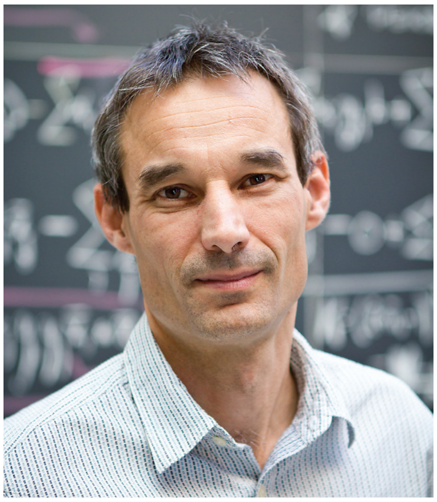
DOI:10.1145/2043174.2043201

Leah Hoffmann

Q&A
Scaling Up

M. Frans Kaashoek talks about multicore computing, security, and operating system design.

M. FRANS KAASHOEK'S interest in computing was sparked, like many others in the field, by an early love for programming. At Vrije Universiteit, he discovered he could turn his hobby into a career, and studied with MINIX creator Andrew S. Tanenbaum before accepting a professorship at Massachusetts Institute of Technology's Department of Electrical Engineering and Computer Science. Kaashoek has since conducted wide-ranging research in computer systems, including operating system design, software-based network routing, and distributed hash tables, which revolutionized the storage and retrieval of data in decentralized information systems. He also helped found two startups: Sightpath, a video broadcast software provider that was acquired by Cisco Systems in 2000, and Mazu Networks, which was acquired by Riverbed Technology in 2009. Kaashoek was named an ACM fellow in 2004 and elected to the National Academy of Engineering in 2006. Last year his work was recognized with an ACM-Infosys Foundation Award (see "Unlimited Possibilities" in the June 2011 issue of *Communications*).



You have said that your work on the exokernel operating system, which enables application developers to specify how the hardware should execute their code, was driven by intellectual curiosity. Can you elaborate?

We wanted to explore whether we could build a kernel interface that defines no abstractions other than what the hardware already provides, and that exports the hardware abstractions directly to applications. Traditionally, the kernel provides a fixed set of unchangeable abstractions. For example, you have a very complex, unchangeable kernel interface like traditional Unix systems, or you have a small, unchangeable microkernel interface, which defines a few carefully chosen abstractions. An exokernel design allows the programmer to define its own operating system abstractions.

For its minimalism, it sounds almost like an extreme version of microkernel design.

The main goal with a microkernel is to make the kernel small. That was not necessarily our goal. So, for example, we would have been perfectly happy to put a device driver inside the kernel if we thought it was the right thing to do.

How did the project evolve?

We were able to build a prototype that demonstrated the approach could work in practice. But I don't think there's any direct technology transfer from our ideas into products although there was one startup that used our code. The impact has been more indirect. Academically, it influenced other systems that were built afterward. On the more commercial side, it also has been credited in work on machine monitors for handheld devices.

Operating systems design has become such a partisan issue. What is your take on it?

I have a pragmatic view. In research, taking an extreme position is interesting because it forces you to clarify your thinking and solve the hard case. In practice, I think people are going to do whatever helps solve the particular problems they have. If you look at a monolithic kernel like Linux—I know you can't call it a microkernel system, but some of the servers run as applications in user space and some run in the kernel, and it really becomes shades of gray. And some people draw this line slightly differently than others. But if the kernel is already working fine, why change it?

Since your work on exokernels, you have done several other projects on operating systems design, in particular as it relates to multicore computing.

You might say that multicore has nothing to do with the operating system because it is, in many ways, already inherently parallel; it provides processes that can run on different cores in parallel. But many applications rely heavily on operating system services, particularly systems applications like email and Web servers. So if the operating system services don't scale well, those applications can't scale well, either.

So your work is focused on building scalable operating systems.

Originally, we thought we would have to write [CONTINUED ON P. 143]

PHOTOGRAPH BY DOMINIC CASSERLY

[CONTINUED FROM P. 144] an operating system from scratch to figure it out, which we did. Then we looked at our findings and realized they should be applicable to any standard operating system. So, with a few of my colleagues and students, we did a study to see how much work would be necessary to scale the Linux kernel to a large number of cores. If you have enough manpower, it's certainly doable.

This is the system you built in which eight six-core chips were used to simulate the performance of a 48-core chip.

Yes, indeed. There are a lot of interesting problems to be solved, but my general sense is that things are going to evolve in the right direction, and that there won't be a point in time where we have to throw everything away and start over again.

Another insight to come out of that work was that it can be difficult to identify the root cause of performance issues. Is that what inspired your work on MOSBENCH, a set of application benchmarks designed to measure the scalability of operating systems?

Yes, MOSBENCH came out of that project. Typical benchmarks are just application benchmarks, where all the action is in the application itself. But we needed a benchmark that included a lot of system-intensive applications. Otherwise, you don't stress the operating system, and if you don't stress the operating system, it isn't scalable by default. So we collected several applications to stress different parts of the operating system—essentially, it's a workload generator.

What conclusions has it led to so far?

The Linux kernel scales pretty well. But there might be interesting future problems. One direction is having the operating system give you more control over the caches in which the data lives. The traditional view is that the cache is hidden from the operating system and the hardware just does its job of caching. In multicore, caches are spread all around the chip, some close by and others that are far away. There are cases where you want control over where the data is placed so you can get better performance. Something else we're looking at are abstractions that allow you to build operating systems that are scalable by design, as opposed to scaling every subsystem one by one. New concurrent data structures that exploit weak consistency semantics are another direction.

"One of the big advantages of academia is that if you decide the problem's not interesting, you can change. That's a hard thing to do in a startup."

You have also done work on systems security, using information flow control to prevent the unauthorized disclosure of data.

The idea is simple. Typically when you build an application, and you want to make it secure, you put a check before every operation that might be sensitive. The risk is that you can easily forget a check, which can then be exploited as a security vulnerability. We tried to structure the operating system in such a way that even if you forget some of these checks, security is not immediately compromised. The way we do it is to draw a box around the operating system and label all data. Then we have a guard that checks whenever data is being sent across the border to make sure it's going to the right place, based on the data's label.

Some of your other security research focuses on making it easier to restore system integrity after an intrusion. So-called "undo computing," for instance, seeks to undo any changes made by an adversary during the attack while preserving legitimate user actions.

Let's say you have a desktop, and you discover it was compromised a couple weeks after an attack. Then the question is, How do you restore its integrity? You could go back to a backup from three weeks ago, when you know it's clean, and reinstall some pieces. But that's clearly a labor-intensive project. Or you try to find all the bad code and files, and remove them, which of course is also labor intensive. There are some automatic virus removers, but they're very specific to a particular virus.

What is your approach?

Here's one direction my colleague Nickolai Zeldovich and our students are exploring: Once you've determined that an adversary sent bad packets to your Web server, you know everything that could be influenced by those packets is suspicious, and all the influenced actions must be undone. We roll the system back to before the attack happened, and roll forward all the actions that were not influenced by the adversary's actions. If everything works out correctly, you will end up in a clean state, but you will still have all the work that you did in the last three weeks.

What if the actions of the adversary are intermingled with the actions of the user?

Undoing that intermingling and keeping track of the dependencies requires some reasonably sophisticated techniques. Another aspect of the problem is that you really don't want to replay or redo every operation. So we have a bunch of clever observations saying, well, this work or this operation could never have been influenced by the attacker's actions, so therefore we don't have to redo them. We have some encouraging results, but we're still trying to figure out whether we can make this work in practice for heavily used complex systems.

Do you have plans to do another startup?

I'm going to wait and see. It's not until the later stages of a project that I think about whether it solves a real problem that people have and, if so, would it be worthwhile to start a company around it. One of the big advantages of academia is that if you decide the problem's not interesting, you can change. That's a hard thing to do in a startup.

Leah Hoffmann is a technology writer based in Brooklyn, NY.

© 2011 ACM 0001-0782/11/12 $10.00