

Research on Really Reliable and Secure System Software (R3S3)

Introduction

Current operating systems have poor reliability and security. Computers crash regularly whereas other electronic devices such as televisions and mobile phones never crash. Furthermore, practically every week one reads in the newspaper about another security hole in Windows. No sooner has Microsoft posted a patch to its website (which only a fraction of Windows users download and install) than another hole turns up, sometimes in the patch itself. Attacks by viruses and worms are rampant. It is no understatement to say that Windows has very serious reliability and security issues. UNIX derivatives such as Linux are not in the press as much. In part it is because they are slightly more secure, but in reality because hackers tend to focus on hitting Windows, where they can get the most publicity for their attacks. As computers become more and more essential for all aspects of society, many people regard this situation as unacceptable. The goal of my proposed research is to conceive, design, implement, and test an operating system that is as reliable and secure as is humanly possible. The job will be finished when the average user has never experienced a crash in his lifetime and RESET buttons on computers have passed into history, like 5¼ -inch floppy disks.

Reliability is not the only issue. The situation with security is much worse. Reliability problems manifest themselves as loss of service for minutes or perhaps an hour, with the associated annoyance and for companies, loss of business and damage to their reputation. Security problems are much worse. Security incidents can result in confidential information leaking out to hostile parties or attackers tampering with or modifying personal, company, or government data. Also a problem is attackers exploiting bugs to take over a computer and use it as part of a zombie army to send spam or mount denial of service attacks elsewhere.

Security incidents can be expensive. Some estimates put the cost of dealing with the Code Red virus at upwards of \$2 billion. The ILOVEYOU worm infected 10% of the computers on the Internet. The Sasser worm caused dozens of Delta Airlines flights to be cancelled or delayed, and so on. Probably no one knows what the real costs of unreliable and insecure software are, but they are substantial. There is really a problem here.

I will get into the details of the problem and my research for solving it shortly, but I should start out by pointing out how ambitious and risky this research is. In effect, I am going to make the case that the hundreds of millions of computers in the world are all based on fundamentally flawed system software that cannot be repaired. Future computers need something completely different. This requires a new and different structure of the software. Figuring out how to do this is definitely frontier research.

Why Are Computer Systems Unreliable and Insecure?

Most reliability and security problems ultimately come down to the fact that programmers are human beings and as such are not perfect. While no one disputes this statement, no current systems are designed to deal with the consequences of it. Studies have shown the number of bugs in commercially available software ranges from about 1 bug per 1000 lines of code to 20 bugs per 1000 lines of code (no references are provided in this synopsis but they are present in the full proposal). Large and complex software systems have more bugs per 1000 lines of code due to the larger number of modules and their more complex interactions. Windows XP consists of 50 million lines of code. From this it follows that Windows XP has somewhere between 50,000 and 1 million bugs in it. And Vista is larger still (70 million lines) and probably has even more bugs. The key insight here is: *Expect bugs*. They are there. Learn to live with them.

Now consider hardware for a moment. It is not perfect either. Is it possible to build a dependable system out of memories that fail, disks that crash, and communication lines that are flaky? Yes! Hardware designs exist that are able to tolerate failures. RAIDs continue to function properly in the presence of intermittent or even fatal disk-drive failures. ECC memories continue to work correctly even when bits in RAM get stuck at 0 or 1. CD-ROMs and DVDs use error-correcting codes to deliver error-free data even from dirty media or when bits are damaged on the medium. TCP can handle lost, misordered, or garbled packets without the users noticing them. In other words, the idea that a system can detect its own errors and repair them automatically is well established. This property is called *fault tolerance*. Most software is *fault-intolerant*: if a bug in the software is encountered, the software crashes. What we want to do is make operating systems fault tolerant, also called *self healing* or *self repairing*. A fully self-healing operating system has never been done before. No one knows how.

Of course, software errors are not the same as hardware errors. At first glance it might seem impossible to recover transparently from failed hardware, but careful design shows that it is possible through redundancy, either in space (e.g., spare drives in a RAID) or in time (e.g., repeated transmission of lost packets in TCP).

While algorithmic errors cannot be masked by running the program again, in operating systems (which are highly concurrent) most errors are triggered by race conditions, unlikely timing sequences, or deadlocks. They are frequently transient and do not appear when the code is executed again. Thus most software failures are much more like hardware failures (such as packets lost on the wire) than one might at first imagine.

The relation between reliability problems and security problems is subtle but important. In many cases, both stem from the same cause: bugs in the code. Sometimes a bug manifests itself in a crash or other incorrect behavior. Sometimes it allows a hacker to break into the system and subvert it. For our purposes, we treat both reliability issues and security issues as both being consequences of the same underlying phenomenon: buggy code. In this proposal I will refer to reliability frequently and security somewhat less so. The thrust of the proposed research on security is to reduce the security damage bugs can do by breaking the system into small components with solid walls around them to limit the damage a compromised component can do.

A lot of security research revolves around making good models of who can access which resource and possibly proving the correctness of the model. The work on SELinux is a clear example (Loscocco and Smalley, 2001). While this work is definitely worth undertaking, nearly all experience with actual security incidents shows that security problems almost always stem from actions that the design and rules forbid but which bugs in the code allow to happen anyway. Banks lock their front doors at night and have strong safes even though there are laws forbidding bank robbery. Having the right rules is fine, but making sure it is they are actually enforced is far better. Thus our approach is making sure it is technically difficult for bugs to lead to security failures rather than working on rules about who can do what. Rules that cannot be enforced are not worth much.

The most serious reliability and security problems are those relating to the operating system. The core problem is that no current system obeys the *POLA: the Principle Of Least Authority*. The POLA states that a system should be partitioned into components in such a way that an inevitable bug in one component cannot propagate into another component and do damage there. Each component should be given only the authority it needs to do its own job and no more. In particular, it should not be able to read or write data belonging to another component, read any part of the computer's memory other than its own address space, execute sensitive instructions it has no business executing, touch I/O devices it should not touch, and so on. Current operating systems violate this principle completely, resulting in the reliability and security problems mentioned above.

To make clear how this principle is violated in current systems, consider an example. Every I/O device needs a device driver to control it. The device driver is written by a programmer working for the device manufacturer and included in the device's box on a CD-ROM. When the device is installed, either by the PC vendor or the user, the device driver is loaded onto the hard disk. When the computer is booted, the operating system looks for the device drivers it needs and loads them into the operating system kernel, the part of the system that runs in the hardware's protected *kernel mode*, which has unrestricted access to all memory, instructions, and devices.

Now realize what this means. Various pieces of code (possibly dozens of device drivers) written by dozens of programmers who do not know each other and of varying skill levels are now present *inside* the operating system. Bugs in any of them can overwrite any system table, any disk file, and any user program. This situation is the electronic equivalent of accepting a package from a total stranger at an airport and carrying it onto your airplane. It is no wonder that airlines try to discourage this behavior.

While it might seem reasonable to run unknown, untrusted software in user mode where it can do less damage, this is not how current systems work. Worse yet, about 70% of a typical operating system consists of device drivers, and these are known to have 3-7 times as many bugs per line of code as the rest of the system. It is also well documented that 63% to 85% of Windows XP crashes are due to driver failures and there is no reason to expect that Linux is any different.

Microkernels and (Para)virtualization

Over the past few years my research group has been investigating tiny microkernels as a basis for reliable and secure operating systems. There are two main thrusts to our research. First, I want to reorganize the operating system as a tiny microkernel that runs in kernel mode, along with some number of user processes that do the real work of the operating system. While a microkernel is by no means a complete operating system, this design achieves a major goal. The goal is getting most of the operating system code into user space where it can be partitioned into processes unable to execute "dangerous" (i.e., control) instructions and prohibited from accessing memory outside itself by the memory management hardware.

Our research has led a system, MINIX 3, based on a microkernel that is the only software running in kernel mode. The microkernel is only about 5000 lines of code, less than 0.1% of the size of the Windows kernel and less than 0.2% of the size of the Linux kernel. Statistically, this means there are somewhere between 5 and 100

bugs in the kernel, which might eventually be found,. Also, 5000 lines of code (about 100 pages of printout) is something a single programmer can read and fully understand in a week. In contrast, no one understands all of Windows or all of the 3 million lines of Linux. The system structure is illustrated in Fig. 1.

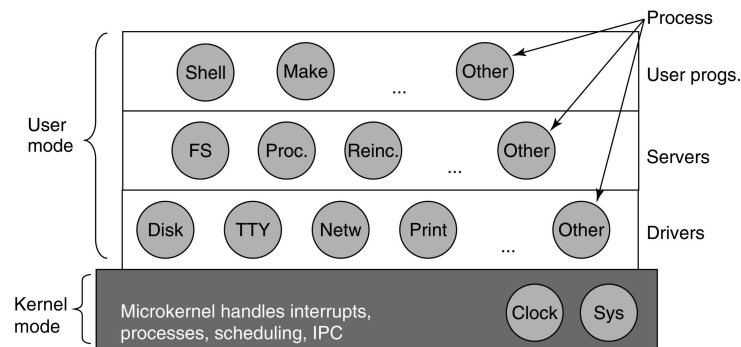


Fig. 1. Structure of the MINIX 3 operating system.

Above the kernel, running in user mode, are the device drivers, each one running as a separate process tightly restricted by the memory management hardware to accessing only its own memory. Device drivers need to do I/O and they do this by making calls to the microkernel to obtain services (such as issuing commands to a physical device). However, before executing any call, the microkernel first checks to make sure the call is permitted. Thus a call from the audio driver to command the sound card will be accepted and processed but a call from the audio driver to command the disk will be rejected.

One of the novel servers running here is the *reincarnation server* (Reinc.) whose job is to monitor the other system components to see if they are working correctly and take action if they are not. The action is typically to report the incident and restart the failed component. Since most failures are due to timing and similar errors, usually a fresh restart brings the system back to a fully working state. This restart does *not* affect the user programs running at the time. They do not even notice. It is entirely internal to the operating system. At present, restarting components can only be done when the component has no internal state information. If the component has internal state information, when it is replaced by a fresh copy (from disk) that state is lost and the driver may fail. This is one of the points that will be addressed in the proposed research.

An alternative way to structure an operating system is to run a *hypervisor* in kernel mode and use it to emulate one or more *virtual machines* running in user mode. Each of these virtual machines can run its own operating system. In some cases, the operating systems running on the user machines have been modified to make the emulation easier. This is called *paravirtualization*. If the operating system in the virtual machine has been very heavily paravirtualized (as is commonly the case), then the hypervisor running in kernel mode needs an extensive API (Application Programming Interface) to handle the calls from the virtual machines to get services from the hypervisor. At that point, the difference between a hypervisor and a microkernel begins to get vague. I believe hypervisors and microkernels will begin to merge although no one knows how to do it right.

The Proposed Research

What I am proposing is a fundamental redesign of the operating system. This is frontier research. Since the beginning of computing 50 years ago, all serious operating systems (a few academic toys aside) have been unstructured monolithic lumps of code running in kernel mode. From MULTICS to MS-DOS, through all versions of Windows, and UNIX (including, Linux, Solaris, FreeBSD) this has been the model. I am proposing replacing the entire monolithic kernel paradigm with a new one consisting of small, tightly constrained modules, running in user mode and each strictly obeying the POLA. Nothing like this has ever been achieved before. I also have many additional goals, such as being able to restart or even replace faulty components on the fly during execution and assign legal liability when faults occur, as described in the full proposal. While my previous work has taken a few baby steps in this direction, there is much more research needed to make this approach viable.

Fortunately, this can be done without affecting the user software by simply having the new operating system emulate the existing and long-stable POSIX interface (the UNIX standard), possibly extended somewhat (e.g., with some of the Linux system calls). As long as the usual POSIX system calls such as OPEN, READ, EXEC, and so on are available, most UNIX software can be made to work without too much effort.

Core Research Areas

To achieve our goal of a modular user-mode operating system, a number of problems must be solved. The list below covers some of the research goals, but by no means all of them. No doubt many others will arise.

Kernel functionality. The kernel provides services to the operating system components and is unrelated to the POSIX (i.e. UNIX) interface the operating system provides to user programs. For example, when a file server needs to copy data to the address space of the calling user process it asks the kernel to do the job. A mechanism should be present to make sure that such copy can only be done with the *explicit permission* of the user process. What is needed to do is make sure *every* kernel call obeys the POLA and gives its caller precisely the power it needs and nothing more. Finding mechanisms to do this for memory management, I/O, and everything else will be part of the research. If this succeeds, it will be much harder for a virus that has infected a system component to inflict heavy damage on other components. This is but one example where strict adherence to the POLA greatly enhances system security.

Components. If most of the operating system is running as user-mode processes, an obvious question is how many components should there be and what should they do? Furthermore, what is the nature of these components? Should they be something like virtual machines, like processes, or something else? In fact, once it is clear that that true virtual machines are not the answer, it is not obvious what the difference is. In a system like L⁴Linux, where a virtual machine runs a complete (highly paravirtualized) operating system complete with its own kernel and user processes, it is fairly clear what a component is, but I am trying to split up the operating system into pieces, so this model does not really apply. What model does apply then? Stepping back from where we are now and knowing what we now know, how should we look at the components?

Communication. When two components are communicating, it is essential that a fault in one of them does not hang the other one due to failures in the communication protocol. For example, if a component acting as a client sends a request to another acting as a server, the communication protocol may hang the client if the server does not respond. This is well studied. But if the client is not prepared to accept the response as it should, it can hang the server, something far less well studied. I want to reexamine the whole issue of communication in the light of fault tolerance. Also, I want to examine which component can talk to which component (including transitive closure) to limit damage due to viruses and other malware.

Stateful components. All current work on recovery has assumed stateless components, that is, components that can be replaced with a fresh copy from disk without users noticing. Some drivers have such a property (for example, disk drivers and network drivers), but others do not. One of the key areas I want to look at in the new research is recovering stateful components. As a simple example, consider an audio driver that has one integer as its state: the volume at which the music is to be played. If the audio driver crashes and a fresh one is loaded automatically, the volume will be reset to the default level and the user will notice this sudden change in volume. I want to be able to recover components with complex state transparently.

An advanced, recoverable, fault-tolerant file system. One of the most critical areas I want to investigate is the file system. Current file systems, especially those used on various UNIX derivatives, are fundamentally offspring of the MULTICS file system, designed in the early 1960s. I want to design a file system for the 21st century, looking at both its features and interface to application programs and its internal structure. Issues include longevity (how to interpret a WordStar file 50 years later), provenance (where did a file come from, who owns it and what may I do with it), and long-term security. The file system should never go down and continue running even in the face of internal faults. This means making the file system itself modular and designing it in such a way that after a crash a new version can be loaded and pick up the pieces and continue running, without affecting user programs currently running. While people have built fault-tolerant hardware (e.g., RAIDs), system software that can tolerate faults and keep going is groundbreaking. While other research is focused on eliminating bugs, ours makes the assumption that programmers are not perfect and bugs will be around for a while, so we have to design systems that do not fail, even in the presence of bugs. If hardware can be made to tolerate faults why not software?

Live update. The idea of fixing a broken component by just starting a fresh copy often solves the problem temporarily (because so many errors are due to race conditions), but does not remove the underlying bug, which may appear again later. A real solution requires replacing the code of the buggy component with a different one with at least some of the bugs fixed. The idea I am thinking about would be to first install a newer (and less buggy) version of the component on the disk. Then a message or signal would be sent to the component, asking it to save all its state somewhere safe. After doing this and getting back an acknowledgement, the component to be replaced would then exit and a fresh one started. The tricky part of this is managing the state. If the new version has a different internal state than the old one, the new one will somehow have to read the saved state, understand it, and convert it to the new format. How this could be done in a general way is part of the research.

Enforcing least authority. Normally, the entire operating system runs in kernel mode because pieces of it need special powers not generally granted to user processes. For example, a device driver needs a way to access its I/O device and the file system needs a way to copy data to and from user processes. At present we have a series

of ad hoc powers with various bit maps controlling who can do what. It would be better if a unified scheme for delegating powers and formalizing this in policies could be found. In this way a device driver could publish what powers it needs to do its work in some kind of specification and these could be checked in advance of starting the driver for compatibility with the systems requirements.

Controlling DMA. At present, many I/O cards (e.g., Ethernet cards) can do DMA to any address in physical memory. This clearly circumvents the protection mechanisms present in the software and creates a gigantic reliability and security hole. Fortunately, some new PCs just starting to be shipped come with an I/O MMU that makes it possible for the operating system to set up restrictions that DMA engines cannot bypass. This is an area I want to look at carefully as it plugs one of the last remaining holes in any proposed protection scheme. Since I/O MMUs are brand new, little is known about using them to enforce the POLA on I/O software.

Recovering the reincarnation server. The reincarnation server is a crucial component in the recovery process. It constantly monitors the health of the system and restarts components when they fail. If it itself dies, there is no way to recover it. I would like to make it recoverable.

Other Research Areas

Legal accountability. If it is possible to exactly pinpoint which component failed, it will become possible to assign blame for a failure and potentially to assess damages. In other words, if it can be unambiguously proven (in court) that a particular component failed (for example, by inspecting a digitally signed core dump), the maker of that component may be liable, which may provide an incentive to try to debug the software before shipping it. This would also require being able to collect and save forensic evidence.

Use of a type-safe Language. A different approach to reliability and security is to use a type-safe language, as is done in Singularity. While this approach has major problems (such as lack of compatibility with everything ever done in the past), it also has promise. Fortunately, the modular design envisioned allows us to experiment gradually by rethinking and reprogramming the components one-by-one in a type-safe language to see how well it works. I regard Cyclone as a possible candidate language since it is type safe, yet resembles C in other ways. However, in principle, the modular structure allows each component to be written in a different language.

Multicore chips. Another recent development is the advent of dual-core, quad-core, and in general multicore chips (and also hyperthreaded chips). No one has a good idea of what the consequences of multicore systems are for modular operating systems running in user mode. Should different components be (permanently) assigned to different cores? Should they be dynamically assigned? What are the consequences for issues such as concurrency control, locking, and performance? These issues are certainly worth investigating carefully.

Methodology and Resources.

Three researchers are being requested as well as two programmers. One of the researchers will focus on the structure of the system, which includes the kernel function and API, componentization, and communication issues. Another will look at managing, guarding, and recovering components with internal state, something rarely addressed so far. The third will look at some of the other issues. Postdocs can get started faster than Ph.D. students, but universities also need to train new people, so a student is included. Who will do which part depends on the interests of the people. For the most part, the research questions to be addressed are sufficiently isolated that they can be examined in parallel. None of the subprojects are on the critical path of the other ones. In general, computer science does not have a methodology like physics with a series of experiments. In that respect, it is more like mathematics. It is very difficult to make a detailed multiyear plan, but I have 30 years experience in systems research and have run many projects before.

A very important part of the research is building a realistic working prototype to show that the research ideas developed work in practice. This is why two programmers are requested. The prototype will be subjected to heavy fault-injection testing to see how it stands up. This proof-of-concept prototype has to run a substantial amount of real software to be convincing. I will use UNIX (including Linux and FreeBSD) applications since all the source code is available. All software produced will be open source. I already have enough ideas so the programmers can start on day 1.

If this work succeeds, it will open up new areas to explore in terms of componentized, multiserver operating systems. People will then examine other kinds of new software components and how they fit in.

The project budget is just under 2.5 M€ and described in the full proposal.