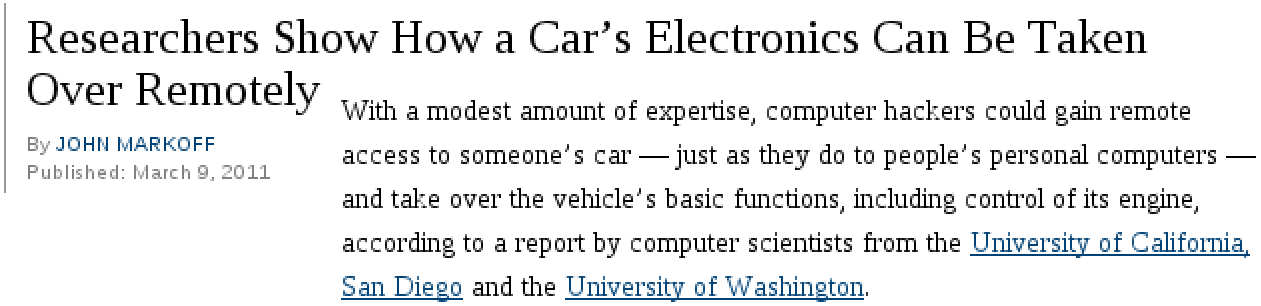
TrackOS: a Security-Aware RTOS

Lee Pike, Pat Hickey, Trevor Elliott, Aaron Tomb, Eric Mertens (Galois Inc.)
David Kapp (AFRL)

HCSS | 2013

Embedded System (in)Security

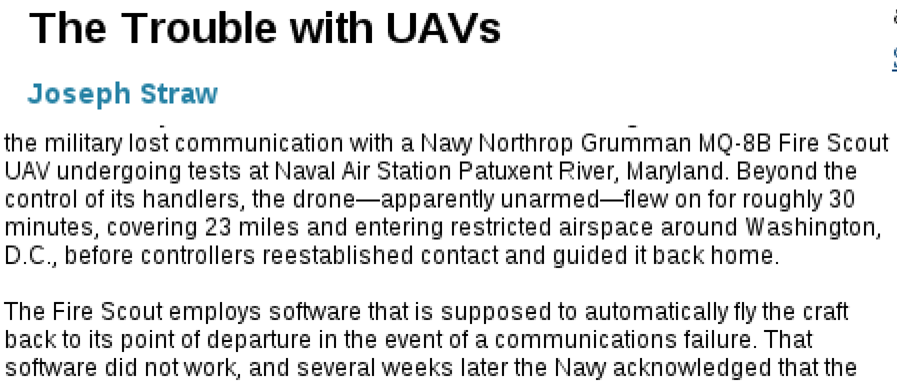
src: The New York Times



Researchers Show How a Car's Electronics Can Be Taken Over Remotely

By JOHN MARKOFF
Published: March 9, 2011

With a modest amount of expertise, computer hackers could gain remote access to someone's car — just as they do to people's personal computers — and take over the vehicle's basic functions, including control of its engine, according to a report by computer scientists from the University of California, San Diego and the University of Washington.



The Trouble with UAVs

Joseph Straw

the military lost communication with a Navy Northrop Grumman MQ-8B Fire Scout UAV undergoing tests at Naval Air Station Patuxent River, Maryland. Beyond the control of its handlers, the drone—apparently unarmed—flew on for roughly 30 minutes, covering 23 miles and entering restricted airspace around Washington, D.C., before controllers reestablished contact and guided it back home.

The Fire Scout employs software that is supposed to automatically fly the craft back to its point of departure in the event of a communications failure. That software did not work, and several weeks later the Navy acknowledged that the

src: http://www.securitymanagement.com/news/trouble-with-uavs-008118

Stuxnet

Siemens Simatic S7-300.
src: http://en.wikipedia.org/wiki/File:S7300.JPG

src: http://en.wikipedia.org/wiki/File:FIRESCOUT-VUAS.jpg

How to Attack Embedded Software

- Let's focus on software *integrity*
- The same attacks from the 80s still work!
  - Typical approaches:
    - Shellcode: inject a new program
    - Return-oriented programming: build a new program from spare parts
    - Reflashing: overwrite data (including the original program)
  - In addition, for real-time programs, you can also change timing
    - Used in Stuxnet to destroy centrifuges

Traditional Approaches to Security

Traditional security approaches make attacks harder...

- Address-space layout randomization
- Write XOR Execute (W^X)
- Stack canaries

Traditional Approaches to Security

Traditional security approaches make attacks harder...

- Address-space layout randomization
- Write XOR Execute (W^X)
- Stack canaries

But not impossible:

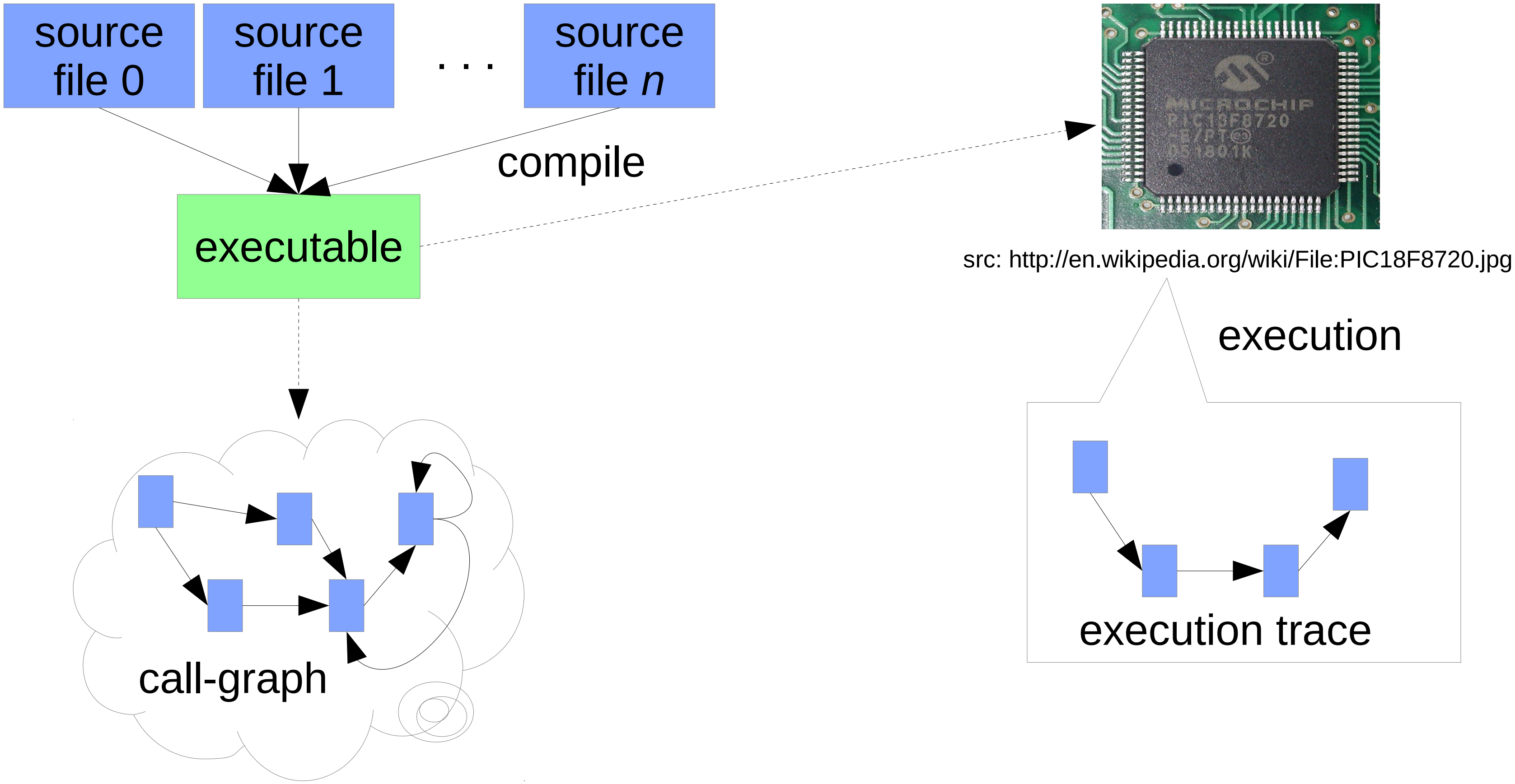
> *Nobody ever defended anything successfully, there is only attack and attack and attack some more.*
>
> – General George S. Patton

From Roadblocks to Unbypassable Detection

Control-flow Integrity (CFI): does a program respect its statically-computed call-graph?

From Roadblocks to Unbypassable Detection

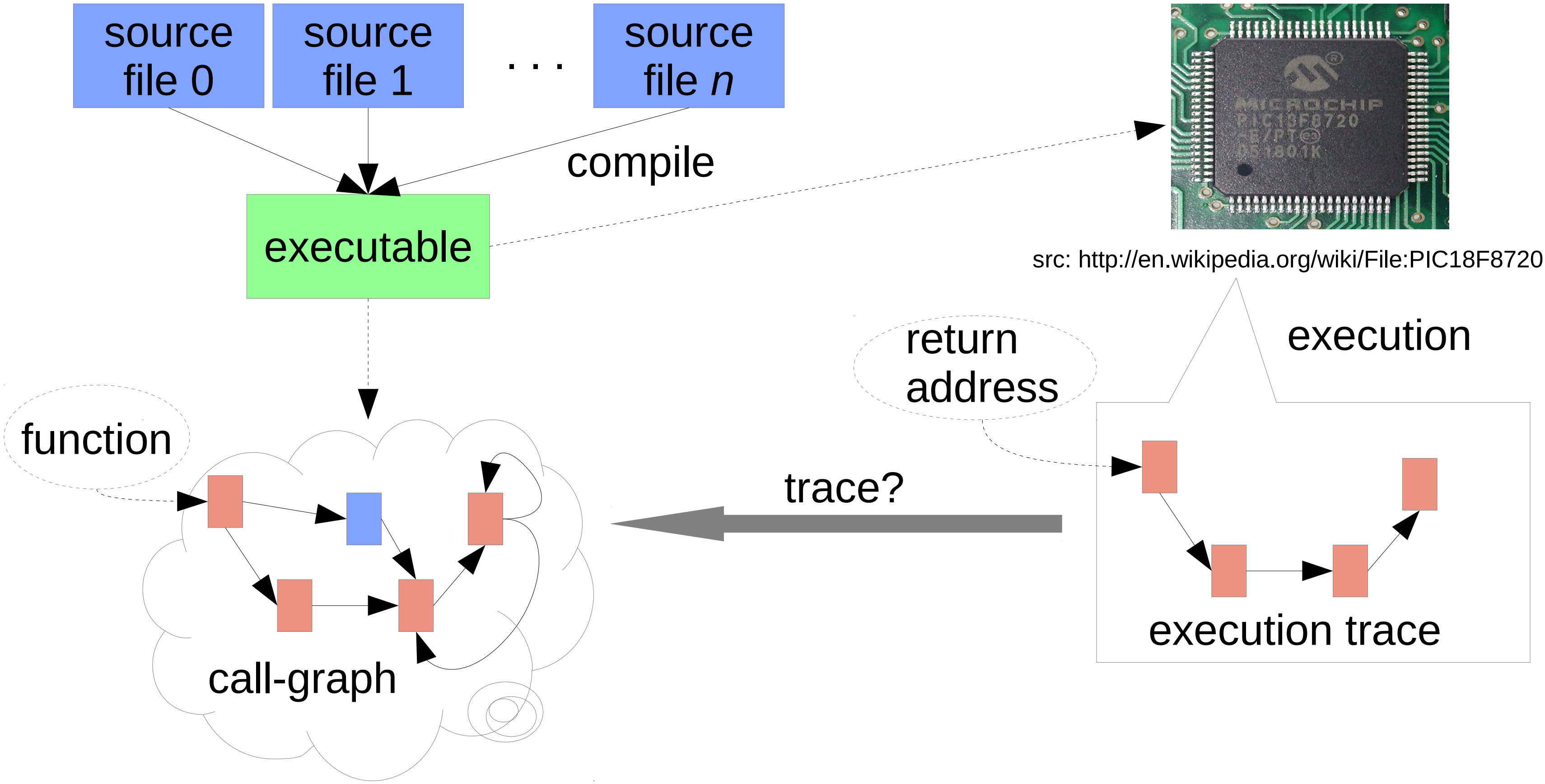
Control-flow Integrity (CFI): does a program respect its statically-computed call-graph?



src: http://en.wikipedia.org/wiki/File:PIC18F8720.jpg

From Roadblocks to Unbypassable Detection

Control-flow Integrity (CFI): does a program respect its statically-computed call-graph?



source file 0

source file 1

. . .

source file *n*

compile

executable

src: http://en.wikipedia.org/wiki/File:PIC18F8720.jpg

execution

return address

function

trace?

call-graph

execution trace

CFI Philosophy

Don't focus preventing specific attacks, but unbypassable detection of any control-flow violation

> *What we show is that these defenses would not be worthwhile even if implemented in hardware. Resources would instead be better spent deploying a comprehensive solution, such as CFI.*
>
> *– Checkoway et al. "Comprehensive experimental analysis of automotive attack surfaces," USENIX 2011*

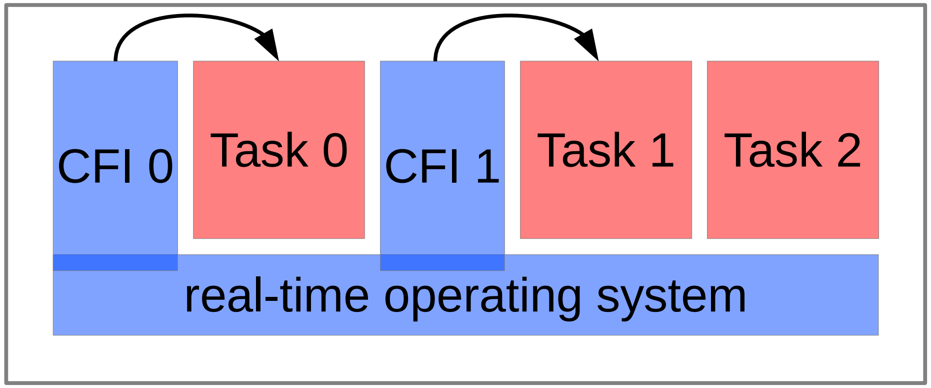
CFI for High-Integrity Embedded Systems

- CFI makes a new instrumented program. Now you traded one problem (insecurity) for two:
  1. Timing: instrumented code has new (possibly unpredicable) timing
  2. Certification: new programs may require re-certification
- State-based CFI (SBCFI) extends CFI: sample control-flow to find CFI violations
  - Decomposes the monitor and the observed program
  - Approaches have relied on virtualization and OS debugging features
  - And they don't do full CFI checks (i.e., return-oriented programming is not detected)

Our contribution: SBCFI for real-time embedded systems

RTOS-Based SBCFI

- Scheduling:
  - An RTOS already handles scheduling—CFI checker is just another task
  - Specialize CFI checks to specific applications—don't worry about concurrency
- Trust:
  - An RTOS is a small (< 5KB binaries) trustworthy basis
  - The weaknesses in embedded systems is often the application code

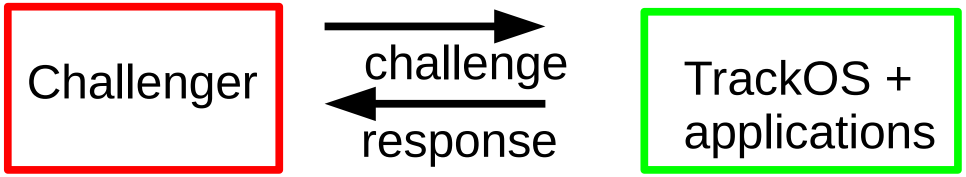


TrackOS

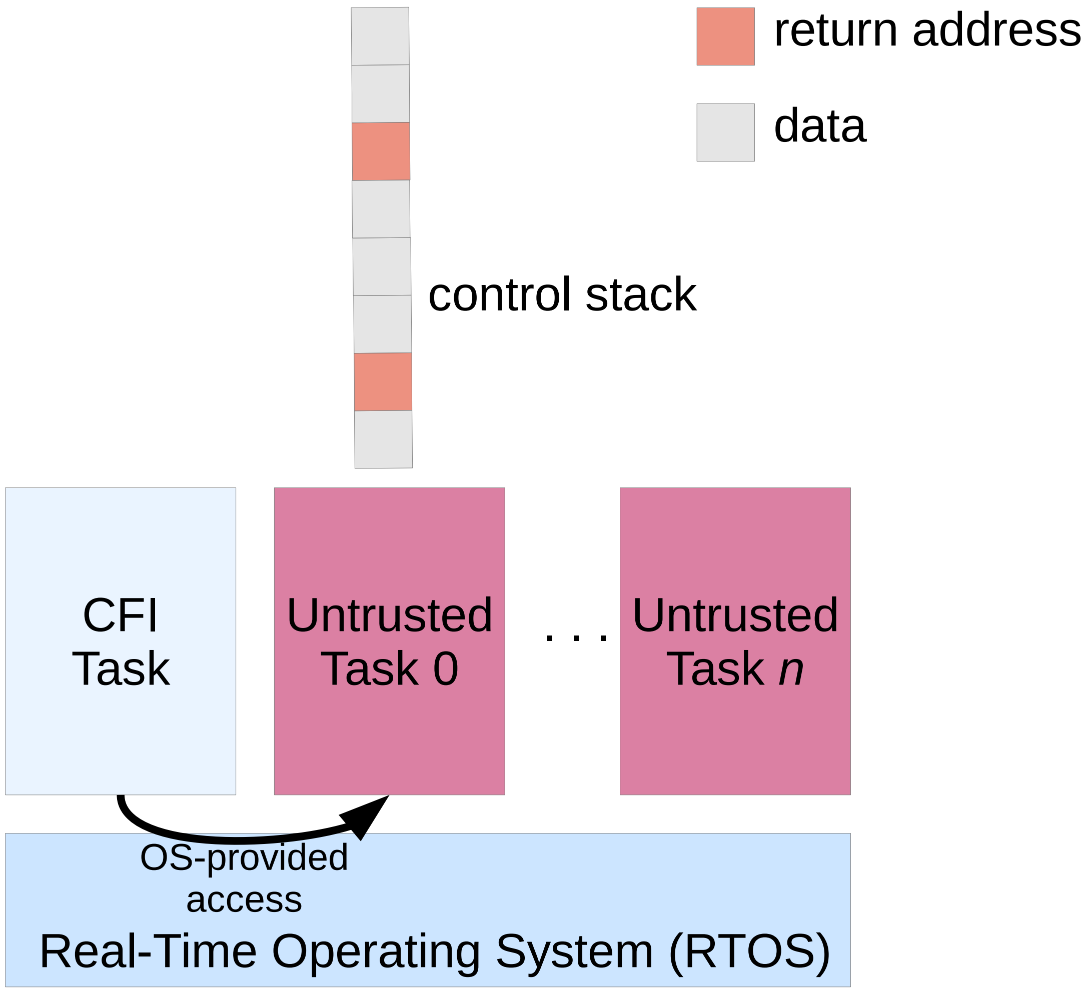
Additional aspects:

- Lightweight static analyzer to generate control-flow graphs
  - Static analysis of binaries—don't trust the compiler, or need to see sources
  - No frame-pointers—we compute stack data-usage
  - Configuration data for function pointers, assembly code
  - Able to analyze a 10k LOC sources/200KB machine imagine in ~10secs
- Data-integrity protection: software-based attestation to ensure the CFI checkers/data is not modified

See SWATT: Arvind Seshadri et al. SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy*, May 2004.



TrackOS



🟥 return address

⬜ data

control stack

CFI Task

Untrusted Task 0

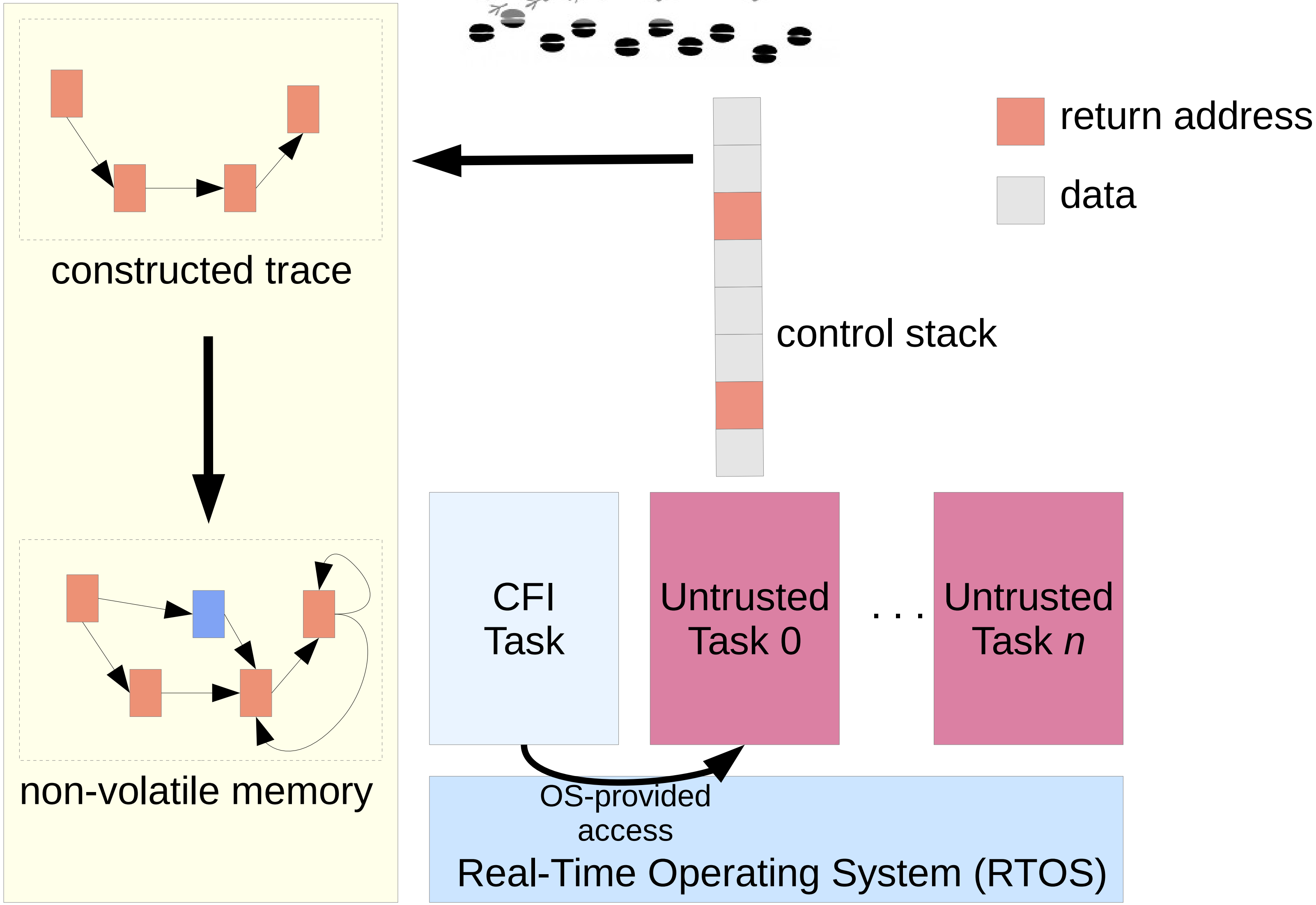
. . .

Untrusted Task *n*

OS-provided access

Real-Time Operating System (RTOS)

TrackOS



constructed trace

non-volatile memory

control stack

🟥 return address

⬜ data

CFI Task

Untrusted Task 0

. . .

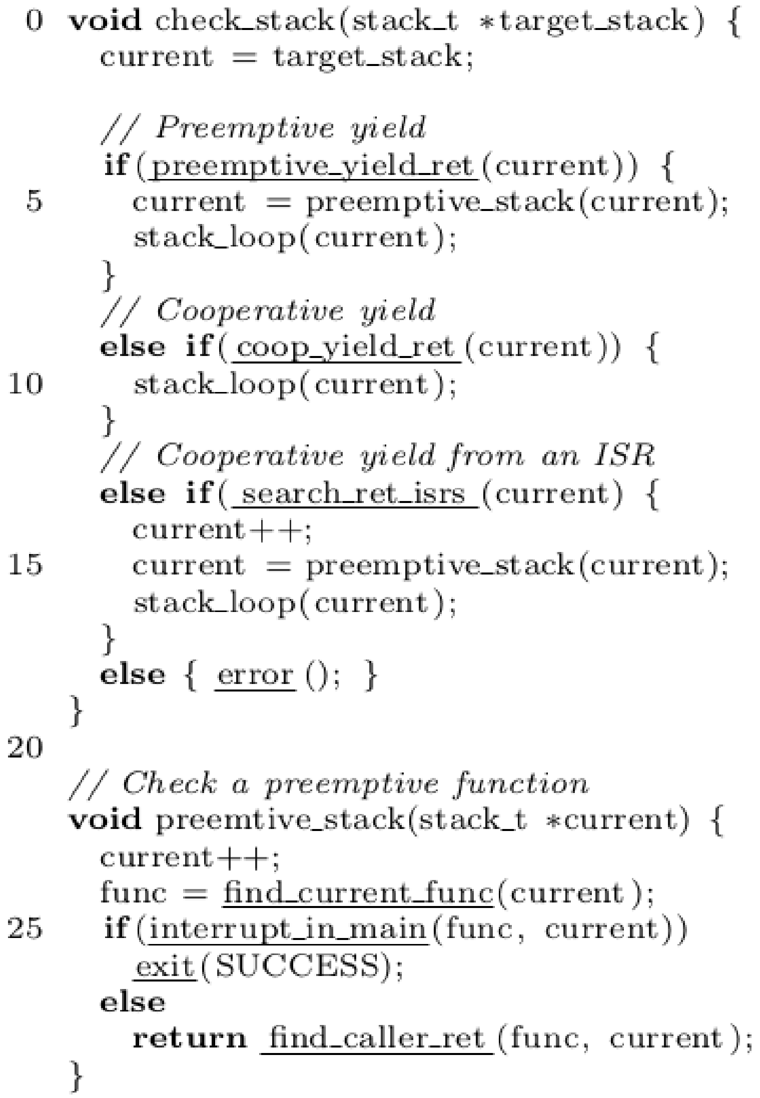
Untrusted Task *n*

OS-provided access

Real-Time Operating System (RTOS)

CFI Algorithm

1. Easier: Walk down a control stack from a known return address
2. Harder: discovering the first valid return address

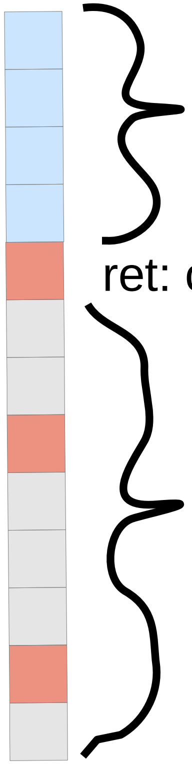


```
void check_stack(stack_t *target_stack) {
  current = target_stack;

  // Preemptive yield
  if(preemptive_yield_ret(current)) {
    current = preemptive_stack(current);
    stack_loop(current);
  }
  // Cooperative yield
  else if(coop_yield_ret(current)) {
    stack_loop(current);
  }
  // Cooperative yield from an ISR
  else if( search_ret_isrs (current) {
    current++;
    current = preemptive_stack(current);
    stack_loop(current);
  }
  else { error (); }
}

// Check a preemptive function
void preemtive_stack(stack_t *current) {
  current++;
  func = find_current_func(current);
  if(interrupt_in_main(func, current))
    exit(SUCCESS);
  else
    return find_caller_ret (func, current);
}
```

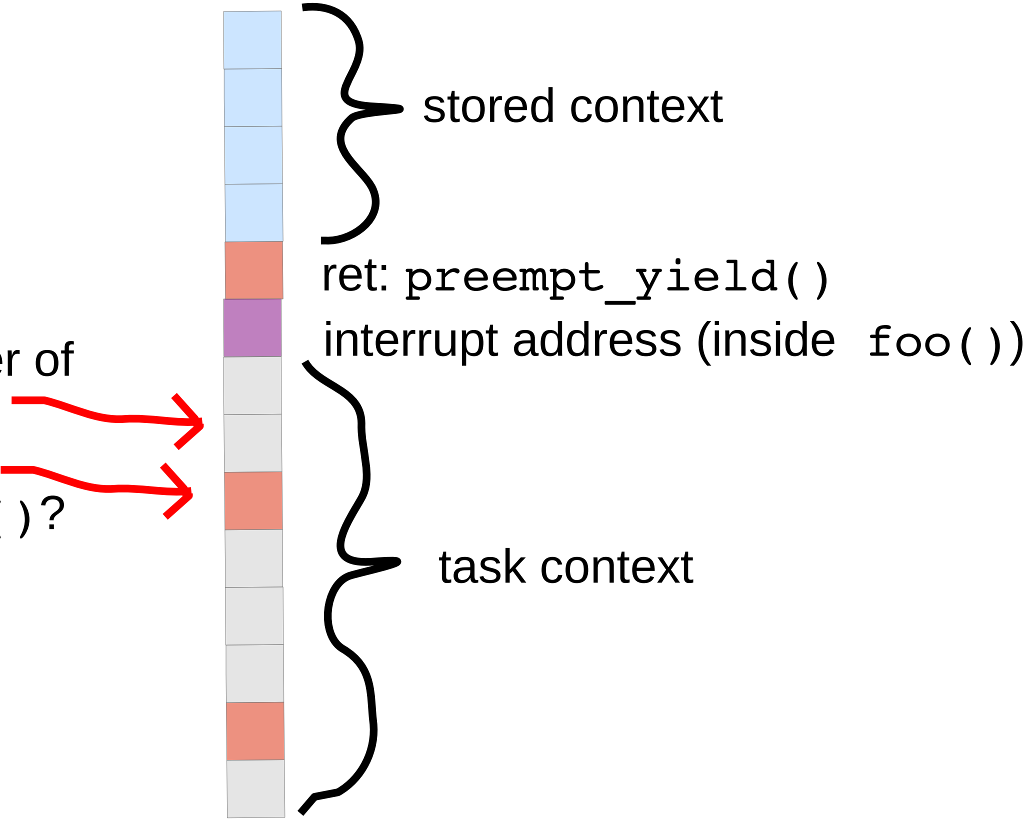
Cooperative Yield


stored context

ret: coop_yield ( )

task context

Pre-emptive Yield



stored context

ret: `preempt_yield()`

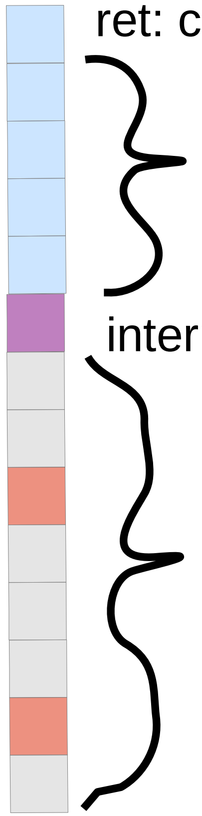
interrupt address (inside `foo()`)

valid number of data bytes?

ret: `foo()`?

task context

Yield from ISR



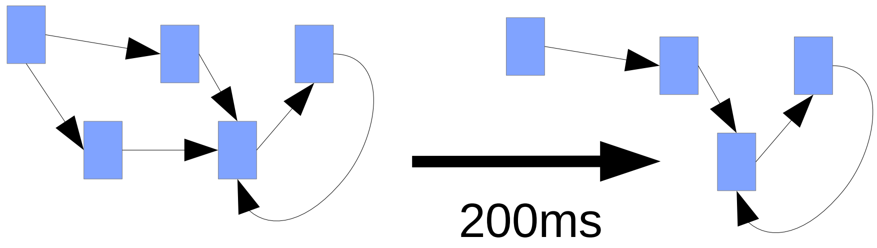
ret: coop_yield( )

stored context for ISR call

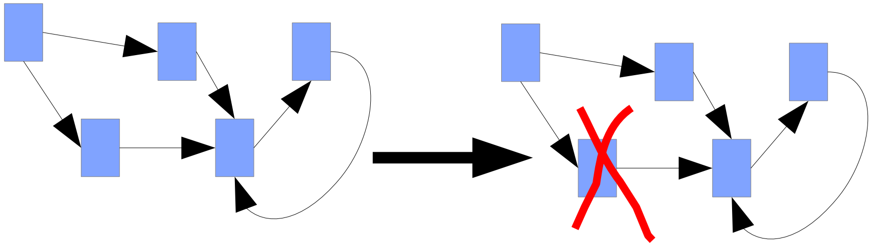
interrupt address

task context

Beyond CFI: TrackOS Extensions

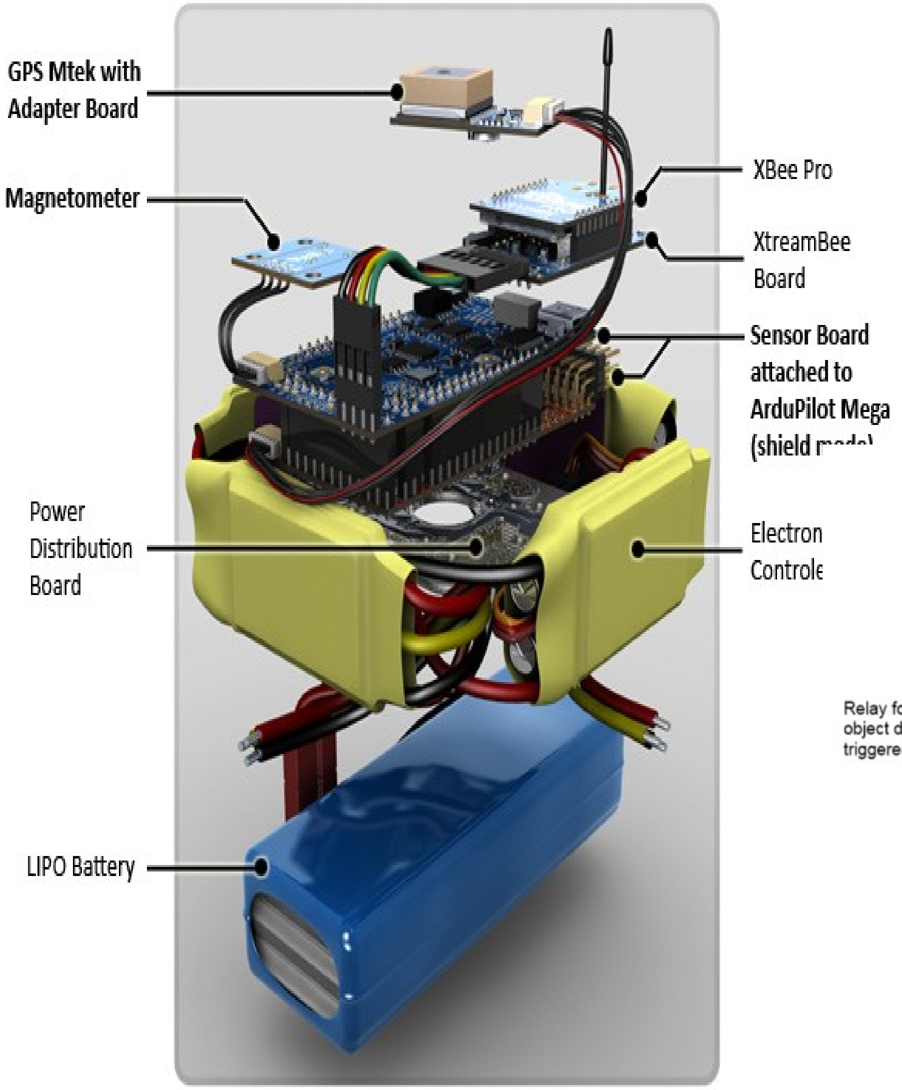


- Timing integrity: did control transfer in the expected time?
    - Example: is GPS data parsed in the expected time?



- Blacklisting: check the control stack for functions in the call-graph that become invalid
    - Example: calling startup code after initialization completes
- Temporal Logic

src: https://github.com/diydrones/ardupilot



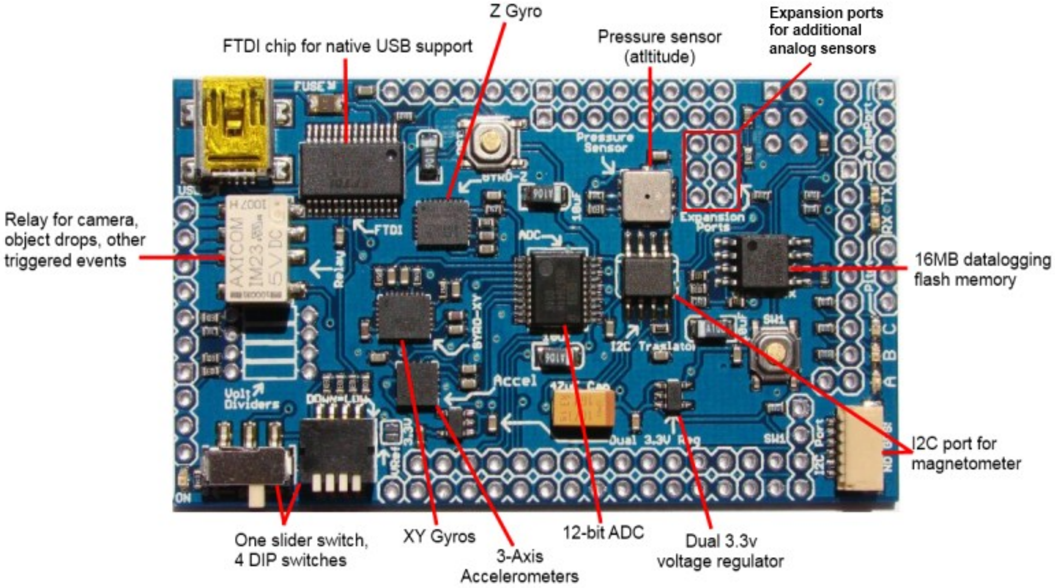


image srcs: diydrones.com

Setup

- Software tasks:
  - Fast task 1: read the pilot input, adjust attitude, signal servos
  - Fast task 2: read SPI-bus devices: gyro, barometer
  - Slow task: read GPS data, read navigation data from the ground station radio, update altitude, throttle

  New functionality:
  - Program-data task: response to SWATT attestation
  - Recovery task: disable ground-control station, disable attitude/position change
  - CFI monitor: monitoring the slow task
- Attack: latent buffer overflow in the slow task
- CFI monitor runs at 20Hz (16 MHz processor)

Summary

- TrackOS is real-time and returns no false-positives (assuming conservative static analysis)
- TrackOS provides unbypassable detection of malicious control-flow modifications (assuming sufficient frequency)
- TrackOS scales to large current programs
- TrackOS requires no access to the source code of the analyzed program and is compiler-independent

Questions?

leepike@galois.com