Toward Automated Firmware Analysis in the IoT Era

Grant Hernandez and Farhaan Fowze | University of Florida
Dave (Jing) Tian | Purdue University
Tuba Yavuz, Patrick Traynor, and Kevin R.B. Butler | University of Florida

Firmware for Internet of Things devices can contain malicious code or vulnerabilities, which have already been used in devastating attacks. In this article, we discuss the problems in analyzing firmware for security, offer case studies, and propose challenge tasks to improve firmware analysis.

Born from traditional embedded systems, the devices that comprise the Internet of Things (IoT) bring with them both opportunities and new challenges. Voice-controlled devices such as Google Home and Amazon Echo allow users to easily control their appliances at home. Thermostats and cameras from Nest interface with the cloud to enable remote monitoring and control. Devices such as these are here to stay, with plenty more to come. Between smart homes, intelligent buildings, and industrial and medical environments, the number of IoT devices had already reached 7 billion as of early 2018.[10] These embedded systems are often built using a variety of processors and microcontroller architectures, all programmed with custom firmware. Most of this firmware is closed source and proprietary, leading to a deficit of public scrutiny. Users often have no idea what is inside their firmware and rarely perform firmware updates. If they do, they often find themselves performing manual updates by flashing an opaque blob of binary code onto a device and hoping that everything still works.

Digital Object Identifier 10.1109/MSEC.2019.2926462
Date of publication: 1 August 2019

Unfortunately, attacks targeting or residing in the firmware have shown that "everything still works" is not always the reality. For instance, the BadUSB attack[12] allows attackers to add extra functionality to USB firmware, e.g., adding keyboard functionality to a USB thumb drive to automatically and rapidly inject malicious keystrokes to the host as a way of compromising it. BlueBorne[3] and BleedingBit[4] allow attackers to inject shellcode into target systems due to vulnerabilities within Bluetooth and Bluetooth Low Energy software stacks in controller firmware. Smartphones with Android firmware were found to have backdoors triggered by ATtention (AT) commands,[15] allowing local attackers to bypass Android security mechanisms. The Mirai botnet[2] was responsible for some of the most massive distributed denial-of-service attacks seen to date, building an army of IoT devices by exploiting different zero-days and weak credentials within device firmware. Even automobiles, which represent traditionally closed systems, might be stopped remotely on the highway due to a vulnerability in their telematics controller.[1]

In short, firmware can contain malicious or vulnerable components that are effectively hidden from users

 Copublished by the IEEE Computer and Reliability Societies

and that can be leveraged by bad actors. The ubiquitous connectivity and proliferation of IoT devices in all aspects of society raise the stakes of these deficiencies. As such, firmware analysis for these devices is getting more critical and urgent, yet there is no systematic way to assess the wide variety of firmware that exists across the IoT universe. In this article, we examine the challenges of firmware analysis for the IoT era. We discuss our experience and lessons learned from analyzing USB and Android firmware and look for future guidelines about how to make such analysis broader, more scalable, more accurate, and more automated.

Firmware Analysis

The goals of firmware analysis are similar to those of traditional software analysis. In particular, a goal of security-focused analysis is the ability to automatically discover vulnerabilities or malicious components within firmware. To achieve this objective, researchers apply well-known program analysis techniques to reason about the code and data. We can broadly categorize these techniques as *static*, *dynamic*, or *symbolic*. Figure 1 connects these concepts together in the context of firmware analysis. These three techniques may be applied at either the source code or binary level. With firmware, though, source code is rarely available, leaving binary analysis as the only option. This is more challenging, since analysts lose access to valuable information that can be found only in source code, such as source-level types and code boundaries.

Static analysis relies solely on firmware's code and data. It does not require a working device or emulation environment. As long as the underlying assembly instructions and their relations to data can be understood and extracted, static analysis is possible. A conventional static analysis technique when disassembling binaries is control flow recovery, which creates edges between procedures and helps define the skeleton of a program for later analysis. With a list of functions and their connections to each other, analysts can then answer questions such as "Who are the callers of function X?" and "Which bytes represent code, and which represent data?" Static approaches have a fixed view of a binary image, which has inherent limitations. Unless there is an explicit call or jump in the assembly, some functions and data boundaries may be undiscovered. For example, tracking interprocedural control flow through indirect branches is a path-sensitive problem. Static analysis does not have the program state necessary to track the branch targets accurately. This becomes an issue for languages such as C++ in which classes use dynamically assigned lists of function pointers to call methods. Static analysis has its limitations, and overall it is forced to overapproximate.

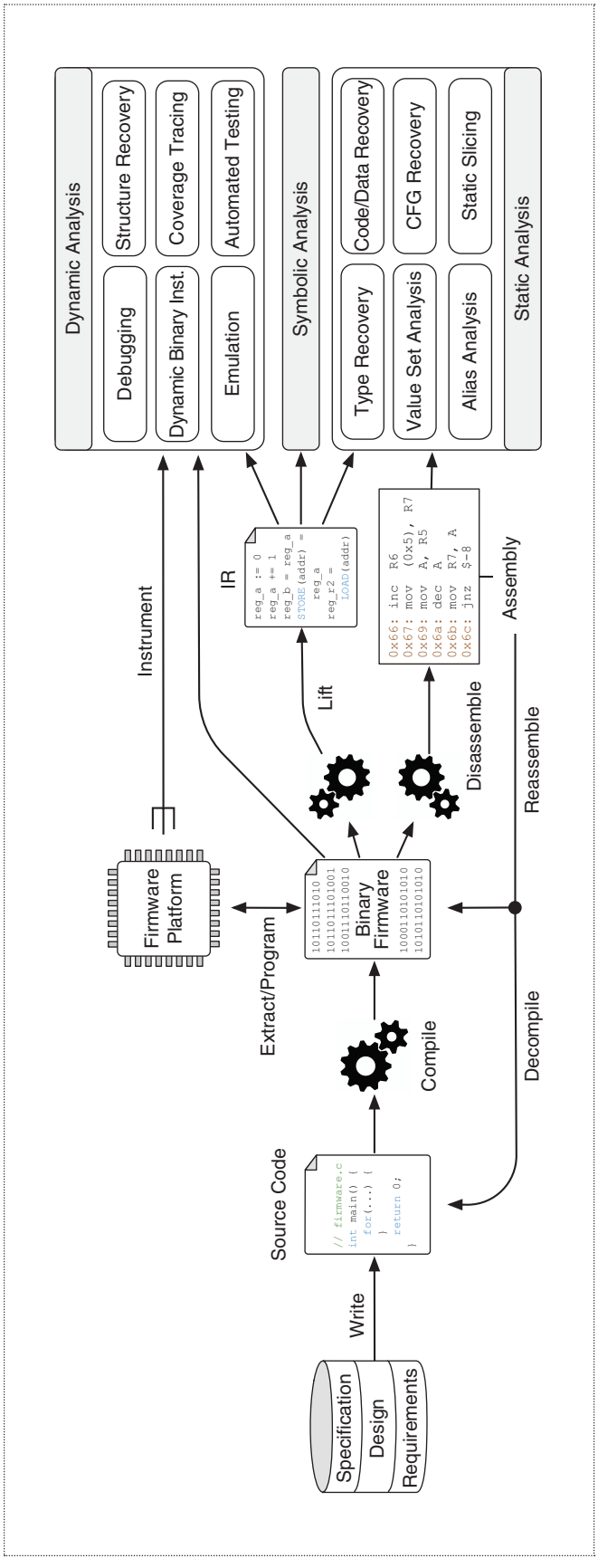

Figure 1. An overview of program analysis concepts in the context of firmware. CFG: control flow graph; Inst.: instrumentation.

In contrast, dynamic analysis is performed using an actual execution environment. Whether this is a physical platform, an emulated platform, or a combination of both depends on the availability of hardware or the completeness of an emulator. For example, to perform hardware-based dynamic analysis, you will likely need many copies of a device and a harness to monitor and control them. Emulation-based analysis executes a programmed model of the target system. QEMU[5] is arguably the most famous emulation platform and implements many architectures and machine types. With an accurate emulation model, analysis scales with the amount of available computing resources instead of device-specific hardware. However, if no emulation model exists, then creating one is a costly endeavor, especially if the target platform combines a significant amount of application-specific hardware. Emulators vastly improve the process of discovering vulnerabilities and debugging flaws, as they are easily instrumented as compared to closed-source software running on locked-down hardware. If a partial model exists but additional modeling would be too costly, we can use real hardware for the missing components, an approach taken by the Avatar[2] framework.[11]

> **With firmware, though, source code is rarely available, leaving binary analysis as the only option.**

Symbolic methods take an approach similar to that of dynamic analysis, with the key difference being that program memory can be symbolic during execution. This means that bytes in memory can be assigned a range of values instead of a single concrete number. Programs are run using a *symbolic execution engine*, which records individual operations on symbolic memory such as adds, subtracts, shifts, and others. The resulting contents of symbolic memory and registers are represented as expressions. When these expressions reach a conditional branch and comparison, a constraint solver is employed to determine whether the branch is taken, not taken, or both. Symbolic execution splits into different paths when the comparison cannot be concretely decided. Effectively, with all memory defined as symbolic, execution would yield program paths for all combinations of input. In practice, any sufficiently complex program will undergo *state explosion*, an exponential creation of new states from existing paths, leading to memory exhaustion. To limit this, we can use targeted symbolic execution and combine traditional dynamic analysis. For instance, if a firmware image takes input from a known memory range, only that memory range can be made symbolic, limiting path explosion.

Case Study: FirmUSB

When the USB protocol was first released in 1996, there were many competing peripheral standards. USB has largely replaced previous interfaces such as serial, parallel, and FireWire ports. However, the ubiquity of USB and its lack of built-in security mechanisms have allowed for simple attack vectors, such as spreading malware from infected devices and the previously discussed BadUSB attack against device firmware. Unfortunately, while host-based protection suites aim to protect USB devices at the file system layer, they do not provide insight into the actual device functionality present within the firmware.

To provide insight into the actual device behavior, we created FirmUSB,[9] a system for analyzing USB-controller firmware and performing behavioral firmware analysis. FirmUSB is designed to allow an understanding of device functionality; rather than focusing on manual reverse engineering, we concentrate on automating our methods by employing a combination of static and symbolic analysis.

Supporting New Microcontroller Architectures

We quickly ran into a major stumbling block: USB controllers are often based on the Intel 8051 microcontroller architecture. In contrast to Intel x86's and ARM's instruction sets, which are the common targets of analysis frameworks, 8051 represents a Harvard architecture design that dates to 1980. Despite being considered as one of the world's most-copied microcontroller designs, and while basic binary analysis has been possible for about a decade, there was no support for deep analysis, including symbolic analysis, on 8051 by any binary analysis framework prior to our research.

To analyze binary firmware compiled for the Intel 8051, we first developed binary lifters for LLVM and VEX intermediate representation (IR). A lifter converts assembly instructions into a platform- and architecture-independent IR. Effectively, this is the first step in reversing the compilation process to recover a high-level understanding of the binary. Most importantly, by targeting LLVM IR and VEX IR, we were able to leverage the FIE[7] (based on the KLEE symbolic analysis framework) and ANGR[14] symbolic execution frameworks.

During the lifting process, we ran into many embedded-system-specific issues. First, 8051 does not have any virtual memory hierarchy and operates directly on processor registers and memory-mapped input–output (I/O). Capturing these low-level semantics in the target IRs was difficult, given their inherent assumptions. As an example, 8051 has multiple memory regions that are accessed by specific instructions. When lifting to generic IR memory operations, we lost these semantics during symbolic execution of the IR. To address this, we needed to hook all loads and stores and adjust the operations before continuing. Processing the IR alone was not sufficient to correctly execute 8051 for either LLVM or VEX.

Another limitation was the lack of bitwise semantics. For example, a real processor will perform bitwise operations on a system control register affecting only a specific bit but not an adjacent one, since every individual bit can have implications for the processor's state. However, as they operate at the granularity of byte-level instructions, neither LLVM nor VEX succinctly captures bitwise operations, which are a common occurrence in firmware images. Therefore, we had to access and manipulate individual bytes to extract and manipulate the appropriate bits. As a result, the symbolic execution engine needed to execute many more IR instructions, slowing down execution and lowering the precision of memory operations. This subtle difference at the IR level could affect how an environment model is recreated at the symbolic execution level.

In general, a new microcontroller architecture in a symbolic analysis framework must precisely replicate the firmware's environment model, such as its interrupt handling. For FirmUSB, FIE supported interrupts for the MSP430 architecture, but we needed to adapt these handlers to the 8051 architecture. However, ANGR did not support interrupts, since it had been designed as a binary analysis framework for user-level programs. As such, we extended ANGR with interrupt handling, scheduling, and other specifics of Intel 8051, such as special or alias registers. The limitations we experienced when developing FirmUSB demonstrate that there is a need for firmware specific IRs and symbolic execution engines.

Checking Behavioral Properties

One of the mysteries of closed-source, end-user firmware is its actual functionality. In the IoT domain, understanding a device's behavior is of utmost importance to assess the overall system's security and privacy. Consumers often wonder what they can expect from their smart devices: e.g., "Why is my smart camera blinking at this moment?" or "Is my smart speaker leaking my private conversations?" In an ideal world, device firmware would be vetted rigorously before deployment, yet in practice, vulnerable firmware is frequently found in the wild. This leaves end users and security researchers with the responsibility of vetting firmware themselves.

For FirmUSB, we wanted to be able to ask "What are the functionalities of this USB firmware?" and "Does this firmware act consistently with the functionalities it declares?" and be able to find the answers automatically using program analysis. To do this, we used symbolic execution to check the reachability of code locations relevant to these questions. For the functionality question, we checked whether the firmware reaches a point in the disassembled code that corresponds to specific functionality, such as USB mass storage or human interface device (HID) capabilities. To identify candidate code locations relevant to this behavioral query, we used static data-flow analysis and tracked memory locations that stored the USB-specific objects. To answer the consistency question, we verified that, for example, HID firmware forwarded keyboard scan codes that were obtained from an external I/O port. If it did not, instead of replaying hard-coded keystrokes, we were able to detect this inconsistency.

Symbolic execution is a powerful tool that can be employed to answer queries such as these. In our research, our queries were specific to the USB domain, but similar questions could be answered about different firmware. For example, using similar domain-informed techniques to analyze Bluetooth device firmware—once the underlying architecture has been lifted to an IR and query algorithms have been developed with Bluetooth operation in mind—could allow us to similarly reason about the behavior of these devices. We could even imagine having an agreed upon set of domain-specific queries to apply to firmware, to perform a sort of compliance testing, similar to Underwriters Laboratories testing.

Domain-Informed Analysis

Since our goal was to behaviorally analyze USB firmware, we formulated our questions in the context of the USB protocol. A USB device communicates with the host by responding to requests. Among those, the Get_Descriptor and Get_Configuration requests tell the operating system about the device's functionality. All USB devices must support responding to these requests, but for specific device classes, other responses also exist. For example, the HID device class is sufficiently complex that it needs its own "report" descriptor with additional information for the host machine. FirmUSB leverages USB constants in all of these descriptor types and searches the firmware for references to them. Using this domain knowledge as a starting point simplifies all of the following analysis, as it gives the

symbolic execution engine known interesting locations to investigate.

Finally, during symbolic execution, we were able to apply USB-specific constraints to focus on specific code paths. As an example, USB speed-change request/response pairs are not as relevant to understanding device functionality as the Get_Descriptor requests made by the host. Applying these constraints is known as *preconditioned symbolic execution*, and it allowed us to achieve a 7× speedup during the analysis phase—from almost 7 min to perform reachability analysis with fully symbolic memory to fewer than 56 s, and to 7.7 s with partially symbolic memory. Such a speedup makes analysis of real-world firmware far more tractable. Without incorporating knowledge of the USB protocol, FirmUSB not only would be slower but also would not have a starting point for more in-depth analysis. With such knowledge, we can ask questions relating to device functionality, determine input sources and sinks, and assure consistency of behavior.

The takeaway for firmware analysis, in general, is that incorporating domain knowledge is beneficial for a variety of reasons, but unfortunately, protocols are usually under- or informally specified. The lack of a formal, or even a machine-parseable, model of many protocols found in firmware hinders the application of domain knowledge beyond a high-level understanding of the firmware type.

Case Study: Android AT Commands

The FirmUSB case study reflects a more traditional type of firmware, which is a single binary handling interrupts and running on the microcontroller unit directly without an operating system. Android firmware represents another type of firmware that is more complex. Android firmware images contain an operating system, e.g., Linux, and a corresponding file system (e.g., rootfs) that provides all necessary user-space tools and daemons. In this case study, we discuss the challenges of collecting and extracting Android firmware images across multiple vendors. We sought to investigate the scope of "AT commands" being used in Android devices through static and dynamic analysis techniques.

AT commands were designed in the early 1980s to control modems. Not only are they still being used by modern smartphones to support telephony-related functions, but we discovered that they can also provide a conduit for accessing powerful functionality in mobile devices through access to privileged operations. We systematically retrieved and extracted 3,500 AT commands from more than 2,000 Android firmware images by building an Android firmware-image process pipeline as shown in Figure 2. The vast majority of these commands have never been publicly documented. In short, we found AT commands that can flash device firmware, bypass Android security mechanisms, exfiltrate sensitive device information, perform screen unlocks, and inject touch events.

Firmware Collection

We designed the firmware collection and extraction process as a three-stage process, as illustrated in Figure 2. These stages included firmware-image collection, firmware-image analysis to extract AT commands, and construction of an AT command database (DB). The first challenge came from collecting different Android firmware images from a wide range of smartphone vendors. Vendors such as Google and ASUS allow firmware downloads from their official websites, while other vendors hide download links behind complicated device-query interfaces or do not provide firmware downloading at all. Third-party websites (e.g., AndroidMTK.com) collect different vendor firmware images but often deploy multiple redirections before revealing the final download URL. We managed to crawl more than 2,000 Android firmware images across 11 different vendors, although the whole firmware collection stage took more than a month. An insight we gained from this exercise was the need for easy and standardized access to firmware images, which is as important to end users as it is to analysts. We provide our DB of extracted firmware as a service to the community through our website at https://atcommands.org.

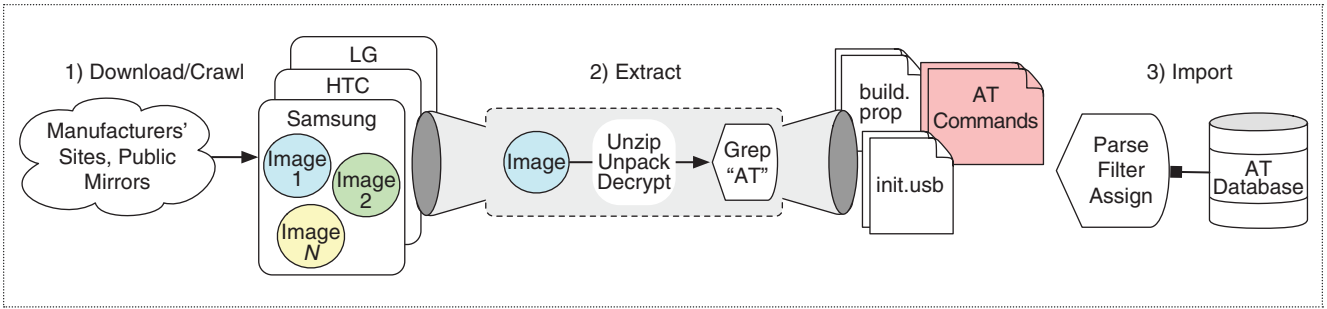

Figure 2. A graphical depiction of the smartphone firmware-image processing pipeline.

Tool Chain and Domain Knowledge

Since each Android firmware image contains a complete system file system with native daemons and preinstalled apps, we needed to unpack the image before we could mount the file system for further stepping through. The next challenge was finding the right tool to unpack the firmware image. We ended up with a different tool for almost every vendor per every Android version, due to the fragmentation of Android platforms and vendor customization. Once inside the system file system, we needed to go through each file looking for potential AT commands. For native binaries, we could directly extract text symbols, while for Android apps, we decompiled the code and extracted AT commands from the source files. We also used our domain knowledge of AT command formats to develop regular expressions to automatically extract commands while suppressing false positives, Additionally, we developed a heuristic algorithm to further filter out false positives. The end result is more than 3,500 unique AT commands extracted from these firmware images. One takeaway from this exercise is that standardizing firmware images and creating more modular frameworks to support vendor customization can facilitate analysis.

Static and Dynamic Analyses

Once we had extracted a corpus of AT commands, we needed to understand their functionality. In some cases, such as for commands extracted from apps, we could examine source code and comments from decompiled binary code to understand usage. In other cases where only binaries were available, we manually reversed the code and examined the disassembled output to find function boundaries and to reconstruct command usage.

However, these static approaches do not in themselves indicate whether the code examined lives in the firmware image. We thus needed to use real Android devices to test and confirm the functionality of these AT commands. We built a unified phone-testing framework using USB connections to achieve this. Because this was dynamic analysis in real-world environments, commands that ended up crashing the device required manual intervention, and ambiguous results required further examination of source or binary code to confirm the observed functionality. As discussed previously, through this combination of static and dynamic analysis, we were able to discover a broad range of functionality, including previously undisclosed vulnerabilities that we disclosed to smartphone vendors. There are very likely more interfaces to investigate and further vulnerabilities to be mitigated. We believe online documentation with technical details and lessons learned from the firmware analysis can aid the community with future analysis. Additionally, a testbed comprising numerous makes and models of smartphones, such as the "Droid Army" developed by Drake,[8] accessible to mobile security researchers, would tremendously benefit the community.

Future Directions for Improving IoT-Firmware Analysis

Historically, firmware was considered an intermediary between hardware and upper-layer software, such as the operating system. However, as devices have replaced traditional interfaces, reduced their size, and grown more complex, the line between software and firmware has begun to blur. As past work by us and others has demonstrated, there is a wide range of firmware for devices used within the IoT context. This firmware varies considerably in its complexity, functionality, and target-device architecture, and firmware can look substantially different even between devices that appear functionally similar.

When analyzing firmware and the underlying platforms on which it runs, we quickly encounter difficulties in applying known techniques to reason about them automatically. Dynamic, static, and even symbolic analysis methods can be generally used. Ultimately, however, they must be applied to real platforms. We need to rework our analysis toolbox for the type of challenges and problems present on primarily firmware-based systems. Unfortunately, this also means that, currently, there is no prescriptive, "one-size-fits-all" approach to firmware analysis, given the disparity of architectures and capabilities. However, there are advances in technology to ensure that analysis can be performed more easily and automatically in the future.

For example, symbolic execution is one of the major program analysis techniques used in analysis frameworks such as, but not limited to, Firmalice,[13] ANGR,[14] S^2E,[6] and FIE.[7] As discussed, this is because of its ability to combine concrete and symbolic input values and achieve relatively high precision and measurable coverage. However, we surveyed processors and microcontrollers used in IoT settings, and our results, summarized in Table 1, show that many microcontroller architectures lack IRs that can allow their easy inclusion into symbolic-execution frameworks. While many architectures support lifting to QEMU's Tiny Code Generator (TGR) IR, and some support Binary Ninja's BIL representation, comparatively few can be lifted to LLVM or VEX, which are necessary for frameworks such as FIE and ANGR, respectively. Our experience lifting 8051 to LLVM and VEX demonstrated to us the painstaking and time-intensive efforts necessary to build lifters into new IRs. Thus, one of the challenges of firmware analysis is the inability of

Table 1. A partial survey of popular architectures and available architecture lifters and corresponding symbolic execution engines.

Architecture	Manufacturer	Lifter	Symbolic execution	Supported IRs
AVR/AVR32	Atmel	◐	○	RREIL
PIC	Microchip	◐	○	TCG
Sparc	Sun	●	○	VEX, TCG
RISC-V	RISC-V	●	○	TCG
Blackfin	Analog Devices	●	○	TCG
s390x	IBM	●	○	VEX, TCG
SuperH	Hitachi	●	○	TCG
CRIS	Axis	●	○	TCG
i960	Intel	●	○	TCG
MSP430	TI	●	●	LLVM
MIPS	MIPS Technologies	●	●	VEX, TCG
Power/PPC	IBM	●	●	VEX, TCG
8051	Intel	●	●	VEX, LLVM, LLIL
ARM/ARM64	ARM	●	●	TCG, VEX, BIL
x86/x86_64	Intel	●	●	TCG, LLVM, VEX, BIL
AVR32	Atmel	○	○	None
ARM64	ARM	●	●	TCG, VEX
x86_64	Intel	●	●	TCG, LLVM, VEX, BIL

NOTE: Many architectures have lifters for QEMU's TCG IR, but lack support for symbolic execution. The "Lifter" and "Symbolic Execution" columns use ○ for no support, ◐ for partial or unofficial support, and ● for full support. For example, there is a third-party lifter from AVR to RREIL but no symbolic execution engine, while SuperH has a lifter but no symbolic execution engine. RISC: reduced instruction set computing; CRIS: code reduced instruction set; RREIL: Relational Reverse Engineering Intermediate Language; LLIL: Low-Level Intermediate Language.

existing analysis frameworks to cover the wide variety of microcontroller architectures used in the wild.

We propose that as a community, we must develop methods for automating the lifting process to better facilitate analysis. This can be done by using the QEMU lifters as a starting point and learning how mappings are made from CPU definitions and compilers. Equally important is finding ways for existing analysis tools to work better together. There are a number of important binary-analysis and symbolic-execution tools being used for analysis, but they support only a single IR and often have different strengths. For example, ANGR shines as a means for recovering control flow graphs from binaries, while KLEE has a rich set of tools designed for sophisticated symbolic execution. We need to develop ways to better interface these tools with each other or develop ways of translating IRs so that they can reap the benefits of multiple analysis platforms. Approaches such as the Avatar platform,[2,11] which allows for the orchestration of different binary analysis tools, could be a promising direction for continued research in this area.

It is not just the tooling that is challenging; environment models are complicated, as are the communications used. We believe solutions that leverage domain knowledge of underlying architectures, environments, programming models, and protocols will be more effective in dealing with the challenges.

As an example, particularly in the IoT context, many devices need to support some communication protocol to be useful. Firmware implements protocol-related functionality by recognizing specific types of messages defined by the protocol and then responding to these messages. The specification of a typical communication protocol is an informal yet structured and lengthy document. Microcontroller vendors may provide some example firmware in their software development kits to help developers understand how to program when they want to support a specific protocol. Although this type of program-based documentation supports firmware development, we cannot easily incorporate it into firmware analysis. As such, one of the challenges of firmware analysis is the lack of formal representations for domain knowledge. An important aspect of using domain knowledge is to extract it from firmware and/or from the associated artifacts in an automatic or semiautomatic way. Another aspect is to represent it in a formal way so that it can be incorporated into automated analysis and decision making. Techniques such as machine learning to generalize extracted knowledge and associating it with semantics can also help with this domain-knowledge representation.

We also reiterate another challenge in firmware analysis, the lack of standardization of firmware images and modularization of vendor customizations. We found this lack of standardization to be an impediment with our analysis of smartphone-firmware images. Standardizing formats around base images and including customization features as modular additions could facilitate rapid and automated analysis. This is true not only for resource-rich firmware environments, but also in microcontroller architectures where vendors may add customizations to well-known architecture designs such as the 8051/MCS-51 microcontroller family. Such standardized architectures can aid in developing tooling that works across products and would be beneficial even to companies developing these architectures by assuring their consistency and inseparability

while reducing the costs of developing and maintaining in-house tooling.

Finally, while analysis tools have been more fully developed for ARM and x86 processor architectures and the sophisticated firmware that runs on these platforms, considering these platforms holistically demonstrates the need to better understand how different firmware programs interoperate. Many sophisticated devices such as smartphones and other well-resourced IoT devices have not only application processors such as ARM processors that might be running an operating system but also sensors providing other functionality that are themselves built on separate microcontroller architectures (such as 8051) within the system on chip; frequently, vendors are unaware of the full functionality of these controllers and place guard chips in front to constrain their behavior. Thus, a comprehensive approach to firmware analysis on these devices must consider both the high-level application processor firmware and the low-level firmware found on these embedded microcontrollers.

To conclude, we believe that large recent advances in firmware analysis frameworks, combined with the vast number and diversity of embedded devices being deployed in the IoT era, offer unique challenges. We are also, however, at the cusp of being able to provide guarantees to ensure that computing on IoT devices can be made safer and more secure. The research challenges are not trivial, but they offer rich rewards, and we encourage the community to embrace and review these issues. ■

Acknowledgments

This work is supported in part by the U.S. National Science Foundation under grants CNS-1540217, CNS-1526718, CNS-1564140, CNS-1815883, and CNS-1617474 as well as the Semiconductor Research Corporation.

References

1. A. Greenberg, "Hackers remotely kill a Jeep on the highway—with me in it," *Wired*, July 21, 2015. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
2. M. Antonakakis et al., "Understanding the Mirai botnet," in *Proc. 26th USENIX Security Symp. (USENIX Security 17)*, 2017, pp. 1092–1110.
3. Armis, Inc., "BlueBorne," 2017. [Online]. Available: https://www.armis.com/blueborne/
4. Armis, Inc., "Bleeding Bit," 2018. [Online]. Available: https://armis.com/bleedingbit/
5. F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Technical Conf., FREENIX Track*, 2005, p. 46.
6. V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, Mar. 2011.
7. D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proc. 22nd USENIX Security Symp. (USENIX Security 13)*, 2013, pp. 463–478.
8. J. J. Drake, "Researching Android Device Security with the Help of a Droid Army," Black Hat, Aug. 6, 2014. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Drake-Researching-Android-Device-Security-With-The-Help-Of-A-Droid-Army.pdf
9. G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler, "FirmUSB: Vetting USB device firmware using domain informed symbolic execution," in *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS '17)*, Dallas, TX, 2017, pp. 2245–2262.
10. K. L. Lueth, "State of the IoT 2018: Number of IoT devices now at 7B—Market accelerating," IoT Analytics, Aug. 8, 2018. https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/
11. M. Muench, A. Francillon, and D. Balzarotti, "Avatar^2: A multi-target orchestration platform," in *Proc. Workshop on Binary Analysis Research*, San Diego, CA, 2018, pp. 1–11.
12. K. Nohl and J. Lell, "BadUSB–On accessories that turn evil," Black Hat, Aug. 2014. [Online]. Available: https://www.blackhat.com/us-14/video/badusb-on-accessories-that-turn-evil.html
13. Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice—Automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proc. Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, Feb. 2015, pp 1–15.
14. Y. Shoshitaishvili et al., "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Security and Privacy*, San Jose, CA, 2016, pp. 138–157.
15. D. J. Tian et al., "ATtention spanned: Comprehensive vulnerability analysis of AT commands within the Android ecosystem," in *Proc. 27th USENIX Security Symp. (USENIX Security 18)*, 2018, pp. 273–290.

Grant Hernandez is a Ph.D. candidate in the Department of Computer and Information Science and Engineering at the University of Florida. His research focuses on automated embedded binary firmware analysis to discover vulnerabilities at scale. Contact him at grant.hernandez@ufl.edu.

Farhaan Fowze is a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of Florida. He received a B.Sc. from Bangladesh University of Engineering and Technology in

2012. His research interests include model extraction, binary analysis, and program analysis. Contact him at farhaan104@ufl.edu.

Dave (Jing) Tian is an assistant professor in the Department of Computer Science at Purdue University. His research involves systems infrastructure, security, and storage. He received a Ph.D. from the Department of Computer and Information Science and Engineering at the University of Florida. Contact him at daveti@purdue.edu.

Tuba Yavuz is an assistant professor with the Department of Electrical and Computer Engineering at the University of Florida. Her research is in the intersection of formal methods, software engineering, and systems security. She received a Ph.D. in computer science from the University of California, Santa Barbara, in 2004. She is a Member of the IEEE and of ACM. Contact her at tuba@ece.ufl.edu.

Patrick Traynor is a professor and the John and Mary Lou Dasburg Preeminent Chair in Engineering in the Department of Computer and Information Science and Engineering at the University of Florida. He received a Ph.D. from The Pennsylvania State University in 2008 and is a Senior Member of the IEEE and ACM. Contact him at traynor@ufl.edu.

Kevin R.B. Butler is an Arnold and Lisa Goldberg Rising Star Associate Professor of Computer Science at the University of Florida and associate director of the Florida Institute for Cybersecurity Research. His research focuses on establishing the trustworthiness of computer systems and embedded devices. He received a Ph.D. in computer science and engineering from The Pennsylvania State University in 2010. He is a Senior Member of the IEEE and ACM. Contact him at butler@ufl.edu.

Digital Object Identifier 10.1109/MSEC.2019.2933709