Lock Lock Who's There: Reverse Engineering a Smart Lock

Aaron Barber, Garrison Bellack, Sarah Paris, Ben Burgess, Travis Finkenauer, J. Alex Halderman
University of Michigan
{asbarber, gbellack, sarparis, baburges, tmfink, jhalderm}@umich.edu

Abstract

The rise of the Internet of Things has prompted lock manufacturers to integrate connectivity into their products. These new smart locks promise to offer many convenience and security features, such as time limited keys, keyless entry, and audit logs. However, locks are commonly the primary line of defense for most homes and businesses, and the added complexity these systems introduce has the potential to introduce many serious vulnerabilities.

In this paper, we select a very popular smart lock and perform an in depth security audit of it's firmware, authentication protocol, and mobile app. We present several practical attacks that can be performed against this device, including a backdoor that is able to provide on-demand surreptitious entry and a malicious mobile app that will steal the user's credentials. Finally, we show how a user's credentials can be stolen from the mobile app.

1 Introduction

Locks are ubiquitous in modern society, acting as a primary defense against opportunistic and dedicated attackers alike. Mechanical locks are an aging product, however, and lock manufacturers are being pressured to maintain relevance by producing new connected locks that merge an embedded system, servos, and a traditional mechanical lock. These features promise consumers enticing new features such as keyless entry and time limited keys. Additionally, smart locks are able to provide businesses new security features such as audit logs and theoretically perfect key control.

By introducing significant additional complexity to a traditionally very simple device, the attack surface of these devices is greatly increased. We can look to previous embedded system exploits that target systems such as printers [8], server lights-out management controllers [6], and baby monitors [18] to see how the added complexity can lead to vulnerabilities in these systems. Any vulnerability in a smart lock is potentially catastrophic because locks act as the front line of defense for most installations. Additionally, consumers largely overlook the potential for vulnerabilities in the underlying embedded system, rather just considering the new features or the security of the physical lock.

For this paper we select the Kevo smart lock, a very common smart lock developed by one of the most popular residential lock supplier in the United States: Kwikset. We reverse engineer the firmware to allow a vulnerability to be inserted which will allow anyone to gain surreptitious access to lock by simply having one time access to the lock. Additionally, we investigate the security of the underlying protocol by reverse engineering the mobile application and reconstructing the protocol. Finally, we show how a user's credentials can be stolen from the mobile app and how a denial of service attack can be performed to render the lock inoperable.

2 Related Work

Although smart locks are relatively new, research has shown that they are very vulnerable to exploits. Smart car locks have been found to have simple replay vulnerabilities [10]. The August smart lock mobile application contained an exploit that would allow an adversary to grant themselves guest access to the lock [12]. The most relevant work to ours showed how a Kwikset 910 TRL smart lock could be remotely unlocked and controlled. However, the lock's firmware itself was not exploited but rather the Z-Wave protocol and the Samsung SmartThings app that was trusted by the lock was exploited. There has also been research into the general vulnerabilities of the BLE networking protocol [15], which the Kwikset Kevo lock uses. In this paper, we present a firmware exploit of the Kwikset Kevo platform that is not reliant on an external framework, communication protocol, or mobile app being vulnerable.

Existing research on firmware vulnerabilities can

also be applied to smart locks. The Firmalice framework detects authentication bypass vulnerabilities in firmware [17]. The authors even hypothesize that the Firmalice framework could potentially be applied to smart locks.

The security of traditional locks has also been extensively research with numerous attack vectors being found. Many of these attacks such as the teleduplication [5] combined with 3D printing attacks [7] will provide repeat surreptitious access to traditional mechanical locks. The Kevo lock is a hybrid lock with a Kwikset SmartKey mechanical lock inside it which is vulnerable to aforementioned attacks along with many others. However, by analyzing the firmware of this lock and exploiting it, we hope to provide insight into common pitfalls and mistakes made for when the next generation of smart locks are developed and the traditional locks are removed from them.

3 Threat Model

We propose a threat model where an attacker would have either one time or limited access to a space and would like to elevate their access to an essentially unlimited access level. The Kevo electronic key distribution model facilitates this threat model since a babysitter, for example, could be granted a temporary electronic key that only works between certain times of the day. Additionally, other classical physical attacks such as under the door tools lock picking [1], and bump keys can be used to gain one time access to the lock. These attacks are not optimal for reoccurring surreptitious entry since they either require extended amounts of time on site or leave evidence with repeated use such is the case with bump keys but they can easily provide one time access.

We assume that during these periods an adversary would have full access to the rear panel of the lock and would be able to disassemble it. The disassembly process can be performed relatively quickly and the door can remain closed so it is reasonable that an attacker could perform this covertly.

4 Kevo Lock System

The Kevo smart lock provides a traditional mechanical interface where a physical key can be inserted along with a bluetooth low energy (BLE) interface which can be used with a mobile app or physical key fob. The lock has a capacitive faceplate which looks for a user's touch, indicating they would like to enter. Once a user touches the faceplate the lock will search for a paired key fob or mobile device running the Kevo app on the user and will unlock if it detects one. The user is able to pair new devices by simply pressing a programming button on the inside of the lock.

4.1 Hardware

The lock contains three PCBs: one is mainly responsible for BLE authentication, which will herein be called the BLE breakout; one is responsible for user configuration and actually locking/unlocking the door, which will be herein called the main PCB; and one is responsible for collecting user input and providing feedback to the user on the insecure side of the door, which will herein be called the external PCB. The main PCB can be seen in Figure 1 and the BLE PCB can be seen in Figure 2. A high level block diagram of how these PCBs are linked can be seen in Figure 3.

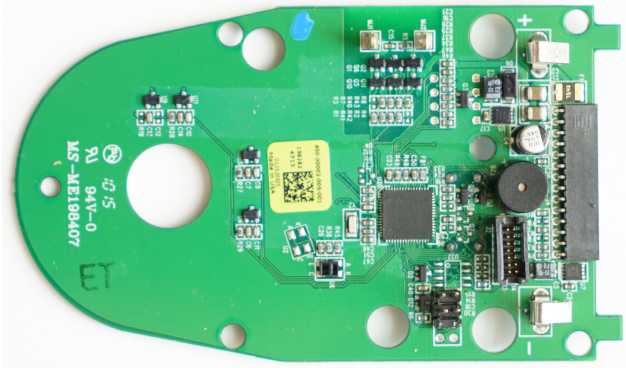

Figure 1: *Picture of the main PCB from the Kevo.*

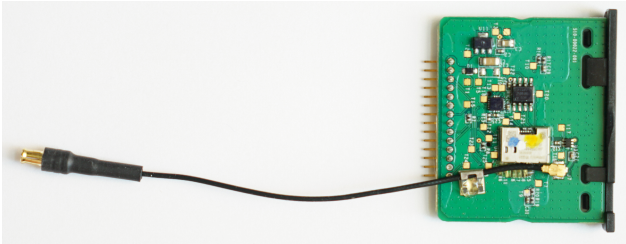

Figure 2: *Picture of the breakout from PCB in the Kevo*

The external PCB contains a NXP9539R which communicates with the main PCB over an I2C bus. The sole function of the external PCB is to sense user touch on the face plate and to provide the user with feedback using LEDs on the faceplate. No authentication functions are performed on this external PCB since it is accessible by anyone on the outside of the door by tearing off the faceplate.

The BLE breakout contains a Panasonic PAN1721 system on a chip (SOC) which manages the bluetooth communications. The SOC then transmits the received eKeys

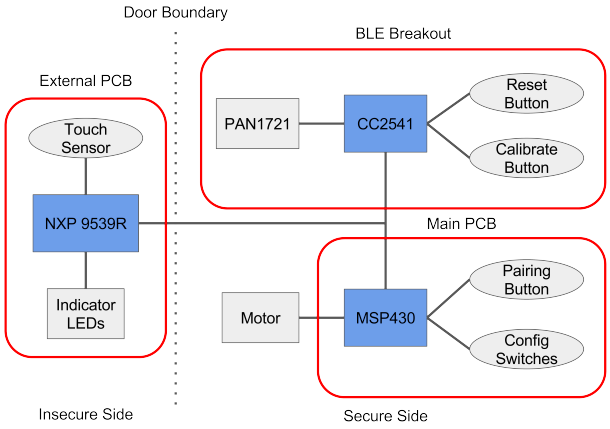

Figure 3: *Hardware system design. Gray ovals represent input sensors; gray rectangles represent output; blue rectangles represent microprocessors.*

to a Texas Instrument's CC2541 which performs the authentication operations. If the authentication is successful, the main PCB will be notified.

The main PCB contains a Texas Instrument's MSP430 which is responsible for controlling locking, unlocking the lock, and also reading the user customizable DIP switches as can be seen in Figure 3. The unlock/lock operation is triggered when the BLE breakout indicates a successful authentication and the external PCB signals that the user has touched the capacitive touch sensor on the face plate. For the purposes of this paper we will focus on the MSP430's firmware and leave the other microprocessors to future work.

4.2 Software

The Kevo uses encrypted electronic keys called eKeys as a means of authenticating a user. These keys are centrally managed by the Kevo website which enables users to send and generate eKeys for anyone they would like. The Kevo mobile app will pair with the lock over BLE and transmit the eKeys to the lock to authenticate the user.

5 Discussion

5.1 MSP430 Firmware

A joint test action group (JTAG) header was soldered on to the debug port on the main PCB and used to extract the firmware from the MSP430. The security bit on the chip was not set so this extraction process was trivial. After analyzing the original firmware, we were unable to find any vulnerabilities so we began the development of a malicious firmware package that could be covertly flashed to a lock an adversary had one time access to in order to gain unlimited repeat access.

5.1.1 Malicious Firmware

The interrupt service routine (ISR) that was triggered by a tap on the faceplate was identified along with the subroutine that is called to perform the lock/unlock operation. The MSP430 features a real time clock which was used to perform build a secret knock where the number of taps per minute is counted and if it reaches a certain count the lock will unlock. An adversary could flash this malicious firmware within approximately five minutes of time with access to the inside of the lock and would be granted surreptitious access anytime they pleased. Unlike with repairing attacks where an adversary would press the reset button on the inside of the lock to pair their device, this attack will not affect the operation of the lock with valid eKeys.

Though this malicious firmware package is specifically targeted towards the Kwikset Kevo Bluetooth lock, it should be compatible with many modern Kwikset locks which use a similar system architecture. A different Kwikset product that did not feature bluetooth was sampled and appeared to have the same architecture as the Kevo we analyzed. However, we did not have time to investigate it further and fully confirm this.

5.2 Mobile Application

The Kevo mobile app is responsible for handling the bluetooth pairing with the lock and the transmission of the eKeys. Reverse engineering the mobile app provides an excellent way to reconstruct the protocol specifics being used. While the Kevo mobile app is available for both Android and iOS we choose to target the Android application as the binaries are easier to obtain and reverse engineer.

5.2.1 APK Analysis

The APK was decompiled into a Java Archive (JAR) using dex2jar. The JAR could then be imported in Eclipse using the JD-Eclipse plugin. The Kevo mobile app authors appear to have obfuscated the source code, most likely using a tool such as ProGuard [14]. This obfuscation replaces the names of classes, functions, and variables with short, meaningless names and additionally rearranges code to make analysis more difficult.

To better analyze the control flow, apktool [22] was used decompile the APK into smali code and the Baksmali assembler to transform the smali code into the dex format [2] used by Android. The smali language is a pseudo-assembly language with register manipulation but well-labeled function calls. This allowed easy modification of

the Kevo application with apktool and SignAPK [4] being used to rebuild the application. The combination of these tools and Android's logging system, logcat [3], allowed debug logging statements to be built into the Kevo app to reveal the control flow and runtime data.

5.2.2 Malicious APK

The Kevo mobile app can be trivially modified to send user credentials to a server of the attackers choosing using a similar method as that used for above for determining the control flow. A malicious version of the mobile app could be distributed through the Android Market place with a similar name to cause users to accidentally install it and have their credentials compromised. The malicious app we created performs all of the functions of the normal app so the attack would be transparent to the user. Gaining access to the user's credentials is very dangerous because it would allow the attacker to sync all of the user's eKeys from the central storage and gain access to any Kevo lock they had access to.

5.2.3 Communication Protocol

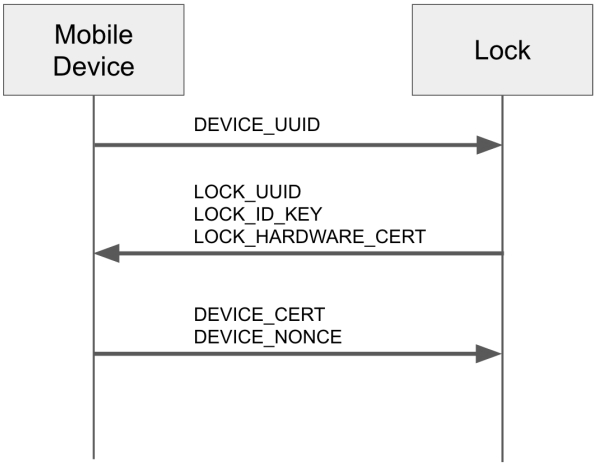

Figure 4: *Overview of the pairing protocol between a mobile device running the Android Kevo app and the Kevo lock.*

Pairing

To begin reverse engineering the Kevo protocol the BLE characteristics it was using had to be determined. The Kevo mobile app sets these parameters when it pairs with the lock which enabled us to extract these parameters using debug statements as mentioned previously. The Kevo app advertises characteristics with UUIDs of 0999 and 0979 for the lock to write data to and characteristics with UUIDs of 0989 and 0979 for the lock to read data from. These channels were confirmed by configuring a computer to use these channels and then executing the Kevo app to see if it detected the computer as a lock. The lock writes to characteristic 0979 to send information to the mobile app, such as the LOCK_UUID. The lock reads from characteristic 0979 to read information about the mobile app, such as the DEVICE_UUID. The lock uses characteristic 0999 to modify the state of the mobile app. In particular, a write to characteristic 0999 changes the runtime type of the object that will respond to any subsequent BLE requests therefore changing the behavior of the app for future reads and writes. The control flow could be followed from where these parameters were set to where the exchange took place allowing us to determine that during the pairing the lock and mobile app exchange certificates and 32 byte nonces in addition to the UUIDs. An illustration of this can be seen in Figure 4.

Unlock

The Kevo app generates a shared secret with the Kevo lock using a curve25519 implementation of Elliptic Curve Diffie-Hellman. Debugging the first several packets sent to the Android app show function calls in two classes: *com.unikey.kevo.bluetooth.l* and *com.unikey.kevo.bluetooth.o*. These classes have a few unobfuscated function and string names which indicate the creation of a curve25519 shared secret. The *com.unikey.crypto.LibCrypto* class obtains a certificate public key and private key from the Android Keystore System. A brief step-through of the decompiled Android app shows that a state modifying pattern similar to the pairing protocol also occurs during the unlock sequence. This pattern significantly slows down reverse engineering, since the the Java objects responding to BLE reads and writes changes at runtime. We leave a full debugging the unlock protocol to future work.

5.2.4 Replay Experiments

We captured the data sent during the pairing process with our added logging. We were successful in attacking the *pairing* process; we can pair any mobile device with the Kevo lock.

We successfully performed a replay attack on the *unlock* process, but the attack uses a strong assumption. We were only able to unlock the Kevo lock only when using the original mobile device that had paired with the Kevo lock. Surprisingly, a replay of the unlock process could be performed from any Kevo account. Thus, if the adversary gained access to victim's mobile device, the adversary would not additionally need access to the victim's Kevo account.

A key weakness in the replay experiments was that the handshake needed to be recorded at the application layer.

We hypothesize that it is possible to eavesdrop at the BLE layer and obtain application data using Crackle [16].

After investigating the pairing protocol, we began experimenting with the unlock protocol. We had some success with replaying an unlock. Consider an honest user Alice who unlocks her Kevo lock. If Eve captured the pairing sequence with Alice's lock, Eve could associate any mobile device with Alice's smart lock. We found that Eve could then replay the unlock sequence only using Alice's original mobile device. Eve could not replay the pairing using her mobile device and then replay the unlock using her mobile device. Although when Eve used Alice's original mobile device, Eve could sign in as any Kevo user and still replay the unlock. These results show the replay is dependent on the physical mobile device rather than the the user's account.

Thus the pairing and unlock can only both be replayed together given access to the original mobile device paired with the lock: an adversary cannot replay an unlock using their own mobile device's Kevo app.

6 Countermeasures

6.1 PCB Reverse Engineering

While current techniques to hamper circuit board reverse engineering are not perfect, they should still be employed whenever possible and economically feasible. In most cases, implementing such defenses incur minimal cost to the developer but exponential work for the adversary.

We first suggest two defenses involving the design of the circuit board: using multi-layered PCBs and components with ball-grid-array packaging. These techniques are often both used on circuit boards due to board constraints [13]. We encourage designers to consider their security advantages when making board design decisions. A multi-layered PCB prevents the adversary from simply visually following the traces to determine how the peripherals are connected as the traces disappear into the board: Kwikset did use a multilayered PCB. Similarly, ball-grid-array packaging obscures the microprocessor pins and prevents the adversary from being able to connect leads to the pins by hiding the connection under the processor. Both of these defenses can be defeated by X-raying the board. However, this requires special equipment and still does not provide as much information as the information gained from having physical access to the pins and connections.

Two other related defenses include making the components indistinguishable, and removing pin labels on the final product. The identity of a microprocessor told us which assembly language to convert the binary into and also the pin layout. Kwikset tried to obscure the label on the MSP430 by painting over it, but the label was still visible through the paint and the paint could simply be removed with rubbing alcohol. In order to completely obscure microprocessor labels, the microprocessor manufacturer would need to provide an unlabeled option.

Additionally, Kwikset left the debugging silkscreen on. Thus many important pins were labeled, which greatly accelerated our understanding of the board. Leaving the debugging silkscreen on also allowed us to debug the microprocessor. We advise that modifications be made to the manufacturing process to remove labels on boards marked for deployment.

6.2 Malicious Firmware

While the defenses discussed in Section 6.1 simply make reverse engineering more difficult, there are defenses that will completely defeat a malicious firmware attack. The primary defense involves restricting access to read/write capabilities of the flash memory on the microprocessor. On the MSP430, the JTAG debug interface was labeled and unlocked, which allowed us to read and write the firmware. The JTAG also provided debug functionalities, allowing us to step through and set breakpoints while the firmware was running on the lock. The JTAG can be locked in a number of ways — including blowing the fuse — preventing debugging after preliminary testing at the factory [21].

Along with locking the JTAG, the bootloader should also be locked. The TI MSP430 default bootloader used by Kwikset is locked by default when any firmware is written to it. This default bootloader is password protected by 32 bytes of the interrupt vector table [19]. Furthermore, Kwikset could have written a custom Bootloader requiring a signed hash of the firmware to prevent alteration [20]. Finally, by completing frequent required firmware updates, any malicious firmware can be removed on the next update cycle. The Kevo Android app could make it extremely apparent that the firmware is out-of-date, rather than requiring the user to click through multiple menus to find how to update the firmware.

If an adversary can gain access to the lock's binaries, assembly-level code obfuscation can prevent easily understanding the code, making the ability to add a backdoor significantly more difficult. Tools such as the Movfuscator compiler [9] turn C code into obfuscated assembly. However, obfuscation tools would need to be adapted to target the MSP430 ISA. We believe we encountered some obfuscation: a specific set of instructions which would cause undetermined behavior in the entire system when shifted. However, we were able to simply add our backdoor after that instruction to prevent shifting that line.

6.3 Mobile App

Although there is no immediate solution to preventing mobile app reverse engineering, the developers could have taken advantage of tools similar to DexGuard [11], which provides string and class encryption, hindering static analysis of parts of the code. Alternatively, the Kevo developers could have included a pre-compiled C library to handle authentication and would be more difficult to reverse engineer. These suggested mitigations will merely slow down an adversary, not prevent reverse engineering.

7 Future Work

Future work into the Kevo and smart locks in general should aim towards finding vulnerabilities that can be exploited without requiring physical access to the inside of the lock or a victims phone. This could be done by fully reverse engineering the unlock protocol by retrieving the firmware from the processor responsible for the authentication functions and looking for potential implementation issues in the cryptography being used. For the Kevo lock specifically the threat would arise from a vulnerability in the implementation of the curve25519-based crypto protocol on the CC2541 microprocessor.

The likely exploit would lie in an incorrect assumption about the data being sent between the app and the lock. From our success in replaying a pairing of devices as discussed in Section 5.2.4, it may be possible to spoof values needed for an unlock. Further work would also need to be spent observing the data from a pairing or an unlock at the BLE layer. Since the lock only targets the closest device during a pairing and an unlock, it would be difficult to log the exchanged data. We believe it possible to eavesdrop at the BLE layer and decrypt using Crackle [16] to obtain the exchanged data. Alternatively, it may be possible to man in the middle a pairing since the mobile device and lock certificates are exchanged during the pairing.

We have not evaluated any smart locks beyond the Kwikset Kevo lock. We leave analyses of other brands of smart locks to future work.

8 Conclusion

Smart locks are the foundation of security for the IoT Smart Home as they are the gateway into the home. Despite smart locks being a security system, we find in practice that the technical security of the analyzed Kwikset Kevo smart lock is far from the best security practices expected from a smart lock. We believe this stems from a lack of experience with technical security as the smart lock is a relatively new concept to most lock manufacturers. In addition, due to few high profile adversarial exploits, smart lock manufactures may be optimizing for cost, and technical security has yet to become a priority as it is for systems such as cars.

In this paper, we presented our analysis of the Kwikset Kevo lock, as well as lessons learned in reverse engineering the system. We discussed several attacks: a secret knock firmware attack, a credential stealing Kevo mobile app, and the underlying protocol user to perform the authentication between the lock and mobile device . We suggested many defenses, ranging in cost and complexity. A key lesson learned is that there are many low-cost solutions that should be employed for these security critical systems. IoT manufactures could go a long way to making the future Smart Home secure by simply locking their JTAG interfaces.

Acknowledgments

The authors first and foremost thank Eric Wustrow for his initial work on analyzing the Kevo Kwikset lock. We would also like to thank the kind folks at the University of Michigan's Lab 11: Brendan Ghena, Josh Adkins, and Neal Jackson provided helpful insight during the hardware reverse engineering process, while Noah Klugman was generous enough to loan out a Nexus 9 tablet for our use. Additionally, we would like to thank John Mamish for providing insight into the cost and potential techniques for bypassing a JTAG lock.

References

[1] Locking-picking, 2004. http://www.encyclopedia.com/topic/Lock-Picking.aspx.

[2] Android. Dalvik executable format, 2015. https://source.android.com/devices/tech/dalvik/dex-format.html.

[3] Android. logcat Command-line Tool, 2016. https://developer.android.com/studio/command-line/logcat.html.

[4] Appium. Apk sign, 2013. https://github.com/appium/sign.

[5] Kai Wang Benjamin Laxton and Stefan Savage. Reconsidering physical key secrecy: Teleduplication via optical decoding. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CSS)*, 2008.

[6] Anthony Bonkoski, Russ Bielawski, and J. Alex Halderman. Illuminating the security issues surrounding lights-out server management. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX.

[7] Ben Burgess, Eric Wustrow, and J. Alex Halderman. Replication prohibited: Attacking restricted keyways with 3D-printing. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[8] Ang Cui, Michael Costello, and Salvatore J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*. The Internet Society, 2013.

[9] Chris Domas. M/o/Vfuscator, 2016. https://github.com/xoreaxeaxeax/movfuscator.

[10] Andy Greenberg. Watch this wireless hack pop a car's locks in minutes, August 2014. http://www.wired.com/2014/08/wireless-car-hack/.

[11] Guard Square. *DexGuard*. https://www.guardsquare.com/dexguard.

[12] Stephen Hall and Paul Lariviere. Making smart locks smarter, March 2014. http://blog.perfectlylogical.com/post/2015/03/29/Making-Smart-Locks-Smarter/.

[13] Intel. *Ball Grid Array (BGA) Packaging*. http://www.intel.com/content/dam/www/public/us/en/documents/packaging-databooks/packaging-chapter-14-databook.pdf.

[14] Eric Lafortune. Proguard, 2015. http://proguard.sourceforge.net/.

[15] Mike Ryan. Bluetooth: With low energy comes low security. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX.

[16] Mike Ryan. crackle, 2016. https://github.com/mikeryan/crackle.

[17] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice: Automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*. The Internet Society, 2015.

[18] Craig Silverman. Seven creepy baby monitor stories that will terrify all parents, July 2015. https://www.buzzfeed.com/craigsilverman/creeps-hack-baby-monitors-and-say-terrifying-thing.

[19] Texas Instruments. *MSP430 Programming With the Bootloader (BSL)*, September 2015. http://www.ti.com.cn/cn/lit/ug/slau319k/slau319k.pdf.

[20] Texas Instruments. *Secure In-Field Updates for MSP430 MCUs*, November 2015. http://www.ti.com/lit/an/slaa682/slaa682.pdf.

[21] Texas Instruments. *MSP430 Programming Via the JTAG Interface*, February 2016. http://www.ti.com/lit/ug/slau320w/slau320w.pdf.

[22] Ryszard Wiśniewski and Connor Tumbleson. Apktool, 2016. http://ibotpeaches.github.io/Apktool/.