TOCK: A SECURE OS FOR EMBEDDED PLATFORMS

Amit Levy, PhD Candidate @ Stanford

January 7th, 2016

SECURING THE INTERNET OF THINGS

A Security Disaster

The Economist | World politics | Business & finance | Economics | Science & technology | Culture

Cyber-security

The internet of things (to be hacked)

Hooking up gadgets to the web promises huge benefits. But security must not be an afterthought

Jul 12th 2014 | From the print edition | Timekeeper | Like 217 | Tweet 594

How the Internet of Things Could Kill You

By Fahmida Y. Rashid JULY 18, 2014 7:30 AM - Source: Tom's Guide US | 5 COMMENTS

Hacking the Fridge: Internet of Things Has Security Vulnerabilities

JESS SCANLON | MORE ARTICLES
JUNE 28, 2014

Philips Hue LED smart lights hacked, home blacked out by security researcher

By Sal Cangeloso on August 15, 2013 at 11:45 am | 7 Comments

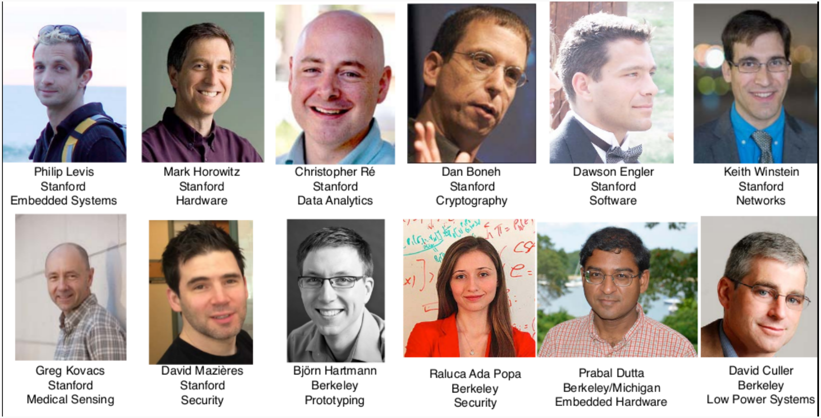
HP conducted a security analysis of IoT devices

- 80% had privacy concerns
- 80% had poor passwords
- 70% lacked encryption
- 60% had vulnerabilities in UI
- 60% had insecure updates

Securing the Internet of Things

- Secure Internet of Things Project
  - 3 universities: Stanford, Berkeley, and Michigan
  - 12 faculty collaborators
- Rethink IoT systems, software, and applications from the ground up
- Make a secure IoT application as easy as a modern web application

Who we are?


Philip Levis
Stanford
Embedded Systems
Mark Horowitz
Stanford
Hardware
Christopher Ré
Stanford
Data Analytics
Dan Boneh
Stanford
Cryptography
Dawson Engler
Stanford
Software
Keith Winstein
Stanford
Networks
Greg Kovacs
Stanford
Medical Sensing
David Mazières
Stanford
Security
Björn Hartmann
Berkeley
Prototyping
Raluca Ada Popa
Berkeley
Security
Prabal Dutta
Berkeley/Michigan
Embedded Hardware
David Culler
Berkeley
Low Power Systems

The Internet(s) of Things

Networked Devices

Tens/person
Uncontrolled Environment
Cloud integration
Stationary
Safety requirements

WiFi/802.11
TCP/IP
IEEE/IETF

Industrial Automation

Thousands/person
Controlled Environment
Closed systems
Stationary
Industrial requirements

WirelessHART, 802.15.4
6tsch, RPL
IEEE/IIC/IETF

Home Area Networks

Hundreds/person
Uncontrolled Environment
Proprietary standards
Stationary
Consumer requirements

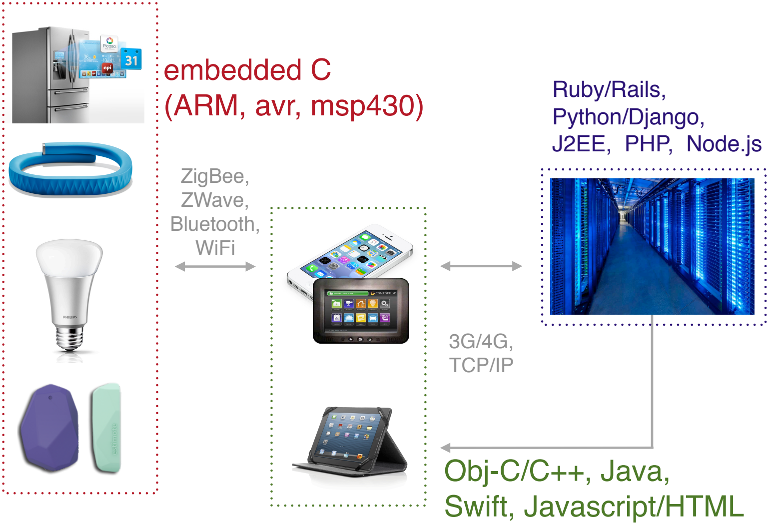
ZigBee, Z-Wave
6lowpan, RPL
IETF/ZigBee/private

Personal Area Networks

Tens/person
Personal environment
Open standards
Mobile/pervasive
Fashion vs. function

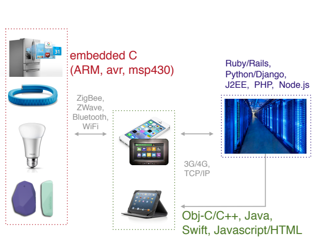
Bluetooth, BLE
3G/LTE
3GPP/IEEE

IoT: MGC (eMbedded Gateway Cloud) Architecture



IoT Security is Hard

- Complex, distributed systems
  - 10^3 - 10^6 differences in resources across tiers
  - Many languages, OSs and networks
  - Specialized hardware



- Just *developing* applications is hard
- *Securing* them is even harder
  - Enormous attack surface
  - Reasoning across hardware, software, protocols etc
  - What are the threats and attack models?
- Valuable data: location, presence, medical...
- Rush to development + hard = avoid now, deal later

Architectural Principles

End-to-end: consider security holistically, from data generation to end-user display.

Transparency: we must be able to observe what our devices are saying about us.

Longevity: these systems will last for up to 20 years and their security must too.

TOCK: A SECURE OS FOR EMBEDDED PLATFORMS

Tock is an embedded operating system we’ve been building for about a year.

Tock is an embedded operating system we’ve been building for about a year.

- Event-driven

Tock is an embedded operating system we've been building for about a year.

- Event-driven
- Flexible/extensible to any platform

Tock is an embedded operating system we've been building for about a year.

- Event-driven
- Flexible/extensible to any platform
- Multi-programmable

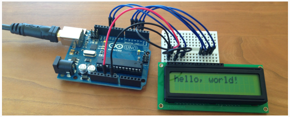
Tock is an embedded operating system we've been building for about a year.

- Event-driven
- Flexible/extensible to any platform
- Multi-programmable
- Principle Least-privilege

WHY NOW?

- Rapid prototyping
- Open Source
- "Ship early"
- "Ship often"
    - How many software systems go unchanged for 20 years?
- Small-batch hardware

Embedded Systems as Platforms



GOALS

Untrusted Applications

- Isolated from each other and kernel
- Can only access hardware subject to policies
- Cannot crash the system
- Updatable at runtime

Untrusted Kernel Subsystems

- Memory-isolated from each other, core kernel
- Only trusted by applications that use them
- Hardware access through limited interface (e.g. virtualized)

Small (and simple) Trusted Core

Reliable & Performant

Reason about memory requirements at *compile-time*

- Either the kernel fits or it doesn't

Applications cannot starve system resources

- Hardware access non-blocking
- Time-sliced scheduling

Isolation shouldn't impact performance

- Satisfy real-time constraints

Portable and Flexible

Cortex-M based microcontrollers

- Memory Protection Units
- Reasonable memory requirements: ~3KiB kernel

Platform-specific configuration

- Drivers hardware agnostic
- Construct a platform declaratively

Small & extensible system call interface

- Currently 4 system calls

WHY IS IT HARD?

(Limited) Hardware Isolation Mechanisms

Traditional multi-programming OSs rely on virtual addressing

- Isolation
- Over-provisioning (e.g. swapping to disk, paging)
- Dynamic application loading
    - don't need to know physical memory location ahead of time

We only have "Memory Protection"

- Read/write/execute bits
- ...but no virtualization
- Limited number of regions

40 Years of Programming Language Research

- Memory safety
  - e.g. no buffer overflows
- Strict type enforcement
  - e.g. no unsafe type casts
- Richer type systems
  - Generics
  - Interfaces
- High-level features
  - Closures
  - Map/Fold/Iterators...

40 Years of Programming Language Research

- (almost) All type-safe languages have a runtime
    - Automatic memory management (via Garbage collection) for safety
- Need control over memory layout
- Performance and reliability issues:
    - Garbage collection vs. timing constraints
    - Dynamic memory allocation vs. compile-time memory requirements
- Porting runtime systems for each chip is hard

- Memory and type safety
- Eliminate large classes of bugs at *compile time*
- Strong type-system can allow component isolation
- Low-level primitives can enable rich security systems

Rust: A Type-Safe “Systems” Programming Language

- Memory and type safety
- Eliminate large classes of bugs at *compile time*
- Strong type-system can allow component isolation
- Low-level primitives can enable rich security systems

We don’t know how to build systems in such a language

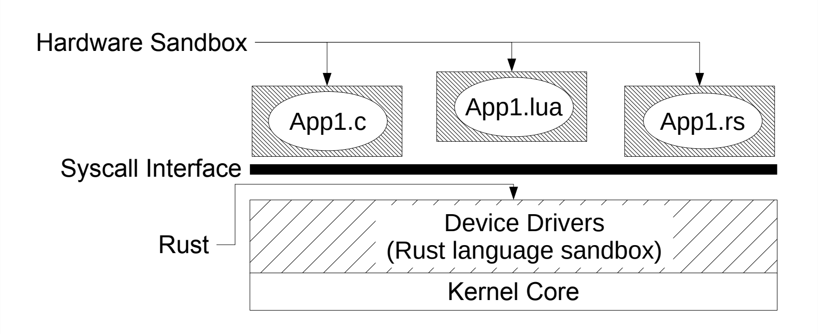
- Memory and type safety
- Eliminate large classes of bugs at *compile time*
- Strong type-system can allow component isolation
- Low-level primitives can enable rich security systems

We don't know how to build systems in such a language

Yet!

DESIGN

System Architecture



Memory Protection for Application Isolation

Applications run in “user-land”

- No direct access to hardware
- Can only access memory it owns

Flexible programming environment

- Written in any language[1]
- Dynamic memory allocation
- Can “lend” memory to drivers (e.g. for buffers)

[1]Currently have a C runtime, experimental Lua and C++ runtimes

Constraints

- Write-only text-segment
- Text and data segments not near each other
- No virtual addressing

Dynamic Application loading

Constraints

- Write-only text-segment
- Text and data segments not near each other
- No virtual addressing

Solution

In short: gnarly GCC options

- Compile apps with position independent code (PIC)
- Kernel dynamically sets PIC base

Language Sandbox

Leverage the Rust language's type-system to isolate untrusted drivers

- Drivers only have access to explicitly allowed hardware resources
- Cannot address arbitrary memory
- Only consensual access to applications

Hope for even richer security policies:

- Resource constraints?
- Mandatory access control?

RUST

Why Rust?

Two distinguishing properties from other safe languages:

- Enforces memory and type safety without a garbage collector
- Explicit separation of trusted vs. untrusted code
    - Untrusted code is strictly bound by the type system
    - Trusted code can circumvent the type system

Rust avoids the runtime overhead of garbage collection by using *ownership* to determine when to free memory at *compile-time*.

Each Value has a Single Owner

Key Property

When the owner goes out of scope, we can deallocate memory for the value.

Each Value has a Single Owner

Key Property

When the owner goes out of scope, we can deallocate memory for the value.

Memory for the value `43` is allocated and bound to the variable `x`.

```
{
  let x = 43
}
```

When the scope exits, `x` is no longer valid and the memory is "freed"

Each Value has a Single Owner

Single owner means *no aliasing*, so values are either copied or moved between variables.

Each Value has a Single Owner

Single owner means *no aliasing*, so values are either copied or moved between variables.

This is an error:

```
{
  let x = Foo::new();
  let y = x;
  println("{}", x);
}
```

because `Foo::new()` has been moved from `x` to `y`, so `x` is no longer valid.

Functions must explicitly hand ownership back to the caller:

```
fn bar(x: Foo) -> Foo {
  // Do stuff
  x // <- return x
}
```

Or can use `borrows`: a type of reference which does not invalidate the owner.

```
fn bar(x: &mut Foo) {
  // Do stuff
  // the borrow is implicitly released.
}

fn main() {
  let mut x = Foo::new();
  bar(&mut x);
  println!("{}", x); // x still valid
}
```

Borrows

Borrows are resolved at compile-time, with some constraints:

- A value can only be *mutably* borrowed if there are no other borrows of the value.
- Borrows cannot outlive the value they borrow.
- Values cannot be moved while they are borrowed.

The Bad

Ownership doesn't allow circular dependencies.

But circular dependencies are everywhere in real systems.

Ownership: The Good and The Bad

The Bad

Ownership doesn't allow circular dependencies.

But circular dependencies are everywhere in real systems.

The Good

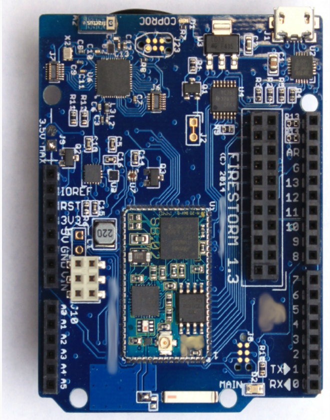
Once the compiler verifies type safety, the resulting code looks very close to compiled C-code.

CONCLUSION

Tock

- Event-driven
- Flexible/extensible to any platform
- Multi-programmable
- Principle Least-privilege

Our Progress So Far



- Second clean re-design iteration
    - I think we're done this time :)
- Implementation for Atmel SAM4L based Firestorm platform
- Drivers for virtualized UART, TMP006, GPIO
- Coming very soon:
    - Bluetooth Low Energy (using nrf51822)
    - 802.15.4 (using rf233)

Challenges & Questions

- What are the real threat models?
- How to leverage a safe type system for OS security?
- Multi-programming without virtual memory
- What's the interface for untrusted kernel drivers?