

Countering IPC Threats in Multiserver Operating Systems

(A Fundamental Requirement for Dependability)

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum
VU University Amsterdam, FEW/CS, De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
E-mail: {jnherder, herbertb, beng, philip, ast}@cs.vu.nl, Phone: +31 6 2489 7177

Abstract

Multiserver operating systems have great potential to improve dependability, but, paradoxically, are paired with inherently more complex interprocess communication (IPC). Several projects have attempted to run drivers and extensions in isolated protection domains, but a systematic way to deal with IPC threats posed by untrusted parties is not yet available in the literature. IPC is fundamental to the dependability of multiserver systems.

In this paper, we present a classification of IPC threats in multiserver systems with unreliable and hostile senders and receivers, such as resource exhaustion, spoofing, and unauthorized access. We also introduce an extended asymmetric trust model, describing two new IPC vulnerabilities relating to caller blockage. Based on our classification of IPC threats we present the IPC defense mechanisms and architecture of MINIX 3.

Keywords: Classification of IPC Threats, Dependable IPC Architecture, Multiserver Operating Systems

1 IPC MUST BE DEPENDABLE

Multiserver systems have great potential to improve operating system dependability. By running untrusted code in separate protection domains, bugs are better contained than in monolithic systems. In addition, the modular designs provide better abstractions and debugging and development support. Regardless of the reasons for using a modular multiserver design, one important caveat is that services can no longer make direct function calls. Instead, the kernel provides IPC services that allow sequential processes to synchronize and communicate [8]. However, since software cannot be trusted to be correct [1, 5, 12, 26, 28], this poses new challenges with respect to IPC threats caused by, for example, programming bugs, design and run-time faults, or misconfigured extensions, or malicious intent.

To get an impression of what multiserver IPC entails, it is important to realize that seemingly simple system calls can cause intricate IPC patterns, whereas, applications can perform thousands of system calls per second. In MINIX 3, for example, a *read* or *write* call may lead to over 25 IPC messages between the application, virtual file system, file

server, disk driver, and kernel. Measurements on Mac OS X and OpenDarwin, which use the Mach IPC mechanism, reported 102,885 and 29,895 messages from system boot until the shell is available, respectively [27]. We conducted the same measurement on MINIX 3 and obtained a similar number of 61,331 messages. Another experiment with *ls -alR* in */usr/src* yielded 68,973 messages, whereas *wget minix3.org* initiated 4,862 messages. Background activity, such as the random driver gathering entropy, caused 476 messages/sec—with one IPC call taking in the order of a microsecond, this costs less than 0.05% of the CPU.

The above examples constitute legitimate IPC, but without precautions, buggy senders and receivers (as shown in Fig. 1), can easily disrupt the system. Early tests on MINIX 3 already showed that the system could be deadlocked if a driver violated its IPC protocol [20]. More recently, fault-injection tests revealed other potentially fatal IPC operations, such as unauthorized IPC endpoints, bad message buffers and unexpected message contents. For example, one experiment in which we injected 1,000,000 randomly selected faults led to, among other things, 218,962 IPC calls denied by the IPC subsystem and 1,524,516 unexpected IPC requests for the kernel. The experiment is further detailed in Sec. 5. Important for now is that these faults would have wreaked havoc on the system if it were not for our protection mechanisms.

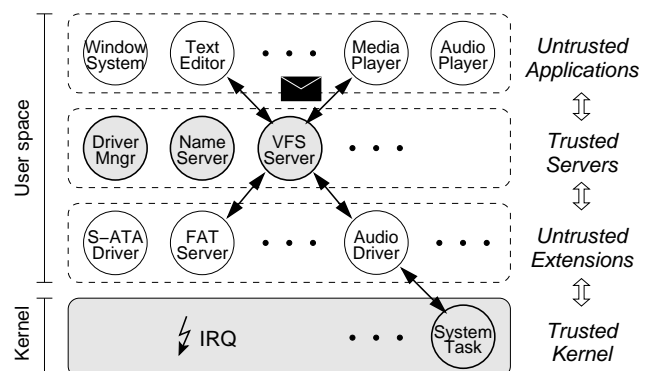


Figure 1: Design of a typical multiserver operating system where IPC is done between layers with varying degrees of trust.

1.1 Threat Model

Before we present our exact threat model, we briefly introduce the reader to a generalized multiserver operating system model, shown in Fig. 1. In such a design, applications and drivers are generally provided by untrusted third parties, whereas the core operating system servers and kernel tasks are assumed to be correct. Typical IPC interactions thus involve varying trust relationships, between both trusted and untrusted clients and servers. These trust relationships are representative for operating systems in general, including Windows, FreeBSD and Linux.

In particular, we define an *IPC threat* as any unintended IPC action that originates in a buggy component and that may disrupts the core operating system. For example, consider a buggy driver that does not respond to a request, sends the reply to the wrong party, or corrupts the message contents. Malicious drivers are outside the scope of this work, since they could do far more dangerous things such as wiping the entire disk. We are concerned with problems relating to (a) the IPC subsystem itself, (b) individual message delivery, and (c) group interactions with untrusted processes. Sec. 2 further classifies these IPC threats.

The focus of this work is IPC interactions with third-party drivers and extensions. Extensions not only can have intricate communication patterns with multiple parties, but also comprise up to 70% of the operating system code and have error rates 3–7 times higher than other Linux kernel code [3]. Driver code was shown to be responsible for up to 85% of the crashes in operating systems like Windows XP [25]. While multiserver systems potentially provide better isolation of untrusted code, the design of a dependable IPC architecture poses several challenges.

1.2 Basic IPC Principles

Several basic IPC approaches can be distinguished. First, synchronous IPC is a two-way interaction, where the initiating process blocks and waits for the other party. Only if both parties are ready, is the message copied or mapped from the sender to the receiver and both resume execution. Second, asynchronous IPC allows the caller continuing immediately by buffering the message. The IPC subsystem then delivers the message at the first opportunity on behalf of the caller. Both approaches may be combined with a timeout to abort the IPC call if it did not succeed within the specified interval. However, finding sensible timeout values is nontrivial, so zero or infinite timeouts are commonly used. The former is also referred to as nonblocking IPC; delivery is tried once and the status is immediately returned.

Looking at how IPC is implemented, we assume the IPC properties listed in Fig. 2 to be present. The stub code linked with the application puts the IPC parameters on the stack or in CPU registers and executes a trap instruction, causing a

software interrupt that puts the kernel in control. The kernel is responsible for the actual message passing and guarantees the properties listed. For example, consider how reliable delivery, integrity and confidentiality of synchronous IPC are implemented: depending on the destination address' validity, either an error is raised or the message is copied or mapped to the party specified, but there is no middle ground. Atomicity and isolation are harder to achieve if the kernel is reentrant or multithreaded, but techniques to do so are also known and can be found in class room text books.

IPC property	Explanation
Atomicity	IPC is either delivered or not
Reliable delivery	Messages can never get lost
Trusted addressing	IPC endpoints cannot be forged
Message integrity	Message contents is preserved
Isolation	Multiple IPC calls are independent
Confidentiality	Snooping on IPC traffic is impossible

Figure 2: IPC properties provided by the kernel implementation.

1.3 Performance Considerations

This work is not concerned with IPC performance because several research groups already have focused on optimizing IPC performance. For example, work on L4 showed how the costs of message passing can be reduced to a few microseconds by putting arguments in CPU registers, memory mapping, and minimizing cache-miss rates [9, 10, 18]. Furthermore, SawMill Linux [7] showed how trade-offs in the design of multiserver IPC protocols affect performance. For example, shifting functionality from servers to clients or data sharing helps to minimize context switches and copying. All these efforts have proven to be extremely helpful to increase the performance and usability of multiserver systems. It was shown, for example, that overhead of a multiserver system can be within 5% of a monolithic design [10]. Nevertheless, performance does not help increasing dependability, which we think is a more urgent need for the acceptance of today's multiserver systems.

1.4 Contribution and Paper Outline

Motivated by the above examples and backed by several years of hands-on experience with multiserver operating systems, in this paper we address the design of a dependable IPC architecture. While several projects attempt to contain unreliable drivers [14, 17, 25] and others explicitly take driver exploits into account [2], IPC threats posed by untrusted components have not yet been addressed. Some problems have been pointed out before [6, 15, 19, 22, 23], but a thorough classification of all IPC threats is lacking in the literature (Sec. 2). Moreover, we recently identified two new vulnerabilities in synchronous IPC for which

we present an extended asymmetric trust model (Sec. 2.2). Then we discuss the design of a dependable IPC architecture using MINIX 3 as a case study. We first present the design choices and trade-offs that led to the MINIX 3 IPC infrastructure (Sec. 3). Then we show how and why this infrastructure can be used safely in multiserver operating systems (Sec. 4). We also evaluate our design using fault-injection experiments on a prototype implementation (Sec. 5). Finally, we survey related work (Sec. 6) and present our conclusions (Sec. 7).

2 MULTISERVER IPC THREATS

Even if the IPC invariants listed in Fig. 2 are in place, there are still numerous IPC threats in multiserver systems. For example, a driver that sends arbitrary messages or violates the message passing protocol in another way may cause denial of service if countermeasures are lacking.

2.1 Classification of IPC Threats

We identified three orthogonal classes of IPC threats relating to (a) the IPC subsystem itself, (b) message delivery, and (c) group interactions with untrusted processes. This led to the classification shown in Fig. 3 and discussed below.

(a) IPC Subsystem: Exceptional IPC requests can potentially be used as attacks on the IPC subsystem itself (rather than on other processes) in the following two ways.

a1. Call parameters: The IPC call parameters passed by the user may be invalid. Typically this includes the IPC primitive, the message buffer and an IPC endpoint, but depending on the IPC infrastructure, other user input such as timeout values or IPC capabilities also have to be checked. Examples of wrong input include a disallowed IPC call, an invalid target address and a pointer to a message buffer outside a process' address space.

a2. Global resources: The use of global resources, such as message buffers, might lead to resource exhaustion when one or several clients collectively perform too many IPC requests. The IPC subsystem may run out of memory if message buffers are dynamically allocated. CPU exhaustion could occur if processing takes indefinitely. Misattribution of costs is one of the underlying causes [22].

(b) Message Delivery: Although the IPC invariants of reliable delivery, trusted addressing and message integrity guarantee messages will be delivered to the destination specified, problems relating to message delivery still exist.

b1. Addressing: We identified three problems for IPC between two processes, A and B. First, A may not be authorized to access services of B and thus should not be allowed sending an IPC message. Second, a message may be sent to the wrong party due to a mistake. Third, spoofing is a threat

Class (a) IPC Subsystem	
<ul style="list-style-type: none"> • <i>a1. Call parameters</i> + Bad IPC primitive + Nonexisting endpoint + Illegal message buffer 	<ul style="list-style-type: none"> • <i>a2. Global resources</i> + Memory exhaustion + CPU exhaustion
Class (b) Message Delivery	
<ul style="list-style-type: none"> • <i>b1. Addressing</i> + Unauthorized target + Unintended target + Spoofing - Sender spoofing - Receiver spoofing 	<ul style="list-style-type: none"> • <i>b2. Message contents</i> + Oversized message - Memory corruption - Unhandled page fault + Bad contents - Malformed message - Unauthorized request - Message timing
Class (c) Group Interactions	
<ul style="list-style-type: none"> • <i>c1. Flow control</i> + Scheduling order - Priority inversion - Starvation + Denial of service - Service degradation - Resource exhaustion 	<ul style="list-style-type: none"> • <i>c2. Caller blockage</i> + Deadlocks - Cyclic dependency - Unexpected state + Asymmetric trust - Untrusted client - Untrusted server

Figure 3: Classification of IPC threats. Class (a) represents bad things done to the IPC subsystem; classes (b) and (c) represent misuse of IPC facilities that potentially harms the system.

if IPC endpoints are not uniquely associated with actual services. For example, such semantics is lost in dynamically configured systems where services are started on the fly and may be assigned any endpoint. An IPC party may fake its real identity either by advertising itself as somebody else (receiver spoofing) or forging the source address in the IPC message (sender spoofing).

b2. Message contents: The IPC subsystem only passes messages from one process to another and is not concerned with the message contents. If dynamic payload lengths are used, oversized messages might cause buffer overruns and lead to memory corruption if the receive buffer is too small. Furthermore, if processes are *self-paged*, the sender may be blocked if the receiver does not handle a page faults caused by messages that do not fit in the receive buffer [22]. Even if a message is successfully received, the recipient may be surprised by unexpected input from a buggy sender and not be able to handle the message. The message may contain, for example, a bad request or argument or the receiver may be in an incorrect state to handle the IPC.

(c) Group Interactions: Finally, multiserver IPC present the following IPC threats relating to group interactions in the face of unreliable and hostile senders and receivers.

c1. Flow control: The continuity of IPC traffic and represents message queuing and process scheduling problems rather than a fundamental IPC problem. Processes in a multiserver system generally run with a given scheduling pri-

ority, which may cause priority inversion and starvation of low-priority processes [23]. A more interesting threat is denial of service, for example, by flooding a server in a coordinated attack [19]. This may lead to service degradation in other parts of the system or, if requests cause state to accumulate at the receiver, resource exhaustion.

c2. Caller blockage: When synchronous IPC is used, a process that wants to send a message may be blocked when the callee is not in the correct state to receive the message, and vice versa. This can have various causes. A deadlock can occur if processes that want to communicate with each other have a cyclic dependency and are all waiting for each other. Another problem is asymmetric trust relationships where the caller cannot be sure if the other party is willing to cooperate. Problems relating to untrusted clients that are not willing to receive a reply were already known [22], but we recently identified two new vulnerabilities relating to device drivers acting as untrusted servers that are not listening or responding, as detailed in Sec. 2.2.

2.2 Extended Asymmetric Trust Model

Asymmetric trust was originally defined in the context of EROS as a relationship that “occurs whenever multiple client applications call a single server application,” since an *untrusted client* may block a server using synchronous IPC [22]. However, this client-centric definition of asymmetric trust is not sufficient, since the problem of device drivers acting as *untrusted servers* is ignored. Although in-kernel drivers are trusted in EROS, this is not generally the case. In multiserver systems like MINIX 3 drivers run independently, and are, in fact, untrusted servers to components at a higher level. We have seen the fatal consequences in practice [20]. Therefore, we extend the asymmetric trust model to include both untrusted clients and untrusted servers. To the best of our knowledge, this threat has not been acknowledged in operating system context.

The precise threats relating to caller blockage due to un-

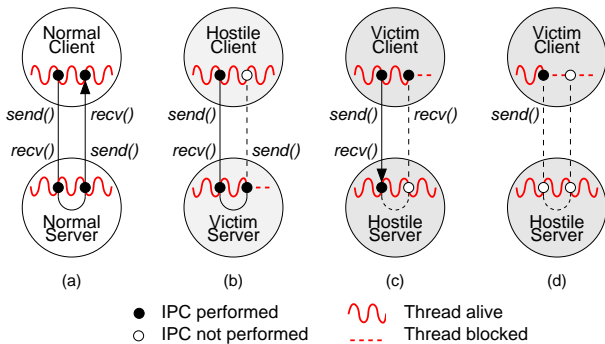


Figure 4: Asymmetric trust and vulnerabilities in synchronous IPC: (a) normal client-server roundtrip, (b) untrusted client blocks server, and (c,d) untrusted server blocks client [NEW].

trusted clients and untrusted servers that communicate using synchronous IPC are shown in Fig 4. Fig 4(a) shows a normal IPC roundtrip, in which a client sends a request and awaits the reply from the server. Fig 4(b) illustrates the original asymmetric trust model in which a client can block a server by not receiving the server’s reply. Finally, Fig. 4(c) and (d) give the two new vulnerabilities that we identified: a client can be blocked by an untrusted server that is not willing to receive a request or send a reply.

Asymmetric trust is most important if one trusted party interacts with multiple untrusted parties in parallel. A client that depends on a server to make progress basically has to trust fully the server—since there is not much practical difference between blocking or unwittingly getting the wrong answer. However, if a single client happens to be a *decoupling point* and has a 1:N relationship with several untrusted servers, blockage on a single defective server cannot be tolerated. For example, consider the virtual file system (VFS) in Fig. 1. One defective driver should not be able to hang the VFS and disrupt services provided by other drivers.

3 MINIX 3 IPC INFRASTRUCTURE

Based on the above threat model, we have developed a dependable IPC architecture for the MINIX 3 multiserver operating system. This section discusses the rationale behind MINIX 3’s design and briefly describes its IPC infrastructure. Sec. 4 discusses in detail how each of the IPC threats presented in Sec. 2 are countered.

3.1 Design Choices and Trade-offs

While searching for suitable building blocks for our IPC defenses, it became soon evident that the IPC threats relating to group interactions (Sec. 2.1, class (c)), and caller blockage due to asymmetric trust (Sec. 2.2) in particular, are most important for the high-level design. While the other IPC threats also must be dealt with, the possible countermeasures against unsafe group interactions are most influential on the implementation style. Defenses can be based on (1) language-based solutions, (2) IPC timeouts, (3) multithreading, or (4) asynchronous IPC. (Concrete systems using these approaches are discussed in Sec. 6.) Each approach comes with a certain complexity:

1. Language support can be used to verify formally that group interactions are safe. However, this approach comes with a complex tool chain and run-time system. Moreover, such high-level defenses, in the end, must be supported by a lower-level IPC infrastructure, such as the solutions below.
2. Timeouts abort the IPC when it does not succeed within a predefined period of time. Timeout values are

Primitive	Semantics	Storage	Mode	Blocking
int SEND(int dst, message *ptr)	Block until message is sent	Caller	Synchronous	Blocking
int RECEIVE(int src, message *ptr)	Block until message arrives	-	Synchronous	Blocking
int SENDREC(int dst, message *ptr)	Send request and await reply	Caller	Synchronous	Blocking
int NBSEND(int dst, message *ptr)	Send iff peer is receiving	Caller	Synchronous	Nonblocking
int ASEND(asynmsg *table, int size)	Buffered delivery by kernel	Caller	Asynchronous	Nonblocking
int NOTIFY(int dst)	Event signaling mechanism	Kernel	Asynchronous	Nonblocking

Figure 5: The synchronous, asynchronous and nonblocking IPC primitives implemented by MINIX 3's IPC subsystem.

hard to get correct for programmers, however, and arbitrary or overly conservative timeout values are not uncommon. Furthermore, superfluous, late reply messages complicate exception handling. Finally, timeouts may cause extended periods of blockage.

3. With multithreading a new thread can be spawned for handling IPC interactions with an untrusted party. This approach requires a more complex thread-aware IPC subsystem. Multithreading also introduces synchronization issues and may be susceptible to resource exhaustion when the system runs out of threads.
4. Asynchronous IPC designs avoid caller blockage by using nonblocking IPC whenever talking to untrusted parties such as drivers. This introduces complexity for maintaining message queues. It also requires a state machine-like programming model since request and reply messages do not necessarily arrive in order.

The IPC defenses in MINIX 3 are based on the fourth approach, asynchronous and nonblocking IPC. An important reason for selecting this approach is that it required only a few changes to the original IPC model found in MINIX 2: the asynchrony is only applied at a few well-defined decoupling points where (trusted) system servers have to communicate with (untrusted) device drivers. In particular, changes to the virtual file system (VFS) and network server (INET) were required, but most drivers remained unchanged. Moreover, VFS and INET are likely to remain stable over time, whereas drivers come and go with hardware changes.

Now that MINIX 3 uses asynchronous IPC to deal with untrusted drivers, the VFS and INET servers have been reprogrammed as a state machine: after sending a driver request, they save the peer's state, return to the main loop in order to await the reply message (or a new request), and look up the caller's status if an IPC message comes in. This also facilitates recovery after a driver crash because pending requests can be replayed from the queue [13].

3.2 Summary of IPC Infrastructure

We now briefly describe the IPC infrastructure implemented in MINIX 3.

IPC Primitives The IPC primitives of MINIX 3 are summarized in Fig. 5 and support both synchronous, asynchronous, and nonblocking IPC semantics. The most basic primitives are the synchronous, blocking SEND and RECEIVE. The SENDREC primitive combines these primitives in a single call, doing a synchronous send following by an implicit receive. No buffering is required as the caller is automatically blocked until the message has been copied from the sender to the receiver. The remaining IPC primitives are nonblocking. NBSEND is a nonblocking variant of the synchronous SEND call that returns an error if the other party is not ready at the time of the call. ASEND supports asynchronous IPC with buffering of messages local to the caller. The caller is not blocked, but immediately returns, and the kernel scans the table with messages to be sent, promising to deliver the messages at the first opportunity. Finally, the asynchronous NOTIFY primitive supports servers and drivers in signaling system events. The call is nonblocking and if the notification cannot be delivered directly, the kernel marks it pending in a statically allocated bitmap in the destination's process structure.

IPC Endpoints In MINIX 3, each IPC endpoints uniquely identifies a single processes with respect to IPC. Endpoints are defined as 32-bit integers consisting of the process' 16-bit slot number in the kernel's process table and a 16-bit generation number. In order to disambiguate between processes that may (over time) occupy the same slot, the generation number is incremented every time a new process occupies a process table slot. Slot allocation is done round robin in order to maximize the time before endpoint reuse. (Note that we solely focus on buggy rather drivers and do not protect against malicious drivers attempting to overtake a IPC endpoint using a brute-force attack that quickly cycles through the 16-bit generation number space.) Moreover, in the event that a system server exits, all dependent processes will be immediately notified about the invalidated endpoint. This design ensures that IPC directed to an exited process cannot end up at a process that reuses a slot.

IPC Message Format For reasons of simplicity, we use only small, fixed-length messages. The message type is defined as a structure with a fixed message header—consisting

of an IPC endpoint and the message type—followed by a union of different payload formats to hold the message arguments. For all IPC primitives, the message source is reliably patched into the message by the kernel, whereas the sender can freely decide on the message type and payload. This way the receiver can always check who called.

To counter the argument that dynamic payloads are necessary [22], we briefly introduce the reader to the *safe copy* scheme we use to exchange data that does not fit an IPC message. The process that wants to grant access to its memory, needs to create a capability specifying the base and size of the memory area (anywhere in the address space), the grantee’s IPC endpoint, and whether reading, writing, or both is allowed. This capability resides in a table in the owner’s address space and an index to it can be sent to other processes using ordinary IPC. A safe copy then is a capability-protected kernel call to copy data to or from the specified memory area. Since the kernel can access all memory, it can quickly check the capability before copying the data. Compared to dynamic payloads, the same functionality is realized, while maintaining the simplicity and reliability of fixed-size messages.

IPC Restriction Mechanisms Since IPC is a powerful construct that should not be available to unprivileged tasks, we enforce two per-process restrictions. First, we restrict the set of IPC primitives available to each process. Second, we restrict which services a process can send to using *send masks*. In principle, send masks can be used to restrict the possible destinations for each individual IPC primitive, but policy definition in the face of multiple, per-primitive IPC send masks proved impractical. Therefore, send masks restrict the allowed IPC destinations regardless of the primitive that is used. Furthermore, send masks are defined as a symmetric relation; if A is allowed to send a request to B, B is also allowed to send the response to A. Receiving is not protected, since it is meaningful only if an authorized process sends a message. These protection mechanisms are implemented by means of compact bitmaps that are statically declared as part of the process table. This is space efficient, prevents resource exhaustion, and allows fast permission checks based on simple bit operations.

In addition, we have implemented deadlock detection to prevent infinite blockage. This is a structural, system-wide restriction that is enforced by the kernel at run-time. The IPC subsystem checks for cyclic dependencies in synchronous IPC interactions and returns an error to the caller that completes the cycle. The only nonfatal cycle is the common case in which two parties directly SEND and RECEIVE to each other. Deadlock detection is especially useful for debugging purposes, but is not practical to rely on in standard communication patterns. Therefore, the design of a deadlock-free message-passing protocol is still needed.

4 IPC DEFENSE MECHANISMS

With the necessary background provided by Sec. 3, we now discuss in detail the defense mechanisms of MINIX 3’s IPC architecture. Fig. 6 gives an overview of IPC interactions for a representative set of components, as implemented in MINIX 3. Several IPC patterns stand out. For example, untrusted clients that request services from the virtual file system (VFS) or system task use a SENDREC rendezvous, whereas the response is always sent using non-blocking NBSEND. Furthermore, driver requests are sent using asynchronous ASEND, whereas drivers use synchronous SEND for the reply message. Finally, asynchronous NOTIFY is used for system events such as hardware interrupts and timeout notifications. The following sections detail why and how and why these and other design choices safeguard the system’s dependability for each class of the IPC threats discussed in Sec. 2.

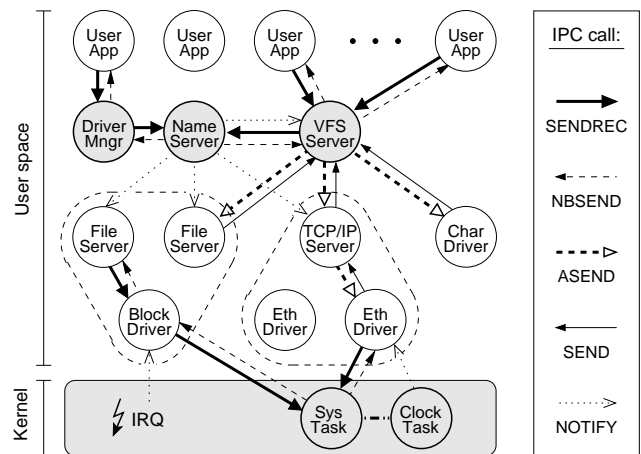


Figure 6: IPC patterns to safeguard the system’s dependability in the face of asymmetric trust. Trusted parts are in gray.

4.1 Defenses (a): IPC Subsystem

a1. Call parameters: All user input is checked for validity. Depending on the call, this concerns the IPC primitive, the IPC endpoint, message buffer or table with asynchronous messages. The validity of the IPC primitive is determined in a switch statement, which returns an error in the default case. The IPC endpoint is verified by checking that (1) the process number part points to a valid entry in the kernel’s process table and (2) the generation number found in that process table entry matches. The pointer to the message buffer or table with asynchronous messages is verified by comparing the process’ memory map kept in the kernel’s process table with the buffer’s base and length. Any problem causes an error to be returned to the caller.

a2. Global resources: Resource exhaustion is structurally prevented since the IPC subsystem does not use dynamic resource allocation. The synchronous SEND and RECEIVE calls use a rendezvous, meaning that messages are never buffered, but always directly copied from sender to receiver. The asynchronous primitives cannot cause resource exhaustion. ASEND uses storage in the caller's address space plus limited kernel administration for pending messages. All space needed is statically allocated: the address and size of the ASEND table are kept in the caller's process table entry and a bitmap indicating that asynchronous messages are pending is kept at the recipient. The size of ASEND's table is maximized to a multiple of the number of process table entries in order to get an upper bound on the needed processing time. Finally, NOTIFY only requires a small bitmap in the recipient's process table entry to mark pending notifications.

4.2 Defenses (b): Message Delivery

a1. Addressing: Unauthorized access is prevented by using the IPC masks to restrict IPC destinations based on the principle of least authority. Ordinary applications, for example, are permitted to send to only the POSIX servers and nothing else. Each untrusted driver is associated with an isolation policy—a simple text-based configuration file that the administrator passes to the driver manager upon loading the driver—that lists its authorized IPC primitives and destinations. Upon loading a driver the driver manager maps the user-readable names in the policy to the low-level IPC endpoints (using the information kept in the name server) and informs the kernel about the driver's privileges. After loading the driver its label and IPC endpoint are also stored in the name server for future reference. The IPC restrictions are enforced at run-time using bitmap operations.

Spoofing is prevented because the IPC subsystem reliably puts the IPC endpoint of the sender in the IPC message. In addition, services can be identified using both compile-time and run-time information. The system servers that are running directly after booting the system, such as the POSIX servers, have fixed, globally known IPC endpoints. Endpoints of dynamically loaded extensions are published in a name server so that system servers can look up who is who. The name server exhibits publish-subscribe semantics, so that all dependent (registered) components are automatically notified about invalidated endpoints, for example, after a driver crash.

b2. Message contents: Problems due to oversized messages are structurally prevented, since our design uses only fixed-length messages. MINIX 3 also does not support self-paging so that the sender cannot be blocked due to unhandled page faults. Upon receiving, the receive buffer is validated (as discussed above) before copying the IPC message.

Likewise, the kernel verifies that the message indeed is located in the data segment of the sender. If not, the IPC subsystem immediately returns an error in order to prevent memory corruption.

However, since the kernel is only concerned with passing messages and is not aware of the message contents, the receiver must implement its own checks for unexpected input. The send masks are augmented with restriction mechanisms internal to the servers in order to filter out unauthorized service requests. For example, the driver manager informs the kernel task about the individual request types permitted by the driver's isolation policy, whereas the driver manager bases its access control on the caller's UID. Message arguments are checked using a defensive programming style. The fault-injection experiments presented in Sec. 5 prove that these additional checks are crucial to catch unexpected IPC.

4.3 Defenses (c): Group Interactions

c1. Flow control: Issues such as priority inversion and starvation cannot be countered by IPC infrastructure design, as discussed in Sec. 2.1, but are, in fact, message queuing and scheduling issues. MINIX 3 uses multilevel feedback queues (MLFQ) where processes on each of the priority queues are scheduled 'first in, first out' (FIFO). A process' priority is decreased whenever it consumes a full quantum, so that low-priority processes will also be scheduled in the end. Periodically, the priority of processes that do not run at their maximum priority are increased.

Denial of service is prevented, in part, by MINIX 3's MLFQ scheduler, which ensures all processes get scheduled. In addition, system servers such as the virtual file system (VFS) eliminate resource exhaustion by restricting clients to a single system call at a time. Repeated POSIX system calls are denied if a previous request is marked pending in the caller's process table entry at the VFS. This approach prevents resource exhaustion for calls where the VFS maintains state, such as *read* or *write* requests that are forwarded to the driver layer.

A related issue is an unresponsive driver that is not willing to receive IPC due to a bug rather than malicious behavior. This is dealt with by the driver manager, which replaces the driver with a fresh copy if it does not respond to periodic heartbeat requests [13].

c2. Caller blockage: Deadlocks are prevented, in the first place, by the design of a safe message passing protocol. In most cases, IPC send masks could be used to restrict the IPC destinations permitted and structurally prevent cyclic dependencies. However, a small number of interactions with the virtual file system could not be protected in this manner due to its central role in the system. Although the kernel's deadlock detection mechanisms will be triggered if

a faulty driver violates its protocol, such unexpected errors lead to complicated recovery strategies and should be prevented. Therefore, we augmented the safe message ordering with another deadlock-prevention strategy for the core system servers: in order to give hard structural guarantees we changed several interactions to asynchronous and non-blocking IPC, as discussed next.

Blockage due to asymmetric trust relationships (Sec. 2.2) is primarily prevented by using asynchronous and non-blocking IPC in the core system servers. In principle, untrusted clients (Fig. 4(b)) can be dealt with by restricting clients to SENDREC, which does a roundtrip IPC and ensures the client is blocked waiting for the reply. This solution is too restrictive for drivers, however, since they need other IPC primitives as well. Therefore, servers always use the nonblocking NSEND to reply and simply drop the reply if the other party is not ready. Interestingly, this forces clients to use SENDREC in order to prevent race conditions in receiving the response. Not doing so merely results in a waste of CPU time, but can never block the server.

Drivers acting as untrusted servers (Fig. 4(c,d)) are a threat to the virtual file system and network server. Therefore, they are programmed as a state machine and send driver requests asynchronously (using the buffered ASEND), save the state, and return to their main loop. Here, an ordinary synchronous RECEIVE from ANY is done to await the reply from the driver or a new request from another client. This ensures that the servers can never block either when sending the request or receiving the reply. Drivers are minimally affected, although they must respond using synchronous IPC or asynchronous IPC, but not nonblocking IPC, or the reply might be lost due to a race condition. In practice, drivers use the synchronous SEND to reply, since their isolation policies do not allow using ASEND.

As discussed in Sec. 2.2, however, not all asymmetric trust relationships need special care. For example, protection against caller blockage for the mutually dependent file servers and block driver in Fig. 6 does not improve the system’s overall dependability, since if the driver misbehaves, the file server becomes unusable in any case. Hence, we simply use SENDREC in the file server to talk to the driver. For interactions with the rest of the operating system, however, the group is treated as an untrusted entity.

5 FAULT-INJECTION TESTING

To evaluate our design we used software-implemented fault-injection (SWIFI), which show that the IPC architecture can effectively mitigate IPC threats posed by untrusted extensions. Our fault-injection test suite traces a running process and performs binary code mutations in its text segment at run-time. We have used an existing injector [21, 25] that works with address faults that change the source or des-

termination register, pointer faults that change address calculation, interface faults that ignore a parameter given, faults that invert the termination condition of a loop, text faults that flip a bit in an instruction, and nop faults that elide an instruction. These fault types were shown to be representative for common operating system errors [4, 24].

5.1 Experimental Setup and Workload

The experimental setup and workload were as follows. We injected faults into a Realtek RTL8139 Ethernet driver that was restricted by both the structural protection mechanisms and its isolation policy, which permits only synchronous IPC to a limited number of IPC destinations. The driver was continuously servicing TCP requests to a *day-time* server on a remote host in order to ensure that the injected faults were triggered—causing the driver to perform all kinds of unauthorized access attempts. In order to determine the effectiveness of our IPC defenses we instrumented the system with debugging output and inspected the system logs afterwards. We observed many different kinds of violations during the fault-injection tests, but focus on the IPC violations below.

Our experiments were conducted in an iterative process that led to a few design changes and various bug fixes in our prototype implementation. With respect to the IPC threats of class (a), two bugs in the IPC subsystem itself were detected (subclass (a1)). First, the driver used an invalid IPC endpoint, which went undetected and caused a panic. Second, a missing error condition allowed the driver using the nonblocking flag with the SENDREC call, which led to race condition if the callee was not yet receiving. We also experienced problems relating to class (b). Unauthorized IPC happened when the driver’s isolation policy was not properly defined (subclass (b1)). Furthermore, minor problems relating to missing error checks on message contents had to be fixed (subclass (b1)). Finally, two problems occurred in class (c). A scheduling problem occurred when the driver manager’s priority was set too low and heartbeats could not to detect a driver that winded up in an infinite loop (subclass (c1)). More importantly, a design change was needed to deal with asymmetric trust relationships (subclass (c2)). The virtual file system and network server were rewritten to use only nonblocking and asynchronous IPC and no longer can be blocked by a buggy driver [20]. The resulting system is able to withstand millions of fault injections.

5.2 Faults-Injection Results

The fault-injection test that we report on below consisted of 10,000 fault-injection trials that each injected 100 randomly selected faults, adding up to a total of 1,000,000 faults. Fatal errors, such as MMU exceptions and internal panics, caused 8,607 driver crashes and an equal number of

successful driver restarts, but the core system was never affected. Most errors did not crash the driver, however, so that it could repeatedly perform unauthorized operations. Our defenses against, for example, resource exhaustion and caller blockage, structurally prevented the problems relating to IPC threat classes (a2), (c1) and (c2), so these were never observed. However, inspection of the system logs provided overwhelming evidence for violations of classes (a1), (b1) and (b2) that rely on run-time protection.

Fig. 7 shows that the IPC subsystem itself denied 218,940 IPC calls. Most problems concerned invalid parameters (class (a1)), but the driver’s send mask also caught 1,260 unauthorized IPC destinations (class (b1)). Fig. 8 shows that the driver’s allowed IPC destinations rejected another 1,527,271 IPC calls due to unsupported and unauthorized requests (class (b2)). Valid requests with bad arguments were excluded from this number, since they are not directly related to IPC and require a defensive programming style in any system. The large number of bad requests rejected by the kernel’s system task (1,524,516) is because it is called all the time to perform privileged operations such as I/O and safe copies. The kernel call protection thus proved extremely useful.

All in all, the fault-injection tests strengthened our trust in the IPC defense mechanisms employed in MINIX 3 and show that they can be used as a foundation to build a dependable system. Out of the 1,000,000 randomly injected faults for this experiment, no fault was able to harm the IPC subsystem itself or misuse the IPC services to disrupt other parts of the operating system.

IPC Call Status	IPCs Failed
IPC call failed due to bad primitive	915
IPC call failed due to bad IPC endpoint	14,542
IPC call failed due to bad message buffer	202,223
IPC primitive disallowed by IPC mask	0
IPC destination disallowed by IPC mask	1,260
Total failed in the IPC subsystem	218,940

Figure 7: Bad IPC calls caught and denied by the IPC subsystem (classes (a1) and (b1)).

Allowed IPC Destination	Bad Requests
Kernel task (privileged operations)	1,524,516
Terminal driver (debug output)	1,388
Log driver (debug output)	604
POSIX servers (POSIX system calls)	352
TCP/IP server (network requests)	253
PCI bus driver (device initialization)	90
Driver manager (heartbeat pings)	61
Name server (lookup endpoints)	7
Total bad requests detected by recipient	1,527,271

Figure 8: Allowed IPC destinations and bad IPC request types detected by recipient (class (b2)).

6 RELATED WORK

We are not aware of work that provides a comprehensive coverage of IPC threats and countermeasures in multiserver operating systems as we have done. While individual problems have been addressed [6, 15, 19, 22, 23], a systematic approach towards dependability is lacking. Nevertheless, we discuss four different IPC designs below.

1. *Interposition and Monitoring:* Violating the trusted addressing principle discussed in Sec. 1.2, IPC interposition allows intercepting IPC traffic in a reference monitor in order to decide whether the IPC was permitted. Initially ‘clans and chiefs’ [19] were used, but this model fixed the access control policy in the microkernel and caused too much performance overhead. IPC redirection allows setting kernel redirections to provide efficient and flexible access control. Transparent monitors [15] extended this model in order to keep IPC over monitors fully synchronous. However, IPC interposition is mainly concerned with access control and cannot be used to address all of the IPC threats we identified. For example, untrusted servers (Fig. 4(d)) are still a threat, since IPC is not delivered to the first monitor until the destination becomes ready to receive it [15].

2. *Multithreading and Timeouts:* The L4 microkernel is used in several multiserver systems, such as NIZZA [11]. The latest IPC design, part of L4.Sec [16], manages access control by mapping send and receive capabilities into the address space of other threads. L4.Sec has a single, synchronous IPC primitive, `SHORT_IPC`, with a send and receive phase. Each phase is optional and IPC roundtrips cannot be enforced. The caller remains blocked until either the timeout expires or the IPC call finishes. The message payload length is dynamic. Since L4 supports self-paging, a second timeout can be set to prevent caller blockage due to page faults. L4.Sec has multithreading-aware IPC and provides a control operation that allows one thread unblocking another thread blocked on IPC.

3. *IPC Roundtrips and Truncation:* EROS is a capability-based OS that provides transparent persistence. The system has a multiserver design with a limited asymmetric trust model where drivers are part of the kernel and thus trusted. EROS’ capability-based protection is integrated with the IPC system to perform operations on objects. Access control is capability-based. An untrusted client that has a start capability to a server can invoke `CALL` to start a synchronous IPC roundtrip. The server can respond by invoking `RETURN`, which sends the reply and blocks until the next request arrives. There is no separate receive operation. A nonblocking `SEND` exists to support event notifications. Dynamic payloads are supported, but the message is truncated if the IPC causes a page fault.

4. *Buffering and Language Support:* Recently, it was shown how Singularity uses the `Sing#` programming lan-

guage to realize fast and reliable channel-based IPC [6]. Static verification of IPC is supported using state machine-like channel contracts defined by the programmer. Each contract specifies the message types a channel can carry as well as which state transitions are allowed. Sending is done asynchronously with buffer management done by the run-time system. Efficient data exchange is done by passing a pointer into the ‘exchange heap’ and transferring ownership. Receive operations are blocking and can be wrapped in a switch receive statement to specify which messages are desired. The IPC fails if the run-time system detects that the IPC is not allowed in the process’ current state.

7 CONCLUSIONS

In this paper, we presented a comprehensive overview of IPC threats in multiserver systems and showed how they can be countered. The design of a dependable IPC architecture is a fundamental requirement for dependability, because components such as third-party drivers and other extensions cannot be trusted to respect the message passing protocol. Therefore, we presented a classification of IPC threats as well as an extended asymmetric trust model covering two new vulnerabilities in synchronous IPC.

We presented the IPC architecture that is used by MINIX 3 in order to systematically protect the operating system against each of the IPC threats. We explained the rationale behind our design, detailed the IPC infrastructure that we implemented, and discussed how and why this infrastructure can be used safely. Fault-injection experiments on a prototype implementation suggests that the proposed IPC defenses indeed can effectively mitigate the IPC threats posed by untrusted extensions and safeguard the dependability of multiserver operating systems.

8 ACKNOWLEDGMENTS

Supported by Netherlands Organization for Scientific Research (NWO) under grant 612-060-420.

REFERENCES

- [1] V. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *CACM*, 21(1):42–52, Jan. 1984.
- [2] J. Cache and D. Maynor. Device Drivers. In *Presented at Black Hat USA’06*, Las Vegas, Nev, USA, July 2006.
- [3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proc. 18th Symp. on Oper. Syst. Prin.*, pages 73–88, 2001.
- [4] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults–Based on Field Data. In *Proc. 26th Int’l Symp. on Fault Tolerant Computing*, 1996.
- [5] T. Dinh-Trong and J. M. Bieman. Open Source Software Development: A Case Study of FreeBSD. In *Proc. 10th Int’l Symp. on Software Metrics*, pages 96–105, 2004.
- [6] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proc. 1st EuroSys Conf.*, pages 177–190, Apr. 2006.
- [7] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. 9th ACM SIGOPS European Workshop*, pages 109–114, Sept. 2000.
- [8] W. M. Gentleman. Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software - Practice & Experience*, 11(5):435–466, 1981.
- [9] A. Haebleren, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. Stub-code performance is becoming important. In *Proc. 1st Workshop on Industrial Experiences with Systems Software*, Oct. 2000.
- [10] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ -Kernel-Based Systems. In *Proc. 6th Symp. on Oper. Syst. Prin.*, pages 66–77, Oct. 1997.
- [11] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proc. 1st Int’l Conf. on Collaborative Computing*, Dec. 2005.
- [12] L. Hatton. Reexamining the Fault Density-Component Size Connection. *IEEE Software*, 14(2):89–97, 1997.
- [13] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure Resilience for Device Drivers. In *Proc. 37th Conf. on Dependable Systems and Networks*, pages 41–50, June 2007.
- [14] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber. Sealing OS Processes to Improve Dependability and Safety. In *Proc. 2nd EuroSys Conf.*, pages 341–354, Mar. 2007.
- [15] T. Jaeger, J. Tidswell, A. Gefflaut, Y. Park, J. Liedtke, and K. Elphinstone. Synchronous IPC over Transparent Monitors. In *Proc. 9th ACM SIGOPS European Workshop*, Sept. 2000.
- [16] B. Kauer and M. Völp. L4.Sec Preliminary Microkernel Reference Manual, Oct. 2005.
- [17] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. 6th OSDI*, pages 17–30, Dec. 2004.
- [18] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC Performance. In *Proc. 6th HotOS Workshop*, pages 28–31, May 1997.
- [19] J. Liedtke, N. Islam, and T. Jaeger. Preventing Denial-of-Service Attacks on a μ -Kernel for WebOSes. In *Proc. 6th HotOS Workshop*, May 1997.
- [20] Matej Kosik. How to Completely Crash Minix. Post on the MINIX newsgroup: comp.os.minix, Sep. 2007.
- [21] W. T. Ng and P. M. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proc. 29th Int’l Symp. on Fault-Tolerant Computing*, pages 76–83, 1999.
- [22] J. S. Shapiro. Vulnerabilities in Synchronous IPC Designs. In *Proc. IEEE Symp. on Security and Privacy*, pages 251–262, May 2003.
- [23] U. Steinberg, J. Wolter, and H. Härtig. Fast Component Interaction for Real-Time Systems. In *Proc. 17th Euromicro Conf. on Real-Time Systems*, pages 89–97, 2005.
- [24] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability – A Study of Field Failures in Operating Systems. In *Proc. 21st Int’l Symp. on Fault-Tolerant Computing*, 1991.
- [25] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. *ACM TOCS.*, 23(1):77–110, 2005.
- [26] T.J. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *Proc. Int’l Symp. on Software Testing and Analysis*, pages 55–64, July 2002.
- [27] K. Wong. Mac OS X on L4, Dec. 2003. Bachelor’s Thesis. University of New South Wales, Sydney, Australia.
- [28] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proc. 6th Pacific Rim Int’l Symp. on Dependable Computing*, pages 178–185, 1999.