FiE on Firmware

Finding Vulnerabilities in Embedded Systems using Symbolic Execution

Drew Davidson
Ben Moench
Somesh Jha
Thomas Ristenpart

FiE in a Nutshell

- Symbolic execution tailored to embedded firmware
  - Detects common firmware vulnerabilities
  - Deals with domain-specific challenges
  - Able to verify small programs
- Tested on 99 programs
  - Found 22 bugs
  - Verified memory safety for 52 programs

Example Attack: WOOT 2012

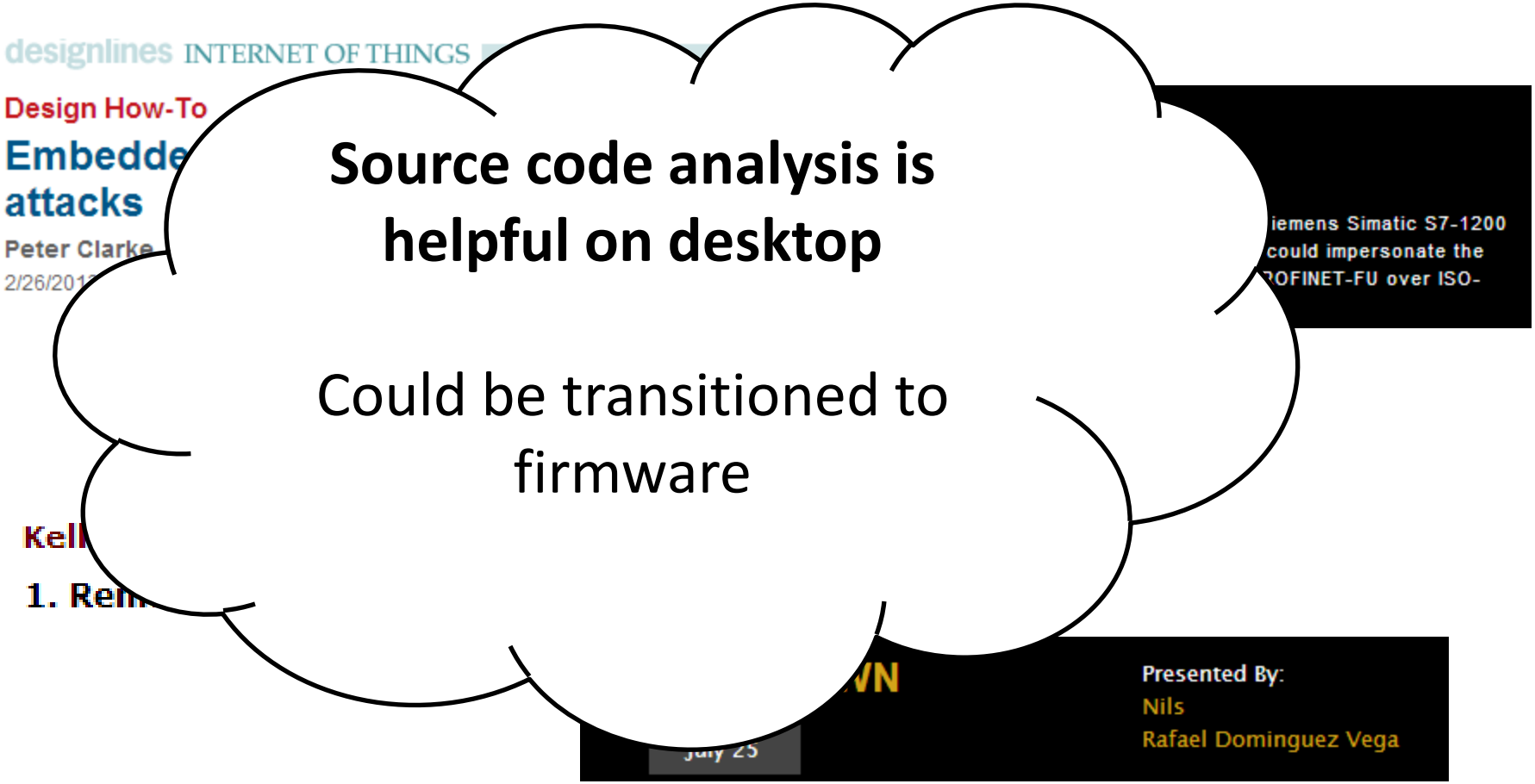
16-bit low power device
C firmware
Low-level hardware interaction

Buffer Overflow!

Encrypted card data

[Frisby et al., 2012]

Embedded Systems: Lots of Attacks

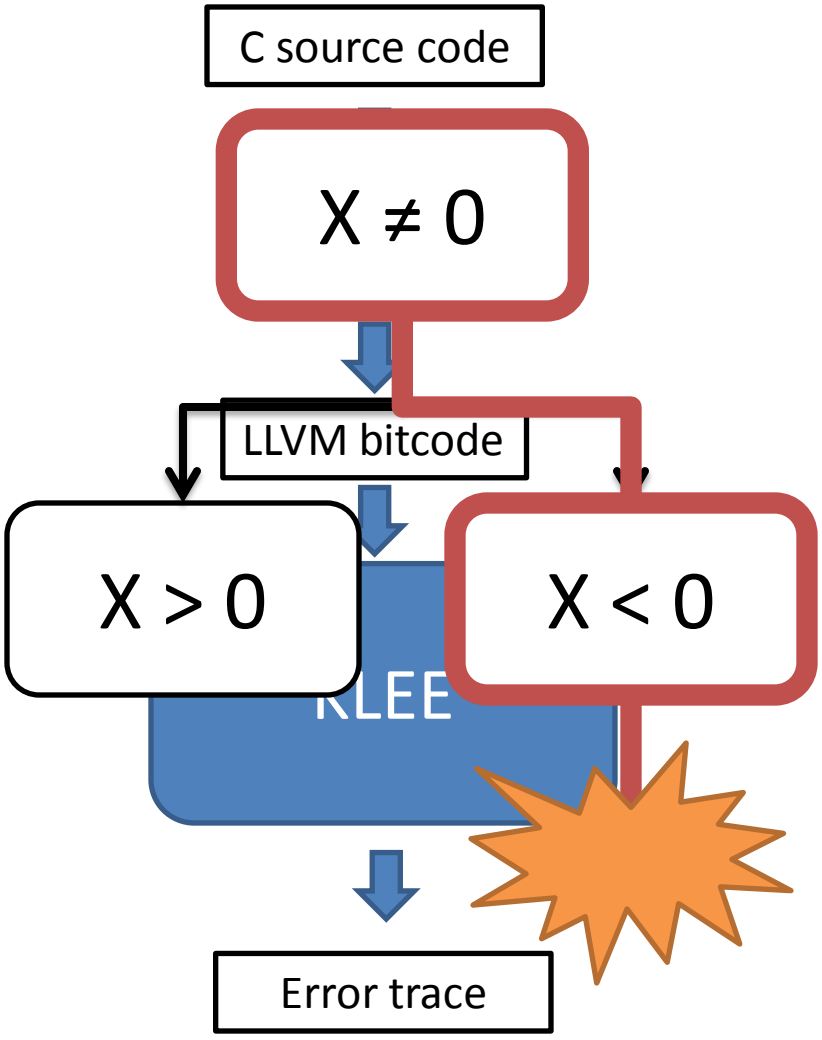


Source code analysis is helpful on desktop

Could be transitioned to firmware

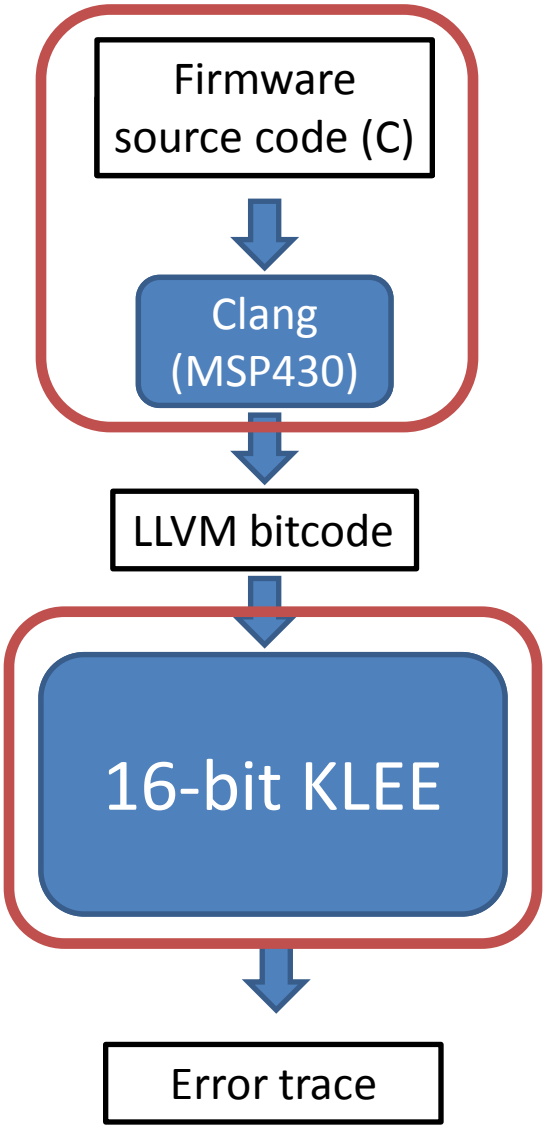
… Little Work on Detecting Vulnerabilities

Symbolic Execution



- Represents program input as sets of constraints
- Explores multiple feasible paths for bugs
- Provide detailed trace to vulnerability
- KLEE
  - Popular, mature tool
  - Average > 90% line coverage
  - Finds memory safety violations

KLEE: Performance on MSP430



- Why MSP430?
  - Popular, widely deployed
  - Security applications
  - Has clang support
- KLEE ported to 16-bit
- Evaluated 99 programs
  - 12 TI Community
  - 78 Github
  - 8 USB protocol stack
  - 1 Synthetic (cardreader)
- Average instruction coverage for MSP430 < 6%
  - Most programs < 1%

Challenges of MSP430 Code

- Peripheral access with I/O Ports
- Environment interaction via implicit memory mapping

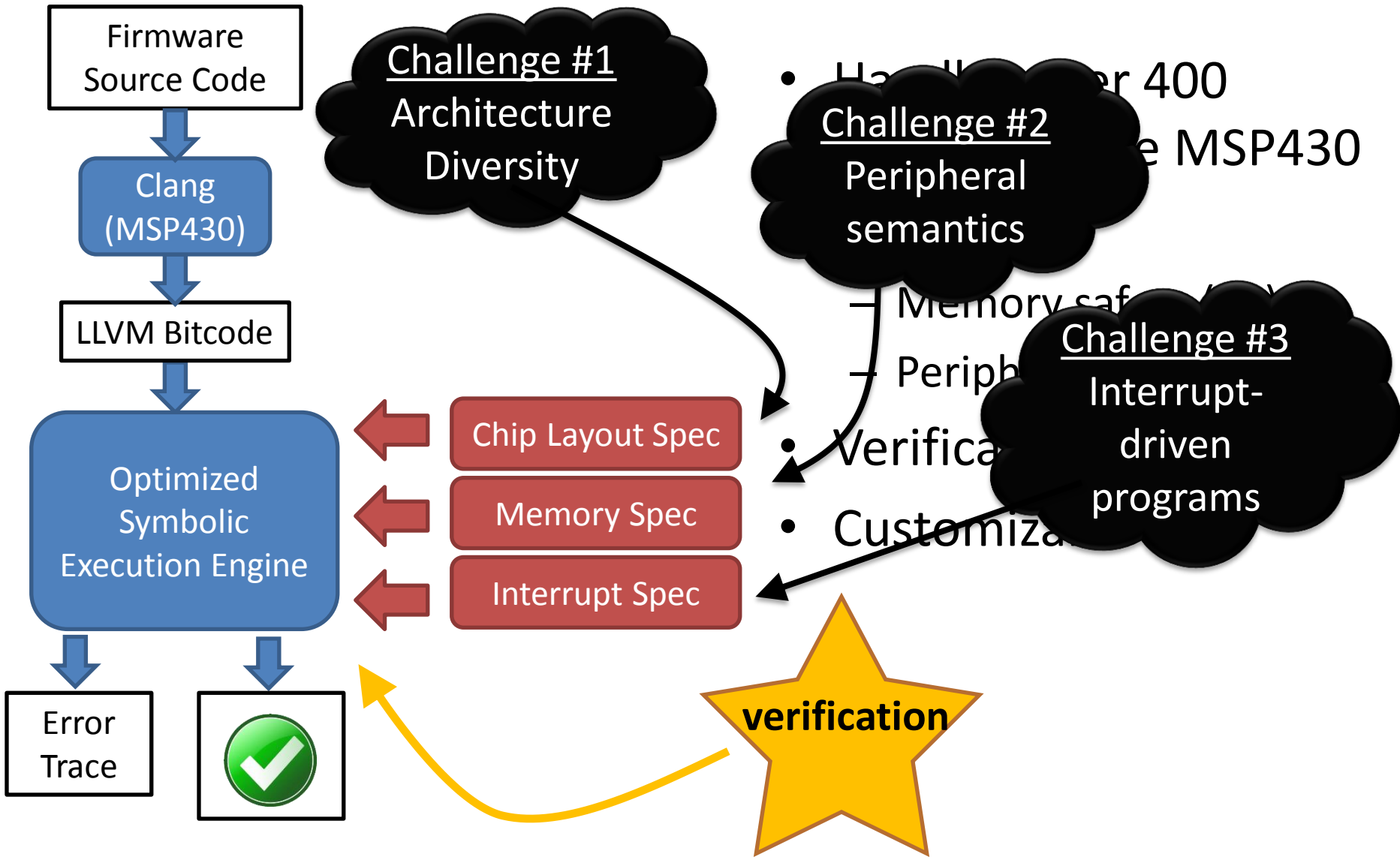
```
while (true){
   if (*0x21)
      len = *0x21;
   _BIS_SR(GIE);
   if (!*0x21)
      strncpy(dst,src,len);
}

   PORT_2_ISR
P1IFG = 0x0;
```

Challenge #1
Architecture Diversity

> 400 variants of MSP430

Challenge #2
Peripheral semantics

FiE on Firmware



FiE on Memory

Assume adversary controls peripherals
Allow users to supply custom libraries

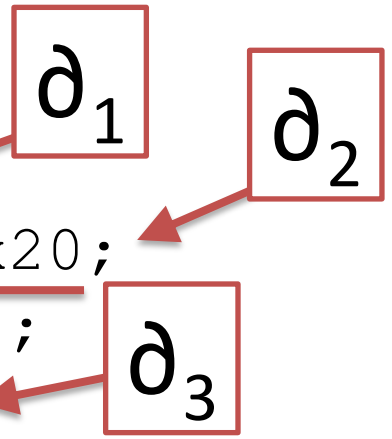
Chip Layout Table

```
addr P1IN 0x20 1
```

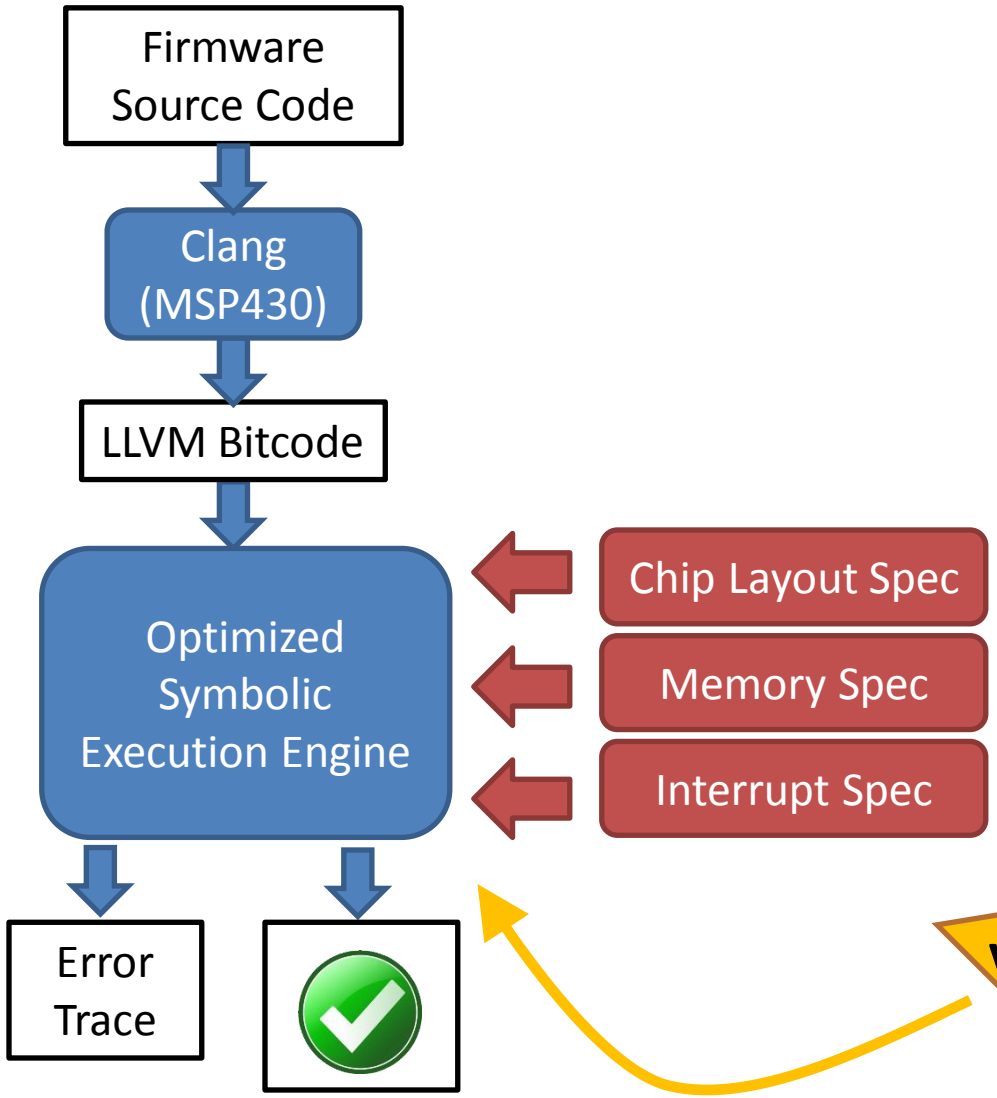
Memory Library

```
P1IN_READ:
fresh_symbolic()
```

```
while (true){
   if (*0x20)
      len = *0x20;
   _BIS_SR(GIE);
   if (!*0x20)
      strncpy(dst,src,len);
}

   PORT_2_ISR
*0x22 = 0x0;
```

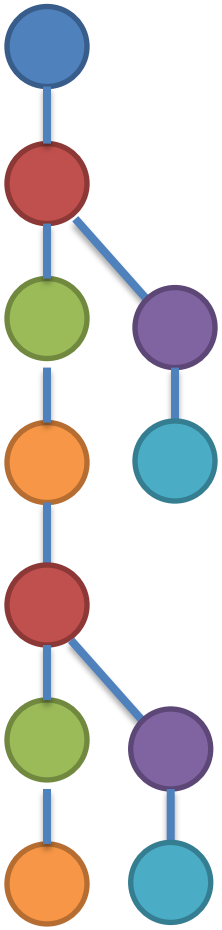


Challenges and Opportunities



- Verification
  - Modular architecture
    - Outside scope of traditional symbolic execution
      - State space intractable
- Key Insight
  - Firmware state space much smaller

FiE on Verification



Infinite program paths

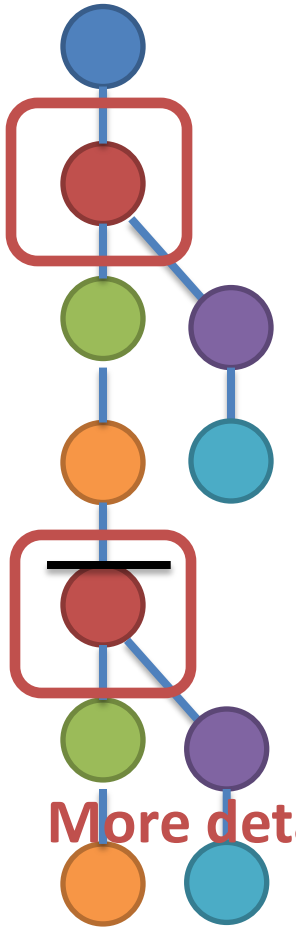
Analysis stuck executing already-seen states

Prevents verification

```
while (true){
   if (*0x20)
      len = *0x20;
   _BIS_SR(GIE);
   if (!*0x20)
      strncpy(dst,src,len);
}
```

```
   PORT_2_ISR
*0x22 = 0x0;
```

FiE on Verification



More details in the paper

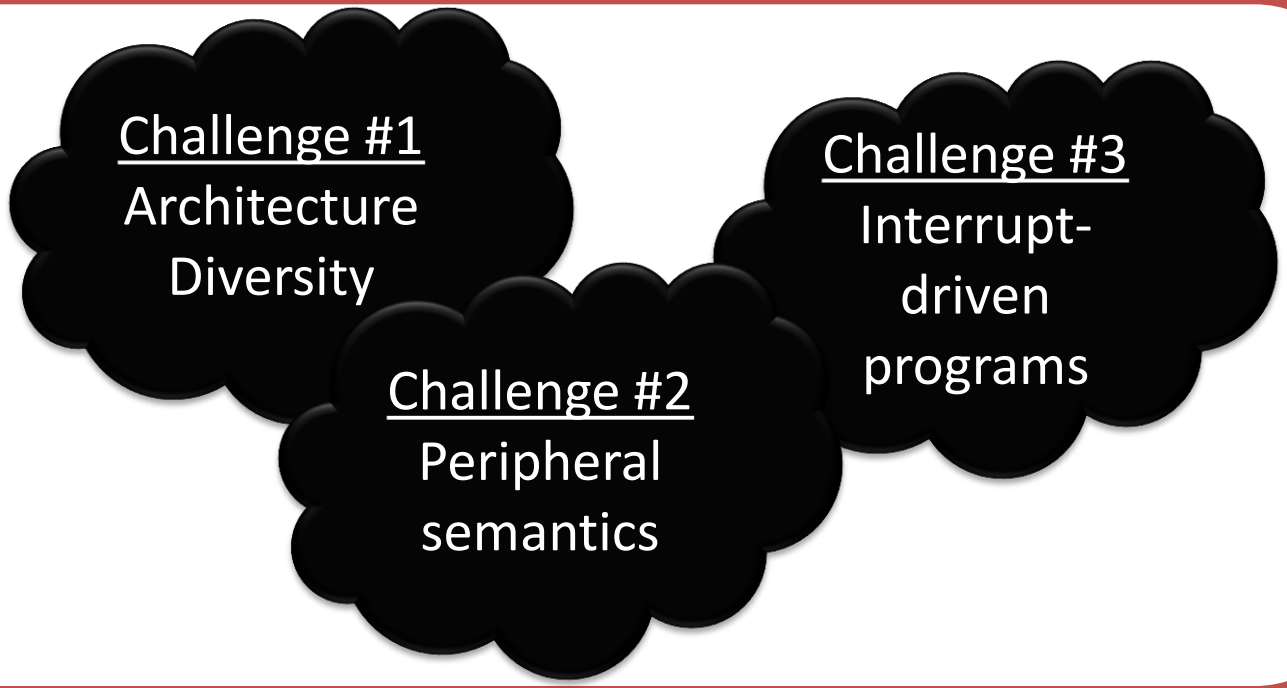
- Log all execution states
- Pruning
  - Detect redundant states and terminate them
  - Redundant states; redundant successors
- Smudging
  - replace frequently-changing concrete memory with symbolic
  - Complete
    - May have FPs

FiE on Firmware

Chip Layout Spec

Memory Spec

Interrupt Spec



Optimized
Symbolic
Execution Engine

Evaluation

- Amazon EC2
  - Automated tests (scripts available)
  - 50 minute runs
- Test Versions:
  - 16-bit KLEE
    - baseline
  - FiE
    - Symbolic + plugin
  - FiE + pruning
  - FiE + pruning + smudging

Corpus:

12 TI Community
1 Synthetic (cardreader)
8 USB protocol stack
78 Github

Bugfinding Results

- 22 bugs across the corpus (smudge)
  - Verified manually
  - 21 found in the MSP430 USB protocol stack
  - 1 misuse of flash memory
- Emailed developers

Coverage Results

Mode	Average % Coverage	False Positives	Verified
Baseline	5.9	92	0
Symbolic	71.1	0	7
Prune	74.4	0	35
Smudge	79.4	1	53

Thanks!

Summary

Initiated work for MSP430 automated bugfinding

Modular, conservative symbolic execution

Supported verification and bugfinding

Download FiE

www.cs.wisc.edu/~davidson/fie