A Rudimentary Bootloader for Computational RFIDs

Benjamin Ransford
ransford@cs.umass.edu

April 12, 2010

Abstract

The advent of batteryless, transiently powered, embeddable, effectively maintenance-free computers that operate solely on harvested energy has enabled new applications in sensing and actuation. In particular, computational RFIDs (CRFIDs) [17] serve as the basis of *RFID sensor networks* [7] that embed computational capabilities wherever radio waves can reach.

A key problem facing deployments of transiently powered computers in hard-to-reach places is that those computers can become inaccessible to physical contact, making it impossible to update device firmware after deployment.

This paper's main contribution is Bootie, a proof-of-concept bootloader for transiently powered computers that enables them to change their behavior without any physical contact. Bootie's current implementation comprises compile- and run-time components to bundle two or more unmodified firmware programs into a single bootable image, select a firmware program at boot time, supervise that firmware program's execution, and cycle through available firmware images.

This paper's secondary contribution is the preliminary design of a firmware update protocol that will allow a future version of Bootie to accept and install firmware updates wirelessly, thereby enabling experimenters for the first time to change the behavior of their deployed devices. The current implementation is a stepping stone toward this goal.

Bootie is implemented for the MSP430 family of microcontrollers. Source code is available at the author's web page.

1 Introduction

The recent advent of ultra-low-power microcontrollers is leading to an entirely new class of low-power embedded computers.[1] Maintenance-free *computational RFIDs* (CRFIDs; see [17, 21] for background) enable general-purpose computation with only harvested radio frequency (RF) energy and can operate in contexts where replacing or recharging a battery is inconvenient or hazardous (e.g., implantable medical devices [12]) or where integrated circuits and small surface-mount capacitors enable economies of scale for manufacturing and miniaturization. The primary challenge to CRFIDs, and similarly transiently powered computers, is completing time- or energy-intensive computations in spite of continual power interruptions that result in complete loss of computational state.

A second challenge concerns device maintainability. Computational RFIDs, like sensor motes and other embedded platforms, require a developer to physically plug a wire into the device to reprogram it via certain programming pins. When devices are embedded in places that are difficult to reach to obtain physical contact, they can no longer be reprogrammed with new firmware. An RFID sensor network [7] offers to make computation a ubiquitous property of an environment, but it threatens to fix the computational landscape in a single mode because its constituent devices cannot have their software upgraded.

This paper identifies an opportunity to make transiently powered devices' firmware *wirelessly upgradable*. A first step is to enable these devices to switch between firmware programs; without the ability to switch to a newly uploaded firmware program, there is no use for wireless firmware uploads. **Our contributions** in this paper are (1) Bootie, a bootloader for transiently powered devices that enables a developer to change a device's behavior without touching it; and (2) a preliminary protocol for wirelessly uploading new device firmware that is based on the EPC class-1 generation-2 standard RFID protocol [10].

[1]Portions of this introduction are adapted from [17].

Bootie combines compile-time and run-time mechanisms to facilitate switching among two or more firmware images. At compile time, Bootie packages multiple firmware programs into a single firmware image suitable for uploading to a microcontroller via a physical electrical connection. At run time, Bootie supervises the device's boot sequence and decides which firmware program to run.

The current proof-of-concept implementation of Bootie cycles through the firmware programs it bundled at compile time; it does not require a programmer or RFID reader infrastructure to interact with it. Besides introducing Bootie, this paper discusses future extensions to Bootie that will enable (1) wireless firmware uploading and (2) wireless firmware selection.

1.1 Related work

Bootloaders are common on desktop computers, especially computers that are configured to run different operating systems or operating system versions based on a boot-time decision. Examples of desktop bootloaders include GNU GRUB [18] and Apple Inc.'s Boot Camp [6].

The Deluge system [13] for TinyOS-based sensor motes, now integrated into TinyOS, considers "retasking" nodes—reprogramming them after deployment—to be a crucial capability of sensor networks, and accordingly develops a protocol to allow sensor motes to disseminate new programs via an *epidemic* strategy that guarantees eventual consistency across a network. Its predecessor, Xnp [9], allowed firmware to be updated from a central location. Both systems, especially Xnp, are similar in spirit to Bootie, but both benefit from an infrastructure (TinyOS, running on continuously battery-powered nodes with plentiful RAM and nonvolatile memory) that is not available to computational RFIDs. Bootie's protocol for firmware updates superficially resembles both systems' update protocols, but CRFIDs are non-autonomous nodes that are incapable of initiating communication; in particular, they cannot communicate with each other. The imposition of a powerful reader that provides all power and communication necessitates some changes from the sensor network model.

Researchers have considered the problem of *task scheduling* on the WISP prototype CRFID, but we consider that problem distinct from the problem of selecting alternative firmware images. Buettner's scheduler for CRFIDs [8] performs incremental trials to find an appropriate starting voltage for a given task, but does not specify how tasks are chosen at run time. Additionally, that system does not have a goal of enabling wireless firmware uploads.

In the spirit of task scheduling, the Mementos state checkpointing system for CRFIDs [17] introduced the notion of energy-aware instruction reordering. Mementos shares Bootie's realizations that reboots are frequent and operating energy is unpredictable, but it focuses on single-program scenarios.

2 Proposed Work

I propose to design and implement a simple bootloader for computational RFIDs. A computational RFID [17] (CRFID) is a highly constrained, batteryless computer that harvests energy to operate a microcontroller. To date, all published experiments with computational RFIDs (see, e.g., [5] for a snapshot of the state of the art) have cast them as single-purpose devices that perform a single task when operating power is available. Such devices have a simple boot sequence: execution starts at a predetermined point, the device completes its own internal routines (zeroing registers, calibrating timers, and so on), and execution eventually reaches the beginning of the single task; after that, the device acts as a single-purpose computer.

Currently, programming a CRFID—for example, uploading a new firmware image to change the device's behavior—involves gaining physical access to a set of hardware pins on the device. The need for physical access partially defeats the purpose of CRFIDs; a main reason they lack batteries is so that they can be deployed in hard-to-reach places where having a battery would be a liability. CRFIDs have been proposed for use in infrastructure monitoring [17], wildlife tracking [11], and dense stationary sensing [11] (so many nodes that changing batteries

would be unwieldy). In each of these cases, the ability to update CRFIDs' firmware in situ would enable faster iteration and easier experimentation. A bootloader for CRFIDs is a first step toward wireless firmware updates; it would enable a CRFID to start from newly uploaded firmware once that firmware has been successfully uploaded, and not before then. Uploading new firmware over the air is challenging because the operations (e.g., cryptographic hashing) required to verify firmware images severely tax the limited onboard energy reservoir. Initial measurements [17] suggest that a transiently powered CRFID may lose power several times per second while it is computing. Further measurements (e.g., [11]) suggest that the communication link is unreliable, especially with sub-optimal antenna orientation or occluding bodies. It is therefore reasonable to take defensive measures against unreliable firmware updates.

For this project, I will implement a proof-of-concept task switcher that toggles back and forth between two programs on a CRFID. My task switcher will, via a compiler optimization pass at compile time, wrap itself around the two programs, and all three entities (the task switcher and the two programs to be switched) will be compiled into a single firmware image and uploaded to a CRFID via the wired interface. At run time, the CRFID will read a flag from nonvolatile memory and select the program to run based on the value of the flag. It will then start the selected program. If the program fails to complete because the device runs out of energy, the flag will remain unchanged and the same program will be rerun at next boot. However, if the program completes, the task switcher will update the flag so that the other program will be run at next boot. Maintaining the flag in nonvolatile memory is nontrivial because a flash memory bit can be flipped cheaply in only one direction; the other direction requires an expensive erasure. I will discuss coding schemes that minimize the necessity of erasure.

In the future, my bootloader could be generalized to support more than two alternative firmware images, and it could be modified to accept "switch to firmware N" commands from a controlling RFID reader. A large part of the synthesis project report will be devoted to discussion of the design principles concerning the implementation of an N-way bootloader. I will discuss the problem of encoding an N-way switch in flash memory, and I will discuss in depth the procedure by which new firmware can be uploaded. Given the aforementioned limitations of current CRFID deployments, I believe this work will be of interest to the CRFID community and perhaps the networking community.

3 Design

This section discusses the current implementation of Bootie. For a discussion of future extensions, in particular a facility for wireless firmware updates, see Section 6.

3.1 Challenges

A fundamental property of computational RFIDs is that they have no onboard power supply; rather, they are equipped only with a small energy buffer in the form of a capacitor. Even when the distance between a CRFID and the reader that powers it via radio waves is fixed, constant energy availability is not guaranteed. Figure 1 depicts periods of operating energy and power failure in a laboratory experiment. Previous work [11, 17] has noted that energy shortfalls are a constant concern for transiently powered CRFIDs.

Each time its power drops below a cutoff threshold, the MSP430's *brownout protection* feature cuts power to the microcontroller. (Similar features are common on other microcontrollers as well.) Each power loss event—such events are common on computational RFIDs, as Figure 1 shows—results in the total loss of computational state on the microcontroller. Fortunately, executable code is stored in (and executed from) onboard nonvolatile flash memory instead of RAM, so the program itself is not lost with each power loss event.

In a regime in which the total loss of computational state is a common occurrence rather than a rare event, it is necessary to reconsider certain assumptions about program execution. For example:

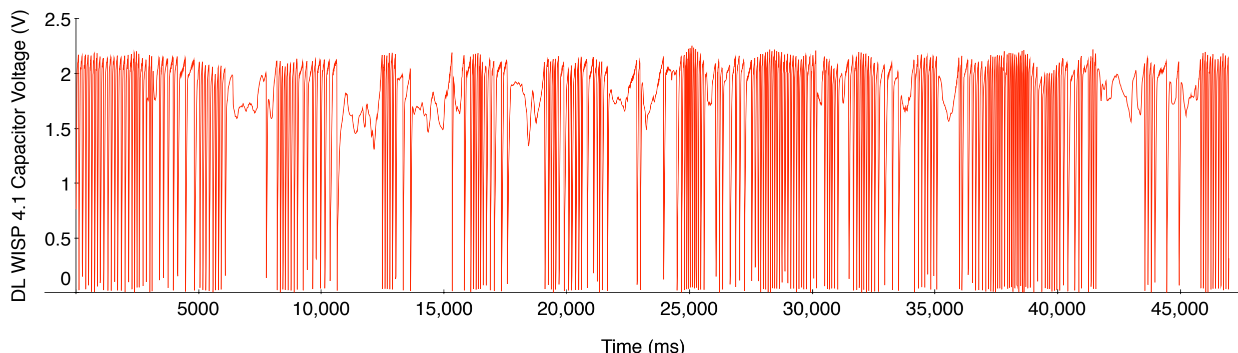

Figure 1: Time (milliseconds) versus capacitor voltage (volts) for a DL WISP 4.1 prototype computational RFID over a timespan of 47 seconds. The DL WISP 4.1 was held at a constant distance of 3 feet from an RFID reader that was broadcasting queries at approximately 100 Hz. The DL WISP 4.1's orientation was manipulated by a human experimenter to simulate motion in a pocket during fast walking. Each vertical dip represents a loss of operating power and computational state followed by a recharging and device reset. The spans between vertical dips are usable for computation.

- If a single program is loaded into the microcontroller's code memory, a power failure will result in the program being rerun from the beginning each time operating power returns to a sufficient level. The boot sequence runs each time operating power is restored, so any code that runs at boot time must run quickly to allow time for subsequent computations to execute.

- A state machine is a common programming device for embedded applications on microcontrollers. Unless the programmer is careful to copy program state to nonvolatile storage (which copying takes time and energy above normal instruction execution), the state machine will enter its start state frequently, and expected transitions later in the state sequence may not occur. A particular challenge for a bootloader is that it must durably store its own state if the action of selecting a firmware program is meant to be durable.

- Operations that are not idempotent may be performed more than once if computational state is lost in between instances of the operation. A programmer must be wary of non-idempotent operations, and a bootloader must try to respect requests that a firmware program be run a limited number of times (in a row, for example).

- If a bootloader is to perform cryptographic operations on firmware programs (e.g., integrity verification via a cryptographic checksum), it must execute CPU-intensive tasks. If operating power disappears during firmware verification, the same task may need to be re-executed from the beginning at next boot.

A further challenge to running multiple alternative programs on a computational RFID is related to a distinct advantage of the class of devices. The ability to embed CRFIDs in hard-to-reach places—inside concrete, for example, or as part of an implanted medical device [12]—results in a lack of reprogrammability. CRFIDs, like most experimental microcontroller-based devices, are typically programmed once while connected to a desktop computer, then deployed. Reprogramming such a device involves repeating the programming step after making physical (electrical) contact with the device. To enable firmware updates after deployment, a bootloader must allow binary images to be uploaded wirelessly, and it must be able to patch new firmware into

an existing environment, which may involve difficult address space manipulation, among other issues.

3.2 Design goals

The version of Bootie described in this manuscript sets two goals: run multiple firmware images on an MSP430 microcontroller (though not at the same time); and switch among firmware images without requiring physical contact with the device.

The goals of the current implementation are as follows:

- Minimize code size. The MSP430F2131 microcontroller on which we developed Bootie has 8 KB of flash memory available for developer-supplied code, and only 256 bytes of RAM. We must therefore conserve code space where possible. In particular, the bootloader should minimize redundancy when bundling firmware images together.

- Minimize boot time. Figure 1 demonstrates that CRFIDs may reboot many times per second, with timespans usable for computation being very short; we must therefore minimize the run time of the boot sequence.

- Make it easy to combine two or more projects using the bootloader. The mechanics of switching between firmware programs should be handled automatically to the maximum extent possible, as should the mechanics of integrating two or more projects' compilation phases.

- The bootloader should never leave the system in an unrecoverable state. If the bootloader fails, it should do so in such a way that the device runs one of the input firmware programs.

The long-term goals for future versions of Bootie are described in Section 6. Briefly, a future version of Bootie will be able to accept, verify, and install firmware updates wirelessly, *despite interruptions to operating power.*

4 Implementation

The current version of Bootie comprises 231 lines of code and includes C, C++, and Perl programs as well as a Makefile that assembles multiple input firmware programs into a single bootable firmware image. The code is freely available at the author's web page [1]. It requires a development copy of the LLVM compiler toolchain [14] and a copy of GNU binutils [3] that supports the microcontroller architecture.[2] Bootie targets the TI MSP430 [4] family of microcontrollers. It is implemented for, and has been tested exclusively on, MSP430F2131 microcontrollers mounted on Olimex test boards [16]. It could be made to target the DL WISP 4.1's MSP430F2132 microcontroller via a one-line code change in its Makefile, but a design flaw in the DL WISP 4.1 precludes the sort of nonvolatile write Bootie uses to record its state.[3] Because Bootie uses no WISP-specific features of the DL WISP 4.1, we chose the Olimex development board for simplicity. The MSP430F2131 and MSP430F2132 share a flash memory size of 8 KB, but the former has 256 bytes of RAM versus the latter's 512 bytes.

Bootie's present implementation serves as a *proof of concept*. It accepts two or more C programs as input, generates another C program containing boot routines and program switching logic, and bundles all of the programs together into a single firmware image. When loaded with Bootie, the device cycles through the input firmware programs on successive boots.

4.1 Proof of concept

An illustrative example of Bootie's compile-time operation is described below and diagrammed in Fig-

[2] At the time of this writing, a Web search for "llvm-msp430" finds a project of that name that includes appropriate versions of all of the necessary tools.

[3] The DL WISP 4.1 uses a 1.8-volt voltage regulator to ensure that the microcontroller always receives at most 1.8 volts. However, the voltage threshold for flash writing and erasure on the MSP430F2132 is 2.2 volts; therefore, the DL WISP 4.1 cannot write reliably to its own microcontroller's flash memory. A driver exists that allows the DL WISP 4.1 to use an off-chip nonvolatile EEPROM for storage at 1.8 volts [11].

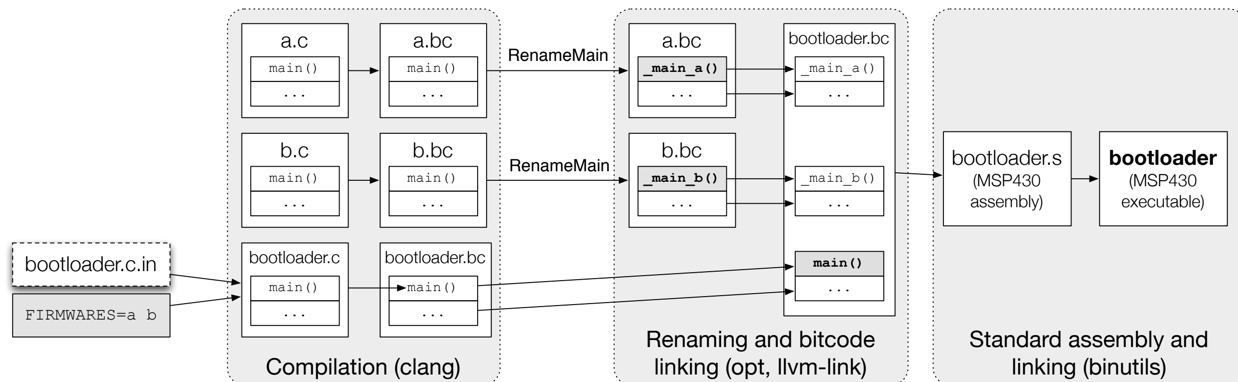

Figure 2: Overview of the compile-time operation of the current version of Bootie. Standalone firmware programs `a.c` and `b.c` are compiled, their main methods' names changed, and the results combined with a wrapper program `bootloader.c`. Standard assembly and linking of the combined program produces a firmware image called `bootloader` that can be loaded directly onto the target microcontroller.

ure 2. For demonstration purposes, Bootie was given two input firmware programs: `a.c`, which blinks an LED attached to the microcontroller's pin 1.1, and `b.c`, which blinks an LED attached to pin 1.3.[4]

Compile-time operation. To give Bootie the requisite control over the device's operation, we use a standard method of ensuring that it runs at boot time: we name its startup procedure `main()` and use the compiler toolchain's linker to build an executable that calls `main()` when it runs. However, each input firmware program already has its own `main()` procedure that expects to run at boot time. To avoid collisions during the linker phase, we use a program manipulation pass at compile time to rename each input firmware's `main()` procedure according to the source file in which the procedure is found. (A future version of Bootie will recursively check for collisions with its proposed names as well.)

In detail:

- In Bootie's Makefile, we specify `FIRMWARES=a b` to indicate that we want the bootloader to toggle between the program in `a.c` and the program in `b.c`.
- The Makefile invokes clang [2], the LLVM C compiler front end, to compile `a.c` and `b.c` into LLVM bitcode.
- The Makefile invokes LLVM's `opt` tool to run an optimization pass from `RenameMain`. RenameMain, adapted from a pass of the same name in the Mementos [17], takes an input program named like `foo.bc` and renames its `main()` function to `_main_foo()`. The pass runs on `a.bc` and `b.bc`, renaming their `main()` functions to make way for the bootloader's own `main()` function.
- The Makefile invokes a Perl script that generates a file called `bootloader.c`. This file contains code to toggle among the available firmwares.
- The Makefile invokes clang to compile `bootloader.c` into `bootloader.bc`.
- The Makefile invokes `llvm-link`, LLVM's bitcode linker, to link all the bitcode files (`a.c`, `b.c`, bootloader.bc) together. The only `main()` function comes from `bootloader.bc`.

[4] It is not unusual for a complete device firmware program to be written as one file; for example, the WISP's firmware is maintained as a single C file. The chief advantage of such a single-file structure is that global variables—frowned upon in general use but common in embedded systems—can be made available to the entire program without any complicated post-compilation linkage between units.

- The Makefile invokes `llc`, LLVM's bitcode-to-assembly translator, to generate MSP430 assembly code.
- The Makefile invokes `msp430-gcc`, mspgcc's compiler driver (which invokes the standard *binutils* tools), to assemble and link the MSP430 assembly code into a firmware image called `bootloader`. This final image is what we load onto the microcontroller.

Run-time operation. Figure 3 depicts the run-time operation of Bootie in the same example scenario. Bootie's compile-time operations result in its `main()` function being called immediately each time the microcontroller boots. This `main()` function invokes one of the input programs based on information available at run time. In detail:

- At boot, the device reads a location in nonvolatile memory (`0x10BE`, the last 16-bit word of information memory segment C in an MSP430F2131 microcontroller's flash memory) that contains the address of a `main()` function to run, `_main_a()` or `_main_b()` in the example above. The location in nonvolatile memory acts as an $n+1$-way switch if there are n input firmware programs; the extra switch state is the default value for initialized flash memory and indicates that no firmware is currently marked as the next to run. The switch state is compared to a hard-coded lookup table of function pointers computed at compile time. If the switch does not point to a known `main()` function, it is set to point to the default function—`_main_a()` in our example. The switch state is now guaranteed to give the memory address of a `main()` function, so it is cast to a function pointer `fn`.
- The next switch state (in a rotating fashion) is looked up in the list of pointers to main functions and cast to another function pointer `nextfn`.
- Bootie calls the function pointed to by `fn`.
- If that function returns, the `nextfn` pointer is written to the $n+1$-way switch location to indicate that it should be run at next boot. If, however, the function pointed to by `fn` does *not* return, the $n+1$-way switch still matches `fn`, so the same function will be run again at next boot.
- Execution terminates; Bootie puts the microcontroller in its CPUOFF power mode. The microcontroller will not execute any more code until its next boot. (This step is strictly optional.)

Some of the behavior defined above is specific to the test case; for example, a real application may require more than a simple rotation among firmware programs. Section 5 discusses such practical concerns.

5 Evaluation

This section evaluates the current implementation of Bootie with respect to the goals stated in Section 3.2. For a discussion of extensions to Bootie that will expand its capabilities and usefulness in future versions, see Section 6.

5.1 Versus goals

Bootie meets the stated goals listed in Section 3.2. Specifically:

- Bootie's code size is small relative to the total size of the firmware image downloaded to the microcontroller. Its code size when compiled without optimizations is 240 bytes in 60 instructions, or 3% of the 8 KB code memory; with optimization (at gcc's `-O3` level) it is 178 bytes in 56 instructions, or 2% of the code memory. We consider this overhead acceptable.

  After compilation (with no optimization) and linking against required libraries (e.g., the C language runtime for the target microcontroller), the executable bootloader including two sample programs is 3 833 bytes long. Because we wait until all components (the input programs' and Bootie's) have been compiled before linking them together, we include the required libraries only once, satisfying our goal of minimizing duplicated code.

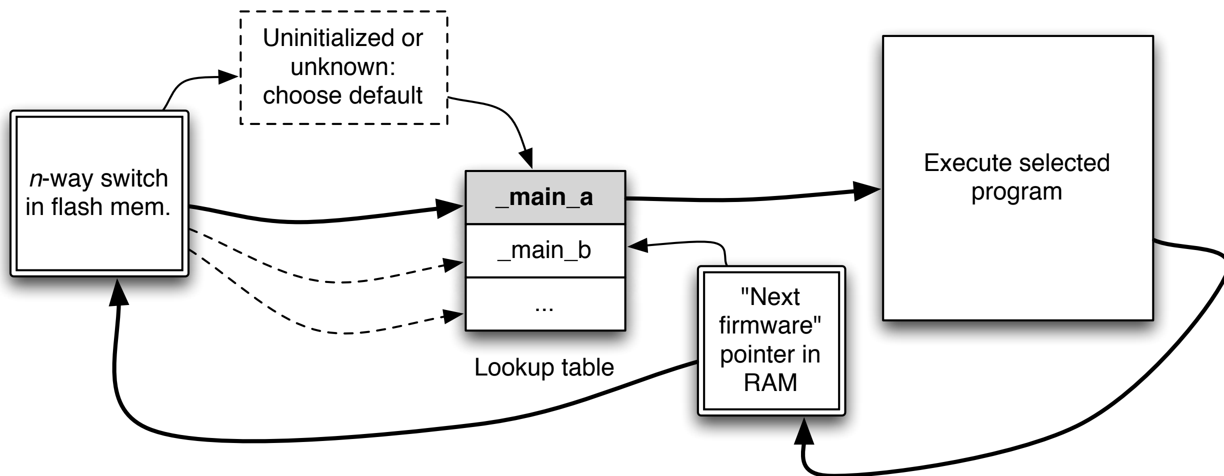

Figure 3: Overview of the run-time operation of the current proof-of-concept version of Bootie. Bootie's `main()` function examines an $n + 1$-way switch (a function pointer stored in flash memory) to determine which firmware program to run. Once the firmware is selected, a "next firmware" pointer is made to point to the next firmware program in the lookup table. The firmware program is run through its function pointer. If it completes successfully, Bootie writes the "next firmware" pointer to the $n + 1$-way switch so that the next firmware will be run at next boot. If, however, the program fails to finish or the device runs out of energy before it updates the $n + 1$-way switch, the firmware selected at the beginning of the lifecycle will be run at next boot.

- At 16 MHz with a cycle time of 62.5 ns, Bootie's boot sequence—which comprises simple memory reads and jumps, along with a possible flash segment erasure—requires 7.6 μs (121 cycles) if no flash erasure is necessary or 2.7 ms (approximately 49 000 cycles, 48 800 of which are spent erasing flash) if flash erasure is necessary. Such timings can support up to approximately 370 full boot cycles per second, which we find qualitatively acceptable given the adverse energy harvesting conditions under which CRFIDs operate.

- Bootie's compile-time functionality is implemented in compiler passes and a Makefile. The input programs in this paper's proof-of-concept implementation are individual C source files that can independently be compiled into firmware images. No changes to the input programs' code were necessary, satisfying our goal of making it easy to combine multiple firmware images. Furthermore, our implementation of Bootie requires no physical interaction with the device to switch between firmware images, satisfying our goal of automatically changing device functionality.

- Bootie maintains only one word (2 bytes) of state in nonvolatile memory. If the $n + 1$-way switch in nonvolatile memory acquires an invalid value (because of, e.g., insufficient voltage during a flash write [20]), Bootie defensively assumes that it should run a default firmware program. Furthermore, because the current implementation of Bootie does not overwrite any firmware via wireless upload (or any other means), it preserves the integrity of programs stored in code memory. Bootie therefore satisfies our goal of never leaving the system in an unrecoverable state.

Scalability. We have tested Bootie's program-rotation scheme with up to 5 input programs; the

example in this paper uses 2 input programs only for simplicity. Because Bootie's $n + 1$-way switch is stored as a 16-bit word, Bootie supports up to $2^{16} - 1 = 65\,535$ input programs; in practice, the number of input programs is limited by the code memory size (8 KB on our test devices). Adding another program to the rotation requires two steps: copy its C source code into Bootie's build directory (alongside the existing programs' source files) and edit the `FIRMWARES` line in the Makefile to add the new program's name.

5.2 Suitability for CRFIDs

We consider the current proof-of-concept implementation of Bootie to be compatible with the CRFID model (constrained microcontroller, frequent power loss, task switching without physical contact). However, we question the utility of a program that toggles either one pin or another pin in an endless repeating sequence.

It is important to note that Bootie is not limited to programs that toggle pins. Given two input programs that perform useful tasks (e.g., sensing from an onboard analog sensor), a cooperating RFID reader could effectively duty-cycle the CRFID's different behaviors by repeatedly turning its transmission of the energy-supplying carrier wave on and off.

The current proof-of-concept implementation demonstrates that it is possible to load multiple firmware programs onto a CRFID's microcontroller. Section 6 describes extensions to the current implementation that will make Bootie more useful to CRFIDs that are physically inaccessible.

6 Extensions

In this section, we discuss several extensions to Bootie's current proof-of-concept implementation. In particular, we propose a preliminary design for wireless firmware updates for computational RFIDs.

6.1 Wireless firmware updates

For Bootie to be useful on a deployed CRFID, it must support wireless uploading and verification of new firmware programs.

Fortunately, the EPC class-1 generation-2 RFID protocol specification [10], upon which current prototype CRFIDs operate, includes a *BlockWrite* command that can write up to 4 096 bits at a time; each *BlockWrite* is acknowledged separately by the recipient. It also includes a *Write* command that can be used to update a single word of memory at a time (e.g., the $n + 1$-way switch Bootie uses to select the program to run).

A protocol sketch—which neglects error handling, bookkeeping work for the CRFID related to resuming a firmware upload or incorporating new firmware into the boot sequence, and a notion of authentication—appears in Figure 4. The protocol is similar to other file transfer protocols that support chunked content transfers. Because the RFID protocol mandates that a participating tag reply within 20 milliseconds of a *BlockWrite* command from a reader, and we have timed a 256-byte flash write at approximately 10 ms, we suggest that the reader send data in 256-byte (128-word) chunks.

We propose that Bootie be made to support the cryptographic authentication of newly uploaded firmware, both after a complete transfer and one chunk at a time. We propose to use a simple message authentication code (MAC) scheme that has already been implemented on a CRFID [19]. A property associated with the use of a MAC is that the secret key used in the MAC computation must be shared among (i.e., stored on) all participating tags and readers. If that key were disclosed by any device (e.g., by an adversary capturing and decapsulating a CRFID's microcontroller [15]), the security of the entire collection of devices would be compromised. However, considering the much higher computational cost typically associated with public-key cryptography, we suspect that a MAC scheme is more appropriate as a default for Bootie.

Challenges to uploading firmware. The problem of computing a binary patch to send to a CRFID is a matter for future consideration. We assume

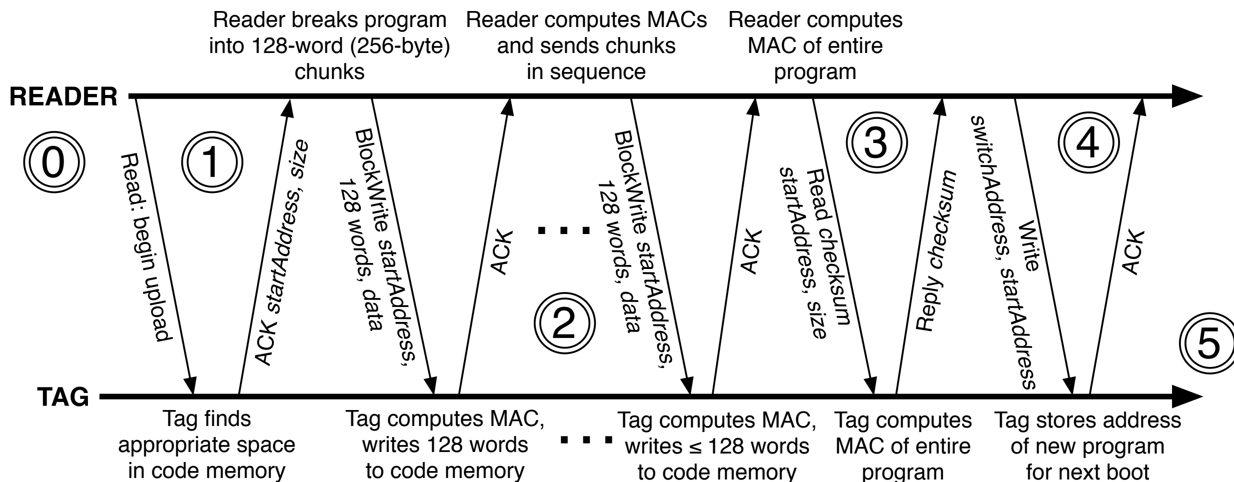

Figure 4: Firmware upload protocol sketch for a CRFID ("Tag"). Step 0: An RFID reader that wishes to upload new firmware to a CRFID must first perform all of the necessary work to assemble a binary patch (i.e., must resolve all symbols, strip out duplicate information, etc.). Step 1: the reader issues an RFID *Read* command indicating to the tag that it would like to upload new firmware. The tag confirms that there is appropriate space for the new firmware and sends the starting address of the space. Step 2: the reader sends a sequence of 128-word (256-byte) program chunks and their cryptographic MACs via RFID *BlockWrite* commands. Step 3: the reader and tag both compute a cryptographic MAC over the entire program sent by the reader. Step 4: if the checksums match, the reader sends an RFID *Write* command to select the new firmware on the tag. Step 5: the new firmware is usable on the tag.

that such computations are too CPU-intensive for a CRFID and must therefore be carried out on the more powerful reader infrastructure.

Another challenge is that current CRFID prototypes (e.g., the DL WISP 4.1) do not support the full EPC class-1 generation-2 RFID protocol in software. We suspect that Bootie could be integrated into device hardware at the same time as full RFID protocol support is moved into hardware; hardware protocol support has long been a goal of CRFID development.

A further challenge is tolerating protocol violations or errors. If the RFID reader providing chunks of a firmware image fails to send all of them—i.e., if it silently aborts the protocol in step 2 as shown in Figure 4—then the CRFID must, at some subsequent time, cancel its end of the firmware upload. A constantly powered computer might use a timeout period to expire stale upload sessions, but a transiently powered computer that lacks a built-in notion of time cannot. Further complicating the firmware upload process, a partial upload may be used as an attack vector by a malicious party, for example to fill precious nonvolatile memory or to disrupt the next legitimate transmission of firmware.

6.2 Addressing limitations

The current proof-of-concept implementation of Bootie has no recourse in cases of colliding interrupt handlers. If two or more input firmware programs use the same interrupt vector—i.e., they both try to install their own handlers for certain events at the same memory location—then a compile-time error results. We plan to evaluate two possible solutions to this problem. First, Bootie's optimization passes could recognize interrupt vectors at compile time, transform them into conventional functions without the `interrupt` attribute, and install its own interrupt

vectors that examined the value of the $n + 1$-way switch and dispatched the appropriate function. A second approach is to enforce a compile-time requirement that at most one input program provide interrupt vectors. Bootie's current behavior is in line with this second approach, except that it does not enforce the compile-time requirement.

Another simple extension would extend Bootie's collision avoidance mechanism (which currently attempts to ensure that only the input programs' `main()` functions are renamed to avoid a collision). Bootie could add another optimization pass that renames *every* symbol with external linkage, then correcting any references to those symbols where they appear. For example, an input program `f.c` with a `main()` function and `foo()` function might be altered to have `_main_f()` and `_foo_f()`.

Further collision avoidance: rename *all* functions in an input program rather than just the `main()` functions.

7 Conclusion

We present Bootie, a rudimentary bootloader for computational RFID (CRFID) devices. We present a proof-of-concept implementation of Bootie that switches between multiple CRFID firmware programs without requiring physical interaction with the device. Bootie can be extended to support the wireless upload of new firmware programs to CRFIDs; we sketch a protocol to support such an upload between an RFID reader infrastructure and a CRFID.

References

[1] B. Ransford home page. `http://www.cs.umass.edu/~ransford/`.

[2] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[3] Gnu binutils. `http://www.gnu.org/software/binutils/`.

[4] MSP430 Microcontroller (MCU), Low Power Mixed Signal Processors. `http://ti.com/msp430`.

[5] First Workshop on Wirelessly Powered Sensor Networks and Computational RFID (WISP Summit). Berkeley, CA, November 2009.

[6] Apple Inc. Windows compatibility—how Mac works with PCs. http://www.apple.com/macosx/compatibility/.

[7] Michael Buettner, Ben Greenstein, Alanson Sample, Joshua R. Smith, and David Wetherall. Revisiting smart dust with RFID sensor networks. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks (HotNets-VII)*, October 2008.

[8] Michael Buettner, Benjamin Greenstein, and David Wetherall. An energy aware scheduler for computational RFID programs. WISP Summit (presentation), November 2009.

[9] Crossbow Technology Inc. Mote in-network programming user reference, March 2003.

[10] EPCglobal Inc. *EPC^{TM} Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz—960 MHz*, version 1.2.0 edition, October 2008.

[11] Jeremy Gummeson, Shane S. Clark, Kevin Fu, and Deepak Ganesan. On the limits of effective micro-energy harvesting on mobile CRFID sensors. In *Proceedings of the ACM Annual International Conference on Mobile Systems, Applications, and Services (Mobisys)*, June 2010.

[12] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *IEEE Symposium on Security and Privacy*, May 2008.

[13] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol

for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[14] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[15] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse-engineering a cryptographic RFID tag. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 185–194. USENIX Association, 2008.

[16] Olimex Ltd. MSP430-H-2131 MPS430F2131 Header board. `http://www.olimex.com/dev/msp-h2131.html`.

[17] Benjamin Ransford, Shane Clark, Mastooreh Salajegheh, and Kevin Fu. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *USENIX Workshop on Power Aware Computing and Systems (HotPower)*, December 2008.

[18] Allen Beye Riddell. GNU GRUB. `http://www.gnu.org/software/grub/`.

[19] Mastooreh Salajegheh, Shane Clark, Benjamin Ransford, Kevin Fu, and Ari Juels. CCCP: Secure remote storage for computational RFIDs. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[20] Mastooreh Salajegheh, Kevin Fu, and Erik Learned-Miller. Half-wits: Non-volatile storage for low power devices. Synthesis project report, University of Massachusetts Amherst, November 2009.

[21] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. In *IEEE Transactions on Instrumentation and Measurement*, 2008.