GERNOT HEISER

secure embedded systems need microkernels

Gernot Heiser is professor of operating systems at the University of New South Wales and leader of the research program in embedded, real-time, and operating systems at National ICT Australia (NICTA). His research interests include microkernels and microkernel-based systems, operating systems for embedded systems, and OS-level power management, as well as general performance and scalability issues in operating systems.

gernot@nicta.com.au

THE IMMENSE POPULARITY OF ALL sorts of electronic devices means that they have become an integral part of our lives; it is becoming difficult to imagine living without them. In the process, they are increasingly trusted with sensitive data, the loss of which can cause serious distress or financial harm. Security is therefore becoming a significant issue. Yet, as we know from the PC world, commodity computer systems are not well-defended against security threats. In this article we examine the security threats facing embedded systems, and what needs to be done to make them secure.

Embedded Systems Security Threats

Embedded systems—computers which are part of a larger system that is not primarily a computing device—are commonplace; in industrialized countries they outnumber people by about an order of magnitude. This includes cell phones, PDAs, entertainment devices, cars, washing machines, smart cards, broadband modems, and many more.

With our increasing dependence on embedded systems, their reliability and security become more and more of an issue. For example, cell phones and PDAs are used to perform financial transactions, which means that they are trusted with account access codes. Embedded devices also store increasing amounts of sensitive personal data, from address books to medical data. Hence, the security of such systems is a serious concern.

The main reason that embedded systems are becoming increasingly vulnerable is the pervasive use of wireless communication. Besides the already ubiquitous mobile phones, PDAs, and laptops, there is a set of devices, now quite common, whose primary purpose is not communication but which benefit from wireless communication. These include vehicles, access tokens, domestic appliances, and medical devices, among others.

In the world of wireless connectivity, physical access is no longer required in order to compromise a device, and the environment in which such devices operate is increasingly hostile. Devices can be attacked by an invisible foe behind an opaque wall. If the device is connected to the Internet, the attacker can be located anywhere in the world. Furthermore, users who download executable code on their mobile devices open up these devices to attacks from within (by viruses and worms).

Previously, compromised equipment would most likely result in inconvenience and annoyance. Now that devices hold increasing amounts of sensitive personal data, the consequences of security breaches are much more serious.

Moreover, the whole wireless communication infrastructure is potentially vulnerable. Until recently, low-level communication operations were all done by hardware, which is secure from subversion except by the application of physical force. But now even the lowest-level functionality has moved into software (software-defined radio), making it vulnerable to attacks that change the software.

For example, a compromised mobile phone handset could be turned into a jammer, disabling all communication of a particular carrier within a radius of potentially several kilometers. If a large number of compromised handsets launched a concerted attack, a country's wireless communication infrastructure could be disabled within minutes—a disaster which would be very difficult to recover from. Such an attack is not out of the question. The Kabir cell-phone virus, which spreads via Bluetooth, is estimated to have infected millions of phones already.

Protecting Embedded-Systems Security

The key to preventing such disasters is to equip mobile devices with software that is *secure by design*. As the experience of the PC world shows, this is not easy—clearly, the standard of mobile device security must be much higher than what we are used to from the PC world. The problem is that the software driving mobile devices is becoming as complex as that of PCs, owing to the dramatic increase in functionality of such devices. Top-of-the-line cell phones already run software that is composed of millions of lines of code (LOC), and top-of-the-line cars contain in excess of a gigabyte of software.

Such large systems are impossible to make fault-free. Experience shows that even well-engineered software averages at least one fault every few thousand lines of code (and well-engineered software is rare). This is made worse by the traditional approach to embedded-systems software, which tends to be built on top of a real-time executive without memory protection. In such a system every bug in any part of the system can cause a security violation.

In security terms, the part of a system that can circumvent security policies (and must therefore be fully trusted) is called the *trusted computing base* (TCB). In a system without memory protection, the TCB is the complete system (of potentially millions of lines of code). Clearly, such a large TCB cannot be made *trustworthy*.

Trustworthy TCB?

Over the past few years the embedded-systems industry has been moving toward the use of memory protection, and operating systems which support it. With this comes the increasing popularity of commodity operating systems, particularly embedded versions of Linux and Windows. Those systems, if stripped to a bare minimum for embedded-systems use, may have a kernel (defined as the code executing in the hardware's privileged mode) of maybe 200,000 LOC, which is a lower bound on the size of the TCB. In practice, the TCB is larger than just the kernel; for example, in a Linux system every root daemon is part of the TCB. Hence the TCB will, at an optimistic estimate, still contain hundreds if not thousands of bugs, far too many for comfort.

If we want a secure system, we need a secure, trustworthy TCB, which really means one free of bugs. Is this possible?

Methods for guaranteeing the correctness of code (exhaustive testing and mathematical proof, a.k.a. formal methods) scale very poorly; they are typically limited to hundreds or, at best, thousands of lines of code. Can the TCB be made so small?

Maybe not, but maybe it doesn't have to be. Modularity is a proven way of dealing with complexity, as it allows one to separate the problem into more tractable segments. However, with respect to trustworthiness, modularizing the kernel does not help, as there is no protection against kernel code violating module boundaries. As far as assertion goes, the kernel is atomic.

The situation is better for non-kernel code. If this is modularized, then individual modules (or *components*) can be encapsulated into their own address spaces, which means that the module boundaries are enforced by hardware mechanisms mediated by the kernel. If the kernel is trustworthy, then the trustworthiness of such a component can be established independently from other components. That way, the TCB can be made trustworthy even if it is larger than what is tractable by exhaustive testing or formal methods.

Minimal Kernel

The key to a trustworthy TCB is therefore a very small kernel, small enough to be verified. A minimal kernel will *only contain code that must be privileged;* any functionality that can be performed by unprivileged code should remain unprivileged (i.e., outside the kernel). Such a kernel is called a *microkernel*. It contains little more than the fabric required to enforce the interfaces between components: protection (in the form of address spaces) plus a mechanism, called *interprocess communication* (IPC), for controlled communication across address space.

A true microkernel in this strict sense has not been built to date. However, there are good approximations, specifically the L4 microkernel. Its most mature and most widely used implementation, L4Ka::Pistachio, developed at the University of Karlsruhe, consists of about 10,000 lines of code (counting only code required to build it on a particular architecture, e.g., ARM). Ten thousand LOC is still large for a system that is aimed to be completely bug-free, but the goal is within reach. In fact, at NICTA we have two projects underway that aim to achieve exactly this (and similar activities are under way at Dresden University of Technology).

The project called *seL4* seeks to produce a new version of L4 that is a better approximation of a microkernel and, at the same time, an API that is better matched to the requirements of secure systems. We expect the seL4 kernel to consist of only 5000–7000 LOC.

The second project, called *L4.verified,* aims at a mathematical proof of the correctness of the seL4 kernel. Specifically, the project aims to prove that the kernel's implementation is consistent with its specification (i.e., a formal model of its ABI). The formal model of the kernel can then be used to prove security properties of systems built on top of the kernel.

While it will take a few years to achieve this goal of a formal correctness proof, the small size of the existing kernel already provides an excellent base for building a more trustworthy TCB. Although testing and code inspection cannot give complete assurance of the kernel's correctness, the small size makes it possible to reduce the number of defects to maybe a few dozen. Debugging is aided by the fact that the kernel provides only a very small number of fundamental mechanisms. This means that any non-trivial system built on top exercises almost the complete kernel functionality—bugs do not have many places to hide.

Minimal TCB

The L4 kernel supports the construction of a small TCB. We have developed a minimal operating system, called *Iguana,* specifically for use in embedded systems. Iguana provides essential services, such as memory management, naming, and support for device drivers—enough for many embedded applications. The complete resident[1] TCB of such a system, consisting of L4, Iguana, and a few drivers, can be as small as about 20,000 LOC. We expect that a minimal TCB of seL4-based systems will be 10,000–15,000 LOC.

It should be noted that the TCB (and its size) depends a lot on what functionality a system is to provide. Specifically, systems with non-trivial user interfaces (such as graphics displays and pointer devices) tend to have larger TCBs, which may include a trustworthy window system that guarantees that the user's input is consumed by the right program. The sizes quoted in the preceding paragraph are for a system with minimal requirements.

L4/Iguana is mature and has excellent performance, good enough to be deployed in commercial products; it will ship in a major consumer item early next year. Its users will have an upgrade path to a provably correct seL4-based system, once the L4.verified project succeeds.

Fine-Grained Access Control

Besides the large size of the TCB, there is at least one other reason why traditional operating systems such as Linux and Windows are a poor match for the requirements of embedded systems. They have a model of access control that originated in time-shared mainframes: different users of the system must be protected from each other, while there is no reason to restrict a particular user's access to their own data.

Embedded systems, on the other hand, are typically single-user systems, and the protection issue is quite different: different programs run by the same user should have different access rights, determined by their function rather than the identity of the user. This is an instance of the security principle of *least privilege.* Traditional systems violate least privilege, by running every program with the full set of access rights (to files and other objects) of the user. This is one of the reasons why viruses and worms can cause so much damage: A game program should only have access to the I/O devices needed to play it, its own executable, a file to save its state, and (for networked games) a well-defined communication channel. Having full access permits a virus embedded in the game program to destroy the user's files or steal their contents.

In a microkernel-based system, where software is encapsulated into components with hardware-enforced interfaces, all communication must employ the kernel-provided IPC mechanism. This means that the kernel is in full control over all communication between components. It also means that it is possible to transparently interpose security monitors between components, which can be used to enforce system-wide security policies. Such a policy could be that a program imported into the system (such as a game) is only allowed to access files that have been explicitly assigned to it by the user, thus preventing the theft of sensitive information.

Virtual Machines

Microkernels have a lot in common with virtual machine monitors (VMMs): both provide a substrate on top of which the "real" operating system is implemented. The key difference is that microkernels are designed to be a minimal layer to support arbitrary systems, while modern VMMs such as Xen are designed specifically to support (multiple) legacy operating systems. This means that virtual machines *increase* rather than decrease the size of the TCB, compared to simply running a legacy OS. Furthermore, most modern VMMs are actually much larger than a well-designed microkernel. This is not inherent—as demonstrated by L4Linux, which shows that L4 makes an excellent VMM—but is a result of the different design goals.

The story is similar for so-called *process virtual machines*, such as the Java Virtual Machine, which provide a higher-level API than classical VMMs. Here the complete language environment is part of the TCB, in addition to the operating system on which the virtual machine is hosted. They provide a good way to encapsulate untrusted applications (such as mobile code) but are no solution to the overall security problem in embedded systems.

Conclusion

The idea of microkernels has been around in one form or another for about 35 years. After a boom in the late '80s they lost popularity, mostly as a result of very poor performance exhibited by systems built on top of the popular Mach kernel. We now understand much better how to build microkernels with good performance, and it has been shown that microkernel-based systems achieve performance close to traditional (*monolithic*) systems. Still, microkernels have retained a reputation for poor performance and as academic toys. However, industry, seeing microkernels' potential as a solution to the security problems of embedded systems, is now ready to embrace them.

Further Reading

The philosophy behind L4 and microkernels was presented by Jochen Liedtke, "Towards Real Microkernels," *Communications of the ACM,* vol. 39, no. 9, pp. 70–77, September 1996. Hermann Härtig et al., "The Performance of μ-Kernel-Based Systems," *16th ACM Symposium on OS Principles* (SOSP) 1997, examined the performance of L4-based systems and described L4Linux, which in today's language is a paravirtualized Linux on L4 as a VMM. More information about L4, its implementations, and systems built on top can be found at http://l4hq.org. The home of L4Ka::Pistachio is http://l4ka.org.

Information about the seL4 and L4.verified projects can be found at http://ertos.nicta.com.au/research/. This site contains links to further publications, including Harvey Tuch et al., "OS Verification—Now!" *Tenth Workshop on Hot Topics in Operating Systems (HotOS X),* 2005.

Related is the EROS OS, which is much less of a minimal system but is more security-focused than existing L4 implementations, and is providing many of the ideas for seL4. The main publication on EROS is Jonathan S. Shapiro et al., "EROS: A Fast Capability System," *17th ACM Symposium on OS Principles (SOSP),* 1999.

NOTE

1. The C compiler is, strictly speaking, also part of the TCB, but it is not part of what is shipped to the customer and therefore cannot be compromised by worms or viruses.