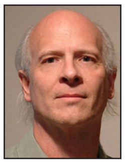
Interview with Amit Levy

RIK FARROW

Amit Levy is a PhD student in the Department of Computer Science at Stanford University. His work focuses on building pragmatic, secure systems that increase flexibility for application developers while preserving end-user control of private data. amit@amitlevy.com



Rik is the editor of *;login:*. rik@usenix.org

I'd met Amit Levy a couple of times during luncheons at system conferences. Amit is not shy about talking about his projects. I liked hearing about them, as Amit would clearly tell me about the motivations behind his projects and answer any questions I had.

So this time after we talked at OSDI '16, I asked him if I could create a more formal version of our post-luncheon conversations, and he agreed. In particular, we talked about his work on Tock using Rust and leveraging type safety.

Rik Farrow: You've done a lot of things, including your side-project MemCachier [1], but you've published more about security-related topics. What got you interested in building a replacement for TinyOS [2]?

Amit Levy: Almost all of my work has had something to do with using type safety as a means of building secure systems. Even MemCachier really started as a an exercise to learn Go and with the idea that building a memcached clone in a type-safe language would make it relatively easy to also build a safe, multi-tenant cache service. So, in that sense, rethinking the embedded operating system in the context of IoT security was a pretty natural extension of much of what I'd been working on, just a different application space. For *me* the exciting thing about Tock [3] is really figuring out how to provide safety and isolation properties to a system with extremely limited resources. And the context is allowing IoT platforms to run untrusted programs.

The actual story is just more coincidental. My roommates and I wanted to build an automatic lock for our front door after we forgot to lock it a couple times and two of our bikes were stolen. So I started looking into IoT and, particularly, low-power computers and Bluetooth low energy. Phil Levis was also interested in Bluetooth (for much less frivolous reasons), so we started reading the spec together and talking about ideas. Eventually, Phil, Prabal Dutta, and David Culler decided their students should start having weekly phone calls about software/hardware co-design, and the need for a replacement for TinyOS just came out of those weekly phone calls.

RF: You've mentioned that Tock will run on a SAM4L processor, which certainly does appear to be low power, as well as much simpler and much slower (under 100 MHz clock) than what most systems use. Do platforms like this have any hardware features that support security, things like memory management or the system call interface?

AL: Yes. Most of the new ARM Cortex-M series microcontrollers (including the SAM4L) have a feature called a memory protection unit (MPU). The MPU does not provide memory virtualization (so there is only a single address space) but does enable setting read/write /execute permission bits on ranges of memory as granular as 16 bytes. In fact, Tock uses the MPU to enable a limited number of traditional OS processes. ARM also recently released a specification for TrustZone-M, which has similarities to TrustZone on "application"-grade ARM processors like the ones in our cell-phones. TrustZone-M has some additional interesting features (like allowing interrupts to trap to untrusted code directly), which could help increase performance of embedded systems that rely on hardware protection. I think we're expecting to see some SoCs (system-on-chip) with TrustZone-M available in the next couple of years.

However, there just isn't enough memory on these microcontrollers to use a protection model based on memory isolation (e.g., processes) as a ubiquitous means of protection in the system.

In general, though, I think the simplicity of microcontrollers can be viewed as a hardware security feature. What I mean is that in many use cases, we also care about hardening embedded systems against hardware-based side-channel attacks—like timing and power analysis. TPMs (trusted platform modules), two-factor authentication devices, and HSMs (hardware security modules) are a few examples of systems where it's really important to mitigate side-channel attacks. To thwart these attacks, it's important for the hardware to be simple. Caches, like the TLB on higher-grade processors, are notoriously leaky.

RF: How does type safety improve security?

AL: Type safety serves two primary roles. It helps programmers avoid many common errors like buffer-overflows. When hardware protection is available, it's possible to catch some of these kinds of bugs at runtime. Type safety lets us catch them at compile time, before we run our program, and saves us from them when hardware protection isn't an option.

The second role is that we can leverage type safety to express really fine-grained security policies. For example, hardware protection lets me expose only certain regions of memory to untrusted code—say a memory-mapped I/O register. However, I have no control over what values are written to that memory. Type safety lets me restrict the manner in which the untrusted code uses a region of memory. For example, I can ensure that only a certain range of values is ever written to a particular register or that the value was created by a trusted module. Importantly, the compiled binary looks nearly identical to one compiled from source code in C that doesn't have these protections. There's nothing particularly magical going on. The type system just lets the compiler reject code that violates certain rules, and, in most cases when we're writing C, we don't really want to violate those rules anyway.

RF: So you have some untrusted code, and you can't distinguish it from code written in C once it's compiled. That implies to me that you can't rely on type safety here, because the untrusted code could have been compiled from C, and thus you don't know what types it can write to your target memory. I am likely just missing something here, so could you clear this up?

AL: You're right, if all you have is a pre-compiled binary, the type system doesn't help. You have to be able to compile the code yourself. In Tock, this is part of what motivates which systems components go where. Applications, which may even be loaded by an end user in some cases, typically live in a process. The process is isolated by hardware protection, so it doesn't rely on the type system and a binary is fine. Conversely, components like peripheral drivers are specific to a hardware platform—my particular embedded product has a different set of sensors, actuators, radios, etc. from other embedded products—but don't change when I change applications. The system integrator wants to make sure that if they use a driver for a particular temperature sensor they found on the Web that it's not able to leak secret encryption keys or access other peripherals on the same bus, but if they can verify safety when they compile the kernel that's fine.

RF: In some of your work [4], you talk about problems you have when using Rust. Can you explain?

AL: Rust kind of provides the lowest-level of abstraction you need to guarantee type safety. This ends up surfacing some fundamental safety tradeoffs into the language. One of the simpler examples is that if you want to use closures-based callbacks (e.g., as is common in Node.js), you need to dynamically allocate those closures—they can't be on the stack or statically allocated. Most type-safe languages assume that more or less everything is dynamically allocated, so this is implicit, while in Rust it's explicit.

In Tock, we disallow dynamic allocation in the kernel (that's a common practice for reliable systems), so this is good for us because it means we can use closures as long as we can prove to the compiler that they don't need to be dynamically allocated. However, it also means that when we try to adopt common coding styles from other frameworks that don't actually work with our system constraints, we get a compiler error. I think it's tempting as a systems builder to look at type-safe languages and think that they are magic, and so you get to stop thinking about system constraints. That's not true. There's nothing magic about type safety. It just lets you guarantee things you already knew how to do.

Unfortunately, I think it's easy to draw the wrong conclusion from that paper—that there are drawbacks with Rust that are artifactual rather than fundamental. There were three issues that we ran into building Tock in Rust, and all three of them turned out to be fundamental (or at least nearly fundamental) and, on balance, were the right design decisions for the language. There is a great paper by Dan Grossman from 2002 called "Existential Types for Imperative Languages" [5] that explains this really well. If you're going to read our paper, it's worth reading that one as well.

References

[1] MemCachier: https://www.memcachier.com/.

[2] TinyOS: http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Documentation_Wiki.

[3] TockOS: http://www.tockos.org/.

[4] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto, "Ownership Is Theft: Experiences Building an Embedded OS in Rust," in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*, October 2015: https://sing.stanford.edu/site/publications/59.

[5]: Dan Grossman, "Existential Types for Imperative Languages," in *Proceedings of the 11th European Symposium on Programming Languages and Systems (ESOP '02)*, pp. 21–35: https://homes.cs.washington.edu/~djg/papers/exists_imp.pdf.