SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–13
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

The Use of mTags for Mandatory Security: A Case Study

Ahmad Saif Ur Rehman, Augusto Born de Oliveira, Mahesh Tripunitara, Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo, Canada
{asaifurr,a3olivei,tripunit,sfischme}@uwaterloo.ca

SUMMARY

mTags is an efficient mechanism that augments inter-thread messages with lightweight metadata. We introduce and discuss a case-study we have conducted in the use of mTags for realizing a kind of mandatory security. While mTags can be implemented for any message-passing thread-based system, we consider an implementation of it in the POSIX-compliant QNX Neutrino, a commercial microkernel-based system. The approach to mandatory security that we adopt is Usable Mandatory Integrity Protection (UMIP), which has been proposed in recent research. We call our adaptation of UMIP using mTags, μMIP. We discuss the challenges we faced, and our design and implementation that overcomes these challenges. We discuss the performance of our implementation for well-established benchmarks. We conclude with the observation that mTags can be useful and practical to realize mandatory security in realistic systems. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Security, research, microkernels

1. INTRODUCTION

mTags [1] is an efficient mechanism for attaching lightweight metadata with threads. Such metadata can then be propagated via message-passing, which is a common approach to communication between threads. In prior work [1], we have introduced mTags, discussed its implementation, and suggested several applications. In this paper, we discuss, in more depth, an application we do not address in our prior work on mTags: mandatory security. In particular, we discuss a case-study we have conducted in realizing Usable Mandatory Integrity Protection (UMIP) [2] using mTags in the QNX Neutrino microkernel-based system [3].

mTags We call the metadata that we associate in mTags, *tags*. When two threads communicate using messages, we propagate some or all tags from one thread to the other. We do this by extending the message with the tags. We call such a message a *tagged message*. In message-passing between threads, an instance of communication typically involves a message from one thread to another to initiate communication, and a message in response. The former is called a request message, and the latter a response message.

We typically propagate the tags from a tagged request message to the thread that receives it. A main aspect of using mTags is coming up with the rules as to whether or not a tag propagates when two threads exchange a message. This is governed by rules that are specific to the use of mTags. For the case-study that we consider in this paper, we discuss our tag-propagation rules in Section 2.4.

The propagation of tags occurs within the kernel, and therefore, typically, no application source-code needs to be modified.

UMIP As we mention above, our case-study realizes UMIP using tags. UMIP [2] is a recent approach to mandatory protection that has been proposed for processes in Linux. It works as follows.

Every process is associated with an integrity level, which is a bit that indicates whether the process is of low or high integrity. This integrity level 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 (see, however, our discussions on exceptions below): (1) receive network traffic, (2) receive Inter-Process Communication (IPC) from a low integrity process, and, (3) read a low integrity file.

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 read only those 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.

The mindset behind UMIP is that the main threats to a system come from the network. It has the drawback that all such taint-based systems have. In systems in which several or all processes need to communicate over the network, they can all quickly be downgraded to low integrity. UMIP mitigates this by providing for exceptions that are designed carefully. For example, a process may be excepted from being downgraded even though it receives network traffic or IPC from a low integrity process. We may award such an exception to, for example, *sshd* (the secure shell daemon), and the desktop manager. Another exception is that notwithstanding the integrity level of a process, it may be allowed to perform some privileged actions. (We discuss the specific exceptions we have adopted for our system in Section 2.4.)

We have chosen UMIP for our work for its practicality and newness. It is a state-of-the-art approach to realizing mandatory security in realistic systems. The work on UMIP [2] asserts that it is effective for realistic systems with a low administrative overhead, notwithstanding its potential drawbacks.

UMIP, in turn, is based on the prior work of Biba [4], and subsequent work on mandatory security for monolithic systems such as LOMAC [5], SELinux [6] and securelevel [7]. We discuss these pieces of work and others in Section 4 on related work. We give a brief overview here of the main differences between UMIP and work prior to it. Compared to Biba's work, UMIP coexists with the discretionary access control (DAC) that is common in modern systems. Compared to LOMAC, UMIP is far less rigid. In LOMAC, for example, an object's integrity level never changes once assigned. UMIP is easier to configure and administer than SELinux and securelevel.

Our work We have designed a version of UMIP for microkernel- and thread-based systems that we call μMIP. In μMIP, we propagate integrity levels using mTags. We did this to validate the use of mTags to realize mandatory security. We have chosen the commercial microkernel-based system, QNX Neutrino, in which to implement μMIP. We point out, however, that mTags and μMIP can be implemented for other thread- and message-passing based systems, such as POSIX, as well.

Process vs. thread UMIP, and therefore μMIP, is designed for protecting processes. mTags are intended to work with message-passing, which we typically associate with threads. In the remainder of this work, we use the term "process," as our focus is μMIP. That is, we consider processes each of which comprises a single thread. Our approach naturally and easily extends to processes with multiple threads – a process is of low integrity if and only if one of its threads is of low integrity.

Challenges that we faced We faced several technical challenges in the design and implementation of μMIP. In the next section, we discuss how we overcame these challenges.

DOI: 10.1002/spe

A challenge regards UMIP's use of the discretionary access control settings (e.g., the Unix file permission bits) to determine whether a file should be deemed to be high or low integrity. In microkernel-based systems, and specifically QNX Neutrino, 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.

Another 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 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 has effect. 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.

Layout In the next section, we discuss the design and implementation of μMIP. We discuss the evaluation of our implementation against benchmarks in Section 3. We discuss related work in Section 4, and conclude with Section 5.

2. REALIZING μMIP

In this section, we discuss our design and implementation of μMIP. We begin with an illustrative example of its main features.

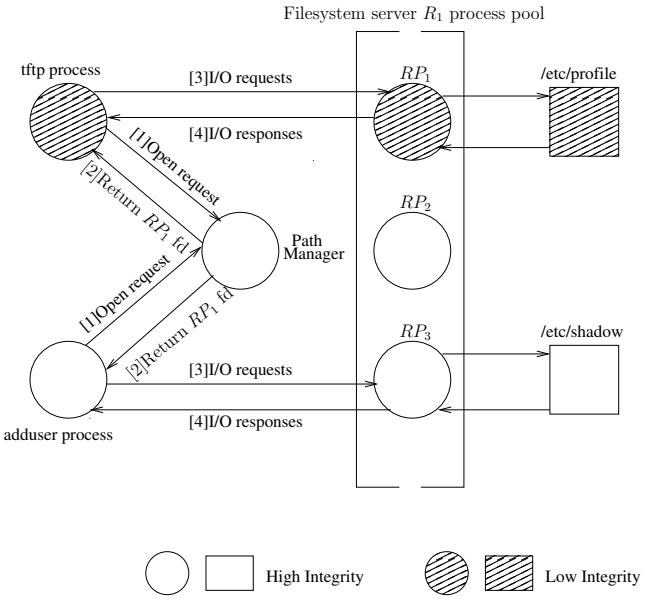

Figure 1. μMIP example

DOI: 10.1002/spe

Example In Figure 1, we show a sequence of operations, their handling by the kernel with μMIP in place, and their consequences on processes' integrity levels. Processes are shown as circles, files as rectangles, shading represents low integrity, and arrows indicate the flow of information.

The example shows an attempt by a *tftp* server to access a sensitive file, the shadow password file, */etc/shadow*, to which we refer simply as 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 part of mediating the open("/etc/shadow") request from *tftp*, the kernel checks the integrity value of that file. As 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* are 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.

In this example, we assume that we have (correctly) configured or caused *tftp*'s integrity level to be low. This would happen, for example, as a consequence of *tftp* receiving network traffic. We could excempt *tftp* from being downgraded in this manner. That depends on our deployment scenario. That is, if, in a particular deployment of the system, we believe *tftp* to be potentially compromised, we would not grant it an exemption from being downgraded.

If an attacker gains control of *tftp*, μMIP protects the system in two ways. First, if the compromised *tftp* sends IPC requests to a high-integrity process, we employ a technique called tag-termination, that causes the high-integrity process to simply drop the request and retain its integrity level. We discuss this in Section 2.4. In particular, if the process to which the compromised *tftp* sends an IPC request is a file-manager process, the file-manager acts as a tag-terminator. Tag-termination also prevents the system from entering a denial-of-service state where high-integrity objects become inaccessible. Second, μMIP 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 1. If it were to open the shadow file, it is forwarded by the kernel to one of the high-integrity servers, exactly like tftp was. When it writes to the high-integrity file, its request is allowed as it is also of high-integrity. If it were to open and read from the low-integrity file */etc/profile*, its requests are forwarded to RP_1 and its integrity value is lowered.

The integrity checks that we discuss above happen in addition to usual file-access checks. In the example above, if *tftp* is executed under a non-root user, it would be kept from reading */etc/shadow* regardless of its integrity value. μMIP's integrity protection relates only to those operations that would already normally be allowed by the usual filesystem access control.

2.1. Processes

In μMIP, a process either has the tag, or does not. Its presence indicates that the process is of high integrity. From the standpoint of μMIP, we have two kinds of servers.

Boot-time servers are responsible for initializing the system at boot-time. An example of such a server is one that performs initial disk management. Boot-time servers terminate after completing the initialization. In μMIP, we trust boot-time servers. As a justification for this trust, we point out that the code for such servers is included in the immutable system image. For example, the boot-time filesystem server is responsible for mounting disks and loading files that are 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.

After the boot process finishes, *run-time servers*, and dynamic integrity tracking and protection become 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. (However, see the exceptions below in Section 2.4.)

DOI: 10.1002/spe

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

Table I. Relationship between DAC permissions and file integrity for administrator-owned files

2.2. *Files*

In a microkernel architecture, each server, including the filesystem server, enforces 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 for files based on their DAC permissions. By "DAC permissions," we mean the effective permissions that a user has from the file-permission bits of the file and the directories that contain it, any file access control lists (ACLs) that are configured on the file and the directories that contain it, and any groups of which a user is a member. This integrity mapping is done for system files only. 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 I summarizes the mapping of DAC permissions to integrity levels for system files. As shown in Table I, 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.

2.3. *How the integrity levels are used*

The integrity level associated with a file and process is used in two ways. One is that 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 restricts the types of messages a low-integrity process can send. Table II shows a list of the sensitive operations that μMIP restricts for low-integrity processes. Some of these are denied to a low-integrity process, and some are handled differently than if the process is high-integrity.

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 II. Sensitive operations in QNX Neutrino and their message types

With respect to the actions in Table II, spawning a new process is allowed, however, a low-integrity process is restricted to spawning low-integrity processes only. A low-integrity process is allowed to write or lock a low-integrity file only. And, a low-integrity process is not allowed to configure a path, nor is it allowed to send an IO_CHMOD message that changes a file's permission.

Apart from restricting the messages a low-integrity process can send successfully, the interactions of a process with a file are also affected with μMIP. Table III indicates the different relationships between processes and files depending on their respective integrity levels. The first three columns

DOI: 10.1002/spe

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.

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 III. 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.
- **Low-integrity process accessing a high-integrity file:** In this scenario, the kernel does not permit the process to modify the file.
- **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 reads, 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.

We point out that the above rules are essentially Biba's rules [4]. As we mention in Section 1, as in UMIP, we incorporate also the discretionary permissions on files.

2.4. Rules for propagating integrity levels

We have two kinds of interactions for which we need to specify how integrity levels are propagated: process-process communication, and process-file access. For the former, as a general rule, whenever an untagged (i.e., low-integrity) process send a message via IPC, the receiver loses its tag, i.e., is downgraded. We recognize that this can quickly cause every process in a system to be downgraded. Consequently, we realize exceptions to this rule via a mechanism called tag-termination.

Any process can be designated as what we call a tag-terminator. What this means is that under specific conditions, that process retains its tag (i.e., high integrity) even if the general rule specifies that it should be downgraded by losing its tag. Two examples of this arise when a process attempts to access a file via the server process that mediates access to the file. We discuss these two examples below in the context of exceptions; under these exceptions, we do not downgrade the server process. Indeed, even if a process receives network traffic, we could deem it to be a tag-terminator so that it retains its high-integrity. We may do this, for example, with a highly trusted server, such as *sshd*.

When a process requests to open a file, the kernel redirects the request to the appropriate server according to the file's integrity level; In the last column of Table III, we specify the resultant integrity level of a process once it accesses a file. As the table indicates, we have two exceptions that we realize using tag-termination.

Both exceptions pertain to servers that mediate accesses to the filesystem. Those servers are not necessarily downgraded even though they receive and service requests for file access (via IPC) from low-integrity processes. The first exception is a read request for a high-integrity file from a low-integrity process. We redirect such a *read* request to the high-integrity filesystem server. The

DOI: 10.1002/spe

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.

2.5. Implementation Issues

To implement μMIP in QNX Neutrino, the most significant change we had to make is to its path manager. As 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 the path manager to perform the following three operations: integrity check, resource manager instantiation, and, cryptography.

2.5.1. Integrity Check As soon as the kernel gets an IO_OPEN request from a process, it looks 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. After lookup, the rules of μMIP are applied. The integrity table is part of the path manager component of the kernel. We have implemented the internal integrity table as a hash table, resolving collisions with chaining (i.e., linked lists). The worst-case complexity of the lookup and insert operations is, therefore, $O(n)$; however, it is constant-time in the average case.

2.5.2. Resource Manager Instantiation After the integrity check, if μMIP allows the file IO operation, the kernel redirects the request to the appropriate resource manager (server). This involves checking the integrity of the file, and choosing between servers of different integrity values if multiple ones exist. At this point, there is a danger that the kernel drops the request, 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, the client process can use a resmgr_attach() call to initialize a new resource manager of the appropriate integrity level.

2.5.3. 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 such as the address of the file, file name and amount of space on the storage media.

As an example, consider the 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. If the disk driver is compromised, the attacker has complete control over the disk. To protect the disk-contents from a compromised disk driver, μMIP uses cryptography. It stores all information about files on the disk in encrypted form. The attacker cannot get 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 should point out also, that we did not address any key-management issues as part of this work.

We have implemented and tested μMIP with QNX Neutrino's embedded transaction filesystem (ETF) [8]. We have used AES encryption to encode the metadata of high-integrity files. Prior work on filesystem encoding such as VPFS [9] and I3FS [10] are complementary to μMIP. VPFS and I3FS can be used with μMIP to encode the entire contents of the disk.

DOI: 10.1002/spe

3. EVALUATION

In this section, we discuss our evaluation of the run-time overhead of our implementation of μMIP in QNX Neutrino. We have modified the message-passing and path-management routines. Excluding the additional code that runs at boot-time and the code for AES, our implementation of μMIP in QNX Neutrino adds 120 lines of C code to the kernel. To measure μMIP's overhead, we compared the original version of the kernel with the modified version on the following benchmarks: lmbench 3.0 [11], unixbench 4.1.0 [12] and iozone 3.53 [13]. We have encrypted the metadata only of files, and not the contents of the file. Our main focus is the overhead of the use of mTags, and therefore the policy component, within μMIP .

Test	**μMIP-enabled**	**μMIP-isolated** (ms)
Dhrystone 2 using register variables [l/s]	4244181	1.77
Double-Precision Whetstone [MWI/s]	549.5	0.73
Execl Throughput[l/s]	340.7	2.32
File Copy 1024 bufsize 2000 maxblocks [KB/s]	21134	5.04
File Copy 256 bufsize 500 maxblocks [KB/s]	11758	0.085
Pipe Throughput [l/s]	60299	23.52
Pipe-based Context Switching[l/s]	32978	30.18
Shell Scripts (8 concurrent) [l/s]	1.07	10.72
System Call Overhead [l/s]	41278	10.11

Table IV. Results for the unixbench benchmark. For μMIP-isolated, we report, in milliseconds, the portion of the second column that we recorded for it.

Test	**μMIP-enabled**	**μMIP-isolated**
syscall	8.80	0.03
read	5.17	0.004
write	4.69	0.096
stat	46.85	0.42
fstat	10.63	0.20
open/close	50.15	2.83
Select on 100 fd's	300.28	3.64
Select on 250 fd's	774.92	10.99
Select on 100 tcp fd's	99.93	3.02
Select on 250 tcp fd's	248.43	2.84
Signal handler installation	0.69	0.02
Signal handler overhead	2.53	0.01
Protection fault	1.85	0.09
Pipe latency	29.97	0.49
AF_UNIX sock stream latency	29.27	0.33
Process fork+execve	6606.59	58.68
Process fork+/bin/sh -c	12291.61	21.74
File write bandwidth	12605.0	32.0
Pagefaults	8462.05	40.78
UDP latency using localhost	36.72	0.27
TCP latency using localhost	36.53	0.20
TCP/IP connection cost to localhost	163.58	1.80

Table V. Results for the lmbench benchmark (μs).

3.1. Benchmarks and Empirical Results

The tagging operations through which integrity is propagated incur overhead for every message pass. Besides the tagging operations added to the message passing mechanism, every file operation also causes a lookup in the internal file integrity table.

Prepared using speauth.cls
DOI: 10.1002/spe

Test	**μMIP-enabled**	**μMIP-isolated** (ms)
write	292609	0.15
rewrite	256002	0.11
read	341339	0.002
reread	409603	0.003
random read	341336	0.003
random write	256000	0.004
bkwd read	409600	0.003
record rewrite	227556	0.004
stride read	341390	0.19
fwrite	227555	0.12
frewrite	227577	0.009
fread	186181	0.005
freread	186183	0.005

Table VI. Results for the iozone benchmark (KB/sec). For μMIP-isolated, we report, in milliseconds, the portion of the second column that we recorded for it.

To measure the collective overhead of these additions, we performed two sets of synthetic system benchmark suites: lmbench and unixbench. 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.

To gauge the filesystem overhead specifically, we ran the iozone benchmark suite dedicated to evaluating file read and write performance. iozone is a filesystem benchmark with focus on file I/O operations. These operations range from simple reads to random reads, and to mmap calls. Put together, these different benchmark suites allow us to confidently evaluate the overhead of μMIP as implemented in QNX Neutrino.

As μMIP extends the file I/O operations to propagate and track the integrity level to the files, we configured the benchmark suites to perform all file I/O operations on a particular mount point where we mounted our modified memory-based filesystem. Both high and low-integrity instances of the memory-based filesystem server ran during the benchmark tests. To measure the overhead caused by the integrity-level propagation, we executed all benchmarks with and without μMIP enabled in the kernel. The test machine ran QNX Neutrino 6.5.0 hosting a 1.8GHz Pentium 4 with 1GB of RAM. All code was compiled with full optimizations. Our timing measurements are from the ClockCycles() call [14], which returns the value of a 64-bit cycle-counter.

Each benchmark suite uses its own data collection and data processing mechanisms. To permit future comparison with our work, we report the raw values that are produced by lmbench, unixbench, and iozone. For example, lmbench collects measurements internally before aggregating the results, unixbench reruns three times before reporting the results, and iozone collects ten measurements.

3.2. Results

Tables IV, V and VI show the results for the unixbench, lmbench and iozone suites, respectively. The first column of each table lists the name of the microbenchmark, the second column lists the results for the kernel with μMIP enabled and the last column shows the time we isolated for μMIP. Each benchmark has its own score. The tests in the unixbench are scored as lines per second, or KB per second as we indicate in Table IV. The lmbench (Table V) is scored in microseconds, and therefore offers an easy way to intuit the μMIP isolated time. The iozone (Table VI) is scored in KB/second.

We computed the mean and standard deviation across at least 50 benchmark iterations. We then computed a 95% confidence interval. If the confidence intervals for the original system and the system with μMIP overlap, then this means that we cannot conclude that the mean values for the two are statistically distinct. In only four cases did we see a statistical difference in introducing

DOI: 10.1002/spe

μMIP, and in these cases, the difference was between 1 and 3% overhead. These higher overheads are seen in those benchmark tests that particularly exercise IPC; for example, Process fork+execve and TCP/IP connection cost to localhost in the lmbench tests in Table V, and stride read in the iozone in Table VI.

We attribute μMIP 's small overhead to our small code-size (120 lines in C), and the fact that we exclude the time for the AES operations. Our results are not surprising, given prior observations for similar systems. For example, the overhead of IPC introduced by Flask [15] over the underlying system (Fluke, in their case) is between 1 and 9%, which is roughly what we observe for our benchmark tests that are close to pure IPC calls, e.g., syscall in Table V for the lmbench. Similarly, the work on Janus [16] reports negligible overhead in two applications that they use to assess their system. For other benchmark tests, for example those shown in Table IV, the bulk of the time taken by a test does not involve the propagation of mTags. For example, the test that involves pipe throughput in Table IV involves only a few tag propagations at the time the pipe is setup and first used. The bulk of time is in reading from and writing to the pipe. We comment further on our tests and results in comparison with those from similar prior work in the next section on Related Work.

It is certainly possible that the same implementation of μMIP in a different microkernel-based system would result in a more significant overhead. We chose QNX for this work given our access to and familiarity with it, and the fact that it is a widely-deployed commercial system. We propose to investigate the use of μMIP for other microkernel-based systems in future work. In the previous section, we justified our choice of benchmarks. Our results are limited by our benchmarks. It is certainly possible that for applications whose profile is different from our benchmarks, the overhead is more significant.

Indeed, it is easy to come up with a scenario in which μMIP severely impedes the performance of the kernel. An example is a low-integrity process that rapidly sends IPC requests to all high-integrity server processes. In this case, it is likely that each high-integrity server process acts as a tag-terminator (see Section 2.4), thereby invoking additional code, depending on the kind of IPC request that is received. Furthermore, the kernel is a bottleneck. This can cause significantly additional overhead than we see from the benchmarks. More generally, as we observe from the "μMIP isolated" column from our three tables above, IPC-intense tasks and processes will see a higher overhead from μMIP.

4. RELATED WORK

The μUMIP model relates to the past work along three aspects: integrity models, systems security and labeling techniques.

From the standpoint of integrity models, the Biba integrity model [4] can be seen as the basis for our work, and several other recent pieces of work, including UMIP, on mandatory security. However, there are some important differences between the work of Biba, and ours, which is based on UMIP. An important difference is that UMIP and μMIP rely not only on the integrity level of a file, but also its DAC permissions. UMIP [2] is recent work on integrity propagation and the one we have adopted for 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. UMIP can be differentiated also from LOMAC [5], which is more rigid than UMIP in how it handles the integrity level of an object (e.g., a file). The Clark-Wilson model [17] is quite different from work such as ours that is based on Biba's model. In the Clark-Wilson model, we attach 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.

Microsoft Windows Vista [18] 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

DOI: 10.1002/spe

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 [19] and PACL [20]. 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.

From the standpoint of past work in systems security, AppArmor [21] 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 [7] 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 [6] 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. Among other research-based microkernels, MACH [22], Flask [15], Janus [16], DTE Unix [23] and Mungi OS [24] provide mandatory access control.

Of these, only Flask and Janus are implemented on top of existing systems, and allow us to compare our results with theirs. Janus provides empirical results for two applications (mpeg_play and ghostscript) only. The observation there is that the performance penalty added by Janus over the existing system is negligible, and barely statistically significant. In other words, our observations for μMIP and theirs for Janus are similar.

The overhead seen in Flask is similar to ours as well. They report an overhead of between 1 and 9% on IPC calls. We observe that in certain IPC-intense tests, we get an overhead of between 1 and 3% with μMIP. Note that in some ways, Flask and μMIP are very different systems, and therefore a direct comparison of the two systems is difficult. Specifically, Flask caches security decisions in the Object Manager, which is a process that mediates accesses to objects. In μMIP, on the other hand, we associate tags with processes. Thus, the difference between the two systems is similar to the difference between Access Control Lists (ACLs) which are bound to objects, and capabilities, which are bound to subjects.

From the standpoint of labeling techniques, Asbestos [25] presents 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 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 tagging mechanism the circulation of tags is insensitive to different levels of processes and does not authenticate or restrict processes to send or receive information from other processes. In our work, tags are attached to processes to, for instance, maintain the integrity of the system rather than to control their interaction. Also, our approach implements the tracking of information flow with minimum overhead.

The labeling approaches like HiStar [26] and LoStar [27] 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.

DOI: 10.1002/spe

The Data Tomography [28] 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 processes 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 implementable in a real-time kernel rather than using it just for instrumenting purposes.

TaintDroid [29] is an Android-based system that introduces an information-flow tracking system. TaintDroid provides taint tracking at different levels i.e., message level tracking, variable level tracking and method level tracking. TaintDroid attaches a taint tag with the variable. The VM interpreter stores the taint tag in a virtual taint map and propagates those tags according to data flow rules. The taint tag storage mechanism in TaintDroid may result in large memory overhead.

5. CONCLUSION

We have discussed our case-study of realizing a particular approach to mandatory security. Our focus in the use of our approach, called mTags, to tagging and tag-propagation between processes and threads. We have discussed our design and implementation, which we call μMIP. While our implementation is on a microkernel-based system, we have pointed out that it is applicable in any process- or thread-based system that uses message-passing for IPC. We have discussed our implementation of μMIP in the QNX Neutrino commercial microkernel-based system. Apart from discussing design and implementation issues, we have presented empirical results for the overhead imposed by μMIP across three well-established benchmarks for POSIX-compliant systems. The overhead from the policy component of μMIP appears to be small.

As future work, an aspect for us to investigate are approaches such as UMIP and others for mandatory security in newer microkernels than QNX. Specifically, whether they provide anything more than the features that those microkernels already provide. We seek to also 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 [30] 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.

REFERENCES

1. de Oliveira AB, Saif Ur Rehman A, Fischmeister S. mtags: augmenting microkernel messages with lightweight metadata. *SIGOPS Oper. Syst. Rev.* Jul 2012; **46**(2):67–79, doi:10.1145/2331576.2331587.
2. Li N, Mao Z, Chen H. Usable mandatory integrity protection for operating systems. *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, IEEE Computer Society: Washington, DC, USA, 2007; 164–178, doi:http://dx.doi.org/10.1109/SP.2007.37. URL http://dx.doi.org/10.1109/SP.2007.37.
3. Hildebrand D. An Architectural Overview of QNX. *Proc. of the Workshop on Micro-kernels and Other Kernel Architectures*, USENIX Association: Berkeley, CA, USA, 1992; 113–126, doi:http://doi.acm.org/10.1145/1269843.1269857.
4. Biba. Integrity Considerations for Secure Computer Systems. *MITRE Co., technical report ESD-TR 76-372* 1977; .
5. Fraser T. Lomac: Low water-mark integrity protection for cots environments. *Proceedings of the IEEE Symposium on Security and Privacy*, 2000; 230 –245, doi:10.1109/SECPRI.2000.848460.
6. National Security Agency – Central Security Service, Security-Enhanced Linux. http://www.nsa.gov/research/selinux/index.shtml, accessed November 2012.
7. securelevel. BSD kernels. http://www.freebsd.org/doc/en/books/faq/security.html.
8. QNX. Embedded transaction file system. http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/fsys.html#ETFS.

9. Weinhold C, Härtig H. Vpfs: building a virtual private file system with a small trusted computing base. *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, ACM: New York, NY, USA, 2008; 81–93, doi:http://doi.acm.org/10.1145/1352592.1352602. URL http://doi.acm.org/10.1145/1352592.1352602.
10. Patil S, Kashyap A, Sivathanu G, Zadok E. Fs: An in-kernel integrity checker and intrusion detection file system. *Proceedings of the 18th USENIX conference on System administration*, USENIX Association: Berkeley, CA, USA, 2004; 67–78. URL http://portal.acm.org/citation.cfm?id=1052676.1052684.
11. Larry McVoy CS. lmbench. http://www.bitmover.com/lmbench/.
12. unixbench. Byte Magazine. http://code.google.com/p/byte-unixbench/.
13. Norcott WD. Filesystem benchmark iozone. http://www.iozone.org/.
14. QNX Software Systems. ClockCycles(). http://www.qnx.com/developers/docs/6.5.0_sp1/index.jsp?topic=%2Fcom.qnx.doc.neutrino_lib_ref%2Fc%2Fclockcycles.html&cp=13_5_6_31, accessed April 2013.
15. Spencer R, Smalley S, Loscocco P, Hibler M, Andersen D, Lepreau J. The flask security architecture: System support for diverse security policies. *in Proceedings of The Eighth USENIX Security Symposium*, 1999; 123–139.
16. Goldberg I, Wagner D, Thomas R, Brewer EA. A secure environment for untrusted helper applications (confining the wily hacker). *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, USENIX Association: Berkeley, CA, USA, 1996; 1–1. URL http://dl.acm.org/citation.cfm?id=1267569.1267570.
17. Clark DD, Wilson DR. A comparison of commercial and military computer security policies. *Security and Privacy, IEEE Symposium on* 1987; **0**:184, doi:http://doi.ieeecomputersociety.org/10.1109/SP.1987.10001.
18. Windows Integrity Mechanism. http://msdn.microsoft.com/en-us/library/bb625957.aspx.
19. McIlroy MD, Reeds JA. Multilevel security in the unix tradition. *Softw. Pract. Exper.* August 1992; **22**:673–694, doi:10.1002/spe.4380220805. URL http://portal.acm.org/citation.cfm?id=139006.139012.
20. Wichers D, Cook D, Olsson R, Crossley J, Kerchen P, Levitt K, Lo R. PACLs: An access control list approach to anti-viral security. *Proc. of the 13th National Computer Security Conference*, 1990; 340–349.
21. Immunix. Apparmor. https://apparmor.wiki.kernel.org/index.php/Main_Page.
22. Accetta M, Baron R, Bolosky W, Golub D, Rashid R, Tevanian A, Young M. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the USENIX Summer Conference*, 1986; 93–112.
23. Badger L, Badger L, Sterne DF, Sterne DF, Sherman DL, Sherman DL, Walker KM, Walker KM, Haghighat SA, Haghighat SA. A domain and type enforcement unix prototype. *In Proceedings of the Fifth USENIX UNIX Security Symposium*, 1996; 127–140.
24. Heiser G, Elphinstone K, Vochteloo J, Russell S. The mungi single-address-space operating system. *Software Practice and Experience* 1998; .
25. Efstathopoulos P, Krohn M, VanDeBogart S, Frey C, Ziegler D, Kohler E, Mazières D, Kaashoek F, Morris R. Labels and event processes in the asbestos operating system. *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, ACM: New York, NY, USA, 2005; 17–30, doi:http://doi.acm.org/10.1145/1095810.1095813. URL http://doi.acm.org/10.1145/1095810.1095813.
26. Zeldovich N, Boyd-Wickizer S, Kohler E, Mazires D. Making information flow explicit in histar. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, USENIX Association: Berkeley, CA, USA, 2006; 19–19. URL http://portal.acm.org/citation.cfm?id=1267308.1267327.
27. Zeldovich N, Kannan H, Dalton M, Kozyrakis C. Hardware enforcement of application security policies using tagged memory. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, USENIX Association: Berkeley, CA, USA, 2008; 225–240. URL http://portal.acm.org/citation.cfm?id=1855741.1855757.
28. Mysore S, Mazloom B, Agrawal B, Sherwood T. Understanding and visualizing full systems with data flow tomography. *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, ACM: New York, NY, USA, 2008; 211–221, doi:http://doi.acm.org/10.1145/1346281.1346308. URL http://doi.acm.org/10.1145/1346281.1346308.
29. Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, USENIX Association: Berkeley, CA, USA, 2010; 1–6. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.
30. Provos N. Preventing privilege escalation. *In Proceedings of the 12th USENIX Security Symposium*, 2003; 231–242.