WatchWitch: Interoperability, Privacy, and Autonomy for the Apple Watch

Nils Rollshausen
Secure Mobile Networking Lab
Technical University of Darmstadt, Germany
nrollshausen@seemoo.de

Alexander Heinrich
Secure Mobile Networking Lab
Technical University of Darmstadt, Germany
aheinrich@seemoo.de

Matthias Hollick
Secure Mobile Networking Lab
Technical University of Darmstadt, Germany
mhollick@seemoo.de

Jiska Classen
Hasso Plattner Institute
University of Potsdam, Germany
jiska.classen@hpi.de

Abstract

Smartwatches such as the Apple Watch collect vast amounts of intimate health and fitness data as we wear them. Users have little choice regarding how this data is processed: The Apple Watch can only be used with Apple's iPhones, using their software and their cloud services. We are the first to publicly reverse-engineer the watch's wireless protocols, which led to discovering multiple security issues in Apple's proprietary implementation. With *WatchWitch*, our custom Android reimplementation, we break out of Apple's walled garden—demonstrating practical interoperability with enhanced privacy controls and data autonomy. We thus pave the way for more consumer choice in the smartwatch ecosystem, offering users more control over their devices.

Keywords

Wearables, Apple Watch, Privacy, Reverse Engineering

1 Introduction

Of all our devices, our smartwatches know us best: Always on our wrists, they collect intimate data even as we sleep. As users, however, we have little control over this data. The dominating smartwatch vendors—including Apple [90, p. 39–41], Samsung [66], and Google [23]—all rely on ecosystem lock-in effects: For example, users can only use their Apple Watch with an iPhone, and no third-party smartwatch can offer the same level of integration with an iPhone as Apple's watches. Apple Watch owners are forced to use their watches on Apple's terms—using their devices, their software, and their cloud services. This leaves users of many smartwatches, from Apple or other vendors, with little control over their devices and nebulous promises of privacy.

This vendor lock-in has not gone unnoticed by legislators: An antitrust motion in the US identifies the Apple Watch as part of Apple's alleged smartphone monopoly and demands more interoperability [90]. Similar efforts are being implemented in the EU with the Digital Markets Act [38]. Apple claims to have researched interoperability for three years, concluding that it was infeasible [67].

In this context, we introduce *WatchWitch*: Based on extensive reverse-engineering of the Apple Watch, we create an open-source Android app that allows users to use their Apple Watch on their own terms and with an Android phone. Unlike the closed-source iOS system, Android lets users control much larger parts of the software stack. Our proof-of-concept *WatchWitch* app shows that interoperability is feasible in practice and demonstrates how all users, iOS and Android, can benefit from using custom software built with their needs in mind—gaining independence from vendors with a history of privacy violations [22, 26, 39, 70].

Beyond reimplementing features provided by Apple, *WatchWitch* gives users better privacy controls and full autonomy over their data: We allow users to make fine-grained decisions about the watch's network connections through a user-controlled firewall and keep all their health data securely on-device. At the same time, we give them full access to the data their watch collects beyond what is displayed in Apple's standard user interface (UI).

In the process of reverse-engineering the Apple Watch, we analyze its underlying security architecture—including how sensitive health data is protected in transit from the watch to the phone.

In short, our paper makes the following contributions:

(1) We analyze and document several previously unknown and widely deployed protocols used by the Apple Watch, thus opening them up to further security research.
(2) We provide a tool suite for working with and analyzing the Apple Watch's proprietary protocols.
(3) We demonstrate real-world interoperability using our open-source *WatchWitch* app, supporting several core smartwatch features and maintaining hardware-backed security.
(4) We show usable privacy enhancements within *WatchWitch*, giving users more control of—and insight into—the intimate data their watch collects.
(5) We analyze the security of the Apple Watch's wireless communication and discuss several vulnerabilities. The corresponding mitigations improve security for all users.

The source code of WatchWitch, with further tooling and a demo video, can be found at `github.com/seemoo-lab/watchwitch`.

Responsible Disclosure

We disclosed two vulnerabilities to Apple in November 2023. Apple acknowledged both vulnerabilities. They decided not to fix the first

Proceedings on Privacy Enhancing Technologies 2025(4), 94–111

https://doi.org/10.56553/popets-2025-0121

vulnerability as they claim it cannot be exploited (Section 6.3). They released a fix for our second issue in March 2024 (Section 6.2) [8]. We reported a third issue in May 2024 and are awaiting a fix.

2 Background

We introduce smartwatches, the health data they collect and current interoperability efforts. Furthermore, we give background information on IPSec as it plays an important role for the Apple Watch.

2.1 Smartwatches

Smartwatches are wearable devices worn in place of an analogue watch. They offer a variety of features in addition to telling the time, typically including health and fitness tracking, instant messaging, and integration with a connected smartphone. Smartwatches that go beyond simple fitness trackers can allow users to install third-party apps. This market segment includes the Apple Watch and various watches running Google's *Wear OS* operating system. In this class of devices, Apple alone holds a market share of 45% [29].

Only a few existing smartwatches focus on open hard- and software [56, 74, 83]. Only PineTime goes beyond a simple hobbyist project, and none of them can compete with the capabilities of flagship commercial watches. Furthermore, some commercial smartwatches running Wear OS can be modified to run the Asteroid operating system [15], emphasizing user control and privacy at the cost of feature support.

When it comes to health and fitness data collection, even budget models typically come with heart rate sensors and step counters. What distinguishes the more expensive flagship models are more advanced sensors [6, 78]. These additional hardware capabilities are paired with more software features to enable precise tracking and estimation of health metrics.

Current models of the Apple Watch, which are the focus of our work, are consistently at the top of the market regarding sensor capabilities and software features. The current Apple Watch Ultra 2, for example, comes with electrical and optical heart sensors, a pulse oximeter, a skin temperature sensor, a GPS receiver, a gyroscope, and an accelerometer. Equipped with this hardware, it offers electrocardiogram (ECG) readings, extensive heart rate monitoring, blood oxygen measurements, activity recognition, sleep phase tracking, ovulation time estimation, and more [6, 46].

2.2 Health Data

Modern smartwatches collect a trove of highly intimate health data. With watches offering sleep tracking [12], users are encouraged never to take off their devices. As a result, an Apple Watch may have access to a long-term, near-uninterrupted stream of sensor readings—a level of medical surveillance reserved to clinical settings only a few years prior.

If the very same data *were* collected in a clinical setting, strict regulations, such as HIPAA [91] in the US, would likely apply to storing, using, and sharing of said data. However, since smartwatches are not marketed as medical devices, most regulations on collecting and using medical information do not apply, giving manufacturers significantly more leeway to store, share, and sell user data [73].

Consider how sensitive the information collected by these devices is: As certain bodies become more and more politicized, so does the data quantifying them. Period tracking apps on user's smartphones have already been used to prosecute unlawful abortions in the UK [33]. Thus, the long-term skin temperature recordings and user-entered cycle tracking information present on a smartwatch can become a dangerous liability for the user. Similar concerns apply to other measurements that could be used to infer a wide range of physical and mental health conditions.

When users opt to track their health using a smartwatch, they have to trust the device manufacturer in regard to this highly private data. Many manufacturers, however, do not have great track records when it comes to privacy and security [22, 26, 39, 70].

While Apple claims that all health data is exchanged using end-to-end encryption even if shared using iCloud and remains inaccessible while the device is locked [7], users still have to trust Apple to maintain this behavior over future updates. As an additional risk factor, the same reasoning applies to any third-party app that is granted access to *HealthKit* data. Large-scale leaks of private HealthKit data via third parties have already occurred [70].

Given a sufficiently authoritarian government, Apple could also be compelled to disclose some types of data collected by their watches to government entities by altering their software. The Google-owned fitness tracker manufacturer Fitbit has already stated that they would comply with government requests for user health data, including period tracking information [39].

For all these reasons, we stress the need for smartwatch infrastructure that places us as users in control of our most intimate data—free from third parties and with full autonomy over our bodies and the data traces they leave behind.

2.3 Interoperability

Manufacturers often market smartwatches as companion products for their latest smartphone. With the deep integration between a smartwatch and smartphone being a key selling point, vendors have little incentive to ensure interoperability with other devices.

The Apple Watch in particular is incredibly tightly coupled with Apple's iPhone and iCloud ecosystem [10, 14], using proprietary protocols that are unavailable to third parties. Third-party watches can, for example, receive push notifications from an iPhone but not reply to them [90, p. 40]. And while iPhones work with third-party smartwatches to some degree, the Apple Watch cannot be used with non-Apple smartphones in any way.

Unlike Apple's smartwatches, Wear OS devices can technically be used with iPhones. Recent flagship models from both Samsung and Google, however, have abandoned iOS compatibility due to concerns about an imperfect user experience [23, 66]. Samsung also requires a paired Galaxy phone to use some features of its latest watches [78]. This means that users of either platform are essentially locked in to their choice of smartphone after purchasing a smartwatch at significant expense—and vice versa.

This is also part of the US antitrust complaint brought against Apple in March 2024. The complaint poses that Apple is deliberately limiting the use of third-party watches on iOS and restricting Apple Watch interoperability to expand their monopoly in the US smartphone market [90, pp. 39–41]. According to reporting by

9to5Mac [67], Apple stated that they had researched possible Android interoperability for three years but ultimately concluded it was not technically feasible.

However, in Section 5, we show that such interoperability between the Apple Watch and an Android smartphone is indeed possible: We create a working Android prototype that supports core functionality and allows developers to add support for further services. We found that there were multiple technical difficulties Apple may have encountered, which we had to solve with WatchWitch. In particular, Android's network stack does not expose essential functionality on non-rooted devices, and storing health-related data with hardware-backed encryption is only supported on recent Android devices [64]. Switching communication from Bluetooth to Wi-Fi might also incur higher energy costs, lowering the watch's battery runtime. Nonetheless, the obstacles that limit our prototype (see Section 5.4) are not insurmountable and could likely be eliminated by Apple.

Aside from technical feasibility, the reason for Apple not providing Android interoperability could also lie in their interest not to weaken the iPhone's market position. The claims made in the antitrust complaint, which cites Apple-internal documents, match this reasoning [90, pp. 40]. If the antitrust motion were to succeed and result in Apple opening up their proprietary interfaces, this would certainly benefit a wide range of consumers: The Apple Watch is a remarkable piece of hardware, and presumably many Android users would use it with their phones if given the ability to do so. Similarly, iPhone users could choose from a much wider variety of smartwatches while enjoying the same deep, seamless integration with their phones that an Apple Watch provides.

2.4 Apple Watch Features

As a modern flagship smartwatch, the Apple Watch is designed to integrate tightly with a paired iPhone. But even on its own, the watch can measure and collect health data from its sensors, track workouts, play audio content, and more. When connected to Wi-Fi or cellular, users can also check weather and stock market data or install new apps from the App Store.

To use the full extent of the watch's capabilities, however, it has to be connected to its paired iPhone at least periodically. While the Apple Watch continues to collect data when not connected to the phone and allows users to add new workouts and other data points, the views of this data offered on the watch are minimal. Users can, for example, only check their current heart rate, current noise levels, and a summary of their last night's sleep.

A full overview of the user's health metrics with long-term data can only be seen in the Health app on the connected iPhone, which receives a copy of all collected samples when it connects to the watch. An active connection to the paired phone is also required to receive most notifications (iMessage being the exception), synchronize calendar events, contacts, and photos, share the phone's cellular connection, or access its camera.

Some of the Apple Watch's features, particularly those related to health measurements, are unavailable in some regions for regulatory reasons. This includes the ability to record ECGs and detect irregular heart rhythm and atrial fibrillation [13]. The pulse oximeter found on recent watch models cannot be used in the US due to a patent dispute [46]. With WatchWitch, we can circumvent these restrictions and enable features regardless of geographical location.

2.5 IPSec

The Apple Watch uses IPSec to create a Virtual Private Network (VPN) connection between the watch and the phone. IPSec defines protocols to establish secure tunnels between two devices.

With the Internet Key Exchange Protocol (IKEv2) [53] the two devices negotiate cryptographic parameters and perform authenticated key exchanges. During an IKEv2 handshake, parties typically use long-term asymmetric keys to establish ephemeral symmetric shared secrets. These ephemeral secrets are then used to encrypt application traffic between the devices using IP Encapsulating Security Payloads (ESP). IKEv2 is not designed to carry application traffic but can perform a range of signaling tasks, communicating parameters, network addresses, link states, etc. A single IKEv2 message typically consists of several *payloads*, containing different kinds of information—for example supported algorithms, key material, or certificates. Implementors may also extend the protocol with their own private payloads.

ESP is only a thin wrapper protocol applying the encryption and authentication mechanism negotiated by IKEv2 to application traffic using the given ephemeral key material.

3 Methodology

Our work is based on reverse-engineering the network protocol stack of the iPhone and the Apple Watch. We look at the communication between both devices from two angles: **dynamically** by observing live communication and **statically** by analyzing files and system binaries. Our general reverse-engineering approach follows related work on iOS protocols [25, 48, 49, 57, 76, 84]. In addition to these methods, we develop our reimplementation jointly with our reverse engineering in a process of *iterative reimplementation*.

Before performing reverse engineering, we must ensure system-level access to our devices. Since iOS does not allow rooting, we use public jailbreaks to gain root access to the operating system.

3.1 Dynamic Analysis

System logs. We inspect the iPhone and Apple Watch system logs using the macOS Console app. These logs give a first look into which processes are used when both devices exchange data.

Frida. We use Frida [75], a dynamic reverse-engineering toolkit, which allows intercepting functions in running processes on iOS. Whenever a process calls a function that we deem interesting, Frida intercepts the function execution and lets us read and modify variables. Its JavaScript application programming interface (API) allows automating analysis of system functionality, such as dispatching data to threads and verbose logging [24]. This is crucial for us in dissecting the watch's different encryption layers, as it lets us automatically extract private key material and perform decryption and custom message parsing on the fly. Without this runtime access to the device, we could not gain any visibility into the exchanged messages, as every session uses fresh ephemeral key material.

Network interfaces. We observe Bluetooth and Wi-Fi interfaces directly using developer tools such as `tcpdump` [86] and Apple's

Bluetooth Packet Logger [9]. We use these tools to build a corpus of observed messages that helps us understand both the static message structure as well as the dynamic protocol behavior, such as the semantics of sequence numbers or different data streams.

3.2 Static Analysis

Binary analysis. Internally, multiple iOS processes handle communication between the iPhone and the Apple Watch. Using disassemblers we can more closely inspect the behavior of relevant functions. This lets us uncover codepaths that are not taken during regular usage: Most notably deprecated or rarely used messages and the behavior of protocols in the presence of various errors.

3.3 Iterative Reimplementation

We develop our reimplementation of the protocol stack simultaneously with our reverse engineering efforts. By iteratively integrating discovered protocols and features, we create a lab setup where we have full control over the messages sent by our implementation. This allows us to trigger previously unreachable behavior on the watch and lets us access the watch's deeply nested protocols in a targeted manner. As many of the watch's features are only used in specific scenarios or for a short period during connection establishment, this greatly aids our overall reverse engineering effort.

3.4 Research Framework

Using Frida, we create powerful custom tooling that is able to capture, decrypt, and analyze watch traffic in real time—including full parsing of exchanged messages across all layers of the protocol stack. As we expect these tools to be useful to future researchers investigating the Apple Watch, we publish them at https://github.com/seemoo-lab/watchwitch-tools.

4 Wireless Communication

Wireless interfaces form the core of the communication between an Apple Watch and its paired iPhone. A deep understanding of these interfaces is the foundation of our research: Knowing the details of these proprietary and undocumented protocols lets us analyze their security properties, access the Apple Watch with an Android phone, and add new features promoting user privacy and control.

The Apple Watch has two main wireless interfaces, Bluetooth and Wi-Fi. Both interfaces can be used interchangeably to connect to the paired iPhone. The watch and the phone communicate using a common Wi-Fi access point they are both connected to, without using Apple's proprietary Wi-Fi enhancements [85].

On a high level, all Apple Watch communication is routed over an IPSec tunnel that provides encryption and authentication. This tunnel is established using IKEv2 [53] and carries data using ESP [54]. Both of these protocols are designed to run on top of the *Internet Protocol (IP)*. For Wi-Fi connections, this happens transparently. For Bluetooth connections, Apple uses the proprietary *Network Relay Link Protocol (NRLP)* to carry IKEv2 and ESP payloads.

Figure 1 shows how incoming messages from the Apple Watch are handled on the iPhone. Message handling on the watch should be nearly identical as both devices share a common codebase. However, we focus our investigations on the iPhone side due to the existing jailbreak and tool support for iOS [42, 71].

(1) IKEv2 packets arrive over the wireless link (Bluetooth or Wi-Fi) and are passed to the *terminus* daemon, which establishes an IPSec tunnel between the devices.
(2) ESP packets carrying data for the established tunnel arrive and are decrypted by the kernel networking stack. For Bluetooth connections, ESP packets arrive in NRLP payloads, and the terminus daemon forwards them to the kernel.
(3) Decrypted Transmission Control Protocol (TCP) packets are passed on to their destinations—the terminus daemon for Internet-bound traffic using the *Shoes* proxy and the identity services daemon for device-to-device traffic using *Alloy*.
(4a) For Alloy messages, the identity services daemon performs additional decryption using the *MessageProtection* framework if necessary and passes messages on to their destination services via Inter-Process Communication (XPC) based on the message topic.
(4b) For Internet sharing traffic, the terminus daemon passes packets to their intended remote host using the phone's Internet connection.

Regardless of the wireless link technology used, the higher-level protocols remain unchanged, as shown in Figure 2.

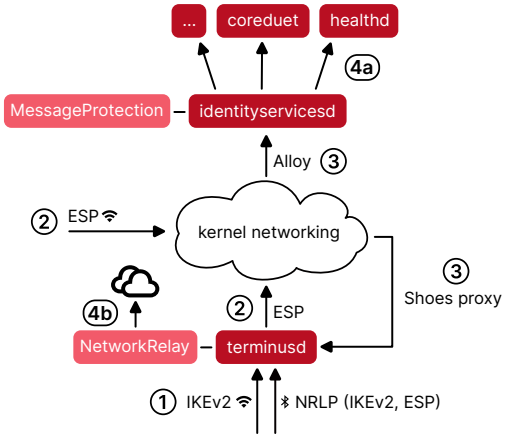

Figure 1: Watch message handling logic on iOS. Darker cells show daemons involved in the communication, lighter cells show selected frameworks with related functionality.

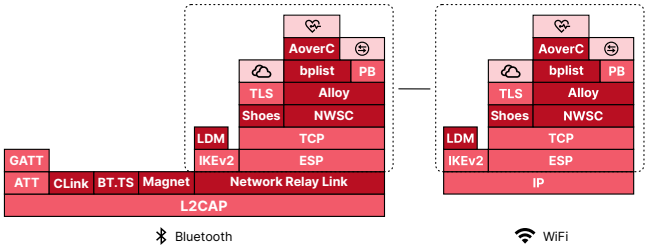

Figure 2: Protocol stack for Bluetooth and Wi-Fi connections. Cells with a lighter background are standardized open protocols/formats; darker cells are proprietary and largely undocumented. *PB* is *ProtoBuf*. Icons represent health data (heart), Internet access (clouds), and local messages (arrows).

4.1 NRLP & Magnet

The foundation of the protocol stack for Bluetooth connections is the *Network Relay Link Protocol (NRLP)*. NRLP forms a transport that carries multiple higher-level protocols over Logical Link Control and Adaptation Protocol (L2CAP) channels. It appears to be used exclusively for communication with the Apple Watch and is handled by the terminus daemon.

NRLP utilizes dynamically allocated L2CAP channels, which are negotiated using Apple's *Magnet* protocol. Magnet was first documented by Heinze et al. [48]. We expand on their work with a detailed list of message types in Appendix A.1.

We mainly observe NRLP carrying ESP and IKEv2 payloads but, perhaps for historic reasons, other protocol types are supported, including plain IP and an echo service. We provide more detail on NRLP message types and structures in Appendix A.2.

4.2 IKEv2 & ESP

While Magnet and NRLP are used exclusively for Bluetooth-based connections, IKEv2 and ESP—as well as all the protocols building on top of them—are used for both Bluetooth and Wi-Fi connections with almost no modification. As both protocols are designed to be used with the Internet Protocol, they can be used as-is in the Wi-Fi case. The lower-level NRLP provides a Bluetooth compatibility layer for using IP-based protocols in this context.

IKEv2, as used here by Apple, conforms largely to the standard described in RFC 7296 [53]: It serves as a mutually authenticated key exchange at the start of every connection that establishes ephemeral secrets that will secure all further communication.

Payload data for higher-level protocols will then be encrypted using the negotiated secrets and cryptographic algorithms using ESP, which Apple uses as described in RFC 4303 [54]. We list the ciphers used for IKEv2 and ESP in Appendix A.4.

4.2.1 Data Protection Classes. Apple uses the concept of *data protection classes* to separate data into different sensitivity levels [4].

These data protection classes are distinguished by when their decryption keys are available: Class A keys are only accessible when the iPhone is currently unlocked, whereas class C keys are available continuously once the phone has been first unlocked after booting, and class D keys are always available.

The watch opens two separate IPSec tunnels, using the phone's class C and D public keys respectively. The resulting tunnels are used for different kinds of data: The lower-security class D tunnel is used to provide Internet sharing and backup watch settings, which will also work when the paired iPhone has not yet been unlocked after booting. Most other traffic uses the class C tunnel instead, which is only available once the phone has been unlocked.

Health data sent by the watch uses the highest protection class A, for which no dedicated IPSec tunnel exists. Instead, class A data is carried over the class C tunnel with additional *A-over-C* encryption applied. We discuss this encryption in more detail in Section 4.4.

4.2.2 Notify Payloads. In a deviation from 'plain' IKEv2, Apple extensively uses vendor-specific *Notify* payloads. These payloads can be included in any IKEv2 message and carry arbitrary, vendor-specified data typically used for signaling purposes. Apple uses these payloads to communicate device names, software versions, and IPv6 addresses used to set up routing for the IPSec tunnel. Beyond that, they also build an entirely separate protocol operating on top of custom IKEv2 notify payloads centered around so-called *Link Director Messages (LDMs)*.

LDMs have their own protocol header and carry a number of Type-Length-Value (TLV) encoded substructures. Most interesting of these are the types *UpdateWi-FiAddressIPv4* and *-IPv6*, which are used by the watch and the phone to discover their Wi-Fi IP addresses when connecting over Bluetooth. During the initial connection, or as part of an IKEv2 keepalive when a device's address changes, both parties share their local IP addresses and ports on which they accept IKEv2 connections over Wi-Fi.

Other TLVs can signal that a connection has restarted or negotiate preference for a certain link type (Wi-Fi or Bluetooth). The complete list of supported TLVs, alongside other custom notify types and the LDM header structure, can be found in Appendix A.3.

4.3 Alloy

The Alloy protocol is the heart of all communication between the Apple Watch and the phone. Operating on top of the IPSec tunnel, it forms the main messaging bus that delivers messages to many different services. As such, it is rather complex: Alloy defines a total of 54 message types and 231 message topics, which it delivers to and from more than 150 different binaries on the iOS side.

Alloy uses several long-standing TCP connections—one control channel and multiple data channels, distinguished by data protection class and urgency. The *identity services daemon* handles all these connections, listening on port 61315 (for the control channel) and 61314 (for most data channels).

To open multiple distinct data channels using a single server port, Apple uses yet another proprietary protocol called *Network Service Connectors (NWSC)*. NWSC is of little importance for the operation of the other protocols beyond negotiating TCP connections, which is why we do not discuss it in detail.

4.3.1 Control Channel. Once a control channel has been established, both devices send *Hello* messages, containing version information, device identifiers, and support for various features. Immediately after the Hello message, each device sends a *SetupChannel* message for every data channel it wants to open with the remote device. These messages contain TCP port numbers, Universally Unique Identifiers (UUIDs) used to refer to the channel being established, as well as a service identifier composed of *account*, *service*, and *name* parts. All channels we observe share the common account *"idstest"* and service *"localdelivery"*. The name components differ from channel to channel—some values include `UTunDelivery-Default-Urgent-C` or `UTunDelivery-Default-Default-D`, where C or D denote data protection classes.

After all desired channels have been established, the control channel has served its main purpose. While channels may be closed and reestablished, there is little control channel activity after the initial connection setup. Table 1 shows a list of supported control channel messages, including many apparently deprecated commands.

4.3.2 Data Channels. Alloy data channels carry application data between the watch and the iPhone. A message containing this

Table 1: Types of Alloy control channel messages.

#	Name
1	Hello
2	SetupChannel
3	CloseChannel
4	CompressionRequest
5	CompressionResponse
6	SetupEncryptedChannel

#	Name
7	FairplayHostSessionInfo
8	FairplayDeviceInfo
9	FairplayDeviceSessionInfo
10	OTRNegotiationMessage
11	EncryptControlChannel
12	SuspendOTRNegotiationMsg

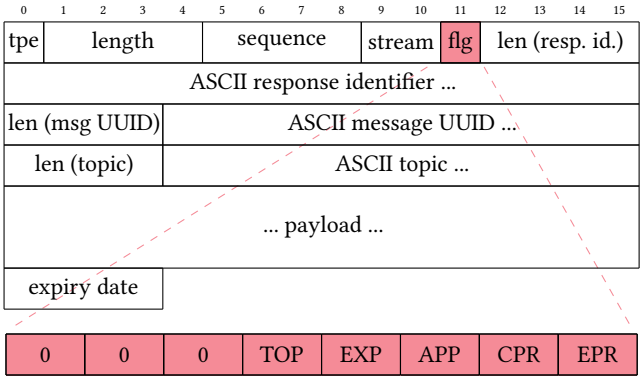

Figure 3: Common byte structure of Alloy application data messages. Topic, topic length, and expiry date are only present if the matching flags are set. Flags are TOP: hasTopic, EXP: hasExpiryDate, APP: wantsAppAck, CPR: compressed, EPR: expectsPeerResponse.

data can be either a generic *DataMessage*, a *DictionaryMessage* containing a dictionary encoded as a binary *plist*, a *ProtobufMessage* containing data encoded using Google's *Protocol Buffers* [45], or a *ResourceTransferMessage* containing data fragments of a file transfer.[1] All message types share the format shown in Figure 3. Message payloads may optionally be compressed using gzip/deflate [35], although the *compressed* flag does not actually indicate if compression was applied and appears to be deprecated.

Every data message has a *stream* associated with a *topic*. The topic determines which service on the device will receive and handle the message—health data, for example, is directed to the *health* daemon using the topic `com.apple.private.alloy.health.sync⟷.classc`. The first message sent for a given stream includes the topic explicitly, while all following messages omit the topic string that can now be inferred from the stream id. Each message also carries its own *message UUID*. If a message is a direct response to a prior message, it will set its *response identifier* to the UUID of that message. Some messages also contain an *expiry date*. If a message is received after its expiry date, the receiver discards it and acknowledges it with an *ExpiredAck* instead of an *Ack* message.

The receiving service, such as the health daemon, can define its own data format for the payload carried by Alloy. Most services use either binary property lists (bplists) or ProtoBuf encoded data.

[1]These four of about 40 Alloy message types carry the bulk of all communication. Further messages are used for signaling; most appear to be deprecated or unused.

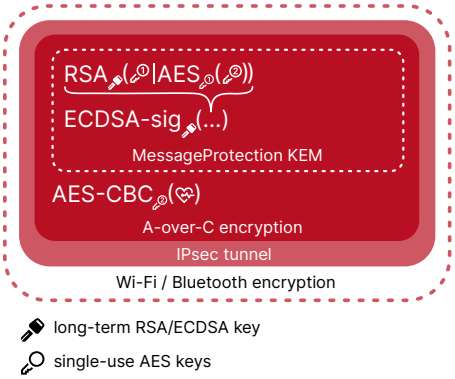

Figure 4: A-over-C layer with surrounding encryption layers.

4.4 Additional Data Protection (A-over-C)

So far, we have seen how Alloy is used to transport data for protection classes C and D over their respective IPSec tunnels. Meanwhile, health data collected by the Apple Watch falls into the highest protection class A. Since there is no dedicated tunnel for this class, Apple applies additional protection to these messages on a per-message basis using what is referred to as *A-over-C* encryption. Plaintext messages are encrypted using a class A key and then delivered via the class C tunnel. On the iPhone the identity services daemon decrypts them and forwards plaintext messages to the destination services as usual. Understanding the details of A-over-C is crucial for our ability to judge its security, which we will discuss in Section 6. Figure 4 illustrates the steps performed to encrypt a given message plaintext p:[2]

(1) Choose random 128-bit ephemeral keys k_1, k_2
(2) Encrypt p using AES in cipher block chaining (CBC) mode with a zero IV and key k_2: $sed = AES - CBC_{k_2}(p)$
(3) Encrypt k_2 using AES in counter mode with key k_1: $c_1 = AES - CTR_{k_1}(k_2)$
(4) Encrypt k_1 and c_1 using RSA-OAEP with the receiver's long-term public key pk_r: $c_2 = RSA - OAEP_{pk_r}(k_1||c_1)$
(5) Sign the resulting ciphertext using the sender's ECDSA long-term private key sk_s: $s = ECDSASign_{sk_s}(c_2)$
(6) Encode s and c_2 as $ekd = version||len(c_2)||c_2||len(s)||s$
(7) Return a *DataMessage* with a bplist-encoded dictionary containing *ekd* and *sed*.

The encapsulated key *ekd* is computed using the *MessageProtection* framework, which uses the same cryptography as Apple's iMessage—which has previously been analyzed by Garman et al. [44].

4.5 Health Data Synchronization

In this section we discuss the synchronization of health data as a service that powers a core feature of the Apple Watch and carries private and sensitive data. Many other services share the same vocabulary of techniques and encodings used.

When the watch has new health samples available, such as new heart rate measurements, it sends them in an Alloy *DataMessage*

[2]The encryption steps performed by MessageProtection are simplified below for the case in which c_1 is no longer than 114 bytes, which is the case for A-over-C.

with the topic `com.apple.private.alloy.health.sync.classc`. The message receives additional A-over-C encryption as health data falls into the most sensitive data protection class A. On reception, the phone decrypts the A-over-C ciphertext once it is unlocked and has access to the corresponding class-A keys. As for all Alloy messages, the message is also protected by the IPSec tunnel in transit—A-over-C merely forms an additional encryption layer.

The resulting plaintext is forwarded to the health daemon, which decodes the ProtoBuf payload into a *NanoSync* message (Figure 5). NanoSync is a lightweight abstraction layer that synchronizes *SQLite* databases on the watch and the phone. The protocol is centered on the notion of *changes*: Each change contains a collection of *samples* of the same type to be inserted into the database. A sample may be a single heart rate measurement or the energy burned by the user over five minutes. Removal of samples is handled similarly: A *change* with a collection of *deleted samples* instructs the health daemon to mark the referenced samples as deleted.

Changes are ordered by their *sync anchors*. A sync anchor is a counter used for a particular domain of the database that is incremented with every change. This method allows the receiving device to apply updates in the correct order, determine if changes are missing, and acknowledge receipt of new data. After receiving a NanoSync message, it will respond with an updated list of its local sync anchors until the sending device communicates that it has no further changes and the synchronization is complete. We show two typical messages in Figure 5.

4.6 Shoes

The Apple Watch can share a paired phone's cellular Internet access, thus allowing Internet-enabled features on the watch while on the go. Internet sharing differs from other services in that it does not use Alloy as a messaging bus. The watch uses a protocol referred to as *Shoes* to open connections to the Internet via the iPhone. The phone then internally uses components of a SOCKS proxy server to forward traffic from the watch to the destination host and back [59].

To open a new connection, the watch sends a *Shoes request* (Figure 6) to the phone on TCP port 62742. This request includes the desired destination host and port, usually in the form of a

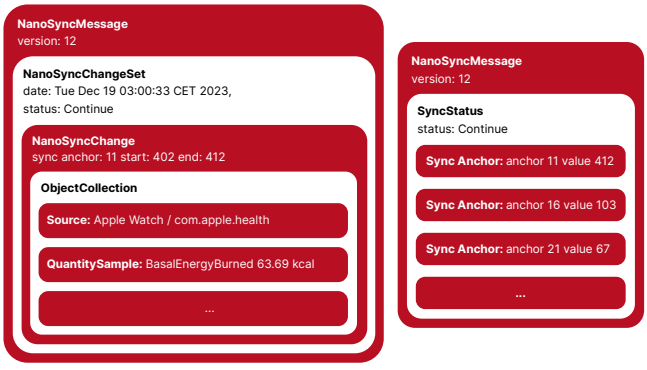

Figure 5: A simplified illustration of a NanoSync message containing new health samples from the watch (left) and a reply message acknowledging receipt of these changes (right).

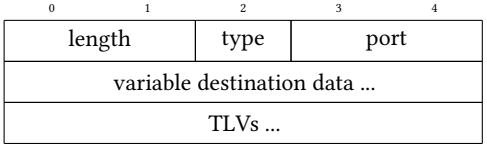

Figure 6: A generic Shoes request. Destination data is dependent on the request type (hostname, IPv4, IPv6, bonjour).

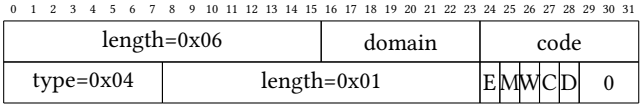

Figure 7: Bit structure of a Shoes reply. The second word is a *network info* TLV field. Flag names: (E) expensive, (M) cellular connection, (W) Wi-Fi connection, (C) constrained connection, (D) connection denied.

hostname or IP address. Typically, the watch also includes optional TLVs containing the name of the requesting process and a set of flags indicating in which network conditions the request should be fulfilled. The watch can, for example, specify that a large file transfer should only be completed when connected to Wi-Fi, while an expensive cellular connection may be used to fetch weather data.

On the phone, the terminus daemon receives these requests and checks the specified conditions against the phone's current network connection. It then responds with a *shoes reply* (Figure 7) communicating whether the request was accepted and what the phone's current connection status is. On success, the phone will forward any following bytes received on the connection to the destination host, sending replies to the watch the same way. From this point onwards, the watch can treat the connection as a regular TCP connection to the destination host. It typically continues to open a Transport Layer Security (TLS) session with the remote server, but would be free to continue with other protocols as well.

5 WatchWitch for Android

With the understanding of the protocol stack we gained from reverse-engineering, we can reimplement these protocols in the *WatchWitch* app to demonstrate that interoperability with Android is practically possible. As part of our reimplementation, we will also extend WatchWitch with several features designed to give users autonomy over their data and better privacy controls.

5.1 Design Goals

One explicit goal of our work is to show that meaningful interoperability between the Apple Watch and third-party smartphones is possible, despite Apple's claims to the contrary. To do so, we reimplement a usable selection of **essential smartwatch features**, including support for push notifications, Internet sharing, and access to health data. While full support of most or all watch features is far beyond the scope of this work, this already provides value to potential users and proves that there are no fundamental barriers to true interoperability. Our architecture also provides all the required infrastructure to allow technically adept users to implement support for any other services they require.

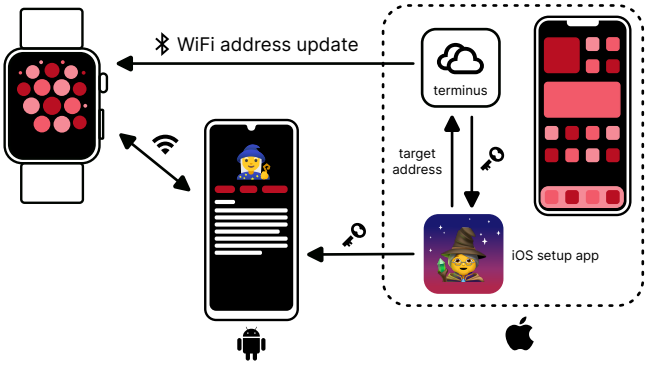

Figure 8: The WatchWitch app in context, showing the Apple Watch and the paired iPhone as well as the Android phone running the app.

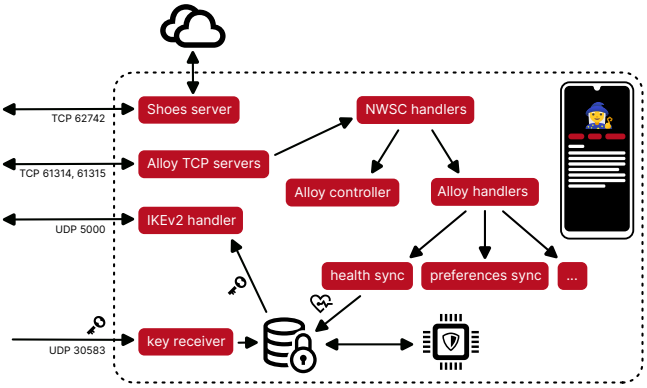

Figure 9: Overview of the internal components of the WatchWitch app, showing how incoming traffic is handled.

A common argument against interoperability are security concerns. The recent US antitrust complaint picks up on this, stating that *"Apple deploys privacy and security justifications as an elastic shield that can stretch or contract to serve Apple's financial and business interests"* [90, p. 12]. To show that interoperability does not necessarily sacrifice security, we therefore **maintain security** on a level comparable to the security provided by Apple out of the box. We do not circumvent or downgrade any protection mechanisms and complete all encryption and authentication steps as required by the respective protocols using secure key material. To ensure that long-term keys are stored with a level of protection comparable to the system keychain [5] used on iOS, we keep our cryptographic secrets in Android's hardware-backed KeyStore [1, 28].

Beyond providing an experience on par with what Apple offers for iOS users, we strive to **enhance privacy** in our implementation, allowing users to use their smartwatch entirely offline (without connecting to any commercial cloud servers) and putting them in explicit control of their data. We hope to demonstrate that in this way, open interfaces and interoperable devices do not just benefit people with different smartphones but all users across the spectrum in a sort of curb-cut effect. We want our implementation to **enable users** to use their devices to the fullest, going beyond the at times limited uses intended by the manufacturer.

5.2 Architecture

For WatchWitch, we focus on the Wi-Fi part of the Apple Watch communication infrastructure—since the underlying wireless link is transparent to application-level messages, this does not limit the available features. Using standard Wi-Fi makes it easier for us to interact with the watch from an Android app.[3]

Since the watch requires Bluetooth connectivity for the initial setup and Wi-Fi discovery, our setup (shown in Figure 8) includes a jailbroken iPhone that performs the watch setup and hands over communication to the Android phone by sending a Wi-Fi address update. After the setup is completed, we extract the cryptographic long-term keys from the phone and securely transfer them to the Android device using a custom app and tweak.[4] At this point, the iPhone is only required to perform the Wi-Fi discovery mechanism occasionally, e.g. after the watch is restarted. It is no longer part of the communication between the watch and the Android phone.

On the Android phone, we replicate the protocol handling we analyzed on the iPhone within our WatchWitch app (see Figure 9): The phone handles incoming IKEv2 messages to set up an IPSec tunnel, receives packets on that tunnel and forwards them to the Shoes or Alloy handling components. Internet-bound Shoes traffic is forwarded using the phone's network connection, while Alloy messages are handled locally and forwarded to service reimplementations registered for particular message topics.

Our Android app currently requires root access to the phone to set up the IPSec tunnel and associated routing. We discuss these limitations in more detail in Section 5.4.

5.3 Features

By providing implementations of the foundational protocols discussed in Section 4, we create a base layer to build support for various features. For the initial version of WatchWitch, we focus on three core smartwatch features: Receiving notifications, sharing Internet connectivity, and synchronizing health data.

5.3.1 Notification Forwarding & Message Replies for Android Apps. The push notifications displayed on the Apple Watch—most notably for instant messaging using iMessage and other apps—internally use a service referred to as *bulletin distributor*. The bulletin distributor on the iPhone sends ProtoBuf-encoded messages to the watch, keeping it informed about the state of the notifications present on the iPhone. These messages include information about the source and content of the notification, as well as the actions that can be taken in that context: Acknowledging, snoozing, or dismissing a message, as well as composing a reply on the watch.

The WatchWitch app can optionally register as a notification listener on Android. If the app then observes an incoming instant messaging notification (e.g. a Signal message), it generates a *bulletin request*, which is sent to the connected watch, instructing it to

[3] Real Bluetooth connectivity should also be possible using raw L2CAP sockets on Android. Work on extending WatchWitch to this use case is ongoing.

[4] We use Theos [87] and Cephei [47] to build a tweak and accompanying iOS app that modify the behavior of the terminus daemon and give us access to long-term keys.

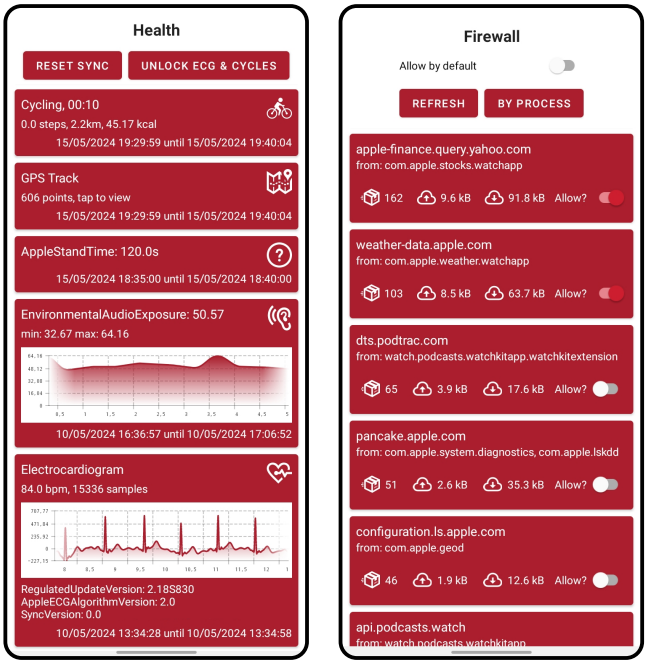

Figure 10: WatchWitch's health log and firewall views.

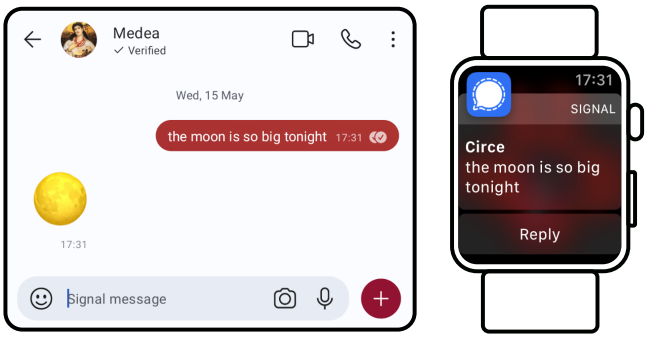

Figure 11: A Signal message sent to the Android phone is displayed on the watch, and the emoji reply sent from the watch received at the sender.

notify the user. While Apple famously does not allow third-party smartwatches to reply to incoming notifications [90, p. 40], we can use our knowledge of the bulletin distributor combined with Android's Notification Listener Service [3] to send messages with reply actions to the watch from our Android phone. Registering a notification listener requires explicit user consent but is possible for regular apps without root privileges. Responding to messages does not require any modifications to the app sending the notification but does rely on it supporting quick reply actions within its notifications. As a demonstration, we include support for receiving and replying to Signal messages in our WatchWitch app, as shown in Figure 11.

5.3.2 Internet Sharing & User-Controlled Firewall. WatchWitch includes an implementation of *Shoes* (Section 4.6) that responds to connection requests from the watch and forwards traffic using the phone's Internet connection. As the watch prefers to use the phone's connection even when it is connected to a Wi-Fi network, this effectively puts the app into a proxy position for the entirety of the watch's Internet-bound traffic.

With WatchWitch, we can use this position to show users which processes on the watch connect to which hosts, and how much data they transfer—be that for app data, logging, or tracking. Going one step further, we implement a user-controlled firewall that allows users to selectively block connections to certain hosts—or cut off apps from the Internet entirely. This makes it possible to, allow connections to the weather API required by the watch's built-in weather app, but block all traffic that could be detrimental to user privacy. With the amount of tracking present especially in third-party apps [18], this presents a powerful tool for users to make fine-grained trade-offs between privacy and functionality.

To see how our firewall is beneficial in a real-world setting, we test it with four watch apps that are, as of May 2024, prominently featured in Apple's watchOS app store: *SmartGym* [82], *Pedometer++* [31], *Carrot Weather* [21], and *Outcast* [32]. Both SmartGym and Pedometer++ attempt to connect to a remote server as we use them, but continue to function if we completely deny them Internet access. Carrot Weather needs a connection to its Weather API to work, but also connects to a second host for what we assume are tracking or analytics purposes. Using host-based blocking, we maintain the the app's functionality while blocking this secondary non-essential connection. We can use a similar approach with the podcast app Outcast: Besides some Apple services, the app connects to its own API server, which is required to add new podcast feeds. Once a podcast is added, however, we can block all of these connections, only allowing the app to reach the server hosting the actual podcast. This shows we can improve user privacy in the real world by *(a)* keeping some apps entirely offline and *(b)* only allowing essential hosts for other apps.

5.3.3 Health Data Synchronization. With what we learned about the ProtoBuf-based *NanoSync* protocol in Section 4.5, supporting health data synchronization in WatchWitch is relatively straightforward: We copy the database structure found on the iPhone in a dedicated SQLite database on the Android phone and insert new samples received from the watch using the same Structured Query Language (SQL) statements used in the health daemon on iOS.

On the iPhone, health database files are protected using the keychain's data protection, ensuring that they remain encrypted and inaccessible when the device is locked. To achieve a comparable level of security, we take inspiration from how Signal protects its message databases [80]: Using SQLCipher [96], we encrypt the database on the fly using key material protected by the Android KeyStore. Keys only remain in-memory while the app is running, giving us similar hardware-backed protection as on the iPhone.

WatchWitch can run arbitrary queries against the health database to surface sensor measurements, workouts, etc. to the user. Because we are not bound by the same legal constraints as Apple, we can expand the watch's functionality here: As discussed in Section 2.4, some features are only available in certain geographic regions. Apple enforces this by requiring users to manually enable these features while their iPhone is connected to the cellular network of an allowed country. With WatchWitch, we can generate

the required feature unlock messages ourselves and send them to the watch, regardless of our physical location. We demonstrate this for the ECG app, but the same approach should also translate to other features. It seems that researchers at Masimo—who are in a patent dispute with Apple over the watch's pulse oximeter—have already used a similar technique to show that watches sold in the US with a disabled oximeter can be reactivated [27].

Thus, WatchWitch allows us to use the watch's full capabilities as a health and fitness tracker with complete privacy—keeping an encrypted, near-perfect drop-in copy of the iPhone's health database on the phone without any data ever leaving the device. Even if Apple were to send health data directly to the cloud from the watch (which we have no evidence of in current versions), WatchWitch could still block these cloud services using the firewall feature, keeping health data completely on-device. With full database access, we can also give users access to data not usually available in Apple's UI, including long-term ambient noise levels and historic GPS tracks (see Figure 10). Detailed GPS tracks in particular are usually only stored for a set amount of time on the iPhone [65].

5.3.4 Further Features. The WatchWitch app supports receiving screenshots taken on the watch; it allows users to see which apps are currently running; which alarms are set; and if the watch is muted. Much of this state information is not easily accessible to the user on iPhones: We tap the watch's configuration backup mechanism for access, thus once again showing that we can meaningfully extend the features of the Apple Watch beyond what is intended by Apple and expand user's visibility into their devices.

To developers, WatchWitch provides a simple interface that allows them to add support for more services based on Alloy. This interface abstracts from the lower-level transport protocols, formats, and connection management—from a developer perspective, the new service can simply send and receive arbitrary Alloy messages.

Finally, our app stores full communication transcripts and file transfers received from the watch for later analysis. We provide an Alloy parser that can process these transcripts and may be used to debug or reverse-engineer Alloy communications with ease.

5.4 Limitations

In its current form, the setup required to use WatchWitch is rather complicated: Beyond a rooted Android phone and an Apple Watch, it also requires the presence of a jailbroken iPhone on the same Wi-Fi network to bootstrap the connection and does not work with a Bluetooth link. This setup is impractical for day-to-day use.

Allowing users to setup their watch without ever connecting to an iPhone would require a reimplementation of the Bluetooth-based pairing process, of which we do not yet have a sufficient understanding. As part of the pairing, both devices must obtain cryptographic certificates from Apple's servers, which Apple deliberately restricts to their own devices. Bypassing these restrictions, however, is not impossible [51].

Many of the complications in our setup stem from our position as unprivileged third-party developers. A party with deeper access on either side of the connection (i.e. Apple as manufacturer of the Apple Watch or Google as developer of Android) could make the interoperability process significantly simpler with only small changes to their products. In its current state, WatchWitch requires root access to instruct the kernel to decrypt incoming ESP packets. If Google were to extend its existing IPSec API [2] to support ESP in tunnel mode, this would no longer require root privileges. Alternatively, if Apple allowed User Datagram Protocol (UDP) encapsulation of ESP traffic on the watch, we could receive this traffic on Android without special privileges and perform decryption within the app. We could also circumvent the need for a rooted phone by reimplementing the initial pairing process, using the Bluetooth link for data transfer, and handling the entire IPSec setup ourselves within the app. The pairing process, however, is not well understood.

As a result of these barriers, the WatchWitch app that we showcase here is merely a proof of concept: Some features require 'stealing' a protocol session from the previously connected iPhone, which does not work with the reliability expected from a consumer application. With many different services that are not yet supported, the app may sometimes behave unexpectedly and fail to parse certain messages. The previously mentioned setup requiring multiple phones currently limits the usability of our app. We therefore opt to focus on WatchWitch's technical capabilities rather than providing a fully user-friendly drop-in replacement for Apple's native implementations—especially as such recreations of Apple's UI would raise copyright issues. We note, however, that the infrastructure provided by WatchWitch reduces the task of providing a usable replacement for the Health app, e.g., to a simple exercise in UI design, as would be the case for any regular Android app. Given all these challenges, the app runs remarkably well for long periods once connected to the watch. Our testing allowed performing workouts and mobile connections with a hotspot for over two hours. At home, we maintained an active connection for over 24 hours.

We are actively working to address these limitations as best as we can given the restrictions placed on us by Apple and Google as device manufacturers: In the future, we would like to extend WatchWitch to work on the Bluetooth layer, eliminating the need for an iPhone in the loop as well as a shared Wi-Fi network. Advances in this direction would also allow us to focus on real usability, as well as provide interoperability in the opposite direction: allowing the use of third-party smartwatches on iOS with the same level of deep integration as the Apple Watch.

6 Security Analysis

While reverse engineering the Apple Watch protocol stack, we have familiarized ourselves with its security architecture.

Overall, Apple's multi-layer encryption approach offers strong protection for all application traffic—the IPSec encryption layer alone should prevent most real-world attacks on the watch. This distinguishes the Apple Watch from other smartwatches that rely exclusively on link-layer encryption.

The parts of the protocol stack where Apple veers from established standards, however, are haunted by legacy support and questionable decisions: We find violations of common cryptographic practices, malleable encryption, and unintended interactions between standard and non-standard protocols. Beyond these issues, the protocol stack contains large amounts of complexity, much of it due to legacy versions and deprecated features. This presents a significant and previously unexplored attack surface.

6.1 Threat Model

We analyze the security of the Apple Watch in an everyday scenario, during which the watch is worn throughout the day in public. The watch connects to the phone to exchange messages over Bluetooth or shared Wi-Fi networks. We do not consider the setup of a new watch as part of this scenario, as this step is performed only once and likely happens in a private space, making it difficult for attackers to get into radio range at exactly the right time. The initial Bluetooth pairing of the Apple Watch has been briefly analyzed in prior research and follows best practices [81].

Our focus lies on Apple's proprietary protocols, as these parts distinguish the Apple Watch from other devices. In this section we consider Dolev-Yao attackers [37] with the ability to read, modify, and inject messages at will. We assume that all attackers are within radio range of both devices and are members of the same wireless network. With these assumptions, attackers attempt to interfere with the Apple Watch with one of the following goals:

Confidentiality: Gain access to private data, especially sensitive health information.

Integrity: Place or manipulate data on the watch or phone.

Availability: Disrupt communication between watch and phone or delete data on either device.

A peculiarity of the Apple Watch's protocol stack is its highly layered nature: The Wi-Fi or Bluetooth transport layer *may* already provide some security, the IPSec tunnels provide a first and tested layer of encryption, and the A-over-C layer provides another additional encryption layer. With a simple network attacker, we could not analyze these nested layers independently. We therefore simulate attackers that have already broken some of the protection layers by giving them access to the corresponding keys. While this key-access is artificial, these attackers still model plausible real-world attacks: For the Wi-Fi / Bluetooth layer (Attacker 1), this matches the capabilities of someone with access to a Bluetooth or Wi-Fi vulnerability, or someone with access to the same shared Wi-Fi network. Attackers able to bypass the IPSec layer (Attacker 2) might be able to do so after gaining physical access to a locked phone and extracting parts of its memory [30].

Attacker 1. This attacker models a broken transport layer, which is plausible since there have been many generic attacks against Wi-Fi and Bluetooth in the past [41, 92, 93], as well as Apple-specific security issues [48, 77, 95]. Furthermore, transport layer encryption does not provide any protection against attackers if the devices communicate over a Wi-Fi network that the attacker can join, such as open Wi-Fis or Wi-Fis with a static WPA2 passphrase. The main attack surface for Attacker 1 is the IPSec layer: Extracting or manipulating data requires circumventing the encryption and authentication provided by the VPN tunnel. We assume IPSec in its well-known standard form is secure as it has been formally analyzed [52] and the chosen cryptographic primitives are secure (see Appendix A.4). We focus on the security implications of Apple's deviations from the standard in Section 6.2.

Attacker 2. This attacker has access to IPSec key material, giving them 'legitimate' access to most of the watch's communication. Such a severe break in the IPSec security layer is unlikely—but this attacker corresponds to a scenario where adversaries briefly gain physical access to the iPhone while it is locked—such as in a border control setting. In this state, class C and D key material is available and could be extracted. Companies like Cellebrite or Magnet Forensics offer such services commercially [30, 63]. Key material for class A, however, should remain secure, and attackers should not be able to decrypt or modify traffic protected with these keys. Attacker 2, having access to the IPSec tunnel, already has a large amount of control over the system. For example, it can disrupt communication by continuously resetting the watch. The only remaining trust boundary for this attacker is gaining access to class-A protected data. The main attack surface is the A-over-C protocol, which we analyze in Section 6.3.

6.2 Insecure IKEv2 Extensions

Where Apple extends the proven IKEv2 standard with their own LDM protocol, they fail to account for the fact that such payloads may be included in unencrypted, unauthenticated IKEv2 messages. As the first messages of any IKEv2 handshake are—by necessity—unencrypted, the IKEv2 standard allows *notify* payloads in unencrypted contexts. While Apple only uses custom notify payloads once encryption is established, they never check if the payload was received in an encrypted message. They also continue to accept unencrypted messages after encryption is successfully established.

Thus, an attacker with the ability to inject Bluetooth or Wi-Fi packets (Attacker 1) can send forged Link Director Messages—for example to manipulate or jam the Wi-Fi discovery mechanism discussed in Section 4.2, as shown in Figure 12. While this vulnerability allows attackers to redirect watch traffic to an attacker-controlled device, it does not give them the ability to interact meaningfully with the watch without the required cryptographic keys. Nonetheless, this oversight by Apple opens up all of the complexity of their custom notify payloads to unauthenticated attackers—a similar attack could, for example, redirect Shoes proxy traffic to unintended or malicious destinations. This could be avoided by only accepting custom notify payloads in encrypted and authenticated contexts.

6.3 Forging Health Values in A-over-C

The A-over-C protocol (Section 4.4) uses the same cryptography as iMessage, which has been shown to be vulnerable in the past and has historically employed short key lengths and obscure protocol design [44]. Beyond that, A-over-C awkwardly composes iMessage-based encryption with other primitives, which leaves the payload data entirely unauthenticated—data carried by A-over-C is encrypted using only unauthenticated AES-CBC encryption (see Section 4.4). A drawback of the CBC mode of operation is its malleability: If an attacker flips bits in a ciphertext block c_1 (creating $c_1' = c_1 \oplus x$), this will cause the corresponding plaintext block p_1' to be corrupted and essentially random. However, the *following* ciphertext block c_2 will decrypt to a plaintext that reflects the bit flips from the previous block: $p_2' = p_2 \oplus x$.

We show that an attacker with access to A-over-C ciphertexts (Attacker 2) and partial knowledge of their plaintext content can use this property to change the type of transferred health samples, inserting forged values into the health database. This exploit relies on the 16-byte UUIDs present in transferred health samples: When this UUID aligns with the blocks of the block cipher, we can flip

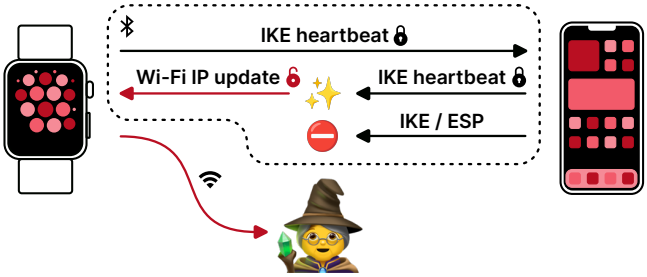

Figure 12: Attacking the Apple Watch by replacing an IKEv2 heartbeat with a forged, unencrypted Wi-Fi IP address update and redirecting Wi-Fi traffic to the attacker.

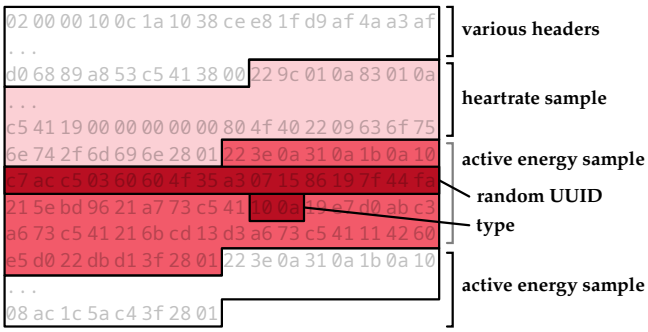

Figure 13: Anatomy of a vulnerable A-over-C message in plaintext. Each row corresponds to 16-byte cipher blocks.

bits in the ciphertext block containing it. The modified block will then decrypt to random bytes which, crucially, still form a valid random UUID. We can then control the type byte of the health sample located in the following block—for example changing a heart rate into a step count. An illustrated example of this attack, performed on a real-world A-over-C health sync message, is shown in Figure 13. Concretely, using CBC malleability to flip the last four bits of the type field will make the receiver interpret the modified *active energy* sample (0x0a) as a *heartrate* sample (0x05), inserting the forged heart rate measurement into the database.

Notably, this behavior appears untouched by recent changes in iMessage cryptography [11]: Comparing our findings based on iOS 14.8 to the recent iOS 17.5.1, we find that A-over-C cryptography remains unchanged. Fortunately, the fix for this issue is simple: Apple could use the MessageProtection framework to encrypt the entire message rather than just encapsulate the key material, or use an authenticated encryption algorithm for the payload encryption.

6.4 Health Data Deletion & Cycle Tracking

While not directly related to the Apple Watch, we also notice how health samples, most prominently symptoms logged in the cycle tracking app on either watch or phone, are—or rather are not—deleted. Some thought went into the security of this: Samples are not deleted using a basic SQL delete query, which might leave artifacts in the database file, but are manually marked as deleted and have most fields overwritten with null values. This removes, for example, the actual measurement contained in a heart rate sample. What stays, however, is the type of the sample and its deletion time.

In the context of the cycle tracking app, this means that attackers can discover the number of deleted entries, their type, as well as the time they were deleted. While the database gives no information about the order or timing of these entries, this may still be troubling for users who have their devices seized by law enforcement as we briefly discussed in Section 2.2: A set of symptoms associated with pregnancy that were deleted just before a device was seized might certainly look incriminating when being charged with an unlawful abortion. With no particular reason to keep deleted symptoms in the database, we do not see why these entries could not be overwritten and deleted entirely, thus avoiding this issue.

7 Lessons for Secure Smartwatches

At its core, the Apple Watch's security lies in **defense in depth** and strong, **standard protocols**. The watch does not have to rely on the varied security guarantees of lower-level transports, and the overall architecture remains largely secure even in the presence of the issues we discovered. Conversely, the security weaknesses we found stemmed from **proprietary protocol extensions** and **nonstandard cryptography**.

Based on these observations, we argue that a VPN tunnel using well-understood security protocols can and should form the basis for future smartwatches. This tunnel can serve as a strong first protection layer that isolates higher-level logic from the lower-level transports and shields it from most network attackers. Any deviations from such standards, however, should be very carefully considered as they might have non-obvious security implications. Especially in cryptographic protocols, even small changes can have dire consequences. If non-standard cryptography cannot be avoided, it should therefore provide mechanisms for future flexibility: To keep systems secure in the long term, it is important to be able to change algorithms, update parameters, or rotate keys. IKEv2, for example, provides this flexibility through algorithm negotiation. Apple makes use of this, tweaking the preferred cipher suites from version to version. In the custom A-over-C protocol, on the other hand, parameters are hard-coded, and any change would break compatibility with prior versions—meaning that Apple is stuck with weak, unauthenticated cryptography for the foreseeable future.

7.1 Other Smartwatch Architectures

When looking at wearable devices, we can distinguish two fundamental architectures that differ significantly in their privacy and security properties (see Figure 14): *Local-first* devices communicate directly with a paired phone, and only involve remote servers for features that require explicit cloud synchronization. *Cloud-first* devices only meaningfully communicate with a remote server, with the paired phone acting as a proxy that forwards messages to the cloud without the ability to perform any local processing.

The Apple Watch is a local-first device: Messages are always handled locally on the connected phone, and in most cases, no external server is involved at all. For messages that require a cloud service—such as Internet sharing or instant messaging—the phone forwards relevant payloads to a destination server. Notably, data is **end-to-end encrypted** between the watch and the phone.

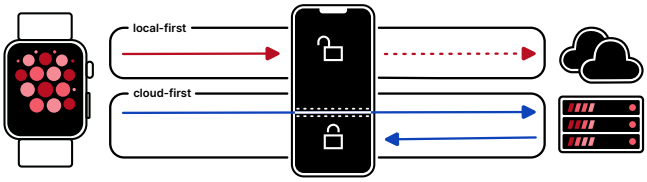

Figure 14: Local-first (top) versus cloud-first (bottom) smartwatch architectures.

Watches from Google-owned competitor Fitbit follow a cloud-first approach: Their devices establish an encrypted connection to Fitbit's servers using a factory-set key and a custom protocol [26]. We confirm that the same architecture is present in the latest Google Pixel Watch. The phone still receives and forwards messages, but is not able to see their contents. Collected data is sent to the server using this encrypted connection before the server returns aggregated information to the phone.

While cloud-first devices offer better protection against local attackers by using pre-shared keys embedded in the hardware and communicating directly with a trusted server, this same server is also the main downside: A cloud-first scheme inherently places trust in a third party operating the server and cannot provide true end-to-end encryption between the user's devices. The third party will typically have access to plaintext user health data and extensive connection metadata. We therefore argue that a local-first model is preferable for user privacy and security as it facilitates end-to-end encryption, reduces metadata, and remains functional while offline.

8 Related Work

To the best of our knowledge, this is the first work to publicly reverse-engineer and document the communication protocols used by the Apple Watch, and the first demonstration of meaningful interoperability between the Apple Watch and an Android phone.

There is, however, prior work investigating the communication mechanisms of other fitness trackers, including devices from Fitbit [26, 40] and Xiaomi [22]. Classen et al. [26] in particular provide a full reimplementation of the Fitbit protocol stack alongside a custom app [79] that can receive fitness data from the tracker, similar to the custom app we develop to interact with the Apple Watch.

Early versions of Android Wear smartwatches were analyzed for security, especially with regard to physical access attacks [36]. Similar scenarios of data extraction given physical access have been studied in the context of digital forensics, including case studies of smartwatches from Samsung and LG [16] as well as Fitbit and Garmin [62]. Health data collected by the Apple Watch has also been subject to forensics research; however, only synchronized data on the paired iPhone was considered [65].

Some of the lower-level protocols for phone-to-watch communication are also used with other Apple devices and have been analyzed or described in these contexts. The *Magnet* protocol and its message structures have been described by Heinze et al. [48]. The same paper also identifies the *terminus* daemon as closely related to Apple Watch communication. The *CLink* protocol, which we observe during early connection establishment over Bluetooth, also shows up in the work of Stute et al. [84] as *Pair-Verify* and appears to be used with many Apple peripherals. Neither protocol, however, is of significant relevance for high-level watch communication.

Outside of academia, the Apple Watch has long attracted the attention of the jailbreaking community: Despite the lack of jailbreaks for modern watchOS versions, there is a variety of *tweaks* for modifying the watch's behavior [20, 60, 72, 89]. As it is not currently possible to execute custom code not signed by Apple on modern watches,[5] these tweaks operate entirely on the paired (jailbroken) iPhone. Most of these tweaks are only cosmetic modifications to the notification logic on the iPhone, such as the *WinterMode* tweak that makes notifications alert the user on both the watch and the phone, as opposed to the watch only [72]. Some tweaks, however, hook into and modify the communication between watch and phone. *WatchMuteMirror* [20], for example, silences the iPhone when the user silences the watch, presumably listening for messages used to backup the watch's state to the phone to do so.

The most advanced of these tweaks is *Legizmo* [60, 61], which dramatically expands the version compatibility between iOS and watchOS beyond what is supported by Apple. This allows users to pair phones running older iOS versions with newer watchOS versions and vice versa. Legizmo developer *lunotech11* told us that the tweak is based on extensive reverse engineering of the Apple Watch's communication protocols, bridging gaps and translating between different versions where necessary. These compatibility efforts go so far that Legizmo patches older apps to backport features only added in newer iOS versions, such as the advanced sleep tracking introduced in watchOS 9 and iOS 16. The tweak remains closed-source and we did not receive access for this work.

9 Conclusion

Our work on WatchWitch shows that true interoperability between the Apple Watch and third-party Android devices is feasible despite prior contrary claims: We have reimplemented several essential smartwatch features on Android, including push notifications, Internet sharing, and health data synchronization. We have also shown that we can achieve this level of interoperability while maintaining security—employing the same cryptographic protocols and storing keys and data with comparable hardware-backed security.

Going beyond interoperability, we have seen how opening the Apple Watch ecosystem to open-source implementations can benefit users by offering better privacy, more complete access to data, and even entirely new features such as a fine-grained firewall.

Our research makes the security and privacy properties of the Apple Watch visible and presents a way towards autonomy and independence, allowing users to use their devices on their own terms and beyond the manufacturer's intentions. We look forward to seeing researchers, tinkerers, and manufacturers build upon our work—be it in terms of alternative software, hardware, or entirely new applications.

[5] The only publicly available watchOS jailbreak [88] targets the Apple Watch Series 3 running watchOS 4.1 and is, therefore, about seven years behind modern watch models.

Acknowledgments

With thanks to Apple for working with us throughout the disclosure process in fixing the vulnerabilities we discovered. We also thank lunotech11 for insights on some of the Apple Watch's peculiarities. This work has been funded by the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research, and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Android Open Source Project. 2023. *Android Keystore system.* Retrieved March 27, 2024 from https://developer.android.com/privacy-and-security/keystore

[2] Android Open Source Project. 2023. *IpSecManager.* Retrieved May 13, 2024 from https://developer.android.com/reference/android/net/IpSecManager

[3] Android Open Source Project. 2024. *NotificationListenerService.* Retrieved May 13, 2024 from https://developer.android.com/reference/android/service/notification/NotificationListenerService

[4] Apple Inc. 2021. *Data Protection classes.* Retrieved April 10, 2024 from https://support.apple.com/guide/security/data-protection-classes-secb010e978a/web

[5] Apple Inc. 2021. *Keychain Data Protection.* Retrieved March 27, 2024 from https://support.apple.com/guide/security/secb0694df1a

[6] Apple Inc. 2023. *Apple Watch Ultra 2.* Retrieved April 2, 2024 from https://www.apple.com/apple-watch-ultra-2/

[7] Apple Inc. 2023. *Health Privacy Overview.* https://www.apple.com/ios/health/pdf/Health_Privacy_White_Paper_May_2023.pdf

[8] Apple Inc. 2024. *About the security content of watchOS 10.3.* Retrieved May 15, 2024 from https://support.apple.com/en-us/HT214060

[9] Apple Inc. 2024. *Bluetooth.* Retrieved May 22, 2024 from https://developer.apple.com/bluetooth/

[10] Apple Inc. 2024. *Hand off tasks from Apple Watch.* Retrieved May 15, 2024 from https://support.apple.com/guide/watch/hand-off-tasks-from-apple-watch-apdc40081790/watchos

[11] Apple Inc. 2024. *iMessage with PQ3: The new state of the art in quantum-secure messaging at scale.* Retrieved May 23, 2024 from https://security.apple.com/blog/imessage-pq3/

[12] Apple Inc. 2024. *Track your sleep with Apple Watch.* Retrieved May 21, 2024 from https://support.apple.com/guide/watch/track-your-sleep-apd830528336/10.0/watchos/10.0

[13] Apple Inc. 2024. *Track your sleep with Apple Watch.* Retrieved May 23, 2024 from https://www.apple.com/watchos/feature-availability/

[14] Apple Inc. 2024. *Use Camera Remote and timer on Apple Watch.* Retrieved May 15, 2024 from https://support.apple.com/guide/watch/camera-remote-apda6e61c287/watchos

[15] The AsteroidOS Project. 2024. *Free your wrist - AsteroidOS.* Retrieved April 2, 2024 from https://asteroidos.org/

[16] Ibrahim M. Baggili, Jeff Oduro, Kyle Anthony, Frank Breitinger, and Glenn McGee. 2015. Watch What You Wear: Preliminary Forensic Analysis of Smart Watches. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015.* IEEE Computer Society, 303–311. https://doi.org/10.1109/ARES.2015.39

[17] Elaine Barker and Quynh Dang. 2015. *Recommendation for Key Management - Application-Specific Key Management Guidance.* Technical Report NIST Special Publication (SP) 800-57 Part 3, Rev. 1. National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.SP.800-57pt3r1

[18] Reuben Binns, Ulrik Lyngs, Max Van Kleek, Jun Zhao, Timothy Libert, and Nigel Shadbolt. 2018. Third Party Tracking in the Mobile Ecosystem. In *Proceedings of the 10th ACM Conference on Web Science* (Amsterdam, Netherlands) *(WebSci '18).* Association for Computing Machinery, New York, NY, USA, 23–31. https://doi.org/10.1145/3201064.3201089

[19] R. Braden, D. Borman, and C. Partridge. 1988. Computing the Internet checksum. RFC 1071. https://doi.org/10.17487/RFC1071

[20] CardboardFace. 2022. *WatchMuteMirror.* https://havoc.app/package/watchmutemirror

[21] CARROT. 2024. *CARROT Weather for Apple Watch.* Retrieved May 23, 2024 from https://www.meetcarrot.com/weather/applewatch.html

[22] Marco Casagrande, Eleonora Losiouk, Mauro Conti, Mathias Payer, and Daniele Antonioli. 2022. BreakMi: Reversing, Exploiting and Fixing Xiaomi Fitness Tracking Ecosystem. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 3 (2022), 330–366. https://doi.org/10.46586/TCHES.V2022.I3.330-366

[23] Julian Chokkattu. 2022. *Google's Long-Awaited Pixel Watch Is Finally Here.* https://www.wired.com/story/google-pixel-watch-features-release-date-price/

[24] Jiska Classen. 2024. *Frida Scripts for iOS.* Retrieved May 22, 2024 from https://github.com/seemoo-lab/frida-scripts

[25] Jiska Classen, Alexander Heinrich, Robert Reith, and Matthias Hollick. 2022. Evil Never Sleeps: When Wireless Malware Stays On after Turning Off iPhones. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (San Antonio, TX, USA) *(WiSec '22).* Association for Computing Machinery, New York, NY, USA, 146–156. https://doi.org/10.1145/3507657.3528547

[26] Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. 2018. Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 1 (2018), 5:1–5:24. https://doi.org/10.1145/3191737

[27] Juli Clover. 2024. *Apple Watch Pulse Oximetry Can Be Reactivated Through Software in 2028 or With Successful Appeal.* Retrieved May 23, 2024 from https://www.macrumors.com/2024/03/12/apple-watch-blood-oxygen-sensor-software/

[28] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. 2014. Analysis of Secure Key Storage Solutions on Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (Scottsdale, Arizona, USA) *(SPSM '14).* Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/2666620.2666627

[29] Counterpoint. 2023. *Global Smartwatch Market Rebounds; Huawei and Fire-Boltt Hit New Peaks.* https://counterpointresearch.com/insights/global-smartwatch-market-rebounds-huawei-and-fire-boltt-hit-new-peaks/

[30] Joseph Cox. 2024. Leaked Docs Show What Phones Cellebrite Can (and Can't) Unlock. Retrieved August 14, 2024 from https://www.404media.co/leaked-docs-show-what-phones-cellebrite-can-and-cant-unlock/

[31] Cross Forward Consulting LLC. 2024. *Pedometer++.* Retrieved May 23, 2024 from https://pedometer.app/

[32] Crunchy Bagel Pty Ltd. 2018. *outcast - The podcast player for Apple Watch.* Retrieved May 23, 2024 from https://outcastapp.com/

[33] Phoebe Davis. 2023. *British police testing women for abortion drugs.* https://www.tortoisemedia.com/2023/10/30/british-police-testing-women-for-abortion-drugs/

[34] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G. Paterson. 2021. The Security of ChaCha20-Poly1305 in the Multi-User Setting. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21).* Association for Computing Machinery, New York, NY, USA, 1981–2003. https://doi.org/10.1145/3460120.3484814

[35] L. Peter Deutsch. 1996. GZIP file format specification version 4.3. RFC 1952. https://doi.org/10.17487/RFC1952

[36] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. 2017. Is the data on your wearable device secure? An Android Wear smartwatch case study. *Softw. Pract. Exp.* 47, 3 (2017), 391–403. https://doi.org/10.1002/SPE.2414

[37] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory* 29, 2 (1983), 198–207. https://doi.org/10.1109/TIT.1983.1056650

[38] European Parliament. 2022. *Deal on Digital Markets Act: EU rules to ensure fair competition and more choice for users.* Retrieved May 23, 2024 from https://www.europarl.europa.eu/news/en/press-room/20220315IPR25504/deal-on-digital-markets-act-ensuring-fair-competition-and-more-choice-for-users

[39] Matt Evans. 2023. Where is all your health data going? The Google and Fitbit scandal explained. Retrieved March 28, 2024 from https://www.techradar.com/health-fitness/fitness-trackers/google-and-fitbit-scandal-explained

[40] Hossein Fereidooni, Jiska Classen, Tom Spink, Paul Patras, Markus Miettinen, Ahmad-Reza Sadeghi, Matthias Hollick, and Mauro Conti. 2017. Breaking Fitness Records Without Moving: Reverse Engineering and Spoofing Fitbit. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10453),* Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer, 48–69. https://doi.org/10.1007/978-3-319-66332-6_3

[41] Marc Fischlin and Olga Sanina. 2021. Cryptographic Analysis of the Bluetooth Secure Connection Protocol Suite. In *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13091),* Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, 696–725. https://doi.org/10.1007/978-3-030-92075-3_24

[42] Lars Fröder, kok3shidoll, and Évelyne. 2023. *Dopamine Jailbreak.* Retrieved May 21, 2024 from https://ellekit.space/dopamine/

[43] David E. Fu and Jerome Solinas. 2007. *ECP Groups For IKE and IKEv2.* Request for Comments RFC 4753. Internet Engineering Task Force. https://doi.org/10.17487/RFC4753

[44] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. 2016. Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016,* Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 655–672. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/garman

[45] Google LLC. 2024. *Protocol Buffers.* Retrieved May 5, 2024 from https://protobuf.dev/

[46] Scharon Harding. 2024. *Apple Watch no longer sold with blood oxygen monitoring after patent battle loss.* Retrieved April 2, 2024 from https://arstechnica.com/gadgets/2024/01/apple-watch-no-longer-sold-with-blood-oxygen-monitoring-after-patent-battle-loss/

[47] HASHBANG Productions. 2021. *Cephei Reference.* Retrieved April 23, 2024 from https://hbang.github.io/libcephei/

[48] Dennis Heinze, Jiska Classen, and Matthias Hollick. 2020. ToothPicker: Apple Picking in the iOS Bluetooth Stack. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/heinze

[49] Dennis Heinze, Jiska Classen, and Felix Rohrbach. 2020. MagicPairing: Apple's take on securing Bluetooth peripherals. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Linz, Austria) *(WiSec '20)*. Association for Computing Machinery, New York, NY, USA, 111–121. https://doi.org/10.1145/3395351.3399343

[50] IANA. 2023. *Internet Key Exchange Version 2 (IKEv2) Parameters.* Retrieved August 14, 2024 from https://www.iana.org/assignments/ikev2-parameters/ikev2-parameters.xhtml

[51] JJTech. 2023. *iMessage, explained.* Retrieved May 24, 2024 from https://jjtech.dev/reverse-engineering/imessage-explained/

[52] Daniel Jost. 2014. *A constructive analysis of IPsec.* Master's thesis. ETH-Zürich.

[53] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. 2014. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296. https://doi.org/10.17487/RFC7296

[54] Stephen Kent. 2005. IP Encapsulating Security Payload (ESP). RFC 4303. https://doi.org/10.17487/RFC4303

[55] Mika Kojo and Tero Kivinen. 2003. *More Modular Exponential (MODP) Diffie-Hellman Groups for Internet Key Exchange (IKE).* Request for Comments RFC 3526. Internet Engineering Task Force. https://doi.org/10.17487/RFC3526

[56] Jakob Krantz. 2024. *ZSWatch.* Retrieved March 27, 2024 from https://github.com/jakkra/ZSWatch

[57] Tobias Kröll, Stephan Kleber, Frank Kargl, Matthias Hollick, and Jiska Classen. 2021. ARIstoteles – Dissecting Apple's Baseband Interface. In *Computer Security – ESORICS 2021*, Elisa Bertino, Haya Shulman, and Michael Waidner (Eds.). Springer International Publishing, Cham, 133–151.

[58] Adam Langley, Mike Hamburg, and Sean Turner. 2016. *Elliptic Curves for Security.* Request for Comments RFC 7748. Internet Engineering Task Force. https://doi.org/10.17487/RFC7748

[59] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. 1996. *SOCKS Protocol Version 5.* Request for Comments RFC 1928. Internet Engineering Task Force. https://doi.org/10.17487/RFC1928

[60] lunotech11. 2022. *Legizmo 'Jupiter' (watchOS 6-8).* https://chariz.com/buy/legizmo-jupiter

[61] lunotech11. 2023. *Legizmo.* https://legizmo.app

[62] Áine MacDermott, Stephen Lea, Farkhund Iqbal, Ibrahim Idowu, and Babar Shah. 2019. Forensic Analysis of Wearable Devices: Fitbit, Garmin and HETP Watches. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–6. https://doi.org/10.1109/NTMS.2019.8763834

[63] Magnet Forensics. 2024. *Magnet Graykey.* https://www.magnetforensics.com/products/magnet-graykey/

[64] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Trans. Priv. Secur.* 24, 3, Article 19 (apr 2021), 35 pages. https://doi.org/10.1145/3448609

[65] James McGee. 2023. Enriching Investigations with Apple Watch Data Through the healthdb_secure.sqlite Database. *DFIR Review* (2023). https://dfir.pubpub.org/pub/xqvcn3hj

[66] Axel Metz. 2023. *Exclusive: Samsung explains why it dropped iOS support on the Galaxy Watch.* https://www.techradar.com/health-fitness/smartwatches/exclusive-samsung-explains-why-it-dropped-ios-support-on-the-galaxy-watch

[67] Chance Miller. 2024. Apple says it spent three years trying to bring Apple Watch to Android. Retrieved March 27, 2024 from https://9to5mac.com/2024/03/21/apple-watch-android-apple-work/

[68] Yoav Nir and Simon Josefsson. 2016. *Curve25519 and Curve448 for the Internet Key Exchange Protocol Version 2 (IKEv2) Key Agreement.* Request for Comments RFC 8031. Internet Engineering Task Force. https://doi.org/10.17487/RFC8031

[69] Yuichi Niwa, Keisuke Ohashi, Kazuhiko Minematsu, and Tetsu Iwata. 2015. GCM Security Bounds Reconsidered. In *Fast Software Encryption*, Gregor Leander (Ed.). Springer, Berlin, Heidelberg, 385–407. https://doi.org/10.1007/978-3-662-48116-5_19

[70] Charlie Osborne. 2021. Over 60 million wearable, fitness tracking records exposed via unsecured database. Retrieved April 2, 2024 from https://www.zdnet.com/article/over-60-million-records-exposed-in-wearable-fitness-tracking-data-breach-via-unsecured-database/

[71] palera1n team. 2024. *palera1n - Jailbreak for A8 through A11 devices, on iOS 15.0 and higher.* Retrieved May 21, 2024 from https://palera.in/

[72] Ayden Panhuyzen. 2020. *WinterMode.* https://github.com/aydenp/WinterMode

[73] Paige Papandrea. 2019. Addressing the HIPAA-potamus sized gap in wearable technology regulation. *Minn. L. Rev.* 104 (2019), 1095.

[74] PINE64. 2024. *PineTime.* https://pine64.org/devices/pinetime/

[75] Ole André V. Ravnås. 2017. *Frida • A world-class dynamic instrumentation toolkit.* Retrieved May 22, 2024 from https://frida.re/

[76] Thomas Roth, Fabian Freyer, Matthias Hollick, and Jiska Classen. 2022. AirTag of the Clones: Shenanigans with Liberated Item Finders. In *2022 IEEE Security and Privacy Workshops (SPW)*. 301–311. https://doi.org/10.1109/SPW54247.2022.9833881

[77] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. 2020. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 19–36. https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

[78] Samsung Inc. 2023. *Galaxy Watch6.* Retrieved April 2, 2024 from https://www.samsung.com/us/watches/galaxy-watch6/#desc-section

[79] SEEMOO Lab. 2019. *Fitbit Open Source Android App.* https://github.com/seemoo-lab/fitness-app

[80] Signal. 2024. *Database Secret Provider.* Retrieved May 22, 2024 from https://github.com/signalapp/Signal-Android/blob/main/app/src/main/java/org/thoughtcrime/securesms/crypto/DatabaseSecretProvider.java

[81] Alejandra Guadalupe Silva-Trujillo, Mauricio Jacobo González González, Luis Pablo Rocha Pérez, and Luis Javier García Villalba. 2023. Cybersecurity Analysis of Wearable Devices: Smartwatches Passive Attack. *Sensors* 23, 12 (2023). https://doi.org/10.3390/s23125438

[82] SmartGym. 2023. *SmartGym - Apple Watch App.* Retrieved May 23, 2024 from https://smartgymapp.com/watch

[83] Paul Smith. 2024. *Open-SmartWatch.* Retrieved March 27, 2024 from https://open-smartwatch.github.io/

[84] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. 2021. Disrupting Continuity of Apple's Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3917–3934. https://www.usenix.org/conference/usenixsecurity21/presentation/stute

[85] Milan Stute, David Kreitschmann, and Matthias Hollick. 2018. One Billion Apples' Secret Sauce: Recipe for the *Apple Wireless Direct Link* Ad hoc Protocol. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, MobiCom 2018, New Delhi, India, October 29 - November 02, 2018*, Rajeev Shorey, Rohan Murty, Yingying (Jennifer) Chen, and Kyle Jamieson (Eds.). ACM, 529–543. https://doi.org/10.1145/3241539.3241566

[86] The Tcpdump Group. 2024. *Tcpdump & Libpcap.* Retrieved May 22, 2024 from https://www.tcpdump.org/

[87] Theos Community. 2021. *Documentation Home.* Retrieved April 23, 2024 from https://theos.dev/docs/

[88] tihmstar. 2018. *jelbrekTime.* Retrieved April 04, 2024 from https://github.com/tihmstar/jelbrekTime

[89] udevsharold. 2021. *NanoFi.* https://github.com/udevsharold/nanofi

[90] United States of America et al v. Apple Inc., No. 2:24-cv-04055. 2024. Complaint. (D.N.J. Mar. 21, 2024). Retrieved March 27, 2024 from https://www.justice.gov/opa/media/1344546/dl

[91] U.S. Department of Health and Human Services. 2022. *Summary of the HIPAA Privacy Rule.* Retrieved May 21, 2024 from https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html

[92] Mathy Vanhoef and Frank Piessens. 2017. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM.

[93] Mathy Vanhoef and Eyal Ronen. 2020. Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 517–533. https://doi.org/10.1109/SP40000.2020.00031

[94] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS .... In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2332)*, Lars R. Knudsen (Ed.). Springer, 534–546. https://doi.org/10.1007/3-540-46035-7_35

[95] Yu Wang. 2022. *Dive into Apple IO80211FamilyV2 Vol. II.* Retrieved August 21, 2024 from https://i.blackhat.com/USA-22/Wednesday/US-22-Wang-Dive-into-Apple-IO80211Family-Vol-II.pdf

[96] Zetetic LLC. 2024. *SQLCipher - Full Database Encryption for SQLite.* Retrieved April 22, 2024 from https://www.zetetic.net/sqlcipher/

A Appendix

A.1 Magnet

The basic packet structure of Magnet has already been described by Heinze et al. [48]—we take a closer look at the Magnet handling logic in the Bluetooth daemon and identify the supported message types, shown in Table 2. References to the debug string `"Received %d remote services from the remote master %p !"` may be used to find the main Magnet handling function in our iOS version (14.8) and likely many other versions as well.

Table 2: Magnet opcodes and message types.

opcode	meaning
0x01	remote services
0x02	remote services response
0x03	create channel for service
0x04	accept channel for service
0x05	service added
0x06	service removed
0x07	service removed acknowledge
0x08	error response
0x09	version info
0x70	send time sync correction
0x71	time data
0x72	time data
0x90	DID info
0x91	CL data

A.2 NRLP

NRLP is handled in the terminus daemon. As of iOS 14.8, the main parsing is performed in the `NRLinkBluetooth:handleReadData` function. NRLP packets may be fragmented across several L2CAP frames. Every L2CAP frame also contains a *sequence number* and *packets received* byte before the actual NRLP data. As these bytes are also present for non-NRLP traffic including *CLink* and *BT.TS*, we do not consider them to be part of NRLP.

We list the supported payload types of NRLP in Table 3. Of these, we only see ESP and IKEv2 related types in active use, with occasional Encapsulated6LoWPAN packets appearing as well. The echo service replies to *ping* messages starting with the byte 0x01 with an identical *pong* message starting with 0x02, but does not appear to be used.

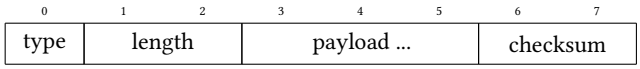

Figure 15: NRLP message byte format.

The calculation of the checksum present at the end of each NRLP packet (see Figure 15) differs between message types. For packet types above 0x63—i.e. all ESP and TCP types—the checksum is a common Internet Checksum as described in RFC 1071 [19]. For other messages, the checksum only covers the type and length header fields. The high and low byte of the checksum is calculated in this case as follows:

$$checksum_{high} = length_{high} \oplus (type \gg 4)$$
$$checksum_{low} = length_{low} \oplus (type \ll 4)$$

Table 3: NRLP message types. ECT0 refers to the Explicit Congestion Notification flag of the Internet Protocol.

Type	Meaning
0x00	Pad0 / noop
0x01	PadN / noop
0x02	UncompressedIP
0x03	Encapsulated6LoWPAN
0x04	IKEv2
0x05	echo service
0x64	ESP
0x65	ESP_ECT0
0x66	TCP
0x67	TCP_ECT0
0x68	ESP_ClassC
0x69	ESP_ClassC_ECT0

A.3 IKEv2 Custom Notify Payloads

As of iOS 14.8, the IKEv2 handshake with the Apple Watch is also handled by the terminus daemon. Handling of the private notify payloads in particular is performed in the `NRLinkBluetooth::↩ handleNotifyCode:payload:` method.

Beyond the LDMs used for Wi-Fi discovery, Apple uses a variety of other private notify payloads for signaling purposes. This includes the communication of version information as well as tunnel IP addresses and various configuration flags. An overview of all notify types is shown in Table 4. The byte structure of Link Director Message (LDM) payloads in shown in Figure 16. The LDM TLVs (Table 5) other than the previously mentioned Update Wi-Fi Address messages are related to management of the wireless links and appear to be only rarely used.

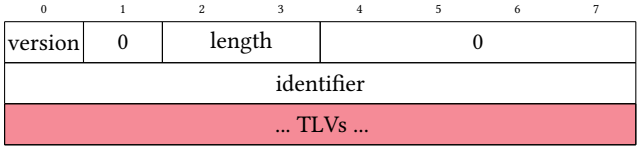

Figure 16: Structure of the LDM notify payload. The only observed version is 2, the length field encodes the byte length of the TLVs only.

Table 4: Private notify types used by Apple in their IKEv2 implementation. *IAdr* short for *InnerAddress*.

ID	Name	Comment
48601	Encrypted prelude	Bluetooth only, echoes prelude sent at the start of a NRLP connection
48602	Terminus version	e.g. 0x00d, 0x00c
48603	Device name	e.g. "iPhone", "Apple Watch"
48604	Build version	e.g. "18H17", "18S830"
50701	ProxyNotify	IPv6 address and port of Shoes server on the phone
50702	LinkDirectorMessage	used for link state signaling and Wi-Fi discovery
50801	IAdrInitiatorClassD	IPv6 tunnel address used by the watch for class D traffic
50802	IAdrResponderClassD	IPv6 tunnel address used by the phone for class D traffic
50811	IAdrInitiatorClassC	IPv6 tunnel address used by the watch for class C traffic
50812	IAdrResponderClassC	IPv6 tunnel address used by the phone for class C traffic
51401	Always-On Wi-Fi	1 byte boolean flag
51501	IsAltAccountDevice	1 byte boolean flag

Table 5: Types of Link Director Message TLVs.

#	Name	Comment
1	Hello	no payload, signals restart
2	UpdateWiFiAddressIPv6	2 byte port followed by 16 byte IP
3	UpdateWiFiAddressIPv4	2 byte port followed by 4 byte IP
4	UpdateWiFiSignature	variable length
5	PreferWiFi	no payload
6	DeviceLinkState	1 byte, 1: Bluetooth, 2: Wi-Fi
7	PreferWiFiAck	1 byte boolean
8	ForceWoW	no payload, WoW is Wake-on-Wireless

A.4 Cryptographic Algorithms

As explained in Section 4, the watch uses different modes of encryption when communicating with iOS. This section, focuses on the cryptographic algorithms used and their security aspects.

A.4.1 IKEv2/ESP. We include a full list of cryptographic primitives supported in IKEv2 for the watch models we tested in Table 6. The algorithm identifiers match the constants defined by IANA for use in IKEv2 [50]. For the Series 5 watch, we can observe that the ESP encryption algorithms match the ones offered for IKEv2. However, on newer Apple Watch models we are unable to observe the provided algorithms due to a lack of a jailbreak for iOS 17 or watchOS 10 at the time of writing. We expect that the ESP algorithms also match the algorithms offered for IKEv2 for these newer versions.

Table 6: Cryptographic algorithms advertised by watch models for use in IKEv2, in order of preference.

Series 5, watchOS 7.3.3	Series 9, watchOS 10.0.2
Encryption	
ChaCha20-Poly1305	AES-GCM-16 (256bit)
AES-GCM-16 (256bit)	ChaCha20-Poly1305
Pseudo-Random Function	
HMAC-SHA2-512	HMAC-SHA2-512
HMAC-SHA2-256	
Diffie-Hellman Group	
Curve25519	Curve448
521-bit random ECP group	Curve25519
8192-bit MODP Group	
Signature Hash Algorithm	
SHA2-256	Identity
Identity	SHA2-256

Diffie-Hellmann Key Exchange. The Apple Watch primarily uses Diffie-Hellmann (DH) key exchanges based on elliptic curve cryptography (ecc), allowing for smaller key sizes with security levels comparable to larger modular exponentiation groups. When using ecc, the use of an appropriate curve is paramount to its security. Apple decided to select only curves which result in 256-bit symmetric keys and all of these curves are standardized by the IETF [43, 55, 68]. Curve448 and Curve25519 are the recommended curves to be used for security purposes [58] and are the only curves supported from watchOS 10 on.

Encryption. To perform symmetric encryption in IKEv2 and ESP, ChaCha20-Poly1305 or AES-GCM-16[6] with 256-bit keys are used. Both algorithms are authenticated encryption with associated data (AEAD) schemes, authenticating the encrypted message and optional additional plaintext data using an authentication tag. When decrypting the ciphertext, the algorithm checks if the authentication tag matches the expected value and throws an error if not. An adversary in a machine-in-the-middle (MitM) position modifying the ciphertext or authenticated data can be detected and the integrity of the message is protected. The formal security of both ciphers has been proven [34, 69] and there exist no known attacks against them.

A.4.2 A-over-C. Apple uses IKEv2 and ESP for general data transfer between the Apple Watch and the connected iPhone. However, for sensitive data they add a second layer of a custom encryption scheme called A-over-C (see Section 4.4). A-over-C uses RSA-Optimal Asymmetric Encryption Padding (OAEP) with 1280-bit keys and AES-CTR (counter mode) with a 128-bit key to encapsulate an ephemeral 128-bit key, which is chosen randomly for every message. The encapsulated key is authenticated using an ECDSA signature with a 384-bit key. The resulting encapsulation scheme provides confidentiality and authenticity, and has been formally analyzed in the context of iMessage, which uses an identical

[6] *16* denotes the number of octets used for the authentication tag.

construction [44].[7] While key sizes for AES and ECDSA match recommended values, the RSA keys fall well short of the recommended minimum of 2048 bit [17], making them vulnerable to potential brute-force attacks in the future.

The payload itself is encrypted using AES-CBC (cipher block chaining mode). Like AES-CTR, AES-CBC provides confidentiality if used correctly but does not provide authentication as would be the case for AEAD schemes such as AES-GCM. We detail this issue in Section 6.3. AES-CBC as used by the Apple Watch is also vulnerable to *padding oracle* attacks which could reveal message plaintext under the right circumstances [94].

[7] The malleability attacks from [44] do not apply here as the entire encapsulated key is contained in the RSA-OAEP ciphertext, which is non-malleable.