Lecture Notes: FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution

Rui Zhang

March 6, 2017

1 Embedded microprocessors, Firmware

Typical low-power embedded systems combine a software-driven microprocessor, together with peripherals such as sensors, controllers, etc.

Firmware: The software on such devices is referred to as firmware, and it is most often written in C.

2 Motivation

To improve firmware security:

1. Source-code analysis tools: insufficient for analyzing firmware
   (a) e.g. the microcontrollers used in practice have a wide range of architectures
   (b) firmware exhibits characteristics dissimilar to traditional programs: frequent interrupt-driven control flow and continuous interaction with peripherals
2. Fuzzing and reverse engineering: requires significant manual effort and knowledge of the firmware under analysis

3 Background

3.1 MSP430 Microcontrollers

1. used in security critical applications such as credit-card point of sale systems, smoke detectors, motion detectors, seismic sensors, etc.
2. RISC instruction set, von Neumann architecture, 16-bit addressing
3. Built-in peripherals: flash memory, timers, power management interfaces, etc.
4. External peripherals: USB hardware, modems, sensors, etc.

5. Operates by setting up configuration for the program, spinning in an infinite loop while waiting for input from the environment

6. These event-driven programs use interrupt handlers, busy waiting, and the like to drive computation in response to I/O from peripherals. Interrupt handlers often contain the bulk of the program logic.

7. A typical firmware will initialize several registers specifying which interrupts to activate and then go to sleep either by setting the chip to a low-power sleep mode or by entering an explicit infinite loop.

3.2 Firmware Corpus

1. Cardreader: gets as input card data from the stripe reader, loads a stored cryptographic key from flash memory, applies AES encryption to the card data before writing the result to the UART.

2. USB drivers: CDC Driver and HID Driver.

3. Community projects and GitHub: 83 firmware programs

4. Contiki: provides an operating system for microcontrollers.

3.3 Symbolic Execution Challenges

1. Architecture ambiguity: Firmware programs make a number of assumptions about the hardware: the overall layout of memory and location of memory-mapped hardware controls. These assumptions are not made explicit in a firmware's source code.

2. Intensive I/O: Need to support analysis without peripherals and often without knowing the intended peripheral. Need to support the detection of peripheral misuse bugs, when a peripheral and its behavior are known.

3. Event-driven programming: Deep or infinite loops are frequent, and most programs logic happens in interrupt handlers. Failure to follow possible control flow paths through interrupt handlers will result in very low coverage results.

4 Overview of FIE

1. Threat model: all inputs from peripherals are untrusted.

2. Target: to achieve complete analyses for simple firmware programs.

3. FIE frontend: Three necessary features that CLANG alone does not provide:

   (a) definitions for compiler intrinsics that are not expanded by CLANG

   (b) definitions of standard library functions that would normally be included at linke time

   (c) definitions of hardware-defined behavior

Handling (1) and (2): provide a wrapper around CLANG which links pre-compiled bitcode for functions in stdlib and for compiler intrinsics. For (3): it is often unknown at compile time, address this at runtime using an analysis specification

4. Core execution engine:
    (a) Memory spec: supplies the semantics of special memory such as attached devices, flash memory. FIE comes with a set of default specifications, conservatively returns unconstrained symbolic values to any read and ignores write.
    (b) Interrupt spec: informs the analysis of when and which interrupts should be simulated to have fired at any given point in symbolic execution.
    (c) Runs a modified version of KLEE:
        i. major changes: a new memory manager to ensure that all memory objects are allocate within a 16-bit value, the use of memory spec and library.
        ii. enhancements: state pruning, memory smudging

5 Details of FIE's Architecture

5.1 Main Execution Loop

1. Execution state: an immutable snapshot of the symbolic execution at a given point in time. Includes: all values used to emulate LLVM bitcode, a program counter, stack frames, global memory.
2. Pseudocode

5.2 Modeling Chips and Peripherals

1. Analysis specification
    (a) A plaintext file adhering to a simple format and specifying how the analysis should be configured.
    (b) The layout file can be synthesized automatically from firmware source code and files included in the compiler.
    (c) The chip layout specification explicitly fixes architecture details that are implicit in firmware, but it does not specify the actual behavior of these special features.
2. Memory spec
    (a) FIE uses a library of functions that form a model of special memory behavior.
    (b) Read and write functions are passed the entire symbolic execution state, and output a set of states.
    (c) FIE provide a default memory spec which is automatically generated from the analysis spec.

3. Interrupt spec

    (a) Determining the enabled set of interrupts requires knowledge of the architecture and the current firmware state.

        i. the MSP430 design documents specify a partial order of priorities over interrupts

        ii. some interrupts are only enabled when appropriate status register flags are set

    (b) The interrupt spec contains a number of gate functions, one for each possible interrupt that can occur on an MSP430.

5.3 State Pruning

1. Redundant states: if one state will execute equivalently to another already seen state, we call this state redundant.
2. PLAIN: if all successors generated via interrupt spawning or evaluation are simply added to the set of active states, we refer to this variant as the PLAIN operating mode of FIE.
3. Redundant states arise frequently and PLAIN is slowed down considerably by them.
4. One source of redundant state: interrupt firings can lead to two different paths leading to the same state.
5. A second source of redundant states: symbolic execution of loops generates redundant states.
6. Prior systems deals with redundant states by state selection heuristics that favored new lines of code.
7. FIE detect and prune redundant states.

5.4 Memory Smudging

1. At analysis time, the analyst supplies a modification threshold t to FIE.
2. Before adding a successor state to the set, the function MemorySmudge checks if any memory locations have been modified t times.
3. If so, the location's value is replaced by a special value *.
4. This wildcard value may take any value allowed by the type and cannot be constrained.

6 Evaluation

1. Firmware size and coverage

    (a) Firmware size: the number of executable LLVM instructions (NEXI)

        i. compile the firmware into LLVM bitcode using CLANG

        ii. run LLVM optimization passes
        iii. take the number of LLVM instructions in the resulting bitcode
    (b) Code coverage: the fraction of LLVM instructions executed in the course of the analysis divided by the NEXI of the target firmware

2. Coverage under different FIE modes

    (a) Baseline: for most firmware, the Baseline analysis performs very poorly, with a median of 1.7% coverage.
    (b) Fuzz: this mode takes advantage of knowledge of the memory layout, special registers, and interrupt handling semantics. Fuzzing provides surprisingly good coverage for many of the firmware programs, in fact beating symbolic execution modes in many cases.
    (c) Plain, Prune, and Smudge: Smudge provides better coverage than all others, including Fuzz.

3. 50-minute analysis outcomes

    (a) FIE can either stop because it runs out of memory, the requested amount of execution time has been reached, or there exist no more active states.
    (b) Pruning and smudging help reduce memory usage and increase the number of analyses that finish.

7 Limitations

1. There exist firmware for which complete analyses are intractable, and soundness is only with respect to the symbolic execution framework.

2. There exist various sources of imprecision in analysis that may lead to false positives or false negatives.

    (a) Bugs in the analysis software or misconfiguration.
    (b) Discrepancies between the firmware as symbolically executed in FIE and natively in deployment.
    (c) The most conservative analysis models peripherals and interrupt firing as adversarially controlled.

3. FIE fails execution paths that include inline assembly.