Using Program Analysis to Synthesize Sensor Spoofing Attacks

Ivan Pustogarov, Thomas Ristenpart, and Vitaly Shmatikov
Cornell Tech

ABSTRACT

In a sensor spoofing attack, an adversary modifies the physical environment in a certain way so as to force an embedded system into unwanted or unintended behaviors. This usually requires a thorough understanding of the system's control logic. The conventional methods for discovering this logic are manual code inspection and experimentation.

In this paper, we design a directed, compositional symbolic execution framework that targets software for the popular MSP430 family of microcontrollers. Using our framework, an analyst can generate traces of sensor readings that will drive an MSP430-based embedded system to a chosen point in its code. As a case study, we use our system to generate spoofed wireless signals used as sensor inputs into AllSee, a recently proposed low-cost gesture recognition system. We then experimentally confirm that AllSee recognizes our adversarially synthesized signals as "gestures."

1. INTRODUCTION

The explosion in the popularity of embedded systems is fueled in large part by their ability to sense, interpret, and react to physical environments. For example, gesture recognition systems measure signals and interpret them as user commands [1,2,7,18], cameras that sense motion or light [22] help autonomous systems navigate, and temperature sensors inform household HVAC systems.

Software controlling embedded systems is often designed assuming a benign environment, for example, a device owner interacting with his own device. In real-world deployments, however, it is necessary to consider *sensor spoofing* threats. An attacker with direct or indirect access to the physical environment can emit physical signals that maliciously force unwanted or unintended behaviors in the victim software and, consequently, in the system controlled by this software. Recent examples include disrupting the flight of autonomous drones using sound generated by off-the-shelf computer speakers [30] or laser pointers [11], injecting bogus signals into medical devices [20], controlling speech recognition interfaces with nonsensical noise [33], and many others [24, 25, 29, 34]. These attacks are usually developed by manual reverse engineering and experimentation.

Synthesizing sensor spoofing attacks. In this paper, we initiate research on software analysis tools that can help discover and exploit sensor spoofing vulnerabilities in embedded software. We focus on a popular class of embedded systems: lightweight microcontrollers attached to one or more sensor components. The sensors convert physical readings (temperature, light, sound, electromagnetic signals, etc.) into digital values that are read by the microcontroller's software (often called *firmware* in this context). Such systems are ubiquitous in so-called Internet-of-Things (IoT) applications, autonomous vehicles, and elsewhere.

Sensor spoofing attacks involve an adversary generating malicious inputs in the physical layer to cause certain behaviors in the target firmware. They can be decomposed into a "software" phase and a "physical" phase.

In the software phase, the adversary infers which *digital inputs* to the firmware will force the behavior the adversary wants. In the physical phase, the adversary determines the *analog signals* that will cause the target sensor to output the desired digital readings. In this paper, we assume that the specifications that describe the sensor's physical-to-digital conversion are sufficient for signal generation in the physical phase. Therefore, we focus primarily on the software phase.

Our goal is to build a tool that can automatically discover sequences of digital sensor readings that drive the firmware to an adversarially chosen state. This includes possible sequences that the firmware designers did not expect. As with other types of vulnerabilities (memory safety errors, race conditions, etc.), automated analysis tools will in turn help analysts harden their systems in the face of adversarial behavior.

This is a problem well-suited for symbolic execution [19], given its ability to automatically generate inputs that drive the program into particular execution paths. Existing symbolic execution tools, however, do not work in our setting. In contrast to the large body of research on symbolic execution for traditional architectures such as x86 [3, 5, 12, 16, 23, 27, 28], there are few tools for lower-end embedded architectures [10, 21]. As we will see, even these tools do not perform well on the code patterns characteristic of the embedded code that deals with sensor readings. Interrupt-driven sensor-measurement loops prevent traditional forward symbolic execution from reaching the relevant program points (e.g., those that cause an embedded system to take actions in response to certain sensor inputs). These code constructs

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

ISBN 978-1-4503-4944-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3052973.3053038

cause *path explosion*, i.e., the number of feasible paths that must be considered grows too fast.

Our contributions. We present DrE, the first tool specifically engineered to analyze embedded firmware code with respect to signal input spoofing. To handle the challenges of analyzing embedded firmware, DrE uses a modular approach that takes advantage of both *directed* and *compositional* symbolic execution.

Instead of attempting to explore all code paths, directed symbolic execution [3, 23] starts with a target, i.e., a particular point of interest in the program. In our context, these points are (easily identified) sections of the firmware code corresponding to the control actions that the attacker wants to firmware to take. The output of a successful directed symbolic execution is a *path*, defined as a set of constraints on inputs (i.e., sensor readings) that, if satisfied, will drive the program to the designated point.

To mitigate path explosion, we devised a modular strategy to efficiently find paths to program points of interest. It uses a combination of call graph and control flow graph analysis with compositional symbolic execution. DrE starts by finding possible call chains from the program's entry point to the function containing the target line. It then symbolically executes the function containing the target point and proceeds through a call chain backwards. Each function along the way is (symbolically) executed independently. To generate inter-procedural path constraints, DrE then stitches together the execution paths of individual functions. To do this, DrE collects, rewrites, and propagates the relevant constraints and performs a forward pass to check them.

For efficiency, DrE employs several additional heuristics. DrE biases state selection towards shorter paths leading to the target program point and employs a state-pruning strategy [6, 10] to break out of infinite wait loops.

We applied DrE to a recently proposed gesture recognition system called AllSee [18]. In normal operation, this MSP430-based system attempts to recognize several hand gestures by extracting information from ambient wireless signals (for example, TV or Wi-Fi). We show how to use DrE with a simple model of the AllSee signal strength sensor to generate spoofing attacks. The output of DrE can be converted by the adversary into a sequence of radio signals in the appropriate Mhz range using the sensor's specification. These signals trick the firmware into "recognizing" a gesture of the adversary's choosing.

We experimentally demonstrate feasibility of our synthesized spoofing attack against the actual AllSee firmware with reduced sampling rate connected to an emulated sensor. Using a software-defined radio to generate spoofed signals, we were able to successfully spoof all gestures. The spoofed signals can be used to maliciously control AllSee-equipped devices without the device owner's knowledge or consent. In addition to demonstrating that our approach is capable of synthesizing non-trivial spoofing attacks, our results call into question the security of unauthenticated wireless gesture recognition.

2. OVERVIEW

MSP430. Our program analysis tool targets the MSP430 family of 16-bit RISC microcontrollers from Texas Instruments. It is one of the most popular microcontrollers [14] today, used in a wide range of low-powered embedded systems [32]. Among its features are low power consumption, wide range of peripheral modules, and ease of use.

MSP430 chips have 16-bit-addressable memory, including peripheral control registers, RAM, and Flash memory (the exact amount of each kind of memory depends on the chip). Peripherals are accessed through a set of chip-specific, predefined memory locations called control registers. The actual memory layout for a specific MSP430 chip can usually be found in its specification published by TI. MSP430 programs are typically written in C and make extensive use of memory locations related to peripherals.

MSP430 firmwares are typically interrupt-driven programs. The firmware puts the chip into an infinite loop that sleeps while waiting for inputs from the environment, which are then handled by interrupt handlers.

Threat model. The general class of attacks we are investigating is sensor input spoofing [9, 11, 20, 24, 25, 29, 30, 33, 34]. In the context of gesture recognition systems (see Section 5.3), the goal of a spoofing attack is to force the system to "recognize" gestures that were not actually made by a human body. To do so, the adversary emits wireless signals from a nearby device that drive the recognizer's firmware into a code path that ends with the firmware concluding that a particular gesture was performed. This can enable the adversary to attack higher-level programs relying on gesture input.

One plausible strategy for a gesture spoofing attack is to actually generate a signal (i.e., perform the gesture) that drives the device into the desired state, measure that signal next to the device, and replay the measured signal to other devices. In some scenarios, this approach would work well, but it requires *a priori* understanding of what signal needs to be produced in order to drive the device to the desired state. For more complex systems, this could involve significant manual analysis even given the device's source code. Also, this approach may not work for signals that are difficult to measure precisely in noisy conditions, and it will not help analysts explore firmware behavior in the face of signal sequences not envisioned by the firmware designer.

In this paper, we show how the adversary can use program analysis of the device's source code to determine what the signal should "look like" without actually performing the physical action that generates the signal. Consider a firmware that reads values from an analog sensor connected to the chip through the ADC (analog-to-digital conversion) port and takes a particular action if it recognizes a predefined pattern in these values. We translate this problem of *signal pattern extraction*—how to find sensor readings that cause the program to "believe" that it received a certain signal—into a program analysis problem, namely, *line reachability*.

The attacker first identifies the lines in the code where a pattern-dependent action is taken. By setting each of these lines in turn as the target, our tool generates input constraints necessary to reach the line. These constraints can then be used together with the physical properties of the sensor (e.g., the ADC sampling rate) to generate signal patterns that cause the firmware to perform a given action.

Symbolic execution. When a program is symbolically executed [19], its inputs are replaced by *symbolic values*. Whenever the program's control flow reaches a branch instruction involving a symbolic value, the execution forks into

two *states*. Each resulting state is assigned the corresponding *path condition* on the symbolic variable and the execution proceeds along both branches. One of the key features of symbolic execution is that collected path conditions can be used to generate program inputs for each explored path.

Consider a program with a single input x and a branch instruction conditioned on $x > 5$. At the start of the program, x is replaced by a symbolic variable α. Symbolic execution of this program starts with one state and empty path conditions. When the execution reaches the instruction "`if( x > 5 )`", it forks and a copy of the initial state is created. The first state proceeds along the true branch and adds $\alpha > 5$ to its path conditions. The second state proceeds along the false branch and $\alpha \leq 5$ is added to its path conditions. As the execution goes through more branch instructions, more states are created; the number of states grows exponentially with the number of branches involving symbolic values, usually causing *path explosion*.

In typical applications of symbolic execution, the goal is maximum code coverage, i.e., to to cover as many branches in the execution tree as possible. Most open-source symbolic execution tools are optimized for this goal. *Directed* symbolic execution [3, 23], on the other hand, tries to find an input that would steer the program to a target line of code, exploring as few paths as possible. Since some states have a higher chance to reach the target line than others and non-trivial programs have a large number of states, deciding which states to execute next (i.e., the *state selection strategy*) plays an important role for directed symbolic execution.

3. FORWARD SYMBOLIC EXECUTION

Symbolically executing firmware code for embedded systems has nuances specific to microcontrollers: interrupt-driven control flow, extensive use of peripheral devices, and hardware-related memory areas. Consider a common code pattern[1] for MSP430 microcontrollers in Figure 1. This code poses two problems for conventional tools. First, it uses symbols peripheral-related memory locations, `WDTCTL` and `BCSCTL1`. Before symbolic execution can start, these symbols need to be resolved and the corresponding memory locations initialized (possibly with symbolic values). Second, the code snippet includes an interrupt handler. Since there is no explicit call to the handler, the execution engine should be "educated" on when to execute it.

We chose the FIE [10] symbolic execution engine as our basis for DrE since it already provides memory and interrupt models for MSP430. FIE, in turn, reuses several components from an earlier symbolic execution tool for x86 called KLEE [8]. We started by augmenting FIE with a new state selection strategy that prefers states closer (in terms of the number of basic blocks) to the target line in the program's ICFG (interprocedural control flow graph). We call this mode of DrE "Forward Only Mode." We build the program's ICFG path and select states as follows.

Build call chains. We first choose k shortest call chains $(f_n \leftarrow \ldots \leftarrow f_1 \leftarrow f_0)$ from the target line to the entry point. The first function in this chain, f_n, contains the target line. In our implementation we use $k = 4$.

Build interprocedural control flow paths. For each such chain,

[1]Taken from http://www.simpleavr.com/msp430-projects/rtc-clock

```
1
2  void main(void) {
3
4    uint8_t pos=0;
5    WDTCTL = WDTPW + WDTHOLD + WDTNMI + WDTNMIES; //
         stop WDT, enable NMI hi/lo
6    BCSCTL1 = CALBC1_1MHZ;       // Set DCO to 1MHz
7          ...
8    _BIS_SR(GIE);          // enable interrupt
9    while (1) {
10         ...
11         }
12 }
13
14 interrupt(PORT1_VECTOR) PORT1_ISR(void) {
15   P1IE &= ~(BIT2|BIT7); // disable pin interrupt
16   _BIC_SR_IRQ(LPM4_bits); // wake up, got
         keypressed
17 }
```

Figure 1: Simple MSP430 code example.

we build an ICFG path by concatenating the shortest CFG paths of each f_i (from f_i's entry to the callsite of f_{i+1}).

Expand inner calls. If a CFG path contains an inner call to g, we (recursively) expand it: find g's shortest path from its entry to the return statement and replace the call to g with this path. We then choose the shortest among the k constructed ICFG paths.

Select state. During execution, DrE chooses the next state according to the following strategy. First, it prefers states for which the current instruction is in a function that belongs to the chosen ICFG path. Among these states, it chooses the one that is closest to f_n (in terms of the number of functions). If two states belong to the same function, DrE chooses the one that is closest to the function's return statement (or a call to the next function in the ICFG) in terms of the number of basic blocks.

Limitations of FIE's interrupt model. In FIE, interrupts can be fired either for each basic block or for each instruction. Because interrupt firings cause new states to be generated (one state in which the interrupt fired and one in which it did not), this quickly creates a large number of execution paths, the vast majority of which are not feasible in practice because interrupts would not fire on every instruction (or every basic block) in a real execution. Consider the code snippet in Figure 2: (1) a read from the ADC port is followed by (2) some logic consisting of several basic blocks, and (3) another read from the ADC. During the ADC interrupt, the ADC value is saved and the average of previous values is computed.

The developer's intention here is to check if the last ADC value is bigger than the average after adding exactly this new value. If the ADC interrupt is called every instruction or every basic block, the average will contain redundant values (whose number is equal to the number of basic blocks between lines 6 and 10) and thus end in a program state that is impossible to reach in real executions.

In addition to the issue of infeasible paths, the interrupt model used by FIE hinders performance due to the interrupt-related path explosion. Consider the code snippet in Figure 3 which has an infinite loop in which the first 1000 iteration are idle, they simply wait until the peripherals settle down. With interrupts enabled, FIE will produce a new fork at each iteration which will cause path explosion well

```
1
2  int foo()
3  {
4    ...
5    //sleep until woken up by ADC
6    _bis_SR_register(LPM3_bits | GIE);
7    ...
8    if (x>5) f() else g();
9    //sleep again  until woken up by ADC
10   _bis_SR_register(LPM3_bits | GIE);
11   ...
12   if(val > avg)
13     h()
14 }
15
16 // ADC Interrupt
17 #pragma vector=ADC10_VECTOR
18 __interrupt void ADC10_ISR(void) {
19   ...
20   val = ADC10MEM0;
21   values[index++] = val;
22   avg = compute_avg(values)
23   ...
24 }
```

Figure 2: If interrupts are fired in every basic block, the average will contain an incorrect value.

```
1  while(1)
2  {
3    if(tick >= 1000) // Wait for settling down
4    {
5        acceleration = process_adc();
6        if(acceleration == 20)
7          assert(0);
8    }
9    tick++;
10 }
```

Figure 3: Wait loops create too many interrupt forks.

before the analysis of actual code starts. This means that FIE cannot handle even some simple firmwares.

To overcome these limitations, we added a new interrupt model in which timer-based interrupts can be fired each time the firmware goes into the low-power interrupt-enabled mode. While this may miss some paths that would arise in practice, it is a reasonable starting heuristic since it often aligns directly with the developer's intention: put the microcontroller into sleep mode until an interrupt wakes it up. This offers significant advantages over firing interrupts every block or instruction as in [10]. Considering again the code snippet in Figure 2, if we fire a new interrupt only when the firmware goes into sleep mode, the average value will contain the correct symbolic expression. In the code snippet in Figure 3, no unnecessary interrupts will be produced.

MSP430 allows software to enable ADC conversion by setting appropriate bits in the ADC control registers. FIE's memory model, however, is stateless in the sense that it does not preserve previously written values and all subsequent reads will return a fresh symbolic value. In our implementation, we fixed this limitation so that special memory locations and registers can store symbolic values.

4. MODULAR DIRECTED SYMBOLIC EXECUTION

Both FIE and DrE's forward mode can be used to extract signal patterns for moderately complicated firmware, but it is inefficient when the conditions required to reach the target line are deeper in the call chain.

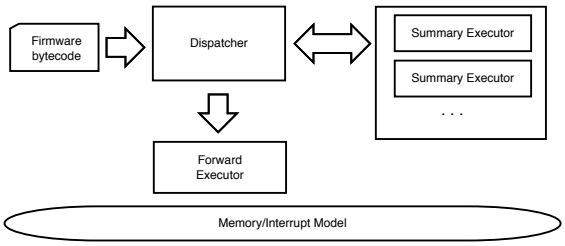


Figure 4: System components

For example, the firmware code for the AllSee gesture recognition system [18] includes a function that contains two loops and is responsible for obtaining amplitude samples of the ambient wireless signal; one loop iteration corresponds to one sample. In this concrete example, the simplest gesture requires at least 16 samples. If this code is analyzed using a basic breadth-first state selection strategy, the probability of reaching the program point responsible for recognition is 2^{-16}. With two invocations of the function (required for some gestures in AllSee), the probability becomes 2^{-32}. DrE, when used in its forward symbolic execution mode, also does not get beyond the point where the simplest gesture is recognized. This motivates the modular approach described in this section.

4.1 System overview

Figure 4 shows the architecture of our system. The firmware is compiled into LLVM bytecode. The bytecode is analyzed by the Dispatcher which first performs some basic static analysis: it searches for possible interprocedural paths from the program start to the target line using the program call graph and control flow graphs of individual functions (similar to the forward mode described in the previous section). For each function from this path, the Dispatcher starts collecting execution paths by running an instance of Summary Executor. Once enough are collected for all functions along the call path (this includes building function summaries for all inner calls, too) the Dispatcher tries to stitch them together and the final candidate interprocedural paths are checked by calling Forward Executor. Both Summary Executors and Forward Executor make use of the MSP430 memory and interrupt models.

This strategy has several attractive features. First, execution paths are collected on demand, starting from the bottom-most function and going all the way up to the entry function (including all inner function calls).

Second, when building execution paths for a function, we propagate path constraints on its arguments and return value from the execution paths of other functions, thus steering symbolic execution towards paths that are more likely to result in a feasible interprocedural path to the target line.

Third, we reuse already collected execution paths.

4.2 Example

Consider a simple example in Figure 5. We set line 5 in function `printGesture()` as the target line.

We start by building an interprocedural control flow path (which may turn out to be infeasible) from `main()` to the target line. There is one such path that starts in `main()`, then goes twice inside function `getGesture()`, and then into function `printGesture()`.

```
1
2  int printGesture(int g1, int g2)
3  {
4    if ((g1 == 1) && (g2 == 2))
5      assert(0);
6    ...
7  }
8
9  int getGesture()
10 {
11   ...
12   read_adc_values();
13   ...
14   // classify gesture
15   ...
16   return g;
17 }
18
19 int main()
20 {
21   int gesture1 = getGesture();
22   int gesture2 = getGesture();
23
24   if ((gesture2 != 3) && (gesture1 != 4))
25     printGesture(gesture1, gesture2);
26 }
```

Figure 5: A simple example program

```
1 int main()
2 {
3   int gesture1 = <Γ1>;
4   int gesture2 = <Γ2>;
5
6   if((gesture2 != 3) && (gesture1 != 4))
7     printGesture(gesture1, gesture2);
8 }
```

Figure 6: A version of `main()` (Figure 5) modified to replace function calls with symbolic variables.

Next, we collect execution paths for `printGesture()` with the goal of finding those that call `assert()`. We replace formal arguments `g1` and `g2` with new symbolic values γ_1 and γ_2. One of the execution paths w_1 with path constraints

$$\mathcal{C}_1 = (\gamma_1 = 1 \wedge \gamma_2 = 2)$$

goes through the assertion statement, so we save this execution path along with its constraints.

Next, we collect execution paths for `main()`. We are interested in paths that call `printGesture()` and satisfy conditions $\mathcal{C}_1$ on its arguments. Note that `main()` has two calls to `getGesture()`. Because of this, we replace these function calls with new symbolic variables and re-write `main()` as shown in Figure 6.

We proceed with collecting execution paths for `main()` and find execution paths w_2 with path constraints

$$\mathcal{C}_2 = (\Gamma_1 \neq 3 \wedge \Gamma_2 \neq 4).$$

Our goal is to "glue" together paths w_1 and w_2. Therefore, we use the following rewriting rule: $(\Gamma_1 = \gamma_1 \ \wedge \ \Gamma_2 = \gamma_2)$ and check $(\mathfrak{C} := \mathcal{C}_1 \wedge \mathcal{C}_2)$ (which is satisfiable and thus we "glue" paths w_1 and w_2):

$$\mathfrak{C} := (\Gamma_1 \neq 3 \ \wedge \ \Gamma_2 \neq 4 \ \wedge \ \Gamma_1 = 1 \ \wedge \ \Gamma_2 = 2)$$

The final step is to collect execution paths for `getGesture()` and find those that would not violate $\mathfrak{C}$. We denote the return value of function `getGesture()` as r and rewrite $\mathfrak{C}$ using the following rules: for the first call we use $(\Gamma_1 = r)$, for the second call we use $(\Gamma_2 = r)$. As the result, we are interested in execution paths that have the following constraints on the return value:

$$\mathfrak{C}_1 := (r \neq 3 \ \wedge \ \Gamma_2 \neq 4 \ \wedge \ r = 1 \ \wedge \ \Gamma_2 = 2)$$

in which Γ_2 is a free variable. And

$$\mathfrak{C}_2 := (\Gamma_1 \neq 3 \ \wedge \ r \neq 4 \ \wedge \ \Gamma_1 = 1 \ \wedge \ r = 2)$$

in which Γ_1 is a free variable.

Finally, we follow the execution paths that our analysis was able to glue together. This resolves the previously undefined calling context.

Algorithm 1: Collecting execution paths

```
1  CollectPaths(f, C);
   input  : f - function to execute.
   input  : C - set of constraints.
   output: EP_f - set of f's execution paths.
2  makeArgsSymbolic(s_init);
3  refreshGlobals(s_init);
4  EP_f ← ∅;
5  S ← {s_init};
6  while S ≠ ∅ do
7  |  s0 ← selectState(S);
8  |  if mergeState(s0, S) then continue;
9  |  SS ← executeInstruction(s0);
10 |  S ← S ∪ SS;
11 |  if isTarget(s0) ∨ isRet(s0) then
12 |  |  EP_f ← EP_f ∪ {path(s0)};
13 |  |  w = path(s0);
14 |  |  C' = rewriteConstraints(C, w);
15 |  |  if isSolvable(C') then
16 |  |  |  pause execution and return EP_f
17 |  |  end
18 |  end
19 |  if isCallIntruction(s0) then ConstructInnerCall(s0);
20 end
```

4.3 Collecting execution paths

In DrE, during the first phase, each function along the call chain is executed separately and in isolation. The corresponding pseudocode is shown in Algorithm 1. We are interested in collecting execution paths that satisfy specific conditions on the function's formal arguments and return value and that end up calling the next function in the chosen call chain.

We start by constructing the initial execution state[2] and adding it to the set of active states $\mathcal{S}$ (lines 4–5). Because the future context in which function f will be called is unknown, we set function f's arguments to symbolic values and assign fresh symbolic expressions to global variables (lines 2–3).

In the main loop, a state is chosen from $\mathcal{S}$ according to a state selection heuristic. Line 8 checks if the state has similar memory and constraints configuration to some previously seen state.[3] If a match is found, the current state is merged with the matching state. This state merging functionality allows us to eliminate redundant states, which arise frequently in embedded firmware (e.g., due to waiting loops).

The system then symbolically executes the next instruction. If the executor encounters a conditional statement, it queries the underlying STP solver, and if both branches are

[2]An execution state includes program counter, stack frames, and global memory (heap, global variables, etc.).

[3]The executor keeps a list of memory configurations of all previously seen states.

Algorithm 2: Dispatcher main loop

```
1  DispatcherMainLoop(target_line);
   input  : target_line - line of code to reach.
   output: O - set of constraints on the firmware input that would
             drive the firmware to the target_line.
2  call_chain ← pick_call_chain();
3  candidates ← ∅;
4  repeat
5      f ← pickLastFunction(call_chain);
6      EP_f ← CollectPaths(f,{target = target_line});
7      foreach w ∈ EP_f do
8          if path w does not end at the target line then
9              continue
10         end
11         stitchInnerCalls(w, ∅);
12         stitchNextInCallchain(w, ∅);
13         put(candidates, w);
14     end
15     foreach c ∈ candidates do
16         if (forward_check(c)) then
17             O ← O ∪ {c};
18             print c;
19         end
20     end
21 until call_chain_switch_timeout;
```

possible the state is forked and the newly produced states are added to the list $\mathcal{S}$ of active states (lines 9–10).

If the state's instruction is a return instruction or a call to the next function in the call chain, the current execution path is stored in $\mathcal{EP}_f$ (lines 11–17). In lines 14–17, we check if the execution path satisfies the conditions, in which case the executor pauses, saves its current configuration, and returns the set of collected execution paths to the Dispatcher.

Special semantics for call instructions. In order to increase scalability of our system, we implemented a special semantics for call instructions. Line 19 describes this behavior. Whenever the executor encounters a call to function f_{inner}, it does not expand this call immediately. Instead, we create a new symbolic variable representing the result of the call and proceed with the next instruction. As global variables may have changed in that function call, we assign them new fresh symbolic values (we will resolve this approximation at a later stage). At the same time, we extract conditions on f_{inner}'s actual arguments from the current state's constraints and save them for the future. These conditions will be used for stitching function summaries.

Furthermore, we call the underlying interrupt handling model in order to check if an interrupt can fire at the current instruction. If an interrupt can fire, we refresh the global variables and proceed with the execution of f.

4.4 Dispatcher main loop

The Dispatcher is the central component of the system. It schedules the process of collecting execution paths and stitches them together. Its main executing loop is shown in Algorithm 2. It starts at line 2 by choosing a *call path*, i.e., a call sequence ($f_n \leftarrow ... \leftarrow f_1 \leftarrow f_0$) from the target line to the entry point. The first function in the call path is function f_n containing the target line. The dispatcher then (lines 5–6) picks f_n and starts collecting f_n's execution paths that end up calling the target line.

Once execution paths are collected, the Dispatcher analyzes them individually (lines 7–20). Assume path w reached the target line. At this point (see Algorithm 1), w may have some call instructions that were replaced by symbolic values. The Dispatcher calls recursive function stitchInnerCalls() (line 11) to try to find proper execution paths for each of them (see the next section for the details). Thus w becomes linked to the execution paths of inner calls.

The Dispatcher then tries to find execution paths of the next upper-level functions $f_{n-1}, f_{n-2}, ...$ that it can stitch with w by calling function stitchNextInCallchain() (line 12). This function tries to find executions path of the next functions $f_{n-1}, ...$ in the call chain that can be stitched with w. It goes through and finds execution paths for each function in $(f_{n-1}, ..., f_0)$ all the way toward the program's entry point; it also accumulates the constraints of the already stitched execution paths. As recursion ends by reaching the top-level function f_0, a complete "stitched" candidate path c is returned. It is added to the *candidates* list.

Since full memory context was unknown at the time when execution paths were collected, candidate full paths might be infeasible. The Dispatcher thus makes another forward pass along this path using Forward Executor (lines 15–20). Forward Executor implements a simple state selection strategy that follows the provided interprocedural execution path.

If none of the candidate interprocedural paths turn out to be feasible, the Dispatcher proceeds to the next iteration of its main loop. New execution paths are collected, stitched, and checked by the Forward Executor. After a timeout, another call path is tried.

4.5 Stitching execution paths

DrE "stitches" execution paths of different functions based on the constraints on formal/actual arguments and constraints on return values. In this subsection, for clarity of exposition we use the same symbol w_k to denote both the execution path and the corresponding path constraints.

Connecting call-chain functions. Consider two functions which belong to the call chain f_{next} and f_{prev} (f_{next} calls f_{prev}) and their execution paths w_{prev} and w_{next}; w_{prev} has already been stitched to some execution paths ($w_n \leftarrow \ldots \leftarrow w_{prev}$) of the previous functions in the call chain.

Algorithm 3 checks if w_{prev} and w_{next} can be stitched. Informally, this algorithm recursively rewrites all occurrences of formal arguments in $(w_n, \ldots, w_{prev})$ with the corresponding actual arguments from w_{next} and checks if the conjunction $(w_n \wedge \ldots \wedge w_{prev} \wedge w_{next})$ is solvable. In lines 2–3, it chooses the next function from the call chain and collects its execution paths. In lines 4–6, for each collected execution path w_{next}, it replaces formal arguments in w_{prev}'s path constraints C with the actual arguments from w_{next} and checks if they are solvable. It then resolves all inner calls for w_{next} by calling stitchInnerCalls() and follows the recursion for the next function in the call chain in line 8. It then links w_{prev} and w_{next} in line 9.

Executing and stitching inner calls. Consider a function's execution path w (with accumulated constraints C, which include all constraints starting from the target function f_n) with calls that were replaced by symbolic values. The Dispatcher tries to find execution paths for each such inner call f_{inner}. It runs an instance of Summary Executor (according to Algorithm 1) to collect a set of execution paths $\{w_{inner}\}$. The dispatcher then tries to stitch them with w using algorithm 4. Informally, it (1) rewrites $\{w_{inner}\}$'s formal arguments with the actual arguments from w; and (2) replaces all occurrences of previously unconstrained sym-

Algorithm 3: Stitching call-chain functions

```
1  stitchNextInCallChain(w_prev, C);
   input : w_prev - execution path of the previous function f_prev
           in the call chain
   input : C - accumulated constraints from bottom-level function
           up to f_prev
2  f_next ← pickNextFunction(call_chain);
3  EP_{f_next} ← CollectPaths(f_next, C);
4  foreach w_next ∈ EP_{f_next} do
5      C' ← rewriteConstraints(C, w_next);
6      if isSolvable(C') then
7          stitchInnerCalls(w_next, C');
8          stitchNextInCallchain(w'_next, C');
9          link(w_prev, w_next);
10     end
11 end
```

Algorithm 4: Stitching inner calls

```
1  stitchInnerCalls(w, C);
   input : w - execution path containing inner calls.
   input : C - accumulated set of constraints from the call chain
           function down to this execution path
2  foreach f ∈ inner_calls(w) do
3      EP_f ← CollectPaths(f, C);
4      foreach w_inner ∈ EP_f do
5          C' ← rewriteConstraints(C, w_inner);
6          if isSolvable(C') then
7              stitchInnerCalls(w_inner, C');
8              link(w, w_inner)
9          end
10     end
11 end
```

bolic values in C with the actual return values from w_{inner} and checks if the resulting conditions are solvable. The rewriting of function arguments and return values happens recursively if w_{inner} has inner calls itself (line 7).

4.6 Limitations

Under-defined calling context. In our modular approach, functions are executed independently within an under-defined calling context. Constraints on global variables may thus be lost, causing extra states that would not have been created had the program been symbolically executed in a normal (forward) way. To compensate for this, DrE, upon executing a function f, recursively identifies global variables in f that are also used in f's callers. Only such global variables are assigned fresh unconstrained symbolic values.

Similarly, global variables can change after f calls some function g (because we delay executing inner calls and immediately step over to the next instruction). DrE assigns fresh unconstrained symbolic values only to those variables that are also used by g and its callees.

Pointers. Global pointers and function arguments of pointer type which would be pointing to valid memory locations during normal execution might become invalid when dereferenced. To tackle this problem, we adopted the following approach. Every time a new function is executed, for each pointer used by the function (global pointers and pointers that appear in the formal arguments list) we allocate a symbolic array of fixed size (64 bytes by default). We found that this limit usually holds for MSP430 firmware and does not lead to false overflows. Our current prototype does not support pointers to pointers, however.

Allocating new symbolic arrays for (undefined) pointers and assigning new symbolic values for global variables and formal arguments results in paths that might turn out to be infeasible (i.e., when the execution runs from the program's entry point). Many infeasible paths are eliminated during the path stitching procedure when calling context is partially defined. The remaining infeasible paths are eliminated during the final forward pass in which the calling context is fully defined. Therefore, checking infeasible interprocedural paths can make finding feasible interprocedural paths slower (in one case significantly—see discussion in the next section) but does not impact the soundness of the analysis.

5. EXPERIMENTAL RESULTS

In this section, we compare DrE's efficiency with FIE [10]. We first use two synthetic examples to explain FIE's limitations, then test DrE on several real-world firmwares, and finally use DrE to extract signal patterns from the AllSee gesture recognition system [18].

All experiments were run on a machine with four Intel Core i7 3.07 GHz cores) and 24 Gb of memory. Table 1 shows the results. The run time of each experiment was at most one hour; '-' means that the analysis tool failed to reach the target line within one hour. We ran DrE in two modes: (1) compositional, or modular, mode in which functions are executed in isolation and then stitched; and (2) forward mode in which no execution paths are pre-collected and the program is executed top-to-bottom.

5.1 Synthetic examples

The first example (Figure 8 in the Appendix) features an infinite loop in which the first 1,000 iterations are idly waiting until the peripherals settle down. FIE was not able to find the target line due to state explosion: a new fork was generated with each basic block during the first idle loop iterations. In contrast, DrE's interrupt model allowed it to find the target line in under one minute. While both DrE modes were able to find the target line, forward mode was significantly faster; this is due to the fact the program control logic and its call graph for the first example are quite simple and thus do not benefit from modular execution.

The second example (Figure 9 in the Appendix) demonstrates the problem of path explosion caused by sensor input processing logic. The main function contains an infinite `while` loop that assigns values to a three-element array based on the sequence of ADC readings. If all three elements are assigned a fixed value, the target line of code is executed. The problem here is that the function responsible for processing the ADC inputs contains a loop which should be executed at least 15 times in order to return the right value. This function should be executed at least 3 times which results in 2^{45} possible states. FIE failed on this example. In less then 10 minutes it generated more than 4,000 states.[4] DrE's forward mode was also not able to reach the target line due to a large number of states. In contrast, it took DrE's compositional mode less then 20 seconds to produce the sensor input values required to reach the target line: DrE decomposed the program into separate functions

[4]In addition to path explosion, we also found an implementation limitation in FIE: it shares one common 16-bit address space for all execution states, which puts a limit on the number of states. This resulted in FIE halting after it generated about 4,000 states.

	FIE	DrE	
		Forward	Compositional
Synthetic 1	-	11s	43s
Synthetic 2	-	-	20s
softmodem-for-msp430*	-	-	35s
FreeStanding	-	8s	-
mbtronics_temperature	31m	17m	27s
mrothe_temp	-	8m	13s

Table 1: DrE vs. FIE. "-" means that the target line was not reached within one hour.

and inferred the proper conditions on their arguments and return values. This modular strategy quickly eliminates a significant fraction of the irrelevant paths.

5.2 Sensor firmware

Despite the fact that DrE is still a research prototype, it performed well on several real-world firmwares. For this set of experiments, we use MSP430 programs from the corpus used by [10]. This set includes projects of different complexity downloaded from the Internet. Some of them consist of no more than 2 functions. Both DrE and FIE performed well on those examples and in many cases achieved full code coverage. We thus chose several non-trivial firmware programs for testing DrE. We manually selected target points that involve loops and non-trivial control logic.

softmodem-for-msp430.[5] This firmware is an implementation of the ITU V.21 modem specification (for use in a general switched telephone network). The firmware consists of 38 functions and 1,188 lines of code. It starts with some initialization code (to set up the basic clock system, set up modem functions, and enable analog-to-digital (ADC) conversion). We set the first line after the initialization code as the target line. We performed our experiments on a modified version of this firmware. The original code contains two idle "for" loops (one line of code each) with a large number of iterations to settle the oscillator ADC reference voltage, which frustrated both DrE and FIE. These loops can be easily identified and eliminated by static code analysis.

The rest of the initialization code is rather complex and involves multiple loops (including an infinite one, with a specific break-out condition) and inner calls with depth of five. This cause FIE to get stuck in the initialization code. DrE was able to find a path through the initialization code in 35 seconds and avoided path explosion by discovering the specific condition for breaking out of the infinite loop.

mbtronics_temperature.[6] This firmware continuously samples an ADC connected to a temperature sensor, compares the result against a pre-specified value and sets PWM (Pulse Width Modulation) based on the measured ADC offset. It then transmits temperature values via UART to the PC. The firmware contains 377 lines of code and 14 functions. It starts with initialization code and then enters an infinite loop (with several inner calls and branch instructions). The loop needs to be executed 8 times to refresh the state and start a new sequence of temperature measurements. While FIE reached this refresh in 31 minutes, DrE's compositional mode reached that part of the code in 27 seconds. DrE's forward mode reached the same code in 17 minutes.

[5]https://github.com/dyno/softmodem-for-msp430

[6]https://github.com/mbtronics/msp430_projects

We also considered another variant of this firmware (named mrothe_temp) found in the firmware set. It shares most of the code with *mbtronics_temperature* but includes some additional logic. FIE was not able to find a path to the target line, while DrE reached it in 13 seconds and 8 minutes in compositional and forward modes, respectively.

FreeStanding.[7] This firmware uses an accelerometer connected to the ADC port to measure the time spent in a handstand position by the person wearing a device. The firmware consists of 7 functions and 318 lines of code. In spite of the firmware's seeming simplicity, FIE was unable to reach at least one part of the code responsible for control logic within one hour. DrE's compositional mode did not perform well in this case either due to extensive use of global variables throughout different functions (see Section 4.6). In contrast, DrE's forward mode with the shortest distance search strategy found a path to the relevant parts of the code in 8 seconds.

5.3 AllSee gesture recognition system

As a concrete target for a signal spoofing attack, we chose an MSP430-based gesture recognition system called AllSee [18] which we believe is a good example of an embedded system that makes decisions based on sensing its environment. AllSee attempts to recognize hand gestures by extracting information from ambient wireless signals (for example, TV or Wi-Fi).

AllSee aims for extremely low power consumption. It uses a combination of custom-built sensor hardware (an envelope detector that extracts low-frequency amplitude changes from the ambient RF signal) and an MSP430 controller that implements gesture recognition logic. The envelope detector is connected to the controller through the analog-to-digital converter (ADC) port. The MSP430 firmware has a set of 8 hard-coded amplitude change patterns that correspond to 8 different hand gestures. Once the received sequence of values matches one of the patterns, the firmware can relay this command to higher-level software (in the proof of concept, it simply prints the name of the recognized gesture over the UART interface).

An important parameter of the AllSee classification logic is the minimal amplitude change threshold. This is used as a threshold for detecting when a significant change in amplitude has occurred. The value is hardcoded and presumably was chosen based on the sensitivity of the envelope detector and observed signal attenuation caused by hand gestures. Note that AllSee's logic recognizes not the specific amplitude changes but only the *coarse shape* of the signal (up/down/-down, etc.) As long as the coarse shape is preserved, the spoofed signal will be recognized. This makes the attacker's life easier, as he does not need to reproduce an exact signal but only to mimic its shape.

AllSee's source code [17] is available under a public copyright license. It consists of about 300 lines of code but includes infinite loops and complex logic that frustrate non-directed symbolic execution tools, as we show below.

We used DrE's compositional mode to extract gesture patterns from the AllSee firmware. In Table 2, we compare DrE's efficiency (both compositional and forward modes) with FIE. In this case, "-" means that the pattern for the gesture was not found within 36 hours.

[7]https://github.com/johnhowe/FreeStanding

	FIE	DrE	
		Forward	Compositional
Push	-	-	53s
Pull	-	-	4m 37s
Flick	-	-	2h 8m
Zoom out	-	28m	51s
Zoom in	-	-	43s
Punch	-	-	2m 21s
Reverse Punch	-	-	2m
Double Flick	-	13m	34s

Table 2: Gesture pattern extraction times

FIE failed to reach the target lines in all cases. There are two main reasons for this. AllSee's code includes a function that contains two loops and is responsible for obtaining amplitude samples of the ambient wireless signal; one loop iteration corresponds to one sample. Recognizing a gesture involves executing the loop 16 times in a row (once it deviates, the counter drops to zero). The firmware invokes this function at least twice, which results in path explosion. The second reason is that AllSee computes a moving average over the last 64 ADC values. We observed that when the number of ADC values is large, it can take tens of seconds for the STP solver to solve a single formula in the path constraints.

DrE's forward mode was able to extract patterns for two of the gestures. Its state selection strategy gives preference to states that are closer to the target line. Therefore, once the execution breaks out of the first loop and enters the second loop, it keeps choosing states in the second loop forever.

DrE's compositional mode is much more efficient. It was able to extract signal patterns for 7 out of 8 gestures in less than five minutes. For the last gesture, however, it took a bit more than two hours. In the latter case, the tool spent most of the time in the last forward pass verifying different combinations of obtained execution paths. Because the code computes the moving average over the last 64 ADC values, most of the time was spent in the STP solver.

Extracted patterns. The extracted patterns of signal amplitude changes (read from ADC after sampling) corresponding to different gestures are shown side by side with the original copies from [18] in Figures 10 and 11 in the Appendix. The patterns obtained via symbolic execution in most cases are simpler than the original ones. This could make spoofing attacks easier compared to the case when an attacker needs to reproduce the exact patterns from AllSee's description [18], which is an extra benefit of our approach (in addition to automated extraction).

6. SIGNAL SPOOFING

So far we obtained sequences of ADC values that make AllSee firmware recognize specific gestures. In this section, we report on proof-of-concept experiments that show how to trigger these ADC values by sending wireless signals such that the results of sampling by AllSee's sensor have the shapes shown in Figures 10 and 11 in the Appendix.

6.1 Experimental setup

Porting firmware. The original AllSee source code was written for the now obsolete MSP430F5310 launchpad, and we could not readily purchase this legacy launchpad from Texas Instruments. We therefore ported the firmware to the MSP430G2553 microcontroller. This involved different assignments to control registers (in order to properly set up the clock system, UART, etc.) We configured the ADC to measure voltage between 0 and 2.5 volts. We did not modify any of the core functions responsible for gesture recognition.

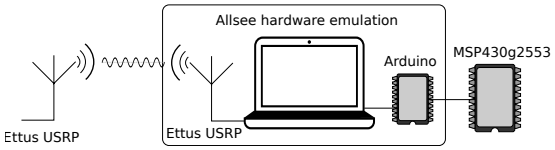


Figure 7: AllSee hardware emulation

Hardware emulation. In order to minimize overall power consumption, AllSee uses custom-built hardware: an envelope detector and low-pass filter to extract low-frequency amplitude changes from the carrier signal. Since we did not have this hardware, in our experiments we emulated it with software-defined radio. Our emulation configuration is shown in Figure 7. An Ettus USRP B210 [13] device equipped with an omnidirectional antenna is connected to a laptop with gnuradio [15]. Using emulation also allowed us to use a different carrier frequency of 903 MHz (which is at the beginning of ISM[8]) band. Corresponding gnuradio blocks are responsible for extracting the signal amplitude and removing high-frequency amplitude changes (we chose cutoff frequency of 100 Hz). The extracted samples are then converted back to voltage levels and sent to the MSP430 device through a digital-to-analog converter (we chose to use ArduinoDue [4] for that).

To reduce internal noise from the power supply and voltage reference of the A/D converter, we applied power supply decoupling as specified in [31]. We also added a low-pass filter on the ADC input (i.e., between Arduino DAC and MSP430). The resulting accuracy of ADC was approximately 2 bits.

In order to support a large number of devices, gnuradio abstracts the actual signal amplitude values; instead of concrete signal strength levels each transmitted/received sample is characterized by a unitless value in the range $[0; 1]$. These values are proportional to the actual amplitude, but getting the actual voltage requires calibrating the device.

We performed a series of gestures and measured the distortion in the background signal (in terms of gnuradio unitless values). We then took this value and matched it with the signal threshold (in terms of ADC steps) from the AllSee firmware, thus computing the correspondence between USRP/gnuradio unitless amplitude and ADC level on the MSP430 device. We believe that this calibration methodology is similar to how one would calibrate the original AllSee.

Once the hardware emulation was calibrated, we performed a series of gestures. While ultimately the shapes of the distorted background signals were consistent across repeats of the same gesture, only one (Flick) gesture had similar amplitude changes to that reported in [18]. Figure 12 in the Appendix shows the signal patterns for two gestures obtained through our emulation next to the patterns from [18]. We believe that the difference in shapes is caused by the difference in hardware configurations and environment condi-

[8]The industrial, scientific and medical (ISM) radio bands are radio bands reserved internationally for the use of radio frequency energy for industrial, scientific and medical purposes other than telecommunications.

tions. To have a "flick" recognized as such in our environment would require additional tuning of the firmware to the new conditions. Because we wanted to assess as unmodified an AllSee firmware as possible, we abstained from any additional modifications to the firmware.

Reduced sampling rate. The original firmware uses an ADC sampling rate of 200 samples per second. In the following experiments, we reduced it to 2 samples per second to facilitate synchronization between the emulated hardware and the MSP430 ADC port. These synchronization issues do not reflect a deficiency of DrE, but instead are the byproduct of our three-component hardware emulation and the added low-pass filter on the ADC input. They would not be present in the original non-emulated hardware, and so attacks would work at the original sampling rate.

Roughly speaking, in the case of 200 Hz sampling rate, the signal that the attacker will need to generate to ensure the firmware reads the same sequence of ADC outputs will have the same shape as in the case of 2 samples per second, but because it is transmitted faster, it will be "shrunk." For example, assume that DrE's output indicates that the second ADC reading should be larger than the first one. In the 2 Hz case, the amplitude should rise in approximately 0.5 seconds after the first reading. In the 200 Hz case, the amplitude should rise in approximately 0.005 seconds after the first reading.

6.2 Experiments

Once the ADC configuration is set, a signal pattern corresponds to specific voltage levels at the time the ADC reads a sample. In the first set of experiments, we applied voltage patterns produced by our symbolic execution tool to the ADC-enabled MSP430 pin directly. Each pattern triggered the corresponding gesture recognition in almost 100% of cases: due to internal ADC noise, a negligible number of gestures were not recognized correctly. We also observed sporadic voltage spikes that randomly triggered gesture recognition even though no pattern was sent.

In the second set of experiments, we carry out a complete signal spoofing attack and measure how accurately we can transmit signal patterns over the air. As an attacker we used another USRP B210 at 903 MHz. We carry out the experiments in the following setting: the attacker's USRP and the receiving antenna were placed 2 feet apart within line of sight of each other. This setting corresponds to use cases described in [18] (in which reported accuracies were greater than 90% at the distance of 2 feet).

In our experiments we were able to successfully spoof all 8 gestures, often by sending just one copy of the signal pattern. The chance (based on 100 experiments) that a *single* pattern is recognized as the corresponding gesture by AllSee is shown in the third column of Table 3. The second column shows the number of transmissions per signal pattern. The more transmissions the attacker needs to send, the longer (in milliseconds) it takes him to spoof a gesture. Note that in our experiments we reduced the sampling rate; it is possible (and expected) that the success rate will be lower for the original sampling rate of 200Hz.

We use the numbers from Table 3 to compute the time for an attacker to spoof a gesture, on average and with 80% and 90% quantiles. Before we describe the actual times, we need to take into account some of the AllSee control logic. In order to avoid random human motion near the device

Gesture	# of samples in pattern	Success %
Flick	148	39%
Push	68	63%
Pull	100	56%
Double Flick	36	81%
Punch	84	58%
Reverse Punch	84	66%
Zoom in	52	77%
Zoom out	52	79%

Table 3: Percentage of 100 transmissions of each spoofed pattern that successfully triggered recognition of the associated gesture. The second column is the number of transmissions that must be sent for that pattern.

Simple Gesture	Average	80% quantile	95% quantile
Flick	184	241	448
Push	54	55	102
Pull	89	98	182
Double Flick	22	17	32
Punch	72	78	145
Reverse Punch	64	63	117
Zoom in	34	28	53
Zoom out	33	27	50
Gesture with guard	**Average**	**80% quantile**	**95% quantile**
Flick	234	390	726
Push	67	117	218
Pull	110	181	337
Double Flick	27	54	101
Punch	89	152	283
Reverse Punch	78	126	235
Zoom in	41	72	135
Zoom out	40	69	129

Table 4: Time (in seconds) before a gesture is successfully spoofed assuming 2 samples per second. Upper part: single gestures. Lower part: combined with guard gesture.

from being classified as the target gestures, AllSee uses the double flick gesture as a "guard" that must be performed for AllSee to start accepting subsequent gesture commands. This means that forcing a gesture to be recognized actually requires spoofing two sub-gestures.

Table 4 shows the time (in seconds) for a gesture to be successfully spoofed at sampling rate of 2 samples/sec. The upper part of the table shows the times without the double-flick preamble. The lower part corresponds to the case when each gesture must be preceded by the double-flick preamble. For 7 gestures out of 8, it takes less then 90 seconds on average to be spoofed. If the double-flick preamble is required, for 7 gestures out of 8 it takes less then 110 seconds to be spoofed on the average. For the original sampling rate of 200 samples/sec, assuming the same probabilities as in Table 3, gesture spoofing would be 100 times faster: it would take less then 3.5 seconds including double-flick preamble to spoof 7 gestures out of 8 with 95% probability.

Finally we set the distance between the attacker and AllSee to six feet and had people walk occasionally in the vicinity. Even in this environment, for which AllSee was not designed, we were able to successfully spoof four of the eight gestures.

7. RELATED WORK

FIE [10] is an existing tool for the symbolic execution of MSP430 firmware source code that built on an earlier x86 system, KLEE [8]. In our experiments, FIE often produced

a large number of paths that are, in fact, infeasible with properly functioning hardware. It also omits important (and sometimes the only reachable) paths. Overall FIE's explicit design goal of exhaustively exploring all code paths results in shallow code coverage, making it hard for FIE to reach points of interest in non-trivial firmware.

Modular approaches for (non-directed) symbolic execution appeared in several previous papers. In compositional symbolic execution [16], a set of execution paths of a function can be represented as a function summary. In [16], function summaries are built using the top-to-bottom approach. This approach is well-suited for increasing code coverage but it can take a long time to reach a specific point in the code (e.g., if it is reached only when a rare condition is satisfied). A demand-driven variant of this approach [3] inherits the same problems. In this paper, we develop a bottom-up approach instead. This allows us to capture the constraints defining reachability of the target program point early in the analysis. We use the final forward pass, after stitching function summaries, to eliminate infeasible paths.

Under-constrained symbolic execution [12, 26] checks single functions. Executing functions in isolation is part of our approach. Because we deal with line reachability problem, we developed a method for propagating constraints between execution paths of different functions and for stitching these paths into interprocedural paths from the program's entry point to the target line.

Similar to our paper, [23] uses symbolic execution to tackle line reachability problems. In [23], the execution starts at the target function, with function parameters and global variables replaced by symbolic values.

The modular approach in [23] is rather coarse-grained. When the execution encounters a call to a function that does not belong to the shortest call chain, it does not try to summarize it but instead steps into this function and continues normally. This results in (1) a long chain of inner calls which causes path explosion and makes it hard and/or unlikely to reach the target, spending most of the time on paths that do not reach the target; (2) the same function must be re-executed multiple times even when the calling context is the same. Our approach is more flexible. It re-uses previously analyzed inner calls and thus avoids re-executing the same execution paths. Another difference is that [23] checks whether two paths can be concatenated by executing them. We use a constraint propagation/rewrite subsystem instead to efficiently check if a chain of execution paths is feasible. In the common case when there are many paths to be checked, we round-robin between k shortest call chains.

Finally, all symbolic execution tools surveyed in this section [3, 12, 16, 23, 26] (except FiE [10]) lack appropriate interrupt and memory models to be applied to embedded firmware.

8. SENSOR SPOOFING LIMITATIONS

In this paper, we focused on systems where (a) control logic is implemented in software, and (b) it is possible to define unwanted (and thus wanted by the attacker) states.

There are also other kinds of sensor spoofing attacks. If control logic is implemented in hardware or a sensor-specific property is used for the attack, symbolic execution of the firmware may not be very useful. For example, in [24] sensor saturation was used to generate fake drops in an infusion pump and cause change in the amount of injected fluids. Furthermore, many sensors simply produce binary or numeric values, in which case symbolic execution is not as important as physical spoofing itself. For other systems it may be challenging for the adversary to define the unwanted behavior of the target firmware.

In this paper we used AllSee as a running example and dealt with the relatively simple case of wireless signals. Sensor spoofing can be harder for other types of sensors, e.g., in [30] the authors used sound to cause resonation in drones' internal gyroscopes forcing the drones to crash.

9. CONCLUSION

In this paper we initiated the exploration of program analysis tools for synthesizing sensor spoofing attacks. Specifically, we show how an adversary can use program analysis of a device's source code to find sensor inputs that cause the program to "believe" that it received a certain signal. We translated the problem of signal pattern extraction into a line reachability problem—which is well-suited for symbolic execution. We then presented a new symbolic execution tool, DrE, that aims to solve this problem for the MSP430 family of microcontrollers.

DrE uses a combination of traditional static analysis techniques and a bottom-up variant of directed compositional symbolic execution. By combining control flow and call graph information with interprocedural data dependencies collected during symbolic execution, DrE is able to filter out a significant number of infeasible paths and generate conditions on the environmental inputs that cause desired behavior in the MSP430 firmware.

As a case study we applied DrE to the AllSee gesture recognition system and were able to extract signal patterns for all of its gestures. We then manually converted digital signal patterns into physical ones using the ADC sensor specification. This resulted in a complete, end-to-end proof-of-concept signal spoofing attack against the AllSee firmware when run with emulated sensor hardware. The attack enables the attacker to control AllSee-equipped devices without their owner's knowledge or consent. In the ideal "noise-free" environment, we were able to successfully spoof all eight gestures.

We believe that our techniques will generalize to other types of sensor-based systems and physical modalities, particularly when converting from digital sensor readings back to physical signals is straightforward (as was the case with AllSee). Examples may include light, sound, and other sensors. Converting the digital patterns provided by program analysis into a physical signal is still sensor-specific, and, in many cases, going to be much more challenging than the relatively simple case of wireless signals used by AllSee. We leave full exploration of such sensors to future work.

Acknowledgments. This work was partially supported by NSF grant CNS-1223396 and United States Army Research Office (ARO) grant W911NF-16-1-0145.

10. REFERENCES

[1] H. Abdelnasser, M. Youssef, and K. A. Harras. WiGest: A ubiquitous WiFi-based gesture recognition system. In *INFOCOM*, 2015.

[2] A. Alanis, T. Thai, G. Dejean, R. Gilad-Bachrach, and D. Lymberopoulos. 3D gesture recognition through RF sensing. Technical Report MSR-TR-2014-81, June 2014.

[3] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, 2008.

[4] Arduino Due. https://www.arduino.cc/en/Main/ArduinoBoardDue, 2016. [Online; accessed 16-May-2016].

[5] S. Asankhaya. Exploiting undefined behaviors for efficient symbolic execution. In *ICSE Companion*, 2014.

[6] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS*, 2008.

[7] A. Butler, S. Izadi, and S. Hodges. SideSight: Multi-"touch" interaction around small devices. In *UIST*, 2008.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[9] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou. Hidden voice commands. In *USENIX Security*, 2016.

[10] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, 2013.

[11] D. Davidson, H. Wu, R. Jellinek, T. Ristenpart, and V. Singh. Controlling UAVs with sensor input spoofing attacks. In *WOOT*, 2016.

[12] D. R. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA*, 2007.

[13] Ettus Research. USRP B210. https://www.ettus.com/product/details/UB210-KIT, 2016. [Online; accessed 16-May-2016].

[14] S. Evanczuk. Slideshow: The most-popular MCUs ever. www.edn.com/electrical-engineer-community/industry-blog/4419922/9/Slideshow--The-most-popular-MCUs-everKJ, 2013. [Online; accessed 16-May-2016].

[15] Gnuradio. The Free and Open Software Radio Ecosystem. http://gnuradio.org/, 2016. [Online; accessed 16-May-2016].

[16] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[17] B. Kellogg. The code used in AllSee. http://allsee.cs.washington.edu/#code. [Online; accessed 16-May-2016].

[18] B. Kellogg, V. Talla, and S. Gollakota. Bringing gesture recognition to all devices. In *NSDI*, 2014.

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[20] D. F. Kune, J. Backes, S. S. Clark, D. Kramer, M. Reynolds, K. Fu, Y. Kim, and W. Xu. Ghost talk: Mitigating EMI signal injection attacks against analog sensors. In *S&P*, 2013.

[21] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC*, 2010.

[22] S. G. Lee, Y. Diaz-Mercado, and M. Egerstedt. Multirobot control using time-varying density functions. *IEEE Transactions on Robotics*, 31(2):489–493, April 2015.

[23] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, 2011.

[24] Y. Park, Y. Son, H. Shin, D. Kim, and Y. Kim. This ain't your dose: Sensor spoofing attack on medical infusion pump. In *WOOT*, 2016.

[25] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and LiDAR. In *Blackhat Europe*, 2015.

[26] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security*, 2015.

[27] F. Saudel and J. Salwan. Triton: A dynamic symbolic execution framework. In *SSTIC*, 2015.

[28] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA*, 2009.

[29] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava. Non-invasive spoofing attacks for anti-lock braking systems. In *CHES*, 2013.

[30] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *USENIX Security*, 2015.

[31] Texas Instruments. MSP430x2xx Family. User's Guide. http://www.ti.com/lit/ug/slau144j/slau144j.pdf, 2013. [Online; accessed 16-May-2016].

[32] Texas Instruments. Applications for Low-power MCUs . http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/applications.page, 2016. [Online; accessed 16-May-2016].

[33] T. Vaidya, Y. Zhang, M. Sherr, and C. Shields. Cocaine noodles: Exploiting the gap between human and machine speech recognition. In *WOOT*, 2015.

[34] C. Yan, X. Wenyuan, and J. Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. In *DEF CON*, 2016.

APPENDIX

```c
#define THRESHOLD 15
int adc = 0;
int process_adc() {
  int val = 0, val_prev = 0, i = 0, diff = 0, j =
      0;
  __bis_SR_register(CPUOFF + GIE);
  val = adc;
  while(diff < THRESHOLD) {
    val = adc;
    diff = val-val_prev;
    i++;
    val_prev = val;
    __bis_SR_register(CPUOFF + GIE);
  }
  if(diff <= 20) return 20;
  if(diff <= 100) return 100;
  if(diff <= 300) return 300;
  return 500;
}

int main() {
  int val = 0, val_repv = 0;
  int tick = 0;
  int acceleration = 0;
  __bis_SR_register(CPUOFF + GIE);
  while(1) {
    if(tick >= 1000) { // Wait for settling down
        acceleration = process_adc();
        if(acceleration == 20) assert(0);
    }
    tick++;
  }

}

int num_of_adc_reads = 0;
void __attribute__ ((interrupt(ADC10_VECTOR)))
     ADC10_ISR (void) {
  adc = ADC10MEM;
  num_of_adc_reads++;
  __bic_SR_register_on_exit(CPUOFF);
}
```

Figure 8: Synthetic example 1

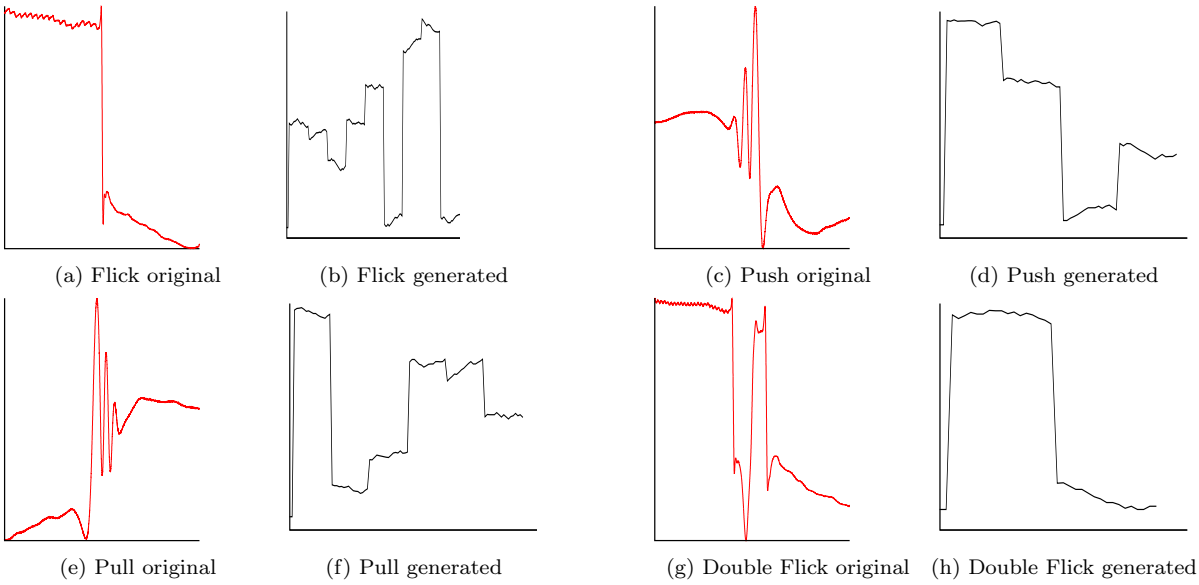
```c

#define THRESHOLD 15
int adc = 0;
get_acceleration(int mode) {
  int i = 0;
  __bis_SR_register(CPUOFF + GIE);
  while(adc >= 15) {
    i++;
    __bis_SR_register(CPUOFF + GIE);
  }
  if(i > 15 && (mode == 'a')) return 1;
  else if(mode == 'a') return 2;
  else return 3;
}

int main() {
  int val = 0, val_repv = 0;
  int tick = 0;
  int acceleration[] = {0, 0, 0};
  int j = 0;
  int mode;
  klee_make_symbolic(&mode, sizeof(mode), "mode");
  while(1) {
    acceleration[j] = get_acceleration(mode);
    j++;
    if(j==3) j=0;
    if( (acceleration[0] == 1) &&
        (acceleration[1] == 1) &&
        (acceleration[2] == 1))
      break;
  }
  assert(0);
  return 0;
}

int num_of_adc_reads = 0;
void __attribute__ ((interrupt(ADC10_VECTOR)))
     ADC10_ISR (void)
{
  adc = ADC10MEM;
  num_of_adc_reads++;
  __bic_SR_register_on_exit(CPUOFF);
}
```

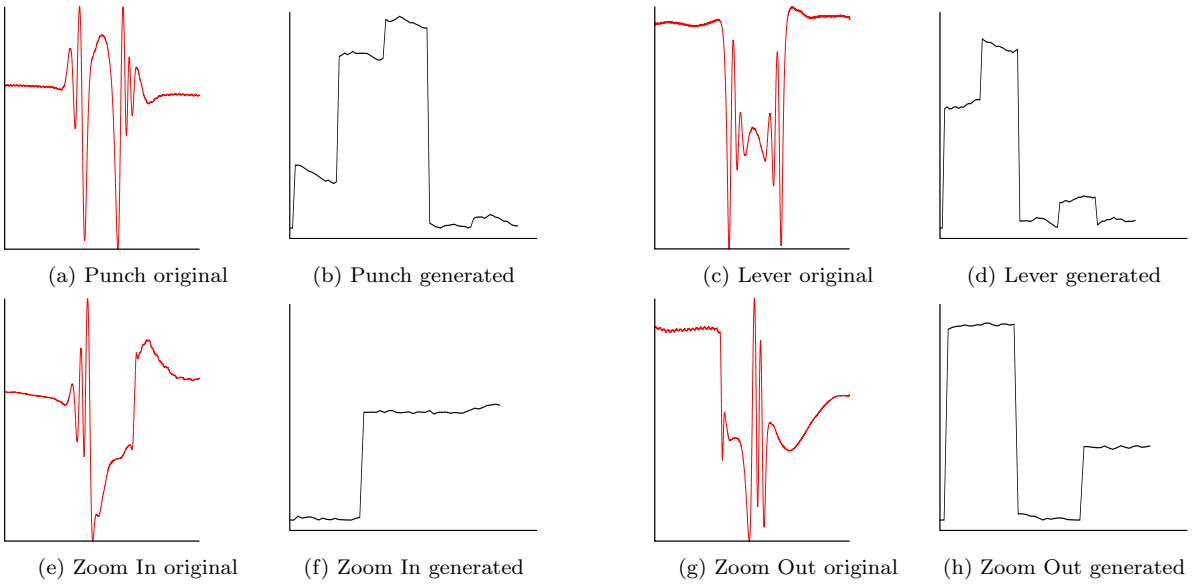
Figure 9: Synthetic example 2



(a) Flick original (b) Flick generated (c) Push original (d) Push generated

(e) Pull original (f) Pull generated (g) Double Flick original (h) Double Flick generated

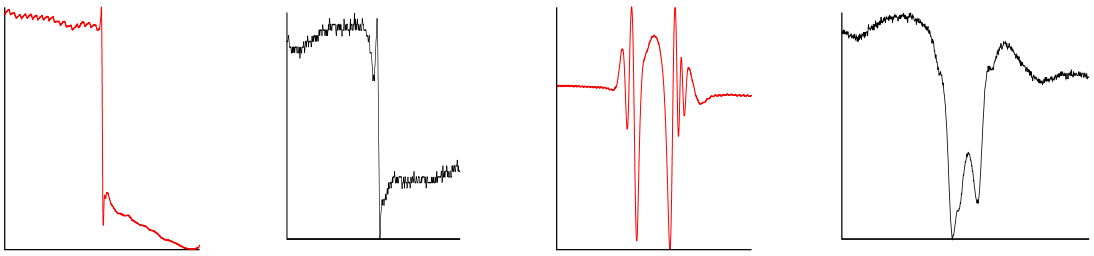
Figure 10: AllSee original (left) and generated by DrE (right) gesture patterns. Original gesture patterns were extracted from [18] unmodified.



(a) Punch original (b) Punch generated (c) Lever original (d) Lever generated

(e) Zoom In original (f) Zoom In generated (g) Zoom Out original (h) Zoom Out generated

Figure 11: AllSee original (left) and generated by DrE (right) gesture patterns (continuation). Original gesture patterns were extracted from [18] unmodified.



(a) Original (flick) (b) Emulation (flick) (c) Original (punch) (d) Emulation (punch)

Figure 12: Our emulation (right) vs AllSee original (left). Original gesture patterns were extracted from [18] unmodified.