Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage

Christina Garman
Johns Hopkins University
cgarman@cs.jhu.edu

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

Gabriel Kaptchuk
Johns Hopkins University
gkaptchuk@cs.jhu.edu

Ian Miers
Johns Hopkins University
imiers@cs.jhu.edu

Michael Rushanan
Johns Hopkins University
micharu1@cs.jhu.edu

Abstract

Apple's iMessage is one of the most widely-deployed end-to-end encrypted messaging protocols. Despite its broad deployment, the encryption protocols used by iMessage have never been subjected to rigorous cryptanalysis. In this paper, we conduct a thorough analysis of iMessage to determine the security of the protocol against a variety of attacks. Our analysis shows that iMessage has significant vulnerabilities that can be exploited by a sophisticated attacker. In particular, we outline a novel chosen ciphertext attack on Huffman compressed data, which allows *retrospective* decryption of some iMessage payloads in less than 2^{18} queries. The practical implication of these attacks is that any party who gains access to iMessage ciphertexts may potentially decrypt them remotely and after the fact. We additionally describe mitigations that will prevent these attacks on the protocol, without breaking backwards compatibility. Apple has deployed our mitigations in the latest iOS and OS X releases.

1 Introduction

The past several years have seen widespread adoption of end-to-end encrypted text messaging protocols. In this work we focus on one of the most popular such protocols: Apple's iMessage. Introduced in 2011, iMessage is an end-to-end encrypted text messaging system that supports both iOS and OS X devices. While Apple does not provide up-to-date statistics on iMessage usage, in February 2016 an Apple executive noted that the system had a peak transmission rate of more then 200,000 messages per second, across 1 billion deployed devices [12].

The broad adoption of iMessage has been controversial, particularly within the law enforcement and national security communities. In 2013, the U.S. Drug Enforcement Agency deemed iMessage "a challenge for DEA intercept" [22], while in 2015 the U.S. Department of Justice accused Apple of thwarting an investigation by refusing to turn over iMessage plaintext [11]. iMessage has been at the center of a months-long debate initiated by U.S. and overseas officials over the implementation of "exceptional access" mechanisms in end-to-end encrypted communication systems [7, 26, 33], and some national ISPs have temporarily blocked the protocol [32]. Throughout this controversy, Apple has consistently maintained that iMessage encryption is end-to-end and that even Apple cannot recover the plaintext for messages transmitted through its servers [10].

Given iMessage's large installed base and the high stakes riding on its confidentiality, one might expect iMessage to have received critical attention from the research community. Surprisingly, there has been very little analysis of the system, in large part due to the fact that Apple has declined to publish the details of iMessage's encryption protocol. In this paper we aim to remedy this situation. Specifically, we attempt to answer the following question: how secure is Apple iMessage?

Our contributions. In this work we analyze the iMessage protocol and identify several weaknesses that an attacker may use to decrypt iMessages and attachments. While these flaws do not render iMessage completely insecure, some flaws reduce the level of security to that of the TLS encryption used to secure communications between end-user devices and Apple's servers. This finding is surprising given the protection claims advertised by Apple [10]. Moreover, we determine that the flaws we detect in iMessage may have implications for other aspects of Apple's ecosystem, as we discuss below.

To perform our analysis, we derived a specification for iMessage by conducting a partial black-box reverse engineering of the protocol as implemented on multiple iOS and OS X devices. Our efforts extend a high-level protocol overview published by Apple [9] and two existing partial reverse-engineering efforts [1, 34]. Armed with a protocol specification, we conducted manual cryptanalysis of the system. Specifically, we tried to determine the system's resilience to both back-end infrastructure attacks and more restricted attacks that subvert only client-local networks.

Our analysis uncovered several previously unreported vulnerabilities in the iMessage protocol. Most significantly, we identified a *practical* adaptive chosen-ciphertext attack on the iMessage encryption mechanism that allows us to retrospectively decrypt certain iMessage payloads and attachments, provided that a single Sender or Recipient device is online. To validate this finding, we implemented a proof of concept exploit against our own test devices and show that the attack can be conducted remotely (and silently) against any party with an online device. This exploit is non-trivial, and required us to develop novel exploit techniques, including a new chosen ciphertext attack that operates against ciphertexts containing `gzip` compressed data. We refer to this technique as a *`gzip` format oracle* attack, and we believe it may have applications to other encryption protocols. We discuss the details of this attack in §5.

We also demonstrate weaknesses in the device registration and key distribution mechanisms of iMessage. One weakness we exploit has been identified by the reverse engineering efforts in [34], while another is novel. As they are not the main result of this work, we include them in Appendix A for completeness.

Overall, our determination is that while iMessage's end-to-end encryption protocol is an improvement over systems that use encryption on network traffic only (*e.g.,* Google Hangouts), messages sent through iMessage may not be secure against sophisticated adversaries. Our results show that an attacker who obtains iMessage ciphertexts can, at least for some types of messages, *retrospectively* decrypt traffic. Because Apple stores encrypted, undelivered messages on its servers and retains them for up to 30 days, such messages are vulnerable to any party who can obtain access to this infrastructure, *e.g.,* via court order [11], or by compromising Apple's globally-distributed server infrastructure [36]. Similarly, an attacker who can intercept TLS using a stolen certificate may be able to intercept iMessages on certain versions of iOS and Mac OS X that do not employ certificate pinning on Apple Push Network Services (APNs) connections.

Given the wide deployment of iMessage, and the attention paid to iMessage by national governments, these threats do not seem unrealistic. Fortunately, the vulnerabilities we discovered in iMessage are relatively straightforward to repair. In the final section of this paper, we offer a set of mitigations that will restore strong cryptographic security to the iMessage protocol. Some of these are included in iOS 9.3 and Mac OS X 10.11.4, which shipped in March 2016.

Other uses of the iMessage encryption protocol. While our work primarily considers the iMessage instant messaging system, we note that the vulnerabilities identified here go beyond iMessage. Apple documentation notes that Apple's "Handoff" service, which transmits personal data between Apple devices over Bluetooth Low Energy, encrypts messages "in a similar fashion to iMessage" [9]. This raises the possibility that our attacks on iMessage encryption may also affect intra-device communication channels used between Apple devices. Attacks on this channel are particularly concerning because these functions are turned on by default in many new Apple devices. We did not investigate these attack vectors in this work but subsequent discussions with Apple have confirmed that Apple uses the same encryption implementation to secure both iMessage and intra-device communications. Thus, securing these channels is one side effect of the mitigations we propose in §7.

1.1 Responsible disclosure

In November 2015 we delivered to Apple a summary of the results in this paper. Apple acknowledged the vulnerability in §5 and has initiated substantial repairs to the iMessage system. These repairs include: enforcing certificate pinning across all channels used by iMessage,[1] removing compression from the iMessage composition (for attachment messages), and developing a fix based on our proposed "duplicate ciphertext detection" mitigation (see §7). Apple has also made changes to the use of iMessage in inter-device communications such as Handoff, although the company has declined to share the details with us. The repairs are included in iOS 9.3 and OS X 10.11.4, which shipped in March 2016.

1.2 Attack Model

Our attacks in §5 require the ability to obtain iMessage ciphertexts sent to or received by a client. Because Apple Push Network Services (APNs) uses TLS to transmit encrypted messages to Apple's back-end servers, exploiting iMessage requires either access to data from Apple's servers or a forged TLS certificate. We stress that while this is a strong assumption, it is the appropriate threat model for considering end-to-end encrypted protocols.

A more interesting objection to this threat model is the perception that iMesssage might be too weak to satisfy it. For example, in 2013 Raynal *et al.* pointed out a simple attack on Apple's key distribution that enables a TLS MITM attacker to replace the public key of a recipient with an attacker-chosen key [34]. One finding of this work is that as of December 2015 such attacks have been entirely mitigated by Apple through the addition of

[1]This feature was added to OS X 10.11 in December, as a result of our notification.

certificate pinning on key server connections (see Appendix A). More fundamentally, however, such attacks are *prospective* – in the sense that they require the attacker to target a particular individual before the individual begins communicating. By contrast, the attacks we describe in this paper are *retrospective*. They can be run against any stored message content, at any point subsequent to communication, provided that one target device remains online. Moreover, unlike previous attacks which require access to the target's local network, our attacks may be run remotely through Apple's infrastructure.

2 The iMessage Protocol

To obtain the full iMessage specification, we began with the security overview provided by Apple, as well as a detailed previous software reverse-engineering efforts conducted by Raynal [34] and others [1]. While these previous results provide some details of the protocol, they omit key details of the encryption mechanism, as well as the complete key registration and notification mechanisms. We conducted additional black-box reverse engineering efforts to recover these elements. Specifically, we analyzed and modified protocol exchanges to and from several jailbroken and non-jailbroken Apple devices.[2] In conformity to Apple's terms of service, we did not perform any software decompilation.

2.1 System overview

iMessage clients. iMessage clients comprise several pieces of software running on end-user devices. On iOS and OS X devices, the primary user-facing component is the Messages application. On OS X computers, this application interacts with at least three daemons: `apsd`, the daemon responsible for pushing and pulling application traffic over the Apple Push Notification Service (APNs) channel; `imagent`, a daemon that pulls notifications even if Messages is closed; and `identityservicesd`, a daemon which maintains a cache of other users' keys. iOS devices also contain an `apsd` daemon, while other daemons handle the task of managing identities.

Apple services. iMessage clients interact with multiple back-end services operated by Apple and its partners. We focus on the two most relevant to our attack. The Apple directory service (IDS, also known as ESS) maintains a mapping between user identities and public keys and is responsible for distributing user public keys on request. iMessage content is transmitted via the Apple Push Notification Service (APNs). Long iMessages and attachments are transmitted by uploading them to the iCloud service, which is operated by Apple using both their own servers and virtual servers provisioned on Amazon AWS, Microsoft Azure, and Google's Cloud Platform.

Identity and registration The basic unit of identity in iMessage is the iCloud account name, which typically consists of an email address or phone number controlled by the user. End-user devices are registered to the iCloud service by associating them with an account. The mapping between client devices and accounts is not one-to-one: a single account may be used across multiple devices, and similarly, multiple accounts can be associated with a single device. We give further information about the registration process in Appendix A.

Message encryption and decryption To transmit a message to some list of Recipient IDs, the Sender's iMessage client first contacts the IDS to obtain the public key(s) $PK_1, \ldots, PK_D$ and a list of APNs push tokens associated with the Sender and Recipient identities.[3] It then encodes the Sender and Recipient addresses and plaintext message into a binary `plist` key-value data structure and compresses this structure using the `gzip` compression format. The client next generates a 128-bit AES session key K and encrypts the resulting compressed message using AES-CTR with $IV = 1$. This produces a ciphertext c, which is next partitioned as $c = (c_1 \| c_2)$ where c_1 represents the first 101 bytes of c. The Sender parses each PK_i to obtain the public encryption key $pk_{E,i}$ and for $i = 1$ to D, calculates $C_i = \mathsf{RSA\text{-}OAEP}(pk_{E,i}, K \| c_1)$ and a signature $\sigma_i = \mathsf{ECDSASign}(sk_S, C_i \| c_2)$. For each distinct push token received from IDS, the Sender transmits (C_i, c_2, σ_i) to the APNs server. This process is illustrated in Figure 1.

For each ciphertext, the APNs service delivers the tuple $(ID_{sender}, ID_{recipient}, C_i, c_2, \sigma_i)$ to the intended destination. The receiving device contacts IDS to obtain the Sender's public key PK, parses for the signature verification key vk_S, then verifies the signature σ. If verification succeeds, it decrypts C_i to obtain $K \| c_1$, reconstructs $c = (c_1 \| c_2)$ and decrypts the resulting AES-CTR ciphertext using K. It decompresses the resulting `gzip` ciphertext, parses the resulting `plist` to obtain the list of Recipient IDs, and verifies that each of ID_{sender} and $ID_{recipient}$ are present in this list. If any of the preceding checks fail, or if the Recipient is unable to parse or decompress the resulting message, the receiving device silently aborts processing.

[2]In this analysis we considered iOS 6, 8, and 9 devices, as well as Mac clients running OS X 10.10.3, 10.10.5, and 10.11.1.

[3]This list includes one entry for each device registered to each Sender and Recipient ID. The Messages client encrypts the message with each Sender public key to ensure that message transcripts can be read across all of the Sender's devices.

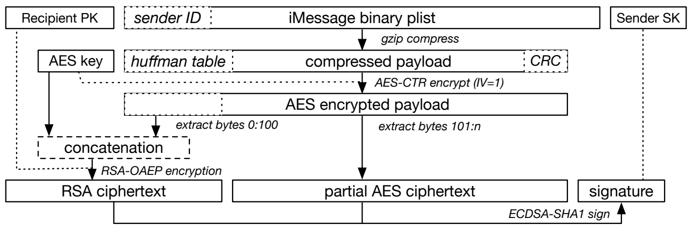


Figure 1: The iMessage encryption mechanism. From the top, each iMessage is encoded in a binary `plist` key/value structure. The structure encodes a list of Sender and Recipient account identifiers, as well as the message contents. This payload is subsequently `gzip` compressed, and encrypted under a freshly-generated 128-bit message key using AES in CTR-mode. The AES key and the first 101 bytes of the AES ciphertext are concatenated and are encrypted to each Recipient's public key using RSA-OAEP. The remaining bytes of the AES ciphertext are concatenated to the RSA ciphertext and the result is signed using ECDSA under the Sender's registered signing key.

Attachments and long messages For long messages and messages containing file attachments (*e.g.,* images or video), iMessage delivers the encrypted data using a separate mechanism. First, the client generates a 256-bit AES key K' and encrypts the attached data using AES in CTR mode. It next uploads the resulting encrypted document to Apple's iCloud service and obtains a unique `icloud.com` URL and an access token for the attachment. In the course of this process, the iCloud service may redirect the client to upload the encrypted file to a third-party storage server operated by an outside provider such as Amazon, Microsoft or Google. Having uploaded the attachment, the client now constructs a standard iMessage `plist` containing the URL and access token, the key K' and a SHA1 hash of the encrypted document. This `plist`, which may also include normal message text, is encrypted and transmitted to the Recipient using the standard message encryption mechanism. Upon receiving and decrypting the message, the Recipient downloads the attachment using the provided URL and access token, verifies that the provided hash matches the received attachment, and decrypts the attachment using K'.

3 Security goals & Threat model

Apple has stated that iMessage is an end-to-end encryption protocol that should be secure against all attackers that do not have control of Apple's network. We base our threat model on a recent survey on secure messaging by Unger *et al.* [38]. This threat model includes the following attackers:

Local Adversary. This includes an attacker with control over local networks, either on the Sender or Recipient side of the connection.

Global Adversary. An attacker controlling large segments of the Internet, such as powerful nation states or large Internet service providers.

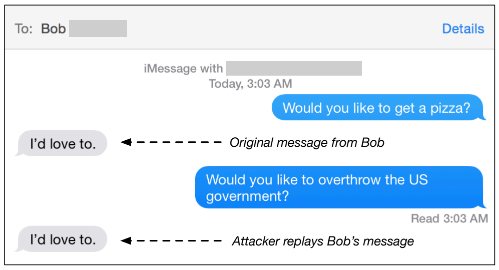
Network operator. Apple operates centralized infrastructure for both public key distribution and message transmission/storage. Potential adversaries include Apple, a government, or a malicious party with access to Apple's servers.

Each of these attackers may be active or passive. A passive attacker simply observes traffic and does not seek to alter or inject its own messages. An active attacker may issue arbitrary messages to any party. In many cases, these adversary classes may interact. As in [38] we assume that adversaries also have access to the messaging system, and can use the system to register accounts and transmit messages as normal participants. We also assume that the endpoints in the conversation are secure, although in some cases we allow for the possibility that an attacker might briefly take physical control of a device and/or convince a user to modify device configurations.

4 High-level Protocol Analysis

An initial analysis of the iMessage specification shows that the protocol suffers from a number of defects. In this section we briefly detail several of these limitations. In the following sections we focus on specific, exploitable flaws in the encryption mechanism.

Key server and registration iMessage key management uses a centralized directory server (IDS) which is operated by Apple. This server represents a single point of compromise for the iMessage system. Apple, and any attacker capable of compromising the server, can use this server to perform a man-in-the-middle attack and obtain complete decryption of iMessages. The current generation of iMessage clients do not provide any means for users to compare or verify the authenticity of keys received from the server.


To: Bob [redacted]

Details

iMessage with [redacted]

Today, 3:03 AM

Would you like to get a pizza?

I'd love to.

Would you like to overthrow the US government?

Read 3:03 AM

I'd love to.

Figure 2: Example of a simple ciphertext replay.

Of more concern, Apple's "new device registration" mechanism does not include a robust mechanism for notifying users when new devices are registered on their account. This mechanism is triggered by an Apple push message, which in turn triggers a query to an Apple-operated server. Our analysis shows that these protections are fragile; in Appendix A we implement attacks against both the key server and the new device registration process.

Lack of forward secrecy iMessage does not provide any forward secrecy mechanism for transmitted messages. This is due to the fact that iMessage encryption keys are long-lived, and are not replaced automatically through any form of automated process. This exposes users to the risk that a stolen device may be used to decrypt captured past traffic.

Moreover, the use of long term keys for encryption can increase the impact of other vulnerabilities in the system. For example, in §5, we demonstrate an active attack on iMessage encryption that exposes current iMessage users to decryption of past traffic. The risk of such attacks would be greatly mitigated if iMessage clients periodically generated fresh encryption keys. See §7 for proposed mitigations.

Replay and reflection attacks The iMessage encryption protocol does not incorporate any mechanism to prevent replay or reflection of captured ciphertexts, leading to the possibility that an attacker can falsify conversation transcripts as illustrated in Figure 2. A more serious concern is the possibility that an attacker, upon physically capturing a device, may replay previously captured traffic to the device and thus obtain the plaintext.

Lack of certificate pinning on older iOS versions iMessage clients interact with many Apple servers. As of December 2015, Apple has activated certificate pinning on both APNs and ESS/IDS connections in iOS 9 and OS X 10.11. This eliminates a serious attack noted by Raynal *et al.* [34] in which an MITM attacker who controls the Sender's local network connection and possesses an Apple certificate can intercept calls to the ESS/IDS key server and substitute chosen encryption keys for any Recipient (see Appendix A for further details). We note that devices running iOS 8 (and earlier) or versions of OS X released prior to December 2015 may still be vulnerable to such attacks. For example, at the time of our initial disclosure in November 2015 to Apple, pinning was not present in OS X 10.11.

Non-standard encryption iMessage encryption does not conform to best cryptographic practices and generally seems *ad hoc*. The protocol (see Figure 1) insecurely composes a collection of secure primitives, including RSA, AES and ECDSA. Most critically, iMessage does not use a proper authenticated symmetric encryption algorithm and instead relies on a digital signature to prevent tampering. Unfortunately it is well known that in the multi-user setting this approach may not be sound [21]. In the following sections, we show that an on-path attacker can replace the signature on a given message with that of another party. This vulnerability gives rise to a *practical* chosen ciphertext attack that recovers the full contents of some messages.

5 Attacks on the Encryption Mechanism

In this section we describe a practical attack on the iMessage encryption mechanism (Figure 1) that allows an attacker to completely decrypt certain messages.

5.1 Attack setting

Our attack assumes that an adversary can recover encrypted iMessage payloads, and subsequently access the iMessage infrastructure in the manner of a normal user. The first requirement implies one of two conditions: in condition (1) the attacker is on-path and capable of intercepting encrypted iMessage payloads sent from a client to Apple's Push Notification Service (APNs) servers. Since the APNs protocol employs TLS to secure connections between the client and APNs server, this attacker must possess some means to bypass the TLS encryption layer; we discuss TLS interception in more detail in Appendix B. In condition (2) the attacker can recover iMessage ciphertexts from within Apple's network. This requires either a compromise of Apple's infrastructure, a rogue employee, or legal compulsion. Figure 3 describes the network flow of a single iMessage, along with potential attacker locations.

5.2 Attack overview

There are two stages of the attack. The first exploits a weakness in the design of the iMessage encryption

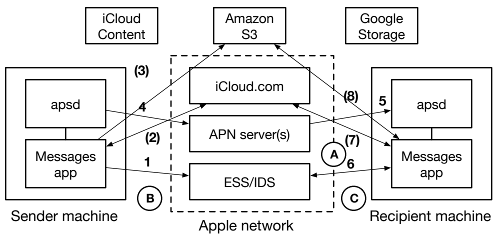


Figure 3: The process of sending an iMessage through the APNS network. The steps are as follows: (1) The Sender contacts ESS/IDS to obtain the public keys for each Recipient; (2) *(optional)* the Sender contacts iCloud to upload an attachment; (3) *(optional)* the Sender uploads the encrypted attachment to an outside storage provider as directed by iCloud; (4) the Sender's `apsd` instance transmits the encrypted iMessage payload to Apple's APNs server; (5) Apple delivers the payload to a Recipient; (6) the Recipient contacts ESS/IDS to obtain the Sender's public key; (7) *(optional)* the Recipient contacts iCloud if an attachment is present; (8) *(optional)* the Recipient downloads the encrypted attachment from an outside storage provider. Potential attacker locations are labeled A, B and C.

composition: namely, that iMessage does not properly authenticate the symmetrically encrypted portion of the message payload. In a properly-designed composition, this section of the ciphertext would be authenticated using a MAC in generic composition [14] or via an AEAD mode of operation. Apple, instead, relies on an ECDSA signature to guarantee the authenticity of this ciphertext. In practice, a signature is insufficient to prevent an attacker from mauling the ciphertext since an on-path attacker can simply replace the existing signature with an new signature using a signing key from an account controlled by the attacker. In practice, the actual attack is slightly more complex; the first phase includes additional operations to defeat a countermeasure in the decryption mechanism, which we discuss below.

The second stage of the attack leverages the ability to modify the AES ciphertext (specifically, the section not contained within the RSA ciphertext). This phase consists of an adaptive chosen ciphertext attack exploiting the structure of the underlying plaintexts. The attack repeatedly modifies the ciphertext and sends it to either the Sender or a Recipient for decryption. If the attacker can determine if decryption and parsing were successful on the target device, she can gradually recover the underlying iMessage payload.

The attack specifics are reminiscent of Vaudenay's padding oracle attack [40], but relies on the usage of *compression* within the iMessage protocol. Specifically, our attack takes advantage of the 32-bit CRC checksum, computed over the pre-compressed message, incorporated into `gzip` compressed ciphertexts. Since CRCs are linear under XOR we can verify guesses about message content by editing the compressed, encrypted message and testing if the corresponding correction to the CRC results in a valid message.

5.3 A format oracle attack for `gzip` compression

The `gzip` format [23] is a variant of DEFLATE compression that combines LZ77 [41] and Huffman coding to efficiently compress common data types. The format supports both static and dynamically-generated Huffman tables, though most encoders use dynamic tables for all but the shortest messages. To compress a message, a CRC32 C is calculated over the uncompressed input. Next, the encoder identifies repeated strings and replaces each repeated instance with a tuple of the form $\langle \mathit{length}, \mathit{backwards\ distance} \rangle$, where distance indicates the relative position of the previous instance of the string. The input is encoded using an alphabet of 286 symbols, comprising the 256 byte literals, an end-of-block (EOB) symbol, and 29 string replacement length values.[4] If dynamic generation is selected, a Huffman table T is calculated using the resulting text as a basis (for static tables, $T = \varepsilon$), and the text is Huffman coded into a string of variable-length symbols $S = (s_1, \ldots, s_N)$ where string replacement symbols are internally partitioned into a pair $\langle \mathit{length}, \mathit{distance} \rangle$. The resulting compressed message consists of (T, S, C). On decompression the process is reversed and the CRC of the resulting string is compared to C. If any step fails, the decompressor outputs $\perp$.

Attack intuition. Our attack assumes that the attacker has intercepted a `gzip` compressed message encrypted using an unauthenticated stream cipher and that we have access to a decryption oracle that returns 1 if and only if the message decrypts and successfully decompresses. Our goal is to recover a substantial fraction of the plaintext message.

For clarity, we assume the attacker knows the Huffman table T and the length in bits L of the uncompressed input. We further assume the attacker knows the exact location in the ciphertext corresponding to some (unknown) ℓ-bit Huffman symbol s that she wishes to recover, as well as the position of the corresponding decoded literal in the uncompressed text. These are simplifying assumptions and we will remove them as we proceed.

Given a ciphertext c, our attack works by first selecting a mask $M \in \{0,1\}^\ell, M \neq 0^\ell$ and perturbing the ci-

[4] A separate Huffman table is used to encode backwards distances.

phertext such that the underlying symbol s will decrypt to $s' = s \oplus M$. This is done by *xor*ing M into the ciphertext at the appropriate location. Let $\mathsf{decode}(T,s)$ and $\mathsf{decode}(T,s')$ represent the Huffman decoding of s and s' respectively, and let $\mathsf{repeats}$ be a boolean variable that is true if and only if s (resp. s') is repeated subsequently via a DEFLATE string replacement reference. The potential values of these three variables can be categorized into the following seven cases:

Case	$\mathsf{decode}(T,s)$	$\mathsf{decode}(T,s \oplus M)$	$\mathsf{repeats}$
1	$[0,255]$	$[0,255]$	False
2	$[0,255]$	$[0,255]$	True
3	$[0,255]$	$[256,285]$	(either)
4	$[0,255]$	$\perp$	(either)
5	$[256,285]$	$[0,255]$	(either)
6	$[256,285]$	$[256,285]$	(either)
7	$[256,285]$	$\perp$	(either)

In the following paragraphs, we consider the outcome of our experiment for each of the cases above.

CASE 1: In this case, when the attacker submits the mauled ciphertext to the decryption oracle, the oracle will internally decode a result that differs from the original input string in exactly one byte position: the position corresponding to symbol s'. However, with overwhelming probability, the CRC C' of the decompressed string will not match C and cause the oracle to output 0.

Because CRC is linear under XOR, the attacker may correct the encrypted value C by further mauling the ciphertext. Let d indicate the bit position of the symbol associated with s (resp. s') in the decoded message. For each $i \in \{0,1\}^8$ the attacker *xor*s the string $\bar{C} = CRC(0^d||i||0^{L-d}) \oplus CRC(0^L)$ with the ciphertext at the known location of C and submits each of the resulting ciphertexts for decryption. Since we have that $\mathsf{decode}(T,s') \in [0,255]$, one of these tests will always result in a successful CRC comparison.

Upon receiving a successful result from the decryption oracle, the attacker now examines the Huffman table T to identify candidate symbols s for which relation $\mathsf{decode}(T,s \oplus M) = \mathsf{decode}(T,s) \oplus i$ holds. If the attacker cannot identify a unique solution for s, she may select a new $M' \neq M \neq 0^\ell$ and repeat the procedure described above until she has uniquely identified s. The attacker can now increment her position in the ciphertext by ℓ bits and repeat this process to obtain the next plaintext symbol.

If this experiment is unsuccessful, it indicates that the ciphertext is not in Case 1 afrom the above table. To determine which case applies, the attacker must conduct additional experiments as described below. Sometimes recovery of the symbol s will not be feasible at all; when this occurs, the attacker must simply continue to the next symbol in S. Occasionally, the adversary may still be able to recover s at some additional cost.

CASES 3-4: In these cases, the original decoding of s was a byte literal, but the decoding of s' is either an invalid symbol or a special symbol (EOB or string replacement symbol). The former case always results in decompressor failure, while the latter will typically cause the decoded string to differ from the original input at multiple locations, resulting (with high probability) in a CRC comparison failure that will not be corrected by the procedure described above.

To address these cases, the attacker may select a new mask $M' \neq M \neq 0^\ell$ and repeat the complete experiment described above. Depending on the structure of the Huffman table T, and provided that $s \in [0,255]$, the new result $s \oplus M'$ may produce an outcome that satisfies the conditions of cases (1) or (2).[5]

CASE 2: In this case, the symbol represented by s (resp s') is referenced by one or more subsequent instances of DEFLATE string repetition. The practical impact is that modifying s will produce an identical alteration at two or more positions in the decoded string, and with high probability none of the experiments indicated for Case 1 will succeed.

In some circumstances, it may be cost effective for the attacker to skip s and simply move on to the next symbol in S. Alternatively, the attacker can experimentally modify the CRC to indicate the same alteration at *all* positions that could be affected by modifying s. Since the attacker does not know the locations at which s is repeated or the number of such locations, this requires the attacker to submit many candidate ciphertexts to the oracle, one for each possible set of locations where s may repeat. In the event that s (resp s') is repeated only once, this requires the attacker to issue $2^8 \cdot (L-d)/8$ queries to the oracle (one for each value of i and for each possible location for the repeated value of s'). This may be feasible for reasonably short strings.

CASES 5-7: These cases occur when the original symbol represented by $\mathsf{decode}(T,s)$ is a string replacement or EOB symbol. In most instances, replacing s with $(s \oplus M)$ produces a decoded string that differs from the original in many positions, making it challenging for the attacker to repair the CRC. If s decodes to a string replacement token, and the replacement reference points to a location that the attacker has already recovered, it may be possible for the attacker to detect the alteration using the technique described under Case 2. Otherwise the attacker must skip s and move on to the next symbol in S.

Recovering the unknowns. The procedure described so far requires the attacker to know the Huffman table T, the

[5]In principle, this approach might require as many as $2^8 \cdot 2^{|M|} = 2^{8+\ell}$ decryption queries to obtain a successful result, or rule out these cases. In practice, however, the number of candidate mask values M' is likely to be much more limited.

length of the uncompressed message L, the location and length of the symbol s, and the byte index of the corresponding decompressed literal. In practice many of these quantities may be determined experimentally by iterating through candidate values for L, ℓ, k and the symbol position. This requires the attacker to issue many candidate decryption requests until one succeeds. In the case of iMessage attachment messages, the length L is fixed and an attacker can generate a representative corpus of messages offline and easily estimate the other parameters *without* oracle queries.

Recovering the Huffman table is more challenging. If the message is encoded using a static table, then the table is known to the attacker. However, if T is dynamically generated, then the attacker learns only the relation $\mathsf{decode}(T, s \oplus M) = \mathsf{decode}(T, s) \oplus i$, but has no clear way of learning s or $\mathsf{decode}(T, s)$. Nonetheless, it might still be possible to recover enough information from these relations to recover the value of the underlying literals.

However, in iMessage this proves unnecessary as we take advantage of iMessage's structure to recover a large fraction of the dynamic table T. iMessage payloads containing attachments embed a URL within the encrypted message. Requests to which can be monitored (described below). In this way, we learn the file path and/or hostname indicated by the plaintext URL within each ciphertext. Given this information, and by mauling individual symbols s contained within the URL string, the attacker can recover the value $\mathsf{decode}(T, s \oplus M)$ for many different values of M. This allows the attacker to identify a relative-distance map of a portion of the Huffman tree. This proves sufficient to recover much of the Huffman table T.

Detecting successful decryption. Our attack assumes that the attacker can detect successful decryption of a modified ciphertext. To simplify this assumption, we focused on messages containing attachments, such as images and videos. These messages include a URL for downloading the attachment payload, as well as a 256-bit AES key to be used in decrypting the attachment. When an iMessage client correctly decrypts such a message, it automatically initiates an HTTPS POST request to the provided URL. A local network attacker can view (and intercept) this request to determine whether decryption has occurred. Moreover, if the attacker blocks the connection, the device will retry several times and then silently abort. Since the client provides no indication to the user that a message has been received, this admits silent decryption of ciphertexts.

This technique can be also extended to situations where the attacker is not on the target device's local network. By mauling the URL field to change the requested hostname (*e.g.,* from `icloud.com` to a domain that the attacker controls), the attacker can simply direct the target device to issues HTTPS to a machine that the attacker controls. This allows the attacker to conduct the attack remotely by transmitting ciphertexts through Apple's APNs network, at which point she obtains the full HTTPS POST request from the target device. Since the attacker controls the request domain, there is no need to MITM the TLS connection.[6]

5.4 An Attack on Attachment Messages

Having provided an overview of the attack components, we will describe each individual step of the complete attack. This attack scenario assumes that a target Sender has transmitted an attachment-bearing message to one or more online receivers, and the attacker has the ability to monitor the local network connection (and intercept TLS connections) on one of the Sender or Recipient devices.

Step 1. *Removing and replacing the iMessage signature.*

Each iMessage is authenticated using an ECDSA signature, formulated using the private key of the iMessage Sender. This signature prevents the attacker from directly tampering with the message. However, a limitation of using signatures for authenticity is that they do not prevent ciphertext mauling when an attacker controls another account in the system. An attacker who intercepts a signed iMessage may simply remove the existing signature from the message and re-sign the message using a different key, corresponding to a separate account that the attacker controls.[7] The attacker now transmits the resulting encrypted payload, signed and delivered as though from a different Sender address. The signature replacement process is illustrated in Figure 4.

In practice, simply replacing the signature on a message proves insufficient. In iMessage, a full list of Sender and Recipient addresses is specified both in the unencrypted metadata for the message, *and* in the encrypted message payload. Upon decrypting each message, iMessage clients verify that the message was received from one of the accounts listed in the Sender/Recipient list, and silently abort processing if this condition does not hold.[8]. While it is trivial to replace the unencrypted Sender field, replacing encrypted envelope information is more challenging. Fortunately, in most cases this field of the iMessage `plist` is contained within the malleable

[6]The current versions of Apple's Messages client do not enforce that this URL contains `icloud.com`, and will connect to any hostname provided in the URL. Similarly, the Messages client does not pin certificates for the HTTPS connection.

[7]On Mac OS X, iMessage signing keys are readily accessible from the Apple Keychain.

[8]Based on our experiments, the participant list does not appear to be ordered, or to distinguish between Sender and Recipients. It is sufficient that the Sender identity appears somewhere in this list.

AES-CTR ciphertext, and we are able to alter the contents of the Sender/Recipient list so that it contains the identity of the replacement Sender account.

Step 2. *Altering the Sender identity.*

To alter the Sender identity, the attacker must selectively maul the AES-CTR ciphertext to change specific bytes of the Sender/Recipient `plist` field to incorporate the new Sender identity she is using to transmit the mauled ciphertext. This is challenging for several reasons.

First, the initial 101 bytes of the AES ciphertext are stored within the RSA-OAEP ciphertext, which is strongly non-malleable. Thus we are restricted to altering the subsequent bytes of the ciphertext. Fortunately, the binary `plist` key-value data structure is *top heavy*, in that it stores a list of all key values in the data structure prior to listing the values associated with each key. In practice, this ensures that the relevant Sender identity appears some distance into the data structure. Moreover, the application of `gzip` compression produces additional header information, including (in many cases) a dynamic Huffman table. In all of the cases we observed, the symbols encoding the Sender identity are located subsequent to the first 101 bytes, and are therefore not included within the OAEP ciphertext.

The use of `gzip` compression somewhat complicates the attack. Rather than mauling uncompressed ASCII bytes, the attacker must alter a set of compressed Huffman symbols which have been encoded using a (dynamically-generated) table T that the attacker does not know. Fortunately, the attacker knows the original identity of the Sender, as this value is transmitted in the unencrypted `apsd` metadata. Moreover, in all iMessage clients that we examined, the Sender identity is transmitted as the first string in the Sender/Recipient list, which – due to iMessage's predictable format – appears in a relatively restricted range of positions within the ciphertext. Even with this knowledge, altering the Sender ID involves a large component of guessing. The attacker first estimates the location of the start of the Sender/Recipient list, then selectively mauls the appropriate portions of the AES ciphertext, while simultaneously updating the CRC to contain a guess for the modified (decoded) symbol. This is a time consuming process, since the attacker must simultaneously identify (1) the appropriate location in the ciphertext for the symbol she wishes to modify, (2) a modification that causes the symbol to change to the required symbol. The target device will silently ignores any incorrect guesses, and will proceed with attachment download only when the mauled Sender ID in the `plist` is equal to the Sender ID from which the the attacker is transmitting.

To simplify the attack, the attacker may restrict her attention to addresses that differ from the original Sender ID in at most one symbol position. This is accomplished by registering new iCloud addresses that are "one off" from the target Sender identity. To increase the likelihood that we will succeed in altering the Sender account to match one that we have selected, we register multiple new Sender identities that are near matches to the original identity. For each attempt at mauling the ciphertext, we must also "repair" the CRC by guessing the effect of our changes on the decompressed message.

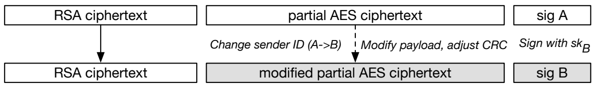


Figure 4: Modifying the partial AES ciphertext, including the Sender ID and CRC, and replacing the signature with a new signature corresponding to an account (and signing key) we control.

In our experiments, we found that an email address of the form `abcdef@icloud.com` could be efficiently modified to a new account of the form `abcdef@i8loud.com` in approximately 2^{17} decryption queries to a target device.[9] Since Huffman tables vary between messages, we cannot mutate every message to the same domain, and thus we need to control several variants of `icloud.com` for this strategy to be successful in all cases. Fortunately, the edits are predictable and our simulations indicate that we require only one domain to recover most messages.

A side effect of this modification is that, due to string replacement in `gzip`, the attachment URL is simultaneously altered to point to `i8loud.com`, which means that attachment HTTPS POST requests are sent to a computer under our control. This makes it possible to conduct the attack remotely.

Step 3. *Recovering the Huffman table.* Given the ability to intercept the attachment request POST URL to `icloud.com`, we now recover information about the dynamic Huffman tree T used in the message. The attachment path consists of a string of alphanumeric digits, which in most instances are encoded as Huffman symbols of length $\ell \in [4, 8]$.

By intercepting the HTTPS connection to `icloud.com`, the attacker can view the decoded the URL path and systematically maul each Huffman symbol in turn, repairing the CRC using the technique described in the previous subsection. This allows the attacker to gradually recover a portion of the Huffman tree (Figure 5). In practice, the attacker is able to recover only a subset of the tree, however, because the iMessage client will silently fail on any URL that

[9]These email addresses are examples, and not the real email addresses we used in our experiments.

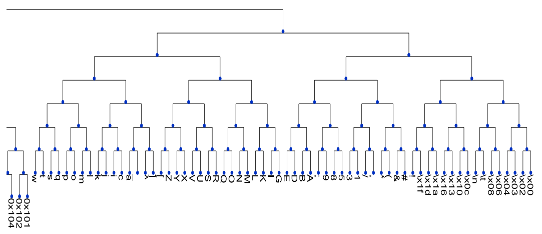


Figure 5: Fragment of a Huffman tree from an attachment iMessage.

contains characters outside the allowed URL character set.[10] Fortunately this set includes most printable alphanumeric characters.

Our implementation recovers a portion of the Huffman tree that is sufficient to identify the characters in the set $0-9, A-F$. Our experiments indicate that this phase of the process requires an average 2^{17} decryption requests and a maximum of 2^{19}

Step 4. *Recovering the attachment encryption key.* When an iMessage contains an attachment, the message embeds a 256-bit AES key that can be used to decrypt the attachment contents. This key is encoded as 64 ASCII hexadecimal characters and is contained within a field named `decryption-key`. An attacker with oracle access to a target device, and information on the Huffman table T, can now systematically recover bytes from this key. Upon recovering the key, they can use the intercepted HTTPS request information to download the encrypted attachment and decrypt it using the recovered key.

The approach used in recovering the attachment key is an extension of the general format oracle attack described above. The attacker first searches the ciphertext to identify the first position of the decryption key field. The attacker identifies a mask M (typically a single or double-bit change to the ciphertext) that produces a change in the decoded message at the first position of the encryption key, which is known due to the predictable structure of attachment messages. To identify this change, the attacker "fixes" the CRC to test for each possible result from the decryption key, then learns whether the decryption/decompression process succeeds. To obtain the full key, the attacker repeats this process for each of the 64 hexadecimal symbols of the encryption key.

This process does not reliably produce every bit of the key, due to some complications described in the general attack description above. Principal among these is the fact that some Huffman symbols represent string replacement tokens rather than byte literals. While it seems counterintuitive to expect repeated strings within a random key, this occurrence is surprisingly common due to the fact `gzip` will substitute even short (3 digit) strings. Indeed, on average we encounter 1.9 three-digit repetitions within each key. In this case, we attempt to identify subsequent appearances of the symbol by guessing later replacement locations. If this approach fails, our approach is to simply ignore the symbol and experimentally move forward until we reach the next symbol.

[10]iMessage does not perform URL coding on disallowed characters.

While it is possible to recover a larger fraction of the symbols in the message by issuing more decryption queries (see §6 for a discussion of the tradeoffs), in many cases it is sufficient to simply to guess the missing bits of the key offline after recovering an encrypted attachment. In practice, the entropy of the missing sections is usually much lower than would be indicated by the number of missing bits, since in most cases the replacement string is drawn from either the URL field or earlier sections of the key, both of which are known to the attacker.

Step 5. *Recovering the message contents.* Each attachment message may also contain message text. This text can be read in a manner similar to the way the key is recovered in the previous step, by mauling the message portion of the text and editing the CRC appropriately. This approach takes slightly more effort than the hexadecimal key recovery step, due to the higher number of potential values for each Huffman symbol in the message text.

6 Implementation and Evaluation

6.1 Estimating attack duration

To validate the feasibility of the attack described in §5.4, we implemented a prototype of the gzip format oracle attack in Python and executed it against the Messages client on OS X 10.10.3. Our attack successfully recovered 232 out of 256 key bits after 2^{18} decryption queries to the target device. The main challenge in running the attack was to determine the correct timeout period after which we can be confident that a message has not been successfully decrypted. This timeout period has a substantial impact on the duration of the attack, as we describe below.

Experimental Setup To deliver iMessage payloads to the device, we customized an open-source Python project called `pushproxy` (hereinafter called the proxy) and used it to intercept connections from the device to Apple's APNs server [3]. This approach models an attacker who can either impersonate or control Apple's APNs servers. While our attack assumed local network interception and did not send messages through Apple's servers, we note that if an attacker is able to capture messages in transit (by bypassing TLS) or by compromising Apple's servers, the remainder of the attack can in principle be conducted remotely (see the end of §5.3 for details). For ethical and legal reasons, we explicitly chose not to test attacks that relayed messages via Apple's production servers. Thus all of our attacks were conducted via a local network.

To address the use of TLS on `apsd` connections, we configured our modified proxy with a forged Apple certificate based on a CA root certificate we created, and change `/etc/hosts` to redirect APNs connections intended for Apple towards our local proxy. We generate the forged certificate by installing our root CA on the target system.[11]

To monitor and intercept attachment download requests, we configured an instance of a TLS MITM proxy (`mitmproxy`) using our self-signed root certificate to intercept all outbound requests from the device made via HTTP/HTTPS. When the target device receives an attachment message, it makes two HTTPS POST requests to {0,...,255}`-content.icloud.com`. Based on the result of these requests, the device issues a second HTTP GET request to download the actual attachment. In our experiments we block both of the POST requests, ensuring that no indication of the message processing is displayed by the Messages client. For each oracle query, the attack code waits for mitmproxy to report an attachment POST request as defined above or, after a set time out, assumes the oracle query resulted in a failed message.

Finally, we created an iMessage account for the attacker that is a single-character edit of the sender's address (e.g. if the sender is alice@example.com, the attacker might be clice@example.com). We only generate one such account for the edit we expect to be successful, although a real attacker might register a large corpus of iMessage accounts and thus increase the success probability of this phase of the attack.

Verifying the existence of the oracle To ensure that iMessage behavior is as expected, we conducted a series of tests using hand-generated messages to determine if we were able to detect decryption success or failure on these messages. Our results were sufficient to confirm the vulnerability of §5, and verify iMessage's behavior sufficiently well that we could construct a simulated oracle for our experiments of §6.2.

Estimating the timeout for failed queries The main goal of our experiment was to determine the maximum timeout period after which we can determine that the device has been unable to successfully decrypt and process a message. To determine this, our attack queries the gzip format oracle by sending a candidate message and waiting until it either sees a resulting attachment download (in which case the message decrypted) or some timeout passes. Too long of a timeout results in unreasonable runtimes and too short of a timeout produces false negatives, which lead to incorrect key recovery.

Small scale experiments proved unable to reliably estimate the maximum timeout: the observed wait time distribution seemingly has a long tail and may be dependent on load not encountered in small experiments (e.g. due to failed decryptions). Using the full attack code to find the max timeout, on the other hand, is impractical, since we must run 2^{18} queries, each lasting as long as the timeout. This would take between 18 hours and 3 days depending on the timeout duration we wish to test.

In order to estimate the correct timeout, we ran our attack on the device in tandem with a local instance of the format oracle which, using the recipient's private key, also decrypts the message and emulates iMessage's behavior. If the candidate message fails to decrypt against the local oracle, we use a short (400ms) timeout period. If the candidate message decrypts successfully on this local oracle, then we wait an unbounded amount of time for the oracle query, and record the necessary delay. We stress that this local-oracle approach was used only to speed up the process of finding the maximum delay; the full attack can be conducted *without* knowledge of the private key.

Results We ran our main experiment on a real message intercepted using the proxy. It recovers 232 out of 256 key bits in 2^{18} queries and took 35 hours to run. The maximum observed delay between a query and the resulting download request was 903ms, while the average was 390ms with a standard deviation of 100ms. Based on this data, and without considering further optimizations, we estimate that the full attack would require approximately 73 hours to run if we naively used 1 second as the timeout.

Optimizing runtime The obvious approach to optimizing our attack is to reduce the timeout period to the minimum period that iMessage requires to successfully process and queue a message. Through experiment, we determined this to be approximately 400ms. Thus one avenue to optimizing the experiment is to reduce the timeout period for all messages to 400ms, using the assumption that a successful experiment may result in a "late" download. Since we would not be able to neatly determine the specific message query that occasioned the download, we would need to temporarily increase the

[11]Since OS X 10.10.3 does not include certificate pinning for APNs connections, this allowed us to intercept and inject iMessage ciphertexts.

delay period and "backtrack" by repeating the most recent *e.g.,* 10 queries to determine which one caused the download. We are in the process of implementing this optimization and will present the results in the full version of this work. Because successful queries are quite sparse,[12] this does not meaningfully affect the number of queries needed for the attack. In our estimation, these techniques will reduce the cost of the full attack down to 35 hours and requires only straightforward modifications to our proof of concept code.

A second optimization is to run the attack against multiple devices with attack queries split and conducted in parallel against them. For *n* devices, the attack time is reduced by approximately a factor of *n*. As many users may have 2 or 3 devices, this can offer substantial reductions.

Finally, we can reduce the raw number of queries needed to mount the attack by refining the gzip-oracle attack techniques. In particular, we can reduce the number of queries needed to recover the Huffman table by inferring the structure of the tree from the partial information we have, and from the observation that the Huffman trees fall within a fairly limited range of distributions. In particular we note that for the Huffman trees used in `gzip`, recovering the symbol lengths alone is sufficient to recover the tree. An approach drawing from techniques in machine learning to recover the Huffman table given only a few queries, the distribution of such tables, and known partial information could offer substantial improvements. We leave a full exploration of these optimizations to future work.

6.2 Simulation results

Although we have conducted our attack on iMessage, we have not explored its effectiveness with a large range of messages. Given the time it takes to run an experiment, doing so is prohibitive. We opt instead to simulate our results.

Simulation To evaluate the overall effectiveness of our format oracle attack, we constructed a simulated message generator and decryption oracle. Messages produced by our generator are distributed identically to real attachment-bearing messages, but contain randomly-generated strings in place of the filename, URL path, Sender and Recipient addresses, decryption key, and "signature" (hash) fields. The decryption oracle emulates the iMessage client's parsing of the inner binary plist. For performance, it skips encryption and decryption.[13] Decompression is done using Python's `gzip` module, which is a wrapper around on `zlib`. We experimentally validate the oracle's correctness against the transcript of a real attack and against separate messages.

Results We ran our simulated attack on a corpus of 10,000 generated messages and show the results in Figure 6. In all cases, our experiments completed in at most 2^{19} queries, with an average of approximately 2^{17} queries. For 34% of the experiments we ran, our attack was able to recover ≥ 216 bits of the attachment AES key. For 23% of the messages we experimented with, we recovered ≥ 224 bits of the key, enabling rapid brute-force of the remaining bits on commodity hardware.[14]

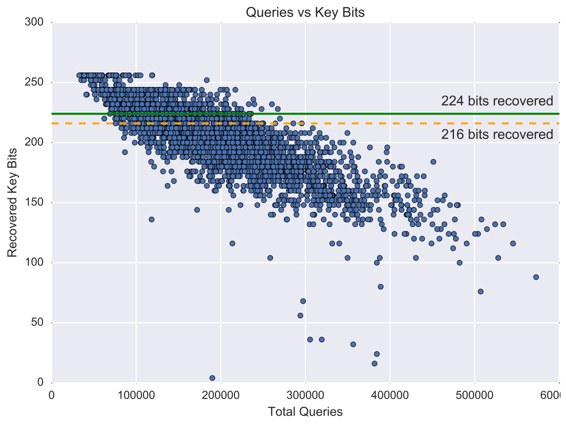
Optimizing success rate Many of the failures we experience in key recovery are caused by issues with string repetition. Recall that repeated substrings in a message are compressed in `gzip` by replacing all subsequent repetitions of the substrings with a backwards-pointing reference. As a result, editing the canonical location of a substring in the compressed message may cause similar changes to future instances of the same substring in the decompressed message. Our CRC correction for a given location fails to compensate for these later changes because we simply do not know where in the uncompressed message the second instance of the substring appears. As a result, our current attack simply skips these bits.

However, we can address this weakness with only a modest increase in the number of oracle queries. By scanning through the remaining bytes and applying the same CRC correction at each subsequent location in the uncompressed message, we can identify the location of the subsequent instances of the substring. This is efficient mainly for strings that are repeated twice, but our experiments indicate this is the most common case. Note that we do not need to scan through the entire message. As a result of the particular format of the messages, there are only a few points where we can get duplicates: most of the message is in lowercase letters or non-printable characters, whereas the `decryption-key` and `mmcs-url` field (i.e. the locations where repeats cause the most serious issues) are upper case alpha-numeric and hence will not contain repeats from the majority of the other fields. For the experiments described above, this would result in a 14% increase in the number of messages for which we can recover 224 bits.

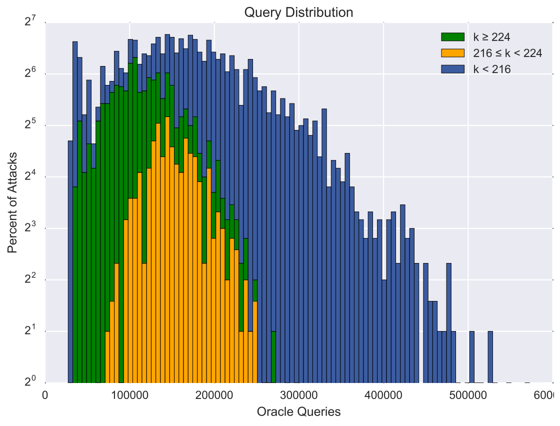
[12] Out of the 2^{18}, only 418 were successful.

[13] Our implementation prevents the attacker from modifying the first 101 bytes of the message, as those are normally contained within the RSA ciphertext. Additionally, the oracle enforces that the alleged Sender identity is included within the `plist`, which is a condition enforced by iMessage.

[14] Experiments on an inexpensive Intel Core i7 show that we can recover 32 missing key bits in approximately 7 minutes using an AES-NI implementation. Therefore recovering 40 missing key bits should take approximately 28 hours on a single commodity desktop.



(a) Number of queries vs number of recovered key bits. The orange dashed line represents 216 bits recovered, the solid green line 224.



(b) Distribution of attack length, measured in queries. The high concentration of attacks near zero is due to a rapid failure when it fails to edit the sender email.

Figure 6: Simulation results for the attachment recovery attack.

7 Mitigations

Our main recommendation is that Apple should replace the entirety of iMessage with a messaging system that has been properly designed and formally verified. However, we recognize this may not be immediately feasible given the large number of deployed iMessage clients. Thus we divide our recommendations into short-term "patches" that preserve compatibility with existing iMessage clients and long-term recommendations that require breaking changes to the iMessage protocol.

7.1 Immediate mitigations

Duplicate RSA ciphertext detection. The attacks we described in §5 are possible because the unauthenticated AES encryption used by iMessage is malleable and does not provide security under adaptive chosen ciphertext attack, unlike RSA-OAEP encryption [15]. Maintaining a list of all previously-received RSA ciphertexts should prevent these replay and CCA attacks without the need for breaking changes in the protocol. Upon receiving a stale RSA ciphertext, the Recipient would immediately abort decryption. This fix does not prevent all possible replays, given that iMessage accounts may be shared across multiple distinct devices. However, it would substantially reduce the impact of our attacks until a more permanent fix can be implemented. *Note: This modification has been incorporated into iOS 9.3 and Mac OS X 10.11.4.*

Force re-generation of all iMessage keys and destroy message logs. iMessage uses long-term decryption keys, and offers no mechanism to provide forward secrecy. If possible, Apple should force all devices to re-generate their iMessage key pairs and destroy previously-held secret keys. In addition, Apple should destroy any archives of encrypted iMessage traffic currently held by the company.

Pin APSD/ESS certificates or sign ESS responses. The current iMessage protocol relies heavily on the security of TLS, both for communications with the key server and as an additional layer of protection for iMessage push traffic. Apple should enhance this security by employing certificate (or public key) pinning within the Messages application and `apsd` to prevent compromise of these connections. Alternatively, Apple could extend their proprietary signing mechanisms to authenticate key server responses as well as requests.

Reorganize message layout. The current layout of encrypted messages includes approximately 101 bytes of the CTR message within the RSA-OAEP ciphertext, which is resilient to ciphertext malleability attacks. Modifying sender-side code to re-organize the layout of the underlying `plist` data structure to incorporate the sender and receiver fields within this section of the message would immediately block our attack. Implementing this change requires two significant modifications: (1) Apple would need to disable dynamic construction of Huffman tables within the gzip compression, and (2) restructure the binary plist serialization code to place the sender address first. We stress that this is a fragile patch: if any portion of the sender ID is left outside of the RSA ciphertext, the ciphertext again becomes vulnerable to mauling. Moreover, this fix will not protect group messages where the list of Recipients is longer than 100 bytes.

7.2 Long term recommendations

Replace the iMessage encryption mechanism. Apple should deprecate the existing iMessage protocol and replace it with a well-studied construction incorporating modern cryptographic primitives, forward secrecy and message authentication (*e.g.,* OTR [17] or the TextSecure/Axolotl protocol [4]). At minimum, Apple should use a modern authenticated cipher mode such as AES-GCM for symmetric encryption. This change alone would eliminate our active attack on iMessage encryption, though it would still not address any weaknesses in the key distribution mechanism. In addition, iMessage should place the protocol versioning information within the public key block and the authenticated portions of the ciphertext, in order to prevent downgrade attacks.

Implement key transparency. While many of the protocol-level attacks described in this paper can be mitigated with protocol changes, iMessage's dependence on a centralized key server represents an architectural weakness. Apple should take steps to harden iMessage against compromise of the ESS/IDS service, either through the use of key transparency [31], or by exposing key fingerprints to the user for manual verification.

8 Related Work

There are a three lines of research related to our work: secure message protocols, attacks on symmetric encryption, and decryptions attacks using compression schemes.

Instant messaging has received a great deal of attention from the research community. Borisov et al. introduced OTR [17], and proposed strong properties for messaging, such as per-message forward secrecy and deniability. Frosh et al. analyze a descendant protocols such as TextSecure [24]. More recent work has focused on multi-party messaging [25] and improved key exchange deniability [39]. In a related area, Chen *et al.* analyzed push messaging integrations, including Apple push networking [20]. For a survey of secure messaging technologies, see [38].

A number of works have developed attacks on unauthenticated, or poorly authenticated encryption protocols. In addition to the padding oracle of Vaudenay [40] and later applications [13], padding oracle attacks have been extended to use alternative side channels such as timing [8, 19]. Some more recent works have proposed attacks on more complex data formats such as XML [27, 30].

Some work has addressed the combination of compression and encryption. Some attacks use knowledge of a relatively small number of bytes in the plaintext to learn information about the compression algorithm and eventually recover an encryption key [16, 37]. Kelsey [28] and others [29, 35] used compression in the (partially) chosen plaintext setting to recover information about plaintexts.

9 Conclusion

In this work we analyzed the security of a popular end-to-end encrypted messaging protocol. Our results help to shed light on the security of deployed messaging systems, and more generally, provide insight into the state of the art in security mechanisms currently deployed by industry. This insight raises questions about the way research results are disseminated an applied in industry and how our community should ensure that widely-used protocols employ best cryptographic practices.

This work leaves several open questions. First, the `gzip` format oracle attack we describe against iMessage may apply to other protocols as well. For example, OpenPGP encryption (as implemented by GnuPG) [18] also employs `gzip` and may be vulnerable to similar attacks when it is used for online applications such as instant messaging [2]. Moreover, our attack requires that the adversary have some access to a portion of the decrypted information. We leave to future work the development of a pure "blind" attack on gzip encryption, one that does not require this additional information.

References

[1] iMessage. In OpenIM Wiki, Available at `https://imfreedom.org/wiki/IMessage`.

[2] MCABBER. Available at `https://mcabber.com/`.

[3] Pushproxy: A man-in-the-middle proxy for ios and os x device push connections. Available at `https://github.com/meeee/pushproxy`.

[4] Textsecure. `https://github.com/WhisperSystems/TextSecure/wiki/ProtocolV2`. Accessed: 2014-11-13.

[5] Trustwave to escape 'death penalty' for SSL skeleton key, 2012.

[6] Apple pulls ad-blocking apps that can 'compromise' security. *Engadget* (October 2015).

[7] ABELSON, H., ANDERSON, R., BELLOVIN, S. M., BENALOH, J., BLAZE, M., DIFFIE, W. W., GILMORE, J., GREEN, M., LANDAU, S., NEUMANN, P. G., RIVEST, R. L., SCHILLER, J. I., SCHNEIER, B., SPECTER, M. A., AND WEITZNER, D. J. Keys under doormats. *Commun. ACM 58*, 10 (Sept. 2015), 24–26.

[8] ALFARDAN, N. J., AND PATERSON, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P (Oakland) '13* (2013), pp. 526–540.

[9] APPLE COMPUTER. iOS Security: iOS 9.0 or later. Available at `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`, September 2015.

[10] APPLE INC. Privacy. Available at `http://www.apple.com/privacy/approach-to-privacy/`, 2015.

[11] APUZZO, M., SANGER, D. E., AND SCHMIDT, M. S. Apple and other tech companies tangle with U.S. over data access. Available at `http://www.nytimes.com/2015/09/08/us/politics/apple-and-other-tech-companies-tangle-with-us-over-access-to-data.html`, September 2015.

[12] BARBOSA, G. Apple execs Eddy Cue & Craig Federighi talk Apple Music, App Store & more in new interview. Available at `http://9to5mac.com/2016/02/12/apple-execs-eddy-cue-craig-federighi-talk-apple-music-app-store-more-in-new-interview/`, February 2016.

[13] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient padding oracle attacks on cryptographic hardware. In *CRYPTO '12*, vol. 7417 of LNCS. Springer, 2012, pp. 608–625.

[14] BELLARE, M., AND NAMPREMPRE, C. Authenticated encryption; relations among notions and analysis of the generic composition paradigm. *J. Cryptol. 21*, 4 (Sept. 2008), 469–491.

[15] BELLARE, M., AND ROGAWAY, P. Optimal asymmetric encryption: How to encrypt with RSA. In *EUROCRYPT '94* (1994), A. D. Santis, Ed., vol. 950 of LNCS, Springer, pp. 92–111.

[16] BIHAM, E., AND KOCHER, P. C. A known plaintext attack on the PKZIP stream cipher. In *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings* (1994), pp. 144–153.

[17] BORISOV, N., GOLDBERG, I., AND BREWER, E. Off-the-record communication, or, why not to use PGP. WPES '04, ACM Press, pp. 77–84.

[18] CALLAS, J., DONNERHACKE, L., FINNEY, H., SHAW, D., AND THAYER, R. RFC 4880: OpenPGP Message Format. Available at `https://tools.ietf.org/html/rfc4880`, November 2007.

[19] CANVEL, B., HILTGEN, A., VAUDENAY, S., AND VUAGNOUX, M. Password interception in a SSL/TLS channel. In *CRYPTO '03*, vol. 2729 of LNCS. Springer Berlin Heidelberg, 2003, pp. 583–599.

[20] CHEN, Y., LI, T., WANG, X., CHEN, K., AND HAN, X. Perplexed messengers from the cloud: Automated security analysis of push-messaging integrations. In *CCS '15* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1260–1272.

[21] CHIBA, D., MATSUDA, T., SCHULDT, J. C. N., AND MATSUURA, K. Efficient generic constructions of signcryption with insider security in the multi-user setting. In *ACNS '11* (2011), pp. 220–237.

[22] COVERT, A. Apple's iMessage is the DEA's worst nightmare. Available at `http://money.cnn.com/2013/04/07/technology/security/imessage-iphone-dea/`, April 2013.

[23] DEUTSCH, P. RFC 1952: GZIP file format specification version 4.3, May 1996.

[24] FROSCH, T., MAINKA, C., BADER, C., BERGSMA, F., SCHWENK, J., AND HOLZ, T. How secure is TextSecure? Cryptography ePrint Archive, October 2014.

[25] GOLDBERG, I., USTAOĞLU, B., VAN GUNDY, M. D., AND CHEN, H. Multi-party off-the-record messaging. In *CCS '09* (New York, NY, USA, 2009), CCS '09, ACM, pp. 358–368.

[26] GRIFFIN, A. WhatsApp and iMessage could be banned under new surveillance plans. *The Independent* (January 2015).

[27] JAGER, T., AND SOMOROVSKY, J. How to break XML encryption. In *ACM CCS '2011* (October 2011), ACM Press.

[28] KELSEY, J. Compression and information leakage of plaintext. In *FSE '02* (2002), vol. 2365 of LNCS, Springer, pp. 263–276.

[29] KOHNO, T. Attacking and repairing the winZip encryption scheme. In *ACM CCS '2004* (2004), ACM Press, pp. 72–81.

[30] KUPSER, D., MAINKA, C., SCHWENK, J., AND SOMOROVSKY, J. How to break XML encryption – automatically. In *Proceedings of the 9th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2015), WOOT'15, USENIX Association.

[31] MELARA, M. S., BLANKSTEIN, A., BONNEAU, J., FELTEN, E. W., AND FREEDMAN, M. J. CONIKS: Bringing key transparency to end users. In *USENIX '15* (Washington, D.C., Aug. 2015), USENIX Association, pp. 383–398.

[32] MESSIEH, N. Apple's iMessage and Facetime blocked in the UAE. *TheNextWeb* (November 2011).

[33] PALETTA, D. FBI Chief Punches Back on Encryption. *Wall Street Journal* (July 2015).

[34] RAYNAL, F. iMessage privacy. Available at `http://blog.quarkslab.com/imessage-privacy.html`, October 2013.

[35] RIZZO, J., AND DUONG, T. The CRIME Attack. Available at `https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1d134dff_1_222`, September 2012.

[36] SHIH, G., AND CARSTEN, P. Apple begins storing users' personal data on servers in China. *Reuters* (August 2014).

[37] STAY, M. ZIP attacks with reduced known plaintext. In *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers* (2001), pp. 125–134.

[38] UNGER, N., DECHAND, S., BONNEAU, J., FAHL, S., PERL, H., GOLDBERG, I., AND SMITH, M. SoK: Secure messaging. In *IEEE S&P (Oakland) '15* (2015).

[39] UNGER, N., AND GOLDBERG, I. Deniable key exchanges for secure messaging. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1211–1223.

[40] VAUDENAY, S. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS. In *EUROCRYPT '02* (London, UK, 2002), vol. 2332 of LNCS, Springer-Verlag, pp. 534–546.

[41] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (1977), 337–343.

A Attacks on Key Registration

While this work focuses on the retrospective decryption of iMessage payloads, in the course of our reverse engineering we were able to implement attacks on Apple's key registration infrastructure. The first attack is an implementation of attacks previously noted by Raynal *et al.* [34]. In these attacks, which work only against versions of iOS prior to iOS 9 and Mac devices prior to OS X 10.11.4 (*i.e.,* devices without key pinning), an attacker with a forged Apple TLS certificate can intercept the connection to the Apple key server in order to substitute chosen public keys. Additionally, we find a novel attack against the device registration process that allows an attack with stolen credentials to circumvent existing protection mechanisms.

The protocol for registering a device is shown in Figure 8. The user first establishes a TLS connection to Apple's IDS server and authenticates using their

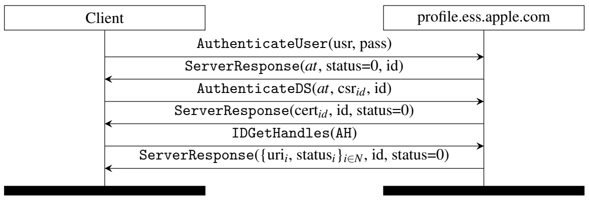

Client
profile.ess.apple.com
AuthenticateUser(usr, pass)
ServerResponse(at, status=0, id)
AuthenticateDS(at, csr_id, id)
ServerResponse(cert_id, id, status=0)
IDGetHandles(AH)
ServerResponse({uri_i, status_i}_{i∈N}, id, status=0)

Figure 7: Profile conversation. usr = username, pass = password, at = authentication token pt = push token, pk_{client} = client's public key, st = session token. `AH` is an authentication header with the following fields: $cert_{device}$ = signed by the Apple Fairplay Certificate, $cert_{id}$ = a certificate associated with the client id, id, pt, $\text{nonce}_{\text{device}}$, nonce_{id}, σ_{device}, and σ_{id}.

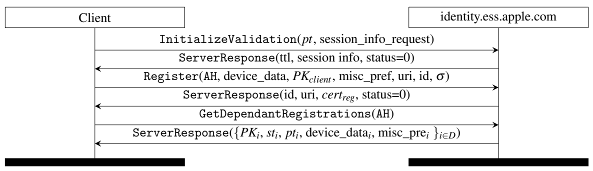

Client
identity.ess.apple.com
InitializeValidation(pt, session_info_request)
ServerResponse(ttl, session info, status=0)
Register(AH, device_data, PK_client, misc_pref, uri, id, σ)
ServerResponse(id, uri, cert_reg, status=0)
GetDependantRegistrations(AH)
ServerResponse({PK_i, st_i, pt_i, device_data_i, misc_pre_i}_{i∈D})

Figure 8: Identity conversation. pt = push token, pk_{client} = client's public key, st = session token. `AH` is an authentication header with the following fields: $cert_{device}$ = signed by the Apple Fairplay Certificate, $cert_{id}$ = a certificate associated with the client id, id, pt, $\text{nonce}_{\text{device}}$, nonce_{id}, σ_{device}, and σ_{id}.

iCloud credentials. The client generates two separate key pairs: a 1280-bit RSA public key pair (pk_E, sk_E) for use in encrypting and decrypting messages, and an ECDSA keypair (vk_S, sk_S) for authenticating messages. The client transmits the public portion of these keys $PK = (pk_E, vk_E)$ to the IDS, which registers it to the user's iCloud account name. We diagram the full login and registration protocols in Figures 7 and 8. To support multiple devices on a single account, the IDS will store and return all public keys associated with a given account.

A.1 Key Substitution Attack

The Apple key distribution systems are accessed each time a legitimate user wants to send an iMessage to a new Recipient. The Messages client first contacts `query.ess.apple.com` to look up the keys for a given username. In response, the server returns the user's public key(s), status, and push tokens for addressing APNs communications to the user. A fragment of the request and response is shown in Figure 9.

The `query.ess.apple.com` response message contains public keys, along with push tokens, for each of the devices registered to an account. Each of the key entries

```
GET /WebObjects/QueryService.woa/wa/query?uri=mailto
%3[redacted]40icloud.com&weight=light HTTP/1.1
Host: query.ess.apple.com
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>identities</key>
  <array>
    <dict>
      <key>client-data</key>
      <dict>
        <key>public-message-identity-key</
key><data>MIH2gUMAQQQzklEBPP0Nu0FHBovCJe+Prn8Rd97qf/j/ER3p2fRSe/
2BaYJnbIfEfQcpooKa3fWayu4+J1DJsIMaIwl52T7agoGuAKwwgakCgaEAoScfeVODb
EMjRrCNMWDQ2E2hWOXn46Mdqx7mLxJMS3LpGQjBoc3PeN1k3yMUqhi0YUYJJIq7dvac
1IJEiQilQDrc18eZ754BBknNmq7wXuDs8rQ2qmiE8/vOnCP4pOwwDQBy/
bdX2J3u2365R2VK6GDuk0zIjCeeAavAXr8kt9SzcvrO9KkYH1JKyKqn6FIYmR8cfeHt
ctJ0Tax8tnlZGQIDAQAB</data>
        <key>public-message-identity-version</key><real>2</real>
      </dict>
      <key>push-token</key><data>CI/
[redacted]=</data>
    </dict>
  </array>
  <key>status</key><integer>0</integer>
</dict>
</plist>
```

Figure 9: Excerpts from an ESS/IDS directory lookup request (top) and response (bottom). The request address and a portion of the response Push token have been redacted.

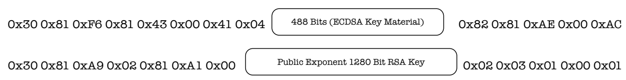

0x30 0x81 0xF6 0x81 0x43 0x00 0x41 0x04
488 Bits (ECDSA Key Material)
0x82 0x81 0xAE 0x00 0xAC
0x30 0x81 0xA9 0x02 0x81 0xA1 0x00
Public Exponent 1280 Bit RSA Key
0x02 0x03 0x01 0x00 0x01

Figure 10: Format of public key payload in ESS server response

is a 332 character long base64 encoded binary payload. When decoded, they takes the form shown in Figure 10.

Upon receiving the RSA public key in the above diagram, the Messages client uses this key to encrypt the outgoing iMessage payload. The ECDSA key is not used when sending a message, but is used to verify the integrity of a message when it is received from that user. iMessage clients appear to accept the most recent key delivered by ESS/IDS even if it disagrees with previous entries cached by the device.

Notably, the only security measures embedded in this conversation are authentication fields in the header of the *request*; the server does not sign the response. Thus the authenticity of the response depends entirely on the security of the TLS connection. This seems like an oversight, given that many other fields in the Apple protocols are explicitly authenticated. Worse, in iOS 8 and versions of OS X 10.11 released prior to December 2015, the Messages client does not use certificate pinning to ensure that the connection terminated by an Apple server. Thus an attacker with a stolen TLS root certificate can intercept key requests and substitute their own key as a response. This degrades the security of iMessage to that of TLS.

We implemented this attack by installing a self-signed X.509 root certificate into the local root certificate store of a Mac device. This allowed us to verify that there were no warning mechanisms that might alert a user to the key substitution. By further intercepting messages transmitted via the APNs network, we were able to respond to all key lookup requests with our own attacker key, and subsequently decrypt any iMessages transmitted via the device.

Our experiments demonstrate that iOS 9 is no longer subject to simple key substitution attacks, due to the addition of certificate pinning on TLS connections. This increases the relative impact of our novel decryption attacks. Surprisingly, our experiments demonstrated that OS X 10.11.1 remained vulnerable as of November 2015. We notified Apple of this oversight, and they have added key pinning as of OS X 10.11.13.

A.2 Credential theft

The first message in the registration process, shown in Figure 7, passes the user's credentials to the `profile.ess.apple.com` server to be verified. As noted in previous sections, OS X 10.10.5 and iOS 8 devices do not employ certificate pinning on this server, and the credentials are sent in plaintext within the TLS connection.[15] By conducting a TLS MITM attack on this connection, we are able to intercept iCloud login credentials. Using this information we can register new iMessage devices to an account, ensuring that we will be able to receive future messages.

Apple's primary defense against registration of new devices is a notification message that is sent to all previously-registered devices. In order to register a new device to a target account without alerting the victim, we also developed a method to overcome these notification mechanisms. We observed two such mechanisms:

1. Upon registration of a new device, all devices logged into the account receive a push notification over the APNs network. In response, each device initiates the `GetDependantRegistrations` call shown in Figure 8.

2. When an iMessage account is registered to a device that has not previously been registered to that account, a notification email is generated and sent to the account's registered email.

[15]OS X 10.11 devices do not employ certificate pinning on this connection either, but they do not appear to send the credentials in plaintext.

In the first instance, once the APNs push notification signaling that a `GetDependantRegistrations` call should be executed has arrived at a client, the client will continuously send the request until it receives a response. An active attacker on the victim's network can simply block all these requests, but this is not sustainable over long periods of time. We discovered that the client is satisfied when it receives any response — even a poorly formatted unreadable one. Thus, an attacker can edit the server response causing it to decode incorrectly. The client will accept this response and terminate the repeated `GetDependantRegistrations` calls. This blocks notifications that would alert the victim to the fact that a new device has been registered to their account. All subsequent iMessage traffic, both incoming and outgoing, will be forwarded to the attack device. Until a user logs out of their iMessage client, logs into a new iMessage client, or manually checks the list of devices associated with their account, they will never notice that their traffic is being forwarded to the attack device.

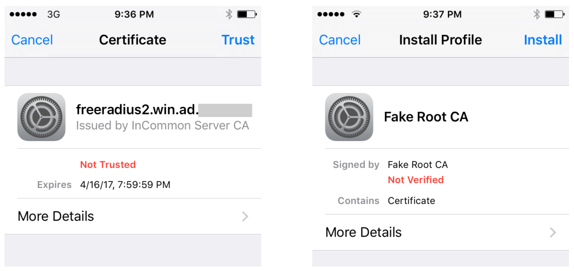
A.3 Updates in OS X 10.11

The ESS messaging protocol changed in a number of ways with the 10.11 update to OS X. The exchange of credentials for an authorization token has moved to point to `gsa.apple.com` and that connection has certificate pinning implemented. Due to this fact, we are unable to MITM this connection, but attempting to login to an account with bad credentials will result only in a message to that server and an error message displayed on the client. Additionally, there is a message sent to `setup.icloud.com` with a username and password pair in which the password is no longer transmitted in plaintext.

The key substitution attack still worked against OS X 10.11 versions as of November 2015, but the additional certificate pinning of `apsd` made it more difficult to intercept the message. In order to make sure the attack still functioned properly, we recovered the encrypted payload of the message from the `apsd` logs and were able to successfully decrypt the message using our own keys. Although we are not able to easily intercept the messages as we could with 10.10.5, this attack still effectively reduces the security of iMessage to that of TLS.

B Bypassing TLS

To execute the attacks described in this paper, the attacker must obtain encrypted iMessages from the APNs link. Since iMessage secures the APNs connection using TLS, this requires the attacker to penetrate to the TLS encryption on the link between Apple and the end-device.


●●●●● 3G 9:36 PM

Cancel Certificate Trust

freeradius2.win.ad.[redacted]
Issued by InCommon Server CA

Not Trusted
Expires 4/16/17, 7:59:59 PM

More Details

●●●●● 9:37 PM

Cancel Install Profile Install

Fake Root CA

Signed by Fake Root CA
Not Verified
Contains Certificate

More Details

Figure 11: On the left is a certificate verification dialog presented on encountering an unknown wireless access point. On the right is a root CA installation dialog.

We identified three approaches to bypassing the TLS on the APNs connections: (1) Apple, or an attacker with access to Apple's infrastructure, can intercept the contents of push messages as they transit the APNs servers; (2) on certain iOS and OS X versions that do not include certificate pinning for APNs, an attacker with access to a stolen CA root certificate may be able to conduct an MITM attack on the TLS connection; or (3) on the same versions, an attacker can "sideload" a root certificate on the target device, by briefly taking physical control of it, or convincing a victim to install a root certificate via a malicious email or web page. The latter technique is particularly concerning due to the similarity between Apple's interface for installing root CAs, and other non-critical certificate installation requests that may be presented to the user (see Figure 11). Since some Apple operating systems do not use certificate pinning, installation of a root certificate allows arbitrary interception of both APNs and HTTPS connections.

We identified attacks (2) and (3) as infeasible on all iOS 9 versions due to the inclusion of certificate pinning on APNs connections in that operating system. As of November 2015 when we first notified Apple of the results in this paper, we discovered that the then-current version of OS X 10.11 did not include certificate pinning. In response to our disclosure, Apple added certificate pinning to OS X as of December 2015.

We stress that given the interest in iMessage expressed by nation-states [26], a compromise of CA infrastructure cannot be ruled out. Even without such attacks, there have been several recent examples of CA-signed root or intermediate certificates being issued for use within corporate middle-boxes, primarily for the purposes of enterprise TLS interception [5]. TLS interception may occur even within Apple OS distributions: a recent incident involving iOS 9 allowed ad-blocking software to install a TLS root certificate [6].