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

Drew Davidson Benjamin Moench Somesh Jha Thomas Ristenpart
University of Wisconsin–Madison, {davidson,bsmoench,jha,rist}@cs.wisc.edu

Abstract

Embedded systems increasingly use software-driven low-power microprocessors for security-critical settings, surfacing a need for tools that can audit the security of the software (often called firmware) running on such devices. Despite the fact that firmware programs are often written in C, existing source-code analysis tools do not work well for this setting because of the specific architectural features of low-power platforms. We therefore design and implement a new tool, called FIE, that builds off the KLEE symbolic execution engine in order to provide an extensible platform for detecting bugs in firmware programs for the popular MSP430 family of microcontrollers. FIE incorporates new techniques for symbolic execution that enable it to verify security properties of the simple firmwares often found in practice. We demonstrate FIE's utility by applying it to a corpus of 99 open-source firmware programs that altogether use 13 different models of the MSP430. We are able to verify memory safety for the majority of programs in this corpus and elsewhere discover 21 bugs.

1 Introduction

Embedded microprocessors are already ubiquitous, providing programmatic control over critical, increasingly Internet-connected physical infrastructure in consumer devices, automobiles, payment systems, and more. Typical low-power embedded systems combine a software-driven microprocessor, together with peripherals such as sensors, controllers, etc. The software on such devices is referred to as *firmware*, and it is most often written in C.

The use of firmware exposes embedded systems to the threat of software vulnerabilities, and researchers have recently discovered exploitable vulnerabilities in a wide variety of deployed embedded firmware programs [12, 18, 19, 21, 22, 24, 27]. These bugs were found using a combination of customized fuzz testing and manual reverse engineering, requiring large time investments by those with rare expertise.

To improve firmware security, one possible approach would be to use the kinds of source-code analysis tools that have been successful in more traditional desktop and server settings (e.g., [2, 4, 8, 9, 11, 13, 17, 26, 28, 31, 36]). These tools, however, prove insufficient for analyzing firmware: the microcontrollers used in practice have a wide range of architectures, the nuances of which frustrate tools designed with other architectures in mind (most often x86). Firmware also exhibits characteristics dissimilar to more traditional desktop and server programs, such as frequent interrupt-driven control flow and continuous interaction with peripherals. All this suggests the need to develop new analysis tools for this setting.

We initiate work in this space by building a system, called FIE, that uses symbolic execution to audit the security of firmware programs for the popular MSP430 family of 16-bit microcontrollers. We have used FIE to analyze 99 open-source firmware programs written in C and of varying code complexity. To do so, FIE had to support 13 different versions of the MSP430 family of 16-bit RISC processors. Our analyses ultimately found 20 distinct memory-safety bugs and one peripheral-misuse bug.

We designed FIE to support analysis of all potential execution paths of a firmware. This means that, modulo standard but important caveats (see Section 6), FIE can verify security properties hold for the relatively simple firmware programs often seen in practice. For example, we verify memory safety for 53 of the 99 firmware programs in our corpus.

Overview of approach: FIE is based on the KLEE symbolic execution framework [10]. In addition to the engineering efforts required to make KLEE work at all for MSP430 firmware programs, we architected FIE to include various features that render it effective for this new domain. First, we develop a modular way to specify the memory layout of the targeted MSP430 variant, the way in which special memory locations related to peripherals should be handled, and when interrupt handlers should be invoked. This all allows analysts to flexibly detail peripheral behavior. We provide a default specification that models worst-case behavior of all peripherals and interrupts. This default enables analysis without any knowledge or access to (simulators of) individual microcontrollers or peripheral components, while ensuring consideration of any possible deployment environment.

Small firmware programs appear to arise frequently (our corpus has many that have less than 100 lines of code) and for these we might hope to achieve complete analyses, meaning all possible paths are checked. Even with very small firmware programs, however, deep or infinite loops arise often and force the analysis to visit already-analyzed states of the symbolic execution. We therefore use a technique called *state pruning* [6], which detects when a program state has been previously analyzed, and if so, removes it from further consideration. Our realization of pruning keeps a history of all changes made to memory at each program point, and while simpler than prior approaches (see Section 7) it proves effective. We also introduce a new technique called *memory smudging*, which heuristically identifies loop counters and replaces them with unconstrained symbolic variables. While smudging can introduce false positives, our experiments show them to be rare. Together, pruning and smudging significantly improve code coverage and support the ability to analyze all possible paths of simpler firmware programs.

Summary: This paper has the following contributions:

- We provide (to the best of our knowledge) the first open-source tool designed for automated security analysis of firmware for the widely used MSP430 microcontrollers.
- We explore use of state pruning and memory smudging to enhance coverage of symbolic execution and to attempt to verify the absence of classes of bugs. Ultimately, FIE is able to verify memory safety on 53 open-source firmware programs.
- FIE found 21 distinct bugs in the firmware corpus, many of which appear to be exploitable memory-safety violations.

To do these analyses at scale, we developed a system for managing FIE-powered analyses on Amazon EC2 [1]. The source code for FIE, the firmware corpus, and the EC2 virtual machine images and associated management scripts will all be made publicly available from the first author's website.[1]

Outline: The remainder of this paper is structured as follows: In Section 2, we give background on embedded systems and the MSP430 family, describe a corpus of open-source firmware that we gathered, and explain some of the key challenges that must be overcome for use of symbolic execution in our context. We then give a high-level overview of how FIE works in Section 3, and explain its mechanisms in greater detail in Section 4. We evaluate FIE on the corpus of firmware examples and discuss the vulnerabilities found in Section 5. Finally we discuss limitations of FIE in Section 6, related work further in Section 7 and conclude in Section 8.

2 Background and Analysis Targets

Our system, FIE, analyzes embedded firmware programs for the MSP430 family of microcontrollers using *symbolic execution* [2, 8–11, 13, 17, 28, 31, 36]. In this section, we describe details of the MSP430 family, discuss a representative corpus of firmware programs that we gathered, review symbolic analysis, and explore the challenges faced in attempting to use existing tools for analysis of firmware programs.

2.1 MSP430 Microcontrollers

We chose Texas Instruments' (TI's) MSP430 family of microcontrollers as our analysis target because of its popularity. MSP430s already find use in security critical applications such as credit-card point of sale systems, smoke detectors, motion detectors, seismic sensors, and more [34]. We believe porting our approach to other, similar low-power microprocessor families would be straightforward.

Architecture and memory layouts: MSP430s use a custom, but simple, RISC instruction set, and have a von Neumann architecture (instructions and data share the same address space) with at least 16-bit addressing. MSP430s have a set of CPU registers, which are accessed via special memory locations. There are over 500 different MSP430 microcontroller products. One example is the MSP430G2*x*53 series, which consists of 5 different chips. These have from 1 kB to 16 kB of non-volatile flash memory and from 256 to 512 bytes of volatile random access memory. The memory layouts for the different models are distinct, meaning some physical addresses are invalid on one variant while valid on another.

Hardware peripherals: MSP430 microcontrollers are used in conjunction with both built-in and external hardware peripherals. Built-in peripherals include flash memory, timers, power management interfaces and the like, whereas external peripherals (USB hardware, modems, sensors, etc.) must be connected to the microcontroller via I/O pins. MSP430s have a limited number of I/O pins, and so they are multiplexed amongst various functions. Usually, one function is general purpose I/O and the other is an internal function. For applications that need to use many different functions of the device, a given pin may be switched between its multiplexed duties several times during execution. Accessing periph-

[1]`http://pages.cs.wisc.edu/davidson/fie`

```
1   FCTL3 = FWKEY;
2   FCTL1 = FWKEY + ERASE;
3   *F_ptr = 0;
4   while (FCTL3 & BUSY);
5   FCTL1 = FWKEY;
6   FCTL3 = FWKEY + LOCK;
```

Figure 1: Code excerpt that clears a flash segment

erals works via memory-mapped I/O or special registers (which are, in turn, accessed via special memory locations). We refer to all memory that serves internal or external peripherals as *special memory*.

Peripherals often have intricate semantics. For example, consider accessing flash memory, which is a built-in peripheral for nearly all MSP430 models. Figure 1 gives a code snippet taken from the USB drivers in our corpus (see next section). This code clears a segment of flash memory using various special registers, which is required before any writes to that segment can occur: flash control register 3 (`FCTL3`) must be set to the special flash write key (`FWKEY`) to unlock the memory and allow writes to it, and flash control register 1 (`FCTL1`) must contain the `FWKEY` value masked with the particular value to indicate the type of write. Finally, after the memory is erased, the flash memory is re-locked by assigning the special value `FWKEY + LOCK`.

Firmware programming: Most MSP430 programs are written in C, using one of three compilers recommended by TI: IAR, CCS, and msp430-gcc. The first two compilers are commercial products packaged in IDEs, while the third is a port of the gcc toolchain to the MSP430. Each of these tools provides a number of extensions to C. Unfortunately, the extensions do not agree on a single syntax. As a result, many programs conditionally include code based on the compiler that is being used. we chose to base our tool on the msp430-gcc syntax as it is popular, open-source, and has straightforward extensions.

Embedded firmware usually operates by setting up configuration for the program and then spinning in an infinite loop while waiting for input from the environment. These event-driven programs use interrupt handlers, busy waiting (e.g., line 4 in Figure 1), and the like to drive computation in response to I/O from peripherals, and so interrupt handlers often contain the bulk of the program logic. 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.

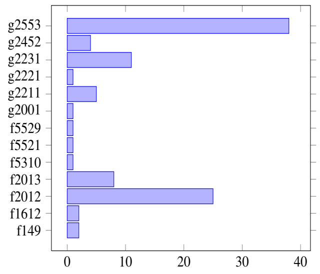
2.2 Firmware Corpus

As mentioned, MSP430s are used in a wide variety of security-critical applications. The diversity of applications is reflected in the firmware programs found in practice, ranging from simple programs for controlling some external hardware peripheral on up to feature-rich lightweight operating systems such as Contiki [35]. To have a concrete set of analysis targets and as well educate our design of FIE, we have gathered a corpus of 99 open-source MSP430 firmware programs, which we now discuss further.

Cardreader: The first firmware in our corpus is cardreader, a secure credit card reader designed for the MSP430g2553 and written by one of the authors independently of the development of FIE. This was motivated by recent attacks against smartphone-based point-of-sale MSP430 devices [20]. Our firmware assumes the presence of a magnetic credit card stripe reader attached to port 1 (configured for general purpose I/O), and a UART connection on port 2 (to transmit gathered credit card data). The cardreader gets as input card data from the stripe reader, loads a stored cryptographic key from flash memory, and applies AES encryption to the card data before writing the result to the UART. cardreader is fully functional, with 1,883 lines of C code as computed by the cloc utility, and incorporates many of the MSP430 programming constructs that, as we will see, can thwart traditional symbolic-execution-based analysis. We made no efforts to tailor the code to be amenable to analysis by FIE. We also performed extensive manual audit of the code to verify the absence of memory or peripheral-misuse errors.

USB drivers: We additionally use two USB driver firmware programs, CDC Driver and HID Driver, taken from the TI-supplied USB developers package. These programs include a full USB code stack, and include 7,453 and 7,448 lines of C code, respectively. The particular programs we chose exercise the CDC (Communications Device Class) and HID (Human Interface Device) USB classes, which represent different device types in the USB specification. The CDC-using firmware, for example, takes string commands from an attached terminal program on a host PC, uses these commands to toggle the LED in various ways, and sends back an acknowledgment string to the host device. The code that we tested was written for the IAR compiler, but we manually wrote Makefiles to compile the source code for our analysis.

Community projects and GitHub: In order to increase the size of our corpus, we searched for open source projects both on the TI MSP430 Community Projects website [33] and GitHub. For the former, we manually crawled the website, and downloaded all projects with a Makefile, of which we found 12 that compiled properly. For the latter, we used the GitHub API to automatically download all projects that matched the keyword "msp430". There were 360 such projects. Of these, we culled out those that either: did not include makefiles,



LOC	#
1 – 50	36
51 – 100	13
101 – 200	28
201 – 500	9
501 – 1000	7
1001 – 2000	2
$\geq$ 2001	4

Table 2: Number of firmware program in the corpus (left) targeting the indicated MSP430 models and (right) having the number of lines of C code in the given range.

were not written in C, or did not compile properly for the MSP430 using their given makefiles (this includes projects such as desktop utilities for connection to an MSP430, and thus matched the keyword without being applicable to FIE). After this culling we had 83 firmware programs.

Contiki: Finally, we add to the corpus Contiki [35], which provides an operating system for microcontrollers. To use Contiki, one writes an application against it as a library, which is then statically linked for a complete firmware. Since we need an entrypoint to the library for testing, we use a "hello world" example program included with the Contiki distribution. The resulting C file for the firmware has only 10 lines of code, but this links against other, larger modules. There are over 200,000 lines of C code in the full Contiki source tree. We note that Contiki supports many architectures, including (amongst others) motes that support MSP430x, an extension of the MSP430 that supports 20-bit addresses. FIE only supports basic 16-bit MSP430, and thus cannot run on these motes. Fortunately, Contiki has support for a basic MSP430 backend: the `esb`, based on the `msp430f1612`. We use this backend in our analysis.

The table in Figure 2 shows a break down of the number of firmware programs whose number of lines of code (computed using cloc, including C and C/C++ header files) falls in the indicated range. As can be seen, the range of sizes of these firmware programs is large, but most are 2,000 lines of C code or less. This is not surprising given that MSP430s are often used to drive relatively simple controllers or sensors: our firmware set includes the large number of small hobbyist projects found on GitHub and the TI community projects webpage. A breakdown of the architectures targeted by firmware programs in the corpus is shown in the graph in Figure 2. (When a single firmware supports multiple target architectures, we restrict attention to one, picked arbitrarily.)

2.3 Symbolic Execution and Challenges

To date, finding vulnerabilities in embedded firmware programs has relied upon specialized fuzzing and reverse engineering [12, 18, 20–22, 24], which requires significant manual effort and knowledge of the firmware under analysis. Almost all previous research on more general software analysis tools (see Section 7) has not focused on the setting of embedded microcontrollers, and so the relatively unique features of our context (relatively small firmware sizes, large diversity of architectures, and complex environmental interactions) mean that traditional approaches need to be revisited. We initiate such work, focusing in particular on symbolic execution. We feel it to be well-suited to firmware analysis, allowing fine-grained modeling of architectural nuances, flexibility in analysis approach, and typical limitations of symbolic analysis (i.e., scalability) may not prove to be as much of an issue for the small firmware programs seen in practice. We leave exploration of other approaches (e.g., static analysis, concolic execution, etc.) to future work.

Symbolic execution: In symbolic execution, variables corresponding to inputs to a program are treated as symbolic. This means one stores a representation of all of the possible values that each symbolic variable may take on. The program is then executed symbolically using an execution engine. A *symbolic state* (just state from now on) is a current program counter, other register contents, stack frames, and memory contents. The latter three may contain a mix of concrete values or symbolic variables and the constraints over those variables. From an initial state, the engine executes the program one instruction at a time and updates the state appropriately, changing concrete values or possibly adding constraints upon symbolic variables.

Should execution reach a control flow decision such as a branch, the executor uses a SAT solver to determine what are the possible next instructions. A new state is generated for each possible next instruction, with appropriate constraints for the outcome. For example, if a variable `x` is assigned symbolic variable α (that is unconstrained), and a branch `if (x < 5)` is encountered, two child states will be spawned: the first executes inside the if condition with the constraint $\alpha < 5$, and the second executes after the if condition with the constraint $\alpha \geq 5$. Once multiple states are active, the engine decides during each iteration which state to progress, based on some state selection heuristic.

(Complete) analyses: Analysis is performed by investigating each state for violations of some specific properties. A common choice is memory-safety violations, which can be checked by ensuring that all reads and writes are to properly allocated memory ranges. Should a state violate such a property, the execution halts and outputs one of the paths that could lead to this state as well as concrete values that drive the program's execution along that path. The latter is facilitated by using the SAT solver to provide a solution for the formulas describing constraints on the symbolic variables.

It is well known that symbolic execution can, in theory, provide both sound and complete analyses of some programs. A sound analysis does not emit any false positives — bug reports that are spurious. We refer to a symbolic-execution-based analysis as being *complete* if it covers all of the finitely many possible symbolic states.

Obviously complete analyses are intractable for many programs. Past work on symbolic execution has therefore focused on achieving high *code coverage*, meaning the number of executable lines of code in a program that have been symbolically executed along any path. High-coverage symbolic execution enables finding bugs along the paths that are explored. One can also use the explored paths to generate inputs for use in testing. In our setting of resource-constrained, small firmware programs, there is hope that in addition to high code coverage, we may be able to sometimes achieve complete analyses as well.

Symbolic analysis (even when sound and complete) has inherent limitations, stemming from the possibility of bugs in the analysis engine or compilers used, source code that depends on memory address values, use of inline assembly, etc. We discuss these limitations more in Section 6.

Challenges: We use the symbolic execution system KLEE [10] as the foundation for FIE. Our problem domain, however, necessitates rethinking several aspects of KLEE's design and use. In particular, we face the following three key challenges:

Challenge 1 (architecture ambiguity): Firmware programs make a number of assumptions about the hardware, including the overall layout of memory and location of memory-mapped hardware controls. These assumptions are not made explicit in a firmware's source code. For example, it is common for a program to store persistent configuration data at a hard-coded memory address in flash. An architecture-agnostic analysis, or one tailored to x86 environments (as most prior tools are), would view code using this feature as having read from uninitialized memory. Making matters worse, the wide diversity of architectures mean that we will need a way to configure an analysis to the architectures of interest.

Challenge 2 (intensive I/O): Firmware programs are highly interactive with the environment throughout the lifecycle of the program and are designed to interact with a huge diversity of peripherals. Handling external inputs to a program is a well-studied issue in prior symbolic execution contexts, for example KLEE implements functions to determine for the symbolic executor the outcome of (a subset of) common Linux operating system calls. In our setting, the peripheral interface is via special registers and memory-mapped I/O and there exists a huge diversity of potential peripheral behaviors. This makes our setting closer to the one targeted by SymDrive, which uses the S2E [14] symbolic execution system to analyze x86 Linux kernel drivers without hardware [26]. Like SymDrive, we need to support analysis without (simulators of) peripherals and often without even knowing the intended peripheral. When a peripheral and its behavior are known, we should support the detection of (what we call) peripheral misuse bugs, in which a firmware incorrectly implements the (sometimes complex) operations involving some peripheral.

Challenge 3 (event-driven programming): The event-driven model of programming used for firmware is problematic for symbolic execution because deep or infinite loops are frequent, and most program logic happens in, or as a direct result of, interrupt handlers. S2E and SymDrive both dealt with the frequent use of loops in code, via path selection heuristics or loop elision. These approaches do not allow complete analyses, which we hope to sometimes achieve in our setting. We note that since interrupts are so crucial to the operation of the program, failure to follow possible control flow paths through interrupt handlers will result in very low coverage results. Furthermore, disregarding the circumstances under which interrupts can occur may cause infeasible paths to be explored in the analysis. At the same time, the number of possible paths that can occur in the program due to interrupts causes state space explosion as, in the worst case, we must consider every instruction as a potential branch.

3 Overview of FIE

Our main contribution is FIE, an extensible tool for symbolic-execution-based analysis of MSP430 firmware. It is based on KLEE [10], but with significant modifications and embellishments to the frontend and core engine as we will explain. In this initial work we focus on analyzing memory safety of firmware programs, the lack of which has been exploited in many of the recent security exploits against embedded systems. We also report on detection of some peripheral misuse bugs. Our analyses, by default, use a conservative threat model in which all inputs from peripherals are untrusted.

In a departure from previous symbolic analysis systems, we target the ability to achieve complete analyses for simple firmware programs. These appear to be common and complete analysis in this context is particularly compelling since it means that (subject to various caveats discussed in Section 6) one can verify security or correctness properties of a firmware before deploying it. As far as we are aware, symbolic execution has not before been used for verification, since in most settings it is not feasible under current techniques.

This goal of completeness will guide our design in several ways, as previous optimizations (such as path selection heuristics) do not support this goal. That said, we will not always be able to provide verification, and when not, FIE will be useful in a more traditional role for bug-finding and test case generation. Here its efficacy will be measured by its ability to provide high-percentage code coverage.

In the remainder of this section, we walk through the workflow of FIE, shown in Figure 3.

FIE frontend: The first step to analyzing firmware is compiling it to a form that can be symbolically executed. The core of this process uses the CLANG compiler to LLVM bitcode. However, there are three necessary features of this process that CLANG alone does not provide: (1) definitions for compiler intrinsics that are not expanded by CLANG; (2) definitions of standard library (stdlib) functions that would normally be included at link time; and (3) definitions of hardware-defined behavior. Handling (1) and (2) is straightforward: we provide a wrapper around CLANG which links pre-compiled bitcode for functions in stdlib and for compiler intrinsics. We took the definitions for the stdlib functions from the `msp430-gcc` source code, and we manually wrote stubs for the compiler intrinsics. As a convenience to the user, the wrapper also embeds a token into the firmware bytecode to specify which MSP430 variant the firmware is compiled for, which simplifies the FIE command line. Providing definitions for hardware-defined behavior is more involved, since it is often unknown at compile time — for instance, the firmware may interact with a variety of peripheral devices to the chip with varying behaviors. We address this at runtime using an analysis specification, which is describe in depth in Section 4.2.

The most predominant compiler in use for our corpus is `msp430-gcc`, a version of `gcc` targeting the MSP430. Fortunately the arguments to CLANG are largely compatible with `msp430-gcc`, so we can drop in our CLANG wrapper in place of `msp430-gcc`, and use many of the original, unmodified Makefiles included in our corpus.

Core execution engine: Once firmware bitcode has been generated, FIE itself can be run. To analyze the LLVM firmware bytecode file `input.bc`, the user issues the command

```
fie -mmodel=<mem> -imodel=<intr> input.bc
```

The *memory spec* is specified by `mem`, which supplies the semantics of special memory such as attached devices, flash memory, etc. FIE comes with a set of default specifications which conservatively returns unconstrained symbolic values to any read from special memory and ignores writes. However, the user may wish to choose a different specification or write their own. We discuss this process in depth in Section 4.2.

The *interrupt spec* `intr` informs the analysis of when (and which) interrupts should be simulated to have fired at any given point in symbolic execution. Should an interrupt be deemed to fire, the state's execution is progressed to the appropriate interrupt handler function within the firmware. The interrupt spec allows us to flexibly model different interrupt firing behaviors. Our default is to allow any enabled interrupt to fire at every program point. See Section 4.2.

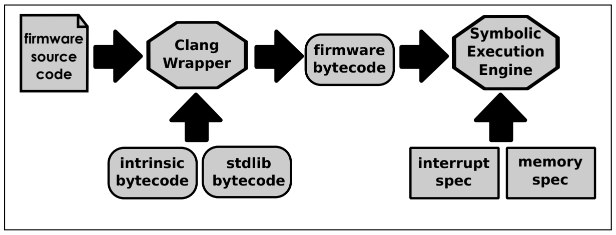


Figure 3: The FIE workflow and system components.

We inherit as well from KLEE various possible command-line options, so the user can optionally specify the wall-clock time to spend on the analysis, the search heuristic to use, etc.

FIE runs a modified version of the KLEE symbolic execution engine (the executor) to perform the analysis over the firmware bitcode. In particular, we use directly from KLEE their existing state selection heuristics, their underlying SAT solver framework, and much of their state management code. Our major changes include porting the entire execution engine to a 16-bit architecture, which includes a new memory manager to ensure that all memory objects are allocate within a 16-bit value, and the use of the memory spec and interrupt library to model execution when the engine interacts with special memory or fires an interrupt. We also implement two enhancements to the symbolic execution engine, state pruning (Section 4.3), first introduced by RWset [6] and adapted to our domain, and memory smudging (Section 4.4), which is novel to this work. These can improve code coverage and, for some programs, enable complete analyses.

FIE finishes when it completes an analysis by visiting every possible state, hits the requested time limit, or finds a memory-safety (or other) violation. In the latter case it outputs a description of a path leading to the bug. We call this a trace. The trace includes concrete examples of inputs (i.e., from peripherals) that cause the firmware to trigger the bug, and includes at what points in the execution interrupts fired to cause a jump to a specific interrupt handler. Currently, a trace is useful as a debugging log, but eventually it could be used to directly drive an MSP430 simulator to validate the potential bug.

We have additionally prepared an Amazon EC2 [1] virtual machine image and control scripts to run analyses on EC2. This made it easy to automate running FIE on our large corpus of programs. We will publicly release an open-source version of FIE, associated scripts, and the EC2 virtual machine image.

4 Details of FIE's Architecture

4.1 Main Execution Loop

For the purposes of describing FIE, we define an execution *state* to be an immutable snapshot of the symbolic execution at a given point in time. That means it includes all values used to emulate LLVM bitcode, including a program counter, stack frames, and global memory (used for global variables, the heap, etc.). Any memory location may have either a concrete value or a symbolic one, the latter represented by a set of constraints.

In our abstraction, the main execution loop of FIE operates by generating successor states from the current immutable state. This allows us a history of past states, which, looking ahead, will be useful for describing our state pruning feature. This treatment differs from [10], which instead described states as mutable objects transformed by the symbolic execution.

Figure 4 gives high-level pseudocode for the main execution loop. A set $\mathcal{AS}$ contains the active states to be run; at the start it holds just one initial state. The loop chooses a state from $\mathcal{AS}$ according to a state selection heuristic R. For this we use the KLEE heuristic that seeks to maximize coverage. Once a state has been selected, new successor states may be immediately spawned according to SpawnInterrupts. This function also outputs a boolean *shouldExec* that can be set to false to force an interrupt to fire, otherwise the instruction at the current state's program counter is symbolically executed.

Should *shouldExec* be true, FIE symbolically executes the next instruction of the current state. Here FIE interposes on memory loads and stores that target memory addresses corresponding to special memory (e.g., peripherals). The addresses of special memory are provided by the memory spec as described in the next section. Other operations are handled by Eval, which works like KLEE's evaluation mechanism, except with a new special-memory-aware memory manager, support for emulation of 16-bit firmware, and compiler intrinsics used by `msp430-gcc`.

Each of SpecLoadEval, SpecStoreEval, and Eval must check that security properties are satisfied. Should one fail, a warning will be generated and the set of successors $\mathcal{S}$ output by the evaluation function will be empty. This allows execution to continue, along other paths, even after one path leads to an error.

The set of possible successor states $\mathcal{SS}$ is then taken to be the union of those output by SpawnInterrupts and one of the eval functions. In a normal symbolic execution engine, the full set $\mathcal{SS}$ would be added to $\mathcal{AS}$. FIE works a bit differently due to state pruning and memory smudging as we explain in Sections 4.3 and 4.4.

```
1: $\mathcal{AS} = \{S_{\text{init}}\}$
2: while $\mathcal{AS} \neq \varnothing$ do
3:     Dequeue $S$ from $\mathcal{AS}$ according to $R$
4:     $(shouldExec, \mathcal{S}_{\text{int}}) \leftarrow \mathsf{SpawnInterrupts}(S)$
5:     if $shouldExec$ then
6:         Let $p$ be the program counter of $S$
7:         Let $I$ be the instruction pointed to by $p$
8:         if $I$ is a load to special memory then
9:             $\mathcal{S} \leftarrow \mathsf{SpecLoadEval}(I,S)$
10:        else if $I$ is a write to special memory then
11:            $\mathcal{S} \leftarrow \mathsf{SpecStoreEval}(I,S)$
12:        else
13:            $\mathcal{S} \leftarrow \mathsf{Eval}(I,S)$
14:    $\mathcal{PS}_p \leftarrow \mathcal{PS}_p \cup \{S\}$
15:    $\mathcal{SS} \leftarrow \mathcal{S}_{\text{int}} \cup \mathcal{S}$
16:    for all $S' \in \mathcal{SS}$ do
17:        Let $p'$ be the program counter of $S'$
18:        if $\mathsf{Prune}(S', \mathcal{PS}_{p'}) = \text{false}$ then
19:            $S'' \leftarrow \mathsf{MemorySmudge}(S', \mathcal{PS}_{p'})$
20:            $\mathcal{AS} \leftarrow \mathcal{AS} \cup S''$
```

Figure 4: Pseudocode of FIE's main execution loop.

4.2 Modeling Chips and Peripherals

FIE must be aware of various aspects of the target architecture, including what are valid memory addresses, whether they correspond to special memory locations, and how interrupt firing should be simulated. With over 400 chips in the MSP430 family, hard-coding this information would be cumbersome. Instead, FIE is configured at runtime to work for particular models of chips, external peripherals, and interrupt firing. In this section, we discuss the details of writing an analysis specification file, together with a memory spec and interrupt spec. Combined these serve as a layer of abstraction between the symbolic execution engine and the actual target chip's hardware details.

Analysis specification: When FIE is run, the analyst indicates the target architecture on the command line. In turn FIE loads an associated *analysis specification* file, which is a plaintext file adhering to a simple format and specifying how the analysis should be configured. An example is shown in Figure 5.

Recall that each MSP430 chip has memory locations that correspond to on-chip peripheral addresses. As well there are other hardware specifics, e.g., the location and length of non-volatile flash memory segments. These memory locations differ amongst chips. For example, PORT 0 input resides at memory location `0x0020` on the MSP430G2221, but resolves to `0x200` on the MSP430F5521. This information is generally not included in firmware source-code. The specification file therefore includes information on the layout of memory and what addresses correspond to special memory. In the example, the file fixes the total size of memory on line 1,

```
layout 0x10000
range 0x1080 0x10bf flash
range 0x10c0 0x10ff flash
addr P1IN 0x0020 1
addr P1OUT 0x0021 1
addr P1DIR 0x0022 1
addr P1IFG 0x0023 1
interrupt PORT2_ISR check_PORT2
```

Figure 5: MSP430g2553 analysis specification excerpt

specifies flash regions on lines 2 and 3, and sets the locations and sizes of several special memory addresses on lines 4–7. The final line indicates that the function `check_PORT2` is used to determine when interrupts handled by `PORT2_ISR` fire.

For any MSP430 chip that is supported by `msp430-gcc`, this layout file can be synthesized automatically from firmware source code (for ISRs) and files included in the compiler. While we could therefore have made specifications completely internal, we expose the layout file explicitly to allow an analyst to modify the hardware model if desired.

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, such as when to fire interrupts and the behavior of special memory. These are handled by the memory and interrupt specifications.

Memory spec: The expected functionality of special memory locations is not available in a firmware, and often not really fixed until the device is deployed with attached hardware peripherals. Thus, FIE uses a library of functions that, together, form a model of special memory behavior. For each special memory location, the memory spec contains a function `n_read` and `n_write`, where *n* is the name of the special memory location (e.g., `P1IN_read` and `P1IN_write`). The SpecLoadEval and SpecStoreEval functions determine which of the `n_read` and `n_write` functions to invoke, based on the target address. (Note that the target address may be symbolic, in which case FIE resolves the set of possible addresses, and generates new successors for each possible resulting behavior.)

Read and write functions are passed the entire symbolic execution state, and output a (possibly empty) set of states. This allows special memory reads and writes to define behavior as an arbitrary computation over the state. Security and domain experts can therefore modify a memory spec to refine models of peripheral behavior.

Although this modeling approach is flexible and expressive, previous work has noted that such models can be quite onerous to develop [14]. To eliminate this drawback, we provide a default memory spec which is automatically generated from the analysis spec. For memory reads, the default memory spec returns a fresh, unconstrained symbolic value. For example, reading from `P1IN` always returns a new, unconstrained, symbolic, 8-bit variable, while writing to `P1OUT` is a no-op. This default conservatively assumes that an attacker has full control over all peripherals and uninitialized memory. This means that our analysis often overapproximates special memory behavior, and in particular might lead to finding vulnerabilities that cannot always be exploited when specific peripherals are used. This approach is in-line with similar work on modeling symbolic hardware [26], and as we will see in Section 5, empirically results in few false positives.

Interrupt spec: Deciding which interrupt is enabled at a given program point is nontrivial: the MSP430 design documents specify a partial order of priorities over interrupts, i.e., a higher priority interrupt cannot be preempted by a lower priority one. Furthermore, some (but not all) interrupts are only enabled when appropriate status register flags are set. Thus, determining the enabled set of interrupts requires knowledge not only of the architecture but also the current firmware state.

FIE handles this using an interrupt spec. It contains a number of *gate functions*, one for each possible interrupt that can occur on an MSP430. The SpawnInterrupts function executes each gate function, passing each a pointer to the entire execution state. The gate functions return a flag indicating that the interrupt: (1) *cannot* fire at the current instruction (usually indicating that the interrupt is disabled at that program point); or (2) *may* occur at the current program point. For case (2), the gateway function additionally returns a successor state S' that is the same as the current state S except advanced to the first instruction of the associated interrupt handler.

SpawnInterrupts collects the returned values produced into a set of successor states $\boldsymbol{\mathcal{S}}_{\text{int}}$ that includes one successor for each gateway that returned *may*. As well, SpawnInterrupts determines if it's valid for execution to proceed without an interrupt. This reflects the fact that when the firmware is in a sleep state the only valid successor states are in $\boldsymbol{\mathcal{S}}_{\text{int}}$ (i.e., the path must traverse an interrupt handler). In this case, SpawnInterrupts returns *shouldExec* set to false, correctly forgoing evaluation of S. Otherwise it is set to true, and S is evaluated.

FIE uses, by default, an interrupt spec that explores an over-approximation of all feasible paths: any interrupt that is enabled at a particular program point may fire. Thus, an instruction for which n interrupts may fire will have at least $|\boldsymbol{\mathcal{S}}_{\text{int}}| = n$ successor states, and possibly multiple more in the case that the current state is evaluated. In practice, even an attacker with physical access to the chip is unlikely to be able to exercise all possible firing sequences. This means that FIE using the default interrupt spec may yield false positives, but without further information about possible adversaries treating all possible firing sequences is necessary for verification.

The default interrupt spec can be used for all the MSP430 variants: if the firmware does not handle a certain type of interrupt, that gate function is simply ignored. However, swapping out interrupt libraries can still be useful as a way to tune the analysis. For example, in Section 5 we evaluate an interrupt spec that, instead of firing at every instruction, allows interrupts only to fire once per basic block. While this relaxed interrupt model misses feasible paths, it improves performance.

4.3 State Pruning

In the course of analysis, the main execution loop will often generate a set $\mathcal{SS}$ including one or more states S' that will execute equivalently to another, already seen state $\hat{S}$. We call such an S' *redundant*. We denote states that lead to equivalent execution by $S' \approx \hat{S}$ and say S' and $\hat{S}$ are *equivalent*.

Most prior symbolic execution frameworks, including KLEE, simply add redundant states to the set of active states, meaning they will potentially be scheduled for execution later. Consider Figure 4, but modified so that lines 14–18 are replaced by a single line $\mathcal{AS} \leftarrow \mathcal{AS} \cup \mathcal{S}_{\text{int}} \cup \mathcal{S}$. That is, 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.

Redundant states arise frequently in our setting, and as we will show experimentally in Section 5, PLAIN is slowed down considerably by them. One reason is that interrupt firings can lead to two different paths leading to the same state. Figure 6(a) shows an example interrupt handler and code. At line 1, interrupts are enabled. By the beginning of line 4, when running PLAIN there would be 4 states resulting from the paths $P_1 = \langle s_2, s_3, s_4 \rangle$, $P_2 = \langle s_2, s_7, s_3, s_4 \rangle$, $P_3 = \langle s_2, s_3, s_7, s_4 \rangle$, $P_4 = \langle s_2, s_7, s_3, s_7, s_4 \rangle$, where s_i represents the statement at line i. The states S_4 and S_4' resulting from execution along paths P_2 and P_3 are equivalent, since both increment a via the interrupt handler once — even though they explore distinct program paths all variables have the same value.

A second source of redundant states arises when symbolic execution of loops generates redundant states. This situation also causes the PLAIN mode of FIE to loop infinitely. Consider when running PLAIN from a state S_3 on the looping line 3 in the code snippet shown in Figure 6(b). The main loop will call SpecLoadEval and in turn invoke the memory spec function associated to P1IN. An unconstrained symbolic variable will be generated and two successor states will be returned: S_4 set to line 4 (the branch condition assumed to fail) and S_3' remaining on line 3 (the branch condition succeeded). When S_3' runs, it will again generate two new states, S_4' and S_3''. Yet, $S_4' \approx_A S_4$ and $S_3' \approx_A S_3''$. This will continue endlessly, generating a large number of states and ultimately ensuring that the analysis will never complete.

In KLEE and most prior systems redundant states were dealt with indirectly, by way of state selection heuristics R that favored new lines of code. We would like to support complete analyses, however, and so we go a different route and instead build into FIE the ability to detect and prune redundant states.

State pruning was used previously by RWset [6], which detects if two states $S', \hat{S}$ are equivalent by checking if the set of values taken by all live variables (plus appropriate context such as the call path) of S' match those seen in $\hat{S}$, giving rise to a narrower notion of equivalence that we denote by $\hat{S} \approx_L S'$. Deciding $\hat{S} \approx_L S'$ uses a live variable analysis at each program point. We do not have a live variable analysis that is sound in the presence of interrupt paths, which are prevalent in our domain. We expect that such an analysis would be costly and less accurate when accounting for interrupts, and so we go a different route. FIE checks equivalence by investigating if every variable, symbolic expression[2], program counter, and all other parts of the state are equal between $\hat{S}$ and S', denoted $\hat{S} \approx_A S'$. This embodies a trade-off between simplicity of equivalance checking (i.e., we forego static analysis) and the ability to prune as aggressively as is theoretically possible.

Lines 14–20 of Figure 4 realize state pruning. There a function Prune checks each potential successor $S' \in \mathcal{SS}$ to see if it is equivalent to any of the previously generated states in $\mathcal{PS}_{p'}$, namely those that have the same program counter p' as S'. To use the $\approx_A$ equivalence relation efficiently, we modify the way KLEE maintains states in memory, storing for each visited program counter a set of diffs of the memory contents of all states that have been seen at that program counter. This also allows fast comparisons to detect redundant states.

4.4 An Optimization: Memory Smudging

As we will see in the next section, FIE as described thus far can already be used to perform complete analyses for some simple firmware and achieves good code-coverage for some more complex firmware. However, it is clear that even small programs can force FIE to attempt to explore an intractable number of states. For example, consider the code snippet in Figure 6(c). The empty for loop on line 4 will force FIE to proceed down at least one path of length at least MAX_LONG instructions. Unlike the loop example in Figure 6(b), state pruning cannot short-circuit evaluation of this long path because the value of

[2]We only consider syntactic equality of constraints, and do not attempt to decide if two different sets of constraints define the same set of possible values.

```
1 int main(){
2   eint();
3   BCSCTL1 = CALBC1_1MHZ;
4   DCOCTL = CALDCO_1MHZ;
5 }
6 ISR(PORT1){
7   a += 1;
8 }
```

(a) Code with equivalent paths

```
1 uint8_t getByte(){
2   ...
3   while(P1IN & BIT2);
4   if (P1IN & BIT2){
5     goto WaitForStart;
6   }
7   ...
8 }
```

(b) Code with an infinite fork

```
1 int main(){
2   ...
3   long i = 0;
4   while(i < MAX_LONG) {
5     i++;
6   }
7   ...
8 }
```

(c) Code with a long loop

Figure 6: State pruning can detect and remove the redundant states produced in code samples (a) and (b). Memory smudging replaces `i` in code sample (c) with a symbolic variable after t iterations (e.g., $t = 100$), enabling analysis to move beyond the loop more quickly.

`i` is monotonically increasing and so states never repeat along the path.

To speed analysis for such settings, while retaining the ability to be complete, we use *memory smudging*. It is represented by the function MemorySmudge on line 19 of Figure 4. At analysis time, the analyst supplies a modification threshold t to FIE. Before adding a (non-redundant) successor state S' to $\mathcal{AS}$, the function MemorySmudge checks if any memory locations in S' have been modified t times. If so the location's value is replaced by a special value $\star$. This wildcard value may take any value allowed by the type and cannot be constrained. To implement this, FIE keeps a count of every distinct value that an instance of a variable takes on at a program point. The count is associated with the activation record of the variable. Thus, if a local variable is smudged, it will again be concrete on the next call to that function while global variables remain smudged.

Smudging allows the analysis to explore more of a firmware at the cost of precision. To see this, consider again Figure 6(c), and let the smudging threshold be $t = 100$. On iteration 100, i takes on the value $\star$. Then, as the loop continues to iteration 101, the condition $i <$ MAX_LONG will cause the execution state to be split into two states: a state S_F that fails the loop condition and proceeds to line 7, and a state S_T that executes the body of the loop at line 5 again. By executing S_T, code that would not be executed until MAX_LONG iterations of the loop can instead be executed after 100 iterations. This approach does lead to the addition of new states (compared to execution without smudging), but we have found that pruning typically eliminates states added due to smudging. When FIE executes S_F in the example above, it will explore the (empty) body of the loop, and i will be incremented. However, since $i = \star$, the update to i will be discarded. Now, S_F is again at the head of the loop, and execution state is identical to the previous iteration: no variable besides i has been touched, and $i = \star$ as it did on the previous iteration. Thus, S_F ends up pruned.

Memory smudging over-approximates a state and so can be a source of false positives, i.e., $\star$ contains values that may never be realized along any path. For example, a pointer modified t times and then dereferenced can result in a false positive. FIE reports in output warnings if any involved values were smudged, making it easier for analyst to detect such a false positive. As we see in the next section, false positives due to smudging seem rare in practice for reasonable values of t.

4.5 Implementation Details

The pseudocode presented in Figure 4 gives the high-level logic of FIE but abstracts away many details for simplicity. Our implementation includes a number of important embellishments, which we can only briefly describe here.

Memory sharing: Since FIE creates at least one new state at nearly every instruction, it is important that the creation and storage of states be as efficient as possible. Thus, we only store one complete state per calling context for each $\mathcal{PS}$. Additional states with the same calling context are then compared to the existing state, and only the incremental difference in that state are stored. We also inherit memory optimizations from KLEE, the most important of which is copy-on-write memory for states.

Pruning frequency: The PRUNE operation shown on line 18 of Figure 4 can become expensive as the number of states at $\mathcal{PS}$ becomes large. Rather than performing this operation at each instruction, the default mode of FIE prunes only at basic block boundaries. We preserve the ability to configure FIE to prune at each instruction, but have found that basic-block-level pruning improves performance in all our tests.

5 Evaluation

To evaluate FIE, we used it to analyze the 99 firmware programs in our corpus. We will investigate the overall efficacy in terms of code coverage, the ability to complete analyses, the utility of pruning and smudging, and the bugs FIE helped us find.

Firmware size and coverage: We first fix some conventions regarding how we measure the size of firmware programs and analysis coverage. For our evaluations, we measure firmware size by the *number of executable LLVM instructions*, denoted by the acronym NEXI. We compute a firmware's NEXI by: (1) compiling the firmware into LLVM bitcode using CLANG; (2) running the resulting bitcode through LLVM optimization passes for global and local dead code elimination; and (3) taking the number of LLVM instructions in the resulting bitcode as the NEXI. This count includes intrinsic functions and library functions called by the firmware. We note that some programs used external modules whose source code was not included in their source tree; we did not attempt to track down these libraries and FIE emits an error should it execute an instruction calling an omitted function. Likewise for inline assembly instructions not supported by FIE. This did not significantly affect our evaluation, e.g., only two programs ever reached missing functions or inline assembly in the 50 minute runs reported on below. Note that usually FIE continues running in such cases along other paths.

Code coverage is the fraction of LLVM instructions executed in the course of the analysis divided by the NEXI of the target firmware. Using NEXI as opposed to C lines of code better aligns our complexity and coverage metrics with the work done by FIE, and avoids any ambiguity in terms of overcounting coverage of partially executed blocks or lines of C code. The NEXI sizes are, on average, 1.5 times larger than the number of lines of C code computed in Section 2. NEXI was smaller than cloc for 23 of the programs due to dead code elimination.

Experimental setup: All the analyses reported on below used Amazon EC2 high-memory, double-extra-large (m2.2xlarge) instances which have 36 GB of RAM and 13 virtual CPUs (each advertised to be the equivalent of an 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor). Unless specified otherwise, FIE was given 50 minutes of runtime[3], and each analysis was performed on a separate EC2 instance. To facilitate this effort, we wrote a set of scripts for launching, monitoring, and retrieving the results of FIE run via a custom EC2 VM image.

Coverage under different FIE modes: We started by analyzing each firmware for 50 minutes for each of five different modes supported by FIE, for a total of 495 executions. The resulting NEXI coverages are shown in Table 7. We now explain the modes and discuss their performance.

Baseline: The BASELINE mode reflects a bare minimum port of KLEE to the MSP430 environment, in particular it has support for: 16-bit addressing; a custom memory allocator that ensures that memory objects do not collide with special memory locations and have addresses within the chip's address range; and implementation of intrinsics supported by msp430-gcc. It does not, however, have any knowledge of the semantics of, special memory or interrupts, etc. For most firmware, the BASELINE analysis performs very poorly, with a median of 1.7% coverage. This is because BASELINE almost always ends prematurely with a false positive since the firmware appears (to the analysis) as if it were reading from an uninitialized memory location. Manual inspection of the code of the two outliers (from GitHub) revealed that they are not using any features of the MSP430 architecture. The poor coverage of BASELINE for the other firmware programs attests to the importance of providing an architecture-aware analysis.

Fuzz: We next use FIE to realize a general-purpose fuzzing tool for MSP430 firmware. This mode, unlike BASELINE, takes advantage of knowledge of the memory layout, special registers, and interrupt handling semantics. We implemented a special memory spec in which any read to a peripheral results in a returned value chosen uniformly in the appropriate range. (Twice reading the same peripheral location leads to two independent values.) Writes to peripherals are ignored. We use the conservative interrupt spec, meaning that in the FUZZ mode the analysis branches off new states to execute interrupt handlers as appropriate. In this mode, then, FIE never generates symbolic variables, and so is able to quickly evaluate on concrete values along many paths. Fuzzing provides surprisingly good coverage for many of the firmware programs, in fact beating symbolic execution modes in many cases. This is because fuzzing can evaluate states more quickly, and for simple programs this can lead to good coverage in a 50 minute test.

% NEXI	BASELINE	FUZZ	PLAIN	PRUNE	SMUDGE
Complete	n/a	n/a	7	35	52
100%	1	43	40	34	46
90–100%	1	10	9	15	15
80–90%	0	7	5	10	6
70–80%	0	5	5	6	4
60–70%	0	4	5	5	5
50–60%	0	4	6	5	3
40–50%	0	8	8	9	5
30–40%	0	0	0	2	3
20–30%	1	8	11	4	5
10–20%	10	5	5	3	3
0–10%	86	5	5	6	4
Total %	1.1	26.1	23.7	29.5	32.3
Avg. %	5.9	74.5	71.1	74.4	79.4
Median %	1.7	96.9	89.5	88.7	98.1

Table 7: Number of firmware programs for which FIE achieves coverage in the indicated range, for 50 minute runs of FIE in each of five operating modes. "Complete" gives the number of programs for which the mode was able to analyze all possible symbolic states.

[3]Setting the time to a bit less than one hour halves the cost of running on EC2.

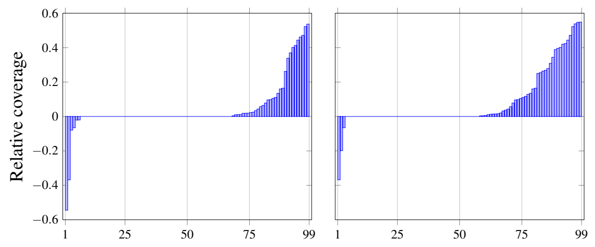


Figure 8: Coverage of SMUDGE relative to FUZZ (top) and SMUDGE relative to PLAIN (bottom) for the 50-minute tests. Here SMUDGE outperforms FUZZ and PLAIN for 32% and 42% of the programs, respectively.

Mode	Termination status			FPs
	No mem	Timeout	Finished	
BASELINE	9	2	88	93
FUZZ	10	79	10	0
PLAIN	7	85	7	0
PRUNE	0	64	35	0
SMUDGE	0	46	53	1

Table 9: Counts of each termination code seen in the 50-minute runs. “No mem”: the analysis exhausted memory; “Timeout”: analysis ran for the full 50 minutes; “Finished”: analysis completed early. The final column is the number of firmware programs with erroneous bug reports.

Plain, Prune, and Smudge: We now turn to modes that use FIE as a symbolic executor with the architecture-aware analysis. To compare the efficacy of the state pruning and memory smudging techniques, we use three different modes: PLAIN (no pruning or smudging), PRUNE (with pruning but not smudging), and SMUDGE (pruning and smudging, with smudging threshold $t = 100$). All three modes used the most conservative interrupt model. Overall SMUDGE provides better coverage than all others, including FUZZ. A comparison of relative performance for each firmware appears in the charts of Figure 8. The x-axes is the firmware (ordered by y-values) and the y-value is $(N_s - N_f)/N_{tot}$ where N_s is, for the left chart, the number of instructions covered by SMUDGE for this firmware, N_f is the number covered by FUZZ, and N_{tot} is the NEXI for the firmware. The right chart is the same except comparing SMUDGE with PLAIN. These graphs surface two facts. First, there exists a large number of firmware programs for which the analyses do equally well (where relative coverage is 0, most often because both analyses had 100% coverage), which is due to the large number of very simple firmware from GitHub and the TI website. Second, SMUDGE can do worse than others on a few firmware programs, but improves performance over FUZZ for 32% of the programs and over PLAIN for 42% of the programs.

50-minute analysis outcomes: In Table 9 we give a breakdown of the emitted termination status for the analyses. FIE can either stop because it runs out of memory (No mem), the requested amount of execution time has been reached (Timeout), or because there exist no more active states (Finished). Additionally, FIE will output bug reports. As can be seen, pruning and smudging help reduce memory usage and increase the number of analyses that finish. Potential bugs were reported for 92 firmware programs by the BASELINE, all false positives. Smudging introduced a false positive in one firmware, since a pointer was smudged. (Smudging a pointer frequently leads to a memory safety violation, because any dereference of it will be viewed as an error.) FIE makes it easy to determine if a warning is related to smudging by marking variables that were smudged as such in the bug report. No true positives were found in these short runs.

Recall that an explicit design goal for FIE was the ability to support complete analyses (all possible symbolic states are checked). The PLAIN, PRUNE, and SMUDGE modes do support this: an analysis is *complete* if the termination status was Finished and no bugs were reported. Modulo the limitations discussed in the next section, this verifies the absence of bugs. The first row of Table 7 shows the number of firmware programs for which PLAIN, PRUNE, or SMUDGE were able to verify the absence of memory safety and (some kinds of) peripheral misuse bugs, in these short 50 minute runs. As can be seen, our pruning and smudging mechanisms enable a huge increase in the number of analyses that FIE can complete: a 6x increase when we use pruning and an additional factor of 1.48x improvement when we add in smudging. In the end, the total number of firmware programs for which one of the analysis modes completed is 53. (One firmware was completed by PRUNE but had a false positive under SMUDGE, and so it was not counted in the SMUDGE column of Table 7.)

We note that these complete analyses revealed that 13 firmware programs have dead code missed by the static optimization passes, which means for these our measured coverage is lower than it should be (e.g., for one firmware we achieved a complete analysis but only 45% coverage). For consistency, we do not correct the NEXI values for these firmware programs.

Firmware complexity measures: The above shows that FIE enables complete analyses for a majority of the firmware programs in our corpus, yet the total amount of code covered across the full corpus indicates that FIE was not able to explore most of the larger firmware programs given only 50 minutes. As can be seen in Figure 10, the coverage is uniformly poor for firmware programs with more than 4,000 executable instructions. More subtly, there exist many much smaller programs for which coverage is also poor (the vertical trend closer to the y-axis), which could be due to complicated but short code con-

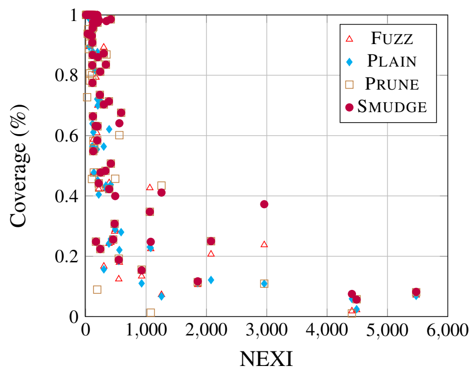


Figure 10: Coverage as a function of firmware size in the 50 minute tests.

Complexity	Criteria	# FWs	SMUDGE coverage
low	≤ 100 NEXI or < 2 loops	49	Avg: 93.6% Med: 100%
medium	≤ 500 NEXI and ≥ 2 loops	37	Avg: 79.5% Med: 93.1%
high	> 500 NEXI and ≥ 2 loops	13	Avg: 27.8% Med: 24.8%

Table 11: Criteria for firmware complexity groups, the number of firmware programs (# FWs) in each group, and SMUDGE's average and median coverage for each group.

structs or, perhaps, undiscovered dead code.

To focus subsequent experiments on the more challenging firmware programs, we partition the programs into three complexity groups based on a simple static analysis. (Using static analysis avoids biasing the set unnecessarily by the nature of FIE's analysis.) The criteria for partitioning our programs into low-, medium-, and high-complexity groups is described in Table 11. To determine the number of loops in a program, we use LLVM's built-in loop detection. We chose this particular partitioning for its simplicity, but admit there are many other possibilities. We give the average and median performance of the 50-minute SMUDGE runs as broken down by each group. Of the 53 programs that FIE is able to complete analysis for in 50 minutes, 38 are low complexity and 14 are medium complexity. The average NEXI for these completed programs is 84.4 and the average number of loops is 2.2; the most complex completed program has a NEXI of 414 and 17 loops.

Effect of smudging threshold: We now measure the effect of the smudging threshold t on coverage and false positive rates for the high complexity firmware programs. By decreasing t one might hope to achieve a trade-off between coverage improvements (by breaking out of loops even more quickly) and increased risk in false positives. We run SMUDGE for 50-minutes for each of $t = 1, 10, 1000$ for the 13 firmware programs and

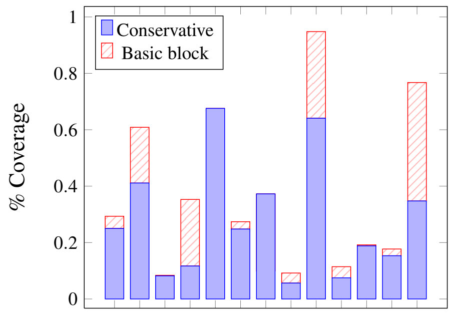


Figure 12: Coverage when spawning interrupts every instruction (Conservative) versus once per basic block for the 13 high-complexity firmware programs.

use as well the $t = 100$ results from the runs discussed above. The average coverages were 23.3%, 25.2%, 25.5%, 25.6% for $t = 1, 10, 100, 1000$, making the differences too small to be significant. The number of false positives increased for small t; with $t = 1$ there were two false positives, and none for the larger values of t. We conclude that $t = 100$ strikes a reasonable balance, but further performance improvements may not be easily obtained by tweaking t.

Relaxing the interrupt model: Recall that we have so far been using FIE with a very conservative interrupt model in which all enabled interrupts fire at every program point. This can mean that most instructions, as opposed to just branches, end up forking off multiple new states. We therefore implement a relaxed interrupt model in which every enabled interrupt fires at the first instruction of each basic block, but not during subsequent instructions. This means analysis will miss possible paths (barring complete analyses) but could speed up performance and thereby increase code coverage. In Figure 12 we compare, for the high-complexity firmware programs, the coverage obtained by SMUDGE with $t = 100$ using both the conservative interrupt model (Conservative) and the new model that only fires at each basic block (Basic block). The results are both from 50 minute runs. Several of the firmware programs see drastic coverage improvements, the last bar on the right represents the largest improvement at 232%. No false positives arose in these basic block runs, however one program hit a code construct[4] currently not supported by FIE.

Finding vulnerabilities: FIE currently supports finding two types of bugs: memory safety violations, such as buffer overruns and out-of-bounds accesses to memory objects like arrays, as well as peripheral-misuse errors in which a firmware writes to a read-only memory location or to locked flash. It will be easy to increase scope to further security properties in the future.

[4]A firmware used a custom variable argument function. We plan to add support in the public release version.

Firmware	NEXI	Types	# bugs
CDC Driver	4,489	Memory safety	10
HID Driver	2,958	Memory safety	11
controleasy	1,255	Flash misuse	1

Table 13: Summary of vulnerabilities discovered by FIE. The final column is the number of distinct vulnerabilities in the firmware.

We supplemented the above analyses with runs in which we allowed SMUDGE to run up to 24 hours, with $t = 100$ and using the conservative interrupt model, on each of the 13 high-complexity programs. Table 13 gives a breakdown of the 22 bugs found across all of the runs. The bugs were spread across three firmware programs, the two USB drivers supplied by TI and one community firmware called Controleasy. Of the bugs, 21 were memory safety violations while one was a flash misuse bug. CDC Driver and HID Driver share some common source files, one of the bugs spans both, while the others are from disjoint source files. The memory safety bugs in the two USB drivers include 18 vulnerabilities in which a USB protocol value (received from off-chip) ends up controlling an index into an array, allowing adversarial reads or the ability to crash the firmware. One of the vulnerabilities, in HID Driver, allows an adversary over the network to inject arbitrarily long strings due to an unprotected `strcat`. This allows crashing the firmware, but may also lead to a complete compromise by way of control flow hijacking. The final two memory safety bugs are present in both programs but arise from the same source file. The bug dereferences a value read from flash, which in our model is untrusted but unlikely to be exploitable in most settings.

The TI community code project controleasy has a peripheral misuse bug in which a read-only I/O port can be written to based on the value of another peripheral. Like the attacker-controlled reads in the USB code, this bug can be used by an attacker that can send data to PORT 1 to cause the firmware to crash.

6 Limitations

The evaluation in the last section evidenced FIE's effectiveness at both finding bugs as well as verifying their absence. Of course, FIE does have some limitations.

The design of FIE arises from a philosophy that sound and complete analysis are valuable and can be feasible for the embedded firmware often found in practice. However, it is simple to show that there exist firmware for which complete analyses are intractable, and likewise soundness is only with respect to the symbolic execution framework (it is possible that reported bugs may not arise in the firmware when run natively, as discussed below). Indeed some of the firmware in our corpus (e.g., Contiki) appear to have, in particular, an intractably large number of reachable states. Here FIE attempts to provide as high as possible code coverage, but improving on the results reported in the last section might require different techniques than currently used. For example, a combination of loop elision [14, 26] and improved state selection heuristics might be more effective than state pruning and memory smudging. Future work might therefore explore incorporation of other techniques into FIE.

Both when achieving complete analyses and when not, there exist various sources of imprecision in analysis that may lead to false positives or false negatives. In developing FIE we often encountered analysis errors due to bugs in the analysis software or misconfiguration (e.g., using the incorrect target architecture almost always yields false positives). These problems were subsequently fixed, and while we are unaware of any outstanding bugs in FIE and have manually verified all the bugs reported in Section 5, it could be that some analysis errors remain.

Imprecision can also arise due to discrepancies between the firmware as symbolically executed in FIE and natively in deployment. In building FIE, we had to implement extensions to C that are (sometimes implicitly) defined by msp430-gcc. We encountered inconsistencies between msp430-gcc and FIE, which were subsequently fixed, but some may remain. These C extensions also differ among the three MSP430 compilers, and so analyzing firmware written to work for the IAR or CCS compilers (e.g., the USB drivers in our corpus) may give rise to analysis errors when using FIE. Even so FIE can still be useful for finding vulnerabilities in such firmware programs, as the bugs found in the USB drivers shows.

As a final source of imprecision, our most conservative analysis models peripherals and interrupt firing as adversarially controlled. This means that FIE may explore states that do not arise in real executions, and errors flagged due to such states would constitute false positives. We feel that fixing even such bugs should be encouraged, since it reduces the potential for latent vulnerabilities. Moreover, it is unclear where to draw the line in terms of adversarial access to a chip. That said, FIE is easily customizable should such false positives prove burdensome, or to receive the speed-ups of other environmental models.

Finally, we note that currently FIE fails execution paths that include inline assembly. While we added some explicit handlers for several inline assembly instructions (e.g., `nop`), this approach would struggle with complex assembly code constructs. Future work might investigate performing symbolic analysis starting with MSP430 assembly, similarly to [7].

7 Related Work

FIE is based off of KLEE and, in turn, builds off the work of KLEE's predecessors such as EXE [11]. These prior systems target generation of high-coverage test suites for non-embedded programs (e.g., Linux CoreUtils). As we saw in previous sections, using KLEE with a minimal amount of porting provides poor coverage. The many systems that extend KLEE [2,9,13,28,31,36] do not target embedded systems, with the exception of KleeNet [28]. It targets wireless sensor nodes running Contiki [35], but only on x86 platforms, and so does not work for our setting of MSP430 firmware programs.

Concolic execution systems extend symbolic execution by concretizing values that cannot be handled by the constraint solver efficiently (or cannot be handled by the constraint solver at all) [14, 30]. Whole-system concolic execution tools like S2E [14] can execute external functions natively by concretizing symbolic arguments, and then providing the concrete value in the call. Their model of concretization makes less sense in our setting, where we have a firmware that specifies all software on the system and interacts only with hardware peripherals. For the latter, we can support concretization in the sense that a memory specification can return concrete values, change symbolic values to concrete, etc.

SymDrive [26] builds off S2E to test Linux and FreeBSD kernel drivers without the need for the actual hardware, and treats many of the same problems as FIE, including modeling hardware, dealing with polling loops, etc. SymDrive uses static analysis to help guide execution along states that reach deep paths and to avoid loops. This improves code coverage, but does not enable complete analyses. We leave incorporating such static analysis techniques into FIE, in order to increase code coverage in conjunction with state pruning and memory smudging, for future work.

Pruning redundant states during an analysis has been considered before in a variety of program analysis contexts [3, 5, 32]. Closest to our work is RWset [6], which extended the EXE [11] symbolic execution engine to track live variables and to discard a state should the values of all live variables have already been executed upon. Our state pruning approach is simpler and does not require an auxiliary live variable analysis (which can be challenging in the face of interrupt-driven code). The trade-off for this simplicity is that FIE may prune less aggressively than possible. On the other hand, FIE goes further than RWset in limiting path explosion via memory smudging, which is effective even when, for example, variables written within a loop are live.

There is a body of work on improving the performance of symbolic execution by merging similar states [23, 25]. State merging seeks to combine identical (or similar) active states, whereas state pruning compares active states to both active and prior states. Only the latter enables complete analysis. Whether the two techniques are useful in conjunction is an interesting open question.

Much effort has gone into improving the scalability of symbolic execution [7, 15, 29]. One such example is Cloud9, which speeds symbolic execution by parallelizing the execution of multiple memory states across a cluster of commodity hardware. We note that such techniques are applicable to FIE, and future work may involve adopting such techniques to improve the performance of FIE for large firmware programs.

Finally, we are aware of two commercial tools of potential relevance to FIE. The first, Codenomicon [16], offers a network protocol fuzzing tool for embedded medical devices. It therefore targets protocol parsing logic, which is a frequent source of vulnerabilities. FIE already supports rudimentary fuzzing, and could perform network protocol fuzzing (or a mixture of fuzzing and symbolic execution) by implementing more detailed memory specs. Second is Coverity [4], a static analysis tool that targets a number of platforms, including the MSP430. While we have access to Coverity, their software license unfortunately prevents head-to-head comparisons in published research.

8 Conclusion

In this paper, we presented the design and implementation of FIE, a tool for performing symbolic-execution-based analysis of MSP430 firmware programs. It provides an extensible platform for finding security vulnerabilities and other kinds of bugs, and has proven effective in analyzing a large corpus of open-source MSP430 firmware programs. To increase code coverage in a way that supports verification of security properties, we incorporate into FIE the techniques of state pruning and memory smudging. We used FIE to verify memory safety for 53 firmware programs and elsewhere found 21 distinct vulnerabilities, some of which appear to be remotely exploitable. All this shows that FIE is particularly well-suited to the small, simple firmware programs often used for microcontrollers and proves useful for analysis of more complex firmware programs as well.

Acknowledgements

We would like to thank Kevin Fu, Matt Renzelmann and the anonymous reviewers for their extensive feedback on earlier drafts of this paper. This work was supported, in part, by DARPA and AFRL under contracts FA8650-10-C-7088 and FA8750-10-C-0237. The views, opinions, and/or findings contained herein are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] Amazon. Amazon elastic compute cloud. `http://aws.amazon.com/ec2`, 2013. Last accessed Jun 2013.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[3] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, July 2011.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007.

[6] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 351–366. Springer Berlin Heidelberg, 2008.

[7] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries. Technical report, In CMU-CS-07-133, 2007.

[8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP)*, pages 2–16. IEEE Computer Society, 2006.

[9] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, pages 183–198, 2011.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, pages 209–224. USENIX Association, 2008.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications security*, pages 322–335. ACM, 2006.

[12] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of USENIX Security*, 2011.

[13] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys*, pages 167–180, 2010.

[14] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, Mar. 2011.

[15] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: a software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, Jan. 2010.

[16] Codenomicon. Codenomicon defensics. `http://www.codenomicon.com`, 2013. Last accessed Jun 2013.

[17] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Symposium on Operating System Principles (SOSP)*, pages 117–130, 2007.

[18] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[19] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Annual Computer Security Applications Conference (ACSAC)*, pages 97–106. ACM, 2010.

[20] W. Frisby, B. Moench, B. Recht, and T. Ristenpart. Security analysis of smartphone point-of-sale systems. In *Proceedings of the 6th USENIX conference on Offensive Technologies (WOOT)*, pages 3–3, 2012.

[21] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *IEEE Symposium on Security and Privacy (SP)*, pages 129–142, 2008.

[22] D. Halperin, T. Kohno, T. Heydt-Benjamin, K. Fu, and W. Maisel. Security and privacy for implantable medical devices. *Pervasive Computing, IEEE*, 7(1):30–39, 2008.

[23] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In *Runtime Verification, 9th International Workshop, RV 2009*, pages 76–92, 2009.

[24] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[25] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204, 2012.

[26] M. J. Renzelmann, A. Kadav, and M. M. Swift. Symdrive: testing drivers without devices. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 279–292. USENIX Association, 2012.

[27] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar. Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study. In *Proceedings of the 19th USENIX conference on Security*, 2010.

[28] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Stockholm, Sweden, April 2010.

[29] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *International Symposium in Software Testing and Analysis (ISSTA)*, pages 225–236, 2009.

[30] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[31] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[32] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *In Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224. Springer-Verlag, 1995.

[33] Texas Instruments. Microcontroller projects website. `http://e2e.ti.com/group/microcontrollerprojects/m/msp430microcontrollerprojects/default.aspx`. Last accessed Jun 2013.

[34] Texas Instruments. MSP430 for security applications. `http://www.ti.com/mcu/docs/mcuorphan.tsp?contentId=33485&DCMP=MSP430&HQS=Other+OT+430security`, January 2012.

[35] The Contiki Project. Contiki. `http://www.contiki-os.org/`. Last accessed Jun 2013.

[36] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.