Memory-Safe Microcontroller Applications with the Bedrock Structured Programming System

Benjamin Ezra Barenblat

SB Computer Science and Engineering
Massachusetts Institute of Technology, June 2013

submitted to the Department of Electrical Engineering and Computer Science in partial fulfilment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

June 2015

© 2015 the Massachusetts Institute of Technology. All rights reserved.

Benjamin Ezra Barenblat
Department of Electrical Engineering and Computer Science
22 May 2015

certified by ______________________

Adam Chlipala
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor
22 May 2015

accepted by ______________________

Albert R. Meyer
Hitachi America Professor of Engineering
Chairman, Masters of Engineering Thesis Committee
22 May 2015

Memory-Safe Microcontroller Applications with the Bedrock Structured Programming System is copyright © 2015 the Massachusetts Institute of Technology. All rights are reserved; no part of this book may be reproduced in any form or by any means without permission from the copyright holder.

This book was designed and typeset by Benjamin Barenblat using the LaTeX document preparation system. Body text is set in Scala Sans, while code is set in DejaVu Sans Mono.

This book was printed and bound in the United States of America on 100% cotton, acid free, 24 lb. Southworth Business stock.

10 9 8 7 6 5 4 3 2 1

Abstract

Microcontrollers – low-power, real-mode CPUs – drive digital electronics all over the world, making their safety and reliability critical. However, microcontrollers generally lack the memory protection common in desktop processors, so memory safety must come through other means. One such mechanism is Bedrock, a library for the Coq proof assistant that applies separation logic to a small C-like language, allowing programmers to prove memory-related properties about their code. I used Bedrock to build a security peripheral out of a Cortex-M3 microcontroller; my peripheral provides both AES encryption and append-only logging to a host system, and I showed the software it runs is memory-safe. Working with Bedrock was challenging but rewarding, and it provides a glimpse into a future where system programmers can prove code correct as a matter of course.

Thesis Supervisor: Adam Chlipala
Assistant Professor of Electrical Engineering and Computer Science
Department of Electrical Engineering and Computer Science

Acknowledgements

My advisor, Professor Adam Chlipala, along with other members of the Programming Languages and Verification Group at CSAIL, wrote Bedrock, which has formed the core of my research over the last year. Without Adam's help, proofs which I can now write in mere hours would have taken months, and I would have become frustrated and given up on Bedrock long, long ago. I also owe many thanks to the rest of the group, especially C. J. Bell, Ben Delaware, Jason Gross, and Peng Wang, for their patience and Coq expertise.

In addition to the Bedrock, this project involved quite a bit of C++ programming, which was a fairly novel experience for me. I am thus deeply grateful to Alex Chernyakhovsky, who provided C++ expertise, code review, and design suggestions throughout my work. If any of my C++ code appears elegant, I assure you, it is due to Alex's help.

During the term of the project, I received invaluable aid from Professor Frans Kaashoek and Professor Nickolai Zeldovich, both of whom continually exhorted me to greater and more ambitious efforts. Our discussions resulted in several key ideas in this work; they also gave me new perspective on the field of formal methods and verification as a whole. I am a better researcher for their time, and I owe them a debt of gratitude.

I received material support from a wide variety of sources during my work. Patrick Stach, of Inverse Limit, provided the AES implementation I used, which has worked flawlessly and saved many engineer-hours of implementation and debugging. I owe even greater thanks, however, to Dom Rizzo, Mudge, and the remainder of Google's Advanced Technology and Projects team, who delivered hardware and research funding by the briefcaseload. Additionally, I truly appreciate Dom's work coordinating this project; working for and with him has been a privilege, and I would recommend his leadership without reservation.

Producing a book is a team effort, and I am fortunate to have had many eyes reading drafts of my work. Alex Chernyakhovsky, Frans Kaashoek, Chelsea Voss, and Nickolai Zeldovich all found minor errors, but they also offered major and substantive feedback which has seriously influenced the final document. Kendra Beckler's paper expertise was extremely helpful; without her help, I doubt I would have settled on as high-quality paper as is used here.

Finally, I owe the greatest thanks to Chelsea Voss for her friendship and support through the term of this project and beyond.

Contents

Contents 7

1 A Microcontroller-Based Security Peripheral 9

2 Proof Assistants ... and Microcontrollers? 11
- 2.1 Why Not an Application Processor? 11
- 2.2 Why Use Coq and Bedrock? 13

3 Bedrock Specifications and Code 15
- 3.1 Specifications 15
- 3.2 Programming 17

4 Embedded Bedrock 19
- 4.1 Generating Thumb-2 Assembly 19
- 4.2 Implementing the Bedrock Socket Model 20
- 4.3 My Bedrock System: A Platform for Certified Services . . 21

5 Next Steps 29
- 5.1 Feature-Complete and More Secure AES 29
- 5.2 Proving AES Works Correctly 29
- 5.3 Proving Data Are Logged 30
- 5.4 Proving No Socket Is Leaked 30
- 5.5 Making Bedrock Go Faster 30

6 Bedrock Programming: A Retrospective 31

Complete Source Listings 33
- Bedrock 33
- C, C++, and Assembly 46

References 71

Chapter 1

A Microcontroller-Based Security Peripheral

Microcontrollers – low-power, real-mode CPUs – drive digital electronics all over the world. Because microcontroller systems lack virtual memory, however, their software cannot leverage the hardware memory protection common in desktop processors. Fortunately, improvements in static analysis and programming languages allow developers nowadays to statically guarantee memory safety before their software ever hits bare metal. To investigate this methodology, I used the domain-specific Bedrock programming language to program a microcontroller, and I then used the Coq proof assistant to prove my Bedrock code memory safe.

The result is a tiny system that demonstrates microcontroller memory safety is not only possible but also quite valuable. Connected to a computer via a serial line, a microcontroller flashed with my software offers important security primitives – AES encryption and append-only logging – to the host, all the while guaranteeing memory isolation at the level generally offered by a memory management unit. Data sent for logging will not leak into or overwrite data sent for encryption, and vice versa. Bedrock also provides several other assurances, including that the system's memory allocator is correct, the application will never stack-overflow, and no null-pointer dereferences are possible.

While these guarantees are exciting, using Bedrock to program a system as simple as a microcontroller might seem like overkill. Why spend time and effort guaranteeing memory safety statically when a few dollars more can buy an MMU that will do so at runtime? However, as I discuss in chapter 2, my work grows more relevant each day: Microcontroller systems are becoming more and more widespread, and we simply cannot afford to lose the data they protect. Furthermore, while safe programming languages are common, options for safe *system* programming languages are limited; Bedrock is one of the only reasonable choices.

It is certainly not C, though, so I spend chapter 3 reviewing the Bedrock system and filling in some gaps in published material, before turning full attention to my software and proof engineering work in

chapter 4. I examine the Bedrock compilation process and how it interacts with the microcontroller environment; look at the C++ support code I needed to write to bring Bedrock's I/O model to bare metal; and show some of the specifications, code, and proofs that ensure my system works safely. I conclude with a qualitative evaluation of my work and an exploration into the future of safe microcontroller software (chapters 5 and 6).

Chapter 2

Proof Assistants ... and Microcontrollers?

This entire report is about building microcontroller software with the Coq proof assistant and the Bedrock programming language. However, it's likely not obvious *why* I would want to do such a silly thing. Why target a microcontroller when full-featured ARM CPUs are both plentiful and inexpensive? Why use a proof assistant to statically verify memory safety when a garbage-collected language can fill the same role? In short, microcontrollers are more plentiful and less expensive than other CPUs, and Bedrock offers more useful guarantees than garbage-collected languages. In this chapter, I'll examine both questions in greater detail.

2.1 Why Not an Application Processor?

Most computer engineers don't think of microcontrollers as a 'serious' software development platform: Microcontrollers run household appliances, Timex watches, and occasionally, student projects. At the same time, general-purpose CPUs are cheaper than ever and offer substantial benefits over microcontroller systems – notably, hardware memory protection via virtual memory. (I'll discuss this further in the next section.) However, my work targets microcontrollers, for two reasons: They're everywhere, and they're not going away.

Consider, for instance, the SD card, the workhorse of modern portable storage and a poster child for modern engineering. Today, $40 buys me a 32-gigabyte high-speed microSD[1] – a flash peripheral sufficiently powerful to boot Linux and sufficiently small to get lost under loose change. Five years ago, however, $40 bought 8 gigabytes; ten years ago, it was just 1. Fifteen years ago, when I purchased my first flash peripheral, I paid $40 – for 32 *megabytes* of flash in a keychain-sized USB stick. So the last fifteen years have seen a thousandfold increase in capacity and a tenfold decrease in physical volume, not to mention universal proliferation of SD-compatible sockets in workstations, laptops, and phones. Flash memory is a symbol of an age in which storage is so cheap and so small that we can store 300 tonnes of paper in a pocket, next to pennies from the cashier at lunch.[2]

1. The current size records for SD cards are 512 gigabytes for the SD form factor ($500) and 200 gigabytes for the microSD form factor ($400), both produced by industry leader SanDisk.

2. LexisNexis quotes 677,963 pages of text per gigabyte,[3] meaning some 135 million pages (70 million sheets) could fit on the microSD described in note 1. This is 135,000 reams of paper, or 34,000 basis reams; assuming 20 lb. paper, this is 700,000 pounds or 300 tonnes.

3. LexisNexis. 'How Many Pages in a Gigabyte?' URL: http://lexisnexis.com/applieddiscovery/lawlibrary/whitePapers/ADI_FS_PagesInAGigabyte.pdf.

And yet, beneath this monumental exterior, hidden among layers upon layers of precisely fabricated memory, sits a beast waiting to pounce: an embedded microcontroller working around defects in the flash!

> In reality, all flash memory is riddled with defects – without exception. The illusion of a contiguous, reliable storage medium is crafted through sophisticated error correction and bad block management functions. This is the result of a constant arms race between the engineers and Mother Nature; with every fabrication process shrink, memory becomes cheaper but more unreliable. Likewise, with every generation, the engineers come up with more sophisticated and complicated algorithms to compensate for Mother Nature's propensity for entropy and randomness at the atomic scale.
>
> These algorithms are too complicated and too device-specific to be run at the application or OS level, and so it turns out that every flash memory disk ships with a reasonably powerful microcontroller to run a custom set of disk abstraction algorithms. Even the diminutive microSD card contains not one, but at least two chips – a controller, and at least one flash chip (high density cards will stack multiple flash dies)....
>
> The embedded microcontroller is typically a heavily modified 8051 or ARM CPU.[4]

4. Andrew 'bunnie' Huang. 'On Hacking MicroSD Cards'. In: *bunnie:studios* (29 Dec. 2013). URL: http://www.bunniestudios.com/blog/?p=3554

That's right, the scourge of every 6.115 student's waking hours[5] lurks underneath virtually every SD card money can buy, mediating access from our every device to the data we take everywhere. And in the era that we live in – this modern era where storage is consumable and we can always buy more – how much more these microcontrollers must worm their way into our lives every single day!

5. 6.115 is currently MIT's Microcontroller Project Laboratory, in which students complete a number of projects using 8051-compatible microcontrollers.

To be sure, it would be wonderful for microcontrollers to leave common use. Replacing a microcontroller with an application processor promises easier programming and better memory protection, and thereby safer systems. However, microcontrollers aren't going away.

> Amazingly, the cost of adding these controllers to the device is probably on the order of \$0.15–\$0.30, particularly for companies that can fab both the flash memory and the controllers within the same business unit. It's probably cheaper to add these microcontrollers than to thoroughly test and characterize each flash memory chip....[6]

6. Huang, 'On Hacking MicroSD Cards'.

When you're building a million units, the difference between a \$2 application processor and a \$0.30 microcontroller becomes massive – large enough to far outweigh the cost of developing for and testing on the more restricted platform. And this phenomenon isn't limited to SD card manufacturers: today, microcontrollers bear complicated or critical software loads across the board,[7], and thanks to the economics of the

7. Texas Instruments. *Automotive: Low-Cost Anti-Lock Braking System*. 2015. URL: http://www.ti.com/solution/antilock_braking_system.

situation, they will continue to do so for many years. As microcontroller systems embed themselves deeper into the modern world, building software that works and works well on them becomes only more critical.

2.2 Why Use Coq and Bedrock?

Now, one might argue that this proliferation of microcontrollers is actually a panacea. After all, the systems seem to have worked adequately so far, and beyond increased production, they don't seem to be changing all that much. But it's precisely this trend of increased production that worries me – how many SD cards already have been produced with latent firmware bugs, left unexposed even by the most aggressive testing and quality control? How many petabytes distributed on SD cards worldwide will be lost when a tiny error in firmware date-handling code suddenly cascades into a massive memory corruption problem? This situation is well within the realm of possibility – on a system with only 128 kilobytes of RAM, a buffer overflow is pretty much guaranteed to wipe out *something* important. It's a moderate miracle it hasn't happened already.

ARM's Cortex-M3 and M4 microcontrollers feature an interesting hardware-based guard against this sort of problem: the *memory protection unit*. An MPU provides no virtual memory management, but it does provide a memory protection table, which allows software to define hardware-enforced access policies for various memory regions. Standard MPU usage usually relies on a tiny kernel (e.g., FreeRTOS[8] or LK[9]) which sets region controls either for itself (if it is the only process running on the system) or for each environment (if it is also handling context switching). In the latter case, a context switch triggers a change in the memory protection regions, just as a context switch on an application processor generally triggers a change in the virtual memory map. If software violates policy, the microcontroller triggers an interrupt.

Unfortunately, this behaviour is fundamentally a fail-safe, and it leaves open the question of exactly how, having discovered a memory fault, firmware should behave. Should it attempt to check its own integrity? Should it notify the user? Should it simply reboot? No matter what, the microcontroller will spend precious cycles recovering from a memory fault, with potentially disastrous results. (Imagine slamming on the brakes while the anti-lock brake controller is inside a high-priority interrupt service routine.)

A better solution is to prevent memory faults before they even occur, by programming firmware in a garbage-collected language like Chicken[10] or Java.[11,12] This technique, however, produces a dependency on a garbage collector, which must necessarily be written in a programming language supporting unsafe memory operations. This rather frustrates the safety goals of using a garbage-collected language to begin with; it simply moves concern about bugs from application code to support code. Additionally, garbage collection can cause severe performance issues, as garbage-collected languages are generally not designed for embedded applications.[13]

8. *FreeRTOS*. URL: http://www.freertos.org/.

9. Travis Geiselbrecht. *The LK Embedded Kernel*. URL: https://github.com/travisg/lk.

10. *Chicken Scheme: A Practical and Portable Scheme System*. URL: http://call-cc.org/.

11. *Java*. URL: https://java.com/.

12. Attentive readers will note that neither of these languages is truly free from runtime faults; Java throws a `RuntimeError` on null pointer dereferences, and Scheme behaves similarly when inspecting the empty list with `car` or `cdr`. However, programmers can generally handle these exception conditions.

13. Java Mobile Edition is a rare case, a garbage-collected language tailored for embedded systems. (Indeed, Java was originally designed for safe embedded programming.) Today, Java ME supports common microcontroller platforms (the Cortex-M3 and M4)[14] and is a reasonable choice for a number of applications. Safety in any Java code, however, is still predicated on correctness in the complicated Java Virtual Machine and associated garbage collection routines.

14. Oracle. 'Frequently Asked Questions: Oracle Java ME Embedded 8 and 8.1'. URL: http://www.oracle.com/technetwork/java/embedded/javame/embed-me/documentation/me-e-otn-faq-1852008.pdf.

Better than a garbage-collected language is a language that uses static analysis to check for memory-related bugs. Cyclone,[15] for instance, looks like c but restricts pointer arithmetic. Coupled with state-of-the-art analysis techniques, this enables the compiler to eliminate buffer overrun, use-before-initialization, and use-after-free bugs. Nonetheless, it remains unable to statically eliminate all memory problems, and the compiler occasionally must resort to runtime checking, which presents all the same problems as hardware memory protection. (Cyclone also has not been actively developed for nearly a decade.) Rust[16], developed by Mozilla Research, is a more radical departure from traditional c-like programming; its compiler can statically verify memory safety, although it requires the programmer to shoulder substantial complexity. Its compiler is also very immature, which raises the question of compiler correctness analogous to the question of runtime correctness discussed previously. On the other end of the spectrum, CompCert c[17] boasts a compiler with a machine-checkable correctness proof,[18] but the language itself admits unsafe operations, making it unsuitable for high-assurance microcontroller development.

The best microcontroller programming language would not permit the programmer to code unsafely, and it would also have a compiler guaranteed to preserve semantics during compilation. Currently, no language satisfying these requirements exists; however, Bedrock[19,20] comes close. Bedrock is a c-like programming language, but its operations correspond to actions in a formal system of *separation logic*,[21] allowing its compiler to thoroughly check memory safety properties. Furthermore, the compiler itself is implemented inside the Coq proof assistant,[22] and it comes with a machine-checkable proof of semantic preservation from the source language to a linear intermediate representation. Because Coq incorporates a fairly small trusted base (fewer than 20,000 lines of OCaml), we may be confident in the Bedrock compiler's correctness and therefore in the safety of executables it generates. Bedrock is far from perfect – compilation is slow, the compiler lacks an optimizer, i/o is constrained to a socket model, and the system fairly bristles with sharp edges. However, working with Bedrock provides real, powerful memory safety guarantees, offering an unmatched level of confidence in compiled code. For truly safe microcontroller software, it is an excellent option.

15. Trevor Jim et al. 'Cyclone: A Safe Dialect of c'. In: *Proceedings of the 2002 USENIX Annual Technical Conference*. (Monterey, California, 10–15 June 2002). URL: https://www.usenix.org/legacy/event/usenix02/jim.html.

16. *The Rust Programming Language*. URL: http://www.rust-lang.org/.

17. INRIA. *CompCert*. URL: http://compcert.inria.fr/.

18. Xavier Leroy. 'Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant'. In: *SIGPLAN Notices* 41.1 (2006), pp. 42–54.

19. Adam Chlipala. 'The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier'. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. (Boston, 25–27 Sept. 2013). New York: ACM, pp. 391–402. DOI: 10.1145/2500365.2500592.

20. Adam Chlipala. 'From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification'. In: *Proceedings of the 42nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. (Mumbai, 15–17 Jan. 2015). New York: ACM, pp. 609–622. DOI: 10.1145/2676726.2677003.

21. Zhaozhong Ni and Zhong Shao. 'Certified Assembly Programming with Embedded Code Pointers'. In: *Conference Record of the 33rd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. (Charleston, South Carolina, 11–13 Jan. 2006). New York: ACM, pp. 320–333. DOI: 10.1145/1111037.1111066.

22. INRIA. *The Coq Proof Assistant*. URL: https://coq.inria.fr/.

Chapter 3

Bedrock Specifications and Code

Bedrock documentation exists in two primary forms: the official Bedrock tutorial[1] and various papers.[2,3] However, when I first began programming Bedrock, I found both sources quite challenging to get through. I thus give here a gentler introduction to programming with Bedrock, which should prepare the reader to examine my code samples through the rest of this book. In the interest of simplicity, I do not discuss the Bedrock language as deeply as either the tutorial or the papers; I strongly encourage the reader to follow up with close reading of both.

Writing a program in Bedrock consists of three steps: writing a specification, writing code, and proving that the code implements the specification. Writing code is fairly easy, and writing specifications can be challenging but not overwhelming. Proving that code implements the specification, however, can be extremely difficult, and understanding how to create Bedrock proofs is not essential to understanding my work. I therefore refrain from discussing the mechanics of Bedrock proofs in this book, referring the interested reader to the aforementioned documentation.

1. Adam Chlipala. 'The Bedrock Tutorial'. 28 Mar. 2013. URL: http://plv.csail.mit.edu/bedrock/tutorial.pdf.

2. Chlipala, 'The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier'.

3. Chlipala, 'From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification'.

3.1 Specifications

Fundamental to the specification-writing process is realizing that one is programming in two different specification languages at once. The first is Coq's calculus of inductive constructions (CIC), the standard, constructive, higher-order logic Coq programmers are familiar with. The second is XCAP,[4] a specialized *program logic* that allows reasoning about pointer arithmetic. Any CIC predicate can be 'lifted' to an XCAP predicate using barred brackets (e.g., `[| x <> 0 |]`).

4. Ni and Shao, 'Certified Assembly Programming with Embedded Code Pointers'.

In some cases, one only need write CIC specifications. For instance, figure 3.1 illustrates the specification for an addition function, a function which

- accepts two arguments, `n` and `m`;
- uses 0 words of stack space for local variables (function parameters are always stack-allocated);

```
Definition addS :=
SPEC("n", "m") reserving 0
  PRE[V] [| True |]
  POST[R]
    [| R = V "n" ^+ V "m" |].
```

Figure 3.1: Bedrock specification for an addition function, as shown in the Bedrock tutorial.

- has a trivial precondition (`[| True |]`); and
- ensures that the return value is `n + m`.

Note the bound variables `V` and `R`, which refer to the environment at the time of function entry and the function return value, respectively. Thus, `V "n"` is the value of `n` at the time the function begins, and `R = V "n" ^+ V "m"` is the proposition that the function computes the correct value. (`^+` is the addition operator on 32-bit words.)

Frequently, though, one must use some of the many XCAP operators Bedrock defines.

- XCAP supports both universal and existential quantification, indicated in Bedrock with `Al x, P` and `Ex x, P`.
- Bedrock denotes implication with a extra-long double arrow (e.g., `P ===> Q`).
- Bedrock defines operators for discussing memory allocation. `p =?> n` is the proposition that `p` points to an `n`-*word* buffer, and similarly, `p =?>8 n` means `p` points to an `n`-*byte* buffer. (In Bedrock, words are 32 bits.)
- Bedrock also defines operators for inspecting memory contents. `p =*> x` means `p` points to the word `x`, and `p ==*> x, y, …` means that `p` points to a word buffer containing `x, y, …`. `array8 lst p` means that the contents of some CIC byte list `lst` also exist in Bedrock's memory at address `p`.
- Conjunction is indicated with the asterisk. (Actually, the asterisk indicates more than simple conjunction; it also makes a statement about disjointness of memory regions. However, for most Bedrock programming, it's best to think of it as a conjunction.)

```
Definition swapS :=
SPEC("px", "py") reserving 2
  Al x, Al y,
    PRE[V] V "px" =*> x
         * V "py" =*> y
    POST[_] V "px" =*> y
          * V "py" =*> x.
```

Figure 3.2: Bedrock specification for a swap function, as shown in the Bedrock tutorial.

As an example, consider figure 3.2, which specifies an in-place swap function. This function accepts two pointer arguments `px` and `py`, and it swaps the values to which they point. Note the universal quantification over those values (`x` and `y`), as well as the `*` conjunction operator.

Bedrock specifications can get much more complicated than this. Figure 3.3 shows the specification for the POSIX `connect` function, which takes an address buffer and a size and returns a socket representing a connection to that address. The function uses 29 words of stack space and requires that `address` points to a `size`-word buffer. In return, it promises that `address` will still point to a `size`-word buffer at function return and that the function will establish some new set of open file descriptors. (`%in` denotes set membership here, and `%<=` denotes the $\subseteq$ relation.) Furthermore, the function requires (and preserves) abstract invariants `Scheduler.invariant` and `mallocHeap` required by the Bedrock multithreading system and Bedrock memory allocator, respectively. And this is a fairly liberal specification! It does not, for instance, require that the function creates a new file descriptor set (after all, `openFDs` might simply equal `openFDs'`), nor does it require that `address` is a valid IP address.

```
Definition connectS := SPEC("address", "size") reserving 29
Al openFDs,
  PRE[V]
    V "address" =?>8 wordToNat (V "size")
    * Scheduler.invariant openFDs
    * mallocHeap 0
  POST[R]
    V "address" =?>8 wordToNat (V "size")
    * Ex openFDs',
        [| R %in openFDs' |]
        * [| openFDs %<= openFDs' |]
        * Scheduler.invariant openFDs'
        * mallocHeap 0.
```

Figure 3.3: Specification for `connect`.

<table>
<tr><th>C</th><th>Bedrock</th></tr>
<tr><td><code>x = 42;</code></td><td><code>"x" <- 42;;</code></td></tr>
<tr><td><code>x += x;</code></td><td><code>"x" <- "x" + 1;;</code></td></tr>
<tr><td><code>x = *y;</code></td><td><code>"x" <-* "y;;"</code></td></tr>
<tr><td><code>*x = y;</code></td><td><code>"x" *<- "y";;</code></td></tr>
<tr><td><code>mem_init();</code></td><td><code>Call "mem"!"init"()
  [condition];;</code></td></tr>
<tr><td><code>x = rand_int(3, y);</code></td><td><code>"x" <-- Call "rand"!"int"(3, "y")
  [condition];;</code></td></tr>
<tr><td><code>assert(condition);</code></td><td><code>Assert [condition];;</code></td></tr>
<tr><td><code>while (condition) {
  action;
  action;
}</code></td><td><code>[loop invariant]
While (condition) {
  action;;
  action
};;</code></td></tr>
</table>

Figure 3.4: Equivalent forms in C and Bedrock. Despite the inclusion of double semicolons on the Bedrock side, double semicolons separate rather than terminate Bedrock statements. Also, the `<--` when binding to the result of a function call is not a typographical error; it's a consequence of Bedrock's (rather silly) parser.

3.2 Programming

In contrast to the specification language, the Bedrock programming language is remarkably simple.

- Bedrock is generally brace-delimited.
- Bedrock code is packaged into modules. Each module contains imports and functions.
- Each function declares parameters, local variables, and an associated specification. Parameters and variables are indistinguishable in function definitions; the specification controls the function's actual arity.
- Double-semicolons separate rather than terminate statements. (Pascal programmers will find this behaviour familiar.)
- All keywords – `Call`, `While`, `If`, etc. – are capitalized.
- Operators generally mimic C's.

Figure 3.4 summarizes the syntactic differences between Bedrock and C at the level of individual statements.

A number of Bedrock constructs (notably, while loops and function calls) require the programmer to provide explicit XCAP assertions. These go in square brackets and appear very similar to function specifications, though they have slightly different semantics. When used at a call site, a precondition actually indicates a precondition *for the rest of the function* – that is, it indicates a *postcondition* for the function call. A postcondition, on the other hand, indicates an *additional postcondition for the function*, which must be satisfied before the function returns. These semantics are evident in figure 3.5, which illustrates Bedrock code to allocate and free a buffer. The bizarre semantic change results mostly from the Bedrock compilation strategy; in general, when reading Bedrock, it's safe to ignore the postconditions in inline invariants.

Figure 3.5: Bedrock code which allocates and then frees a 32-byte (8-word) buffer. The implementation uses four Bedrock instructions: an `Assert`, a `Call` to `"malloc"!"malloc"`, another `Assert`, and one final `Call` to `"malloc"!"free"`. The implementation assumes the memory allocator has been initialized and stores its heap at address `0` (as indicated by the first `Assert`).

```
Assert
  [PRE[_] mallocHeap 0
   POST[_] mallocHeap 0];;
"buffer" <-- Call "malloc"!"malloc"(0, 8)
  [PRE[_, result]
     [| result <> 0 |]
     * [| freeable result 16 |]
     * result =?> 16
     * mallocHeap 0
   POST[_] mallocHeap 0];;
Assert
  [PRE[V]
     [| V "buffer" <> 0 |]
     * [| freeable (V "buffer") 16 |]
     * V "buffer" =?> 16
     * mallocHeap 0
   POST[_] mallocHeap 0];;
Call "malloc"!"free"(0, "buffer", 8)
  [PRE[_] mallocHeap 0
   POST[_] mallocHeap 0]
```

Chapter 4

Embedded Bedrock

Adam's Bedrock research programme has produced a moderate amount of Bedrock software,[1] but all that software is designed to work atop a well-developed, powerful kernel. To explore development closer to bare metal, I created a Bedrock driver for the ARM Cortex-M3 microcontroller that transforms the M3 into a simple system for secure computation. When activated, my system provides a basic serial-based API for AES encryption[2] and append-only logging, both common security-related tasks. Because the M3 is a widely available, representative microcontroller, working with it provides a powerful window into Bedrock's suitability for general embedded tasks.

1. Chlipala, 'From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification'.

2. United States National Institute of Standards and Technology. 'Specification for the Advanced Encryption Standard'. In: Federal Information Processing Standards Publication 197 (26 Nov. 2001). URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

Building my application proceeded in several phases. I first constructed a new backend for the Bedrock compiler targeting the M3's Thumb-2 instruction set. I then built my software in Bedrock, constructing a C++ runtime system to support my code with the system calls Bedrock expects. Along the way, I constructed proofs that my Bedrock code is 'safe' in the sense described in 'The Bedrock Structured Programming System':[3] that it has no invalid jumps, makes no out-of-bounds memory accesses, and (in the case of the logger) maintains a properly laid out singly-linked list.

3. Chlipala, 'The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier'.

4.1 Generating Thumb-2 Assembly

At the start of this project, Bedrock supported only the 32- and 64-bit Intel architectures. However, ARM's CPUs use their own 32-bit RISC instruction set, and the Cortex-M3 uses an even more exotic 16-bit encoding known as Thumb-2. (Figure 4.1 provides an extensional comparison between Intel-64 and Thumb-2.) To run code on the M3, I thus needed to extend the Bedrock compiler to produce Thumb-2.

```
fact:   mov $1, %rax
loop:   test %rdi, %rdi
        je done
        imul %rdi, %rax
        dec %rdi
        jmp loop
done:   ret
```

(a)

```
fact:   mov r3, r0
        movs r0, #1
loop:   cbz r3, done
        muls r0, r3, r0
        subs r3, r3, #1
        b loop
done:   bx lr
```

(b)

Figure 4.1: Factorial function implementations in Intel-64 (a) and Thumb-2 (b) instruction sets. While Thumb-2 is a 32-bit architecture, it uses a register calling convention and is thus more directly comparable to 64-bit Intel machines than 32-bit ones.

The Bedrock IL was designed for Intel systems: it requires only three registers, and it relies on extremely powerful single instructions (e.g., memory-to-memory moves). Translating CISC Bedrock IL to the RISC Thumb-2 instruction set thus produces unavoidable overhead: each Bedrock IL instruction expands to as many as seven Thumb-2 ones. I use six general-purpose registers – three to store the Bedrock IL registers and three as scratchpad temporaries during intermediate

Figure 4.2: Examples of compiling the Bedrock IL. These examples are non-comprehensive, but they are sufficient to illustrate all concepts of the translation. To save space, I have written _heap instead of `bedrock_heap`.

description	Intel-32	Thumb-2
assign constant to register	`mov $42, %eax`	`ldr r3, =42` `mov r0, r3`

multiply register by constant	`imul $42, %ebx`
	`mov %ebx, %eax`
		`mov r3, r2`
		`ldr r4, =42`
		`mul r3, r3, r4`
		`mov r0, r3`

direct branch	`jmp L`	`mov r5, =L` `mov pc, r5`
indirect branch through register	`mov %eax, %edx` `jmp *%edx`	`mov r3, r0` `mov pc, r3`

indirect branch through memory	`mov _heap(%eax), %edx`
	`jmp *%edx`
		`ldr r5, =_heap`
		`add r5, r5, r0`
		`ldr r3, r5`
		`mov pc, r3`

conditional direct branch if less than or equal to	`cmp %eax, %ebx`
	`jb L1`
	`jmp L2`
		`mov r3, r0`
		`mov r4, r1`
		`cmp r3, r4`
		`ite lo`
		`ldrlo r5, =L1`
		`ldrhi r5, =L2`
		`mov pc, r5`

computation. Furthermore, while Thumb-2 has some support for conditional branches (as demonstrated in figure 4.1), it is insufficiently general to implement Bedrock's conditional branch mechanism; I therefore opted to emit `ite` instructions (which cause conditional execution) followed by unconditional jumps. Figure 4.2 contains a non-comprehensive comparison of instructions emitted on Intel platforms and instructions emitted by my backend.

I did not prove my backend preserves the semantics of the Bedrock IL when translating it to Thumb-2. However, this is par for the course as far as Bedrock backends are concerned: without a formal semantics for the target architecture, proving semantic preservation is impossible! Like the other Bedrock backends, mine is small and easy to understand – sufficiently simple as to be obviously correct. I ran into only one bug during development, in which I misunderstood the semantics of the Bedrock IL and accidentally emitted signed comparison instructions instead of unsigned ones.

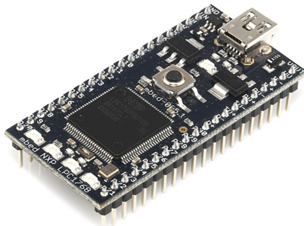

Figure 4.3: My microcontroller platform, the mbed LPC1768, is available for about $50 on SparkFun and offers a Cortex-M3 microprocessor clocked at 96 MHz, 512 kiB of flash memory, 32 kiB of RAM, and an FTDI USB controller.

4.2 Implementing the Bedrock Socket Model

Code generation is enough to run basic Bedrock programs, but any serious system requires input and output – which, as mentioned in chapter 2, Bedrock assumes is socket-based. However, my Cortex-M3 development board, the mbed LPC1768 (figure 4.3), provides only serial input and output, one layer lower in the OSI protocol stack[4] than socket-based communication. I therefore needed to implement some C++ runtime system that would provide socket-based I/O primitives to my Bedrock code, yielding the system illustrated in figure 4.4. My

4. Hubert Zimmerman. 'OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection'. In: *IEEE Transactions on Communications* 28.4 (Apr. 1980), pp. 425–432. DOI: 10.1109/TCOM.1980.1094702.

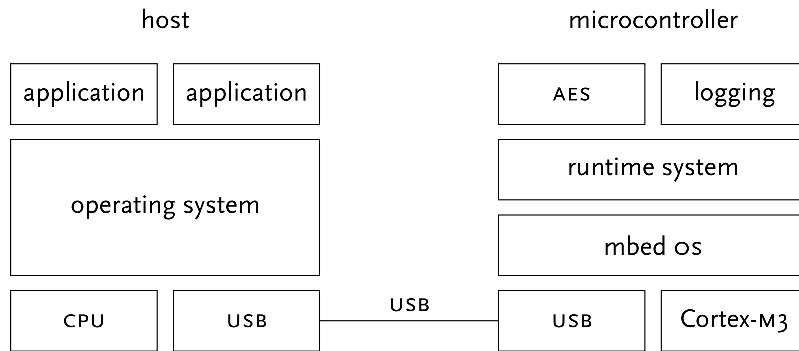

Figure 4.4: Overall system architecture, indicating components running on both the host CPU and the microcontroller. I wrote the AES and logging applications on the microcontroller, as well as the microcontroller's runtime system. That runtime system relies on the mbed OS to mediate USB communication with the host.

runtime system runs atop the mbed OS, a small C++ embedded operating system available publicly from ARM.

While I could have built some well-known protocol (e.g., PPP) into my design, I instead implemented something simpler. Its fundamental data unit is the *message* (figure 4.5), a frame containing a 32-bit session identifier, a 32-bit payload size, and an arbitrary-length payload. The session identifier is used by the endpoints to group messages into *sessions*, streams of related data as would be handled by a POSIX-style socket; clients select a session identifier when connecting. The session identifier also does double duty to associate incoming messages with either the AES service or the logging service – a session identifier with a zero low-order bit indicates a message bound for encryption, while a session identifier with a one low-order bit indicates data to be logged.

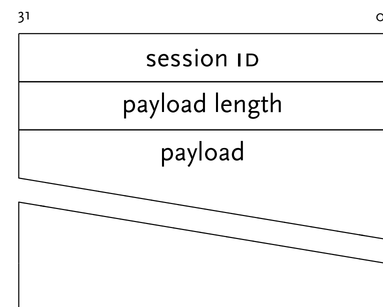

Figure 4.5: Schematic for a message in my frame-based communication protocol.

In my system, the microcontroller plays the role of the server, and the host plays the role of the client. The microcontroller listens for connections; when the host transmits data, my protocol driver interprets its frame metadata and queues it for the appropriate socket, where Bedrock can retrieve it via a `read` system call. Conversely, when Bedrock `write`s to a socket, my protocol driver captures the data, associates metadata with it, and transmits it down the serial line.

4.3 My Bedrock System: A Platform for Certified Services

With frame-based protocol and Thumb-2 code generator in hand, only one component of the system remains: my Bedrock code itself. My code, upon startup, establishes a serial connection with a host computer and waits for data. Depending on the data content, the system either responds with an AES-encrypted version of the data, or it simply logs the data to memory. The AES cryptosystem is simple in implementation but makes heavy use of Bedrock's I/O primitives; the logging system is more complicated, but it gives correspondingly stronger guarantees about its behaviour.

Booting the Microcontroller

When the microcontroller boots, its firmware loads the mbed OS, which saves me the trouble of setting up the microcontroller – installing

interrupt vectors, initializing serial communication over USB, and setting up the C++ memory allocator. Once the microcontroller is up and running, it executes a tiny C++ shim (figure 4.6) that statically allocates a scratchpad buffer for Bedrock, switches from the Thumb-2 calling convention to the Bedrock one, and jumps into Bedrock code.[5]

5. I have optimized this code snippet, like all I present in this section, for readability by removing uninteresting comments and eliding uninteresting assertions or debugging `printf`s. My full code is available at the end of this report.

Bedrock's first order of business is to partition the amorphous blob of memory provided by the C++ shim into heap, stack, and globals regions (figure 4.8). Figure 4.7 shows the code required to do so. There is only one global – `schedule`, which points to a data structure used by the multithreading system. Stack usage is also fairly conservative in my system; Bedrock needs fewer than 100 words of stack space. This allows for a large heap region used primarily in the logging component of the system; the last action Bedrock takes before spawning application threads is to initialize the Bedrock memory allocator on that heap. This allocator, distinct from the mbed OS's memory allocator, is implemented as part of the Bedrock standard library[6] and is, like the rest of my Bedrock code, verified safe with Coq.

6. Chlipala, 'The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier'.

Figure 4.6: Entering Bedrock from C++. At this point, `bedrock_heap` (an unfortunately-named variable which actually points to the entire memory region reserved for Bedrock's use) has already been initialized to point to a valid, statically allocated buffer.

```
uint32_t result;
__asm__ __volatile__(
  // Tell Bedrock how much heap it has.
  "mov r0, %[heap_size]\n\t"
  // Tell Bedrock where to branch after it's done working.
  "adr r1, 0f\n\t"
  // Branch into Bedrock code.
  "b main_main\n\t"
  "0:\n\t"
  // Save the result from Bedrock's main.
  "mov %[result], r2"
  : [result]"=X"(result)
  : [heap_size]"X"(bedrock_environment::kHeapSlots),
    "m"(*bedrock_heap)
  : "r0", "r1", "r2", "r3", "r4", "r5", "r6", "lr",
    "cc");
return result;
```

Figure 4.7: Bedrock initialization. The `"mainPrime"!"main"` goto at the end of this code jumps to a simple function which spawns application threads.

```
Definition m :=
bimport [[ "malloc"!"init" @ [Malloc.initS],
           "mainPrime"!"main" @ [MainPrime.mainS] ]]
bmodule "main" {{
  bfunctionNoRet "main"()
      [Bootstrap.bootS MemoryLayout.heapSize 1]
    (* Set the stack pointer. *)
    Sp <- (MemoryLayout.heapSize * 4)%nat;;
    (* Ensure globals are set correctly and initialize the
    allocator. *)
    Assert
      [PREmain[_] MemoryLayout.schedule =?> 1
                  * 0 =?> MemoryLayout.heapSize];;
    Call "malloc"!"init"(0, MemoryLayout.heapSize)
      [PREmain[_] MemoryLayout.schedule =?> 1
                  * mallocHeap 0];;
    (* Pass off control to the application. *)
    Goto "mainPrime"!"main"
  end
}}.
```

Because this area of my code bridges the gap between C++ and Bedrock, it requires a few hand-checked assumptions to prove safety (figure 4.11) – notably, that the C++ shim actually allocates the correct amount of memory. With these assumptions in hand, however, it's easy to prove that this Bedrock code and everything it calls is correct.

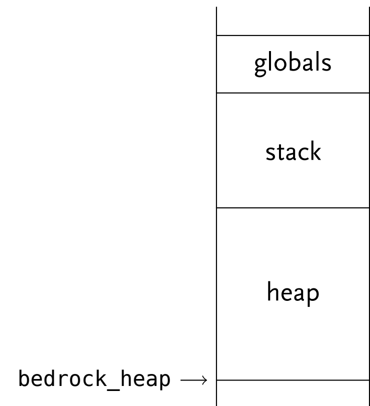

Figure 4.8: Bedrock memory layout. Memory addresses increase as one proceeds vertically up the page, and memory not explicitly marked is reserved for C++.

AES Encryption

The first thread my Bedrock code spawns is an AES encryption service, the core of which is described by the specification in figure 4.9. It accepts sixteen-byte blocks (the native size of the AES cipher) over a socket and returns encrypted versions of same; however, since I have not reified AES within Coq, I can only specify that encryption preserves the size of the block. I implement this specification using the code in figure 4.12, and I present its proof in figure 4.10.

In the initial planning phase for this project, I hoped to work with a microcontroller system with built-in AES hardware. Unfortunately, acquiring appropriate hardware was challenging, and I ended up using a more general-purpose microcontroller instead. However, I still constructed my AES service with a view toward hardware encryption, and it uses the Bedrock socket interface to abstract a handle to the hypothetical AES hardware. Thus, whenever the service receives a sixteen-byte block, it immediately requests a new outgoing connection to the address "AES" and writes the data to the resulting socket.

However, that socket is only a socket in the most limited sense of the word. It's actually a special handle that shares the same namespace as

```
Definition encryptS := SPEC("block") reserving 38
PRE[V] V "block" =?>8 16 (* 128 bit AES block *)
POST[_] V "block" =?>8 16.
```

Figure 4.9: AES encryption routine specification. Because I must treat the AES operation abstractly, this specification is rather weak.

```
Local Hint Extern 2 =>
  match goal with
    | [ H : Datatypes.length _ = _
        |- context[Datatypes.length] ] => rewrite H
  end.

Local Hint Extern 1 (_ = _) => words.

Local Ltac finish :=
  try match goal with
        | [ |- context[please_unfold_buffer] ] =>
          unfold buffer
      end;
  try match goal with
        | [ |- context[please_unfold_buffer_joinAt] ] =>
          unfold Arrays8.buffer_joinAt
      end;
  TAThread.sep EncryptionTacPackage.hints;
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

Figure 4.10: Proof that my AES routine is safe.

Figure 4.11: Top-level safety proof. The theorem at the end, `safe`, is that my entire Bedrock program is safe relative to the C++ runtime system; it is, of course, subject to the previous assumptions.

```
Hypothesis heapSizeLowerBound :
  (3 <= MemoryLayout.heapSize)%nat.

Hypothesis requiredMemoryIsReasonable :
  goodSize (MemoryLayout.wordsAvailable * 4)%nat.

Let heapSizeUpperBound : goodSize (MemoryLayout.heapSize * 4).
Proof.  goodSize.  Qed.

Variable mySettings : settings.
Variable myProgram : program.

Hypothesis labelsInjective :
  forall label1 label2 address,
    Labels mySettings label1 = Some address
    -> Labels mySettings label2 = Some address
    -> label1 = label2.

Hypothesis labelsAgreeWithBlocks :
  forall label pre block,
    LabelMap.MapsTo label (pre, block) (XCAP.Blocks m)
    -> exists address,
         Labels mySettings label = Some address
         /\ myProgram address = Some block.

Hypothesis noCodeShadowing :
  forall label pre,
    LabelMap.MapsTo label pre (XCAP.Imports m)
    -> exists address,
         Labels mySettings label = Some address
         /\ myProgram address = None.

Hypothesis noSysShadowing :
  forall label address,
    Labels mySettings ("sys", label) = Some address
    -> myProgram address = None.

Variable entryPointAddress : W.
Hypothesis atStart :
  Labels mySettings ("main", Global "main")
  = Some entryPointAddress.

Variable executionState : state.

Hypothesis memUpperBound :
  forall address,
    (address < MemoryLayout.wordsAvailable * 4)%nat
    -> executionState.(Mem) address <> None.

Hypothesis memLowerBound :
  forall address,
    $(MemoryLayout.wordsAvailable * 4) <= address
    -> executionState.(Mem) address = None.

Theorem safe : sys_safe mySettings
                        myProgram
                        (entryPointAddress, executionState).
Proof.  safety ok.  Qed.
```

```
Definition m :=
bimport [[ "scheduler"!"close" @ [closeS],
           "scheduler"!"connected" @ [connectedS],
           "scheduler"!"read" @ [readS],
           "scheduler"!"write" @ [writeS],
           "taConnect"!"connect" @ [TAConnect.connectS] ]]
bmodule "encryption" {{
  bfunction "aesEncrypt"("block", "cryptoChannel")
      [Encryption.encryptS]
    (* Connect to the AES service. *)
    "cryptoChannel" <-- Call "taConnect"!"connect"(""AES"", 3)
      [Al openFDs,
         PRE[V, R] [| R %in openFDs |] * V "block" =?>8 16
         POST[_] Ex openFDs', [| openFDs %<= openFDs' |]
                                 * V "block" =?>8 16];;
    (* Wait until we're actually connected. *)
    Call "scheduler"!"connected"("cryptoChannel")
      [Al openFDs,
         PRE[V] [| V "cryptoChannel" %in openFDs |]
                * V "block" =?>8 16
         POST[_] Ex openFDs', [| openFDs %<= openFDs' |]
                                 * V "block" =?>8 16];;
    (* Send the data block to the (fake) AES hardware. *)
    Call "scheduler"!"write"("cryptoChannel", "block", 16)
      [Al openFDs,
         PRE[V] [| V "cryptoChannel" %in openFDs |]
                * V "block" =?>8 16
         POST[_] Ex openFDs', [| openFDs %<= openFDs' |]
                                 * V "block" =?>8 16];;
    (* Get the encrypted block back. *)
    Call "scheduler"!"read"("cryptoChannel", "block", 16)
      [Al openFDs,
         PRE[V] [| V "cryptoChannel" %in openFDs |]
                * V "block" =?>8 16
         POST[_] Ex openFDs', [| openFDs %<= openFDs' |]
                                 * V "block" =?>8 16];;
    Call "scheduler"!"close"("cryptoChannel")
    [Al openFDs,
       PRE[V] V "block" =?>8 16
       POST[_] Ex openFDs', [| openFDs %<= openFDs' |]
                               * V "block" =?>8 16];;
    Return 0
  end
}}.
```

Figure 4.12: AES encryption routine in Bedrock. Bedrock does not actually support the doubly-quoted string literal syntax I use in the call to `connect` (`""AES""`), which makes this call more complicated in actual code.

sockets; writes to the handle invoke AES encryption, and reads from the handle return buffered results. This special semantics is provided by a bit of extra code in my C++ runtime (figure 4.13), which calls out to a trivial AES implementation furnished by Inverse Limit. As this is a prototype, I attempted to ease the coding burden by using electronic codebook mode with a fixed key. In future systems based on this, of course, one would use a stronger cipher mode like cipher block chaining or an authenticated encryption mode like Galois/counter mode, and one would provide an index into a key array stored in the device's nonvolatile memory.

After writing the plaintext block to the AES handle, Bedrock then performs a blocking, sixteen-byte read on the handle. It sends the results – the ciphertext – back to the client and resumes waiting for more. My C++ library, like any sane operating system, ensures that data transmitted to the microcontroller in the interim are appropriately collated and buffered; Bedrock therefore never risks losing data because it's not listening.

Append-Only Logging

While AES encryption is useful, my implementation is little more than a demonstration of Bedrock's I/O primitives. In contrast, my append-only logging service, implemented in figure 4.15, is much more involved and uses substantially more complicated assertions. The result is a trustworthy append-only log system which accepts data blocks and dumps them into a linked list. I prove the trustworthiness of the service in figure 4.14.

The append-only log, like the AES appliance, listens continually for incoming data from the host. (For simplicity's sake, I assume incoming data blocks for the logger, like incoming plaintext blocks for the encryption system, are sixteen bytes in length.) Upon receiving data, the logger performs a standard cons operation: it allocates a cons cell, setting the car to point to the received data and the cdr to point to the old head of the list. It then updates the list pointer to refer to the new head. Currently, logs disappear when power is removed; however, it's feasible to store logs in nonvolatile memory for later administrative retrieval.

```
uint32_t SysWrite(const uint32_t fd, void* const buffer,
                  const uint32_t n_bytes) {
  std::shared_ptr<Handle> handle =
    Handle::DecodeFromBedrock(fd);
  if (handle->IsAESHandle()) {
    if (n_bytes != AES_128_KEY_SIZE) {
      error("write: Invalid AES block size.\r\n");
    }
    AES::Block key =
      {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    AES::Block plaintext;
    std::memcpy(plaintext.data(), buffer, AES_128_KEY_SIZE);
    // Encrypt plaintext, storing the result in a static
    // buffer for SysRead to send back.
    AES::Encrypt(key, plaintext);
    return AES_128_KEY_SIZE;
  } else {
    assert(handle->IsSessionHandle());
    auto session = dynamic_cast<SessionHandle*>(handle.get())
                   ->GetSession();
    if (!session) {
      error("write: Invalid FD.");
    }
    uint8_t* const byte_buffer =
      reinterpret_cast<uint8_t*>(buffer);
    return session->Write(
             ByteVector(byte_buffer, byte_buffer + n_bytes));
  }
}
```

Figure 4.13: C++ implementation of the Bedrock `write` system call; the `read` implementation is analogous. The code sorts file descriptors into AES handles and handles associated with sessions and dispatches appropriately. Note the fixed key (all zeroes) used for AES encryption.

```
Local Hint Extern 2 (_ %in _) =>
  eapply incl_mem;
  [ eassumption | solve [eauto using incl_trans] ].

Local Ltac finish :=
  try match goal with
        | [ |- context[please_unfold_buffer] ] =>
          unfold buffer
      end;
  TAThread.sep BufferList.hints;
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

Figure 4.14: Proof that my logging routine is safe.

Figure 4.15: Append-only logging service. I have elided the import clauses for this Bedrock module. The `sll lst (V "list")` predicate (called `BufferList.singlyLinkedList` in the actual source) asserts that the Bedrock pointer `list` points to a singly-linked list whose contents are equivalent to the Coq term `lst`. However, Bedrock's I/O model does not allow quantification over data that it reads, making it impossible to guarantee all data read actually make it into the log; I discuss this issue in section 5.3.

```
Definition m :=
bimport [[ … ]]
bmodule "logging" {{
  bfunctionNoRet "main"("cell", "inputBuffer", "inputSocket",
                      "list", "listenSocket")
      [Logging.mainS]
    "list" <- 0;;
    "listenSocket" <-- Call "taListen"!"listen"(1)
      [Al openFDs, Al lst,
         PREmain[V, R] [| R %in openFDs |]
                      * sll lst (V "list")];;
    [Al openFDs, Al lst,
       PREmain[V] [| V "listenSocket" %in openFDs |]
                 * sll lst (V "list")]
    While (0 = 0) {
      (* Get a new connection and read in the data. *)
      "inputSocket" <--
        Call "scheduler"!"accept"("listenSocket")
        [Al openFDs, Al lst,
           PREmain[V, R] [| R %in openFDs |]
                        * [| V "listenSocket" %in openFDs |]
                        * sll lst (V "list")];;
      "inputBuffer" <-- Call "buffers"!"bmalloc"(4)
        [Al openFDs, Al lst,
           PREmain[V, R] [| R <> 0 |]
                        * [| freeable R 4 |] * R =?>8 16
                        * [| V "inputSocket" %in openFDs |]
                        * [| V "listenSocket" %in openFDs |]
                        * sll lst (V "list")];;
      Call "scheduler"!"read"("inputSocket",
                              "inputBuffer", 16)
        [Al openFDs, Al lst,
           PREmain[V] V "inputBuffer" =?>8 16
                     * [| V "inputBuffer" <> 0 |]
                     * [| V "inputSocket" %in openFDs |]
                     * [| V "listenSocket" %in openFDs |]
                     * sll lst (V "list")];;
      (* Close the connection so we don’t have to carry its
      invariants around. *)
      Call "scheduler"!"close"("inputSocket")
        [Al openFDs, Al lst,
           PREmain[V] V "inputBuffer" =?>8 16
                     * [| V "inputBuffer" <> 0 |]
                     * [| V "listenSocket" %in openFDs |]
                     * sll lst (V "list")];;
      (* Allocate a cons cell for the new data. *)
      "cell" <-- Call "malloc"!"malloc"(0, 2)
        [Al openFDs, Al lst,
           PREmain[V, R] [| R <> 0 |] * [| freeable R 2 |]
                        * R =?> 2 * V "inputBuffer" =?>8 16
                        * [| V "inputBuffer" <> 0 |]
                        * [| V "listenSocket" %in openFDs |]
                        * sll lst (V "list")];;
      "cell" *<- "inputBuffer";;
      "cell" + 4 *<- "list";;
      "list" <- "cell";;
      Note [please_unfold_buffer]
    }
  end
}}.
```

Chapter 5

Next Steps

The system I described in the previous chapter works – I've checked encryption results against the official NIST AES test vectors,[1] and I can trace the logger to verify that it actually logs incoming data. However, the system remains a research prototype, and much more work is needed before it is ready for use in production.

1. United States National Institute of Standards and Technology, 'Specification for the Advanced Encryption Standard'.

5.1 Feature-Complete and More Secure AES

Currently, the system does not support decryption. However, implementing it would not be difficult. I'd allow Bedrock to connect to two AES addresses – `"AESEncrypt"` and `"AESDecrypt"` instead of just `"AES"` – and then extend the C++ runtime to execute decryption routines appropriately.

The system also currently only permits electronic codebook mode and uses a fixed key. Both of these problems should be fixed, though solutions are a bit more challenging. The runtime will have to change quite a bit to support a better cipher mode, as it will need to track state and initialization vectors. (Currently, the cryptosystem is stateless.) Similarly, supporting multiple keys requires the cryptosystem to keep track of what key is in use, and it should provide Bedrock some mechanism to switch keys.

5.2 Proving AES Works Correctly

The specification I showed in figure 4.9 is disappointing to any verification connoisseur: the encryption function merely guarantees it does not change the block size during encryption, rather than providing any kind of functional specification.

Rectifying this should be a straightforward proof engineering task, but it would not come without substantial effort. The basic idea would be to build a trivial (obviously correct) AES implementation in Coq, and then prove it extensionally equal to some Bedrock implementation. I refrained from attempting this, as it appeared far more difficult than it would be worth and would distract from the more basic theorem of statically guaranteed system memory safety.

Naturally, formalizing AES in Coq would only prove functional correctness; it would not provide any guarantees regarding side channels. However, attempting to prove properties about side channels using a proof assistant is a substantial research topic in its own right, and methodology for doing so is not yet fully developed.

5.3 Proving Data Are Logged

Similarly, the invariants in figure 4.15 are almost totally unbound from the actual data being logged – the only binding is a weak and abstract quantification over the data the logger holds in its linked list. I'd rather prove a much stronger relation between the data which the logger receives and the data the logger stores in the list, but unfortunately, Bedrock currently does not allow quantifying over data passing through its socket interface.

I'm insufficiently familiar with Bedrock internals to estimate the work required to correct this, but it would likely involve rethinking the Bedrock I/O model and possibly providing stronger axiomatic specifications for its system calls.

5.4 Proving No Socket Is Leaked

```
Definition closeGS : spec :=
SPEC("fr") reserving 11
  Al fs,
    PRE[V] [| V "fr" %in fs |]
           * sched fs
           * mallocHeap 0
    POST[_] sched fs
            * mallocHeap 0.
```

Figure 5.1: Axiomatized Bedrock specification for `close`. The specification requires that the input file descriptor is in fact open, but its postcondition does not specify that the file descriptor is closed.

Stronger system call specifications would also allow me to prove more general theorems about resource safety. For instance, consider the specification of `close` in the Bedrock master (figure 5.1). While this is a valid specification for `close`, it doesn't actually state the critical functional property any implementation of `close` should satisfy: that it actually closes the file descriptor passed as an argument! This specification even allows double-closing the same file descriptor, a potential nightmare in a systems language.

Correcting this issue would definitely require deep changes to the Bedrock I/O model: one would need to introduce the notion of file descriptor state into the logic. On the other hand, one could use this work to statically guarantee obedience to the POSIX socket state machine model – e.g., by requiring that a socket is in the listening state before it gets passed to `accept`.

5.5 Making Bedrock Go Faster

Bedrock is in perpetual need of experienced Coq developers to make its automation run faster, but the code the Bedrock compiler generates is also quite inefficient. In fact, the Bedrock compiler has no optimizer at all. Building optimization passes into Bedrock would improve generated code performance substantially – there's a lot of low-hanging fruit, and proving optimization passes correct is an interesting and generally useful body of work.

Chapter 6

Bedrock Programming: A Retrospective

In addition to producing an interesting piece of software, my work over the last year has been enlightening regarding Bedrock's suitability for general-purpose programming. It is certainly a powerful system, and I got done what I needed to get done. But programming and proving with Bedrock is frustrating and complicated at times – it's definitely not for the faint of heart.

It's also not for the newcomer to the field: programmers need both familiarity with Coq and comfort with separation logic to write Bedrock code. When programming Bedrock, one programs inside Coq using Coq's parser and Coq's type checker, so errors are extremely coarse; parse errors manifest as Coq rejecting an entire Bedrock module, leaving the programmer to carefully check their source code for minor typographical errors. References to undefined variables or functions, rather than producing informative compiler output, simply cause proof automation to fail. Furthermore, Bedrock's integration with separation logic could be substantially improved. Current versions of the system require exhaustive annotations at function call sites to properly track invariants, and again, failure to properly annotate calls results in massive unsolved proof goals. And then there are a whole host of minor infelicities: The programmer must manually enter stack space constraints which could be inferred, Bedrock variables must be entered as strings to appease the Coq parser, and the quantity of symbolic operators Bedrock defines is simply massive.

However, Coq knowledge and high confusion threshold are still not quite sufficient to properly program Bedrock: One also needs a lot of time. The extant proof automation is exceedingly slow, with compilation times of over five minutes standard for even my simple embedded systems work. Much of Bedrock's speed problems arise because Bedrock turns Coq to a novel use – Coq was designed for formalizing mathematics with limited automation, not proving programs correct via large-scale proof search. Even with the specialized reflection library Bedrock uses for proof automation,[1] proofs are slow, and iterative development is very difficult.

1. Gregory Malecha, Adam Chlipala and Thomas Braibant. 'Compositional Computational Reflection'. In: *Interactive Theorem Proving. 5th International Conference, ITP 2014.* (Vienna, 14–17 July 2014). Ed. by Gerwin Klein and Ruben Gamboa. Lecture Notes in Computer Science 8558. Springer, 2014. DOI: 10.1007/978-3-319-08970-6.

However, even with these issues, programming in Bedrock was often exciting. Andrew Appel calls Coq 'the world's best video game',[1] and he's right: I've rarely felt as excited or as accomplished as I have when finally proving a difficult theorem in Coq. And Bedrock's programming model is actually rather nice; it is truly a portable assembly language, and it's easy to reason about how Bedrock code will behave. (Contrast this with C, which purports to be a portable assembly but constantly resorts to undefined or unexpected behaviour to achieve this goal.)

With Adam's help throughout this project, I was able to specify and prove most of what I wanted: The limitations were in the current implementation of Bedrock, not in the underlying language model. Bedrock thus provides a tantalizing look into a future in which proofs of memory safety in embedded systems – or, more generally, strong proofs about program behaviour – are easy and fun to produce. The next iteration of Bedrock no doubt will be better than the current one. For the time being, though, proving memory safety in the embedded context remains a highly challenging but quite rewarding pursuit.

1. Andrew Appel. *Software Verification*. Lecture series. At: *The Oregon Programming Languages Summer School*. (Eugene, Oregon, 16–28 June 2014). URL: https://www.cs.uoregon.edu/research/summerschool/summer14/.

Complete Source Listings

Bedrock

BufferList

```
(** * Lists of buffers

Theorems about singly-linked lists of buffers. *)

Require Import PreAutoSep.  Local Open Scope Sep_scope.

(** List invariant.  [singlyLinkedList lst headPointer] states that at
[headPointer], there is a valid singly-linked list containing the same objects
as the Gallina term [lst]. *)

Fixpoint singlyLinkedList (lst : list (list B)) (headPointer : W) : HProp :=
  match lst with
    | nil => [| headPointer = 0 |]
    | buffer :: buffers =>
      [| headPointer <> 0 |]
      * Ex car, Ex cdr,
          (headPointer ==*> car, cdr)
          * array8 buffer car * singlyLinkedList buffers cdr
  end.

Section Facts.
  Variable lst : list (list B).
  Variable headPointer : W.

  Theorem extensional : HProp_extensional (singlyLinkedList lst headPointer).
  Proof.  destruct lst; reflexivity.  Qed.

  (** If the head pointer is a null pointer, the only way for the list to be
  valid is for it to be empty. *)

  Section Empty.
    Hypothesis headPointerNull : headPointer = 0.

    Theorem nil1 : singlyLinkedList lst headPointer ===> [| lst = nil |].
    Proof.  destruct lst; sepLemma.  Qed.

    Theorem nil2 : [| lst = nil |] ===> singlyLinkedList lst headPointer.
    Proof.  destruct lst; sepLemma.  Qed.
  End Empty.

  (** If the head pointer is nonnull, the only way for the list to be valid is
  if its cdr is also valid list. *)

  Section Nonempty.
    Hypothesis headPointerNonnull : headPointer <> 0.

    Theorem cons1 :
      singlyLinkedList lst headPointer
      ===> Ex car, Ex buffer, Ex buffers,
             [| lst = buffer :: buffers |]
             * Ex cdr,
                 (headPointer ==*> car, cdr)
                 * array8 buffer car * singlyLinkedList buffers cdr.
    Proof.  destruct lst; sepLemma.  Qed.

    Theorem cons2 :
      (Ex car, Ex buffer, Ex buffers,
```

```
        [| lst = buffer :: buffers |]
        * Ex cdr,
            (headPointer ==*> car, cdr)
            * array8 buffer car * singlyLinkedList buffers cdr)
      ===> singlyLinkedList lst headPointer.
    Proof.
      destruct lst; sepLemma;
      match goal with
        | [ H : _ :: _ = _ :: _ |- _ ] => injection H; sepLemma
      end.
    Qed.
  End Nonempty.
End Facts.

Module Import Hints.
  Hint Immediate extensional.
End Hints.

Definition hints : TacPackage.
  prepare (nil1, cons1) (nil2, cons2).
Defined.
```

Encryption

```
Require Import Scheduler.

Require TAThread.

Definition encryptS := SPEC("block") reserving 38
  Al openFDs,
    PRE[V]
      V "block" =?>8 16 (* 128 bit AES block *)
      * TAThread.invariant openFDs * mallocHeap 0
    POST[_]
      Ex openFDs’,
        [| openFDs %<= openFDs’ |]
        * V "block" =?>8 16
        * TAThread.invariant openFDs’ * mallocHeap 0.
```

EncryptionImplementation

```
Require Arrays8 Buffers Scheduler.  Import Scheduler.

Require Encryption EncryptionTacPackage TAConnect TAThread.  Import TAThread.

Inductive please_unfold_buffer : Prop := PleaseUnfoldBuffer.
Hint Constructors please_unfold_buffer.

Inductive please_unfold_buffer_joinAt : Prop := PleaseUnfoldBufferJoinAt.
Hint Constructors please_unfold_buffer_joinAt.

Definition m :=
  bimport [[ "buffers"!"bfree" @ [Buffers.bfreeS],
             "buffers"!"bmalloc" @ [Buffers.bmallocS],
             "scheduler"!"close" @ [closeS],
             "scheduler"!"connected" @ [connectedS],
             "scheduler"!"read" @ [readS],
             "scheduler"!"write" @ [writeS],
             "taConnect"!"connect" @ [TAConnect.connectS] ]]
  bmodule "encryption" {{
    bfunction "aesEncrypt"("block", "cryptoChannelName",
                           "cryptoChannel") [Encryption.encryptS]
      (* Bedrock doesn’t support string literals yet, so create a buffer to hold
      the string "AES".  We only need to allocate 3 bytes, but bmalloc must must
      allocate at least 8. *)
      Note [please_unfold_buffer];;
      "cryptoChannelName" <-- Call "buffers"!"bmalloc"(2)
        [Al openFDs, Al cryptoChannelName,
           PRE[V, R]
             V "block" =?>8 16
             * [| R <> 0 |]
             * [| freeable R 2 |]
             * array8 cryptoChannelName R
             * [| length cryptoChannelName = 8 |]
             * [| natToW 0 < natToW (length cryptoChannelName) |]%word
             * TAThread.invariant openFDs * mallocHeap 0
           POST[_]
```

```
      Ex openFDs',
        [| openFDs %<= openFDs' |]
        * V "block" =?>8 16
        * TAThread.invariant openFDs' * mallocHeap 0];;
(* Write "AES" into the buffer. *)
"cryptoChannelName" + 0 *<-8 65 (* A *);;
Assert
  [Al openFDs, Al cryptoChannelName,
    PRE[V]
      (* 1 < 8, so it's safe to write to "cryptoChannelName" + 1. *)
      [| natToW 1 < natToW (length cryptoChannelName) |]%word
      * V "block" =?>8 16
      * [| V "cryptoChannelName" <> 0 |]
      * [| freeable (V "cryptoChannelName") 2 |]
      * array8 cryptoChannelName (V "cryptoChannelName")
      * [| length cryptoChannelName = 8 |]
      * TAThread.invariant openFDs * mallocHeap 0
    POST[_]
      Ex openFDs',
        [| openFDs %<= openFDs' |]
        * V "block" =?>8 16
        * TAThread.invariant openFDs' * mallocHeap 0];;
"cryptoChannelName" + 1 *<-8 69 (* E *);;
Assert
  [Al openFDs, Al cryptoChannelName,
    PRE[V]
      (* 2 < 8, so it's safe to write to "cryptoChannelName" + 2. *)
      [| natToW 2 < natToW (length cryptoChannelName) |]%word
      * V "block" =?>8 16
      * [| V "cryptoChannelName" <> 0 |]
      * [| freeable (V "cryptoChannelName") 2 |]
      * array8 cryptoChannelName (V "cryptoChannelName")
      * [| length cryptoChannelName = 8 |]
      * TAThread.invariant openFDs * mallocHeap 0
    POST[_]
      Ex openFDs',
        [| openFDs %<= openFDs' |]
        * V "block" =?>8 16
        * TAThread.invariant openFDs' * mallocHeap 0];;
"cryptoChannelName" + 2 *<-8 83 (* S *);;
Note [please_unfold_buffer];;
Assert
  [Al openFDs,
    PRE[V]
      (* We don't need to reify cryptoChannelName anymore; simply state
      that it points to an 8-byte buffer. *)
      V "cryptoChannelName" =?>8 8
      * V "block" =?>8 16
      * [| V "cryptoChannelName" <> 0 |]
      * [| freeable (V "cryptoChannelName") 2 |]
      * TAThread.invariant openFDs * mallocHeap 0
    POST[_]
      Ex openFDs',
        [| openFDs %<= openFDs' |]
        * V "block" =?>8 16
        * TAThread.invariant openFDs' * mallocHeap 0];;
(* Connect to the AES service. *)
Assert
  [Al openFDs,
    PRE[V]
      (* Split the "cryptoChannelName" buffer into the part containing
      "AES" (3 bites) and the part that contains garbage (5 bytes).  The
      automation isn't smart enough to realize 3 is less than 8 on its
      own, so we have to remind it. *)
      [| 3 <= 8 |]%nat
      * Arrays8.buffer_splitAt 3 (V "cryptoChannelName") 8
      * [| V "cryptoChannelName" <> 0 |]
      * V "block" =?>8 16
      * [| freeable (V "cryptoChannelName") 2 |]
      * TAThread.invariant openFDs * mallocHeap 0
    POST[_]
      Ex openFDs',
        [| openFDs %<= openFDs' |]
        * V "block" =?>8 16
        * TAThread.invariant openFDs' * mallocHeap 0];;
"cryptoChannel" <-- Call "taConnect"!"connect"("cryptoChannelName", 3)
  [Al openFDs,
    PRE[V, R]
      [| R %in openFDs |]
      * (* Now we can join the buffer back together (we need to before
```

```
              freeing it). *)
            [| 3 <= 8 |]%nat
            * Arrays8.buffer_joinAt 3 (V "cryptoChannelName") 8
            * V "block" =?>8 16
            * [| V "cryptoChannelName" <> 0 |]
            * [| freeable (V "cryptoChannelName") 2 |]
            * TAThread.invariant openFDs * mallocHeap 0
          POST[_]
            Ex openFDs',
              [| openFDs %<= openFDs' |]
              * V "block" =?>8 16
              * TAThread.invariant openFDs' * mallocHeap 0];;
      (* Now that we're connected, free the "AES" buffer so we don't have to
      carry around the invariants. *)
      Note [please_unfold_buffer_joinAt];;
      Call "buffers"!"bfree"("cryptoChannelName", 2)
        [Al openFDs,
          PRE[V]
            [| V "cryptoChannel" %in openFDs |]
            * V "block" =?>8 16
            * TAThread.invariant openFDs * mallocHeap 0
          POST[_]
            Ex openFDs',
              [| openFDs %<= openFDs' |]
              * V "block" =?>8 16
              * TAThread.invariant openFDs' * mallocHeap 0];;
      (* Wait until we're actually connected. *)
      Call "scheduler"!"connected"("cryptoChannel")
        [Al openFDs,
          PRE[V]
            [| V "cryptoChannel" %in openFDs |]
            * V "block" =?>8 16
            * TAThread.invariant openFDs * mallocHeap 0
          POST[_]
            Ex openFDs',
              [| openFDs %<= openFDs' |]
              * V "block" =?>8 16
              * TAThread.invariant openFDs' * mallocHeap 0];;
      (* Send the data block to the AES appliance. *)
      Call "scheduler"!"write"("cryptoChannel", "block", 16)
        [Al openFDs,
          PRE[V]
            [| V "cryptoChannel" %in openFDs |]
            * V "block" =?>8 16
            * TAThread.invariant openFDs * mallocHeap 0
          POST[_]
            Ex openFDs',
              [| openFDs %<= openFDs' |]
              * V "block" =?>8 16
              * TAThread.invariant openFDs' * mallocHeap 0];;
      (* Get the encrypted block back. *)
      Call "scheduler"!"read"("cryptoChannel", "block", 16)
        [Al openFDs,
          PRE[V]
            [| V "cryptoChannel" %in openFDs |]
            * V "block" =?>8 16
            * TAThread.invariant openFDs * mallocHeap 0
          POST[_]
            Ex openFDs',
              [| openFDs %<= openFDs' |]
              * V "block" =?>8 16
              * TAThread.invariant openFDs' * mallocHeap 0];;
      Call "scheduler"!"close"("cryptoChannel")
        [Al openFDs,
          PRE[V]
            V "block" =?>8 16
            * TAThread.invariant openFDs * mallocHeap 0
          POST[_]
            Ex openFDs',
              [| openFDs %<= openFDs' |]
              * V "block" =?>8 16
              * TAThread.invariant openFDs' * mallocHeap 0];;
      Return 0
    end
  }}.

Local Hint Extern 2 =>
  match goal with
    (* TODO: Why can't I match on [?x = _ |- _ < ?x]? *)
    | [ H : Datatypes.length _ = _ |- context[Datatypes.length] ] => rewrite H
```

```
  end.

Local Hint Extern 1 (_ = _) => words.

Local Ltac finish :=
  try match goal with
        | [ |- context[please_unfold_buffer] ] => unfold buffer
      end;
  try match goal with
        | [ |- context[please_unfold_buffer_joinAt] ] =>
          unfold Arrays8.buffer_joinAt
      end;
  TAThread.sep EncryptionTacPackage.hints;
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

EncryptionService

```
Require Import Thread.

Require TAThread.

Definition mainS := SPEC reserving 43
  Al openFDs,
    PREmain[_] TAThread.invariant openFDs * mallocHeap 0.
```

EncryptionServiceImplementation

```
Require Arrays8 Thread.  Import Thread.

Require Encryption EncryptionService EncryptionServiceTacPackage TAThread.
Import TAThread.

(** For reasons explained in the comments in the Bedrock code, this module going
needs an opaque version of multiplication.  (It's declared with [Arguments]
instead of [Opaque] for future compatibility with 8.5.) *)

Definition opaqueMult := mult.
Arguments opaqueMult : simpl never.

Definition m :=
  bimport [[ "encryption"!"aesEncrypt" @ [Encryption.encryptS],
             "malloc"!"malloc" @ [mallocS],
             "scheduler"!"accept" @ [acceptS],
             "scheduler"!"close" @ [closeS],
             "scheduler"!"listen" @ [listenS],
             "scheduler"!"read" @ [readS],
             "scheduler"!"write" @ [writeS] ]]
  bmodule "encryptionService" {{
    bfunctionNoRet "main"("inputSocket", "listenSocket",
                          "scratch") [EncryptionService.mainS]
      (* Create a buffer for the input. *)
      "scratch" <-- Call "malloc"!"malloc"(0, 4)
        [Al openFDs,
           PREmain[_, R]
             R =?> 4
             * TAThread.invariant openFDs * mallocHeap 0];;
      (* Convert the four-word buffer that malloc gives to a sixteen-byte buffer
      suitable for use with AES.  The proof automation for this bit is very
      finicky -- it can't tell that 16 = 4 × 4, so it will fail unless it's very
      obvious exactly what we're doing.  Use the opaque multiplication function
      defined earlier to prevent Coq from expanding [4 * 4] to [16] and thereby
      confusing the automation. *)
      Note [Arrays8.please_materialize_buffer 4];;
      Assert
        [Al openFDs,
           PREmain[V]
             V "scratch" =?>8 opaqueMult 4 4
             * TAThread.invariant openFDs * mallocHeap 0];;
      (* Start listening for connections. *)
      "listenSocket" <-- Call "scheduler"!"listen"(0)
        [Al openFDs,
           PREmain[V, R]
             [| R %in openFDs |]
             * V "scratch" =?>8 16
```

```
          * TAThread.invariant openFDs * mallocHeap 0];;
      (* Loop forever, accepting plaintext and echoing ciphertext back. *)
      [Al openFDs,
         PREmain[V]
           [| V "listenSocket" %in openFDs |]
           * V "scratch" =?>8 16
           * TAThread.invariant openFDs * mallocHeap 0]
      While (0 = 0) {
        (* Take a new connection. *)
        "inputSocket" <-- Call "scheduler"!"accept"("listenSocket")
          [Al openFDs,
             PREmain[V, R]
               [| R %in openFDs |]
               * [| V "listenSocket" %in openFDs |]
               * V "scratch" =?>8 16
               * TAThread.invariant openFDs * mallocHeap 0];;
        (* Read some data out of it. *)
        Call "scheduler"!"read"("inputSocket", "scratch", 16)
          [Al openFDs,
             PREmain[V]
               [| V "inputSocket" %in openFDs |]
               * [| V "listenSocket" %in openFDs |]
               * V "scratch" =?>8 16
               * TAThread.invariant openFDs * mallocHeap 0];;
        Call "encryption"!"aesEncrypt"("scratch")
          [Al openFDs,
             PREmain[V]
               [| V "inputSocket" %in openFDs |]
               * [| V "listenSocket" %in openFDs |]
               * V "scratch" =?>8 16
               * TAThread.invariant openFDs * mallocHeap 0];;
        (* Echo the data back. *)
        Call "scheduler"!"write"("inputSocket", "scratch", 16)
          [Al openFDs,
             PREmain[V]
               [| V "inputSocket" %in openFDs |]
               * [| V "listenSocket" %in openFDs |]
               * V "scratch" =?>8 16
               * TAThread.invariant openFDs * mallocHeap 0];;
        (* Close the connection and exit. *)
        Call "scheduler"!"close"("inputSocket")
          [Al openFDs,
             PREmain[V]
               [| V "listenSocket" %in openFDs |]
               * V "scratch" =?>8 16
               * TAThread.invariant openFDs * mallocHeap 0]
      }
    end
  }}.

Local Hint Extern 2 (_ %in _) =>
  eapply incl_mem;
  [ eassumption | solve [eauto using incl_trans] ].

Ltac finish :=
  TAThread.sep EncryptionServiceTacPackage.hints;
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

EncryptionServiceTacPackage

```
Require Arrays8 PreAutoSep.  Import PreAutoSep.

Definition hints : TacPackage.
  prepare tt Arrays8.materialize_buffer.
Defined.
```

EncryptionTacPackage

```
Require Arrays8 PreAutoSep.  Import PreAutoSep.

Definition hints : TacPackage.
  prepare Arrays8.buffer_split_tagged Arrays8.buffer_join_tagged.
Defined.
```

Extraction

```
Require Import Thumb2_gas.
Require FinalLink.

(** All that's left is to extract the assembly to OCaml for pretty-printing. *)

Definition compiled := moduleS FinalLink.m.
Recursive Extraction compiled.
```

FinalLink

```
Require Import Bedrock.

Require Import Bootstrap Thread.

Require MemoryLayout.
Require
  EncryptionImplementation
  EncryptionServiceImplementation
  LoggingImplementation
  MainImplementation
  MainPrimeImplementation
  TAConnectImplementation
  TAListenImplementation
  TAThread.

Local Notation "f $$ x" := (f x)
  (at level 100, right associativity, only parsing).

Definition m :=
  link EncryptionImplementation.m $$
  link EncryptionServiceImplementation.m $$
  link LoggingImplementation.m $$
  link MainImplementation.m $$
  link MainPrimeImplementation.m $$
  link TAConnectImplementation.m $$
  link TAListenImplementation.m $$
  link Buffers.m
       TAThread.System.m.

Parameter heapSizeLowerBound : (3 <= MemoryLayout.heapSize)%nat.

Parameter requiredMemoryIsReasonable :
  goodSize (MemoryLayout.wordsAvailable * 4)%nat.

Local Ltac use theorem :=
  match goal with
    | [ |- moduleOk (link ?x ?y) ] =>
      cut (moduleOk y);
      [ let H := fresh "H" in intro H; solve [link theorem H] | idtac ]
    | [ |- moduleOk _ ] => exact theorem
  end.

Theorem ok : moduleOk m.
Proof.
  unfold m.
  use EncryptionImplementation.ok.
  use EncryptionServiceImplementation.ok.
  use LoggingImplementation.ok.
  use (MainImplementation.ok heapSizeLowerBound requiredMemoryIsReasonable).
  use MainPrimeImplementation.ok.
  use TAConnectImplementation.ok.
  use TAListenImplementation.ok.
  use Buffers.ok.
  use TAThread.System.ok.
Qed.

Section Safety.
  Hypothesis heapSizeLowerBound : (3 <= MemoryLayout.heapSize)%nat.

  Hypothesis requiredMemoryIsReasonable :
    goodSize (MemoryLayout.wordsAvailable * 4)%nat.

  Let heapSizeUpperBound : goodSize (MemoryLayout.heapSize * 4).
  Proof.  goodSize.  Qed.

  Variable mySettings : settings.
  Variable myProgram : program.
```

```
  Hypothesis labelsInjective :
    forall label1 label2 address,
      Labels mySettings label1 = Some address
      -> Labels mySettings label2 = Some address
      -> label1 = label2.

  Hypothesis labelsAgreeWithBlocks :
    forall label pre block,
      LabelMap.MapsTo label (pre, block) (XCAP.Blocks m)
      -> exists address,
           Labels mySettings label = Some address
           /\ myProgram address = Some block.

  Hypothesis noCodeShadowing :
    forall label pre,
      LabelMap.MapsTo label pre (XCAP.Imports m)
      -> exists address,
           Labels mySettings label = Some address
           /\ myProgram address = None.

  Hypothesis noSysShadowing :
    forall label address,
      Labels mySettings ("sys", label) = Some address
      -> myProgram address = None.

  Variable entryPointAddress : W.
  Hypothesis atStart :
    Labels mySettings ("main", Global "main") = Some entryPointAddress.

  Variable executionState : state.

  Hypothesis memUpperBound :
    forall address,
      (address < MemoryLayout.wordsAvailable * 4)%nat
      -> executionState.(Mem) address <> None.

  Hypothesis memLowerBound :
    forall address,
      $(MemoryLayout.wordsAvailable * 4) <= address
      -> executionState.(Mem) address = None.

  Theorem safe :
    sys_safe mySettings myProgram (entryPointAddress, executionState).
  Proof.  safety ok.  Qed.
End Safety.
```

Logging

```
Require Import Thread.

Require TAThread.

Definition mainS := SPEC reserving 41
  Al openFDs,
    PREmain[_] TAThread.invariant openFDs * mallocHeap 0.
```

LoggingImplementation

```
Require Buffers Thread.  Import Thread.

Require BufferList Logging TAThread TAListen.  Import BufferList.Hints TAThread.

Inductive please_unfold_buffer : Prop := PleaseUnfoldBuffer.
Hint Constructors please_unfold_buffer.

Definition m :=
  bimport [[ "buffers"!"bmalloc" @ [Buffers.bmallocS],
             "malloc"!"malloc" @ [mallocS],
             "scheduler"!"accept" @ [acceptS],
             "scheduler"!"close" @ [closeS],
             "scheduler"!"read" @ [readS],
             "taListen"!"listen" @ [TAListen.listenS] ]]
  bmodule "logging" {{
    bfunctionNoRet "main"("cell", "inputBuffer", "inputSocket", "list",
                     "listenSocket") [Logging.mainS]
      "list" <- 0;;
```

```
      "listenSocket" <-- Call "taListen"!"listen"(1)
        [Al openFDs, Al lst,
         PREmain[V, R]
             [| R %in openFDs |]
             * BufferList.singlyLinkedList lst (V "list")
             * TAThread.invariant openFDs * mallocHeap 0];;
      [Al openFDs, Al lst,
         PREmain[V]
           [| V "listenSocket" %in openFDs |]
           * BufferList.singlyLinkedList lst (V "list")
           * TAThread.invariant openFDs * mallocHeap 0]
      While (0 = 0) {
        "inputSocket" <-- Call "scheduler"!"accept"("listenSocket")
          [Al openFDs, Al lst,
             PREmain[V, R]
               [| R %in openFDs |]
               * [| V "listenSocket" %in openFDs |]
               * BufferList.singlyLinkedList lst (V "list")
               * TAThread.invariant openFDs * mallocHeap 0];;
        "inputBuffer" <-- Call "buffers"!"bmalloc"(4)
          [Al openFDs, Al lst,
             PREmain[V, R]
               [| R <> 0 |] * [| freeable R 4 |]
               * R =?>8 16
               * [| V "inputSocket" %in openFDs |]
               * [| V "listenSocket" %in openFDs |]
               * BufferList.singlyLinkedList lst (V "list")
               * TAThread.invariant openFDs * mallocHeap 0];;
        Call "scheduler"!"read"("inputSocket", "inputBuffer", 16)
          [Al openFDs, Al lst,
             PREmain[V]
               V "inputBuffer" =?>8 16
               * [| V "inputBuffer" <> 0 |]
               * [| V "inputSocket" %in openFDs |]
               * [| V "listenSocket" %in openFDs |]
               * BufferList.singlyLinkedList lst (V "list")
               * TAThread.invariant openFDs * mallocHeap 0];;
        Call "scheduler"!"close"("inputSocket")
          [Al openFDs, Al lst,
             PREmain[V]
               V "inputBuffer" =?>8 16
               * [| V "inputBuffer" <> 0 |]
               * [| V "listenSocket" %in openFDs |]
               * BufferList.singlyLinkedList lst (V "list")
               * TAThread.invariant openFDs * mallocHeap 0];;
        "cell" <-- Call "malloc"!"malloc"(0, 2)
          [Al openFDs, Al lst,
             PREmain[V, R]
               [| R <> 0 |] * [| freeable R 2 |]
               * R =?> 2
               * V "inputBuffer" =?>8 16
               * [| V "inputBuffer" <> 0 |]
               * [| V "listenSocket" %in openFDs |]
               * BufferList.singlyLinkedList lst (V "list")
               * TAThread.invariant openFDs * mallocHeap 0];;
        "cell" *<- "inputBuffer";;
        "cell" + 4 *<- "list";;
        "list" <- "cell";;
        Note [please_unfold_buffer]
      }
    end
  }}.

(* Experimentation with the cost of this hint shows that it has insignificant
effect on the time required for a proof. *)
Local Hint Extern 2 (_ %in _) =>
  eapply incl_mem;
  [ eassumption | solve [eauto using incl_trans] ].

Local Ltac finish :=
  try match goal with
        | [ |- context[please_unfold_buffer] ] => unfold buffer
      end;
  TAThread.sep BufferList.hints;
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

MainImplementation

```
Require Import Bedrock.
Require Bootstrap Malloc Thread.  Import Thread.

Require MainPrime MemoryLayout.

(** Just after entering Bedrock, our memory space is totally unstructured: it’s
just some amorphous blob of memory given to us by the OS.  Our first task is
thus to bring it under Bedrock’s control. *)

Definition m :=
  bimport [[ "malloc"!"init" @ [Malloc.initS],
             "mainPrime"!"main" @ [MainPrime.mainS] ]]
  bmodule "main" {{
    bfunctionNoRet "main"()
        [Bootstrap.bootS MemoryLayout.heapSize 1 (* global *)]
      (* Set the stack pointer. *)
      Sp <- (MemoryLayout.heapSize * 4)%nat;;
      (* Ensure globals are set correctly and initialize the allocator. *)
      Assert
        [PREmain[_]
           MemoryLayout.schedule =?> 1
           * 0 =?> MemoryLayout.heapSize];;
      Call "malloc"!"init"(0, MemoryLayout.heapSize)
        [PREmain[_]
           MemoryLayout.schedule =?> 1
           * mallocHeap 0];;
      (* Pass off control to the application. *)
      Goto "mainPrime"!"main"
    end
  }}.

(** Proving this code correct relies on two critical assumptions -- assumptions
about how well the target platform plays with Bedrock’s memory model. *)

Section Correctness.
  Hypothesis heapSizeLowerBound : (3 <= MemoryLayout.heapSize)%nat.

  Hypothesis requiredMemoryIsReasonable :
    goodSize (MemoryLayout.wordsAvailable * 4)%nat.

  Let heapSizeUpperBound : goodSize (MemoryLayout.heapSize * 4).
  Proof.  Bootstrap.goodSize.  Qed.

  Ltac finish :=
    unfold localsInvariantMain, MemoryLayout.schedule;
    Bootstrap.genesis.

  Lemma ok : moduleOk m.
  Proof.
    vcgen; finish.

    apply H1.  rewrite <- H8.  rewrite -> H7.  rewrite -> H.
    cut (Regs x Sp = Regs x Sp ^+ $4 ^- $ (4)).  intros; solve [auto].
    solve [words].

    rewrite -> H7.  rewrite -> H.
    cut (freeable (Regs x Sp ^+ $4 ^- $ (4)) 50).  intros; solve [auto].
    let H := fresh "H" in
      assert (H : Regs x Sp ^+ $4 ^- $4 = Regs x Sp) by words;
      rewrite -> H.
    solve [assumption].

    apply H1.
    let H := fresh "H" in
      assert (H : Regs b Sp = Regs x Sp ^- $4) by auto;
      rewrite <- H.
    solve [assumption].

    rewrite -> H.  solve [auto].
  Qed.
End Correctness.
```

MainPrime

```
Require Import Thread.

Require MemoryLayout.

Definition mainS := SPEC reserving 49
  PREmain[_]
    MemoryLayout.schedule =?> 1
    * mallocHeap 0.
```

MainPrimeImplementation

```
Require Arrays8 Thread.  Import Thread.

Require EncryptionService Logging MainPrime MemoryLayout TAThread.
Import TAThread.

Definition m :=
  bimport [[ "encryptionService"!"main" @ [EncryptionService.mainS],
             "logging"!"main" @ [Logging.mainS],
             "malloc"!"malloc" @ [mallocS],
             "scheduler"!"accept" @ [acceptS],
             "scheduler"!"close" @ [closeS],
             "scheduler"!"exit" @ [exitS],
             "scheduler"!"init" @ [initS],
             "scheduler"!"listen" @ [listenS],
             "scheduler"!"read" @ [readS],
             "scheduler"!"spawn" @ [spawnS],
             "scheduler"!"write" @ [writeS] ]]
  bmodule "mainPrime" {{
    bfunctionNoRet "main"("socket") [MainPrime.mainS]
      (* Start the scheduler. *)
      Init
        [Al openFDs,
           PREmain[_] sched openFDs * mallocHeap 0];;
      (* Spawn application threads.  Theoretically, we could make one of these a
      goto, but that would constrain the thread main to have the same size stack
      allocation as this function.  Instead, we spawn both threads and then
      permanently yield to the scheduler (by exiting). *)
      Spawn("encryptionService"!"main", 44)
        [Al openFDs,
           PREmain[V]
             TAThread.invariant openFDs * mallocHeap 0];;
      Spawn("logging"!"main", 42)
        [Al openFDs,
           PREmain[V]
             TAThread.invariant openFDs * mallocHeap 0];;
      (* This is /not/ an exit code!  It’s the number of words of stack space
      this function uses. *)
      Exit 50
    end
  }}.

Local Ltac finish :=
  TAThread.sep auto_ext (* empty [TacPackage] *);
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

MemoryLayout

```
(** * Memory layout *)

Require Import Bedrock.

(** ** Available memory *)

(* Ideally, these would be shared with C++ somehow.  However, doing so is
difficult, so for the time being, these are duplicated with values in C++.  If
you update one of these values, you MUST update a corresponding value in
compiler/memory.h! *)
Section Available.
  (* The number of 32-bit words allocated to the Bedrock heap. *)
  Definition heapSize := 1024.
```

```
  (* The number of 32-bit words allocated to the Bedrock stack. *)
  Definition stackSize := 49.

  (* The number of 32-bit words allocated to the Bedrock globals.  In addition
  to the aforementioned duplication, this value better be equal to the number
  of globals defined below. *)
  Definition globalsSize := 1.

  (* The amount of memory initially allocated to the main thread. *)
  Definition wordsAvailable :=
    heapSize
    + stackSize
    + 1 (* return pointer *)
    + globalsSize.
End Available.

(* Treat memory layout constants as opaque to prevent Coq from unfolding them
into giant [nat] terms. *)
Global Opaque heapSize stackSize globalsSize.

(** ** Global variables *)

Section Globals.
  (* This expression is extremely brittle. *)
  Definition schedule : W := (heapSize + (1 + 49)) * 4.
End Globals.
```

TAConnect

```
Require Import Scheduler.

Require TAThread.

(** The Bedrock [connect] specification is unfortunately not quite powerful
enough to prove what I want without effort.  I provide my own instead. *)

Definition connectS := SPEC("address", "size") reserving 29
  Al openFDs,
    PRE[V]
      V "address" =?>8 wordToNat (V "size")
      * TAThread.invariant openFDs * mallocHeap 0
    POST[R]
      V "address" =?>8 wordToNat (V "size")
      * Ex openFDs',
          [| R %in openFDs' |]
          * [| openFDs %<= openFDs' |]
          * TAThread.invariant openFDs' * mallocHeap 0.
```

TAConnectImplementation

```
Require Import Scheduler.

Require TAConnect TAThread.  Import TAThread.

Definition m :=
  bimport [[ "scheduler"!"connect" @ [connectS] ]]
  bmodule "taConnect" {{
    bfunction "connect"("address", "size", "result") [TAConnect.connectS]
      "result" <-- Call "scheduler"!"connect"("address", "size")
        [PRE[_, R] Emp
         POST[R'] [| R' = R |] ];;
      Return "result"
    end
  }}.

Local Ltac finish :=
  TAThread.sep auto_ext (* empty [TacPackage] *);
  try words;
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

TAListenImplementation

```
Require Import Scheduler.

Require TAListen TAThread.  Import TAThread.

Definition m :=
  bimport [[ "scheduler"!"listen" @ [listenS] ]]
  bmodule "taListen" {{
    bfunction "listen"("port", "result") [TAListen.listenS]
      "result" <-- Call "scheduler"!"listen"("port")
        [PRE[_, R] Emp
         POST[R'] [| R' = R |] ];;
      Return "result"
    end
  }}.

Local Ltac finish :=
  TAThread.sep auto_ext (* empty [TacPackage] *);
  try words;
  auto.

Lemma ok : moduleOk m.
Proof.  vcgen; abstract finish.  Qed.
```

TAListen

```
Require Import Scheduler.

Require TAThread.

(** The Bedrock [listen] specification is unfortunately not quite powerful
enough to prove what I want without effort.  I provide my own instead. *)

Definition listenS := SPEC("port") reserving 28
  Al openFDs,
    PRE[_] TAThread.invariant openFDs * mallocHeap 0
    POST[R]
      Ex openFDs',
        [| R %in openFDs' |]
        * [| openFDs %<= openFDs' |]
        * TAThread.invariant openFDs' * mallocHeap 0.
```

TAThread

```
(** Cooperative multithreading system *)

Require Import Thread.  Local Open Scope Sep_scope.

Require MemoryLayout.

(** Instantiate the Bedrock scheduler. *)

Module Private.
  Module SchedulerParameters <: Scheduler.S.
    Definition globalSched := MemoryLayout.schedule.

    Definition globalInv (_ : files) : HProp := Emp.

    Lemma globalInvMonotonic :
      forall x y, x %<= y -> globalInv x ===> globalInv y.
    Proof.  sepLemma.  Qed.
  End SchedulerParameters.
End Private.

Module Export System := Make(Private.SchedulerParameters).

Definition invariant fds : HProp :=
  System.Q''.Sched.sched fds * Private.SchedulerParameters.globalInv fds.

Module Import Hints.
  Hint Extern 4 (himp _ _ _) =>
    eapply Private.SchedulerParameters.globalInvMonotonic; eassumption.
End Hints.

Ltac sep :=
  System.sep ltac:(unfold invariant, Private.SchedulerParameters.globalInv).
```

C, C++, and Assembly

compiler/memory.h

```
// Recall that Bedrock must manage its own memory in order to reason about it
// using separation logic.  One of the primary jobs of the C++ support
// libraries in this project is thus to allocate memory that Bedrock can use.
// In this file, we define constants which describe that hunk of memory.
//
// Ideally, these would be autogenerated from Bedrock code.  However, doing so
// is difficult, so for the time being, these are duplicated with values in
// Bedrock.  If you update one of these values, you MUST update a corresponding
// value in src/MemoryLayout.v!

#ifndef TRUSTANCHOR_COMPILER_MEMORY_H_
#define TRUSTANCHOR_COMPILER_MEMORY_H_

#include <cstddef>

// TODO(bbaren): Move this into a trust_anchor namespace.
namespace bedrock_environment {

// The number of 32-bit words allocated to the Bedrock heap.
constexpr size_t kHeapSlots = 1024;

// The number of 32-bit words allocated to the Bedrock stack.
constexpr size_t kStackSlots = 49;

// The number of 32-bit words allocated to the Bedrock globals.
constexpr size_t kGlobalSlots = 1;

}  // namespace bedrock_environment

#endif  // TRUSTANCHOR_COMPILER_MEMORY_H_
```

runtime/aes/aes.h

```
// Copyright (C) 2014 Inverse Limit

#ifndef AES_H_
#define AES_H_

#define AES_BLOCK_SIZE  16

typedef enum {
        AES_128 = 0,
        AES_192 = 1,
        AES_256 = 2
} aes_algo_t;

typedef struct aes_ctx_s {
        aes_algo_t algo;
        uint32 ks[60];
} aes_ctx_t;

#define AES_128_KEY_SIZE (128 / 8)
#define AES_192_KEY_SIZE (192 / 8)
#define AES_256_KEY_SIZE (256 / 8)

extern uint8 aes_sbox(uint8 U, int inv);

extern void aes_setkey(aes_ctx_t *ctx, aes_algo_t algo, void *key);
extern void aes_ecb_encrypt(aes_ctx_t *ctx, void *data_in, void *data_out);
extern void aes_ecb_decrypt(aes_ctx_t *ctx, void *data_in, void *data_out);

#endif /* AES_H_ */
```

runtime/aes/types.h

```
#ifndef TRUSTANCHOR_RUNTIME_AES_TYPES_H_
#define TRUSTANCHOR_RUNTIME_AES_TYPES_H_

#include <stdint.h>
```

```
typedef uint8_t uint8;
typedef uint32_t uint32;

#endif  // TRUSTANCHOR_RUNTIME_AES_TYPES_H_
```

runtime/aes.h

```
// Utility header file to properly load the AES library.

#ifndef TRUSTANCHOR_RUNTIME_AES_H_
#define TRUSTANCHOR_RUNTIME_AES_H_

extern "C" {
#include "runtime/aes/types.h"
#include "runtime/aes/aes.h"
}

#endif  // TRUSTANCHOR_RUNTIME_AES_H_
```

runtime/boot.cc

```
// Start Bedrock code.
//
// This code bruns as soon as the microcontroller boots.  It's responsible for
// setting up the Bedrock stack, heap, and communication primitives before
// jumping directly into Bedrock code.

#include "runtime/boot.h"

#include <cstdint>

#include <mbed.h>

#include "runtime/io.h"
#include "runtime/io_constants.h"
#include "runtime/singleton.h"

// Define the Bedrock memory region declared in runtime/boot.h.
uint32_t bedrock_heap[
    bedrock_environment::kHeapSlots
    + 1  // for the Bedrock return pointer
    + bedrock_environment::kStackSlots
    + bedrock_environment::kGlobalSlots];

namespace {

// Jumps into Bedrock code, starting at 'main_main'.  Returns the Bedrock
// return value.
uint32_t RunBedrock() {
  // This inline assembly has some rather important side effects, so it needs
  // to be an __asm__ __volatile__ rather than just an __asm__.
  //
  // Every line of assembly except the last ends in \n\t, which produces the
  // prettiest assembly when compiled with gcc -S.
  uint32_t result;
  __asm__ __volatile__(
      // Tell Bedrock how much heap it has.
      "mov r0, %[heap_size]\n\t"
      // Tell Bedrock where to branch after it's done working.
      "adr r1, 0f\n\t"
      // Branch into Bedrock code.
      "b main_main\n\t"
      "0:\n\t"
      // Save the result from Bedrock's main.
      "mov %[result], r2"
      : [result]"=X"(result)
      : [heap_size]"X"(bedrock_environment::kHeapSlots),
        "m"(*bedrock_heap)
      : "r0", "r1", "r2", "r3", "r4", "r5", "r6", "lr", "cc");
  return result;
}

}  // namespace

// Sets up the Bedrock environment and runs Bedrock code.
#define UNUSED __attribute__((__unused__))
int main() {
```

```
  // The Bedrock stack and heap have already been set up (in runtime/boot.h).
  // Set up serial communication.
  trust_anchor::Singleton<trust_anchor::USBSerial>::get()->baud(
      trust_anchor::kMbedBaudrate);
  for (int i = 0; i < 80; i++) {
    SERIAL_DEBUG("-");
  }
  SERIAL_DEBUG("\r\n");
  // Run some Bedrock.
  SERIAL_DEBUG(
      "Bedrock code starting; Bedrock’s memory is at %p\r\n",
      bedrock_heap);
  const uint32_t result UNUSED = RunBedrock();
  SERIAL_DEBUG(
      "Bedrock main returned (should never happen!)\r\n"
      "Return value was %d\r\n",
      result);
  return 0;
}
#undef UNUSED

// Catches hard faults and halts execution for debugging.
extern "C" void HardFault_Handler();
void HardFault_Handler() {
  error("PANIC: hard fault\r\n");
}
```

runtime/boot.h

```
// Symbols essential for running Bedrock code.

#ifndef TRUSTANCHOR_RUNTIME_BOOT_H_
#define TRUSTANCHOR_RUNTIME_BOOT_H_

#include <cstdint>

#include "compiler/memory.h"

// Here’s the Bedrock entry point.  Ideally, this would be extracted from
// Bedrock code somehow, but this is difficult, so for the time being, this
// symbol is duplicated with that defined in the Bedrock main module.  You
// shouldn’t be changing this without also changing that.
extern "C" void main_main();

// Bedrock is very picky about its memory: not only must it be a contiguous
// blob under Bedrock’s total control, but it actually has to be available as a
// linker symbol, unfortunately named ’bedrock_heap’.  (It actually encompasses
// the stack and global variable space as well.)
extern "C" uint32_t bedrock_heap[
    bedrock_environment::kHeapSlots
    + 1  // for the Bedrock return pointer
    + bedrock_environment::kStackSlots
    + bedrock_environment::kGlobalSlots];

#endif  // TRUSTANCHOR_RUNTIME_BOOT_H_
```

runtime/byte_vector.h

```
// Byte vector data type.

#ifndef TRUSTANCHOR_RUNTIME_BYTE_VECTOR_H_
#define TRUSTANCHOR_RUNTIME_BYTE_VECTOR_H_

#include <cstdint>
#include <vector>

using ByteVector = std::vector<uint8_t>;

#endif  // TRUSTANCHOR_RUNTIME_BYTE_VECTOR_H_
```

runtime/connection.cc

```
#include "runtime/connection.h"

#include <algorithm>
#include <cassert>
```

```
#include <deque>
#include <iterator>
#include <type_traits>
#include <unordered_map>

#include "runtime/byte_vector.h"
#include "runtime/io.h"
#include "runtime/lazy_static_ptr.h"

using ::trust_anchor::LazyStaticPtr;
using ::trust_anchor::Session;
using ::trust_anchor::SessionID;
using ::trust_anchor::WireSerializableUInt32Hash;

namespace {

// Session IDs belonging to new sessions.
LazyStaticPtr<std::deque<SessionID>> kNew;

// All sessions, indexed by session ID.
LazyStaticPtr<
  std::unordered_map<SessionID, Session, WireSerializableUInt32Hash>> kAll;

// A template to check if a type T has an iterator that implements
// iterator_tag.  iterator_tag should be one of the five iterator category tags
// described at <http://en.cppreference.com/w/cpp/iterator/iterator_tags>.  The
// result is computed in the value data member of the struct.
//
// Examples:
//
//   HasIterator<int, std::random_access_iterator_tag>::value == false
//
//   HasIterator<std::vector<int>, std::input_iterator_tag>::value == true
//
template<typename T, typename iterator_tag>
using HasIterator = std::is_base_of<
  iterator_tag,
  typename std::iterator_traits<typename T::iterator>::iterator_category>;

// Like Python’s extend function, this function extends one iterable with the
// contents of another, and as in Python’s extend function, the two iterables
// need not have the same type--they need only satisfy the relevant iterable
// constraints.  In particular, the left (target) value must be of a type with
// a RandomAccessIterator, and the right (source) value must be of a type with
// an InputIterator.
//
// Example:
//
//   std::vector<int> x = {1, 2, 3};
//   std::vector<int> y = {4, 5, 6};
//   // extend(x, 3);  // compiler error
//   extend(x, y);
//   // x now has {1, 2, 3, 4, 5, 6}
template<
  typename L,
  typename R,
  typename std::enable_if<HasIterator<L, std::random_access_iterator_tag>::value
                          && HasIterator<R, std::input_iterator_tag>::value>
    ::type* = nullptr>
void extend(L& left, const R& right) {
  left.insert(left.end(), right.begin(), right.end());
}

}  // namespace

namespace trust_anchor {

WireSerializableUInt32 WireSerializableUInt32::Receive(
    BufferAwareSerial* const serial) {
  auto raw = serial->Get(sizeof(value_));
  return WireSerializableUInt32(  (static_cast<uint32_t>(raw[0]) << 24)
                                | (static_cast<uint32_t>(raw[1]) << 16)
                                | (static_cast<uint32_t>(raw[2]) << 8)
                                |  static_cast<uint32_t>(raw[3]));
}

void WireSerializableUInt32::Send(BufferAwareSerial* const serial) const {
  serial->Put({static_cast<uint8_t>(value_ >> 24),
               static_cast<uint8_t>(value_ >> 16),
               static_cast<uint8_t>(value_ >>  8),
```

```
               static_cast<uint8_t>(value_)});
}

Message Message::Receive(BufferAwareSerial* const serial) {
  const auto header = Header::Receive(serial);
  return Message(header.session_id(), serial->Get(header.payload_size()));
}

void Message::Send(BufferAwareSerial* const serial) const {
  MakeHeader().Send(serial);
  serial->Put(payload_);
}

Message::Header Message::Header::Receive(BufferAwareSerial* const serial) {
  return Header(SessionID::Receive(serial),
                WireSerializableUInt32::Receive(serial));
}

void Message::Header::Send(BufferAwareSerial* const serial) const {
  session_id_.Send(serial);
  payload_size_.Send(serial);
}

Message::Header Message::MakeHeader() const {
  return Header(session_id_, payload_.size());
}

bool Session::AcceptWillBlock(BufferAwareSerial* const serial,
                              Session::IDSelector predicate) {
  // Clear out the mbed's serial buffer to try to find new sessions.
  while (!serial->IsEmpty()) {
    // There's at least one byte in the buffer, so a message is either waiting
    // or in the process of being transmitted.  We can afford to block a little
    // bit while receiving it, and it might give us a new session.
    ProcessMessage(serial);
  }
  // The mbed buffer is clear.  Did we get any interesting sessions?
  for (auto new_session_id : *kNew) {
    if (predicate(new_session_id)) {
      // There is an interesting session waiting, so we won't block.
      return false;
    }
  }
  // There are no interesting sessions.
  return true;
}

Session* Session::Accept(BufferAwareSerial* const serial,
                         Session::IDSelector predicate) {
  while (true) {
    // Do we have any interesting sessions?
    for (auto new_session_id = kNew->begin();
         new_session_id != kNew->end();
         new_session_id++) {
      SERIAL_DEBUG("Session::Accept: Looking at session 0x%08x\r\n",
                   new_session_id->value());
      if (predicate(*new_session_id)) {
        // Yes, we do!
        const auto result = Lookup(*new_session_id);
        assert(result);  // We found the session.
        // Clear the session ID from the new queue.
        kNew->erase(new_session_id);
        // All done.
        return result;
      }
    }
    // There aren't any interesting sessions in the new queue yet.  Wait until
    // a new session comes in.
    ProcessMessage(serial);
  }
}

void Session::Close(const SessionID& session_id) {
  for (auto session = kAll->begin();
       session != kAll->end();
       session++) {
    if (session->first == session_id) {
      kAll->erase(session);
      return;
    }
```

```
  }
  assert(false);
}

Session* Session::Lookup(const SessionID& session_id) {
  const auto session = kAll->find(session_id);
  if (session == kAll->end()) {
    return nullptr;
  }
  return &(session->second);
}

ByteVector Session::Read(const ByteVector::size_type& n_bytes) {
  // Prepopulate the read buffer.
  const auto bytes_to_read = CanRead(n_bytes);
  ByteVector result;
  for (ByteVector::size_type i = 0; i < bytes_to_read; i++) {
    result.emplace_back(read_buffer_.front());
    read_buffer_.pop_front();
  }
  return result;
}

ByteVector::size_type Session::CanRead(
    const ByteVector::size_type& n_bytes) const {
  while (read_buffer_.size() < n_bytes && !serial_->IsEmpty()) {
    // We don't yet have |n_bytes| available to return, but there's at least
    // one byte waiting in the serial buffer.  This means that at least the
    // start of a message has arrived at the mbed, and it's likely that the
    // rest will shortly follow.  Thus, calling |ProcessMessage| here is
    // unlikely to trigger blocking.
    //
    // TODO(bbaren): This is a bit hackish; make it more elegant.  Ideally,
    // determine how much data is sitting in the serial buffer and ensure that
    // a message is actually waiting.
    ProcessMessage(serial_);
  }
  return std::min(n_bytes, read_buffer_.size());
}

ByteVector::size_type Session::Write(const ByteVector& bytes) {
  if (CanWrite(bytes.size()) == 0) {
    // The hardware serial buffer is full.
    return 0;
  }
  // At this point, the hardware serial buffer is currently empty.  We're
  // probably going to block a bit doing our write, since the serial is
  // probably clocked slower than the CPU, but it's dangerously complicated to
  // try to make this entirely nonblocking.
  //
  // TODO(bbaren): Actually make this nonblocking, possibly by creating a
  // shadow buffer in front of the hardware.
  Message(session_id_, bytes).Send(serial_);
  return bytes.size();
}

ByteVector::size_type Session::CanWrite(
    const ByteVector::size_type& n_bytes) const {
  // Assume we always can write--i.e., the serial chip has an unlimited buffer.
  // This isn't always the case, but it's a useful abstraction for this
  // particular app.
  return n_bytes;
}

void Session::ProcessMessage(BufferAwareSerial* const serial) {
  const auto message = Message::Receive(serial);
  auto session = kAll->find(message.session_id());
  if (session == kAll->end()) {
    // We've never seen this session before, so create it.
    auto emplace_result = kAll->emplace(message.session_id(),
                                        Session(serial, message.session_id()));
    assert(emplace_result.second);  // The insertion should have taken place.
    session = emplace_result.first;
    kNew->emplace_back(message.session_id());
  }
  // Assign the message payload to the session.
  extend(session->second.read_buffer_, message.payload());
}

}  // namespace trust_anchor
```

runtime/connection.h

```
// The Bedrock system calls are tailored to a session-based protocol like TCP.
// However, the mbed only has a serial line.  To bridge this gap, this module
// implements (the server side of) a simple session-based protocol that can
// execute over a serial line.
//
// WARNING: THIS PROTOCOL HAS SECURITY PROBLEMS.  It got written as a prototype
// during somebody's thesis project, and it needs to be replaced with something
// better before entering production.
//
// At the lowest level, the protocol consists of _messages_.  A message
// contains three parts: a session identifier, a size, and a data set.  The
// first two fields are called the header, and the data set is also called the
// payload.
//
//   31                              0
//   +------------------------------+
//   |          session ID          |
//   +------------------------------+
//   |             size             |
//   +------------------------------+
//   |                              |
//   |             data             |
//   |                              |
//   +------------------------------+
//
// The session identifier and size are both big-endian 32-bit unsigned
// integers.
//
//   - The session identifier is used by the endpoints to group messages into
//     sessions.  It is the client's responsibility to never reuse session
//     identifiers.  (This restriction is obviously unrealistic in a production
//     system, but it simplifies the code substantially and seems appropriate
//     for a research proof-of-concept.)
//
//   - The size field specifies the size of the data field (_not_ of the entire
//     message) in bytes.
//
// The data are simply a free-form byte stream.
//
// Here's an example session between a server and two clients.
//
//   C1: (0x41414141, 12, 'Hello there.')
//   S:  (0x41414141, 15, 'Hello yourself.')
//   C2: (0x80808080, 21, 'I am a second client.')
//   S:  (0x41414141, 16, 'You still there?')
//   C1: (0x41414141, 4, 'Yes.')
//
// The parties should simply ignore any message which does not conform to the
// protocol.

#ifndef TRUSTANCHOR_RUNTIME_CONNECTION_H_
#define TRUSTANCHOR_RUNTIME_CONNECTION_H_

#include <cstdint>
#include <deque>
#include <functional>

#include "runtime/byte_vector.h"
#include "runtime/macros.h"
#include "runtime/io.h"

namespace trust_anchor {

// Abstract base class to describe things which can be wire-serialized.
class WireSerializable {
 public:
  virtual ~WireSerializable() {}

  // Subclasses should define a static member function which receives.
  // Recommended signature: static T Receive(BufferAwareSerial* serial)

  // Serializes the object and sends it along the wire.
  virtual void Send(BufferAwareSerial* serial) const = 0;
};

// Abstract class which provides a default serialization for uint32_ts--in
// particular, as four bytes in big-endian order.  This may seem a bit
```

```
// unnecessary, but it lets us eliminate duplication among classes representing
// header fields.
class WireSerializableUInt32 : public WireSerializable {
 public:
  // This constructor is an implicit constructor and therefore violates the
  // Google style guide.  However, it seems appropriate in this case to be able
  // to say something like
  //
  //   SerializableUInt32 x = 42;  // implicit construction
  //
  WireSerializableUInt32(const uint32_t value) noexcept : value_(value) {}

  ~WireSerializableUInt32() override {}

  uint32_t value() const noexcept { return value_; }
  void set_value(const uint32_t value) noexcept { value_ = value; }

  static WireSerializableUInt32 Receive(BufferAwareSerial* serial);

  void Send(BufferAwareSerial* serial) const;

  bool operator==(const WireSerializableUInt32& other) const noexcept {
    return value_ == other.value_;
  }

 protected:
  uint32_t value_;
};

// Hash routine for WireSerializableUInt32s, allowing that type to be used in
// unordered containers.
class WireSerializableUInt32Hash {
 public:
  size_t operator()(const WireSerializableUInt32& key) const noexcept {
    // If size_t is 32 bits wide, we're hashing a 32-bit value into a 32-bit
    // value and the hash function can be the identity function!  Check to
    // ensure that size_t is actually 32 bits wide, and go for it.
    static_assert(sizeof(size_t) == sizeof(uint32_t),
                  "size_t and uint32_t not sized identically");
    return key.value();
  }
};

// Forward declaration for friend specification in class SessionID
class Message;

// 32-bit value which identifies a session on the wire.  This needs to get
// passed with every wire message so we can keep track of the session.
class SessionID : public WireSerializableUInt32 {
 public:
  explicit SessionID(const uint32_t value) noexcept
      : WireSerializableUInt32(value) {}

 private:
  // Downcast constructor: converts a SerializableUInt32 to a SessionID.  This
  // is a potentially dangerous operation if the SerializableUInt32 in question
  // does not actually represent a SessionID, and it's therefore reserved for
  // internal use.
  SessionID(const WireSerializableUInt32& value) noexcept
      : WireSerializableUInt32(value) {}

  // Message will have to construct SessionIDs by reading them off the wire, so
  // it needs to be able to use the SessionID(WireSerializableUInt32)
  // constructor.
  friend class Message;
};

// Wire message as detailed in the file comments.
class Message : public WireSerializable {
 public:
  Message(const SessionID& session_id)
      : payload_({}), session_id_(session_id) {}

  Message(const SessionID& session_id, const ByteVector& payload)
      : payload_(payload), session_id_(session_id) {}

  const ByteVector& payload() const noexcept { return payload_; }
  const SessionID& session_id() const noexcept { return session_id_; }

  static Message Receive(BufferAwareSerial* serial);
```

```
  void Send(BufferAwareSerial* serial) const;

 private:
  // Wire message header as detailed in the file comments.
  class Header {
   public:
    Header(const SessionID& session_id,
           const WireSerializableUInt32& payload_size)
        : payload_size_(payload_size), session_id_(session_id) {}

    ByteVector::size_type payload_size() const noexcept {
      return payload_size_.value();
    }
    const SessionID& session_id() const noexcept { return session_id_; }

    static Header Receive(BufferAwareSerial* serial);

    void Send(BufferAwareSerial* serial) const;

   private:
    WireSerializableUInt32 payload_size_;
    SessionID session_id_;
  };

  // Generates a Header for this Message.
  Header MakeHeader() const;

  ByteVector payload_;
  SessionID session_id_;
};

// Handle onto a wire session.
//
// A |Session| is either new or active.  Any time the server receives a message
// with an unrecognized session ID, it creates a new session; once a server API
// user calls |Session::accept|, the session becomes active.  (Sessions become
// active in the order they appear.)  Only active sessions can send messages.
class Session {
 public:
  using IDSelector = std::function<bool(const SessionID&)>;

  // Move constructor
  Session(Session&& other) = default;

  const SessionID& session_id() const noexcept { return session_id_; }

  // Determines if |Accept| will block when called with the same arguments.
  static bool AcceptWillBlock(BufferAwareSerial* serial, IDSelector predicate);

  // Blocks until a session appears whose ID satisfies the |predicate|.
  // Returns that session.
  static Session* Accept(BufferAwareSerial* serial, IDSelector predicate);

  static void Close(const SessionID& session_id);

  // Looks up a session, returning |nullptr| if none exists.
  static Session* Lookup(const SessionID& session_id);

  // Performs a non-blocking read of up to |n_bytes| bytes from this session.
  ByteVector Read(const ByteVector::size_type& n_bytes);

  // Determines how many bytes a call to |Read| will return.
  ByteVector::size_type CanRead(const ByteVector::size_type& n_bytes) const;

  // Performs a non-blocking write of up to |bytes.size()| bytes from this
  // session.  Returns the number of bytes actually written.
  ByteVector::size_type Write(const ByteVector& bytes);

  // Determines how many bytes a call to |Write| will write.
  ByteVector::size_type CanWrite(const ByteVector::size_type& n_bytes) const;

 private:
  explicit Session(BufferAwareSerial* serial, const SessionID& session_id)
      : read_buffer_(std::deque<uint8_t>()), serial_(serial),
        session_id_(session_id) {}

  // Blocks until a message is ready to read, and then reads it.  Data are
  // entered into the read buffer for the relevant session, which is created if
  // necessary.
```

```
  static void ProcessMessage(BufferAwareSerial* serial);

  // Data which has been read off the serial line for this Session but has not
  // yet been consumed.  This needs to be a deque so that certain iterator
  // operations can occur (see |extend| in runtime/connection.cc).
  mutable std::deque<uint8_t> read_buffer_;

  BufferAwareSerial* const serial_;

  SessionID session_id_;

  DISALLOW_COPY_AND_ASSIGN(Session);
};

}  // namespace trust_anchor


#endif  // TRUSTANCHOR_RUNTIME_CONNECTION_H_
```

runtime/handle.cc

```
#include "runtime/handle.h"

#include <cassert>
#include <cstdint>
#include <memory>

#include "runtime/aes.h"
#include "runtime/connection.h"
#include "runtime/io.h"

namespace trust_anchor {

uint32_t Handle::EncodeForBedrock() const {
  if (IsListener()) {
      switch (dynamic_cast<const Listener*>(this)->channel()) {
        case Listener::Channel::kEncryption:
          return 0;
        case Listener::Channel::kLogging:
          return 1;
        default:
          assert(false);
      }
  } else if (IsAESHandle()) {
    return 2;
  } else {
    assert(IsSessionHandle());
    return dynamic_cast<const SessionHandle*>(this)->session_id().value() + 3;
  }
}

std::shared_ptr<Handle> Handle::DecodeFromBedrock(const uint32_t encoded) {
  using HandlePtr = std::shared_ptr<Handle>;
  switch (encoded) {
    case 0:
      return HandlePtr(new Listener(Listener::Channel::kEncryption));
    case 1:
      return HandlePtr(new Listener(Listener::Channel::kLogging));
    case 2:
      return HandlePtr(new AESHandle());
    default:
      return HandlePtr(new SessionHandle(SessionID(encoded - 3)));
  }
}

bool Listener::IsReady(const Direction& direction) const {
  switch (direction) {
    case Direction::kRead:
      return !Session::AcceptWillBlock(
          Singleton<USBSerial>::get(),
          [this](const SessionID& id) {
            return id.value() % 2 == channel_;
          });
    case Direction::kWrite:
      // We can never write to a listener.
      return false;
    default:
      assert(false);
```

```
  }
}

Session* SessionHandle::GetSession() const {
  return Session::Lookup(session_id_);
}

bool SessionHandle::IsReady(const Direction& direction) const {
  const auto* session = GetSession();
  assert(session);
  switch (direction) {
    case Direction::kRead:
      return session->CanRead(1);
    case Direction::kWrite:
      return session->CanWrite(1);
    default:
      assert(false);
  }
}

bool AESHandle::IsReady(const Direction& direction) const {
  // We can always read from or write to the AES appliance.  Whether that read
  // or write makes any sense is up to the application to decide.
  return true;
}

void AESHandle::Encrypt(const Block& key, const Block& plaintext) noexcept {
  // The AES code we got is not const-correct, so better create some mutable
  // versions of |key| and |data|.
  auto m_key = key;
  auto m_plaintext = plaintext;
  // Do the AES operation.
  SERIAL_DEBUG("Encrypting!\r\nKey:    %s\r\nPlain:  %s\r\n",
               ShowHex(key.data(), key.size()).data(),
               ShowHex(plaintext.data(), plaintext.size()).data());
  aes_ctx_t context;
  aes_setkey(&context, AES_128, m_key.data());
  aes_ecb_encrypt(&context, m_plaintext.data(), last_result_.data());
  SERIAL_DEBUG("Cipher: %s\r\n",
               ShowHex(last_result_.data(), last_result_.size()).data());
}

}  // namespace trust_anchor
```

runtime/handle.h

```
// Resource handles for the mbed.
//
// Bedrock's system calls need to be able to refer to handles.  On x86, file
// descriptors fill this role; on the mbed, I have to implement them myself.

#ifndef TRUSTANCHOR_RUNTIME_HANDLE_H
#define TRUSTANCHOR_RUNTIME_HANDLE_H

#include <cstdint>
#include <memory>

#include "runtime/aes.h"
#include "runtime/connection.h"

namespace trust_anchor {

// Resource handle as described in file comments.  Bedrock needs to be able to
// refer to these, so I provide a mechanism to serialize and deserialize them
// using |uint32_t|s.
class Handle {
 public:
  enum class Direction {
    kRead,
    kWrite,
  };

  virtual ~Handle() {}

  virtual bool IsReady(const Direction& direction) const = 0;

  uint32_t EncodeForBedrock() const;
```

```
  static std::shared_ptr<Handle> DecodeFromBedrock(const uint32_t encoded);

  // Functions to help with downcasting.
  virtual bool IsListener() const { return false; }
  virtual bool IsSessionHandle() const { return false; }
  virtual bool IsAESHandle() const { return false; }
};

// Resource handle which represents a socket in the listening state.
class Listener : public Handle {
 public:
  // What the socket is listening for.
  enum Channel : uint32_t {
    kEncryption = 0,  // data to be encrypted
    kLogging = 1,  // data to be logged
  };

  Listener(const Channel& channel) : channel_(channel) {}

  const Channel& channel() const noexcept { return channel_; }

  virtual bool IsReady(const Direction& direction) const override;

  virtual bool IsListener() const override { return true; }

 protected:
  Channel channel_;
};

// Resource handle which represents a socket connected to a session.
class SessionHandle : public Handle {
 public:
  SessionHandle(const SessionID& session_id) : session_id_(session_id) {}

  const SessionID& session_id() const noexcept { return session_id_; }

  // Looks up the session associated with this resource.  Returns |nullptr| if
  // no such session exists (i.e., if this resource handle is invalid).
  Session* GetSession() const;

  virtual bool IsReady(const Direction& direction) const override;

  virtual bool IsSessionHandle() const override { return true; }

 protected:
  SessionID session_id_;
};

// Resource handle which represents the AES appliance.
class AESHandle : public Handle {
 public:
  using Block = std::array<uint8_t, AES_128_KEY_SIZE>;

  AESHandle() {}

  const Block& last_result() const noexcept { return last_result_; }

  virtual bool IsReady(const Direction& direction) const override;

  void Encrypt(const Block& key, const Block& plaintext) noexcept;

  void Decrypt(const Block& key, const Block& ciphertext) noexcept;

  virtual bool IsAESHandle() const override { return true; }

 protected:
  Block last_result_;
};

}  // namespace trust_anchor

#endif
```

runtime/io.cc

```
// Serial I/O using the mbed.

#include "runtime/io.h"

#include <cassert>
#include <cstdint>
#include <iomanip>
#include <ios>
#include <sstream>
#include <stdexcept>
#include <string>

#include "runtime/byte_vector.h"

namespace trust_anchor {

ByteVector BufferAwareSerial::Get(const ByteVector::size_type& size) {
  uint8_t raw_bytes[size];
  const auto n_bytes_read = this->read(raw_bytes, size);
  assert(0 <= n_bytes_read);
  assert(static_cast<size_t>(n_bytes_read) == size);
  return ByteVector(raw_bytes, raw_bytes + n_bytes_read);
}

bool BufferAwareSerial::IsEmpty() {
  return !this->readable();
}

void BufferAwareSerial::Put(const ByteVector& data) {
  const auto n_bytes_written = this->write(data.data(), data.size());
  assert(0 <= n_bytes_written);
  assert(static_cast<size_t>(n_bytes_written) == data.size());
}

USBSerial::USBSerial() : BufferAwareSerial(USBTX, USBRX) {
  this->attach(this, &USBSerial::HandleReceiveInterrupt);
}

ByteVector USBSerial::Get(const ByteVector::size_type& size) {
  ByteVector result;
  while (result.size() < size) {
    uint32_t queued_byte;
    try {
      queued_byte = buffer_.Dequeue();
    } catch (std::domain_error) {
      continue;
    }
    result.push_back(static_cast<uint8_t>(queued_byte));
  }
  return result;
}

bool USBSerial::IsEmpty() {
  return buffer_.IsEmpty();
}

void USBSerial::HandleReceiveInterrupt() {
  while (this->readable()) {
    const uint8_t raw_byte = this->getc();
    this->putc('.');
    buffer_.Enqueue(static_cast<uint32_t>(raw_byte));
  }
}

std::string ShowHex(const uint8_t* data, const size_t length) {
  std::ostringstream result;
  for (std::size_t i = 0; i < length; i++) {
    result << std::setw(2) << std::setfill('0') << std::hex
           << static_cast<int>(data[i]);
  }
  return result.str();
}

}  // namespace trust_anchor
```

runtime/io_constants.h

```
// Constants for input and output useful in places other than just the program
// that runs on the mbed.

#ifndef TRUSTANCHOR_RUNTIME_IO_CONSTANTS_H_
#define TRUSTANCHOR_RUNTIME_IO_CONSTANTS_H_

namespace trust_anchor {

constexpr int kMbedBaudrate = 230400;

}  // namespace

#endif  // TRUSTANCHOR_RUNTIME_IO_CONSTANTS_H_
```

runtime/io.h

```
// Serial I/O using the mbed.

#ifndef TRUSTANCHOR_RUNTIME_IO_H_
#define TRUSTANCHOR_RUNTIME_IO_H_

#include <cstddef>
#include <cstdint>
#include <string>

#include <mbed.h>

#include "runtime/byte_vector.h"
#include "runtime/queue.h"
#include "runtime/singleton.h"

namespace trust_anchor {

// A serial line which can read and write STL vectors in addition to character
// buffers.
class BufferAwareSerial : public mbed::Serial {
 public:
  BufferAwareSerial(const PinName& tx, const PinName& rx)
      : mbed::Serial(tx, rx) {}

  ~BufferAwareSerial() override {}

  virtual ByteVector Get(const ByteVector::size_type& size);

  virtual bool IsEmpty();

  void Put(const ByteVector& data);
};

// The serial line connected to the USB port.
class USBSerial : public BufferAwareSerial {
 public:
  USBSerial();
  ~USBSerial() override {}

  virtual ByteVector Get(const ByteVector::size_type& size) override;

  virtual bool IsEmpty() override;

 private:
  void HandleReceiveInterrupt();

  MichaelScottQueue buffer_;
};

// Prints a debugging string to the serial line.
#ifdef PRINTF_DEBUGGING
# define SERIAL_DEBUG(...)                                              \
  trust_anchor::Singleton<trust_anchor::USBSerial>::get()->printf(__VA_ARGS__)
#else
# define SERIAL_DEBUG(...) do {} while (false)
#endif
```

```
// A useful debugging routine to format a buffer.
std::string ShowHex(const uint8_t* data, const std::size_t length);

}  // namespace trust_anchor

#endif  // TRUSTANCHOR_RUNTIME_IO_H_
```

runtime/lazy_static_ptr.h

```
// Copyright (C) 2013 Google, Inc.
// Copyright (C) 2015 the Massachusetts Institute of Technology
//
// Based on lazy_static_ptr, which is licensed under the Apache License v2.0.

// Google's LazyStaticPtr implementation is a bit heavyweight for the mbed, so
// we've modified it to work with our Trust Anchor code base.  It is now most
// decidedly not thread-safe.  On the other hand, though, this is running on a
// microcontroller, so worrying about thread safety seems to be putting the
// cart before the horse.

#ifndef TRUSTANCHOR_RUNTIME_LAZY_STATIC_PTR_H_
#define TRUSTANCHOR_RUNTIME_LAZY_STATIC_PTR_H_

#include "runtime/macros.h"

namespace trust_anchor {

// Lazily allocates an object of the specified type.
template <typename T>
class LazyStaticPtr {
 public:
  LazyStaticPtr() : ptr_(nullptr) {}

  T &operator*() const { return *get(); }
  T* operator->() const { return get(); }
  T* get() const {
    if (ptr_ == nullptr) {
      ptr_ = new T();
    }
    return ptr_;
  }

 private:
  mutable T* ptr_;

  DISALLOW_COPY_AND_ASSIGN(LazyStaticPtr);
};

}  // namespace trust_anchor

#endif  // TRUSTANCHOR_RUNTIME_LAZY_STATIC_PTR_H_
```

runtime/macros.h

```
// Copyright (C) 2013 Google, Inc.
// Copyright (C) 2015 the Massachusetts Institute of Technology
//
// These macros were originally licensed under the Apache License v2.0.

// Various macros useful for high-quality C++.

#ifndef TRUSTANCHOR_RUNTIME_MACROS_H_
#define TRUSTANCHOR_RUNTIME_MACROS_H_

// A macro to disallow the copy constructor and operator= functions
// This should be used in the private: declarations for a class
//
// For disallowing only assign or copy, write the code directly, but declare
// the intent in a comment, for example:
// void operator=(const TypeName&) = delete;  // DISALLOW_ASSIGN
// Note, that most uses of DISALLOW_ASSIGN and DISALLOW_COPY are broken
// semantically, one should either use disallow both or neither. Try to
// avoid these in new code.
#if (201100L <= __cplusplus)
# define DISALLOW_COPY_AND_ASSIGN(TypeName) \
   TypeName(const TypeName&) = delete;      \
```

```
   void operator=(const TypeName&) = delete
#else
# define DISALLOW_COPY_AND_ASSIGN(TypeName) \
   TypeName(const TypeName&);               \
   void operator=(const TypeName&)
#endif

#endif  // TRUSTANCHOR_RUNTIME_MACROS_H_
```

runtime/queue.cc

```
// TODO(bbaren): _Very_ occasionally, this implementation causes the
// microcontroller to crash.  (This isn't entirely surprising, since this
// implementation doesn't use hazard pointers.)  Figure out why the crash
// occurs and fix it.

#include "runtime/queue.h"

#include <cassert>
#include <cstdint>
#include <memory>
#include <stdexcept>

#include <mbed.h>

namespace trust_anchor {

MichaelScottQueue::MichaelScottQueue() {
  std::shared_ptr<Node> node(new Node);
  node->next.pointer = nullptr;
  head_.pointer = node;
  tail_.pointer = node;
}

using Queue = MichaelScottQueue;

uint32_t Queue::Dequeue() {
  uint32_t result;
  Pointer head;
  while (true) {
    head = head_;
    Pointer tail = tail_;
    Pointer next = head.pointer->next;
    if (head == head_) {
      if (head.pointer == tail.pointer) {
        if (next.pointer == nullptr) {
          throw std::domain_error("queue is empty");
        }
        Pointer::CAS(&tail_, tail, Pointer{next.pointer, tail.count + 1});
      } else {
        result = next.pointer->value;
        if (Pointer::CAS(&head_, head, Pointer{next.pointer, head.count + 1})) {
          break;
        }
      }
    }
  }
  return result;
}

void Queue::Enqueue(const uint32_t value) {
  std::shared_ptr<Node> node(new Node);
  node->value = value;
  node->next.pointer = nullptr;
  Pointer tail;
  while (true) {
    tail = tail_;
    Pointer next = tail.pointer->next;
    if (tail == tail_) {
      if (next.pointer == nullptr) {
        if (Pointer::CAS(&tail.pointer->next,
                         next, Pointer{node, next.count + 1})) {
          break;
        }
      } else {
        Pointer::CAS(&tail_, tail, Pointer{next.pointer, tail.count + 1});
      }
    }
```

```
  }
  Pointer::CAS(&tail_, tail, Pointer{node, tail.count + 1});
}

bool Queue::IsEmpty() {
  return head_.pointer == tail_.pointer;
}

bool Queue::Pointer::CAS(Queue::Pointer* const pointer,
         const Queue::Pointer& old_value, const Queue::Pointer& new_value) {
  bool result;
  // This is privileged code, so we can dispense with ldrex and strex and
  // simply disable interrupts during the operation.
  __disable_irq();
  if (*pointer == old_value) {
    *pointer = new_value;
    // Duplicate the __enable_irq call between branches of the if to keep
    // interrupts turned off for as little time as possible.
    __enable_irq();
    result = true;
  } else {
    __enable_irq();
    result = false;
  }
  return result;
}

}  // namespace trust_anchor
```

runtime/queue.h

```
// Michael & Scott lock-free queue
//
// This is a direct implementation of the queue from Maged M. Michael and
// Michael L. Scott, 'Simple, Fast, and Practical Non-Blocking and Blocking
// Concurrent Queue Algorithms', PODC '96, pp 267-275,
// doi:10.1145/248052.248106.

#ifndef TRUSTANCHOR_RUNTIME_QUEUE_H_
#define TRUSTANCHOR_RUNTIME_QUEUE_H_

#include <cstdint>
#include <memory>

#include "runtime/macros.h"

namespace trust_anchor {

class MichaelScottQueue {
 public:
  MichaelScottQueue();

  uint32_t Dequeue();

  void Enqueue(const uint32_t value);

  bool IsEmpty();

 private:
  struct Node;

  struct Pointer {
    std::shared_ptr<Node> pointer;
    unsigned int count;

    static bool CAS(Pointer* const pointer,
                    const Pointer& old_value, const Pointer& new_value);

    bool operator==(const Pointer& other) const noexcept {
      return pointer == other.pointer && count == other.count;
    }
  };

  struct Node {
    uint32_t value;
    Pointer next;
  };
```

```
  Pointer head_;
  Pointer tail_;

  DISALLOW_COPY_AND_ASSIGN(MichaelScottQueue);
};

}  // namespace trust_anchor

#endif  // TRUSTANCHOR_RUNTIME_AES_H_
```

runtime/reservation.cc

```
#include "runtime/reservation.h"

#include <cassert>
#include <map>
#include <memory>

#include "runtime/io.h"
#include "runtime/handle.h"
#include "runtime/lazy_static_ptr.h"

namespace {
using ::trust_anchor::LazyStaticPtr;
using ::trust_anchor::Reservation;

// All valid reservations.
LazyStaticPtr<std::map<Reservation::ID, Reservation>> kValid;

}  // namespace

namespace trust_anchor {

Reservation::ID Reservation::Make(const std::shared_ptr<Handle> handle,
                                  const Handle::Direction& direction) {
  const auto result = kValid->emplace(GenerateID(),
                                      Reservation(handle, direction));
  assert(result.second);  // assert insertion actually took place
  return result.first->first;  // return copy of inserted ID
}

std::unique_ptr<Reservation::ID> Reservation::FindSatisfiable() {
  for (auto reservation_iter = kValid->begin();
       reservation_iter != kValid->end();
       reservation_iter++) {
    auto& reservation = reservation_iter->second;
    if (reservation.handle_->IsReady(reservation.direction_)) {
      // Erase |reservation|, but grab its ID first so we can return it.
      // (Erasing it will invalidate its iterator, so we can't do so
      // afterward.)
      auto result = std::unique_ptr<Reservation::ID>(
          new Reservation::ID(reservation_iter->first));
      kValid->erase(reservation_iter);
      return result;
    }
  }
  return nullptr;
}

Reservation::ID Reservation::GenerateID() {
  // TODO(bbaren): Come up with a better way to do this, preferably one that
  // doesn't have overflow problems.
  static ID max_id;
  return max_id++;
}

}  // namespace trust_anchor
```

runtime/reservation.h

```
// Reservation class for implementing epoll-style I/O control.

#ifndef TRUSTANCHOR_RUNTIME_RESERVATION_H_
#define TRUSTANCHOR_RUNTIME_RESERVATION_H_

#include <cstdint>
#include <memory>
```

```
#include "runtime/handle.h"

namespace trust_anchor {

class Reservation {
 public:
  using ID = uint32_t;

  static ID Make(std::shared_ptr<Handle> handle,
                 const Handle::Direction& direction);

  // Finds a reservation which can be satisfied and invalidates it.  Returns
  // its ID.  Returns |nullptr| if no reservation can be satisfied.
  static std::unique_ptr<ID> FindSatisfiable();

 private:
  explicit Reservation(std::shared_ptr<Handle> handle,
                       const Handle::Direction& direction)
      : handle_(handle), direction_(direction) {}

  static ID GenerateID();

  std::shared_ptr<Handle> handle_;

  Handle::Direction direction_;
};

}  // namespace trust_anchor

#endif  // TRUSTANCHOR_RUNTIME_RESERVATION_H_
```

runtime/singleton.h

```
// Copyright (C) 2013 Google, Inc.
// Copyright (C) 2015 the Massachusetts Institute of Technology
//
// Based on lazy_static_ptr, which is licensed under the Apache License v2.0.

// The mbed libraries don't provide any kind of singleton object support, so
// we've written our own.  It's heavily based off of Google's LazyStaticPtr
// implementation; unlike it, however, it is most decidedly not thread-safe.
// On the other hand, though, this is running on a microcontroller, so worrying
// about thread safety seems to be putting the cart before the horse.

#ifndef TRUSTANCHOR_RUNTIME_SINGLETON_H_
#define TRUSTANCHOR_RUNTIME_SINGLETON_H_

#include "runtime/macros.h"

namespace trust_anchor {

// Singleton object wrapper.  The wrapped object must have a nullary
// constructor, and its destructor will never get called.
//
// Example usage:
//
//   Serial* const serial = Singleton<Serial>::get();
//   serial->puts("foo");
//
template<typename T>
class Singleton {
 public:
  static T* get() {
    // This if statement looks a bit sketchy--when did ptr_ ever get
    // initialized to nullptr?  See just below the class definition.
    if (!ptr_) {
      ptr_ = new T();
    }
    return ptr_;
  }

 private:
  static T* ptr_;

  DISALLOW_COPY_AND_ASSIGN(Singleton);
};
```

```
// Do a static initialization to make sure Singleton<T>::ptr_ = nullptr when
// control hits main.  C++ disallows setting ptr_ to nullptr in the class
// definition, but doing it here is just fine.
template<typename T>
T* Singleton<T>::ptr_ = nullptr;

}  // namespace trust_anchor

#endif  // TRUSTANCHOR_RUNTIME_SINGLETON_H_
```

runtime/syscall_entry.h

```
// Bedrock relies on untrusted system calls.  Numbers for those system calls
// are defined here (as preprocessor macros, so they can be used in both C++
// and assembly).
//
// These system calls use the Bedrock calling convention, not the C++ one.
// They're defined in runtime/syscall_entry.S, which adapts them to the C++
// calling convention.

#ifndef TRUSTANCHOR_RUNTIME_SYSCALL_ENTRY_H_
#define TRUSTANCHOR_RUNTIME_SYSCALL_ENTRY_H_

// Immediately halts Bedrock execution, printing an error.
// Prototype: void sys_abort();
#define SYS_ABORT_NUMBER 0

// (Server) Creates a new socket and sets it to listen on a given port.
// Returns a handle to the socket.  If an error occurs, execution halts.
//
// Currently, two ports--0 and 1--are legal, and they identify the encryption
// service and the logging service, respectively.
//
// Prototype: uint32_t sys_listen(uint32_t port);
#define SYS_LISTEN_NUMBER 1

// (Server) Grabs a new incoming connection on the given socket, returning a
// file descriptor referring to that connection.  Blocks until a connection
// comes through.  If an error occurs, execution halts.
//
// Prototype: uint32_t sys_accept(uint32_t socket);
#define SYS_ACCEPT_NUMBER 2

// (Client) Connects to a socket listening on a given address.  Returns a
// description referring to the created connection.  If an error occurs,
// execution halts.
//
// Prototype:
//   uint32_t sys_connect(const char* address, uint32_t address_length);
#define SYS_CONNECT_NUMBER 3

// (Client/server) Reads a certain number of bytes from a file descriptor into
// a buffer.  Returns the number of bytes read.  Unlike Unix, this does _not_
// block!  If not enough bytes are available, 'read' will read as many as
// possible.
//
// Prototype: uint32_t sys_read(uint32_t fd, void* buffer, uint32_t n_bytes);
#define SYS_READ_NUMBER 4

// (Client/server) Writes a certain number of bytes from a buffer into a file
// descriptor.  Returns the number of bytes written.  Like 'read', this will
// not block.
//
// Prototype: uint32_t sys_write(uint32_t fd, void* buffer, uint32_t n_bytes);
#define SYS_WRITE_NUMBER 5

// (Client/server) Closes a file descriptor.  This is _not_ the function to use
// to close a socket!  (In fact, no such function exists.  Don't create too
// many sockets.)
//
// Prototype: void sys_close(uint32_t fd);
#define SYS_CLOSE_NUMBER 6

// (Client/server) The last two system calls, 'declare' and 'wait', can be used
// to implement blocking I/O on top of these primitives.  They mimic the Linux
// epoll(7) interface.
//
// The most basic use case is waiting for a file descriptor to have data
```

```
// available.  To do so, call
//
//   declare(some_fd, false);
//   wait(true);
//
// The second call will then block until data are available on some_fd.  (Don't
// worry about the boolean arguments for now; you will probably not want to
// change them.  We discuss them later.)
//
// However, we can get much more complicated.  For instance, say you have two
// FDs you'd like to monitor.  You can do
//
//   fd1_reservation = declare(fd1, false);
//   fd2_reservation = declare(fd2, false);
//   event = wait(true);
//
// Now, we're capturing return values from 'declare'.  Every time 'declare' is
// called, it returns a 'reservation'--a receipt that identifies the request to
// monitor an FD.  When 'wait' returns, it will return either fd1_reservation
// or fd2_reservation.  You can thus figure out which FD got an event by
// looking at the value in 'event':
//
//   if (event == fd1_reservation) {
//     // fd1 has data
//   } else {
//     assert (event == fd2_reservation);
//     // fd2 has data
//   }
//
// Note that when 'wait' returns, the receipt it returns becomes invalid.  So
// if you want to repeatedly wait on an FD, you need to get a new reservation
// each time:
//
//   while (true) {
//     declare(some_fd, false);
//     wait(true);
//     // process some_fd
//   }
//
// Finally, the boolean arguments: If you pass 'true' to 'declare', you can get
// notified when _writes_ occur to an FD.  (We're not sure why you would want
// to do this, but the functionality is there.)  More usefully, if you pass
// 'false' to 'wait', 'wait' itself will become nonblocking, and it will return
// 0xffffffff if no event is available.  This might be useful if you're
// managing your own event loop or something like that.
//
// Prototypes:
//   uint32_t sys_declare(uint32_t fd, bool monitor_writes);
//   uint32_t sys_wait(bool blocking);
#define SYS_DECLARE_NUMBER 7
#define SYS_WAIT_NUMBER 8

#endif  // TRUSTANCHOR_RUNTIME_SYSCALL_ENTRY_H_
```

runtime/syscall_entry.S

```
// Interwork between Bedrock calling convention and C++ calling convention so
// the former can call functions in the latter.  I would have done this as C++
// with inline assembly, but current versions of GCC (4.9) have a codegen bug
// involving structs in naked functions, so it's easier to just do it as
// assembly.

.syntax unified
.text

#include "runtime/syscall_entry.h"

// Bedrock system call entry points.  These are the actual system call symbols
// declared in runtime/syscalls.h.

.globl sys_abort
sys_abort:
        mov r2, #SYS_ABORT_NUMBER
        b SyscallEntry

.globl sys_listen
sys_listen:
        mov r2, #SYS_LISTEN_NUMBER
```

```
        b SyscallEntry

.globl sys_accept
sys_accept:
        mov r2, #SYS_ACCEPT_NUMBER
        b SyscallEntry

.globl sys_connect
sys_connect:
        mov r2, #SYS_CONNECT_NUMBER
        b SyscallEntry

.globl sys_read
sys_read:
        mov r2, #SYS_READ_NUMBER
        b SyscallEntry

.globl sys_write
sys_write:
        mov r2, #SYS_WRITE_NUMBER
        b SyscallEntry

.globl sys_close
sys_close:
        mov r2, #SYS_CLOSE_NUMBER
        b SyscallEntry

.globl sys_declare
sys_declare:
        mov r2, #SYS_DECLARE_NUMBER
        b SyscallEntry

.globl sys_wait
sys_wait:
        mov r2, #SYS_WAIT_NUMBER
        b SyscallEntry

// Calls BedrockSyscall with three arguments: the Bedrock stack pointer, the
// Bedrock continuation pointer, and the Bedrock system call number.
SyscallEntry:
        // Stash away a copy of the Bedrock stack pointer and continuation
        // pointer.
        push {r0, r1, lr}
        // Do the syscall.
        bl BedrockSyscall
        // Save the result for Bedrock.
        mov r2, r0
        // Restore the Bedrock stack pointer and continuation pointer.
        pop {r0, r1, lr}
        // Jump back into Bedrock code.
        mov pc, r1
```

runtime/syscall_implementation.cc

```
// Bedrock system call implementations.

#include <cassert>
#include <cstdint>
#include <cstring>
#include <memory>
#include <sstream>
#include <string>

#include <mbed.h>

#include "runtime/aes.h"
#include "runtime/boot.h"
#include "runtime/connection.h"
#include "runtime/io.h"
#include "runtime/handle.h"
#include "runtime/reservation.h"
#include "runtime/singleton.h"
#include "runtime/syscall_entry.h"

// System call implementations.
namespace {

// Constant for |wait| to indicate no reservation can be satisfied.
```

```
constexpr uint32_t kNoReservationIsReady = 0xffffffff;

// Tells you what system call got called.
std::string SyscallName(const int syscall_number) {
  switch (syscall_number) {
    case SYS_ABORT_NUMBER:   return "abort";
    case SYS_LISTEN_NUMBER:  return "listen";
    case SYS_ACCEPT_NUMBER:  return "accept";
    case SYS_CONNECT_NUMBER: return "connect";
    case SYS_READ_NUMBER:    return "read";
    case SYS_WRITE_NUMBER:   return "write";
    case SYS_CLOSE_NUMBER:   return "close";
    case SYS_DECLARE_NUMBER: return "declare";
    case SYS_WAIT_NUMBER:    return "wait";
    default:
      std::ostringstream name;
      name << "<unknown: " << syscall_number << ">";
      return name.str();
  }
}

// Converts a Bedrock buffer address to an actual pointer.
void* BedrockBuffer(const uint32_t bedrock_offset) {
  return reinterpret_cast<void*>(reinterpret_cast<uintptr_t>(&bedrock_heap)
                                 + bedrock_offset);
}

}  // namespace

namespace trust_anchor {

void SysAbort() {
  SERIAL_DEBUG("abort: Called.\r\n");
  error("Bedrock program terminated.");
}

uint32_t SysListen(const uint32_t port) {
  // We can listen for crypto requests or for log requests.
  uint32_t result;
  switch (port) {
    case 0:
      result = Listener(Listener::Channel::kEncryption).EncodeForBedrock();
      break;
    case 1:
      result = Listener(Listener::Channel::kLogging).EncodeForBedrock();
      break;
    default:
      error("listen: Invalid port ‘%u’.\r\n", port);
      break;
  }
  return result;
}

uint32_t SysAccept(const uint32_t socket) {
  // |socket| corresponds to a |Listener|.
  std::shared_ptr<Handle> handle = Handle::DecodeFromBedrock(socket);
  assert(handle->IsListener());
  // Wait until somebody shows up on the serial port asking for a connection to
  // the requested channel.
  const uint32_t channel_id = dynamic_cast<Listener*>(handle.get())->channel();
  const auto new_session = Session::Accept(
      Singleton<USBSerial>::get(),
      [channel_id](const SessionID& id) {
        return id.value() % 2 == channel_id;
      });
  return SessionHandle(new_session->session_id()).EncodeForBedrock();
}

uint32_t SysConnect(const void* const address, const uint32_t address_length) {
  // |address| can only be "AES" right now.
  const std::string aes_address = "AES";
  if (std::memcmp(address, aes_address.data(), aes_address.size())) {
    char bad_address[address_length + 1];
    std::memcpy(bad_address, address, address_length);
    bad_address[address_length] = ’\0’;
    error("connect: Invalid address ‘%s’.\r\n", bad_address);
  }
  return Singleton<AESHandle>::get()->EncodeForBedrock();
}
```

```
uint32_t SysRead(const uint32_t fd, void* const buffer,
                 const uint32_t n_bytes) {
  // |fd| corresponds to a handle.
  std::shared_ptr<Handle> handle = Handle::DecodeFromBedrock(fd);
  if (handle->IsAESHandle()) {
    if (n_bytes != AES_128_KEY_SIZE) {
      error("read: Invalid AES block size ‘%d’ (expected ‘%d’).\r\n", n_bytes,
            AES_128_KEY_SIZE);
    }
    std::memcpy(buffer, Singleton<AESHandle>::get()->last_result().data(),
                AES_128_KEY_SIZE);
    SERIAL_DEBUG("read: filled buffer with ‘%s’.\r\n",
                 trust_anchor::ShowHex(reinterpret_cast<uint8_t*>(buffer),
                                       AES_128_KEY_SIZE)
                   .data());
    return AES_128_KEY_SIZE;
  } else {
    assert(handle->IsSessionHandle());
    // Look up the session.
    auto* session = dynamic_cast<SessionHandle*>(handle.get())->GetSession();
    if (!session) {
      error("read: Invalid FD.");
    }
    // Do the read.
    const auto bytes = session->Read(n_bytes);
    std::memcpy(buffer, bytes.data(), bytes.size());
    return bytes.size();
  }
}

uint32_t SysWrite(const uint32_t fd, void* const buffer,
                  const uint32_t n_bytes) {
  std::shared_ptr<Handle> handle = Handle::DecodeFromBedrock(fd);
  if (handle->IsAESHandle()) {
    if (n_bytes != AES_128_KEY_SIZE) {
      error("write: Invalid AES block size ‘%d’ (expected ‘%d’).\r\n", n_bytes,
            AES_128_KEY_SIZE);
    }
    AESHandle::Block key = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    AESHandle::Block plaintext;
    std::memcpy(plaintext.data(), buffer, AES_128_KEY_SIZE);
    Singleton<AESHandle>::get()->Encrypt(key, plaintext);
    return AES_128_KEY_SIZE;
  } else {
    assert(handle->IsSessionHandle());
    auto session = dynamic_cast<SessionHandle*>(handle.get())->GetSession();
    if (!session) {
      error("write: Invalid FD.");
    }
    // Do the write.
    uint8_t* const byte_buffer = reinterpret_cast<uint8_t*>(buffer);
    SERIAL_DEBUG("write: writing ‘%s’.\r\n",
                 trust_anchor::ShowHex(byte_buffer, n_bytes).data());
    return session->Write(ByteVector(byte_buffer, byte_buffer + n_bytes));
  }
}

uint32_t SysClose(const uint32_t fd) {
  // |fd| corresponds to a handle.
  std::shared_ptr<Handle> handle = Handle::DecodeFromBedrock(fd);
  if (handle->IsSessionHandle()) {
    auto session = dynamic_cast<SessionHandle*>(handle.get())->GetSession();
    Session::Close(session->session_id());
    return 0;
  } else {
    return 0xffffffff;
  }
}

uint32_t SysDeclare(const uint32_t fd, const bool monitor_writes) {
  SERIAL_DEBUG("declare: interested in fd 0x%08x for %s.\r\n",
               fd,
               monitor_writes ? "writes" : "reads");
  return Reservation::Make(Handle::DecodeFromBedrock(fd),
                           monitor_writes ? Handle::Direction::kWrite
                                          : Handle::Direction::kRead);
}

uint32_t SysWait(const bool blocking) {
```

```
  // Busy-wait for a reservation to become ready.  Yes, busy-waiting is nasty,
  // but it's easy to do, and we don't care about power consumption or resource
  // starvation for this demo anyway.
  SERIAL_DEBUG("wait will%s block.\r\n", blocking ? "" : " not");
  do {
    if (const auto reservation_id = Reservation::FindSatisfiable()) {
      return *reservation_id;
    }
  } while (blocking);
  return kNoReservationIsReady;
}

// Generic system call dispatcher.  Make sure the compiler emits code for this!
#define USED __attribute__((__used__))
extern "C" uint32_t USED
    BedrockSyscall(uint32_t, void (*)(), int);
#undef USED

uint32_t BedrockSyscall(const uint32_t stack_offset,
                        void (* const continuation)(),
                        const int syscall_number) {
  SERIAL_DEBUG("Entered C++ syscall dispatcher for syscall %s.\r\n",
               SyscallName(syscall_number).c_str());
  uint32_t result;
  // Get arguments off the Bedrock stack.
  const uint32_t* const arguments =
      bedrock_heap
      + 1  // skip over return pointer
      + (stack_offset / sizeof(uint32_t));  // convert to word offset
  // Dispatch.
  switch (syscall_number) {
    case SYS_ABORT_NUMBER:
      SysAbort();
      result = -1;
      break;
    case SYS_LISTEN_NUMBER:
      result = SysListen(arguments[0]);
      break;
    case SYS_ACCEPT_NUMBER:
      result = SysAccept(arguments[0]);
      break;
    case SYS_CONNECT_NUMBER:
      result = SysConnect(BedrockBuffer(arguments[0]), arguments[1]);
      break;
    case SYS_READ_NUMBER:
      result = SysRead(arguments[0], BedrockBuffer(arguments[1]), arguments[2]);
      break;
    case SYS_WRITE_NUMBER:
      result = SysWrite(arguments[0], BedrockBuffer(arguments[1]),
                        arguments[2]);
      break;
    case SYS_CLOSE_NUMBER:
      result = SysClose(arguments[0]);
      break;
    case SYS_DECLARE_NUMBER:
      result = SysDeclare(arguments[0], arguments[1]);
      break;
    case SYS_WAIT_NUMBER:
      result = SysWait(arguments[0]);
      break;
    default:
      error("Unrecognized syscall.");
      result = -1;
      break;
  }
  // All done.
  SERIAL_DEBUG("System call returns 0x%08x.\r\n", result);
  return result;
}

}  // namespace trust_anchor
```

References

Appel, Andrew. *Software Verification*. Lecture series. At: *The Oregon Programming Languages Summer School*. (Eugene, Oregon, 16–28 June 2014). URL: https://www.cs.uoregon.edu/research/summerschool/summer14/.

Chicken Scheme: A Practical and Portable Scheme System. URL: http://call-cc.org/.

Chlipala, Adam. 'The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier'. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. (Boston, 25–27 Sept. 2013). New York: ACM, pp. 391–402. DOI: 10.1145/2500365.2500592.

Chlipala, Adam. 'The Bedrock Tutorial'. 28 Mar. 2013. URL: http://plv.csail.mit.edu/bedrock/tutorial.pdf.

Chlipala, Adam. 'From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification'. In: *Proceedings of the 42nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. (Mumbai, 15–17 Jan. 2015). New York: ACM, pp. 609–622. DOI: 10.1145/2676726.2677003.

FreeRTOS. URL: http://www.freertos.org/.

Geiselbrecht, Travis. *The LK Embedded Kernel*. URL: https://github.com/travisg/lk.

Huang, Andrew 'bunnie'. 'On Hacking MicroSD Cards'. In: *bunnie:studios* (29 Dec. 2013). URL: http://www.bunniestudios.com/blog/?p=3554.

INRIA. *CompCert*. URL: http://compcert.inria.fr/.

INRIA. *The Coq Proof Assistant*. URL: https://coq.inria.fr/.

Java. URL: https://java.com/.

Jim, Trevor et al. 'Cyclone: A Safe Dialect of C'. In: *Proceedings of the 2002 USENIX Annual Technical Conference*. (Monterey, California, 10–15 June 2002). URL: https://www.usenix.org/legacy/event/usenix02/jim.html.

Leroy, Xavier. 'Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant'. In: *SIGPLAN Notices* 41.1 (2006), pp. 42–54.

LexisNexis. 'How Many Pages in a Gigabyte?' URL: http://lexisnexis.com/applieddiscovery/lawlibrary/whitePapers/ADI_FS_PagesInAGigabyte.pdf.

Malecha, Gregory, Adam Chlipala and Thomas Braibant. 'Compositional Computational Reflection'. In: *Interactive Theorem Proving. 5th International Conference, ITP 2014.* (Vienna, 14–17 July 2014). Ed. by Gerwin Klein and Ruben Gamboa. Lecture Notes in Computer Science 8558. Springer, 2014. DOI: 10.1007/978-3-319-08970-6.

Ni, Zhaozhong and Zhong Shao. 'Certified Assembly Programming with Embedded Code Pointers'. In: *Conference Record of the 33rd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. (Charleston, South Carolina, 11–13 Jan. 2006). New York: ACM, pp. 320–333. DOI: 10.1145/1111037.1111066.

Oracle. 'Frequently Asked Questions: Oracle Java ME Embedded 8 and 8.1'. URL: http://www.oracle.com/technetwork/java/embedded/javame/embed-me/documentation/me-e-otn-faq-1852008.pdf.

The Rust Programming Language. URL: http://www.rust-lang.org/.

Texas Instruments. *Automotive: Low-Cost Anti-Lock Braking System*. 2015. URL: http://www.ti.com/solution/antilock_braking_system.

United States National Institute of Standards and Technology. 'Specification for the Advanced Encryption Standard'. In: Federal Information Processing Standards Publication 197 (26 Nov. 2001). URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

Zimmerman, Hubert. 'OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection'. In: *IEEE Transactions on Communications* 28.4 (Apr. 1980), pp. 425–432. DOI: 10.1109/TCOM.1980.1094702.