	Tagging	Ratio
Sample Count	Median	1 001	1 001	1
Execution	Mean	444.7761	445.6836	1.0020
	Median	6.8564	6.99894	1.0207

Table 6.6: Aggregates for all 138 libMicro-benchmarks.

Figure 6.5 shows the ratio of the execution time for all libmicro benchmarks, and it also demonstrates that our approach incurs only negligible overhead. The x-axis shows the identifier number for the benchmark. The y-axis shows the ratio between the tag-based kernel and the original kernel. The higher the value, the more overhead ourtag-based-kernel has. Error bars have been intentionally suppressed, as the variance of a ratio is highly sensitive to denominator values that approach zero.

Finally, Table 6.6 shows the aggregate results for libMicro. This table shows that the analysis of the two different versions has been thorough and again, overall, the tagging mechanism incurs only negligible overhead.

Pipebench Prior to our statistical analysis, we performed a series of consistency checks on the data and the benchmark: the throughput remains unaffected by the amount of data passed through the pipe, all data values are positive values, and the data contains no drastic outliers.

Table 6.7 shows the throughput results for pipebench in MB/s. The columns show the number of samples, the mean value, the median, the standard deviation, the standard error of the mean, and the 95% confidence interval. Although a difference between the means of the original and the tag-based kernel is statistically significant (at the 95% level), the additional overhead of approximately 1% is negligible in practical terms.

	Samples	Mean	Median	S.D.	S.E.	C.I.
Original	30	81.83	81.84	0.16	0.03	±0.15
intersert()	30	80.98	80.97	0.18	0.03	±0.18

Table 6.7: Results for pipebench. Values except *count* are reported in MB/s and values less than 0.004 show as 0.00.

Other Benchmarks: unixbench, lmbench and IOZone Tables 6.8, 6.9 and 6.10 show the results for the unixbench, lmbench and IOZone suites, respectively. This time we have configured IOZone to work for memory based file system i.e., ETFS [121]. The first column of each table lists the name of the microbenchmark, the second column lists the results for the kernel with tagging enabled and the last column shows the percentage overhead imposed by tagging.Timing measurements (the second column) are in microseconds. In Table 6.10, the speed of file operations is in Kilobytes per second. We show the mean of at least 50 benchmark iterations. "NSD" in the last column (% Overhead) stands for "Not Statistically Distinct." We collected not only the mean, but also the standard deviation, and computed a 95% confidence interval. If the confidence intervals for the original system and the system with Tags overlap, then this means that we cannot conclude that the mean values for the two are statistically distinct. In such cases, we write "NSD" in the column for % Overhead.

As shown in the results of benchmarks, tagging has only negligible overhead compared to the original version. In most cases, we are not even able to statistically distinguish the overhead imposed by tagging. In some cases, we have observed that the mean values for tagging are faster than for the original system. (We reemphasize however, that the confidence intervals overlap in all such cases, and therefore we cannot conclude that the values are indeed distinct.) We premise that this is because of architecture effects such as instruction ordering and cache affinity, some benchmarks are actually faster with the addition of the small numbers of instructions necessary to implement tagging.

We point out that the reading and writing of files is nearly as fast with tagging as it is on the unmodified kernel only because the data itself remains unencrypted.

Impact of Factors on Performance Metrics: Most of the factors in various benchmark suites are varied to increase/decrease the workload of the systems. Factors like file size, amount

of memory can be tuned to control the workload of specific benchmark operation. For example, increase in amount of memory for *malloc()* operation will increase the amount of work that the memory manager has to perform. As shown by the results, the variation in the values of the factors introduces the variation in the performance metric. As value of factors defines the workload of the system, the factor value is inversely proportional to the performance i.e., increasing the factor value results in performance degradation of the system. We can deduce the same relationship from our results. For example in table 6.9, *Select()* operation on 100 file descriptors is faster than the the *Select()* operation on 250 file descriptors.

Test	**Tagging**	**% Overhead**
Dhrystone 2 using register variables [l/s]	4244181	NSD
Double-Precision Whetstone [MWI/s]	549	NSD
Execl Throughput[l/s]	341	NSD
File Copy 1024 bufsize 2000 maxblocks [KB/s]	21134	NSD
File Copy 256 bufsize 500 maxblocks [KB/s]	11758	NSD
Pipe Throughput [l/s]	60299	NSD
Pipe-based Context Switching[l/s]	32978	NSD
Shell Scripts (8 concurrent) [l/m]	64	NSD
System Call Overhead [l/s]	41278	NSD

Table 6.8: Results for the unixbench benchmark. "NSD" stands for "Not Statistically Distinct" (See Section 6.9).

6.10 Performance of intersert()

To demonstrate the feasibility of checking LTL properties at run time with low overhead, we have executed a property verification benchmark on a series of synthetic interaction histories. The property to be checked is the following: "A $\rightarrow$ XB". Interaction histories were generated with the following pattern: [C, C, C, ..., C, A, B]. Interaction histories ranging in size from 100 to 15,000 were generated by increasing the number of C entries at the start of the history. Since all interaction histories end with [A,B], the intersert() will always pass. This also means that the full interaction history will have to be iterated through, which is the worst case for any given assertion.

Figure 6.6 shows the verification times of the property for different history lengths. 200 execution time measurements were collected for each history length. As it is to be expected, the execution time of this particular verification procedure grows linearly with the size of history. Even in the longest history sizes used, the property was checked in under 350 microseconds. This demonstrates that our system could be used at run time with little overhead.

Test	Tagging	% Overhead
syscall	9	NSD
read	5	NSD
write	5	NSD
stat	47	NSD
fstat	11	NSD
open/close	50	NSD
Select on 100 fd's	300	0.01
Select on 250 fd's	775	0.01
Select on 100 tcp fd's	100	0.03
Select on 250 tcp fd's	248	NSD
Signal handler installation	1	NSD
Signal handler overhead	3	NSD
Protection fault	2	NSD
Pipe latency	30	NSD
AF_UNIX sock stream latency	29	NSD
Process fork+execve	6606	0.01
Process fork+/bin/sh -c	12292	NSD
File write bandwidth	12605	NSD
Pagefaults	8462	NSD
UDP latency using localhost	37	NSD
TCP latency using localhost	37	NSD
TCP/IP connection cost to localhost	164	0.01

Table 6.9: Results for the lmbench benchmark (in microseconds). "NSD" stands for "Not Statistically Distinct" (See Section 6.9).

It is worth noting that the complexity of the TGBA also affects the verification times of properties. As arbitrarily complex properties can be created and checked, execution times are bound to vary widely. Therefore, we suggest developers investigate that the execution times for verifying their properties incur acceptable overhead.

Test	**Tagging**	**% Overhead**
write	292609	NSD
rewrite	256002	NSD
read	341339	NSD
reread	409603	NSD
random read	341336	NSD
random write	256000	NSD
bkwd read	409600	NSD
record rewrite	227556	NSD
stride read	341390	0.02
fwrite	227555	NSD
frewrite	227577	NSD
fread	186181	NSD
freread	186183	NSD

Table 6.10: Results for the iozone benchmark (in kb/sec). "NSD" stands for "Not Statistically Distinct" (See Section 6.9).

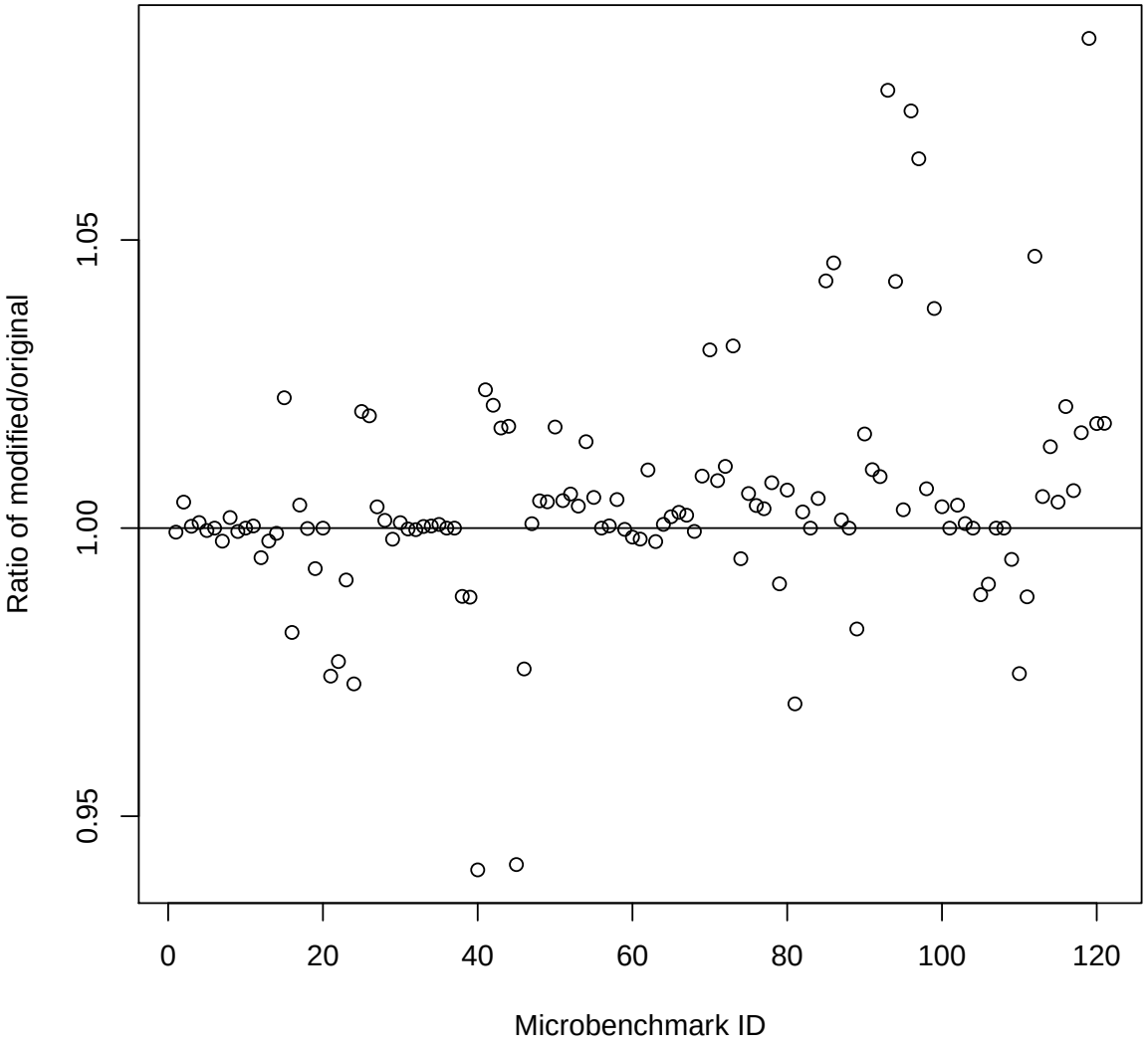

Figure 6.5: Ratio of the execution time for the unmodified and the tagging kernel.

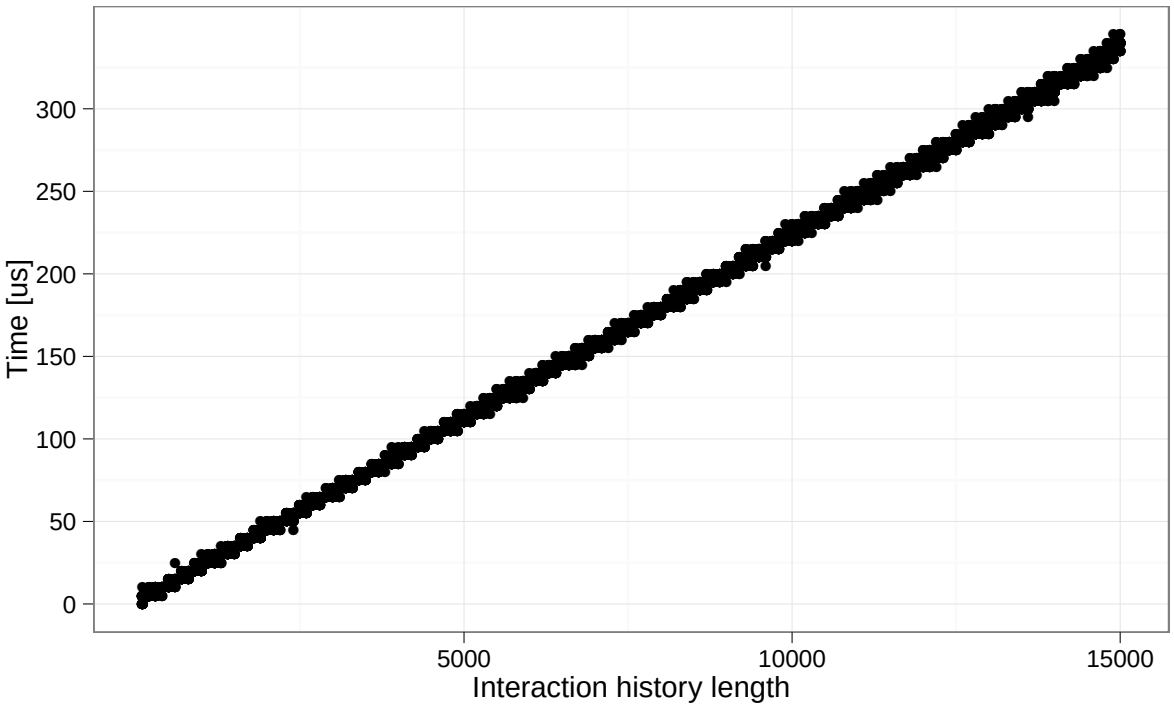

Figure 6.6: Execution times for checking "A $\rightarrow$ XB" with different history lengths.

Chapter 7

Conclusion and Discussion

This chapter discusses a few aspects of the tagging mechanism that we encountered during our investigation of the tagging mechanism. The chapter ends with a conclusion and discusses the future work.

7.1 Discussion

Types of Use-Cases As our investigation of tagging progressed, and as different use cases were implemented, we were able to identify different "levels" of tag integration in programs. Depending on the use case, the developer's use of the tagging mechanism can range from what we call "application-agnostic" to "deeply integrated". We classify these levels of integration as follows, ranging from least to most integrated:

1. Application-Agnostic: This level of integration occurs when the system designer uses tags without modifying any source code and is attempting identify systemwide interactions with a low level of detail. An example of this would be tagging a thread to identify the message chains that include the thread.

2. Source Code Agnostic: when the system designer targets specific interactions, but still does so without modifying the source code. The mandatory use case in Chapter 3 is an example of this.

3. Integrated: This term describes the level of integration that occurs when the system designer targets specific interactions but does so without modifying the source code The *Intersert* use case in Chapter 4 is an example of this mechanism.

4. Deeply Integrated: This term is used when tags are integral to the functionality of the system and their removal or disabling would require the reimplementation of at least part of the application.

Part of our future work will include exploring more use cases at these different levels of integration and investigating what new interactions are enabled by different propagation mechanics and by tagging more elements of the operating system.

Tags vs. Raw Message Logging Logging of message passing in microkernels is either of standard functionality (as is the case with Neutrino) or is easily implementable. The tagging mechanism differs in three fundamental ways from simply logging every message pass. First, the mechanism allows tags to be created in a way that they affect only a subset of all messages, effectively filtering and differentiating the particular message flow in which the developer is interested. For example, two different tags can be created in either outcome of an `if` branch and therefore differentiate between two types of messages that would appear identical to a raw logger.

Second, tags can be easily read and acted on by applications at run time, which is not the case with an eagle-eye" view such as the one provided by the Neutrino message logger. Without tagging, a thread can only know the sender of the messages it receives, whereas tagging enables it to construct a longer history of the message flow that lead to it and to receive tag information from several hops away.

Finally, the propagation of tags cannot be contained without affecting the functionality of the system. The same cannot be said if one tries to limit message passing in any way.

Ease of Implementation Our implementation of tagging on the Neutrino operating system is entirely modular and consists of relatively few lines of code. The non-invasiveness and size, of the code, are valuable features because they limit the likeliness of inserting new bugs into the kernel and facilitate recertification of the tagging-compatible kernel if such a need arises. We believe that these characteristics would carry over to other microkernels as well.

Applicability to Non-Microkernel Systems Message tagging as a concept is not necessarily tied to microkernel operating systems. Equivalent functionality could conceivably be implemented in monolithic systems through a couple of ways. One would be adding tag passing to every method call during the compilation process: the preprocessor can perform code insertion before each call, or tags could be handled by modified calling conventions in the compiler itself. Another way would be through aspect-oriented programming [73], where tagging itself would be an aspect. However, both of these approaches require access to the source code that is meant to use tagging.

Message tagging as a concept is not necessarily tied to microkernel operating systems. Equivalent functionality could conceivably be implemented in monolithic systems through a few different approaches. One approach would be to add tag passing to every method call during the compilation process: the preprocessor could perform code insertion before each call or tags could be handled by modified calling conventions in the compiler itself. Another approach would be through the use of aspect-oriented programming [73], in which tagging itself would be a component. However, both of these approaches require access to the source code that is meant to use tagging.

Kernel Space vs User Space We believe that tagging is best transparently implemented in the operating system. For this reason, we intercept the message-passing functionality of the QNX Neutrino kernel to propagate tags across address space boundaries (or even network nodes). Normal POSIX applications that are ignorant of tags will normally receive and propagate tags. If a simple program that reads data from the network and writes it to disk is to implement tagging in the user space, it must be programmed as follows:

```
1   int main() {
        initialize_tags();
3
        send_tag(filesystem);
5       file = open("filename");

7       send_tag(network);
        socket = create();
9
        send_tag(network);
11      listen(socket);

13      send_tag(network);
        accept(socket);
15
        send_tag(network);
17      while( !socket.empty() ) {
            send_tag(network);
19          read(socket);

21          send_tag(filesystem);
            write(file);
23
            send_tag(network);
25      }

27      send_tag(network);
        close(socket);
29
        send_tag(filesystem);
31      close(file);
    }
```

Listing 7.1: Tagging in user space

Implementation of this program assumes the following: (1) the networking stack sends tag information to the application before each of its messages, (2) the filesystem supports tagging passes along the tags, and (3) the semantics of the tag vector are uniform across all system components. This implementation of tagging requires the modification of the source code of all of the participating components. This way of implementing tagging also requires careful tracking of every message pass to avoid bugs that may arise from omission. Performing all tagging in the kernel as we propose solves both of these problems and is free of assumptions

The implementation of tagging inside the kernel also allows us to enforce access control on tags because such control may be useful, depending on the use case.

Security Model The tagging mechanism provides various options to modify the behavior of the tag propagation. The user can modify the tag features either through the command line options or through APIs provided by the tagging library. The API calls restrict unauthenticated modification by permitting only the respective threads to change the thread level features such as the tag terminator. The security model for the other features, such as TTL, can be implemented only by allowing the tag owner to modify such features. In addition to these restrictions, we can restrict the command line access based on the current user privileges in the system.

7.2 Conclusion

This thesis introduced Tags, a mechanism to augment the messages of microkernel-based operating systems. We showed that tagging is useful in a number of contexts and situations ranging from mandatory security to the verification of components' interaction.

We presented the basic tag propagation mechanics and measured their impact on the system using several standard benchmarks. The measurements show that tagging has a negligible impact on the system performance which demonstrates its adoptability for commercial applications.

7.3 Future Work

Tagging has proved to be a promising and versatile mechanism, and many possible extensions to the current mechanism can be explored.

The tagging infrastructure currently supports file tags, shared-memory tags, and messages tags. Tagging devices and possibly other operating system elements, should make tagging more expressive and allow an entire new class of additional use cases.

Information flow control is an important topic for the security community. An extension to the tagging infrastructure can aid in the development of information flow control for the microkernels. Tagging can exploit well-defined IPC, provided by microkernels, to track information among different components of the system.

A tagging-aware scheduler will be an interesting extension of the tagging infrastructure. Priority can be associated with the tag. The priority will propagate with the messages, as the tag propagates. This priority propagation will add smartness to the scheduler that will be useful for partitioning the CPU.

The propagation mechanics have been deliberately kept simple for reasons of performance and usability reasons: however, some use cases would benefit from different mechanics, such as propagate-on-reply, in addition to or in place of propagate-on-send. We also plan to investigate the passing of data fields along with tags, to add to the expressiveness of tags. The main obstacle in this case would be the added overhead of copying data with every message pass.

Extraction of component interaction patterns can help in designing hardware-software models and simulators. The interaction information can be used to understand, design and optimize the application-specific simulator. Providing component interaction information, to design and optimize the simulator, can be an interesting future application for tagging.

Finally, the use of microkernels enables straightforward tagging of messages, but we believe that through a mix of static analysis and dynamic tracking one could achieve similar if not equivalent functionality on monolithic kernels. The issues would be defining the edges between taggable entities and how to track all of the interactions between them without incurring excessive overhead.

APPENDICES

Appendix A

Full libMicro Experimental Data

libMicro consists of a set of system call microbenchmarks. It is internally structured in a pair of nested loops. Since the execution time of an individual system call is generally too short to measure precisely, the inner loop executes each system call multiple times. A full execution of the inner loop is called a *sample*. The outer loop, therefore, controls how many samples will be collected.

The number of iterations of both the inner and the outer loops are configurable parameters. Our choice of parameters focused on gathering reliable data, with variance estimates that permitted a fair comparison between the original and the tag-based kernels. Wherever the default parameters resulted in data too variable to analyze, we raised the inner loop count until acceptable variances were achieved.

Table A.1 shows both iteration numbers: the number of outer loop iterations is listed in the "Samples" column, and the inner loop iterations is listed in the "Calls" column. The total number of individual calls to each system call is, therefore, Samples $\times$ Calls.

The number of outer loop iterations varies between microbenchmarks because libMicro scales the outer loop according to execution times of the inner loop. It can be seen in the full table that the difference in number of Samples between the original and the tag-based kernels is small, and does not affect statistical analysis in a detrimental way. Nevertheless, to compensate for the difference in sample sizes, we use Welch's t-tests to compare the means.

It should be noted that a small number of benchmarks yielded too large a variance to analyze properly. Since libMicro performs outlier checks internally and does not expose accurate numbers in these cases, these benchmarks were omitted from the analysis. This does not affect the results presented, as the conclusion from the analysis is still valid given the benchmarks that presented reliable results.

Since libMicro is composed of an extensive number of microbenchmarks, the full data set is too large to include in the main paper. Therefore, the full data set is presented in Table A.1. The mean, standard deviation and 95% confidence interval values are in microseconds.

	Original					intersert()-kernel				
Name	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**
bind	102	500	5.3534	0.92435	0.21288	102	500	5.45735	0.87578	0.2017
c_cond_1	910	50000	1.45978	1e-05	0	1001	50000	1.46599	0.00951	0.0007
c_fcntl_1	1002	50	34.61064	8.7255	0.64116	1002	50	34.98911	8.51447	0.62565
c_flock	999	5000	34.98098	0.11821	0.0087	1001	5000	35.17192	0.10614	0.0078
c_lockf_1	1002	50	35.51478	8.18537	0.60147	1002	50	35.8302	7.96672	0.5854
c_mutex_1	1002	50000	0.3135	0.00922	0.00068	1002	50000	0.31375	0.00912	0.00067
close_bad	1002	5000	0.89554	0.10003	0.00735	1002	5000	0.86989	0.09726	0.00715
close_tmp	1002	5000	8.97234	0.06611	0.00486	1002	5000	9.15297	0.08241	0.00606
close_usr	935	5000	8.99863	8e-05	1e-05	999	5000	9.17268	0.06574	0.00484
close_zero	1002	5000	3.34271	0.09147	0.00672	1002	5000	3.35469	0.08621	0.00634
connection	102	500	106.25458	30.38749	6.99849	102	500	107.06708	30.35389	6.99075
dup	969	500	3.99936	0.00092	7e-05	989	500	3.99935	0.00092	7e-05
exit_10	201	10	268.37789	45.86537	7.52483	202	10	269.59058	47.03541	7.69766
exit_100	197	100	287.60332	7.77755	1.2889	197	100	284.60898	8.73791	1.44805
exit_1000	47	1000	346.1747	4.22459	1.43333	43	1000	348.39708	4.68931	1.66335
exit_10_nolibc	201	10	258.38211	49.85477	8.17934	201	10	250.48316	50.01158	8.20507
exp	1001	50000	0.14253	0.00755	0.00055	1001	50000	0.14246	0.00749	0.00055
fcntl_ndelay	1002	5000	4.82155	0.08812	0.00648	1002	5000	4.89788	0.09996	0.00735
fcntl_tmp	1001	5000	6.95515	0.08186	0.00602	937	5000	6.99893	9e-05	1e-05
fcntl_usr	999	5000	6.97091	0.06797	0.005	944	5000	6.99893	9e-05	1e-05
file_lock	1001	5000	16.961	0.08604	0.00633	1001	5000	17.03867	0.09423	0.00693
fork_10	949	500	790.66333	2.98903	0.22569	961	500	795.40098	2.8888	0.21675
fork_100	98	100	776.76355	5.08547	1.19489	101	100	779.67105	3.72027	0.86104
fork_1000	50	1000	813.55711	2.36412	0.77767	50	1000	816.23768	3.07875	1.01274
getenv	1002	5000	1.14577	0.08737	0.00642	1002	5000	1.14493	0.08781	0.00645
getenvT2	1002	5000	6.4926	0.61706	0.04534	1002	5000	6.52603	0.56485	0.04151
getpid	10002	5000000	0.00626	0.0001	0	10002	5000000	0.00626	0.0001	0
getrusage	996	50000	3.81771	0.01223	0.0009	1001	50000	3.88427	0.01164	0.00086
getsockname	1002	50000	5.46619	0.01713	0.00126	1002	50000	5.54408	0.01202	0.00088
gettimeofday	10002	50000	0.09253	0.00956	0.00022	10002	50000	0.09256	0.00956	0.00022
isatty_no	1001	500	23.39637	0.93346	0.06863	1000	500	22.95223	1.03158	0.07588
isatty_yes	1002	500	5.53434	0.82951	0.06095	1002	500	5.66681	0.72914	0.05358
listen	1002	500	4.54605	0.9256	0.06801	1002	500	4.64068	0.95767	0.07037
localtime_r	901	500	7.99873	0.00098	8e-05	1002	500	7.50668	0.84646	0.0622
log	10001	50000	0.09328	0.00932	0.00022	10001	50000	0.09337	0.00928	0.00022

Continued on next page

	Original					Tagging-kernel				
Name	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**
longjmp	880	50000	0.37994	1e-05	0	1002	50000	0.38848	0.00996	0.00073
lrand48	937	50000	0.02	1e-05	0	935	50000	0.02	1e-05	0
lseek_t8k	1001	5000	6.75245	0.08333	0.00613	925	5000	6.79896	7e-05	1e-05
lseek_u8k	1002	500	6.68031	0.96819	0.07114	1002	500	6.71816	0.97683	0.07178
mallocT2_10	999	500	2.25888	0.13701	0.01008	997	500	2.20304	0.13245	0.00976
mallocT2_100	992	500	2.31897	0.12548	0.00927	997	500	2.26576	0.12556	0.00925
mallocT2_100k	996	5000	23.90106	0.37499	0.02764	995	5000	23.67026	0.36834	0.02716
mallocT2_10k	1002	50	3.36433	0.94205	0.06922	1002	50	3.45896	0.90078	0.06619
mallocT2_1k	1000	500	3.15594	0.14391	0.01059	994	500	3.12791	0.13259	0.00978
malloc_10	1002	5000	0.25214	0.00967	0.00071	1002	5000	0.25321	0.00934	0.00069
malloc_100	1002	5000	0.28441	0.0088	0.00065	1002	5000	0.28439	0.00879	0.00065
malloc_100k	986	500	0.39994	9e-05	1e-05	978	500	0.39994	9e-05	1e-05
malloc_10k	1002	5000	0.45011	0.00998	0.00073	1002	5000	0.44462	0.00892	0.00066
malloc_1k	967	5000	0.37994	1e-05	0	1001	5000	0.37712	0.00681	0.0005
memcpy_10	946	500000	0.03399	0	0	956	500000	0.03399	0	0
memcpy_10k	1002	5000	1.13441	0.09256	0.0068	1002	5000	1.1321	0.09343	0.00687
memcpy_10m	1001	5	17708.46021	98.79654	7.26331	1001	5	17705.724	99.17129	7.29086
memcpy_1k	1002	50000	0.15524	0.00835	0.00061	1002	50000	0.15548	0.0082	0.0006
memcpy_1m	1002	500	1702.7926	1.82135	0.13383	1002	500	1704.39069	1.80453	0.1326
memrand	202	100000	0.12061	0.00318	0.00052	202	100000	0.12061	0.00318	0.00052
memsetP2_10m	996	5	21860.73626	111.48476	8.21666	994	5	21841.17108	112.44213	8.29556
memset_10	1002	500000	0.01248	0.0009	7e-05	1002	500000	0.01253	0.00092	7e-05
memset_10k	996	5000	1.62301	0.07347	0.00541	1002	5000	1.62351	0.07402	0.00544
memset_10m	1002	5	10731.17424	54.56777	4.0097	1002	5	10706.88738	46.93887	3.44912
memset_1k	970	50000	0.18455	0.00886	0.00066	1001	50000	0.1849	0.00903	0.00066
memset_1m	996	500	201.39221	1.09629	0.0808	996	500	200.36275	1.22464	0.09026
memset_256	947	500000	0.06302	0.00114	9e-05	922	500000	0.06305	0.00118	9e-05
memset_256_u	971	50000	0.06561	0.00931	0.0007	1002	50000	0.06544	0.00925	0.00068
memset_4k	1002	5000	0.66025	0.09459	0.00695	996	5000	0.65994	0.09445	0.00696
memset_4k_uc	909	5000	4.19936	8e-05	1e-05	911	5000	4.19936	8e-05	1e-05
mktime	1002	500	8.41084	0.86225	0.06336	1002	500	8.1922	0.69978	0.05142
mktimeT2	1002	500	27.55836	0.84203	0.06187	1001	500	28.84191	1.00778	0.07409
mutex_T2	1002	50000	0.37721	0.009	0.00066	1002	50000	0.35528	0.009	0.00066
mutex_mt	1002	50000	0.15056	0.00995	0.00073	1002	50000	0.14864	0.00997	0.00073
mutex_st	1002	50000	0.15062	0.00994	0.00073	1002	50000	0.14873	0.00998	0.00073

Continued on next page

	Original					Tagging-kernel				
Name	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**
open_tmp	1001	500	27.10113	0.98891	0.0727	1001	500	27.57055	0.80991	0.05954
open_usr	1002	50	24.5384	8.84543	0.64997	1002	50	25.35872	9.21505	0.67713
open_zero	894	1000	13.99787	0.00033	3e-05	1001	1000	15.14399	0.39228	0.02884
poll_10	1002	500	10.12462	0.62797	0.04614	1002	500	10.10781	0.60758	0.04465
poll_100	1002	500	55.38586	1.1739	0.08626	1002	500	55.27445	1.18928	0.08739
poll_1000	981	50	594.75383	10.82513	0.80391	933	50	601.10472	6.18888	0.47128
poll_w10	1002	500	10.16669	0.67444	0.04956	1002	500	10.14144	0.64727	0.04756
poll_w100	1002	500	55.42372	0.89112	0.06548	1002	500	55.46157	0.87149	0.06404
poll_w1000	1002	50	574.86484	8.57381	0.63001	1002	50	576.08444	7.71355	0.5668
pthread_128	102	128	26.3632	3.86334	0.88976	102	128	27.2583	3.92221	0.90332
pthread_32	1002	250	29.69475	1.99231	0.1464	1002	250	30.03114	1.99609	0.14668
pthread_512	52	512	35.23031	0.79822	0.25747	48	512	35.15069	0.0008	0.00027
pthread_8	1000	500	34.47889	0.90276	0.0664	999	500	34.77856	0.98828	0.07273
read_t100k	1001	500	16.34064	0.82019	0.0603	1002	500	17.13722	0.98331	0.07226
read_t10k	909	500	13.998	0	0	1002	500	14.52353	0.91718	0.0674
read_t1k	1001	500	9.69529	0.70214	0.05162	1001	500	9.74791	0.64751	0.0476
read_u100k	1000	500	16.72795	0.97889	0.072	1000	500	16.41852	0.86622	0.06371
read_u10k	914	500	13.998	0	0	897	500	13.998	0	0
read_u1k	1002	500	9.72504	0.67222	0.0494	1001	500	9.73949	0.65675	0.04828
read_z100k	1002	500	59.90255	0.82149	0.06036	1002	500	60.54382	1.20854	0.0888
read_z10k	1002	500	8.34567	0.82268	0.06045	1002	500	8.49074	0.90257	0.06632
read_z1k	1002	500	2.88281	0.99856	0.07338	1002	500	2.92066	1.00006	0.07349
read_zw100k	1002	500	60.21791	0.8474	0.06227	1002	500	60.74781	1.08618	0.07981
realpath_tmp	1001	500	37.56893	0.80296	0.05903	1001	500	37.74785	0.64283	0.04726
realpath_usr	1002	50	37.51237	6.44852	0.47384	1002	50	37.70162	6.22958	0.45776
recurse	983	5000	0.79987	0.0001	1e-05	979	5000	0.79987	0.0001	1e-05
scasecmp_10	1002	500000	0.05302	0.001	7e-05	999	500000	0.05171	0.00068	5e-05
scasecmp_1k	932	5000	3.19951	0.0001	1e-05	933	5000	3.19951	0.0001	1e-05
select_10	1002	500	10.10361	0.60232	0.04426	1002	500	10.13303	0.63773	0.04686
select_100	1002	500	54.31138	0.80419	0.05909	1002	500	54.43965	0.88136	0.06476
select_w10	1002	500	10.4211	0.86802	0.06378	1002	500	10.41478	0.86459	0.06353
select_w100	1002	500	55.46368	0.86691	0.0637	990	500	55.99146	0.00087	6e-05
setsockopt	1002	500	4.54185	0.9241	0.0679	1002	500	4.59441	0.94317	0.06931
sigaction	1002	5000	0.3562	0.08114	0.00596	1002	5000	0.3623	0.07663	0.00563
siglongjmp	955	50000	0.37994	1e-05	0	1002	50000	0.38741	0.00982	0.00072

Continued on next page

	Original					Tagging-kernel				
Name	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**	**Samples**	**Calls**	**Mean**	**Std. Dev.**	**95% C.I.**
signal	867	5000	1.3998	0	0	1002	5000	1.43469	0.08241	0.00606
sigprocmask	1002	50000	0.37704	0.00689	0.00051	1002	50000	0.3832	0.00809	0.00059
socket_i	1002	500	25.59438	0.79976	0.05877	999	500	25.73682	0.65741	0.04838
socket_u	1002	500	25.42827	0.89113	0.06548	1002	500	25.43879	0.88586	0.06509
socketpair	1002	500	36.62947	1.60542	0.11797	1002	500	36.62106	1.5994	0.11753
stat_tmp	1001	500	31.05839	0.99419	0.07309	979	500	31.74996	0.64177	0.04771
stat_usr	1002	50	31.01493	9.89387	0.72701	1002	50	31.81396	9.75042	0.71647
strchr_10	996	500000	0.0165	0.00091	7e-05	1001	500000	0.01641	0.00086	6e-05
strchr_1k	1002	5000	1.08437	0.09957	0.00732	1002	5000	1.08479	0.09961	0.00732
strcmp_10	985	50000	0.01764	0.0063	0.00047	993	50000	0.01766	0.00628	0.00046
strcmp_1k	1002	5000	1.24271	0.08715	0.0064	1002	5000	1.24355	0.08759	0.00644
strcpy_10	1002	50000	0.0159	0.00806	0.00059	1002	50000	0.016	0.00805	0.00059
strcpy_1k	1001	5000	0.81524	0.06591	0.00485	1002	5000	0.81501	0.06567	0.00483
strftime	1002	5000	8.79235	0.07807	0.00574	1002	5000	8.69121	0.1024	0.00752
strlen_10	1002	500000	0.01636	0.00083	6e-05	1002	500000	0.01635	0.00083	6e-05
strlen_1k	1002	5000	0.70504	0.09954	0.00731	1002	5000	0.70525	0.09951	0.00731
strtol	944	50000	0.09998	1e-05	0	946	50000	0.09998	1e-05	0
time	1002	50000	0.04897	0.00999	0.00073	1002	50000	0.04895	0.00999	0.00073
times	1002	500	10.90046	0.99949	0.07344	1002	500	11.10442	0.98877	0.07266
write_n100k	1002	500	2.67674	0.96716	0.07107	998	500	2.70842	0.9744	0.07174
write_n10k	998	500	2.69576	0.97149	0.07153	1002	500	2.68726	0.96976	0.07126
write_n1k	1002	500	2.60105	0.94521	0.06946	1002	500	2.6452	0.95871	0.07045
write_t100k	931	500	122.49039	0.92392	0.07043	919	500	122.89537	1.01306	0.07773
write_t10k	961	500	25.92588	0.70313	0.05276	967	500	27.10044	0.98906	0.07398
write_t1k	887	500	14.1451	0.65167	0.0509	996	500	15.28518	0.94663	0.06977
write_u100k	919	500	122.29981	0.80229	0.06156	867	500	123.17686	0.97626	0.07712
write_u10k	988	500	26.20496	0.71401	0.05284	947	500	27.87338	1.0702	0.08089
write_u1k	998	500	14.86058	0.99715	0.07342	909	500	15.99755	0.00082	6e-05
writev_t10k	934	500	123.82102	1.02551	0.07805	925	500	124.30451	0.81208	0.06211
writev_t1k	897	500	25.996	0	0	991	500	26.9332	1.00032	0.07391

Table A.1: The full data set from the libMicro experiments. Mean and std. dev. are reported in [us].

References

[1] Buffer overflow bug in QNX. http://www.cvedetails.com/cve/CVE-2008-3024.

[2] Cisco. http://www.cisco.com/en/US/products/ps5763/.

[3] Format string vulnerability in fontsleuth in QNX Neutrino. http://www.cvedetails.com/cve/CVE-2006-0618.

[4] Fortna- Warehouse Control Systems. http://www.fortna.com/products.php/content/fortna_wcs_warehouse_control_system.

[5] Intalysis- Technology for Online Analysis. http://intalysis.com.au/products/.

[6] LIDS. http://www.lids.org/.

[7] National Security Agency – Central Security Service, Security-Enhanced Linux. http://www.nsa.gov/research/selinux/index.shtml, accessed May 2011.

[8] QNX : Security Vulnerabilities. http://www.cvedetails.com/vulnerability-list/vendor_id-436/QNX.html.

[9] QNX Customers. http://www.qnx.com/company/customer_stories/.

[10] QNX Neutrino. http://www.qnx.com/products/neutrino-rtos/index.html.

[11] Windows Integrity Mechanism. http://msdn.microsoft.com/en-us/library/bb625957.aspx.

[12] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Summer Conference*, pages 93–112, 1986.

[13] Aleph1. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[14] A. Alonso and J. Antonio de la Puente. Implementation of Mode Changes with the Ravenscar Profile. In *Proc. of the 10th International Workshop on Real-time Ada Workshop*, pages 27–32, New York, NY, USA, 2001. ACM.

[15] Android. Android Operating System, 2011. http://www.android.com.

[16] Lee Badger, Lee Badger, Daniel F. Sterne, Daniel F. Sterne, David L. Sherman, David L. Sherman, Kenneth M. Walker, Kenneth M. Walker, Sheila A. Haghighat, and Sheila A. Haghighat. A domain and type enforcement unix prototype. In *In Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 127–140, 1996.

[17] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation0. *Commun. ACM*, 27(1):42–52, 1984.

[18] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14:1–14:64, 2011.

[19] D.E. Bell. Looking back at the bell-la padula model. In *Computer Security Applications Conference, 21st Annual*, pages 15 pp. –351, December 2005.

[20] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaala tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control: verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[21] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[22] Gérard Berry. *The foundations of Esterel*, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.

[23] Biba. Integrity Considerations for Secure Computer Systems. *MITRE Co., technical report ESD-TR 76-372*, 1977.

[24] B. Bouyssounouse and J.Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *LNCS*. Springer, first edition, May 2005.

[25] Mic Bowman, Saumya K. Debray, and Larry L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 1993.

[26] Johannes Braams. Babel, a multilingual style-option system for use with latex's standard document styles. *TUGboat*, 12(2):291–301, June 1991.

[27] I. Branovic, R. Giorgi, and E. Martinelli. A Workload Characterization of Elliptic Curve Cryptography Methods in Embedded Environments. In *Proc. of the 2003 Workshop on Memory Performance (MEDEA)*, pages 27–34, New York, NY, USA, 2003. ACM.

[28] G.C. Buttazzo. HARTIK: A Real-time Kernel for Robotics Applications. In *Proc. Real-Time Systems Symp.*, pages 201–205, 1993.

[29] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for c programs. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 123–124, Washington, DC, USA, 2009. IEEE Computer Society.

[30] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX 2004 Annual Technical Conference*, pages 15–28, 2004.

[31] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1:144–156, May 1983.

[32] Ping Hang Cheung and Alessandro Forin. A C-Language Binding for PSL. In *Proceedings of the 3rd international conference on Embedded Software and Systems*, ICESS '07, pages 584–591, Berlin, Heidelberg, 2007. Springer-Verlag.

[33] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[34] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. *Security and Privacy, IEEE Symposium on*, 0:184, 1987.

[35] Malcolm Clark. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.

[36] J. Clause, W. Li, and A. Orso. Dytan: a Generic Dynamic Taint Analysis Framework. In *Proc. of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[37] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems (3rd ed.): concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[38] Igor D.D. Curcio. ASAP - A Simple Assertion Pre-Processor. *SIGPLAN Not.*, 33(12):44–51, December 1998.

[39] Ewen Denney and Bernd Fischer. Annotation Inference for Safety Certification of Automatically Generated Code. In *Proceedings of the 21st IEEE/ACM International Conference on*

Automated Software Engineering, ASE '06, pages 265–268, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Will Drewry and Tavis Ormandy. Flayer: Exposing Application Internals. In *Proc. of the First USENIX Workshop on Offensive Technologies (WOOT)*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[41] Doron Drusinsky. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, UK, 2000. Springer-Verlag.

[42] Doron Drusinsky. Monitoring temporal rules combined with time series. In *In CAV03, volume 2725 of LNCS*, pages 114–118. Springer-Verlag, 2003.

[43] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *Proc. of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, pages 17–30, New York, NY, USA, 2005. ACM.

[44] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 17–30, New York, NY, USA, 2005. ACM.

[45] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[46] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[47] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19:42–51, 2002.

[48] Advanced Computing for Science Department. NetLogger Toolkit - Example: Lifelines, 2010. http://acs.lbl.gov/NetLoggerWiki/index.php/NetLogger_Toolkit.

[49] John Fusco. *The Linux Programmer's Toolbox*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[50] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new Kernel Approach for Modular Real-time Systems Development. In *Proc. of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 199–206, 2001.

[51] M.P. Gallaher and B.M. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards & Technologg Planning Report 02–03, May 2002.

[52] Dimitra Giannakopoulou, Dimitra Giannakopoulou Riacs, Klaus Havelund, Klaus Havelund, and Kestrel Technologies. Automata-based verification of temporal properties on running programs. In *In Proceedings, International Conference on Automated Software Engineering (ASE)*, pages 412–416. IEEE Computer Society, 2001.

[53] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, Boston, MA, USA, 1983.

[54] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, Berkeley, CA, USA, 1996. USENIX Association.

[55] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM.

[56] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[57] Gregor Gössler, Sussane Graf, Mila Majster-Cederbaum, M. Martens, and Joseph Sifakis. An approach to modelling and verification of component based systems. In *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, pages 295–308, Berlin, Heidelberg, 2007. Springer-Verlag.

[58] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. *SIGPLAN Not.*, 39(4):49–57, 2004.

[59] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the IEEE International Workload Characterization (WWC-4)*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[60] Thomas Habets. pipebench, April 2012.

[61] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical Profiling: Understanding the Behavior of Object-oriented Applications. *SIGPLAN Not.*, 39(10):251–269, 2004.

[62] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[63] Dan Hildebrand. An Architectural Overview of QNX. In *Proc. of the Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[64] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[65] C.E. Hrischuk and C.M. Woodside. Logical clock requirements for reverse engineering scenarios from a distributed system. *Software Engineering, IEEE Transactions on*, 28(4):321 –339, April 2002.

[66] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[67] Immunix. Apparmor. https://apparmor.wiki.kernel.org/index.php/Main_Page.

[68] David M. Jordan. Multics data security. *Scientific Honeyweller*, 2(2), 1981. Available from http://www.multicians.org/multics-data-security.html.

[69] Journal paper citation withheld due to the double-blind process.

[70] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[71] Arun Kejariwal, Alexander V. Veidenbaum, Alexandru Nicolau, Milind Girkar, Xinmin Tian, and Hideki Saito. On the Exploitation of Loop-level Parallelism in Embedded Applications. *ACM Trans. Embed. Comput. Syst.*, 8(2):1–34, 2009.

[72] BSD kernels. securelevel. http://www.freebsd.org/doc/en/books/faq/security.html.

[73] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[74] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Form. Methods Syst. Des.*, 24(2):129–155, March 2004.

[75] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[76] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[77] R. Krishnakumar. Kernel korner: kprobes-a Kernel Debugger. *Linux J.*, 2005(133):11, 2005.

[78] Natalija Krivokapić, Alfons Kemper, and Ehud Gudes. Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis. *The VLDB Journal*, 8:79–100, October 1999.

[79] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[80] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation. *Microsoft Technical Report*, 2006.

[81] Bell Laboratories. Unix Programmer's Manual, January 1979.

[82] Leslie Lamport. *LaTeX User's Guide and Document Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[83] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[84] Carl Staelin Larry McVoy. lmbench. http://www.bitmover.com/lmbench/.

[85] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[86] John Levon. OProfile, 2004. http://oprofile.sourceforge.net.

[87] D. Li, P.H. Chou, and N. Bagherzadeh. Mode Selection and Mode-Dependency Modeling for Power-Aware Embedded Systems. In *Proc. of the 2002 Asia and South Pacific Design Automation Conference (ASP-DAC)*, page 697, Washington, DC, USA, 2002.

[88] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 164–178, Washington, DC, USA, 2007. IEEE Computer Society.

[89] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 164–178, Washington, DC, USA, 2007. IEEE Computer Society.

[90] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.

[91] Spot Library. Spot library, April 2012.

[92] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, 1995.

[93] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39:70–77, September 1996.

[94] Peter A. Loscocco and Stephen D. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the Ottawa Linux Symposium*, 2001. Available from http://www.nsa.gov/research/_files/selinux/papers/ottawa01-abs.shtml.

[95] Byte Magazine. unixbench. http://code.google.com/p/byte-unixbench/.

[96] Milo M. K. Martin. Formal verification and its impact on the snooping versus directory protocol debate, 2005.

[97] MathWorks. Polyspace. http://www.mathworks.com/products/polyspace/.

[98] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. *Softw. Pract. Exper.*, 22:673–694, August 1992.

[99] Marshall Kirk Mckusick. Using gprof to Tune the 4.2BSD Kernel. In *Proc. of the European UNIX Users Group Meeting*, 1984.

[100] D. Molnar and D. Wagner. Catchconv: Symbolic Execution and Run-time Type Inference for Integer Conversion Errors. Technical Report EECS-2007-23, UC Berkeley EECS, 2007.

[101] Richard J. Moore. Dynamic Probes and Generalised Kernel Hooks Interface for Linux. In *Proc. of the 4th Annual Linux Showcase & Conference (ALS)*, pages 35–35, Berkeley, CA, USA, 2000. USENIX Association.

[102] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[103] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000.

[104] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. *SIGARCH Comput. Archit. News*, 36(1):211–221, 2008.

[105] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and visualizing full systems with data flow tomography. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 211–221, New York, NY, USA, 2008. ACM.

[106] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, May 1998.

[107] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[108] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, USA, February 2005.

[109] William D. Norcott. Filesystem benchmark iozone. http://www.iozone.org/.

[110] NSA. Security enhanced linux. http://www.nsa.gov/research/selinux/.

[111] Augusto Oliveira, Ahmad Saif Ur Rehman, and Sebastian Fischmeister. Specification of mtags. Tech Report, September 2011.

[112] OMG UML. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2, 2007.

[113] Ontario Power Generation Inc. SDS1 Software Design Description, NK38-MAN-68258-001, Rev06, 2002.

[114] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes*, 27(4):55–64, 2002.

[115] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. Fs: An in-kernel integrity checker and intrusion detection file system. In *Proceedings of the 18th USENIX conference on System administration*, pages 67–78, Berkeley, CA, USA, 2004. USENIX Association.

[116] Shari Lawrence Pfleeger and Les Hatton. Investigating the influence of formal methods. *Computer*, 30(2):33–43, February 1997.

[117] Kevin Elphinstone Philip Derrin, Dhammika Elkaduwe. sel4 Reference Manual. http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf.

[118] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[119] J.A. Poovey, T.M. Conte, M. Levy, and S. Gal-On. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro*, 29(5):18–29, 2009.

[120] Niels Provos. Preventing privilege escalation. In *In Proceedings of the 12th USENIX Security Symposium*, pages 231–242, 2003.

[121] QNX. Embedded transaction file system. http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/fsys.html#ETFS.

[122] QNX. qconn. http://www.qnx.com/developers/docs/6.4.1/neutrino/utilities/q/qconn.html.

[123] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Lonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems*, 1:39–69, 1991.

[124] Sergio Ruocco. A Real-time Programmer's Tour of General-purpose L4 Microkernels. *EURASIP J. Embedded Syst.*, 2008:1–14, 2008.

[125] S.L. Salas and Einar Hille. *Calculus: One and Several Variable*. John Wiley and Sons, New York, 1978.

[126] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[127] Stefan Schäfer and Bernhard Scholz. Optimal Chain Rule Placement for Instruction Selection based on SSA Graphs. In *Proc. of the 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 91–100, New York, NY, USA, 2007. ACM.

[128] Bernhard Scholz, Bernd Burgstaller, and Jingling Xue. Minimal Placement of Bank Selection Instructions for Partitioned Memory Architectures. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–32, 2008.

[129] Inc. Silicon Graphics. KernProf, 2003. http://oprofile.sourceforge.net.

[130] Bart Smaalders and Phil Harman. libmicro - portable microbenchmarks, April 2012.

[131] Daniel J. Sorin, Manoj Plakal, Anne E. Condon, Mark D. Hill, Milo M. K. Martin, and David A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Trans. Parallel Distrib. Syst.*, 13(6):556–578, June 2002.

[132] Matthew Staats and Mats P. Heimdahl. Partial Translation Verification for Untrusted Code-Generators. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 226–237, Berlin, Heidelberg, 2008. Springer-Verlag.

[133] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 1–1, Berkeley, CA, USA, 2004. USENIX Association.

[134] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222, New York, NY, USA, 2003. ACM.

[135] QNX Software Systems. System Architecture - Interprocess Communication (IPC), 2010. http://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/ipc.html.

[136] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[137] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39:44–51, May 2006.

[138] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[139] Inc. Underbit Technologies. MAD: MPEG Audio Decoder, 2010. http://www.underbit.com/products/mad/.

[140] M. Vuagnoux. Autodafé: an Act of Software Torture. Technical report, Swiss Federal Institute of Technology (EPFL), 2006.

[141] Common Vulnerabilities and Exposures. Linux kernel network stack exploit. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1173.

[142] Carsten Weinhold and Hermann Härtig. Vpfs: building a virtual private file system with a small trusted computing base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 81–93, New York, NY, USA, 2008. ACM.

[143] David Wichers, Douglas Cook, Ronald Olsson, John Crossley, Paul Kerchen, Karl Levitt, and Raymond Lo. PACLs: An access control list approach to anti-viral security. In *Proc. of the 13th National Computer Security Conference*, pages 340–349, 1990.

[144] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[145] Karim Yaghmour and Michel R. Dagenais. Measuring and Characterizing System Behavior using Kernel-level Event Logging. In *Proc. of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[146] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[147] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazires. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[148] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

[149] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

[150] Dino Dai Zovi. Kernel rootkits. http://www.theta44.org/lkr.pdf.