A Formal Methods Safe Harbor

by

Clark Wood

Submitted to the Institute for Data, Systems, and Society
in partial fulfillment of the requirements for the degree of

Master of Science in Technology and Policy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2019

Author .....
Institute for Data, Systems, and Society
May 16, 2019

Certified by .....
Daniel Weitzner
Principal Research Scientist
Thesis Supervisor

Certified by .....
Adam Chlipala
Associate Professor
Thesis Supervisor

Accepted by .....
Noelle Eckley Selin
Director, Technology and Policy Program Associate Professor, Institute for Data, Systems, and Society and Department of Earth, Atmospheric and Planetary Sciences

A Formal Methods Safe Harbor

by

Clark Wood

Submitted to the Institute for Data, Systems, and Society
on May 16, 2019, in partial fulfillment of the
requirements for the degree of
Master of Science in Technology and Policy

Abstract

We discuss a problem: Internet of Things devices running software are vulnerable to accidents and exploitation, a technology solution: preventing exploitable bugs by developing machine-checked proofs of software correctness and security, and a policy lever to incentivize adoption of this solution: a safe harbor from FTC unfairness prosecution for manufacturers that use formal methods to guarantee safer, more secure devices. To motivate the potential of formal methods, we present a technical contribution: a formally verified connected lightbulb switch, proven immune to certain types of software exploits. We discuss a framework, the Common Weakness Enumeration, that the FTC and manufacturers could use as a shared language to explain what classes of software vulnerability a manufacturer will defend against. We outline the authority of the FTC in regards to poor data security practices as unfair practices and how our safe harbor would both provide immunity to participants and be updated over time to continue to incentivize ever stronger software protections.

Thesis Supervisor: Daniel Weitzner
Title: Principal Research Scientist

Thesis Supervisor: Adam Chlipala
Title: Associate Professor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Executive Summary

Internet of Things (IoT) devices that have exploitable software vulnerabilities can harm consumers in many ways. They can expose consumers' data, violate their privacy, or be used by criminals to launch cyberattacks. Formal methods, a branch of computer science concerned with specifying the properties of software and hardware, can provide strong guarantees of security by proving that software has properties, such as memory safety, that render it immune to certain types of exploits. We therefore recommend that the Federal Trade Commission (FTC) incentivize companies to use formal methods and make products safer by offering them safe harbor from certain IoT data-security requirements and the right to market their devices with regards to cybersecurity if they self-certify their use of formally verified software. We also demonstrate the potential viability of state-of-the-art formal methods by implementing a formally verified IoT lightbulb.

The market for IoT devices rewards cheap, first-to-market products, which incentivizes manufacturers to build devices quickly. Because the quickest path is often less secure, this leads to a market flooded with insecure devices. Compounding this issue is the lack of sunsetting requirements for IoT device software: yesterday's devices are still in use throughout the world, unpatched and connected to the Internet, which allows criminals to exploit devices for their data or co-opt them into botnets. As things stand, it seems likely that there will continue to be many insecure devices in the world with access to consumers' private data.

However, it is possible to prove with mathematical certainty that a piece of software is correct. For example, when NASA writes software for the space shuttle, engineers develop specifications for what the software should and should not do, and have computers automatically check that the software matches that specification [17]. Such tools are formal methods, a way of proving a piece of software is correct, rather than trying to find bugs by testing or reduce them by employing more or better developers. Formal methods are one method that manufacturers could use to achieve a high degree of certainty that their devices are more secure. We outline the current state of formal methods, provide a roadmap that discusses what gaps remain for different types of devices and how to fill these gaps, and build our own formally verified IoT device, using a software language built by MIT graduate students and commodity hardware. Our device acts as a simple IoT lightbulb switch, receiving and parsing network packets in order to control a lightbulb remotely. We prove that this IoT device's software does not contain certain types of bugs that hackers tend to leverage to exploit software.

To accomplish this, we write software that receives commands over a network. Based on which command is sent, the lightbulb is switched either on or off. We also define a specification for this simple IoT device. It should read network traffic, correctly parse the packets, and, if and only if a packet and its data are correct, switch a lightbulb on or off. We verify that the implementation matches the specification and that certain security properties, which render the device immune to some types of attacks, hold.

These defenses are not merely academic: consider the Israeli security researchers that hacked Philips Hue lightbulbs by exploiting a vulnerability in the Zigbee network protocol [52]. This hack also involved user input crafted to exploit a vulnerability in a network protocol, following by remotely controlling an IoT device. The researchers did not detail their findings, but most successful exploitation involves reverse engineering the software of a device in order to find bugs that can be leveraged by attackers. Our formally verified device, unlike the Phillips Hue, should not have any exploitable bugs, and would not have been affected by an attack like this.

The Philips Hue research bears similarities to real-world attacks, such as the massive botnets formed of IoT devices that were remotely exploited and taken over. As formally verified protocols, libraries, and operating systems develop, incentivizing their use in IoT devices can help avoid significant harm to consumers by preventing many different types of attacks without costly patching and maintenance.

We next recommend that, in order to encourage the development of secure IoT devices, the FTC nudge manufacturers toward using formally verified software. We discuss why we think the FTC is a promising policy vehicle for incentivizing more secure devices, why alternatives like requiring that manufacturers continue to patch old devices forever or regulating the quality of software development are heavy-handed and will stifle innovation by leaving only well-funded incumbents able to produce new devices. We advocate a light touch: that the FTC establish a safe harbor for companies that build and leverage formally verified software that implements a secure specification.

Chapter 2

Formal Methods

In this section, we discuss how software can be incorrect or insecure, how this can harm people, and how people try to produce correct and safe software using mathematical techniques and tools called formal methods. We provide an overview of proof assistants, a type of formal verification tool that helps a person to interactively prove conjectures, focusing on Coq, the proof assistant used for our technical contribution. We also discuss Bedrock 2, the embedded domain-specific language that we used. We provide an overview of recently finished and ongoing large-scale efforts in formal methods, and then conclude the section with a roadmap enumerating formal methods challenges specific to IoT devices and questions about funding and use outside of academia. This progress in producing highly reliable and secure software at scale greatly benefits consumers, and we contend later in Section IV that, with the proper policy incentives, this progress could be accelerated, benefitting both consumers and government.

2.1 Background

Over the years, the field of formally methods has progressed from computer-assisted proofs that prove the four color theorem to large ecosystems used to prove comprehensive properties about entire operating systems. The seL4 microkernel [40], for example, is a simple operating system that researchers have proven implements a

memory safe specification. It is open-source, runs natively on existing hardware platforms, and provides enough functionality that manufacturers could, in theory, build IoT devices on top of seL4, similarly to how they currently build IoT devices on top of the Linux kernel. There are disadvantages, however. The seL4 microkernel is not nearly as functional as the Linux kernel, and does not run on nearly as many different hardware platforms. But, especially for IoT devices with simple functionality and reasonable performance requirements, we think the overhead is manageable and will shrink as formally verified software improves. There may come a point, for example, at which the ease of using the Linux kernel is outweighed by the many previous and future vulnerabilities that have been found.

2.1.1 Common Software Weaknesses

Formally verified security properties, like those found in the proof of memory safety in seL4, for example, are important because they provide high assurance of security. For example, certain exploitable conditions like buffer overflows, null pointer dereferences, and integer overflows do not exist in the seL4 microkernel. In unverified systems, attackers will find and take advantage of these software weaknesses. But developers should not be required to read the entire specification of formally verified software in order to use it.

Therefore, it would be helpful if developers could communicate what software bugs they had formally verified their software against. We recommend the Common Weakness Enumeration (CWE), a taxonomy developed by MITRE, that describes recurring software flaws. For example, buffer overflows, when a program copies the contents of a larger buffer into a smaller buffer and the difference between the two buffers spills out of the smaller buffer and potentially into critical parts of the program, are CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). Null pointer dereferences, when a program dereferences a pointer that points to NULL, or 0, an invalid virtual memory address, are CWE-476: NULL Pointer Dereference. Integer overflows, or when an integer value is incremented past its maximum possible hardware value, and wraps around to an incorrectly small or

negative value, are CWE-190: Integer Overflow or Wraparound [19]. When a specific example of a general software weakness is discovered and disclosed to MITRE, it is assigned a Common Vulnerabilities and Exposures (CVE) ID. Heartbleed, for instance, was assigned CVE-2014-0160, the 160th vulnerability posted in 2014, and the report links to resources about the technical details of the vulnerability and describe it as an instance of CWE-119: Buffer Errors [48].

The CWE system, however, covers more than just memory safety bugs, because there are many different conditions that attackers can exploit. And seL4's proof does not, for example, guarantee that it uses sufficiently random values (CWE-330), because it does not include a formally verified cryptography library, or path traversal bugs (CWE-20), because it doesn't include a file system. Further, immunity to certain classes of software weaknesses, while admirable, doesn't necessarily make a device secure. The Mirai malware, which co-opted hundreds of thousands of devices into a botnet used to conduct Distributed Denial of Service (DDoS) attacks against journalists websites and web security companies, primarily infected devices by trying a shortlist of common default passwords [5].

2.1.2 Harm Caused by Software

We argue that formal methods should be considered more broadly as a way to mitigate the damage caused by unsafe software. First, software bugs can lead to accidents that result in significant harm to consumers. Take, for example, the fatal doses of radiation delivered by a misconfigured medical device at the Panama National Institute of Oncology in the 1990s [37]. If a Theratron 780-C Cobalt 60 teletherapy system, a medical device used to deliver radiation therapy to cancer patients, was configured improperly, it would interpret data input by technicians in such a way that it would deliver far too much radiation to the patient, sometimes as much as double the appropriate dose. Software bugs, combined with the understandable human errors that overworked medical professionals might occasionally make, directly contributed to accidents leading to the deaths of at least 5 patients. If developers had instead developed a specification for the Theratron, working with medical professionals to

figure out safety properties, like the appropriate dose of radiation, and then formally verified that the software never delivered an overdose, then this accident could have been prevented.

And buggy software does not only allow for the possibility of serious accidents. It also presents a weakness that attackers may exploit. Consider, for example, the 2017 breach of Equifax. Equifax is a large credit reporting agency, and as part of its business, it stores important personal information, like Social Security Numbers and credit card numbers, for millions of U.S. citizens. Sometime in 2017, attackers breached Equifax's networks, and over the course of several months and multiple campaigns, stole at least 145.5 million US citizens' personal information. The attackers initially achieved access to Equifax's systems by leveraging a known exploit in the open-source Apache Struts software, specifically CVE-2017-5638, an example of CWE-20: Improper Input Validation in software that parsed Multipart HTTP requests [18]. Although this vulnerability had only been publicly released a few days earlier, a patch already existed to fix it. Equifax had failed to apply this patch, leaving their systems exploitable by anyone capable of scanning for known vulnerabilities with existing proof-of-concept exploitation code and the willingness to break the law [20]. If they instead used software that had been formally proven to correctly implement Multipart HTTP requests, then attackers would have had to find another way to steal consumer data.

Whether or not consumers actually suffered harm is an ongoing question, but it appears that plaintiffs, 96 consumers that allege injury because of Equifax's data security practices, will at least be heard in court, after a Georgia district judge denied Equifax's motion to dismiss in January 2019 [24].

2.1.3 Methods of Mitigation

Ideally, software engineers would be able to give a high degree of assurance that no bugs, exploitable or otherwise, existed in any part of a program or system. However, for most sufficiently large programs written in common general purpose programming languages, this is intractably difficult [54]. Instead, organizations have, over time,

developed simpler methods that are cheaper to implement. They might, for example, develop and require adherence to coding standards, such as Mozilla's Secure Coding Guidelines for Web Applications [67] or NASA's Software Safety Guidebook [45]. They could also manually write or generate tests for software, or test software for reliability by providing random and unexpected input and watching to see if this input produces unexpected output or crashes.

However, neither of these methods of improving software quality can give the same level of guarantee as a machine-checked proof. Take, for example, the array buffer overflow bug. A team of software developers could try to develop a program that was safe with respect to buffer overflows in C. They might research existing secure coding standards for C, then make sure that developers adhered to these standards, and write tests to try and exercise various paths within the software. On the other hand, this team could instead develop software in Java, a programming language with a specification that mandates that all array accesses are checked at runtime, and that if an access falls outside the bounds of the array, then an exception is raised [39]. This means that a program will crash, but does not allow attacker-controlled input to overflow into other parts of the program. Assuming that developing C is not a hard requirement, this is a much more straightforward way to write safer software.

Using a managed language, like Java, is one method of leveraging formal methods. The Java language has a specification that precludes certain unsafe properties, like buffer overflows. When a programmer writes Java that is compiled and then run, the Java runtime ensure that array accesses are valid. Similarly, programmers could write C and use a compiler and static checker that verify security properties.

But formal methods encompasses a broad array of techniques, used to prove properties in as varied disciplines as hardware, through formal, machine-checkable specifications for hardware instruction sets, rather than the informal, human-language instruction set manuals common today; operating systems, such as by proving the absence of memory corruption bugs in the kernel; and networked, distributed systems, through formally verified networking protocols that can be compared to real world examples to find deviations and bugs. One major advantage of formal methods is

that they apply a scientific method to computer security. Other methods of securing software may rely on humans that will make mistakes, or testing that will be difficult to apply exhaustively [15].

However, formal methods should not be construed as a magic bullet. Much of industry will probably need plug-and-play whole system guarantees before they can adopt formal methods, but systemic guarantees are hard because different components may be incompatible. For example, liveness, or the property that a system will always make progress, rather than stalling indefinitely under certain circumstances, may be proven for an operating system [35]. But some protocols, like TCP/IP, can be implemented such that deadlocks are possible [21]. A developer must decide whether to forego a systemic proof of liveness, or limit implementation to a specific version of TCP/IP. Another problem is that, although tools have improved over time, they still present usability difficulties, which hampers widespread adoption. The formal methods community is trying to fix this problem through educational efforts and tool development*[15].

2.2 Proof Assistants

The formal methods tool we used to develop our proof of concept IoT device is Coq. Coq is an open-source proof assistant, written in OCaml and C, that allows users to interactively develop software programs, which can be emitted in functional languages like Haskell and OCaml, and to write mathematical proofs [59]. Developers have used Coq to develop file systems that do not crash [10], optimizing C compilers that are correct [16], and to machine-checked versions of mathematical proofs like the Four Color Theorem [34].

Proofs are a means of using predefined rules to reason about new rules. Users start with simple, axiomatic statements and progressively build new statements using old ones. Mathematicians have proved statements manually for years, but may introduce mistakes to proofs, and some proofs require too many manual steps for humans to

*Erbsen, Andres. Interview by Clark Wood. April 9, 2019.

finish them in a reasonable time. Therefore, it would be helpful if there existed a means of automating the laborious task of recording and verifying each step in a proof. Proof assistants provide this functionality by implementing the most basic proofs necessary in a small, trusted kernel, and allowing users to build libraries upon it [33].

Proof assistants are flexible, allowing users to formally prove mathematics or the properties of programs and systems, and allow users to iteratively build and step through proofs. When a user wants to prove some property, they describe the goal that they want to reach, and then build a proof by interactively applying tactics. Tactics allow users to either break down an existing, complicated goal into multiple simpler goals, or to solve simple goals. As the user applies tactics, the proof assistant displays the current goals, along with the hypotheses and other information available in the current proof's context, to the user until the proof has been completely solved. The proof assistant can then check the proof to make sure that it is correct [33]. Because proofs may be complicated to solve and require many manual tactic applications, Coq includes a tactics language, Ltac [22], to help automate the application of tactics to terms in a proof. The IoT device, for example, makes heavy use of Ltac that attempts to automatically break down each line of code we write into smaller parts, and then prove those parts.

2.3 Bedrock 2

Bedrock 2 is the successor to Bedrock [14], a previous effort to build a formally verified low-level systems programming language, similar to C, out of certified low-level macros. Bedrock 2 allows users to write applications in a language similar to C. They can then create specifications for parts of their code, such as specifying that, after a function has returned, it will not have modified any parts of memory, and will return an integer value. They can then check their work by proving that the code that they wrote does indeed function as intended. Finally, the user may emit syntactically correct C from Bedrock 2, which can then be compiled and run on the

device of their choosing. Currently, a user writing software in Bedrock 2 can work entirely in Coq for implementation and proof checking, and then compile emitted C with their compiler of choice.

Although Bedrock 2 is a work in progress, it is mature enough to support automatically proving correctness for many C expressions. With some manual effort, we can also prove memory safety properties, such as properly bounded array accesses. If the C expression is not a loop or conditional, then the straightline tactic can often prove the statement correct on its own. Straightline is custom Ltac written by the developers. It is implemented as a pattern match, which takes an expression as input, and tries to match this expression to a series of patterns that correspond with program functionality like loading values from or storing values to memory, performing arithmetic operations, or calling functions. Thus, we repeat the straightline tactic until it fails to make progress in either simplifying the premises or proving the goal. For simple statements, the repeat straightline idiom will completely solve any goals involved.

For conditional statements, like the if-else in C, Bedrock 2 provides the requires statement, which allows a programmer to specify a condition that must hold. In the case where the condition does not hold, the else statement is executed. To prove the conditional, we use the split tactic to proof both the if and the else case. After I manually proved some of these statements in the lightbulb function, Andres refactored my work into a more concise version. The IoT lightbulb implementation also includes a while loop. The implementation of the while loop in Bedrock 2 is similar to the C syntax, but proving loops in Bedrock 2 is more involved than other statements because we must prove correctness regardless of how many times a loop executes.

2.4 Related Work

Two recent, government-funded formal methods programs provide the start of a roadmap for more secure IoT devices: first, the DARPA High-Assurance Cyber Military Systems (HACMS) project [27], an $18 million effort [23] to build a high-

assurance quadcopter and unmanned helicopter that ran from 2012 to 2017. Teams from across industry helped to build the operating system and control system, proved important properties like system-wide memory safety of the vehicle, and released much of their work as open-source libraries and computer languages. The work was successful, but slow and expensive, and the program manager has identified a lack of personnel trained in formal methods as one major barrier to more widespread verification of software [26]. Our technical contribution, to build a fully formally verified simple IoT device, is inspired by HACMS, but achieved at significantly less time and cost.

Another large, integrative effort is DeepSpec, an ongoing, $10 million NSF-funded program to improve state-of-the-art formal methods and develop training material to increase the amount of software engineers proficient in formal methods [47]. Technical outputs of the project include formally verified cryptographic libraries, operating systems, and eventually a web server. Each year, the DeepSpec Summer School educates hundreds of students in modern formal methods. Our technical work uses libraries developed under DeepSpec.

2.5 Roadmap

2.5.1 Challenges Specific to IoT Devices

From these projects, and our own technical contributions, we identify areas of opportunity, where smaller, concerted efforts at different abstraction layers could have a big impact on IoT device security: first, port formally verified software to more hardware platforms, so that it can run on more devices. For example, seL4 already provides a memory safe and correct microkernel, but it has limited functionality when compared to monolithic kernels like Linux. Our work, for instance, supports the SiFive FE310, a board that implements RISC-V, an open-source Reduced Instruction Set Computer architecture, but could be extended to support more platforms, like common microcontrollers or the Raspberry Pi.

Formal methods developers could take cues from industry surveys, such as the IoT Developer Survey 2017, conducted by the Eclipse Foundation. The group surveyed 713 people from industry about a broad range of topics on IoT development, such as the languages, network protocols, and text editors used by respondents. The most common hardware platform was ARM, with 16-bit microcontrollers also being relatively common in embedded devices, while 32 and 64-bit version of x86 and ARM were the most common architectures for routers and other networking devices [38]. Thus, ensuring that tools and existing formally verified code can target these platforms would benefit a large proportion of IoT devices.

Second, increase the number of high level languages that are supported by existing formally verified systems. As with the above, seL4 provides strong guarantees, but currently requires that programmers use a custom build system to write C programs without many of the standard software libraries that many developers rely on. If seL4 included support for higher-level programming languages, like Python or Java [58], then developers could be more productive and introduce fewer bugs when developing on top of seL4. This is also backed up by the IoT Developer Survey, which found languages like Python, Java, C/C++, and Javascript to be most commonly used.

Third, certain software libraries, like networking and cryptography libraries, should be prioritized, since many connected devices end up using the same software. DeepSpec's Fiat Cryptography project is a good example: researchers proved the correctness of cryptographic primitives, and the verified code is now being used in the boringSSL library [3]. Software that is both widely used and safety-critical should have open-source, formally verified options for developers. If a formally verified version of the Zigbee protocol were available, for example, then it is less likely that Israeli security researchers would be able to exploit the Philips Hue. The 2017 Survey also points to protocols like HTTP, the subject of an ongoing DeepSpec project; MQTT, or Message Queuing Telemetry Transport, for which CVEs leading to remote code execution already exist [38]; and CoAP, or Constrained Application Protocol, designed to be a low-power, UDP-based messaging protocol similar to HTTP. Protocols like QUIC, which may end up replacing much existing HTTP traffic, are other prime

targets for formal verification. Our project verifies a subset of UDP over Ethernet. Prioritization may be driven by domain-specific needs as well: for example, the auto industry uses Uptane [65], a variant of The Updater Framework (TUF), to deliver over-the-air software updates to vehicles. As automobiles become more autonomous and connected to their environment, having strong guarantees about their correctness and security becomes more and more important.

Fourth, because many IoT devices have low power constraints, requirements for efficient power use are found in common IoT protocols, like CoAP or Low-Power Wireless Personal Area Networks (6LoWPAN). Consequently, shifting focus from proving that formally verified code has comparable speed, to proving that formally verified code has comparable power usage, may help strengthen adoption.

2.5.2 Making Progress

Perhaps more difficult than deciding what work to prioritize is ensuring that sufficient funds are allocated to research and development of open-source software. Developing network protocol software that is formally verified to correctly implement a specification based on an RFC and to be memory safe may benefit the owner of the intellectual property, but if that work is not released with a permissive open-source license, then it is less likely that many will prefer to use it over existing, and perhaps buggy, but free and open-source versions. We see two paths forward for this: the first would involve private sector adoption and support of formal verification projects, while the second would rely on government-funded development efforts.

First, consider the private funding path. It is not uncommon for for-profit companies to fund not-for-profit software that is important to their business. Apple, for example, has supported LLVM development, and contributed code to the project, in part because they preferred LLVM's license to GCC, another C compiler [6]. Further, there also exist for-profit, open-source companies like Redhat [2], and companies might collectively pay for important software that a group of related companies agree is important: for example, the auto industry might consider funding the development of a formally verified version of Uptane.

However, consider the reasons behind industry funding of OpenSSL, an open-source cryptography library, used by roughly 17% of web servers supporting SSL, when Heartbleed was publicly disclosed. The Heartbleed attack, leveraging a memory safety error in OpenSSL, allowed attackers to remotely steal web server's private keys [36]. With those keys attackers could decrypt network traffic. At the time, the OpenSSL Project consisted of one full-time employee and three volunteers, with an annual budget of roughly $2,000. Software giants did not support the software until they realized how poorly supported such a critical piece of software was [43]. OpenBSD has found itself in a similar predicament [7], unable to continue operating at times without emergency donations. These examples suggest that there might be room for the government to step in and help maintain the commons.

Chapter 3

Technical Contribution

In this section, we discuss the main technical contributions of our team. Clark is a masters student in the Technology and Policy Program. He proposed the thesis topic, implemented the proof-of-concept, and wrote the thesis. Andres is a graduate student in the Programming Languages & Verification Group at CSAIL who helped me use Bedrock 2. Samuel Gruetter, John Grosen, and Adam Suhl also contributed. Below I explain the methodology used to select the hardware on which to implement the IoT lightbulb, how I developed the software for the device, and the results so far. I conclude by discussing a promising method for explaining the correctness and security benefits of formal methods by mapping the properties proven to well-known cyber-security frameworks like MITRE's Common Weaknesses Enumeration.

3.1 Methodology

In order to gauge the reasonableness of this approach, we wanted to implement a simple IoT device. I proposed three initial candidates: a connected bathroom scale, door lock, or lightbulb. I then evaluated these devices based on two criteria: functionality and salience. First, the device should be simple enough that the technical work could be completed in time, but featureful enough that one could reasonably justify connecting the device to the Internet. Second, the existing commercial examples should have well-documented security problems, and the device in question, if

exploited, should have the potential to cause significant harm to users. Note that, while consumer IoT devices often contain functionality unrelated to the main function of the device, I only considered proof-of-concept implementations that handled the essential elements of the device, along with basic network connectivity.

A simple connected bathroom scale would need to weigh the person that stepped on the scale and report it to the user. This doesn't require a connection to the Internet, which more complicated connected scales might use to automatically keep track of a user's weight over time, and so, without additional complications, the scale appeared to be a poor candidate. In addition, hacker control of a bathroom scale, other than as a means to control any Internet-connected device, seems unmotivating.

A door lock would need to receive commands from a user to lock or unlock a door, and only fulfill commands after successful authentication. The former would involve parsing network packets, as with all three devices, and performing a binary function: either lock or unlock a door. However, the latter function, of authentication, is essential to a door's function, but also adds a layer of complexity that is less than ideal for a proof-of-concept. Exploitation of a consumer door lock would be scandalous, but if there is no authentication then the exploitation is trivial.

Therefore, I decided to implement the IoT lightbulb. A simple connected lightbulb switch must parse network packets and switch on or off based on the data received. While more complicated IoT lightbulbs might report energy usage or flash in complicated patterns, this isn't necessary everyday functionality. And a lightbulb could arguably function without authentication. Attacker control of a lightbulb is much less worrisome than attacker control of a door lock, but there are, at least, already examples [52] of security researchers exploiting real IoT lightbulbs.

Andres and I deliberated over which architecture and board to use to implement the connected lightbulb switch, considering, for example, the Raspberry Pi 2, which runs an ARM CPU, and the Arduino, with both ARM and x86 boards available. However, because we wanted a simple platform that was economical and as open as possible, we decided on the SiFive Freedom E310 (FE310). This is a single-core RISC-V device available for around $50. In contrast to proprietary architectures like

ARM and x86, the RISC-V architecture is free and open-source, and intended to support different industry use cases with a simple instruction set that also facilitates research and education about computer architectures [66].

I began by developing an implementation in C to serve as a reference point. This program is compiled with GCC and loaded onto SPI flash memory, where the FE310 will, by default, begin executing after a reboot. The board is then restarted, and can be debugged over remote GDB. After the initial C implementation, I re-implemented the device in Bedrock 2, iteratively porting code, improving our specifications, and re-building our proofs. Andres and John found an exploitable bug in the initial C implementation, and developed a proof-of-concept exploit for it.

Although this device is less sophisticated than commercial IoT devices, it does bear a strong resemblance to devices that have been publicly hacked, as in our running Zigbee exploit example. There, as here, an IoT device both received user input over a network and controlled a physical device based on that input. The Zigbee protocol is more complex than UDP, but the underlying concept, of a network protocol with exploitable bugs, remains the same.

After verifying that the initial implementation worked, I defined a specification of the connected lightbulb switch and re-implemented the parsing and control functionality in Bedrock 2, a research language under active development at the Programming Languages and Verification Group at MIT CSAIL. Bedrock 2, a domain-specific language implemented in the Coq proof assistant, aims to facilitate low-level systems programming while minimizing the direct use of C and assembly. I then proved that the implementation matches the specification, and that the implementation does not contain certain bugs, such as array out-of-bounds accesses.

3.2 Development

3.2.1 Device

After deciding on the FE310, Andres and Adam Suhl worked to set up the board. For a connected lightbulb switch, we needed the ability to receive and parse network packets, and the ability to write values to peripheral hardware. Because the FE310 uses Memory-Mapped Input/Output (MMIO) to allow software written by the user to interact with hardware devices, we needed to identify which memory addresses correspond to which physical pins on the device. We could then connect a control relay to the pin that our software controlled, and plug a lightbulb into the control relay without directly dealing with high-voltage wiring. Andres developed both the initial startup code for the FE310, which included C macros for handling MMIO, reading and writing over Serial Peripheral Interface (SPI), and the commands used to compile, load, and debug the software. Andres and I pair-programmed functionality for receiving Ethernet packets on top of this core. I began by reading the relevant portions of the FE310 datasheet [57] and programming manual [56], along with specifications for the Ethernet protocol, and then implementing portions of the code, working with Andres when I got stuck. I have only implemented a subset of Ethernet, so that the FE310 can receive packets.

After some initial startup code to enable networking, the program continuously checks to see if network data is available on a receive register, and dequeues it for processing if so. It then writes the available bytes into a buffer for further processing. Once the network bytes are in a local buffer, the program parses the buffer to verify whether it contains a valid Ethernet frame, and then to verify whether the frame contains valid UDP header and data. I implemented a simple connected lightbulb switch protocol on top of UDP. A user sends a UDP packet with one byte of data set to 1 to switch the lightbulb on, or 0 to switch the lightbulb off.

Next, I decided on a power relay, so that the memory-mapped register that the software would set to 0 or 1 could safely control an outlet device, here a lightbulb, that required much higher voltage to power. I manually connected a General Purpose

Input/Output (GPIO) pin on the FE310 to the power relay with a wire, and then plugged a fluorescent lamp into the power relay. I wrote software to write a value to the pin and verified that the code could be used to control the lamp.

3.2.2 Verification

Andres began by introducing me to Bedrock 2 and running through some of the existing code examples. He explained the design decisions and common tricks to apply. He walked me through how to write a specification, and then I wrote an implementation and tried to prove that the implementation passed the specification. I worked iteratively: with each line I first tried common tactics like `straightline`, to see if I made progress towards the goal. When the common approaches failed I started to read the definitions of types involved with the goal and then, after Andres taught me, more advanced debugging techniques like manually copying and pasting in tactics to see where they failed. When I got stuck Andres would help me. I do not have write access to the Bedrock 2 Github repository, so I forked it on my own Github account, and pushed my work there.

While developing the IoT lightbulb switch, I also found two bugs in Bedrock 2. First, while learning how to use Bedrock 2, I often stepped through the Example code to better understand how to create specifications, implementations, and proofs. I found that, in Demos.v, the `bsearch` function, a proved implementation of binary search, would be emitted as syntactically incorrect C. Specifically, the function redefined its parameters inside of its block as new variables, resulting in a compilation error. I informed the Bedrock 2 developers, and this was fixed in a later commit.[70]

Second, I found a bug in the syntax translation, where the double-equals operator was translated into less-than-or-equals, rather than equals, probably as the result of a typo originating from the line above. I provided the line number of the bug to the developers and it was also fixed [69]. I also caused a Coq anomaly [68] in Coq 8.9. After discussing how to proceed with Andres, who believed that the issue was probably fixed on the master branch, I built the master branch of Coq, installed Proof General, and switch to developing in Emacs. I also implemented a lemma describing

under what circumstances an unsigned integer doesn't overflow.

3.3 Results

As a result of these efforts I have written software that implements a connected lightbulb switch. It will repeatedly check to see if a command has been sent over the network, parse the command, and then switch the lightbulb on or off based on the command. Implementing software that correctly parses a network packet can be difficult, and, as discussed above, there are examples of exploitable bugs in these protocols. However, since this IoT device was implemented in Bedrock 2, we have a strong guarantee that the implementation is memory safe, and has no exploitable memory corruption bugs. To do this, I wrote software to perform the above functions, a specification for each of these functions that indicates what should be true before and after these functions execute, and then a proof script for each function that verified, line-by-line, that the function was memory safe and that the implementation adhered to the specification.

3.3.1 Code Review

The Appendix contains figures showing the software for our technical contribution. There is one top-level function, `iot`, that calls a function to read an Ethernet packet into memory, and then calls another function to parse that packet and, based on its data, either switch the lightbulb on or off. For each of these three parts, I implement the function, declare a specification which describes what the function expects and what it will do, and prove that the implementation matches the specification. Implementations begin with the keyword `Definition`. I then declare function variables with `let`, and write the function in `bedrock_func_body`. Specifications begin with `Instance`. I then specify conditions for before the function is called with right arrows, such as that an array of a certain length exists, and lastly specify the function call itself, and what will happen after it exits. For example, one may specify that a function returns a value, or that it modifies memory. Lastly, proofs begin by declaring

a lemma, and then proving it, ending with the keyword `Qed`. Over the course of the proof I iteratively step through all paths in the program.

Figure A-1 shows the implementation for the `iot` function. It first passes an array to the `recvEthernet function`, which will copy a network packet into that array. The `require` statement checks the return value, and if it is -1, the designated error value, then the program exits, otherwise it continues. It then passes the address, and the length of the network packet, into the `lightbulb` function, which will switch the lightbulb on or off. Finally, it returns a 0 for success. The `unpack!` keyword specifies that, because `recvEthernet` and `lightbulb` are functions, in order to verify `iot` one must verify them as well.

Figure A-2 is the specification for the `iot` function. Of interest are the preconditions for `iot`, that there will be an array in memory starting at `p_addr` called `rx_packet`, and that the length of `rx_packet` will be 1520, and the postcondition, that `iot` will return a value.

Figure A-3 is the proof for `iot`. Because `iot` is a wrapper around the `recvEthernet` and `lightbulb` functions, it mostly involves using Bedrock 2 tactics like `straightline`, and its call and if variants, to steps through statements in the function body. The `ecancel_assumption` tactic uses separation logic to ensure that memory accesses are valid.

Figure A-4 is the implementation of `recvEthernet`. It begins by reading machine-specific parts of memory to see if a network packet has been received. The `io!` keyword indicates that a line will be reading memory, and that we must therefore prove that we are allowed to read that part of memory. The program requires that a packet is queued; otherwise it exits. It then reads the size of the packet in bytes and pads that number to the next word, since the Bedrock 2 primitive for reading operates with 4-byte words, but the peripheral device writes the size of the packet in bytes. The function then checks that the size of the packet, `num_bytes`, is smaller than the size of the array used to store the packet. If this is true we enter the while loop, which iteratively reads a word using `lan9250_readword` and stores it in the next index of the `rx_packet` array. Lastly, the function returns how many bytes it

has written.

Figure A-5 is the specification for the `recvEthernet function`. The precondition includes an array, `rx_packet`, of length 1520, as with the `iot` function. The postcondition has two possibilities. In the first case, the function succeeded, so it returns `bytes_written`, and it has modified `rx_packet` by writing a number of bytes in the array equal to `bytes_written`. Lastly, `bytes_written` will be less than or equal to the length of this modified array, so this function will not overflow `rx_packet`. In the second case, the function failed, so it did not modify memory, like `rx_packet`, and returns -1.

Figure A-6 is the proof for `recvEthernet`. This part, and particularly proving the while loop, required the most effort by far. It starts similarly to the other proofs, by repeatedly trying built-in Bedrock 2 tactics like `straightline`, covered above, `interact_nomem`, which handles function calls, and `prove_ext_spec`, which handles primitives like `lan9250_readword`. This proves up to the while loop, where we must prove that, for each iteration of the loop, certain conditions, such as not overflowing the buffer, hold true. As with the functions, I must specify preconditions and postconditions for every iteration in the while loop. The conditions begin at `PrimitivePair.pair.mk`. The preconditions for each loop are that there is a variable `c` used to access the array, that the array has a `SCRATCH` section where the function can continue to write, and that `SCRATCH` is large enough. In addition, because the function writes in words, which are 4 bytes on the FE310, `c` and `num_bytes` must be multiples of 4. This is specified as the two being congruent mod 4. Lastly, for each loop iteration, there is a loop variable equal to `num_bytes` that can be used. The postconditions are that there is a new array that consists of the old array and the newly written contents, and that the size of this array and the number of bytes written hasn't changed.

Once in the loop, existing Bedrock 2 tactics have difficulty proving properties about machine words and about memory writes. Andres helped prove a few examples of each of these types of goals, and then I generalized to other parts of the proof. To prove properties about words, for instance, that a given word does not overflow or

underflow, I would use `replace` to, for example, specify that adding 0 to any other word is the same as that word, use `change` to convert word to Zs, which the Coq standard library can reason about with the `lia` tactic, and use `pose proof` to tell Bedrock 2 that unsigned integers should fall within a certain, good range. To reason about memory writes, I would manually manipulate the `rx_packet` array to prove that all writes fall within the array bounds by splitting up arrays into manageable chunks using `List` lemmas like `firstn_skipn`, and then using existing Bedrock 2 tactics to merge the sub-arrays.

Figure A-7 is the implementation of the `lightbulb` control function. The function takes two parameters, the packet and its length. If requires that the packet's size is appropriate, performs some checks to verify that it is a valid Ethernet packet, and then loads the command, to switch the lightbulb on or off, from the packet. It then prepares to write to an MMIO pin by reading and writing to a specific section of memory and then writing the command to another specific section of memory. The `io!` and `output!` keywords indicate that there is a memory read and a memory write respectively. Lastly, it returns 0 on success.

Figure A-8 is the `lightbulb` function specification. Preconditions are an array, `rx_packet`, and that the `len` parameter is not greater than this array's size. The postcondition indicates that the `lightbulb` function will return a value and will not change memory addresses that are not MMIO.

Figure A-9 is the `lightbulb` proof. Similarly to the `iot` proof, this proof mostly uses built-in Bedrock 2 tactics. Along with `straightline` variants, it uses `seplog_use_array_load1` to produce hypotheses that split the existing array into 3 parts: an array before the byte that will be accessed, the byte accessed in the array, and the remainder of the array. This is used to prove that all memory accesses fall within the array bounds.

Figure A-10, A-11, and A-12 show the C that Bedrock 2 emits after each of the proofs pass. This can be compiled and, with some handwritten initialization code, loaded onto the device. One advantage of proving safe array access before emitting C is that the emitted code doesn't have to check array bounds at runtime, as in

languages like Java. We have high assurance that the C we produce is safe without runtime checks that hamper performance.

Figure A-13 is an incomplete specification of the FE310 that I based on another semantics file written by Andres. I also omit some initialization code and helper macros. For example, `MMIOREAD` and `MMIOWRITE` are currently implemented as small C macros. Andres implemented the original macros, and I added them to the FE310 semantics for Bedrock 2 on his advice, defining, for example, the ranges of memory that can be used for MMIO. This ensures that, if the user tried to write a program that accesses memory outside the regions reserved for MMIO, the proof would not pass.

3.4 Lessons Learned

After having worked with Bedrock 2 for a semester, I have noticed some opportunities to improve the language so that it can be more easily used. These opportunities come from difficulties that arose as I tried to develop the connected lightbulb switch. Some of these challenges stem from the incompleteness of Bedrock 2, and would therefore naturally be solved as the language is finished. Others are usability issues that would be difficult for the development team to see in advance, because they're significantly more familiar with formal verification than their target audience, industry software developers, will probably be.

First, when the pre-defined Bedrock 2 tactics work, proofs are easy and straightforward, but when they fail to make progress it can be difficult to ascertain what to do next. Manual proving requires knowledge of at least the Coq proof assistant, if not also of Bedrock 2 and its dependencies. For example, in the first pass at proving the `lightbulb` function, when proving lines that involved access to memory, I would often start by calling the `straightline` tactic, but I would be left with a complicated goal, composed of many subgoals. I might have to introduce variables and split goals repeatedly, or manually apply lemmas about memory. After Andres incorporated this work into the `straightline` tactic, the proof became much shorter and simpler. I

expect that, as Bedrock 2 is used to prove gradually more complicated and realistic programs, this issue will naturally resolve.

Second, while I was using Bedrock 2, I often encountered problems that I could have more quickly overcome if there were better documentation or error messages. Reading the code is possible, but significantly raises the barrier to entry for users. I did not, for example, know that the else case of the require statement immediately exits, that arrays were commutative, which means that an array might be represented in reverse order of what a programmer would expect[†], or that the precondition specified for a loop must hold before each iteration of the loop, rather than just before the first iteration. It may be expected that other developers unfamiliar with formal methods would similarly misunderstand what is banal for formal methods researchers, and such issues could all be solved with documentation.

Third, while implementing a subset of the machine specification was straightforward with help from Andres, I suspect that a complete and correct machine specification would be effortful and error-prone. This is because it would require reading through datasheets and programming manuals at length, and there is no way to verify correctness without testing on the physical machine. Spending the time to develop a full machine specification up-front would help future developers get started immediately.

3.5 Mapping to Software Standards

One challenge with formal methods, especially for an evaluator, is understanding the consequences of proving properties about software. Our work, for example, proves certain memory-safety properties. As a result of this memory safety, we have mathematically backed assurance that certain types of vulnerabilities do not exist in our connected lightbulb switch. This should mean that the light bulb is less prone to

[†]The $*$ operator merges sections of memory, but those sections of memory do not need to be in any particular order. $(A * B)$ and $(B * A)$ would be considered equivalent. But if A begins at offset 0, and B begins at offset $[0 + i]$, and programmers confuse $*$ for an append operation, like $++$, then the equivalence may be unclear.

exhibiting incorrect behavior and less vulnerable to exploitation attempts. However, it is difficult to quantify how much safer the light bulb is.

Further, because of the diversity of tools, techniques, and procedures in the formal-methods community, we don't want to pick winners by suggesting a specific standard. It is unreasonable to suggest that only embedded domain-specific languages, written in Coq, that emit memory-safe C that can be compiled and loaded onto a RISC-V device, should be considered the pathway toward building formally verified IoT devices.

3.5.1 CWE

Therefore, we recommend a results-based approach that leverages existing frameworks that are familiar to the government and IoT manufacturing community, such as MITRE's Common Weakness Enumeration. For each CWE, the developer could describe whether they had used formal methods that should protect the device or software from this weakness. If, for example, they developed their software with Verasco, a formally verified static analyzer [4], then they could fill out weaknesses related to memory safety as defended. However, without further effort, they could probably not claim a defense against CWE-243: Creation of chroot Jail Without Changing Working Directory, because this CWE is orthogonal to the safe C specification that Verasco checks against. This leads to another benefit of mapping formal-methods work into these frameworks; the exercise can inform future work. If a proof assistant does not currently check that emitted C never calls the chroot system call without first changing the working directory, but this condition is considered important enough to have a CWE devoted to it, then developing a proof for it is promising future work.

There are some CWEs that are probably inapplicable, or at least less interesting to include, such as CWE-1041: Use of Redundant Code. Although the use of redundant code may be a symptom of poor software-engineering practices, it seems unreasonable to expect formal methods to produce a helpful result relating to this weakness. If, for example, a developer formally verified that, across all functions, there never existed more than 10 lines of exactly identical code between two functions in some software,

it is unclear both why 10 lines is the cutoff for an unacceptable level of redundancy and why this is a meaningful security property.

Therefore, we propose evaluating formally verified software against an appropriate subset of the CWEs. For an IoT device implemented with a tool that emits C, we argue that the CWEs that MITRE has compiled into the CWE View: Weaknesses in Software Written in C is an appropriate subset.

3.5.2 Other Standards

There exist other standards for secure coding, such as the SEI CERT C Coding Standard, developed by the Software Engineering Institute, or the OWASP Top Ten. These other standards overlap significantly with the CWE taxonomy, and so we see no reason why they could not also be used. In fact, MITRE has already developed mappings from the CWEs to other coding standards. A reasonable criterion for whether or not a given standard is helpful for evaluating formally verified software is whether a large enough community has formed around it: many researchers submit thousands of vulnerabilities in order to obtain CVEs each year.

Chapter 4

Federal Trade Commission

4.1 Background

The FTC, an independent federal executive branch agency charged with protecting consumers and promoting business competition, was established by the Federal Trade Commission Act in 1914 [49]. The FTC has broad authority to investigate, enforce, and litigate violations of the FTC Act, such as practices that damage consumers or the market, under Section 5. If, after investigating a potential violation, it finds that the acts or practices of an entity were unfair or deceptive, or that they violated certain acts of Congress, then the FTC may issue complaints or make rules and then pursue litigation in federal court [1].

4.2 Unfair and Deceptive Practices

Unfair practices are defined in Section 5 of the FTC Act as "likely to cause substantial injury to consumers which is not reasonably avoidable by consumers themselves and not outweighed by countervailing benefits to consumers or to competition". Over the years, the FTC has pursued cases where a company has engaged in practices that caused harm to consumers. The modern FTC approach to judging unfairness requires that, "in order for a practice to be unfair, the injury it causes must be (1) substantial, (2) without offsetting benefits, and (3) one that consumers cannot reasonably avoid".

The FTC's use of its authority to police unfair acts or practices has changed over time, and, in the past, the FTC has construed its authority over-broadly as a vehicle for policing breaches of public policy, such as when it tried to ban all advertising to children because there existed public policies in place to protect children. But, starting in 1980 with its Unfairness Policy Statement, the FTC shifted its focus to using its unfairness authority to combat consumer injury.

To determine if a practice is unfair, the first test is for substantial consumer injury, either small damage to many consumers or significant damage to a small group. In addition, injuries cannot be purely emotional. Second, the injury must not be offset by other benefits to consumers. High prices, by themselves, would not be considered unfair, because a free market will correct these practices in the long run. And third, the injuries must not be reasonably avoidable by consumers, which would rule out litigation of fast food or homeopathic medicine that consumers could reasonably choose not to purchase. J. Howard Beales, the former director of the Bureau of Consumer Protection, has said that the FTC's unfairness authority should be used when there is far-reaching harm to consumers, outweighing the clear benefits, but no clear deceptive practices [60].

To determine if a practice is deceptive, a similar three-part test exists. First, there must be some representation, omission, or practice that either does mislead or is likely to mislead a consumer. Second, a consumer's interpretation of the practice must be considered reasonable in the context of the situation. Third, the misleading practice must be considered material [25].

As of 2002, the FTC has used its unfair and deceptive authority to bring suit against and reach settlements with 65 companies that have inadequately protected user data. They have, for example, settled with Uber over their failure to disclose a data breach, with Venmo over their misrepresentation of data security practices, and with VTech Electronics Limited for failing to use reasonable and appropriate data security measures to protect consumer information. They have also worked in concert with government agencies like the National Institute for Standards and Technology (NIST) and the Department of Homeland Security (DHS) to educate businesses about

best practices in cyber security [32]. We discuss two similar cases: FTC v. Wyndham and FTC v. D-Link, where the facts of the case clearly show that the defendants had ample opportunity to fix obviously flawed data security practices, and failed to do so. We also discuss a case, LabMD Inc. v. FTC, where the FTC appeared to overstep its bounds by issuing a cease-and-desist order that required a company to not only discontinue poor data security practices, but to implement an improved data security program and begin to enforce it.

4.2.1 Stance on Regulation

The Internet of Things, the ever-expanding collection of everyday devices that are becoming connected to computer networks, is of particular interest to the FTC. As consumer devices, like watches and thermostats, become fully-functional computers connected to the internet, they open consumers up to entirely new threats. Connected home devices, for example, might collect and retain private data, and also contain vulnerabilities that allow attackers to exploit these devices and steal that private data. Therefore, the FTC hosts IoT workshops where it solicits comments on how to ensure privacy and security for consumers, issues guidelines and best practices to help manufacturers build and manage IoT devices correctly, and, when necessary, prosecutes IoT manufacturers that it believes have harmed consumers under the Commission's unfair and deceptive practices mandate.

One example of the FTC's ongoing efforts regarding IoT security is the FTC's 2013 workshop, The Internet of Things: Privacy and Security in a Connected World, where participants from industry, academia and government discussed the benefits and risks associated with the Internet of Things, and what role the FTC should play. The FTC summary of the report finds that the FTC should continue to uphold applicable laws, like COPPA and the HI-TECH Act, and to pursue cases where a company has manufactured an IoT device without reasonable levels of security, or has made unfair and deceptive claims about an IoT device, under its Section 5 authority. The FTC also pledges to continue to educate business and consumers about IoT privacy and security, to work with other stakeholders across government and industry, and

to advocate for consumer protection [50]. Our policy recommendations at the end of this section are in line with the FTC's professed goals.

At the workshop, some participants were for, while others were against, regulation of IoT devices. The FTC sided against regulation, agreeing with some participants that self-regulation would allow innovation to continue unabated while encouraging companies to improve device security. Further, existing self-regulatory schemes that IoT manufacturers may voluntarily join, like the Network Advertising Initiative Code of Conduct, include requirements to provide reasonable security for user data [44]. However, what is considered reasonable is left undefined. And it seems reasonable to expect that the definition of reasonable will change with time.

4.2.2 FTC v. Wyndham

Starting around 2008, hackers gained unauthorized access to the computer systems of Wyndham Worldwide, a hotelier. Over the course of multiple intrusions, the attackers stole consumer information used to cause $10.6 million in fraud. Wyndham maintained questionable data practices with its subsidiaries, including allowing them to store credit card data in plaintext, use default passwords, and to connect to Wyndham systems from a computer that had not applied security updates to its operating system in over three years. Further, Wyndham's privacy policy mentions that the company uses encryption and firewalls to protect customer data, but the FTC found that it did not. Attackers took advantage of these weak data security practices and broke into Wyndham networks in April 2008, March 2009 and again in late 2009. However, Wyndham did not learn about these intrusions until credit card companies notified them after receiving complaints of fraudulent charges in 2010 [31].

The FTC filed suit in district court in 2012. Wyndham argued that the FTC's unfairness authority did not apply to data security practices and that, even if its authority did apply, the FTC had failed to provide fair notice about these data security practices that they allegedly violated unfairly. The District Court for New Jersey disagreed, finding that authority over data security practice wasn't exclusive, so the FTC did have authority in addition to other agencies. On appeal, the US 3rd

Circuit also held that the FTC could regulate data security under unfairness, because the FTC Act was purposefully broad, and that the FTC did provide Wyndham with sufficient fair notice because the FTC had statutory authority over unfair practices, and was merely exercising that authority, rather than interpreting statutes or issuing its own regulations. The FTC's efforts to educate companies through guidelines, conferences, and requests for comment are arguably better for consumers than rigid data security checklists that would quickly become outdated [61].

The Wyndham case helps to establish precedent for the FTC's authority to regulate data security under the unfairness portion of the FTC Act. Wyndham's data security practices, combined with its misleading privacy policy, appear to fulfil the criteria for an unfairness case. The harm, millions of dollars in fraudulent credit card charges, along with the time and effort cost to consumers, is substantial. There appear to be no obvious offsetting benefits derived from Wyndham's poor data security practices. And, both because data security is complicated and often opaque to consumers, and because Wyndham's privacy policy contained untrue statements about their practices, it is unreasonable to expect consumers to be able to avoid this harm. This interpretation of poor data security practices as unfair also provides flexibility to the FTC, because the definition of poor data security practices should change over time, holding companies to higher and higher standards.

4.2.3 FTC v. D-Link

In early 2017, the FTC filed a complaint that charged D-Link with selling routers and IP cameras that were insecure, with advertising that the devices were in fact very secure, and with failing to fix particularly egregious security bugs [30]. D-Link is a foreign-owned company that sells IoT devices in the US. The FTC argued that D-Link failed to secure against vulnerabilities that were widely known to the community, including hardcoded credentials, such as usernames and passwords that were well-known and easily guessable values like "admin", exploitable vulnerabilities that are difficult to prevent ahead of time in languages like C, but that should be straightforward to patch, failure to protect the private key used to sign D-Link's

software by, among other things, publishing the key to a publicly accessible website for months, and for developing and using their own insecure method of storing user credentials in their mobile app, rather than taking advantage of free and open-source software that was widely available and would have been more secure. We note that the contentions of the last charge, 15(c), that the company failed to make use of existing software, will grow in importance as formally verified software becomes more widely available as open-source libraries with feature parity to existing, unverified libraries.

To prove that D-Link should have reasonably known about and fixed these failures, the FTC cites the Open Web Application Security Project's (OWASP) list of critical vulnerabilities, as well as ongoing communications with D-Link about the vulnerabilities. We note that other organizations, such as the National Institute of Standards and Technology (NIST), have regularly published guidance on how to avoid the above security missteps, such as avoiding default passwords, for years [46]. The FTC further charges D-Link with misrepresenting their product by advertising it as being "EASY TO SECURE" and possessing "ADVANCED NETWORK SECURITY" (Paragraph 21). These promotional material might lead consumers to purchase a product that they thought was reasonable secure, but was actually very easy to hack, which could conceivably lead to consumer harm through data theft and loss of privacy.

Although FTC v. D-Link is ongoing as of this paper, and may not become established precedent, we argue that this case provides a good example of how the FTC could use its authority to police deceptive commercial practices to improve the security of devices in cases where a company has made misleading or false claims about the security of their device. The FTC's argument is fairly clear cut if they can prove that a company engaged in misleading advertising or marketing. However, if no incentive is provided for companies to make claims about the security of their devices, then the market may tend toward a complete lack of claims about security in order to avoid liability. If we assume that consumers want to know about the security of the IoT devices they purchase, then FTC enforcement of deceptive security claims without offering benefits to companies that explain their security to customers could

negatively impact the market.

4.2.4 LabMD Inc. v. FTC

There are, however, limits to the FTC's authority to regulate data security practices. In 2013, the FTC alleged that LabMD, a medical lab subject to HIPAA regulations, engaged in unfair data security practices when it installed LimeWire on a lab computer and accidentally shared the folder, which contained the personal information of 9,300 consumers, with the peer-to-peer network. Another company was then able to download this file and bring the matter to LabMD's attention, and although this is the extent of the known consequences, LimeWire was running and potentially sharing files from 2005 until 2008. After learning about this, the FTC issued a complaint and brought the case before an Administrative Law Judge (ALJ). The ALJ found that the FTC had failed to prove that LabMD's data security practices did cause, or would be likely to cause, substantial harm to consumers.

The Commission appealed the ALJ's decision internally, according to the Code of Federal Regulations, and the Commission itself reversed the ALJ's ruling. LabMD then appealed in the 11th circuit. The Court agreed with the FTC's analysis that the installation of file-sharing software, and the subsequent download of personal information, violated consumer expectations of privacy. However, they disagreed with the FTC's cease and desist order, which argued that other LabMD data security practices, taken together as a whole, constituted an unfair act, and required that they completely overhaul their data security practices.

LabMD argues that the order required it to start doing something, rather than to stop doing something unfair, and is thus unenforceable because it requires unspecified acts, rather than prohibits specific acts [41]. This case may suggest limitations to the FTC's authority to regulate data security practices. However, we believe a narrow interpretation of the judgement is most applicable; the FTC's cease-and-desist order appears to require more than the cessation of activity.

4.3 Safe Harbor

These cases demonstrate that the FTC can successfully complain when companies fail to provide minimal levels of security to their customers. However, we do not believe it is reasonable to hold companies accountable for failing to use formally verified software yet. A formally verified software toolchain that is free and open-source, fully functional, and easy to use may someday exist, at which point it may be reasonable to litigate against companies that still fail to make use of it. Until then, however, we think the FTC would better improve the state of IoT device security by establishing a safe harbor from liability for companies that use formal methods to prove that their devices are more secure.

A safe harbor is an agreement between a regulator and a company, regarding some specific law or regulation. The company develops a policy for how to abide by the law or regulation in question, and presents it to the regulator. If the regulator approves, then they hold the company to this policy. So long as the company follows their own policy, they know they will be considered in compliance, but if they fail to follow their policy, then the regulator may consider them in violation of the safe harbor agreement. This reduces uncertainty for both the company and the regulator. The FTC maintains safe harbors for U.S. federal laws like Children's Online Privacy Protection Act (COPPA) and international agreements like the US-EU Privacy Shield, and has worked with companies and solicited comments to help establish practices that would grant companies safe harbor. This approach, offering safe harbor to companies that use state-of-the-art technology like formal methods under self-imposed regulations, could help incentivize innovation and improve consumer protection in the IoT space without imposing heavy regulatory burdens on industry.

4.3.1 COPPA

Children's Online Privacy Protection Act of 1998 requires that online services that are directed to children take certain steps to protect them, such as posting notice, obtaining parental consent, and protecting personal information [13]. COPPA makes

it unlawful for websites directed to children to collect personal information about those children, unless that website satisfies certain regulations. For example, the company must provide notice on the website about what information they collect and how they use collected information. It must also respond to requests from affected parents to know what information has been collected from their child and to cease using their child's information. To avoid unreasonably burdening the private sector, the Act allows for companies to develop and issue their own self-certifications, and charges the FTC with determining whether these self-regulations meet the requirements for Safe Harbor. In order to obtain Safe Harbor, a company that would be regulated under COPPA may submit a set of regulations for itself to the FTC. The FTC will then, after a period of notice and comment, approve or deny the tentative self-regulations within 180 days [12].

The FTC is empowered to issue proposed rules and to modify the existing regulations, and responded to the Act by issuing the Children's Online Privacy Proposed Rule, which proposes that parental consent must be verifiable and provides possible methods of verification, such as the use of a credit card by a parent or a signed consent form received from the parent [11]. It has further revised the rule over the years. For example, in 2011, the proposed updating the definition of personal information to include geolocation data as smartphones became widespread, and browser tracking mechanisms like cookies. It also updated the mechanisms for obtaining parental consent as it became clear that methods that relied on access to a parent's credit card or email account did not provide strong proof that the parent was actually involved, and adding annual audits of Safe Harbor programs [29]. If a company is granted safe harbor by the FTC, then it does not have to worry about legal actions from the FTC as long as it adheres to the mutually agreed upon self-regulations.

However, if a company violates their safe harbor agreement, they expose themselves to prosecution from the FTC. TrustE, a company that sells certifications for various programs like COPPA and the EU Safe Harbor, received safe harbor from COPPA for its certification programs provided it abided by its own, self-issued regulations. In particular, TrustE posted a Program Requirements page on its website,

which stated that companies with the Certified Privacy Seal were required to annually seek recertification. This would ensure that companies under the TrustE program continued to comply with privacy regulations. However, the FTC uncovered more than 1,000 examples of TrustE's failure to recertify a company from 2006 through 2013 [53]. TrustE settled the matter by agreeing to pay a $200,000 fine, to provide additional reports about their COPPA program to the FTC on an ongoing basis for ten years, and to stop misrepresenting their certification process [63].

Another example of the FTC pursuing COPPA violations is the $5.7 million civil penalty that Musical.ly, now known as TikTok, agreed to pay as part of settlement for COPPA violations. Musical.ly did not establish a safe harbor agreement, and decided to settle when the FTC charged it with actual knowledge that they were collecting children's information, and also with failure to obtain parental consent for this information and failure to delete the information upon request [42]. Musical.ly also agreed to comply with COPPA in the future, and although the fine may pale in comparison to its total revenue so far [62], it could have been more costly than establishing a safe harbor would have been.

4.3.2 Privacy Shield

Another safe harbor example is the FTC's program for companies to self-certify themselves as compliant with the EU-U.S. Privacy Shield, an agreement between the United States and European Union about the flow of EU citizen data into the US. Because of privacy and data protections that are afforded to EU citizens under the EU Data Protection Directive and the subsequent General Data Protection Regulation (GDPR), the US and EU negotiated the Privacy Shield framework to help US companies ensure that they handled EU data correctly [51].

Since 2016, the FTC has acted to enforce the Privacy Shield compliance policies that companies submit to the Department of Commerce for safe harbor consideration. US companies may voluntarily join the program by submitting their commitment to adhere to the 23 Privacy Shield Principles, a list of requirements for how companies should handle routine activities involving personal data, such as the requirements

to give individuals notice of data collection and retention, to offer the opportunity to opt-out of data collection, and to provide access to the personal information a company records about an individual. Companies must also commit to re-certifying themselves on a yearly basis [64].

This safe harbor facilitates trade by offering US companies a straightforward path for ensuring they do not run afoul of EU data regulations. Interested companies self-certify themselves by submitting an application. If the company is later found to fail to uphold the practices that they self-certified, then the FTC may prosecute their self-certification statement as a deceptive claim. This benefits consumers, both in the EU and the US, by incentivizing companies to keep their word about data and privacy policies, and companies, by streamlining the compliance process.

The FTC has prosecuted both companies that falsely claim compliance under Privacy Shield and companies that have allowed their certifications to lapse. For example, the FTC found in 2018 that ReadyTech, a training services company, started but did not complete a Privacy Shield application in 2016, and then claimed that it was in the process of certification without making further progress. ReadyTech settled, and agreed to stop falsely advertising its compliance with any data or privacy regulations [8]. Later in 2018 the FTC also found four more companies made false claims about their participation in Privacy Shield: one of which also started but did not finish the application, while the other three failed to re-apply for certification after initially obtaining it in 2016 [28].

4.4 Formal Methods for IoT Devices

We propose that the FTC offer a safe harbor for IoT device manufacturers that use formal methods to provide strong guarantees of correctness and security. Manufacturers can apply for this safe harbor by developing a policy explaining how they will use formal methods to secure their devices, include what software vulnerabilities these formal methods will mitigate and how they will demonstrate to the FTC that they have formally verified the device. After approval, the FTC will consider adherence to

this policy to be fair conduct towards consumers.

We expect that, as the danger of insecure devices becomes more and more clear, what used to be mere recommendations and best practices from the FTC may become federal or international law. For example, the recommendation to avoid default passwords, which is currently an FTC fundamental IoT security practice [9], may become the complete prohibition of default and weak passwords in future legislation, such as the IoT Federal Cybersecurity Improvement Act of 2018 [55].

Similarly, delivering software updates securely is critically important for supporting devices in the long term, because security researchers find more vulnerabilities over time, leaving unpatched devices less and less secure. However, for example, a device running seL4 comes with a mathematically-backed guarantee that it is memory safe. Exploiting user-level programs is still possible, and the seL4 kernel is not "unhackable", but entire classes of vulnerability, like buffer overflows, do not exist in the kernel, which makes it much less likely that an attacker could achieve remote code execution or privilege escalate to root. Thus, a company could submit a plan to the FTC that describes how their use of seL4 mitigates many types of software vulnerabilities, and the FTC could, after approval, grant them safe harbor. A manufacturer of a device running seL4 might want to update the operating system to add new functionality, but there should never be a need to update the operating system to patch a vulnerability.

One existing framework that manufacturers could leverage to explain the material impact that formal methods will have on the security of their IoT devices is the Common Weaknesses Enumeration. By describing what CWEs their approach mitigates, manufacturers connect the abstract technique of formal methods to a well-known taxonomy of real-world problems. When Bedrock 2 was used to implement a device, we can say with high assurance that certain types of CWEs, like buffer overflows, should not exist in the software, and therefore should not be exploitable by attackers. In return, the FTC could provide publicity for companies that self-regulate themselves by using formal methods to prove that their product does not contain buffer overflows.

4.4.1 A Program for Self-Regulation

After a period of review and request for comments, the FTC should propose a safe harbor for IoT device manufacturers that use formal methods to provide strong guarantees of correctness and security. Because the formal methods space is heterogenous and fast moving, the FTC should avoid specific mandates, but instead clearly signal that, if a company publicly commits to selling products that have been formally verified for certain correctness and security properties, then the FTC will consider their behavior fair to consumers, and will not pursue unfairness cases related to data security practices that the formal methods defend against. The FTC could go further, establishing terms of art for this safe harbor, and allowing companies that participate to brand their products with applicable labels denoting high standards for cybersecurity.

The specifics that companies provide are important, and the FTC should release examples of what constitutes an acceptable policy. For example, if a manufacturer releases a policy that states that all software running on its device is checked using Verasco, and that, because of this process, it should be protected against a wide range of CWEs for software written in C, then the FTC should have a good understanding of the company's claims. In this case, providing the specification may not be necessary, but in others, especially if the specification has been developed by the company, then they should produce the specification as part of their application. In all cases, it should be made straightforward for regulators to verify any safe harbor claims made by companies. Further, if some portion of the software has been formally verified, but other portions have not, then omission of this fact, or a mere percentage of total software that is formally verified, will be insufficient. Instead, companies should enumerate all sections of the software, and describe for each what, if any, properties have been formally verified. It is also important that key assumptions of company proofs, like NICTA's assumption that the unverified assembly in seL4 is correct, are explicitly stated.

4.4.2 Incremental Improvements

The LabMD case suggests that courts would be unwilling to consider the mere failure to use formally verified software to be an unfair act. The use of cease-and-desist letters, at the very least, appears to be a weak tool for requiring better behavior from companies, and the burden of either verifying old software, or switching to new software that had already been verified, seems too great to be reasonable.

There are, however, some steps that can be taken immediately that involve formal methods. For example, companies continue to develop software in languages that are memory unsafe, like C and C++, even when they do not need the speed and access to hardware that these languages provide. If the use of outdated software engineering practices raises costs and leads to delays, then some companies may actually benefit from adopting formal methods. To help accelerate the process of adopting formal methods, the FTC could use its existing authorities to create working groups, establish and publish best practices, and interact with the community through notice and comment.

The ideal eventual outcome is that most manufacturers produce devices that are systematically immune to a wide variety of software weaknesses. However, it is unreasonable to expect this transition to occur immediately, considering that existing devices IoT are often exploited because of failures to adhere to well-known best practices like strong passwords and patching vulnerable software. In order to make progress towards systemic formal verification of IoT devices, the FTC should therefore take incremental steps. First, they can offer safe harbor for any companies that are willing to use formal methods. This could involve using existing formally verified software in production devices, or formally verifying sections of the software for valuable properties, such as statically checking software with existing tools. Companies could also formally verify portions of their own software.

For example, since weak passwords are a common means of exploiting IoT devices, manufacturers could use formal methods to prove that the software they use for selecting passwords does not allow weak passwords to be chosen. What consti-

tutes a weak password could be defined using previous literature from NIST or other standards institutes, and then the manufacturer could prove that the software choses initial passwords, and verifies new passwords, according to this specification. They release their specification, and publicly commit to having implemented a formally verified defense against CWE-521: Weak Password Requirements. After a review process, the FTC would approve or deny their safe harbor. If they are approved, the manufacturer could advertise that their device had proven levels of security, and has been granted IoT Formal Methods Safe Harbor. And should someone in the future discover that a weak password, that violated their self-regulated specification, existed on the device, then the FTC would charge them with violating their safe harbor agreement. There is no need for the FTC to prove harm, because the publication of the weak password vulnerability provides proof that the manufacturer engaged in deceptive claims about the device.

If formal methods continue to improve as expected, and if a safe harbor with lower requirements proved to benefit consumers and industry, then the FTC could begin to raise the standards for being granted safe harbor, as they have done over time with the COPPA Safe Harbor, by updating the definition of personal information as technology progresses, or the parental consent mechanisms used when older mechanisms became less effective. For example, if the first safe harbor tended to attract companies that formally verified defenses against a small number of CWEs, then the FTC could increase how many CWEs a safe harbor request would have to protect against in order to be accepted. They could also require more systemic security guarantees.

Chapter 5

Conclusion

We have identified a problem that causes harm to US consumers: insecure connected devices. IoT devices with vulnerable software tend to be in circulation for a long time, but rarely receive software updates. The complexity of software, and the closed-source nature of many products, leads to an information asymmetry–consumers cannot reasonably know how secure their devices are. We discussed one solution to this problem: formal methods, a suite of techniques for proving properties about software. In particular, we think companies could leverage advances in formal methods to prove security properties for IoT devices, reducing harm to the consumer.

We have presented both a technical and a policy contribution. First, we developed a formally verified IoT lightbulb switch in order to show how formal methods have progressed to the point where building real devices with limited resources is possible. This device receives commands over a network and turns a lightbulb on or off in response. We designed a specification for the lightbulb, and proved that our software adhered to this specification. This specification included not only a description of what the lightbulb should do, but also certain things that the software should not do, such as accessing outside the bounds of arrays. We used Bedrock 2, an open-source domain-specific language built in the Coq proof assistant, to implement the IoT lightbulb.

Second, we outlined a way for the FTC to encourage the adoption and use of formal methods to improve the security of IoT devices and reduce harm to consumers.

The IoT lightbulb switch is a proof-of-concept of the viability of creating consumer IoT devices using modern formal methods. However, formal methods have been and will continue to be more expensive and time-consuming than informal, unproven software development. We describe how, because of the FTC's authority and past use of its unfair and deceptive practices mandate and its promulgation of security best practices, especially for IoT devices, it is well-placed to offer safe harbor to manufacturers that adopt formal methods for proving that their software is secure from certain kinds of attacks. This safe harbor could benefit companies by providing differentiating seals or branding in a crowded market and providing incentives for improving their security practices, and benefit consumers by nudging the market towards providing more secure IoT devices.

Appendix A

Figures

```
Definition iot :=
    let p_addr : varname := "p_addr" in
    let bytesWritten : varname := "bytesWritten" in
    let recvEthernet : varname := "recvEthernet" in
    let lightbulb : varname := "lightbulb" in
    let r : varname := "r" in

  ("iot", ((p_addr::nil), (r::nil), bedrock_func_body:(
    unpack! bytesWritten = recvEthernet(p_addr);
    require (bytesWritten ^ constr:(-1)) else { r = (constr:(-1)) };
    unpack! r = lightbulb(p_addr, bytesWritten);

    r = (constr:(0))
  ))).
```

Figure A-1: IoT device implementation

```
Instance spec_of_iot : spec_of "iot" := fun functions =>
  forall p_addr rx_packet R m t,
    (array scalar8 (word.of_Z 1) p_addr rx_packet * R) m ->
    Z.of_nat (List.length rx_packet) = 1520 ->
    WeakestPrecondition.call functions "iot" t m [p_addr]
      (fun t' m' rets => exists v, rets = [v]).
```

Figure A-2: Specification of IoT device

```
Lemma iot_ok : program_logic_goal_for_function! iot.
Proof.
  repeat (straightline || straightline_call || ecancel_assumption).
  { blia. }
  match goal with H : or _ _ |- _ => destruct H end;
  repeat (straightline || straightline_call || straightline_if || ecancel_assumption || eauto).
  match goal with H : _ |- _ => case (H eq_refl) end.
Qed.
```

Figure A-3: Proof that IoT implementation matches IoT specification

```
Definition recvEthernet :=
    let info : varname := "info" in
    let rxunused : varname := "rx_unused" in
    let rx_status : varname := "rx_status" in
    let rx_packet : varname := "rx_packet" in
    let c : varname := "c" in
    let num_bytes : varname := "num_bytes" in
    let num_words : varname := "num_words" in
    let value : varname := "value" in
    let lan9250_readword : varname := "lan9250_readword" in

    let r : varname := "r" in
    ("recvEthernet", ((rx_packet::nil), (r::nil), bedrock_func_body:(
        (* Read RX_FIFO_INF *)
        io! info = lan9250_readword(constr:(0x"7C"));
        rxunused = ((info >> constr:(16)) & ((constr:(1) << constr:(8)) - constr:(1)));
        require (rxunused - constr:(0)) else { r = (constr:(-1)) };

        (* Read Status FIFO Port *)
        io! rx_status = lan9250_readword(constr:(0x"40"));

        (* Pad num_bytes to next word *)
        num_bytes = (rx_status >> constr:(16) & ((constr:(1) << constr:(14)) - constr:(1)));
        num_words = ((num_bytes + constr:(4) - constr:(1)) >> constr:(2));
        num_bytes = (num_words * constr:(4));

        (* num_bytes <= MAX_ETHERNET *)
        require (num_bytes < constr:(1520 + 1)) else { r = (constr:(-1)) };

        c = (constr:(0));
        value = (constr:(0));
        while (c < num_bytes) {
            io! value = lan9250_readword(constr:(0));
            store4(rx_packet + c, value);
            c = (c + constr:(4))
        };
        r = (num_bytes)
    ))).
```

Figure A-4: Receive Ethernet packet function

```
Instance spec_of_recvEthernet : spec_of "recvEthernet" := fun functions =>
  forall p_addr (rx_packet:list byte) R m t,
    (array scalar8 (word.of_Z 1) p_addr rx_packet * R) m ->
    List.length rx_packet = 1520%nat ->
    (* comment on wormhole mentions < 0x400 for lan9250_readword. TODO check against manual *)
    WeakestPrecondition.call functions "recvEthernet" t m [p_addr]
      (fun t' m' rets =>
       (* Success, we wrote 0+ bytes to the buffer *)
       (
         exists bytes_written:word, rets = [bytes_written]
         /\
         exists rx_packet',
                (array scalar8 (word.of_Z 1) p_addr rx_packet' *
                 array scalar8 (word.of_Z 1)
                 (word.add p_addr bytes_written)
                (List.skipn (Z.to_nat
                  (word.unsigned bytes_written)) rx_packet) * R) m'
         /\
         word.unsigned bytes_written <= Z.of_nat (List.length rx_packet')
       )
       \/
       (* Fail, we return -1 and don't write anything *)
       (rets = [word.of_Z (-1)] /\ m' = m)
      ).
```

Figure A-5: Specification of receive Ethernet function

```
Lemma recvEthernet_ok : program_logic_goal_for_function! recvEthernet.
Proof.
  repeat (straightline || straightline_if || eapply interact_nomem || eauto 99 || prove_ext_spec).
  refine (TailRecursion.tailrec
    (HList.polymorphic_list.cons (list byte)
      (HList.polymorphic_list.cons (mem -> Prop) HList.polymorphic_list.nil))
    ["info";"rx_unused";"rx_status";"rx_packet";"c";"num_bytes";"num_words";"value"]
    (fun v scratch R t m info rx_unused rx_status rx_packet c num_bytes_loop num_words value =>
      PrimitivePair.pair.mk (v = word.unsigned c /\
      (array scalar8 (word.of_Z 1) (word.add rx_packet c) scratch * R) m /\
      Z.of_nat (List.length scratch) = word.unsigned (word.sub num_bytes c) /\

      word.unsigned c mod 4 = word.unsigned num_bytes mod 4 /\
      num_bytes_loop = num_bytes
      (fun T M INFO RX_UNUSED RX_STATUS RX_PACKET C NUM_BYTES NUM_WORDS VALUE => exists SCRATCH,
        (array scalar8 (word.of_Z 1) (word.add rx_packet c) SCRATCH * R) M /\
        List.length SCRATCH = List.length scratch /\
        NUM_BYTES = num_bytes)) (* postcondition *)
    (fun n m : Z => m < n <= word.unsigned num_bytes) (* well_founded relation *)
    _ _ _ _ _ _ _);
  (* TODO wrap this into a tactic with the previous refine? *)
  cbn [HList.hlist.foralls HList.tuple.foralls
       HList.hlist.existss HList.tuple.existss
       HList.hlist.apply  HList.tuple.apply
       HList.hlist
       List.repeat Datatypes.length
       HList.polymorphic_list.repeat HList.polymorphic_list.length
       PrimitivePair.pair._1 PrimitivePair.pair._2] in *;
  repeat (straightline || straightline_if || eapply interact_nomem || eauto 99).
  { exact (Z.gt_wf _). }
```

```
{ repeat (refine (conj _ _)); eauto.
  { replace (word.add p_addr c) with p_addr by (subst c; ring).
    rewrite <-
      (List.firstn_skipn (Z.to_nat (word.unsigned (word.sub num_bytes c) ) )  _) in H.
    SeparationLogic.seprewrite_in
      (symmetry! @bytearray_index_merge) H; [|ecancel_assumption].
    rewrite List.length_firstn_inbounds, Znat.Z2Nat.id; trivial.
    { eapply Properties.word.unsigned_range. }
    trans_ltu. eapply Znat.Nat2Z.inj_le. rewrite -> Znat.Z2Nat.id.
    { rewrite -> H0.
      replace (word.sub num_bytes c) with num_bytes by (subst c; ring).
      rewrite word.unsigned_of_Z in H2. change (word.wrap 1521) with (1521) in H2.
      Lia.lia. }
    replace (word.sub num_bytes c) with num_bytes by (subst c; ring).
    eapply Properties.word.unsigned_range. }
  { replace (word.sub num_bytes c) with num_bytes by (subst c; ring).
    rewrite -> List.length_firstn_inbounds.
    { rewrite Znat.Z2Nat.id; trivial; eapply Properties.word.unsigned_range. }
    rewrite H0. trans_ltu.
    change (word.unsigned (word.of_Z 1521)) with (1521) in H2.
    pose proof Properties.word.unsigned_range num_bytes.
    PreOmega.zify.
    rewrite Znat.Z2Nat.id; Lia.lia. }
  { pose proof Properties.word.unsigned_range num_bytes. PreOmega.zify.
    subst c. subst num_bytes. clear.
    { rewrite word.unsigned_mul. unfold word.wrap.
      rewrite (Z.mod_small _ (2^width)).
      { change (word.unsigned (word.of_Z 0) mod 4) with 0.
        change (word.unsigned (word.of_Z 4)) with 4.
        Z.div_mod_to_equations.
        Lia.lia. }
      change (word.unsigned (word.of_Z 4)) with 4. subst num_words.
      rewrite Properties.word.unsigned_sru_nowrap.
      { rewrite Z.shiftr_div_pow2.
        { change (2 ^ word.unsigned (word.of_Z 2)) with 4.
          pose proof Properties.word.unsigned_range
               (word.sub (word.add num_bytes0 (word.of_Z 4)) (word.of_Z 1)).
          Z.div_mod_to_equations.
          (*  0 <= word.unsigned num_words * word.unsigned (word.of_Z 4)
              < 2 ^ width *)
          Lia.lia. }
        { cbv. congruence. }}
      exact eq_refl. }}}
```

```
    (* lan9250_readword *)
    { prove_ext_spec. repeat (straightline || straightline_if).
      (* store *)
      do 2 trans_ltu. rewrite <- (List.firstn_skipn 4 x) in H4.
      SeparationLogic.seprewrite_in (symmetry! @bytearray_index_merge) H4.
      { rewrite List.length_firstn_inbounds.
        { instantiate (1:= word.of_Z 4). eauto. }
        apply Znat.Nat2Z.inj_le. rewrite H5. rewrite word.unsigned_sub.
        unfold word.wrap. rewrite Z.mod_small.
        { revert H6 H3. clear. Z.div_mod_to_equations.
          (* Z.of_nat 4 <= word.unsigned x6 - word.unsigned x5 *)
          Lia.lia. }
        pose proof Properties.word.unsigned_range x6.
        pose proof Properties.word.unsigned_range x5.
        Lia.lia. }
      eapply store_four_of_sep.
      { seprewrite_in @scalar32_of_bytes H8; [..|ecancel_assumption].
        { exact _. }
        { eapply List.length_firstn_inbounds. apply Znat.Nat2Z.inj_le. rewrite H5.
          rewrite word.unsigned_sub. unfold word.wrap. rewrite Z.mod_small.
          { revert H6 H3. clear. Z.div_mod_to_equations.
            (* Z.of_nat 4 <= word.unsigned x6 - word.unsigned x5 *)
            Lia.lia. }
          pose proof Properties.word.unsigned_range x6.
          pose proof Properties.word.unsigned_range x5.
          (* 0 <= word.unsigned x6 - word.unsigned x5 < 2 ^ width *)
          Lia.lia. }}
      do 5 straightline.
```

```
      (* TODO straightline hangs in TailRecursion.enforce *)
      do 8 letexists. split. { repeat straightline. } do 3 letexists.
      repeat split; repeat straightline; repeat split.
      { replace (word.add x4 c)
          with (word.add (word.add x4 x5) (word.of_Z 4)) by (subst c; ring).
        ecancel_assumption. }
      { subst x17. rewrite List.length_skipn. rewrite Znat.Nat2Z.inj_sub.
        { rewrite H5. subst c.
          replace (word.sub x6 (word.add x5 (word.of_Z 4)))
            with (word.sub (word.sub x6 x5) (word.of_Z 4)) by ring.
          rewrite (word.unsigned_sub _ (word.of_Z 4)).
          unfold word.wrap. rewrite Z.mod_small.
          { exact eq_refl. }
          { change (word.unsigned (word.of_Z 4)) with 4.
            pose proof Properties.word.unsigned_range (word.sub x6 x5).
            pose proof Properties.word.unsigned_range x5.
            pose proof Properties.word.unsigned_range x6.
            rewrite word.unsigned_sub. unfold word.wrap. rewrite Z.mod_small.
            { revert H10 H9 H7 H6 H3 H2. clear.
              Z.div_mod_to_equations. Lia.lia. }
            { Z.div_mod_to_equations. Lia.lia. }}}
        apply Znat.Nat2Z.inj_le. rewrite H5. rewrite word.unsigned_sub.
        unfold word.wrap. rewrite Z.mod_small.
        { revert H6 H3. clear. Z.div_mod_to_equations. Lia.lia. }
        pose proof Properties.word.unsigned_range x6.
        pose proof Properties.word.unsigned_range x5.
        Lia.lia. }
      { subst c. rewrite word.unsigned_add. unfold word.wrap.
        rewrite (Z.mod_small _ (2 ^ width)).
        { revert H6. clear. change (word.unsigned (word.of_Z 4)) with 4.
          Z.div_mod_to_equations. Lia.lia. }
        pose proof Properties.word.unsigned_range x5.
        pose proof Properties.word.unsigned_range x6.
        change (word.unsigned (word.of_Z 4)) with 4.
        change (word.unsigned (word.of_Z 1521)) with 1521 in *.
        revert H9 H7 H6 H3 H2. clear.
        PreOmega.zify. Z.div_mod_to_equations. Lia.lia. }
```

```
    { repeat match goal with |- context [?x] => is_var x; subst x end.
      rewrite word.unsigned_add. unfold word.wrap. rewrite Z.mod_small.
      { change (word.unsigned (word.of_Z 4)) with 4. Lia.lia. }
      pose proof Properties.word.unsigned_range x5.
      pose proof Properties.word.unsigned_range x6.
      change (word.unsigned (word.of_Z 4)) with 4.
      change (word.unsigned (word.of_Z 1521)) with 1521 in *.
      revert H9 H7 H6 H3 H2. clear.
      PreOmega.zify. Z.div_mod_to_equations. Lia.lia. }
    { subst v'. subst c.
      rewrite word.unsigned_add. change (word.unsigned (word.of_Z 4)) with 4.
      unfold word.wrap. rewrite (Z.mod_small _ (2 ^ width)).
      { revert H6 H3. clear. Z.div_mod_to_equations. Lia.lia. }
      { pose proof Properties.word.unsigned_range x5.
        pose proof Properties.word.unsigned_range x6.
        change (word.unsigned (word.of_Z 1521)) with 1521 in *.
        PreOmega.zify. Z.div_mod_to_equations. Lia.lia. }}
    { letexists. split.
      { subst x18. subst c.
        repeat match type of H7 with context [?x] => subst x end.
        cbv [scalar32 truncated_scalar littleendian ptsto_bytes.ptsto_bytes] in H7.
        replace (word.add x4 (word.add x5 (word.of_Z 4))) with
                (word.add (word.add x4 x5) (word.of_Z 4)) in H7 by ring.
        SeparationLogic.seprewrite_in (@bytearray_index_merge) H7.
        { exact eq_refl. } { ecancel_assumption. }}
      split.
      { subst SCRATCH. rewrite List.app_length. rewrite H9.
        subst x17. rewrite List.length_skipn.
        change (Datatypes.length (HList.tuple.to_list
          (LittleEndian.split (bytes_per access_size.four)
          (word.unsigned (word.of_Z (word.unsigned v4)))))) with (4%nat).
        assert (4 <= (Datatypes.length x))%nat. 2:Lia.lia.
        PreOmega.zify.
        rewrite H5. rewrite word.unsigned_sub. unfold word.wrap.
        rewrite (Z.mod_small _ (2 ^ width)).
        { revert H6 H3. clear. change (word.unsigned (word.of_Z 4)) with 4.
          Z.div_mod_to_equations. Lia.lia. }
        pose proof Properties.word.unsigned_range x24.
        pose proof Properties.word.unsigned_range x5.
        Lia.lia. }
      { reflexivity. }}}
```

```
    { left. letexists. split. { repeat straightline. exact eq_refl. }
      letexists. split.
      { subst bytes_written. subst c.
        replace (word.add p_addr (word.of_Z 0)) with (p_addr) in H3 by ring.
        replace (word.add p_addr (word.sub x4 (word.of_Z 0)))
          with (word.add p_addr x4) in H3 by ring.
        replace (word.sub x4 (word.of_Z 0)) with (x4) in H3 by ring.
        ecancel_assumption. }
      subst bytes_written. subst rx_packet'. subst c. rewrite H4.
      replace (word.sub x4 (word.of_Z 0)) with (x4) by ring.
      rewrite List.length_firstn_inbounds, Znat.Z2Nat.id; trivial.
      { reflexivity. }
      { pose proof Properties.word.unsigned_range x4. apply H5. }
      { rewrite H0.
        pose proof Properties.word.unsigned_range x4.
        trans_ltu. eapply Znat.Nat2Z.inj_le. rewrite -> Znat.Z2Nat.id.
        { rewrite word.unsigned_of_Z in H2. change (word.wrap 1521)
            with (1521) in H2. Lia.lia. }
        { eapply Properties.word.unsigned_range. }
    }}
    Grab Existential Variables. exact _.
Qed.
```

Figure A-6: Proof of receive Ethernet function

```
Definition lightbulb :=
    let packet : varname := "packet" in
    let len : varname := "len" in
    let ethertype : varname := "ethertype" in
    let protocol : varname := "protocol" in
    let port : varname := "port" in
    let mmio_val : varname := "mmio_val" in
    let command : varname := "command" in
    let MMIOREAD : varname := "MMIOREAD" in
    let MMIOWRITE : varname := "MMIOWRITE" in
    let r : varname := "r" in

  ("lightbulb", ((packet::len::nil), (r::nil), bedrock_func_body:(
    require (constr:(42) < len) else { r = (constr:(-1)) };

    ethertype = ((load1(packet + constr:(12)) << constr:(8)) |
      (load1(packet + constr:(13))));
    require (constr:(1536 - 1) < ethertype) else { r = (constr:(-1)) };

    protocol = (load1(packet + constr:(23)));
    require (protocol == constr:(0x"11")) else { r = (constr:(-1)) };

    command = (load1(packet + constr:(42)));

    io! mmio_val = MMIOREAD(constr:(0x"10012008"));
    output! MMIOWRITE(constr:(0x"10012008"), mmio_val | constr:(1) << constr:(23));

    io! mmio_val = MMIOREAD(constr:(0x"1001200c"));
    output! MMIOWRITE(constr:(0x"1001200c"), mmio_val | command << constr:(23));

    r = (constr:(0))
  ))).
```

Figure A-7: Lightbulb control function implementation

```
Instance spec_of_lightbulb : spec_of "lightbulb" := fun functions =>
  forall p_addr (rx_packet:list byte) (len:word) R m t,
    (array scalar8 (word.of_Z 1) p_addr rx_packet * R) m ->
    word.unsigned len <= Z.of_nat (List.length rx_packet) ->
    WeakestPrecondition.call functions "lightbulb" t m [p_addr; len]
      (fun t' m' rets => exists v, rets = [v] /\ m' = m).
```

Figure A-8: Specification of lightbulb control function

```
Lemma lightbulb_ok : program_logic_goal_for_function! lightbulb.
Proof.
  repeat (eauto || straightline || straightline_if || eapply interact_nomem || prove_ext_spec).

  trans_ltu.
  rewrite word__unsigned_of_Z_nowrap in * by (change (2^width) with (2^32); blia).

  seplog_use_array_load1 H 12.
  seplog_use_array_load1 H 13.
  seplog_use_array_load1 H 23.
  seplog_use_array_load1 H 42.
  repeat (eauto || straightline || straightline_if || eapply interact_nomem || prove_ext_spec).
Qed.
```

Figure A-9: Proof of lightbulb control function

```
uintptr_t recvEthernet(uintptr_t rx_packet) {
  uintptr_t info, rx_unused, rx_status, num_words, value, c, num_bytes, r;
  info = lan9250_readword((uintptr_t)124ULL);
  rx_unused = ((info)>>((uintptr_t)16ULL))&((((uintptr_t)1ULL)<<
    ((uintptr_t)8ULL))-((uintptr_t)1ULL));
  if ((rx_unused)-((uintptr_t)0ULL)) {
    rx_status = lan9250_readword((uintptr_t)64ULL);
    num_bytes = ((rx_status)>>((uintptr_t)16ULL))&((((uintptr_t)1ULL)<<
      ((uintptr_t)14ULL))-((uintptr_t)1ULL));
    num_words = (((num_bytes)+((uintptr_t)4ULL))-((uintptr_t)1ULL))>>
      ((uintptr_t)2ULL);
    num_bytes = (num_words)*((uintptr_t)4ULL);
    if ((num_bytes)<((uintptr_t)1521ULL)) {
      c = (uintptr_t)0ULL;
      value = (uintptr_t)0ULL;
      while ((c)<(num_bytes)) {
        value = lan9250_readword((uintptr_t)0ULL);
        *(uint32_t*)((rx_packet)+(c)) = value;
        c = (c)+((uintptr_t)4ULL);
      }
      r = num_bytes;
    } else {
      r = (uintptr_t)-1ULL;
    }
  } else {
    r = (uintptr_t)-1ULL;
  }
  return r;
}

```

Figure A-10: Emitted C for receive Ethernet function

```
uintptr_t lightbulb(uintptr_t packet, uintptr_t len) {
  uintptr_t ethertype, protocol, mmio_val, command, r;
  if (((uintptr_t)42ULL)<(len)) {
    ethertype = ((*(uint8_t*)((packet)+((uintptr_t)12ULL)))<<
      ((uintptr_t)8ULL))|(*(uint8_t*)((packet)+((uintptr_t)13ULL)));
    if (((uintptr_t)1535ULL)<(ethertype)) {
      protocol = *(uint8_t*)((packet)+((uintptr_t)23ULL));
      if ((protocol)==((uintptr_t)17ULL)) {
        command = *(uint8_t*)((packet)+((uintptr_t)42ULL));
        mmio_val = MMIOREAD((uintptr_t)268509192ULL);
        MMIOWRITE((uintptr_t)268509192ULL, (mmio_val)|(((uintptr_t)1ULL)<<((uintptr_t)23ULL)));
        mmio_val = MMIOREAD((uintptr_t)268509196ULL);
        MMIOWRITE((uintptr_t)268509196ULL, (mmio_val)|((command)<<((uintptr_t)23ULL)));
        r = (uintptr_t)0ULL;
      } else {
        r = (uintptr_t)-1ULL;
      }
    } else {
      r = (uintptr_t)-1ULL;
    }
  } else {
    r = (uintptr_t)-1ULL;
  }
  return r;
}
```

Figure A-11: Emitted C for control lightbulb function

```
uintptr_t iot(uintptr_t p_addr) {
  uintptr_t bytesWritten, r;
  bytesWritten = recvEthernet(p_addr);
  if ((bytesWritten)^((uintptr_t)-1ULL)) {
    r = lightbulb(p_addr, bytesWritten);
    r = (uintptr_t)0ULL;
  } else {
    r = (uintptr_t)-1ULL;
  }
  return r;
}
```

Figure A-12: Emitted C for IoT function

```
Definition MMIOREAD : string := "MMIOREAD".
Definition MMIOWRITE : string := "MMIOWRITE".
Definition lan9250_readword : string := "lan9250_readword".

Instance parameters : parameters :=
  let word := Naive.word32 in
  let byte := Naive.word8 in

  {|
  width := 32;
  syntax := StringNamesSyntax.make BasicCSyntax.StringNames_params;
  mem := SortedListWord.map _ _;
  locals := SortedListString.map _;
  funname_env := SortedListString.map;
  funname_eqb := String.eqb;
  ext_spec t m action args post :=
    if string_dec action lan9250_readword then
    match args with
    | [addr] =>
      ((Ox"0" <= word.unsigned addr < Ox"400"))
      /\ forall v, post m [v]
    | _ => False
    end else
    if string_dec action MMIOREAD then
    match args with
    | [addr] =>
      ((Ox"10012000" <= word.unsigned addr < Ox"10013000") \/
       (Ox"10024000" <= word.unsigned addr < Ox"10025000")  )
      /\ forall v, post m [v]
    | _ => False
    end else
    if string_dec action MMIOWRITE then
    match args with
    | [addr; val] =>
      ((Ox"10012000" <= word.unsigned addr < Ox"10013000") \/
       (Ox"10024000" <= word.unsigned addr < Ox"10025000")  )
      /\ post m []
    |  _ => False
    end else False;
|}.
```

Figure A-13: Specification of the FE310 hardware

Bibliography

[1] *A Brief Overview of the Federal Trade Commission's Investigative and Law Enforcement Authority | Federal Trade Commission*. URL: https://www.ftc.gov/about-ftc/what-we-do/enforcement-authority (visited on 03/20/2019).

[2] *About Red Hat*. URL: https://www.redhat.com/en/about (visited on 04/09/2019).

[3] Andres Erbsen et al. "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises." In: *Proceedings of the IEEE Symposium on Security & Privacy 2019*. (S&P'19).

[4] Jacques-Henri Jourdan et al. "A Formally-Verified C Static Analyzer". In: *42nd ACM symposium on Principles of Programming Languages*. POPL 2015, 2015.

[5] et al. Antonakakis, Manos. "Understanding the mirai botnet." In: *26th USENIX Security Symposium*. USENIX Security 17, 2017.

[6] *Apple's other open secret: the LLVM Compiler*. URL: https://appleinsider.com/articles/08/06/20/apples%7B%5C_%7Dother%7B%5C_%7Dopen%7B%5C_%7Dsecret%7B%5C_%7Dthe%7B%5C_%7Dllvm%7B%5C_%7Dcomplier (visited on 04/09/2019).

[7] *Bitcoin Baron Keeps a Secretive Open Source OS Alive | WIRED*. URL: https://www.wired.com/2014/01/openbsd/ (visited on 04/09/2019).

[8] *California Company Settles FTC Charges Related to Privacy Shield Participation*. URL: https://www.ftc.gov/news-events/press-releases/2018/

07/california-company-settles-ftc-charges-related-privacy-shield (visited on 04/08/2019).

[9] *Careful Connections: Building Security in the Internet of Things*. URL: https://ftc.gov/tips-advice/business-center/guidance/careful-connections-building-security-internet-things (visited on 03/25/2019).

[10] et al. Chen, Haogang. "Using Crash Hoare logic for certifying the FSCQ file system." In: *ACM* Proceeding (2015).

[11] *Children's Online Privacy Proposed Rule Issued by FTC*. URL: https://www.ftc.gov/news-events/press-releases/1999/04/childrens-online-privacy-proposed-rule-issued-ftc (visited on 03/31/2019).

[12] *Children's Online Privacy Protection Act*. 1998.

[13] *Children's Online Privacy Protection Rule (COPPA)*. URL: https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule (visited on 03/25/2019).

[14] Adam. Chlipala. "The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier." In: *ACM Sigplan Notices.* ACM, 2013, Vol. 48. No. 9.

[15] Stephen Chong et al. *Report on the NSF Workshop on Formal Methods for Security*. Tech. rep. USA, 2016. URL: https://dl.acm.org/citation.cfm?id=3040225.

[16] *CompCert - The CompCert C compiler*. URL: http://compcert.inria.fr/compcert-C.html (visited on 07/14/2018).

[17] Judith Crow and Ben L DiVito. "Formalizing space shuttle software requirements". In: (1996).

[18] *CVE-2017-5638 : The Jakarta Multipart parser in Apache Struts 2 2.3.x before 2.3.32 and 2.5.x before 2.5.10.1 has incorrect exception ha*. URL: https://www.cvedetails.com/cve/cve-2017-5638 (visited on 04/07/2019).

[19] *CWE - About CWE*. URL: https://cwe.mitre.org/about/index.html (visited on 04/02/2019).

[20] *DATA PROTECTION Actions Taken by Equifax and Federal Agencies in Response to the 2017 Breach Report to Congressional Requesters United States Government Accountability Office*. Tech. rep. 2018. URL: https://www.gao.gov/assets/700/694158.pdf.

[21] S. Dawson et al. "Known TCP Implementation Problems". In: (). URL: https://tools.ietf.org/html/rfc2525%7B%5C#%7Dsection-2.12.

[22] D. Delahaye. "A Tactic Language for the System Coq." In: *Proceedings of Logic for Programming and Automated Reasoning (LPAR)*. Springer-Verlag, volume 1955 of Lecture Notes in Computer Science.

[23] *Drone security project to go open source • The Register*. URL: https://www.theregister.co.uk/2012/11/19/nicta%7B%5C_%7Ddevelops%7B%5C_%7Ddrone%7B%5C_%7Dprotection/ (visited on 03/20/2019).

[24] *Equifax data breach judge rejects 'blame other hacks' defense - Reuters*. URL: https://www.reuters.com/article/otc-equifax-frankel/equifax-data-breach-judge-rejects-blame-other-hacks-defense-idUSKCN1PN2TJ (visited on 04/07/2019).

[25] *Federal Trade Commission Act Section 5: Unfair or Deceptive Acts or Practices Background*. Tech. rep. URL: https://www.federalreserve.gov/boarddocs/supmanual/cch/ftca.pdf.

[26] Kathleen Fisher. "Using formal methods to enable more secure vehicles: DARPA's HACMS program." In: *ACM SIGPLAN Notices* 49.9 (2014).

[27] Kathleen et al. Fisher. "The HACMS program: using formal methods to eliminate exploitable bugs. Philosophical transactions." In: *Series A, Mathematical, physical, and engineering sciences* vol. 375,2 (2017). DOI: 20150401.

[28] *FTC Reaches Settlements with Four Companies That Falsely Claimed Participation in the EU-U.S. Privacy Shield.* URL: https://www.ftc.gov/news-events/press-releases/2018/09/ftc-reaches-settlements-four-companies-falsely-claimed (visited on 04/08/2019).

[29] *FTC Seeks Comment on Proposed Revisions to Children's Online Privacy Protection Rule.* URL: https://www.ftc.gov/news-events/press-releases/2011/09/ftc-seeks-comment-proposed-revisions-childrens-online-privacy (visited on 03/31/2019).

[30] *FTC v. D-LINK CORPORATION and D-LINK SYSTEMS, INC.* 2017. URL: https://www.ftc.gov/system/files/documents/cases/170105%7B%5C_%7Dd-link%7B%5C_%7Dcomplaint%7B%5C_%7Dand%7B%5C_%7Dexhibits.pdf.

[31] *FTC v. Wyndham Worldwide.* 2015.

[32] *FTC, Privacy & Data Security Update (2018).* 2018. URL: https://www.ftc.gov/system/files/documents/reports/privacy-data-security-update-2018/2018-privacy-data-security-report-508.pdf.

[33] Herman. Geuvers. *Proof assistants: History, ideas and future.* Sadhana, 2009, pp. 3–25.

[34] Georges. Gonthier. "The four colour theorem: Engineering of a formal proof." In: *Asian Symposium on Computer Mathematics.* Springer, (2007).

[35] et al. Gu, Ronghui. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels." In: *12th USENIX Symposium on Operating Systems Design and Implementation.* OSDI 2016, 2016.

[36] *Half a million widely trusted websites vulnerable to Heartbleed bug | Netcraft.* URL: https://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html (visited on 04/09/2019).

[37] C. for D. Health and R. *Alerts and Notices - FDA Statement on Radiation Overexposures in Panama.* URL: https://www.fda.gov/radiation-emitting products/radiationsafety/alertsandnotices/ucm116533.htm.

[38] *IoT Developer Survey 2017.* URL: https://www.slideshare.net/IanSkerret t/iot-developer-survey-2017 (visited on 04/09/2019).

[39] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. "§15.10.4. Run-Time Evaluation of Array Access Expressions." In: *The Java Language Specification, Java SE 8 Edition (1st ed.)* Addison-Wesley Professional., 2014. URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-10.html.

[40] et al. Klein, Gerwin. "seL4: Formal verification of an OS kernel." In: *ACM* Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. (2009).

[41] *LabMD, Inc. v. FTC.* URL: http://media.ca11.uscourts.gov/opinions/ pub/files/201616270.pdf.

[42] *Largest FTC COPPA settlement requires Musical.ly to change its tune.* URL: https://www.ftc.gov/news-events/blogs/business-blog/2019/02/ largest-ftc-coppa-settlement-requires-musically-change-its (visited on 03/31/2019).

[43] Linux Foundation. *Core Infrastructure Initative.* 2014. URL: http://www.linu xfoundation.org/programs/core-infrastructure-initiative (visited on 04/09/2019).

[44] *NAI Code of Conduct 2013 1.* Tech. rep. URL: http://www.aboutads.info/ resource/download/Multi-Site-Data-Principles.pdf..

[45] *NASA Software Safety Guidebook.* (Visited on 04/07/2019).

[46] Nist. *Draft NIST Special Publication (SP) 800-118, Guide to Enterprise Password Management (posted for public comment on.* Tech. rep. 2016. URL: http: //csrc.nist.gov/..

[47] *NSF commits $30 million to expand the frontiers of computing | NSF - National Science Foundation*. URL: https://www.nsf.gov/news/news%7B%5C_%7Dsummary.jsp?preview=y%7B%5C&%7Dcntn%7B%5C_%7Did=137328 (visited on 03/21/2019).

[48] *NVD - CVE-2014-0160*. (Visited on 04/02/2019).

[49] *Our History | Federal Trade Commission*. URL: https://www.ftc.gov/about-ftc/our-history (visited on 03/20/2019).

[50] *Privacy and Security in a Connected World FTC Staff Report*. Tech. rep. 2015. URL: https://www.ftc.gov/system/files/documents/reports/federal-trade-commission-staff-report-november-2013-workshop-entitled-internet-things-privacy/150127iotrpt.pdf.

[51] *Privacy Shield on Shaky Ground: What's Up With EU-U.S. Data Privacy Regulations*. URL: https://www.lawfareblog.com/privacy-shield-shaky-ground-whats-eu-us-data-privacy-regulations (visited on 04/08/2019).

[52] Gavin Rabinowitz. *Israeli hackers show light bulbs can take down the internet*. 2016. URL: http://www.timesofisrael.com/israeli-hackers-show-light-bulbs-can-take-down-the-internet/.

[53] Edith Ramirez et al. *US FTC In the Matter of TRUE ULTIMATE STANDARDS EVERYWHERE, INC., a corporation - d/b/a TRUSTe, Inc.* Tech. rep., DOCKET NO. C. URL: https://www.ftc.gov/system/files/documents/cases/141117trustecmpt.pdf.

[54] *Rice's Theorem*. URL: https://www.cs.virginia.edu/luther/blog/posts/270.html (visited on 04/07/2019).

[55] *Sec. 3 (a)(2)(A)(ii) of the Internet of Things (IoT) Federal Cybersecurity Improvement Act of 2018*. 2018.

[56] *SiFive FE310-G000 Manual v2p3 SiFive FE310-G000 Manual*. Tech. rep. URL: https://sifive.cdn.prismic.io/sifive%7B%5C%%7D2F4d063bf8-3ae6-4db6-9843-ee9076ebadf7%7B%5C_%7Dfe310-g000.pdf.

[57] *SiFive FE310-G000 Preliminary Datasheet v1p5 SiFive FE310-G000 Preliminary Datasheet Proprietary Notice*. Tech. rep. URL: https://sifive.cdn.prismic.io/sifive%7B%5C%7D2Ffeb6f967-ff96-418f-9af4-a7f3b7fd1dfc%7B%5C_%7Dfe310-g000-ds.pdf.

[58] *Suggested Projects | seL4 docs*. URL: https://docs.sel4.systems/SuggestedProjects.html (visited on 03/20/2019).

[59] The Coq development Team. *The Coq proof assistant reference manual: Version 8.9*. Inria, 2019.

[60] *The FTC's Use of Unfairness Authority: Its Rise, Fall, and Resurrection | Federal Trade Commission*. URL: https://www.ftc.gov/public-statements/2003/05/ftcs-use-unfairness-authority-its-rise-fall-and-resurrection (visited on 03/20/2019).

[61] "Third Circuit Finds FTC Has Authority to Regulate Data Security and Company Had Fair Notice of Potential Liability". In: 129 Harv. (2019).

[62] *TikTok Has Made $75 Million So Far from In-App Purchases on the App Store and Google Play*. URL: https://sensortower.com/blog/tiktok-revenue-75-million (visited on 03/31/2019).

[63] *TRUSTe Settles FTC Charges it Deceived Consumers Through Its Privacy Seal Program*. URL: https://www.ftc.gov/news-events/press-releases/2014/11/truste-settles-ftc-charges-it-deceived-consumers-through-its (visited on 04/12/2019).

[64] *U.S. Business | Privacy Shield*. URL: https://www.privacyshield.gov/US-Businesses (visited on 04/08/2019).

[65] *Uptane Securing Software Updates for Automobiles*. URL: https://uptane.github.io/ (visited on 04/09/2019).

[66] et al. Waterman, Andrew. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING and COMPUTER SCIENCES, 2014.

[67] *WebAppSec/Secure Coding Guidelines - MozillaWiki*. URL: https://wiki.mozilla.org/WebAppSec/Secure%7B%5C_%7DCoding%7B%5C_%7DGuidelines (visited on 04/07/2019).

[68] Clark Wood. *Anomaly #9950*. 2019.

[69] Clark Wood. *NotationsCustomEntry: fix == meaning typo*. 2019. URL: https://github.com/mit-plv/bedrock2/commit/56a557ba7a99a7f8973fab2439a31022629fa903.

[70] Clark Wood. *ToCString: do not redeclare arg even if it is ret*. 2019. URL: https://github.com/mit-plv/bedrock2/commit/4ded55e9b176ad85685de51c8a0fb8320fa50a98.