Submitted for publication. Author copy - do not redistribute.

Interrupt-oriented Bugdoor Programming: A minimalist approach to bugdooring embedded systems firmware

Samuel Junjie Tan
Google, Inc.
samueltan@gmail.com

Sergey Bratus
Dartmouth College
sergey@cs.dartmouth.edu

Travis Goodspeed
Straw Hat
travis@radiantmachines.com

ABSTRACT

We demonstrate a simple set of interrupt-related vulnerability primitives that, despite being apparently innocuous, give attackers full control of a microcontroller platform. We then present a novel, minimalist approach to constructing deniable bugdoors for microcontroller firmware, and contrast this approach with the current focus of exploitation research on demonstrations of maximum computational power that malicious computation can achieve. Since the introduction of Return-oriented programming, an ever-increasing number of targets have been demonstrated to unintentionally yield Turing-complete computation environments to attackers controlling the target's various input channels, under ever more restrictive sets of limitations. Yet although modern OS defensive measures indeed require complex computations to bypass, this focus on maximum expressiveness of exploit programming models leads researchers to overlook other research directions for platforms that lack strong defensive measure but occur in mission-critical systems, namely, microcontrollers. In these systems, common exploiter goals such as sensitive code and data exfiltration or arbitrary code execution do not typically require complex computation; instead, a minimal computation is preferred and a simple set of vulnerability primitives typically suffices. We discuss examples of vulnerabilities and the new kinds of tools needed to avoid them in future firmware.

Categories and Subject Descriptors

K.6 [**Management of Computing and Information Systems**]: Miscellaneous; K.6.5 [**Management of Computing and Information Systems**]: Security and ProtectionUnauthorized access (e.g., hacking, phreaking)

General Terms

Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12...$15.00
http://dx.doi.org/10.1145/2664243.2664268

Keywords

ACSAC Proceedings, Security, Hacking, Microprocessor exploitation

1. INTRODUCTION

This paper examines what is arguably the most constrained "weird"[1] programming model to date: one using side-effects of interrupts. This model has, to the best of our knowledge, been overlooked. Although this model is weaker than any previous exploit programming models, it is significant because of the increasing ubiquity of microcontrollers and the ease with which this novel class of innocent-looking "bug doors" can be planted into firmware.

In this paper, we demonstrate this model on synthetic vulnerabilities, and provide prototype tools to help locate similar vulnerabilities on firmware in the wild, which we intend for future work. Moreover, we describe a proof-of-concept "bugdoor" implemented by making the interrupt handler of an MSP430 TinyOS-based firmware application re-entrant. This "bugdoor" is controlled solely by means of well-timed, attacker-triggered interrupts.

Bug doors.

Unlike a traditional backdoor, where specialized code is left in the program to surreptitiously control it, a bugdoor does not leave undeniably malicious code in the target program; it requires the attacker to supply specific inputs to the target to insert the malicious program (e.g. shellcode) and so perform unintended computation. The former type of vulnerability is easier to detect and trace to the source, as was the case with Alcatel's network switch operating system `telnet` backdoor in 2002 [10]. The latter is harder to discover and to definitively frame as deliberate exploitation.

With the building blocks featured in our synthetic vulnerabilities situated in firmware, an attacker would have a deniable bugdoor that involves no common forms of memory corruption and would therefore likely be missed by vulnerability scanning tools. Our prototype tools, on the other hand, show a basic set of tell-tale conditions associated with such hypothetical bugdoors.

"Two gadgets is all you need".

In the following two sections we put our programming model in historical perspective of exploitation programming. We then discuss the microcontroller threat model, the tar-

[1]In the sense of "weird machines", unexpected programming modes of trusted systems that we discuss later.

geted processor MSP430, and our "interrupt-oriented programming" model itself (a pun on ROP), starting with its primitives and continuing with the bug door that realizes it. We then describe our (inconclusive) attempts at finding such code in the wild, and wrap up with ideas for follow-on work.

2. MOTIVATION AND HISTORY: WHITHER EXPLOITATION?

2.1 Exploitation and its programming models

Research in exploitation techniques traditionally focused on feature-rich targets such as servers, desktops, and smart phones, where exploitation co-evolved with protective measures. Indeed, techniques such as return-to-libc [27], return-to-PLT [18], which generalized the reuse of code chunks by chaining stack frames pointing to them [20, 14], emerged in response to protective measures such as Openwall and PaX [28, 29] and co-evolved with such measures (e.g., [18, 8, 17]).

Interestingly, the more significant the barriers posed to attackers, the more expressive and general exploit programming models became. In 2000 [20] Gerardo Richarte observed that chunk chaining techniques could apparently encode "any program"; in 2007, Shacham Hovav [24] proved that x86 stack-based libc end-of-function block chaining was a Turing-complete programming model, at the same time coining the term *Return-Oriented Programming* (ROP) for such models. Since then, completeness has been shown to exist with less and less resources, such as with only certain classes of instructions [6], only constant-length instructions on platforms such as RISC [5]. Completeness has also been shown to exist when payloads are restricted to certain character sets [23]—even to just English dictionary words [15]—and when the payloads had to work on several ISAs at once [9].

This co-evolution highlighted the nature of exploits as *programs written for and assisted by certain combinations of features and bugs in the targets*—so-called primitives [21, 22]—used in a manner similar to assembly instructions (e.g., [13]). These unusual, emergent target machines and programming modes for exploit-programs became known as "weird machines" [4, 7, 2]. Typically, weird machines' existence was a surprise for the target's developers, but deliberate creation of them to provide a deliberate "debug door" or simply a "bug door" is also possible.

2.2 The distraction of Turing-completeness

Most weird machines were proved to be Turing-complete, following in the footsteps of [24]—partly because it pleases a certain kind of computer scientist to have a theorem like a cherry on top of a paper, and partly because other kinds of computer scientists saw offensive papers merely describing a novel exploit technique as incremental and unscientific. Arguably, the original proof of libc-based ROP completeness was necessary to change the academic view of exploits as limited and ad-hoc; it certainly made the great step forward of replacing the outmoded threat model of "malicious code" (and the unhelpful focus on detecting such code) with the more relevant model of malicious computation inherent in most general-purpose systems.

However, the computation performed by an exploit only needs to be powerful enough to achieve the exploiter's actual goals; there are no points awarded for additional expressive power, nor for extra complexity. The golden rule of exploitation is that when a thing can be reliably accomplished with minimal means, it should be accomplished with just such means. However, practical minimality has been overshadowed by pursuing techniques with maximum computational power. This disconnect is the starkest for *microcontrollers*, upon which our trusted and mission-critical systems continually increase their reliance.

2.3 Microcontroller exploitation

It was necessity that drove practical evolution of ROP and other techniques for feature-rich targets towards more expressive programming models. Essentially, protective measures forced exploits to emulate the OS' native linkers (cf. [3] on the "Bring Your Own Linker" pattern), which perform complex computations (e.g., GNU/Linux dynamic linker is a Turing-complete platform for its metadata considered as a program [25]). Accordingly, a rich set of primitives adding up to a computation model in which (nearly) arbitrary algorithms could be expressed helped in practical exploitation of such targets. In other words, computational power (up to Turing completeness) helped even when it wasn't strictly required.

For microcontrollers, the exploiters' situation is quite different. In absence of countermeasures, microcontroller vulnerability primitives need not provide a computationally rich environment; they just need to enable typical goals such as *exfiltration of firmware code*, *exfiltration of sensitive data such as cryptographic keys or configuration parameters* or *code execution in firmware contexts*. Exploit computation in microcontrollers need not be complex, and the primitives it requires to succeed may in fact be limited to a single write to a specific memory address or a single control flow redirection.

Microcontroller exploitation thus poses a different class of research problems, which, to the best of our knowledge, has been overlooked. Namely: what are the minimal combinations of primitives that still serve the attacker's goals (rather than, as in feature-rich targets, allowing the attacker to encode arbitrary algorithms)?

We can pose this question more precisely in the case of *bugdoors*, vulnerabilities deliberately introduced into a system to provide a deniable backdoor. What is a minimal patch that would provide such a backdoor and would be least likely to be spotted by bug-finding tools looking for memory corruption bugs?

3. RELATED WORK

Exploitation of computationally rich targets has a rich history, which was outlined above. The overall direction of this history has been doing more with less—essentially, keeping full control of the target and (Turing-)complete expressiveness of exploit programs with an ever-shrinking set of primitives. A good history sketch of primitives and attacks can be found in [16]. It turned out that many kinds of special-purpose data could also drive Turing-complete computations on the respective standard code that interpreted such data (e.g., [19, 25, 1]).

At the same time, for microcontrollers, the meaning of exploitation was different, reflecting the difference in threat models. In servers, etc., exploitation's typical goal is full control of the target ("root shell", for short); in microcon-

trollers, it is the exfiltration of firmware or data. For most microcontroller deployments, firmware is the important thing; once the attacker has the firmware, the rest of his goals are easily accomplished. This may or may not be due to firmware's general buggyness and lack of higher-grade protection such as NX memory or randomization; sometimes the firmware even contains passwords or private keys.

Goodspeed and Francillon demonstrated a good example of advanced microcontroller attacks in [11]. Their "half-blind" ROP attack circumvents protections of the MSP430, using only knowledge of the MSP430 Bootstrap Loader (BSL) code. The BSL is a small program present in ROM or protected flash of the MSP430 that allows developers to program the memory of the MSP430, typically for installing firmware updates. BSL operations that read memory are password protected in order to prevent unauthorized readout of code or data. In this attack, the entry point of a specific ROP gadget is guessed at random with a 1% success rate, and a buffer overflow is found by fuzzing. Given both this ROP gadget entry point and the buffer overflow vulnerability, a ROP payload can be deployed to circumvent protections in the BSL.

4. THREAT MODEL FOR MICROCONTROLLERS

The security architecture of microcontroller was designed with a specific threat model in mind. The secrecy of microcontroller firmware is crucial to prevent the production of counterfeit products with identical firmwares and theft of service through the alteration of firmware (e.g. changing power usage records on a "smart meter" to reduce electricity bills). To preserve the secrecy of firmware data while still allowing devices to receive firmware updates, microcontroller manufacturers typically use a bootloader and a combination of "fuses"—special write-once registers that contain permissions—to enforce access control on read/write operations on ROM.

An attacker targeting microcontrollers therefore aims to exfiltrate data, such as firmware, from the chip package. We assume that the attacker controls the processor, i.e., can slow down the clock, observe General Purpose Input/Output (GPIO), send signals to any pins, etc. However, we do not assume that the attacker possesses the capability to carry out sophisticated hardware attacks such as the one described in [12], where the integrated circuit board itself is penetrated to expose the device's circuits and allow data exfiltration through microprobing.

Our synthetic attacks assume that the attacker possesses some knowledge of the behavior of firmware, particularly that of the interrupt handlers. This scenario applies to common cases where the attacker has access to source code of older or similar versions of the target device firmware, but does not have access to the source code of the firmware running on the device itself. Given the fact that interrupt handlers are usually boilerplate code that are rarely changed between firmware versions, such a scenario is very feasible.

Our "bugdoor" attack assumes that the attacker can view and modify firmware source code, but not in a manner that is obviously malicious. This scenario applies to cases where the target microcontroller runs open source firmware, such as TinyOS, that an attacker could contribute his own modifications to, subject to peer review.

5. ROP, MEET "IOP"

The classic way to exploit a computing system is via its inputs. Any processed input is really a program; an exploit is merely a program with unexpected effects, format, composition modes, and, sometimes, unexpected input and output channels.

For our *Interrupt-oriented Programming* trick (IOP, a pun on ROP), we will use the fact that, other than data, a processor also takes **interrupts** as inputs. More specifically, the processor can be induced to execute interrupt handler code by events that can often be triggered externally by an attacker (e.g., sending an edge to a specific pin). In the same way that an a vulnerable target can be "programmed" by an attacker with a carefully crafted stack frame passed to it as input, we "program" a processor with a series of well-timed interrupts.

If our timings are precise enough to interrupt the processor while it is executing specific instructions, we can treat consecutive chunks of instructions starting at interrupt vector entry points and ending at another interrupt as our "gadgets", except that the starts of these "gadgets" are fixed, while the endings depend on our ability to deliver the interrupting interrupt at just the right instruction.[2]

Achieving such instruction-level timing precision on state-of-the-art server processors is clearly out of reach, since their high clock speeds make timing interrupts infeasible. However, this technique is feasible for microcontrollers that have far lower clock speeds and support downclocked, low-power processor states. For example, this technique will be applicable to the various, ubiquitous microcontroller-based sensors and "smart" devices, such as personal medical devices, industrial control systems, and "smart homes", on which we are getting increasingly reliant today.

Of course, interesting instructions should be found close enough to the beginning of interrupt vector chunks in the targeted firmware to be of use. It turns out that, with the minimalist primitives we discuss in section 7, we can plant an effective "bug door" there (which we describe in section 10).

Thus our objective is to explore programming with timed sequences of interrupt-triggering signals that would result in the execution of unintended, attacker-controlled computation. Formally speaking, our IOP programs are sequence of pairs (t_i, s_j), where $t_0, t_1, ..., t_{n-1}, t_n$ is the sequence of timings corresponding to CPU clock cycles, and $s_0, s_1, ..., s_{k-1}, s_k$ is the set of interrupt-triggering signals. t_0 is the time before the execution of the first instruction in the target program (clearly, timings t_i only make sense relative to a moment when the processor is in known state, such as its initial reset). For each pair pairs (t_i, s_j), the interrupt that signal s_j triggers must enabled at time t_i (see the discussion of interrupt-enabling bits below).

For this study, we looked at several MSP430-powered devices: EZ430U, EZ430URF, FET430UIF, MSP430FET, and GoodFET. We will describe this processor now, before we launch into IOP primitives.

6. TARGET PROCESSOR: TEXAS INSTRUMENTS MSP430

[2]In other words, our gadgets are almost the exact opposite of ROP ones; there needs to be nothing special about the instruction on which we break out of the gadget, no control flow semantics of RET or JMP.

The Texas Instruments (TI) MSP430 is a 16-bit ultra-low power microcontroller. The native MSP430 CPU implements a Reduced Instruction Set Computer (RISC) architecture with 27 instructions, 7 addressing modes, and 16-bit addressing. The extended MSP430X architecture supports 20-bit addressing, allowing it to directly address a 1-MB address range without paging. TI MSP430 microcontrollers draw low amounts of electric current in idle mode, and support several low-power modes. The MSP430 CPU can be downclocked to further reduce power consumption. More details about the features of the MSP430 family of microcontrollers can be found in [30].

The TI MSP430's low cost, low power consumption and power saving features has made it a popular choice for use in low power embedded devices, such as the Shimmer Wireless Sensor Platform [26] and the Bodymedia FITTM wearable medical devices (e.g., [34]).

For study, we specifically examined the MSP430F2618. The MSP430F2618 uses the MSP430X architecture, and is the embedded processor used in the GoodFET42, an open source JTAG adapter developed by Travis Goodspeed.

6.1 How interrupts work in the MSP430

There are two types of interrupts in the TI MSP430: maskable interrupts and non-maskable interrupts (NMIs).

Maskable interrupts are caused by peripherals with interrupt capabilities. Maskable interrupts can either be disabled individually by a per-peripheral interrupt enable bit, or all maskable interrupts can be disabled by the General Interrupt Enable (GIE) bit in the Status Register (SR). The SR is cleared when a maskable interrupt is received, thus clearing the GIE bit and disabling any further interrupts. Maskable interrupts can be nested only if the GIE bit is set inside of an interrupt handler.

NMIs cannot be disabled by the GIE bit. Instead, they are enabled by individual interrupt enable bits. When an NMI is accepted, all individual interrupt enable bits are reset, disabling any further NMIs. Software must manually set NMI enable bits to re-enable NMIs.

NMIs can only be generated from three sources: an edge on the RST/NMI (reset) pin, an oscillator fault, or an access violation to flash memory.

When an interrupt occurs, the program counter (PC) and SR are pushed onto the stack, interrupts are disabled, and the contents of the interrupt vector of the interrupt with the highest priority are loaded into PC so control flow is transferred to the interrupt handler. This takes 5 or 6 CPU cycles in the MSP430X and MSP430 CPUs respectively.

Interrupt handlers terminate with the `reti` instruction, which pops SR and PC off the stack to restore program state and execution context to the point where the interrupt was triggered. This takes 3 or 5 CPU cycles in the MSP430X and MSP430 CPUs respectively ([30]).

7. IOP PRIMITIVES

For any series of interrupts to affect the processor as we described above, there must exist some interdependency of state between the interrupt handlers' code, and some accumulation of state in interrupt handlers. We call such sequences of instructions *IOP primitives*. These primitives are "pre-vulns", in that they may, in certain combinations, lead to a vulnerability exploitable with IOP; however, by themselves they may well be innocuous and never triggered in common firmware execution patterns.

The following is a non-exhaustive list of IOP primitives:

- **State accumulation.** An instruction sequence that writes to a register/memory location without first setting it or restoring its original value afterwards. Such an instruction sequence would accumulate state in this register/memory location on each invocation of the interrupt handler.
- **Memory write.** An instruction sequence that writes an immediate value, register value, or value in memory to a register/memory location without overwriting the written value afterward. This could be used to perform arbitrary writes.
- **Arithmetic.** An instruction sequence that performs arithmetic on two values from registers or memory. Combined with a state accumulation and memory write primitives, such a primitive would allow arithmetic to be performed on arbitrary values stored in registers/memory and accumulated for later use.
- **Stack growing.** An instruction sequence that grows the stack (i.e., writes to memory pointed to by SP and decrements SP). This could be used to overwrite memory and to instrument return values on the stack. If this instruction could be invoked an arbitrary number of times (e.g. through interrupt nesting), this would be an *unbounded* primitive. Otherwise, it is *bounded*.
- **Stack alignment.** An instruction sequence that grows by a non-standard number of bytes. This could be used with a stack growing primitive to grow the stack to a word-aligned address. For example, if we had an unbounded stack growing primitive that grew the stack by 4 bytes (2 words) each time and a bounded stack alignment primitive that grew the stack by 2 bytes, invoking the stack alignment primitive after invoking the stack growing primitive a certain number of times would allow the attacker to decrement SP to any word-aligned address lower than the current SP.

Unlike ROP gadgets, which produce the same enduring effects whenever they are used, IOP primitives behave differently depending on the context they are invoked in, and produce effects that are sometimes ephemeral. For example, an interrupt handler containing a state accumulating instruction sequence and ending with an state reset instruction can only be used as a state-accumulation primitive if the accumulated state is used in some way before it is reset. These constraints make useful IOP primitives difficult to identify.

8. TOOLS AND SETUP

8.1 Hardware

In order to deliver the timed-signal payload to the target TI MSP430F2618, we needed to be able to observe the state of the microcontroller and send interrupt-triggering signals to its pins. To achieve that, we connected two other microcontrollers to the target device—one to communicate with the hardware debugging interface of the target MSP430F2618, and the other to send signals to the pins of that target. We

used a hardware debugging interface in our laboratory setup to ensure that our attack was behaving correctly. In an actual IOP attack, the attacker would not have full debugging access to the microcontroller, only the ability to observe some of the microcontroller's state and send signals to it.

Our experimental setup can be found in Figure 1.

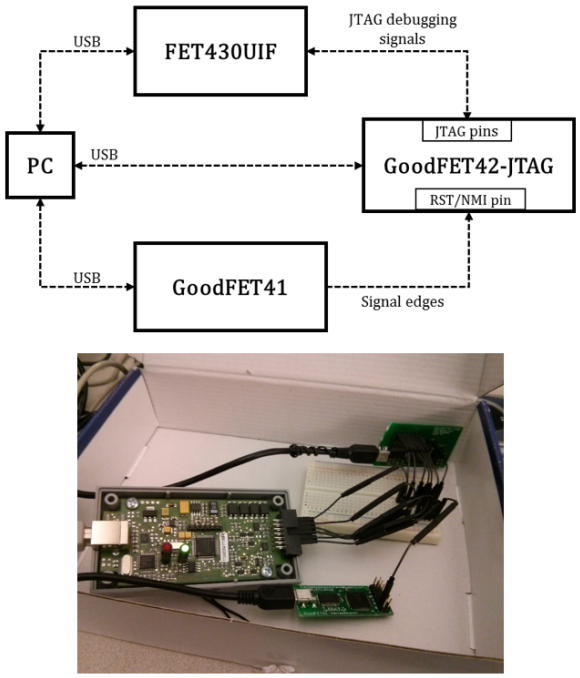

Figure 1: Laboratory setup

8.2 Software

To debug the target MSP430F2618 on the GoodFET42-JTAG, we used `mspdebug` version 0.22, a free debugger for MSP430 microcontrollers. We also extended the MSP430 simulator module in `mspdebug` to perform static analysis of MSP430 firmware.

To control and flash the GoodFET42-JTAG and GoodFET41, we utilized the GoodFET firmware and client software. We also modified the GoodFET firmware to enable the GoodFET41 to send falling edges from its GPIO pin to the RST/NMI pin of the GoodFET42-JTAG.

To disassemble target firmware binaries and perform static analysis on them, we used the IDA Pro interactive disassembler. We also wrote IDAPython scripts for IDA Pro to automate the static analysis of firmware binaries.

9. SPECIFIC EXPLOITS

In this section, we describe two IOP exploits (a.k.a. IOP programs) that utilize the primitive types described in Section 7. The targets of these attacks are simple MSP430 firmware images that we created specifically to allow these exploits to work; we later show how these can be combined into an actual "bug door" in common, working firmware. In all three examples, we use the LED on the GoodFET42-JTAG as a simple output mechanism to indicate the success of the exploit.

9.1 State accumulation exploit

An interrupt handler that predictably and incrementally modifies CPU state without saving or restoring it could be used to accumulate state. If the target program behavior can be influenced by this state, timed invocations of the interrupt handler could be used to control the program.

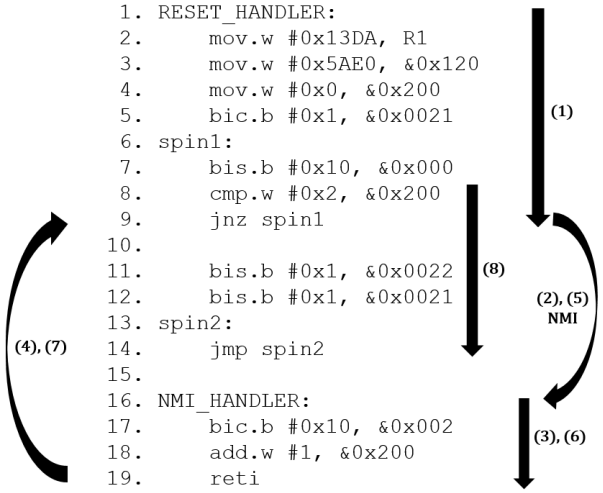

Figure 2: State accumulation exploit

The assembly source code of a simple firmware that contains such a vulnerability can be found in Figure 2. Numbered arrows represent actual instruction execution flow through the code, as caused by our crafted interrupts.

This IOP exploit uses a state accumulation primitive (line 18) in the NMI interrupt handler to cause the firmware to break out of an infinite loop (lines 6-9) and switch on its LED (lines 11-12). The IOP exploit takes place as follows:

(1) The SP (R1) is initialized at line 1, and the microcontroller is configured trigger NMI interrupts upon receiving a falling edge on the RST/NMI pin at line 2. The memory location that will be used to accumulate state, `0x200`, is cleared at line 4. The firmware then spins in lines 6-9, waiting for `0x2` to be accumulated in `0x200`.

(2) An NMI is triggered, which invokes the NMI handler at line 16.

(3) The NMI handler clears the NMI interrupt flag at line 17 to prevent a re- invocation of the NMI handler, and then accumulates `0x1` in `0x200`.

(4) The processor returns from the NMI interrupt to the spin loop. NMI is re-enabled at line 7 by setting the NMI interrupt bit. The firmware spins again, since `0x200` contains the value `0x1`

(5) An NMI is triggered, which invokes the NMI handler at line 16.

(6) NMI handler is invoked, accumulating another `0x1` in `0x200`.

(7) The processor returns from the NMI interrupt to the spin loop.

(8) This time, `0x200` contains the value `0x2`, so the firmware breaks out of the first spin loop, turns on the LED at lines 11 and 12, and spins at the second spin loop from lines 13-14

This exploit was designed to be carried out without any specific interrupt timings required. Simply triggering two NMIs would cause the GoodFET42-JTAG to turn on its LED.

9.2 Loop execution exploit

Besides containing state accumulation primitives, an interrupt handler may contain other useful primitives, such as one that writes to a useful memory address. Proper timing and nesting of interrupts could possibly allow both state accumulation and the writing of the accumulated value to other locations in memory, to be used in combination to affect the target.

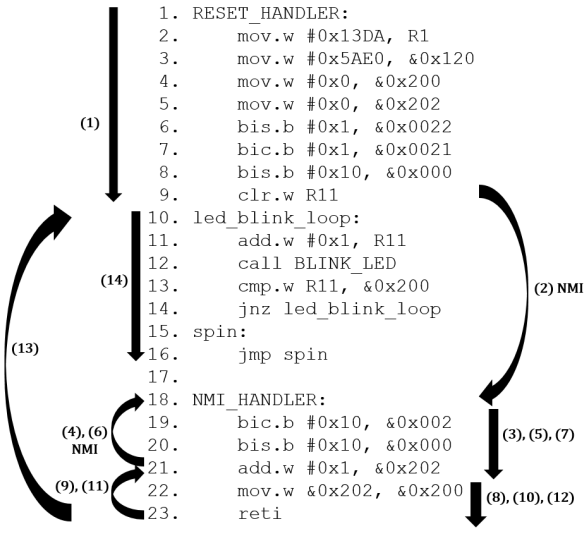

Figure 3: Loop execution exploit

The assembly source code of a simple firmware that contains such a primitive can be found in Figure 3. The interrupt handler in this firmware contains both a state accumulation primitive (line 21) and a memory write primitive (line 22). Using a series of well timed, nested interrupts, this IOP program first accumulates state, then writes it to a destination memory location that controls the number of times the firmware blinks its LEDs. The IOP exploit takes place as follows:

(1) The SP (R1) is initialized at line 1, and the microcontroller is configured trigger NMI interrupts upon receiving a falling edge on the RST/NMI pin at line 2. The memory locations that will be used for state accumulation and the final memory write, `0x200` and `0x202` are cleared in lines 4 and 5. The LED is turned off (if it is already on) from lines 6-7, and the NMI interrupt is enabled by setting the NMI interrupt enable bit at line 8. Finally, the register used as a counter for the LED blinking loop from lines 10-14 is cleared.

(2) An NMI is triggered, which invokes the NMI handler at line 18.

(3) The NMI handler clears the NMI flag at line 19, and re-enables NMIs at line 20 to allow for NMI nesting. It then accumulates `0x1` in `0x202` at line 21.

(4) Another NMI is triggered, which invokes the NMI handler at line 18.

(5) Step 3 is repeated.

(6) Another NMI is triggered, which invokes the NMI handler at line 18.

(7) Step 3 is repeated.

(8) The state accumulated in `0x202` (a value of `0x3`) is written to `0x200`.

(9) The firmware returns to line 21, where the nested NMI was triggered.

(10) Step 8 is repeated. Repeating the memory write is essentially a `nop`, since neither the source or destination has changed.

(11) Step 9 is repeated.

(12) Step 8 is repeated. Repeating the memory write is essentially a `nop`, since neither the source or destination has changed.

(13) The firmware returns to line 10, where the first NMI was triggered.

(14) The LED blinks as many times as the value stored in `0x200` (i.e. 3 times). The firmware then spins from lines 15-16.

This IOP program requires precise timing of nested NMIs to work. Repeating steps 3-4 an arbitrary number of times would allow the attacker to cause the GoodFET42-JTAG to blink its LED an arbitrary number of times.

9.3 Stack growing exploit

All MSP430 interrupt handlers are essentially stack-growing primitives. The invocation of an interrupt handler itself, as described earlier, causes the stack to grow by four bytes since the SR and PC are pushed onto the stack. If an interrupt could be nested (i.e., invoked while still executing its handler), the interrupt handler itself could be seen as a state accumulation tool that can decrement SP by 4 an arbitrary number of times.

The MSP430 has a linear memory layout, and the stack uses RAM, which precedes ROM in a higher address range (see Figure 4). If the stack was grown deep enough (i.e. towards lower addresses) by repeated invocations of the aforementioned interrupt handler until the stack reaches ROM, the `push` instructions executed by the microcontroller will decrement SP without actually writing to top of the stack—since ROM cannot be written to. Consequently, SP can be made to point to any arbitrary word in ROM. Upon execution of the next `ret` or `reti` instruction, a word from ROM will be loaded into PC, thereby redirecting control flow to the code that this word points to.

The hard and soft entry points for the MSP430's Bootstrap Loader (BSL) are stored in ROM at `0xC00` and `0xC02` respectively. As Goodspeed showed in [11], entering the BSL at its soft entry point without clearing the password check

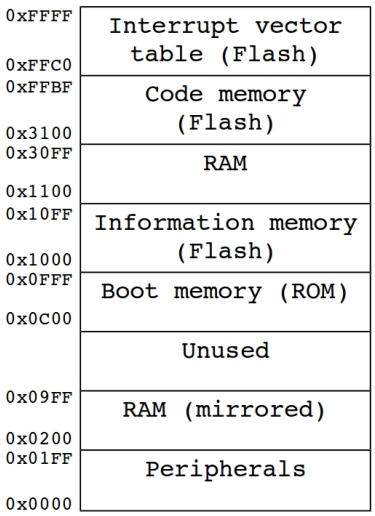

Figure 4: MSP430F2618 16-bit memory map

bit in `R11` allows the attacker to execute privileged BSL operations, such as reading out its firmware. Nesting interrupt handler invocations at a point in firmware where the bits in `R11` are set would therefore allow an attacker to enter the MSP430 BSL with administrative privileges.

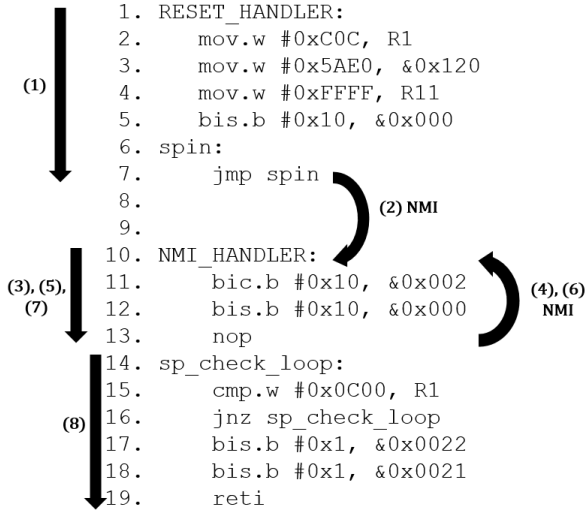

Figure 5: Stack growing exploit

The assembly source code of a simple firmware that contains such a vulnerability can be found in Figure 5. This IOP exploit first sets all bits in R11, then grows the stack using nested NMIs until it is deep enough for the next `reti` instruction to pop the BSL soft entry point into PC. The IOP exploit takes place as follows:

(1) The SP (R1) is initialized at line 1, and the microcontroller is configured trigger NMI interrupts upon receiving a falling edge on the RST/NMI pin at line 2. All bits in R1 are set at line 4, and NMIs are enabled at line 5. The firmware then spins at lines 6-7.

(2) An NMI is triggered, which invokes the NMI handler at line 10. Causing SP to be decremented by two words to `0xC08`. PC and SR are not actually written to memory, since SP is pointing to ROM.

(3) The NMI handler clears the NMI interrupt flag at line 11, and then re-enables NMIs by setting the NMI interrupt enable bit at line 12.

(4) An NMI is triggered, causing SP to be decremented by two words to `0xC04`.

(5) Step 3 is repeated.

(6) An NMI is triggered, causing SP to be decremented by two words to `0xC00`.

(7) Step 3 is repeated.

(8) Since SP is now `0xC00`, the firmware does not spin in the loop from lines 14-16, and instead proceeds to turn on the LED at lines 17-18 and execute the `reti` instruction at line 19. This first pops the word in ROM at `0xC00` into SR, then the word in ROM at `0xC02` (i.e. the BSL soft entry point) into PC. The microcontroller is therefore starts executing its BSL with full privileges (since the password check bit in `R11` is set).

This exploit was designed to be carried out without any specific interrupt timings required. Simply triggering three NMIs would cause the GoodFET42-JTAG to turn on its LED and enter the BSL with full privileges. However, even without the loop checking the value of SP (lines 14 to 16), this exploit is still possible to perform if the NMIs in step 2, 4, and 6 are properly timed.

10. PLANTING A BUGDOOR

An attacker able to read and modify target firmware source code could introduce minimal changes that would leave the target vulnerable to IOP attacks (i.e. leave a "bugdoor" in the firmware). To demonstrate this, we insert a bugdoor into TinyOS [32], an open source operating system targeting low-power wireless embedded systems. Since TinyOS was designed to be lightweight, there is no common kernel running in the background of each TinyOS program. Instead, TinyOS programs are compiled specifically for the target device along with the operating system abstractions required for the program to interface with device hardware. By planting the bugdoor into a TinyOS MSP430 interrupt handler, all MSP430 devices running TinyOS applications that support such an interrupt would be made vulnerable to IOP attacks.

We added two additional lines of code to the USART1 receive interrupt handler generated for the MSP430F1611 on the TelosB mote that makes the interrupt handler reentrant, as shown in Figure 6. This change transforms the USART1 receive interrupt into an unbounded stack growing primitive that can be used to perform the stack growing exploit described in Section 9.3.

The disassembled binary of a TinyOS application (MultihopOscilloscope) compiled for a MSP430 device (TelosB) target compiled with source code modification contains four additional lines of assembly (from `0xA3D2` to `0xA3D8`) in its interrupt handler, as shown in Figure 7.

This bugdoor, which enables privileged access to the MSP-430 BSL, required the addition of a mere two lines of code.

```
TOSH_SIGNAL(UART1RX_VECTOR) {
  uint8_t temp = U1RXBUF;
  WRITE_SR( READ_SR | SR_GIE); //+set GIE
  ME2 |= URXE1;                //+set USART1RX IE
  signal Interrupts.rxDone(temp);
}
```

Figure 6: Modified TinyOS source code

```
A3C6    .def __isr_3
A3C6 __isr_3:
A3C6    push.w  R15
A3C6                        ; sig_UART1RX_VECTOR
A3C8    push.w  R14
A3CA    push.w  R13
A3CC    push.w  R12
A3CE    mov.b   &7Eh, R14
A3D2    mov.w   SR, R15     ; tmp = READ_SR
A3D4    bis.w   #8, R15     ; tmp | SR_GIE
A3D6    mov.w   R15, SR     ; SR = temp
A3D8    bis.b   #10h, &5    ; ME2 |= URXE1
...
```

Figure 7: Disassembly of binary produced by modified TinyOS source

This additional code is not obviously malicious, will unlikely be flagged as suspicious by static analyzers, and it will likely be passed off as a benign error if discovered by hand. This small, malicious change could conceivably be pushed to the TinyOS source tree, and if it was, would expose the many embedded MSP430 devices that use TinyOS to the stack-growing exploit.

11. IOP BUGDOORS IN THE WILD?

Now that we showed a novel technique for "bugdooring" firmware by injecting them with synthetic vulnerabilities exploitable by IOP, we can ask: do such vulnerabilities or backdoors actually occur in the wild?

Our answer for now has to be, "we looked, but we haven't spotted any yet." In the spirit of good science, even though our search attempts were inconclusive, we finish with describing our tools and methods, and hope that they will help other researchers who might be interested in IOP attacks.

11.1 Static analysis

To look for potential IOP programs in actual MSP430 firmware, we attempted static analysis of the firmware of five MSP430-powered devices: EZ430U, EZ430URF, FET430UIF, MSP430FET, and GoodFET. We did this manually using IDA Pro and also added some automation using IDAPython scripts and `mspdebug` functions that we wrote.

11.1.1 Manual examination in IDA Pro

In the firmware binaries we examined in IDA Pro, interrupts were not re-enabled within interrupt handlers themselves, therefore preventing the nesting of interrupts and precluding the stack-growing attack described in Section 9.3. Moreover, registers and memory always seemed to be properly set before use, therefore making state accumulation attack described in Section 9.1 unlikely. However, it was clear that interrupt handlers did not completely save and restore CPU state before and after execution, which meant that most interrupt handlers probably modified CPU state in some way after each invocation.

The following describes the static analysis tools we wrote to automate looking for potential IOP programs. None of these scripts/programs *fully* automated the IOP program discovery and payload creation process; they were written to automate certain portions of the static analysis process and to narrow down segments of code/interrupt handlers that might allow for the creation of an IOP program.

11.1.2 Unset register/memory use scanner

This is an IDAPython script that searches for instances where a register or memory address is used to affect state or program behavior without first setting it. For each instruction that uses a register/memory address as the source operand in an instruction that redirects control flow (e.g. `jmp`, `call`) or performs a write to a register/memory (e.g. `mov`, `add`, `push`), this script searches up to n instructions backwards (where n is provided by the user)—including all possible conditional branch paths to that instruction—to check if that source register/memory address is set (e.g. by a `mov` or `clr` instruction). This tool is meant to identify registers/memory addresses that might affect program behavior if their state is modified or accumulated.

11.1.3 Brute-force search

This is an `mspdebug` function we implemented specifically for the simulator module. Given a certain register value pair (R, V), this program performs a brute-force search of the space of possible CPU states, executing all possible sequences of timed interrupts up to n steps into the program (where n is provided by the user). For each of these CPU states the program reaches, if register R contains the value V, the program terminates and reports that this CPU state is reachable. Otherwise, the search continues until all possible interleavings of up to n steps into the program are searched, and reports that such a state cannot be found. We define a *step* as either a instruction in the target program or the execution of an interrupt handler (assumed to be atomic and thus counted as a single step). In theory, this search would be able to determine whether it is possible for a program and its interrupts to put the processor in a "bad", exploitable state. For example, this tool would be able to check if it is possible for SP(R1) to be set to `0xC00` during the execution of an MSP430 program, and therefore determine whether the program is vulnerable to the stack growing exploit described in Section 9.3.

However, given the fact that the space of possible CPU states grows exponentially as n increases, this brute-force search took long amounts of time for even modest values of n; brute-force searching entire firmware binaries was far out of our reach.

Moreover, the latest version of `mspdebug` as of the time of this study (version 0.22) did not support the MSP430X extended instruction set, which some of the target firmwares (e.g. GoodFET) utilize. Since `mspdebug` could not emulate the behavior of the MSP430 on these instructions, our tool was unable to accurately search the space of possible CPU states in these firmwares.

11.1.4 Interrupt state-change analyzer

This is an `mspdebug` function we implemented specifically for the simulator module to determine the state changes in-

duced by the execution of a given interrupt handler at any given point in the program. This function saves the state of the virtual MSP430 CPU before executing the interrupt, executes it, compares the resultant state with the saved state, and prints all the differences in memory and register values to screen. This tool is meant to make understanding the side-effects of interrupts easier, and thus allow interrupt handlers potentially useful in IOP to be easily identifiable.

As expected, most interrupt handlers found in the target firmwares changed modified CPU state without restoring it. However, most the state change induced by the interrupt handlers tended to vary depending on where in the target program the interrupt handler was invoked, as well as how many consecutive times the interrupt handler was invoked. For example, some interrupt handlers only induced state changes after their first invocations, and did not induce further state changes on subsequent, consecutive invocations. In the several cases we discovered where state is either accumulated or changed in a register/memory location, we either found that this register/memory location is only used internally within the handler (e.g. a counter variable), or could not find—by hand or with the help of the Unset Register/Memory Use Scanner—instances where this register/memory was used elsewhere without being set.

Moreover, as mentioned in Section 11.1.3, `mspdebug` does not support the MSP430X extended instruction set. This tool was therefore unable to accurately determine the state changes induced by interrupt handlers that use instructions unique to the MSP430X architecture.

11.2 Hardware analysis

With the help of the FET430UIF, we were able to control the MSP430F2618 and examine CPU state via the JTAG protocol.

We needed to check if MSP430's registers could potentially be used to accumulate state across resets. This could be possible if the registers were not cleared or loaded each time a power-on reset or reset interrupt was trigger. Since the MSP430 manual [30] only explicitly states the values SR and PC take after resets, we needed to observe CPU states across resets in order to determine whether or not the other registers are loaded or cleared after each reset.

A limitation of using the JTAG protocol to debug the MSP430 is that a JTAG reset is performed each time the microcontroller is taken under and released from JTAG control.[31] Therefore, the CPU state displayed upon entering a JTAG "session" in `mspdebug` is that after a JTAG reset, not after a "true" reset interrupt triggered by a signal edge on the RST/NMI pin. The former is different from the latter in that the former is triggered by signals sent over JTAG interface pins (TDO, TDI, TMS, and TCK) rather than by a signal edge on the RST/NMI pin. While documentation states that CPU state is the same after either reset is performed, we suspected that this claim might not hold.

In order to bypass this limitation, we wrote custom firmware that writes all register values to fixed locations in memory immediately after the reset interrupt handler is invoked, so that they could be read out in a subsequent JTAG debugging session.

Using this custom firmware, we found that the MSP430F2618 does indeed load fixed values into its registers (other than PC and SR) on each reset triggered by an edge on the RST/NMI pin.

These values were generally predictable, though we occasionally saw them vary between our experimental sessions. However, it appears safe to assume that fixed values are indeed loaded into registers between resets, therefore preventing state accumulation in registers across multiple resets.

12. FUTURE WORK

Future work should focus on generalizing and fully automating the IOP program discovery process. The first step to achieving this would be to compiling a formal catalog of IOP "gadgets". This would provide direction for finding the building blocks needed to construct IOP programs.

The next step would be to fully automate the IOP static analysis process. The scripts and programs that we have written only semi-automate the static analysis process; further research should focus devising algorithms to locate IOP gadgets in target program binaries reliably and efficiently. One possible avenue to pursue is using SMT solvers to search the CPU state space, rather than the inefficient brute-force search tool we developed for this study. SMT solvers have helped analyze various kinds of vulnerabilities and exploit programming models [33]; IOP seems a promising application for them.

This application would, of course, require a model of the target ISA and of its interactions with interrupts, a significant investment; once this is done, however, using an SMT solver might yield a feasible way to search and query the states achievable through the triggering of interrupts. Such fully automated analyzers would also be able to identify programs vulnerable to IOP, and alert programmers to specific state dependencies between interrupt handler and target code. This would allow microcontroller programmers to better protect their code against potential IOP attacks.

Future work in IOP exploitation should also aim to construct IOP attacks that assume no access to the any part of the firmware on the target device or similar devices. Such attacks might infer the behavior of interrupt handlers purely by observing their side-effects on CPU state, rather than relying on analyzing disassembled binaries. Successfully achieving this stronger form of an IOP attack would future emphasize the significance of this exploit programming model to microcontroller security.

13. CONCLUSION

We have illustrated the concept of interrupt-oriented programming and have demonstrated synthetic examples of firmware code where IOP programs that achieve unintended computation on a microcontroller could be constructed. Moreover, we have shown that these synthetic exploits could be planted as bugdoors into commercially deployed firmware, and have built tools that will help further researchers discover and explore IOP programs in MSP430 firmware.

14. ACKNOWLEDGMENTS

We would like to thank Daniel Beer for helping us to debug some initial problems we had with interfacing our hardware with `mspdebug`. We would also like to thank the members of Dartmouth's Trust Lab—particularly Rebecca 'bx' Shapiro, Peter Johnson, and Jason Reeves—for feedback and useful discussions. Finally, thanks are due to Aurélien Francillon for discussions on minimalist exploitation of microcontrollers.

15. REFERENCES

[1] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *7th USENIX Workshop of Offensive Technologies (WOOT)*, August 2013. https://www.usenix.org/system/files/conference/woot13/woot13-bangert.pdf.

[2] E. Bosman and H. Bos. Framing Signals – A Return to Portable Shellcode. IEEE Symposium on Security and Privacy, May 2014.

[3] S. Bratus, J. Bangert, A. Gabrovsky, A. Shubina, D. Bilar, and M. E. Locasto. Composition Patterns of Hacking. In *Proceedings of the 1st International Workshop on Cyber Patterns*, pages 80–85, Abingdon, Oxfordshire, UK, July 2012.

[4] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. Exploit Programming: from Buffer Overflows to "Weird Machines" and Theory of Computation. *;login:*, December 2011.

[5] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 27–38, New York, NY, USA, 2008. ACM.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 559–572, New York, NY, USA, 2010. ACM.

[7] T. Dullien. Exploitation and State Machines: Programming the "Weird Machine", revisited. http://www.immunityinc.com/infiltrate/presentations/Fundamentals_of_exploitation_revisited.pdf, April 2011. Infiltrate Conference.

[8] T. Durden. Bypassing PaX ASLR protection. *Phrack 59:9*, 59(9), Jul 2002.

[9] eugene. Architecture spanning shellcode. Phrack 57:14, November 2001. http://phrack.org/issues/57/17.html.

[10] J. Evers. Alcatel leaves LAN switch software back door wide open, 2002. http://www.networkworld.com/news/2002/1122alcatellan.html.

[11] T. Goodspeed and A. Francillon. Half-blind Attacks: Mask ROM Bootloaders Are Dangerous. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, Berkeley, CA, USA, 2009. USENIX Association.

[12] C. Helfmeier, D. Nedospasov, C. Tarnovsky, J. S. Krissler, C. Boit, and J.-P. Seifert. Breaking and entering through the silicon. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 733–744, New York, NY, USA, 2013. ACM.

[13] jp. Advanced Doug Lea's malloc Exploits. Phrack 61:6. http://phrack.org/issues.html?issue=61&id=6.

[14] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/~krahmer/no-nx.pdf, September 2005.

[15] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 524–533, New York, NY, USA, 2009. ACM.

[16] H. Meer. The (Almost) Complete History of Memory Corruption Attacks. BlackHat USA, August 2010.

[17] T. Müller. ASLR Smack & Laugh Reference, 2008.

[18] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), December 2001.

[19] J. Oakley and S. Bratus. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. In *WOOT*, pages 91–102, 2011.

[20] G. Richarte. Re: Future of Buffer Overflows. Bugtraq, October 2000. http://seclists.org/bugtraq/2000/Nov/32.

[21] G. Richarte. About Exploits Writing. Core Security Technologies presentation, 2002.

[22] riq and gera. Advances in Format String Exploitation. *Phrack 59:7*, 59(7), Jul 2002.

[23] rix. Writing ia32 alphanumeric shellcodes. Phrack 57:15, November 2001. http://phrack.org/issues/57/18.html.

[24] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[25] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *7th USENIX Workshop of Offensive Technologies (WOOT)*, August 2013. https://www.usenix.org/system/files/conference/woot13/woot13-shapiro.pdf.

[26] Shimmer. Shop: Shimmer Platform. http://www.shimmersensing.com/shop/shimmer-platform.

[27] Solar Designer. Getting around non-executable stack (and fix). *Bugtraq mailing list archives*, August 1997. http://seclists.org/bugtraq/1997/Aug/63.

[28] Solar Designer. Openwall linux kernel patch. Web, 1998. http://www.openwall.com/linux.

[29] B. Spengler. The Case for GrSecurity. https://grsecurity.net/papers.php, September 2012. H2HC, Sao Paulo, Brazil.

[30] Texas Instruments. *SLAU144I: MSP430x2xx Family: User's Guide*, January 2012.

[31] Texas Instruments. *SLAU320L: MSP430 Programming Via the JTAG Interface*, September 2013.

[32] TinyOS Documentation Wiki. TinyOS. Web. http://tinyos.stanford.edu/tinyos-wiki/index.php/Main_Page.

[33] J. Vanegue, S. Heelan, and R. Rolles. SMT solvers for software security. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, WOOT'12, Berkeley, CA, USA, 2012. USENIX Association.

[34] R. Verma. Designing Portable, Wearable and Implantable Medical Electronics with Ultra-Low-Power Microcontrollers. http://tinyurl.com/lc4rat5.