A SECURE TEXT MESSAGING PROTOCOL

by

Gary Belvin

A thesis submitted to Johns Hopkins University in conformity with
the requirements for the degree of Master of Science in Security Informatics.

Baltimore, Maryland

May, 2011

This work is licensed under a Creative Commons Attribution 3.0 Unported License.

Abstract

Mobile text messages are currently vulnerable to inspection, modification, and replay by network operators and those that influence network operators. This paper describes a set of protocols that provide end-to-end message confidentiality, integrity, and authenticity over the high latency, low bandwidth, Short Message Service provided by GSM networks.

Keywords – Short Message Service, SMS, Encryption, Protocol, Security, KAPS, SSMS

Project Advisors: Mr. Philip Zimmermann and Prof. Matthew Green

Acknowledgements

I would like to thank Mr. Philip Zimmermann who initially suggested this project to me and coached me through its development. Matt Green was my academic advisor and guided me in project management. I would also like to thank the all the faculty and staff at the Johns Hopkins Information Security Institute, especially Susan Hohenberger and Giuseppe Ateniese for instilling in me a love for cryptography. And finally, I'd like to thank my thoroughly awesome friends and family who encouraged, supported, challenged, and refined me over these last two years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Text Messaging is a popular feature of cellular networks that allows users to send and receive short, textual messages to and from other mobile devices. Text messaging, known more formally as the Short Message Service (SMS), has grown exponentially worldwide since 1995 with over 6.1 trillion messages being sent annually in 2010[**?**]. Yet for all its popularity, a protocol for the secure and efficient delivery of text messages using a peer-to-peer topology has not been published.

Previous work in SMS key management has included client-server topologies[**?**] for use in mobile banking; identity based encryption[**?**] which requires a private key generating authority; and adaptations of the expensive Off-the-Record Messaging protocol[**?**]. PKI and shared password based key management solutions have also been released commercially. Currently, however, there is no peer-to-peer based key agreement protocol for SMS besides Moxie Marlinspike's adaptation of OTR. This paper develops a second, more efficient scheme for key agreement and message security using the Short Message Service.

Two separate protocols for secure text messaging are developed in this thesis. The first establishes a secure session on top of the Short Message Service utilizing a shared secret. The second protocol is used to establish that shared secret.

The Secure SMS protocol (SSMS) is inspired by the Secure Real-time Transport Protocol (SRTP)[**?**] for secure Voice over IP (VoIP). SSMS establishes a secure session over SMS like SRTP establishes a secure session over RTP. SSMS encrypts and authenticates each message with a sequence number to prevent replay attacks. SSMS also has forward secrecy properties that safeguard previously transmitted messages in the case of an endpoint compromise.

The Key Agreement Protocol for SMS (KAPS) is inspired by ZRTP[**?**]. It employs the Elliptic

Curve Diffie-Hellman key agreement to establish a shared secret without invoking a trusted third party. Authenticity is provided by key continuity (similar to SSH), and a verbal form of user authentication. Together, key continuity and user authentication prevent man-in-the-middle attacks from going unnoticed. KAPS is also resilient to endpoint compromise as it restores fresh secrecy as soon as the attacker is absent.

Chapter 2

Secure SMS (SSMS)

The relationship between Secure SMS (SSMS) and SMS is analogous to the relationship between Secure RTP (SRTP)[**?**] and the Real-time Transport Protocol (RTP)[**?**]. SSMS provides integrity, confidentiality, and replay protection for SMS messages like SRTP does for RTP media streams. The security of SSMS is built on a single, externally provided, master key that is analogous to the SRTP master key. KAPS is the preferred method for this external key agreement, but other methods such as PKI or a password based key derivation scheme may also be used.

Rather than duplicating efforts, SSMS relies upon the robust message delivery and error correction properties of the Short Message Service. Since SMS is built on top of the network control channel for mobile phones (Signaling System #7[**?**]), the error correction and delivery properties of SMS can be relied upon with the same degree of confidence as the mobile network itself.

Secure SMS simplifies the construction of secure systems by abstracting away the details of message security once a shared secret has been established (much like SRTP does for RTP). For simplicity, SSMS takes a single master key as input, and internally derives all other key material needed for security.

For efficiency reasons SSMS superimposes the concept of sessions on top of the Short Message Service. This saves us from having to perform an expensive key agreement for every message. The sessions are unidirectional, meaning that two sessions with separate key material must be opened for bidirectional conversation. Key material for each direction, however, can be derived from the same shared secret. Within the session, sequence numbers and roll over counters are used to detect out of order messages and replay attempts.

SSMS makes use of some SMS features unique to the GSM standard for protocol disambiguation

and transportation of binary payloads. Messages are sent as SMS Protocol Data Units (PDUs) with a User Data Header (UDH) designating source and destination application port numbers. Application port numbers allow SSMS messages to be efficiently and reliably distinguished from traditional text messages that are sent in the clear. Port numbers can also be conveniently tied to application triggers, removing the need for a client to be continuously draining resource in the background waiting for messages. The PDU message format also supports raw 8-bit data transport rather than using an encoding scheme like Base64.

2.1 Secure Message Format

The full payload format of an SSMS message consists of the User Data Header (UDH), followed by a one octet sequence number. The ciphertext comes next, followed by a 24 bit MAC computed over the ciphertext, header, and message index (section 2.7). The ciphertext itself contains a padding format: first a one octet value denoting the length of the plaintext, the plaintext, optionally followed by null padding to expand the payload to the full 140 octet envelope. In all, the overhead amounts to 14 octets, or about 10% of the message envelope.

$$\text{SSMS Efficiency: } \frac{\text{126 Message bearing octets}}{\text{140 SMS payload capacity}} = 90\% \tag{2.1}$$

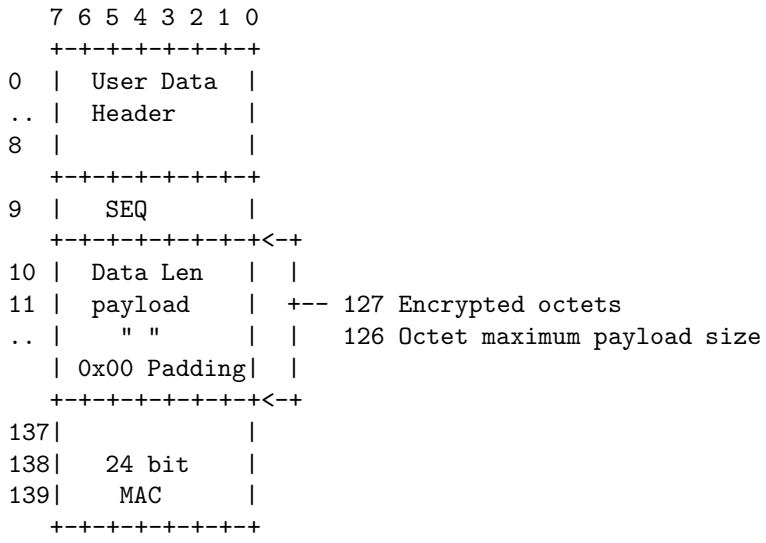

Figure 2.1: SSMS message layout

2.2 Algorithms and Ciphers

The algorithms and ciphers that SSMS uses are externally provided by the key agreement layer (for which KAPS may be used). The hash function used for the session identifier is the hash function in table 3.1. The MAC function used for the KDF is same MAC used in the key agreement layer. The block cipher, and authentication tag algorithm and size are also externally specified in table 3.1. Finally, the port number SSMS uses is the same that the key agreement layer uses, namely port 16474 – a randomly chosen value from the SMS application port range in table B.1.

In addition to protocol defined values, SSMS allows installation specific security settings. The **Rekey Frequency** is a policy driven value that determines the upper bound on the number of messages to transmit under a single master key before halting and requesting a new master key (which is provided from somewhere else). The rekey frequency must be less than 2^{40}, but normal values will be in the 10 – 100 message range. In the event of an endpoint security breach, a lower rekey frequency will reduce the window of readable messages before security is restored. Note that a rekey event may be performed at any time by either party – perhaps especially upon learning of a compromise. More frequent rekeying, however, increases the overhead from the key agreement layer.

2.3 Session Identifier

Each SSMS session is uniquely identified by the identities of the sending and receiving endpoints. Since this is a protocol for telephony, standardized phone numbers are used. The pair of phone numbers is hashed to obfuscate identifying information in memory. In some cases, even the simple identities of cryptographically communicating parties can be sensitive.

$$\text{Session Identifier} = H(\text{src number}||\text{0x3A}||\text{src port}||\ \text{0x00}\ ||\text{dest number}||\text{0x3A}||\text{dest port}) \quad (2.2)$$

To obtain a standard representation, phone numbers are formatted as international style ISDN numbers in accordance with the ITU recommendation E.164[**?**], formatted as UTF-8 strings which may be up to 15 digits in length.

When receiving an incoming message, an end point uses the session identifier as an index into a table of active sessions. If no active session is found for an incoming message, the message is dropped without further inspection.

2.4 Message Index

Each message within a session is uniquely identified by the combination of the sequence number inside the payload and a rollover counter, maintained by the client. The rollover counter allows the sequence number in the message to be small without compromising security.

The first message index in a session is a random value derived from the master key (section 2.5.1). A random starting number obfuscates how long the two parties have been communicating if the adversary was not present for the initial message. It also hides the number of messages remaining until the next rekeying event, which may be more sensitive than ordinary SSMS messages. Subsequent sequence numbers, however, always increase by one (modulus the size of the counter).

The message index is composed of an 8 bit sequence number and a 32 bit rollover counter in the following way:

$$i_m = 2^8 \cdot \texttt{ROLL} + \texttt{SEQ} \pmod{2^{40}} \tag{2.3}$$

2.5 Key Derivation Function

SSMS is given one master key as input. Further key material is derived using a hash based key derivation function that is compliant with NIST Special Publication 800-108[**?**] section 5.1.

The key derivation function is defined as the L left most bits of the MAC computed in the following way, where L must not exceed the size of the MAC function's output:

$$\mathsf{KDF}(K, \text{Label}, \text{Context}, L) = \mathsf{MAC}(K, 1||\text{Label}||\text{0x00}||\text{Context}||L) \tag{2.4}$$

where the notation $\mathsf{MAC}(a, b||c)$ denotes the computation of MAC defined in table 3.1, keyed with a on the value b concatenated with c.

K is a secret random bit string. The output of the KDF, however, is not required to be secret due to the key separation properties of the KDF. SSMS relies on this property for the value of the initial sequence number which is sent in the clear. The output of the KDF is also be used as another K in subsequent calls to the KDF to produce a non-invertible key derivation chain as depicted in figure 2.2. SSMS uses this technique in combination with key erasure to achieve forward secrecy.

Label identifies the purpose of the output key material.

Counter is required by NIST Special Publication 800-108[**?**] and is always 1 since we limit the output length of the KDF to be less than the output length of the MAC. The counter is encoded as a 4 octet integer.

Context is a binary string that ties the output key material to the particular situation in which it is being used. The context includes the session identifier (section 2.3) to tie the key material to the parties involved, and the message index (section 2.5.1) which is used as a nonce.

$$\text{Context} = \text{session identifier}||\text{message index} \tag{2.5}$$

L is the length in bits of the output key material encoded as a 4 octet integer.

2.5.1 Initial Message Index

SSMS uses the KDF in two ways. The first is to compute the initial sequence number, the second is to chain session keys for the purpose of forward secrecy.

The initial message index is derived using the KDF. Since no message index has been computed yet, we use a value of 0 as the nonce. Since this function is only called once for any master key, the static nonce is appropriate.

$$i_0 = \mathsf{KDF}(K_{master}, \text{“InitialIndex”}, \text{session identifier}||0, 40) \tag{2.6}$$

The rollover counter is assigned the 32 left most bits of i_m and the sequence number is assigned the following 8 bits such that $2^8 \cdot \texttt{ROLL} + \texttt{SEQ} = i_0$.

2.5.2 Forward Secrecy

SSMS uses distinct keys to encrypt each message. Each key is computed from the previous using the non-invertible key derivation function. The result is a key chain that protects the security of prior messages when a single key is compromised. Keys are also erased from memory as soon as they are no longer needed to prevent unneeded leakage in the event that a device is forensically analyzed.

The first message key is derived from the master key and the initial message index. Subsequent keys are derived from the previous key and the message index that identifies the current message

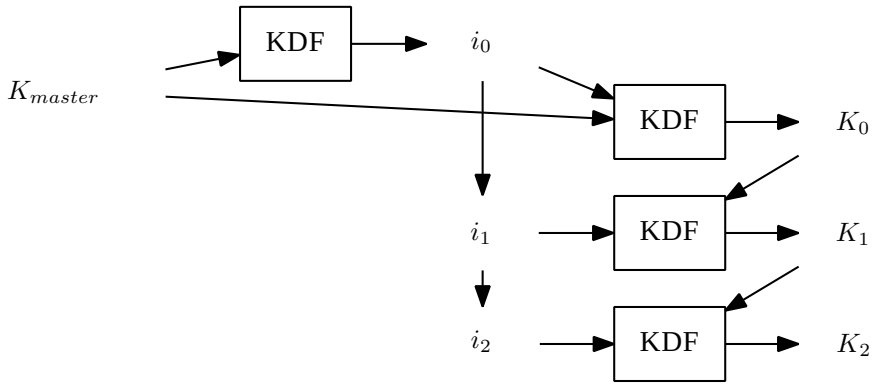

Figure 2.2: SSMS key derivation

being encrypted.

$$K_0 = \mathsf{KDF}(K_{master}, \text{“MessageKey”}, \text{session identifier}||i_0) \tag{2.7}$$

$$K_n = \mathsf{KDF}(K_{n-1}, \text{“MessageKey”}, \text{session identifier}||i_n) \tag{2.8}$$

As soon as the master key has generated both the initial index and initial message key, it is erased from memory, as are subsequent message keys as soon as they are no longer needed.

2.6 Replay Protection

SSMS supports out of order messages by caching a sliding window of message keys in memory. As soon as the message for a key is received and decrypted, the key is erased. There is also a short timeout of 5 minutes, after which the cached keys of unreceived messages are also erased. This sets up the condition that a message must not be out of order by more than 4 sequence numbers *and* no more than 5 minutes late. To prevent adversaries from performing denial of service attacks on legitimate messages, the MAC tags of incoming messages are verified before decrypting the message and erasing the corresponding message key.

Since MAC tags cannot be computed without the secret key, adversaries must consult a MAC oracle to check their guesses. For a 24 bit MAC, over 8 billion messages would need to be forged and checked with the client on average before a message could be successfully injected or modified. Since the user is warned for every invalid MAC, he or she will be able to clearly detect an attack and refuse further responses.

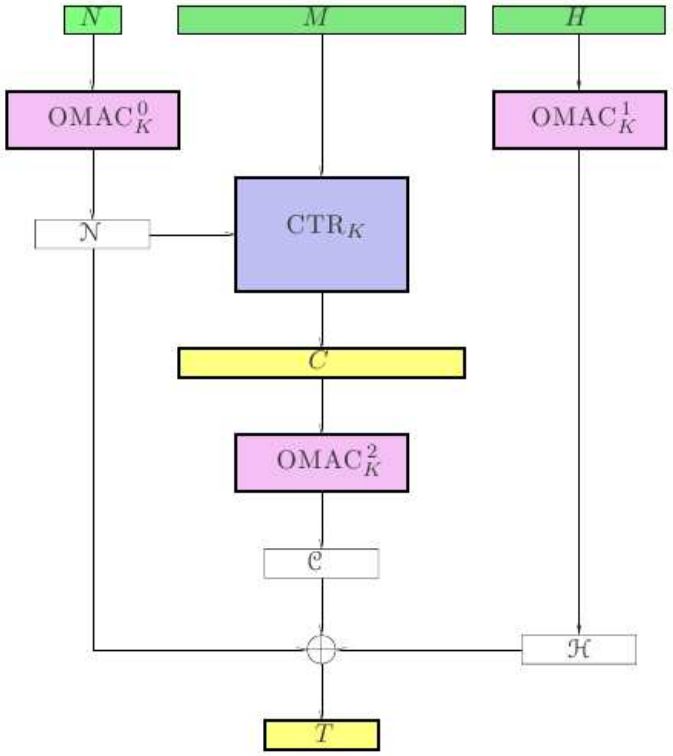

Figure 2.3: Encryption under EAX. The nonce is N, the message is M, the key is K, and the header is H. The ciphertext is $CT = C||T$

Reproduced with kind permission from Springer Science+Business Media: *Fast Software Encryption*, The EAX Mode of Operation, 2004, page 396, Mihir Bellare, Phillip Rogaway, and David Wagner, Fig. 3, ©Springer-Verlag Berlin / Heidelberg, 2004

2.7 Authenticated Encryption with EAX

Actual encryption and message authentication in SSMS is accomplished using the EAX block cipher mode for authenticated encryption. The EAX[**?**] mode of operation defines an authenticated encryption scheme, the security of which rests on a single cryptographic primitive, the block cipher. EAX is also attractive due to its ability to authenticate cleartext header data in addition to ciphertext data.

EAX takes as input a nonce N, an optional cleartext header H, and the message to be enciphered M, and produces the ciphertext C of M using the block cipher in counter mode, and a MAC tag computed on N, H, and M using the block cipher as an OMAC. The nonce, the header, the message, and the mac tag may all be of arbitrary length without breaking the security of the scheme.

For our authentication and encryption needs, we will be using the full message index as a nonce

to seed the counter in AES counter mode. EAX also includes the nonce in the MAC tag calculation, meaning that messages will be authenticated based on client state in addition to payload content. 2.3

$$N = i_m = 2^8 \cdot \texttt{ROLL} + \texttt{SEQ} \tag{2.9}$$

$$H = \text{User Data Header} \tag{2.10}$$

$$M = \text{Datalen}||\text{Plaintext}||\text{Null Padding} \tag{2.11}$$

Finally, we set the length of the MAC tag to 24 bits for medium strength authentication. Insufficient authentication would be a significant security problem for two reasons. First, EAX encryption uses AES in counter mode, which means forging new ciphertexts without a MAC is easy. Second, weak authentication would increase the chances of successful denial of service attacks against the replay protection mechanism. However, at 8 billion attempts per forgery, a 24 bit MAC should provide sufficient security in most cases.

2.8 Message Processing

2.8.1 Variables

Each end point has the following state variables:

Message Count is the number of messages that have been encrypted under the master key. This must not exceed the rekey frequency. In the event that it is approaching the limit, a re-key event is signaled to the key agreement layer.

Sequence Number is a monotonically increasing 8 bit number, transmitted over the air along with the ciphertext.

Rollover Counter is a monotonically increasing 32 bit number, maintained by the end point.

S_1 (For receivers) is an integer of the highest authentic sequence number received.

2.8.2 Sending

Sending messages follows the following procedure:

1. Determine the session identifier based on destination address and port number.

2. Determine the index of the message based on rollover counter and sequence number.
3. Encrypt and Authenticate the message.
4. Advance session key by executing the KDF function on the current key.
5. Update the rollover counter if necessary.

2.8.3 Receiving

Recipt of an incoming message is processed in the following way:

1. Determine the session identifier based on source address and port number in packet.
2. Determine the index of the message:
3. Check if the message has been replayed. If the message has been replayed, discard, and log the event.
4. Verify the authentication tag. If verification fails, the message must be discarded, and the event logged. Warn the user of a security event.
5. Advance session key as needed, storing intermediate values of skipped messages for a short period of time.
6. Decrypt the encrypted portion of the message.
7. Update the rollover counter and highest sequence number. Update replay window by erasing keys.

Chapter 3

Key Agreement Protocol for SMS (KAPS)

The Key Agreement Protocol for SMS (KAPS) establishes an ephemeral shared secret using a minimal set of messages. KAPS uses the Elliptic Curve Diffie-Hellman primitive for share secret computation, with key continuity and one-time verbal authentication for man-in-the-middle detection. The key material that the protocol outputs can be used to key symmetric security protocols to such as SSMS in chapter 2 to complete the security scheme.

KAPS has the advantage of being completely peer-to-peer and ephemeral. No trusted third parties are required, and no long term secrets must be safeguarded against accidental exposure.

The protocol begins with the initiator sending a hash commitment on a freshly generated ECDH public key to the responder. The responder then is required to generate and send back his public key without knowledge of the initiator's public key. Likewise the initiator was required to generate her public key without knowledge of the responder's key. This commitment scheme prevents an adversary from controlling the result of the Diffie-Hellman computation.

Hashes of a cached secret are also included in the first two messages. The cached secret is a bit of key material saved from previous executions of the protocol, if applicable (in the absence of a cached secret a random value is sent). This allows the two parties to determine if they have shared key material that matches. It also allows them to discover when an adversary is attempting a man-in-the-middle attack by impersonating the other party, since the adversary would not have access to private key material from the previous session.

The last step of the protocol involves sending a message authentication code on a known value

to the other party, proving that the protocol completed successfully and that the sender has access to her private key. Once this has been confirmed, both parties refresh their cached secret with a new value derived from the freshly computed shared secret. This action breaks the advantage of an adversary who has previously compromised cached secrets. As soon as an authentic key agreement takes place, a new cached secret is generated and secrecy is restored to the protocol.

Once the protocol has completed, a Short Authentication String (SAS) is displayed to the user to verify the absence of a man-in-the-middle attack. Due to key continuity, the user may verify the SAS with the other party at his or her leisure. By using an out-of-band channel such as a phone call, the user gains confidence in the identity of the other party based on a number of clues rooted in human interaction. An attacker would need to duplicate all these clues when the two parties were checking the SAS and remain undetected while leaving the rest of the conversation undistorted in order to be successful.

Figure 3.1 displays the protocol and its messages.

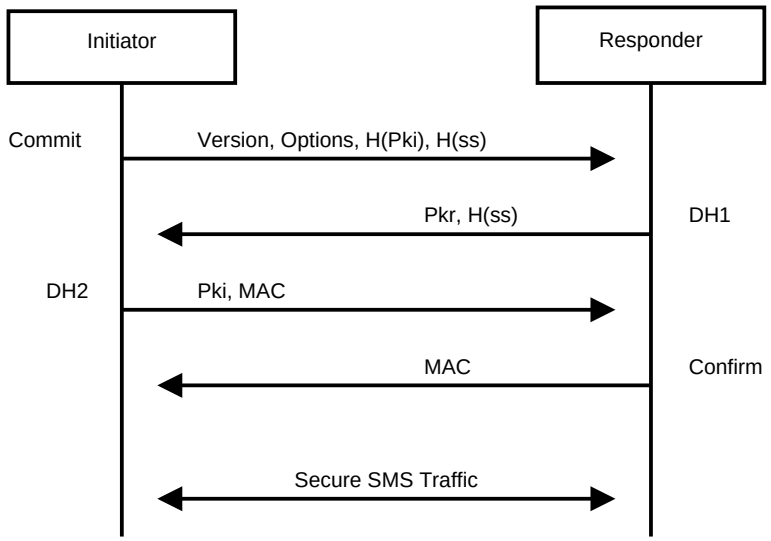

Figure 3.1: KAPS Protocol Diagram

3.1 Retransmission

This protocol is driven by the initiator. Each reply from the responder is an implicit acknowledgement of the prior message. In the event that a message gets lost in transmission, the initiator may resend the last message. Since the Short Message Service provides "best effort delivery" it is recommended that retransmissions be sent at the user's discretion. Sometimes carriers drop messages at

network transfer points, the recipient's device may be off, or some other structural condition may prevent messages from reaching their target. Automatic retransmissions in these cases would simply waste digital as well as monetary resources.

If no DH1 reply is received when initiating a conversation with a new end point, an initiator may elect to send a traditional text message expressing the initiator's wish to start a secure SMS conversation and an installation link.

3.2 Version Negotiation

For the purposes of future protocol refinement, the commit message contains a version number. If a responder does not support the version number in the commit message, the responder simply ignores the commit message.

The version number is formatted as two nibbles: `0xMm` the major version number followed by the minor version number. End points with the same major and minor version numbers are considered interoperable

This first version of the protocol described here is 0.1 (0x01).

3.3 Algorithms and Ciphers

In KAPS, the initiator specifies what algorithms to use in the commit message. If the responder does not support the algorithms requested, the responder ignores that commit message. For version 0.1, only a single option is permitted for some of the algorithm choices. Table 3.1 describes the options and their meanings. Rows in *italic* must be supported by endpoints, but may not be requested by the initiator until a later time.

The commit packet also contains algorithm choices for the message encryption layer – SSMS in this case. These are listed below the second horizontal line.

3.4 Hash Commitment

The hash commitment forces the adversary to select a public key without knowing the other party's key. This restricts the adversary to one attempt at finding a collision in the short authentication string (SAS), thus allowing the SAS to be much shorter than it otherwise would need to be. Without the hash commitment, the adversary could acquire the public keys for his two victims before searching

Use	Value	Meaning	Size
Port Number	16474		16 bit application port number
Version	0x01	0.1	
Key Agreement	0x02	ECDH/NIST P-384	384 bits
	0x01	*ECDH/NIST P-256*	*256 bits*
MAC	0x01	HMAC-SKEIN	512 bits
Hash	0x01	Skein-512	512 bits
SAS	0x02	NATO	4 words
	0x01	Z-Base-32[?]	4 characters
Block Cipher	0x01	AES-256	256 bits
AuthTag	*0x03*	*OMAC/32 bits*	*32 bits*
	0x02	OMAC/24 bits	24 bits
	0x01	*OMAC/16 bits*	*16 bits*

Table 3.1: KAPS Version and Options

for his own pair of keys that would result in an SAS collision. Such a collision would allow the adversary to setup two agreements that generated the same SAS string on both victim devices for a successful and undetected man-in-the-middle attack.

This attack is restricted in KAPS by forcing the adversary to select a public key prior to knowing the public key of the other party. Without the other key, the adversary cannot predict what the SAS value will be, and is forced to make a single blind guess.

The hash commitment H_c is checked when the initiator sends his public key in DH2. If the check fails, a man-in-the-middle attack is in progress. The user is warned, and the protocol stopped.

The hash commitment is performed on the Pk_i as it is sent in the DH2 message – using network byte order with BER encoding.

$$H_c = H(Pk_i) \qquad \text{(First 64 bits)} \tag{3.1}$$

3.5 Commit Contention

Due to the high latency nature of the SMS network, two end points may attempt to initiate a key negotiation at the same time. Each end point may send the other a commit message before receiving the commit message of the former. This contest is broken by comparing the hash values of the hash commitment in big endian integer format, and discarding the message with the lower value. The side that sent the commit with the higher value becomes the initiator and the other side, the responder.

3.6 Cached Secret Comparison

Each endpoint stores a long term cached secret for each party it has successfully executed the protocol with in the past. These cached secrets are tied into the key derivation function to achieve key continuity. Key continuity give us confidence that the end point with which were are communicating with today is the same entity we communicated with last week. Key continuity has been used in a number of protocols including SSH[**?**].

To determine if two parties have cached secrets (`cs`), each party sends a non-invertible hash of their cached secret to the other. The initiator does this in the commit message, and the responder includes it in the the DH1 message. If no cached secret is available, a random value is substituted for `cs` to avoid leaking information about the cache state. H_{csi} is sent in the commit message and H_{csr} is sent in DH1. To further avoid leaking information about the state of the cached secret, the MAC is computed on a salt available in the commit and DH1 messages.

$$H_{csi} = \mathsf{MAC}(\mathtt{cs}, H(Pk_i)||\text{“Initiator”}) \quad \text{(First 64 bits)} \tag{3.2}$$

$$H_{csr} = \mathsf{MAC}(\mathtt{cs}, H(Pk_r)||\text{“Responder”}) \quad \text{(First 64 bits)} \tag{3.3}$$

End points compute these values locally and compare them against the values received from the other party to determine the presence of a cached secret. If the hashes match, the cached secret exists and is shared between both parties. If they don't, a null is used in place of `cs` in the key derivation function in section 3.9.3

If a device has a cached secret stored for a particular party, the cached secret comparison is expected to succeed. If the comparison fails when a cached secret is available, one of two cases may be in play: one of the parties may have lost their secrets due to a device reset, or there may be a man-in-the-middle (MiTM) attack in progress. In the case of a mismatch, the user must be warned that a MiTM attack may be underway, and advise the user to verify the short authentication string as soon as possible to verify that the MiTM is not present.

If a mismatch occurs, the cache is **not** updated until after the user has verified the short authentication string (SAS) with the other party. The user should be warned of this condition on every key agreement until the condition has been resolved.

When no attacker is suspected, the cached secret cache is updated with a new value after a

successful key agreement has been confirmed by receipt of confirmation MACs.

$$\mathsf{cs} = \mathsf{KDF}(Z, \text{“RetainedSecret”}, \text{Context}, 512) \tag{3.4}$$

Because previous cached secret key material is mixed into the key derivation, a successful attack on key negotiation must involve both a compromise of the cached secrets on the device as well as a man in the middle attack on the next key negotiation. If the adversary misses just one key negotiation, the cached secrets are replaced with fresh, secret values and the protocol self-heals.

3.7 Short Authentication String (SAS)

The Short Authentication String (SAS) is the first 20 bits of the SAS hash. The commit message includes a designation of what SAS rendering scheme to use:

0x01 The SAS is displayed to the user as a 4 character base 32 string using the encoding described in section 5.1.6 of RFC 6189[**?**]. That encoding scheme is designed for ease of human use. The output alphabet is all lower case, excludes characters that are easily confused, and padding characters are left out.

0x02 The the 4 characters of the prior scheme are displayed to the user using the NATO phonetic alphabet as an aid to auditory clarity in verbal confirmation. If the end-point interface is not large enough for NATO words, however, this option downgrades to the first option.

$$K_{sas} = \mathsf{KDF}(K_{dk2}, \text{“SAS”}, \text{Context}, 20) \tag{3.5}$$

$$\text{0x01 SAS} = \text{Z-Base-32}(K_{sas}) \tag{3.6}$$

$$\text{0x02 SAS} = \text{NATO}(\text{Z-Base-32}(K_{sas})) \tag{3.7}$$

It is recommended that users make use of an out-of-band channel in which they can establish the identity of the receiving party and verify that they have the same SAS value. A simple phone call would be sufficient for this purpose since confidence building cues such as voice timbre and manner of speech are present. These cues concurrently make it difficult for an adversary to convincingly impersonate the remote party without being detected. If the two parties are physically co-located, they may even be able to visually compare their short authentication strings with their devices

juxtaposed.

Because the verification of the Short Authentication String cannot be automated, the protocol's security is dependent on the initiative of the end user. This opens a window of vulnerability for lackadaisical users: an adversary may successfully perform a man-in-the-middle attack without detection until the first comparison of the Short Authentication String.

This vulnerability is offset by several considerations. The first is that the key continuity properties require the presence of the adversary during every key agreement. An absence during just one key negotiation would alert the user that key continuity has been broken, and subsequent key negotiations would be secure against undetected man-in-the-middle attacks. The second implication from key continuity is that a successful verification of the Short Authentication String means that all previous key agreements have succeed without interference from the adversary.

3.8 Partial Public Key Validation

Before using the public keys in DH1 and DH2 for computation, they must first be verified. Public keys must have the correct modulus and not be the point at infinity. Full validation would also check that the public key is in the same subgroup as our ECC domain parameter curve, but this is an expensive operation for resource constrained devices.

The validation routing is performed as specified in 5.6.2.6 of NIST Special Publication 800-56A[**?**]. If any of the checks fail, the user should be alerted that a weak key attack is under way, and the protocol must be terminated.

1. Q must not be the point at infinity
2. Q must be in the valid range of the elliptic curve group.

3.9 Key Agreement

The shared secret is computed using the process described in section 6.1.2.2 of NIST Special Publication 800-56A[**?**].

3.9.1 Master Key Derivation

Once the public key from the other party has been verified, the Elliptic Curve Cryptography Cofactor Diffie-Hellman (ECC CDH) Primitive is used to compute the shared secret Z as specified in section

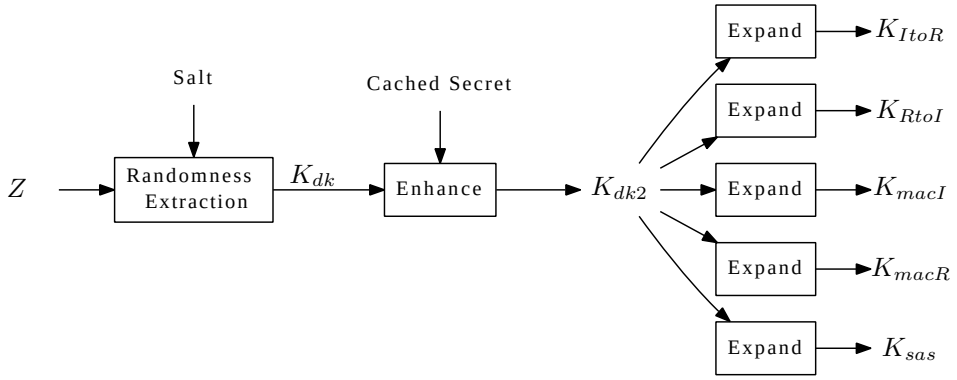

Figure 3.2: Extraction, then Expansion of Key Material

5.7.1.2 of NIST SP 800-56A. Given (q, FR, a, b, G, n, h)[1] as domain parameters, d_A, one's own private key and Q_B, the other party's public key, compute Z:

$$P = hd_AQ_B \tag{3.8}$$

$$Z = x_p(\text{where } x_p \text{ is the x coordinate of } P) \tag{3.9}$$

The next task is to convert Z, which is a field element, into a random bit string suitable for keying material. A naïve approach might be to apply a hash function to Z, but this is problematic for two reasons: Hash functions are not guaranteed to have Pseudo-Random Function (PRF) properties, only collision and pre-image resistance. Secondly, hash functions need to be computed on random values in order to guarantee the randomness of their outputs. If the input is not random, the output of a static hash function can be easily reversed. Therefore, Krawczyk[**?**] and a draft Special Publication 800-56C[**?**] from NIST separate randomness extraction and key expansion into two separate steps.

3.9.2 Extract

Z is transformed into a random key derivation key K_{dk} using a MAC keyed with a salt value. KAPS uses a hash of all the messages sent and received H_{total} for the salt.

$$H_{total} = H(\text{commit}||\text{DH1}||\text{DH2}||\text{confirm}) \tag{3.10}$$

$$K_{dk} = \text{MAC}(H_{total}, Z) \tag{3.11}$$

[1] q is the field size, FR indicates the basis used, a and b are the field elements that define the curve, G is the generating point, n is the order of G, and h is the cofactor, which is equal to the order of the curve divided by n as specified in NIST SP 800-56A[**?**]

3.9.3 Enhance

Before the key derivation key is ready to be expanded for application specific purposes, the cached secret must be mixed in. The mixing is performed using the same function as the expand step, only with cached secrets being used in the context field.

$$K_{dk2} = \mathsf{MAC}(K_{dk}, 1||\text{“MasterSecret”}||0x00||\text{MasterContext}||512) \tag{3.12}$$

$$\text{MasterContext} = \text{AlgorithmID}||\text{PartyUInfo}||\text{PartyVInfo}||\text{SuppPubInfo}||\text{SuppPrivInfo} \tag{3.13}$$

Algorithm ID describes the algorithm: `KAPS-ENHANCE`.

PartyUInfo is identifying information for the initiator:

`len(U phone No)||U phone No||:16474`

PartyVInfo is identifying information for the responder:

`len(V phone No)||V phone No||:16474`

SuppPubInfo is a nonce for which H_{total} will suffice.

SuppPrivInfo is the cached secret (if it exists): `len(cs) || cs`. If no cached secret exists for this agreement, the length of `cs` is 0 followed by nothing: `0`

3.9.4 Expand

Finally, K_{dk2} is expanded using the KDF for KAPS to generate a separate master session key for each direction of the secure symmetric conversation. The context in this case is a simplified version of the master context.

$$K_{ItoR} = \mathsf{KDF}(K_{dk2}, \text{“InitiatorMasterKey”}, \text{Context}, 256) \tag{3.14}$$

$$K_{RtoI} = \mathsf{KDF}(K_{dk2}, \text{“ResponderMasterKey”}, \text{Context}, 256) \tag{3.15}$$

The KAPS key derivation function is defined as the L left most bits of the MAC computed in the following way, where L must not exceed the size of the MAC function's output:

$$\mathsf{KDF}(K, \text{Label}, \text{Context}, L) = \mathsf{MAC}(K, 1||\text{Label}||0x00||\text{Context}||L) \tag{3.16}$$

K is a secret random bit string.

L is the length in bits of the output key material encoded as a 4 octet integer.

Label identifies the purpose of the output key material.

Counter is required by NIST Special Publication 800-108[**?**] and is always 1 since we limit the output length of the KDF to be less than the output length of the MAC. The counter is encoded as a 4 octet integer.

Context is a binary string that ties the output key material to the particular situation in which it is being used.

$$\texttt{Context} = \texttt{AlgorithmID}||\texttt{PartyUInfo}||\texttt{PartyVInfo}||\texttt{SuppPubInfo} \tag{3.17}$$

Algorithm ID describes the algorithm: `KAPS-KDF`.

PartyUInfo is identifying information for the initiator:

`len(U phone No)||U phone No||:16474`

PartyVInfo is identifying information for the responder:

`len(V phone No)||V phone No||:16474`

SuppPubInfo is a nonce for which H_{total} will suffice.

As soon as intermediate keys K_{dk}, K_{dk2}, and other intermediate key material is no longer needed, or if there is an error they must be erased from memory.

3.10 Key Confirmation

The last step in the protocol involves giving the other party assurance that the protocol completed successfully and that he is in possession of the correct symmetric key. This is accomplished by sending a MAC computed on a known value under a key derived from K_{dk2}. The initiator's confirmation is sent in the DH2 message, and the responders confirmation is sent in the Confirm message.

$$K_{macI} = \mathsf{KDF}(K_{dk2}, \text{“InitiatorMACKey”}, \text{Context}, 512) \tag{3.18}$$

$$K_{macR} = \mathsf{KDF}(K_{dk2}, \text{“ResponderMACKey”}, \text{Context}, 512) \tag{3.19}$$

$$\texttt{maci} = \mathsf{MAC}(K_{macI}, H_{total}) \qquad \text{(first 64 bits)} \tag{3.20}$$

$$\texttt{macr} = \mathsf{MAC}(K_{macR}, H_{total}) \qquad \text{(first 64 bits)} \tag{3.21}$$

After the key confirmation has been successfully validated, parties update their cached shared secret `cs`. (Provided a man in the middle attack was not suspected as noted in section 3.7) The previous shared secret is erased and replaced with the new shared secret as calculated in equation 3.4

3.11 Message Processing

3.11.1 Initiator

1. Generates a ECDH key pair.
2. Computes and send the hash commitment.
3. Waits for the responder's DH1 message. Retransmits as needed.
4. Verifies that shared secrets match as expected. If not, warn user of a man-in-the-middle attack.
5. Verifies that the responder's public key is valid using the algorithm specified in section 3.8.
6. Computes master and session keys.
7. Sends DH2 with confirmation MAC. Retransmits as needed.
8. Waits for the responder's confirmation (and/or the first valid SSMS message)
9. Verifies the confirmation code.
10. Updates shared secrets.
11. Starts sending messages!

3.11.2 Responder

1. Receives the hash commitment from the initiator.
2. Generates a ECDH key pair, sends the public key.
3. Receives the initiator's DH2 public key.
4. Verifies the hash commitment received earlier.
5. Verifies that the shared secrets match as expected. If not, warn the user of a man-in-the-middle attack.
6. Verifies the initiator's public key.
7. Compute master and session keys.
8. Verifies the confirmation code.
9. Update shared secrets.
10. Sends a confirmation code.

11. Starts sending messages!

Chapter 4

Implementation

The full Java implementation of the protocols described above has not been completely implemented at this time due to scheduling constraints. However, the implementation challenges that have been addressed are described below.

4.1 J2ME Wireless Messaging API

Since the target audience of these protocols is assumed to be using basic phones, the widest platform for delivery is the Java 2 Platform, Micro Edition (J2ME). J2ME however intentionally prevents applications from reading traditional SMS messages destined for the inbox. The solution was to use GSM's application port numbers, which Java can not only read, but can also trigger specific applications on receipt of messages with a given port number.

The disadvantage of application port numbers is that they are specific to the GSM standard. CDMA networks also support text messaging, but via a very different protocol. When a text message passes from one network to another, the message is translated by an interworking gateway which may only choose to translate the most popular pieces pieces of the protocol, leaving less used features like application port numbers or 8-bit binary encoding unsupported. Unfortunately in practice, there is no indication of failure when sending unsupported messages. Messages are simply dropped.

4.2 Secure Erasure of Java Objects

Best security practices require that encryption keys be erased as soon as they are no longer needed. However, due to Java's use of immutable objects key erasure is difficult; for no immutable object

may be overwritten with a new value. Furthermore, Java's garbage collector is free to copy objects around in memory as needed, potentially leaving many copies of key material lying about memory.

Byte arrays offer a bit of hope since they are mutable and can be passed by reference. In this project, byte arrays have been used to store key material whenever possible per Java security recommendations[**?**, **?**]. Key material that was not eligible for erasure (such as the use of Java's immutable `BigInteger` to compute Diffie-Hellman shared secrets) has been isolated in separate classes and marked with code comments for future solutions.

Upon further inspection, the Garbage Collector also tends to avoid moving objects around in memory due to the cost of changing all the object references. It only makes sense to execute memory compaction when the heap is very close to being completely consumed. The IBM JVM for instance only triggers memory compaction when less than 5% of the active heap is free, or when less than 128Kb of the active heap is free, in addition to some other low memory conditions[**?**]. The implication for key material is that there is a good probability that phantom data left over from an array copy during the compaction stage will also be overwritten with other objects being compacted or new objects being allocated.

Work needs to be done at the language specification level in order to fully address this problem. An `erasable` keyword or a secure object analogous to .NET's `SecureString` could be introduced into the language so that key material may be reliably erased in the future.

The precautions outlined above are needed to avoid leaking unnecessary key material when the device is forensically examined. However, continuous, co-resident memory analysis by custom malware is a powerful attack that at present we do not attempt to defend against.

4.3 Entropy Sources

The importance of acquiring high quality, cryptographically strong random numbers can not be over stated for the security of the protocols presented above. If the nonces or random private keys can be predicted in any way, the security of the protocol will be directly impaired. The random number generator to be used MUST be initialized with an unpredictable, non-deterministic, physical source of entropy – and regularly re-seeded with enough entropy to ensure the security of the output bit stream.

Good physical sources of entropy are abundant in mobile devices[**?**] and include the microphone, the digital camera (if available), network traffic, and user behavior. Higher quality sources such as the SIM card random generator and RF noise are also present but are generally unavailable due to

API restrictions.

Chapter 5

Conclusions

The protocols described above provide end-to-end confidentiality, integrity, and partial authenticity for text messages without the need for additional hardware, trusted parties, or network infrastructure. Parties will be able to have confidence that their messages are opaque to the network and ephemeral, modeling private conversations in a natural and efficient way.

The protocols described here, however, are not mature and would be enhanced by future work in the following areas:

5.1 Future Work

1. **Data at rest** The security of stored messages data on the mobile device itself will need to be protected against search and seizure.

2. **Software Update** Security flaws and bug patching are the nature of software security. A secure life cycle model for mobile application deployment needs to be developed in order for a secure product to be fully realized.

3. **Device Disambiguation** This first version of the protocol does not include a way to distinguish between devices that are using the same phone number to communicate. If the protocol is used in cases that involve a PBX or a call forwarding service such as Google Voice™, the protocol will interpret the other devices as adversaries and raise false intrusion alarms. An additional identity field or dynamic port number selection could be possible solutions.

4. **Two Generals Problem** If a devices goes off line after `DH2` has been sent, but before the `confirm` message has been sent, the state of one cache will be updated but the other party's

will not be. Subsequent key agreements will result in a shared secret mismatch when a match was expected. This will raise a false security alarm. Additional shared secrets could be stored to allow smooth recovery from one hiccup.

Appendix A

KAPS Message Formats

This appendix describes the formats of messages used in KAPS.

KAPS messages are distinguished from other messages on the same port via a "magic cookie". The magic cookie is structured such that it not will accidentally appear during SSMS traffic with a probability greater than $1/2^{64}$.

After the magic cookie, a single octet identifies the type of message that is in the rest of the payload. The general format of KAPS is found in figure A.1, and the values for the message type are listed in table A.1.

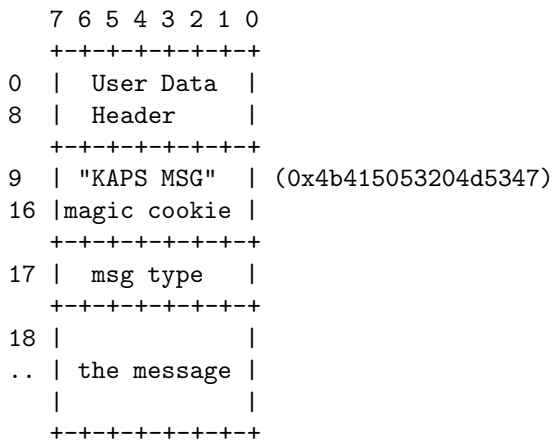

Figure A.1: KAPS general message layout

Value	Meaning
0x00	Commit Message
0x01	DH1
0x02	DH2
0x03	Confirm

Table A.1: KAPS message types

A.0.1 Commit

The commit message in figure A.2 is sent to initiate the KAPS protocol. The commit message may be sent at any time, granted another key agreement is not already in process. The initiator selects the algorithms and ciphers to be used. The values and meanings for each option are specified in table 3.1.

Version comes first so that future versions may adjust the formats of KAPS messages.

Key Agreement selects the ECC domain parameters and key size from the NIST standard curves.

MAC specifies the MAC algorithm that will be used during key derivation and wherever else a MAC is needed.

Hash specifies the hash function that will be used for computing hash commitments and the total hash. Hash is also used by SSMS to obfuscate endpoint identities in memory.

SAS rendering selects the style of SAS to display to the user.

Block Cipher selects the block cipher that will be used in the message security layer (i.e. SSMS)

Auth Tag selects the type and the size of auth tag to use on messages in the message security layer (i.e. SSMS)

H(Pki) is the hash commitment by the initiator.

H(cs) is a hash of the initiator's cached secret.

A.0.2 DH1

When the responder receives a commit message, he generates a new, random ECDH key pair. The DH1 message in figure A.4 is sent from the responder to the initiator. It contains the responser's public key and a hash of his cached secret.

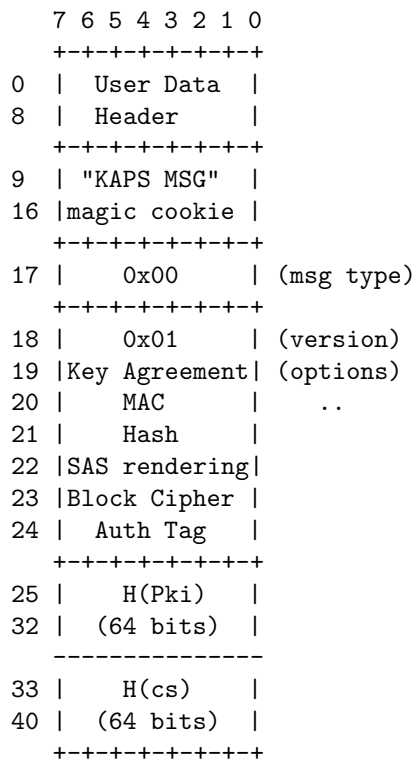

Figure A.2: KAPS Commit message layout

Pkr is the public key of the responder, sent using BER encoding. The length of Pkr is implicitly set by the choice of EC domain parameters given in the commit message.

H(cs) is the hash of the responder's cached secret. A random value is sent if no secret is present.

A.0.3 DH2

After receiving DH1, the initiator has enough information to derive session keys, which the initiator does after first validating the public key of the responder. The initiator then sends back her public key and a MAC that the responder can use to verify that both parties successfully computed the same session keys.

Pki is the public key of the initiator, sent using BER encoding. The length of Pkr is implicitly set by the choice of EC domain parameters given in the commit message.

MAC is the confirmation code described in section 3.10.

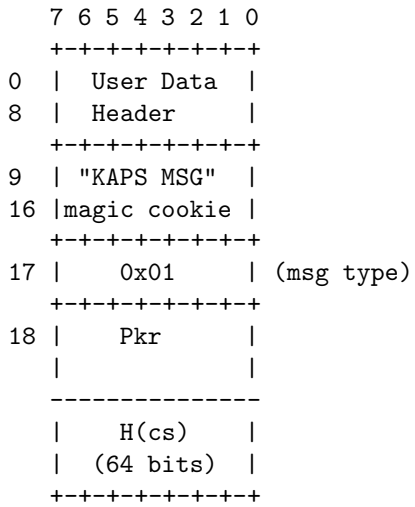

Figure A.3: KAPS DH1 message layout

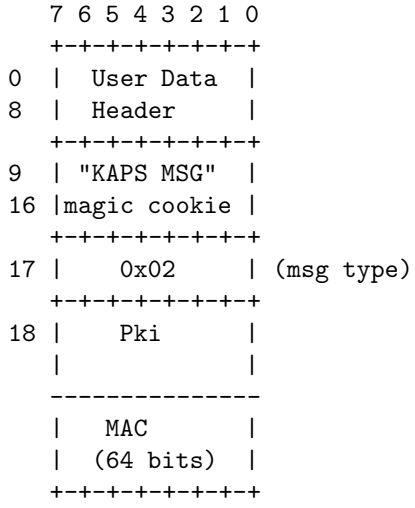

Figure A.4: KAPS DH2 message layout

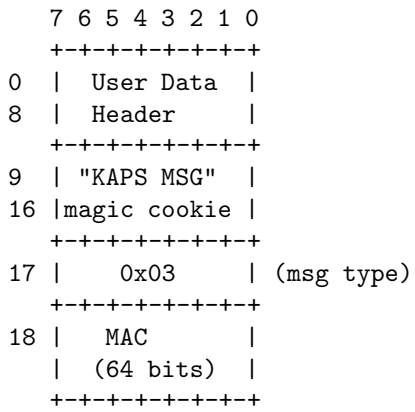

Figure A.5: KAPS Confirm message layout

A.0.4 Confirm

The responder verifies that the initiator's public key is valid, computes the session keys and responds with a confirmation code the the initiator can use to verify that the protocol completed successfully.

MAC is the confirmation code described in section 3.10.

Appendix B

Short Message Service Summary

This appendix provides a short reference guide to the basics of the short message service. A short analysis of SMS security and reliability is given, followed by a description of the network level messages involved in the transmission of a user's text. Extra details are included for the main deliver and submit messages.

B.1 Reliability

SMS rides on the SS7 protocol, which is an out-of-band channel typically used for call setup, teardown, and other control services. As such it benefits from the reliable transport and delivery services of the Message Transfer Part (MTP) of the SS7 protocol[**?**]. Messages are delivered on a "best effort" basis, but in practice messages are rarely dropped.

B.2 Security

CDMA networks (e.g. Verizon, Sprint PCS) are generally not encrypted, but rely on the long chipping codes and spread spectrum transmissions that make eavesdropping on the wireless link difficult. If at some point the future, they turned on true encryption, SMS messages would be automatically protected on the wireless link.

GSM networks provide the most security on the wireless link through the A5 encryption algorithm, though the A5 algorithms can be broken by an attacker in 3 to 5 minutes[**?**]. CDMA networks (e.g. Verizon, Sprint PCS) are generally not encrypted, but rely on the long chipping codes and spread spectrum transmissions that make eavesdropping on the wireless link difficult. If at some

point the future, they turned on true encryption, SMS messages would be automatically protected on the wireless link.

Regardless of the relative security of the wireless link between the mobile device and the cell tower, the Short Message Service is unprotected once it reaches the cell tower. The message is sent without protection across the carrier's network, leaving it open to interception and modification by the carrier or well-placed attackers. In order to protect the confidentiality of the SMS messages, they must be encrypted before entering the network, and decrypted upon receipt on the other end[**?**].

B.3 Messages

Several different types of SMS messages are used to ensure the reliable delivery of the user's message[**?**].

1. **SMS-DELIVER**, conveying a short message from the SC to the MS;
2. **SMS-DELIVER-REPORT**, conveying a failure cause (if necessary);
3. **SMS-SUBMIT**, conveying a short message from the MS to the SC;
4. **SMS-SUBMIT-REPORT**, conveying a failure cause (if necessary);
5. **SMS-STATUS-REPORT**, conveying a status report from the SC to the MS;
6. **SMS-COMMAND**, conveying a command from the MS to the SC.

B.3.1 SMS-Deliver

a **TP-MTI** Message Type Indicator.

b **TP-MMS** More Messages to Send flag. Indicates whether there are more messages to send

c **TP-SRI** Status Report Indicator. Indicates if the SME has requested a status report.

d **TP-UDHI** User Data Header Indicator. Indicates whether the user data header exists. In our case, it is present.

e **TP-RP** Reply Path flag. Indicates whether the reply path exists.

1. **TP-OA** Originating Address.
2. **TP-PID** Protocol Identifier.
3. **TP-DCS** Data Encoding Scheme. This will be 8 bit data for our case.
4. **TP-SCTS** Service Center Time Stamp.
5. **TP-UDL** User Data Length.

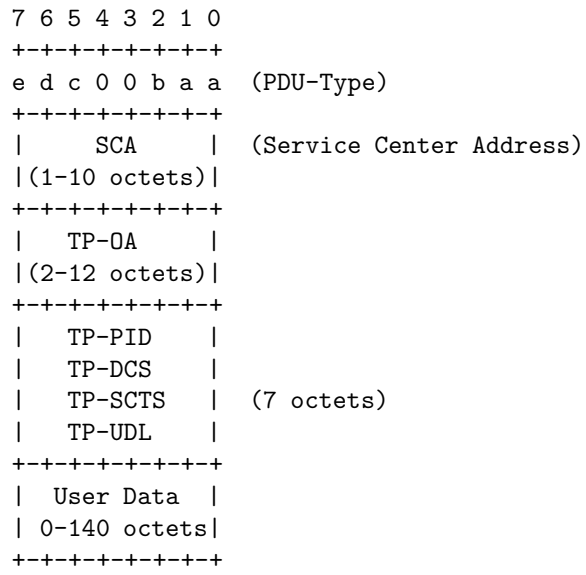

Figure B.1: SMS Deliver Layout

B.3.2 SMS-Submit

a **TP-MTI** Message Type Indicator.

b **TP-RD** Reject Duplicates. Indicates whether the service center should accept an SMS-Submit for a Short Message still held at the service center (SC) which has the same Message Reference (MR) and the same destination address (TP-DA)

c **TP-VPF** Validity Period Format. Indicates whether the validity period is present.

d **TP-SRR** Status Report Request. Indicates if the mobile station (MS) is requesting a status report

e **TP-UDHI** User Data Header Indicator. Indicates that the TP-UD field contains a header.

f **TP-RP Reply** Path. Indicates the request for reply path.

1. **TP-MR** Message Reference. An integer identifying the SMS-Submit.
2. **TP-DA** Destination Address.
3. **TP-PID** Protocol Identifier. Identifies the protocol in the above layer, if any
4. **TP-DCS** Data Coding Scheme. Identifies the
5. **TP-VP** Validity Period.
6. **TP-UDL** User Data Length.

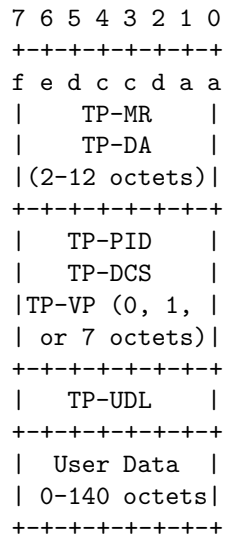

Figure B.2: SMS Submit Layout

B.3.3 User Data Header

1. **UDHL** User Data Header Length.
2. **IEIa** Information Element Identifier. A hex value of 0x05 indicates a 16 bit application port address information element.
3. **IEIDLa** Length of information element. 2 octets are required for the 16 bit addressing scheme
4. **IEDa** Information Element Data: a 16 bit destination port in big endian format as specified in GSM 03.40 subclause 9.1.2.1 [**?**]
5. **IEIb** Information Element Identifier. A hex value of 0x05 indicates an information element used for application port addressing scheme, 16 bit address
6. **IEIDLb** Length of information element. 2 octets are required for the 16 bit addressing scheme
7. **IEDb** Information Element Data: a 16 bit source port in big endian format as specified in GSM 03.40 subclause 9.1.2.1 [**?**]

Value			Meaning
0	–	15999	Reserved (Allocated by IANA)
16000	–	16999	Available to applications
17000	–	65535	Reserved

Table B.1: SMS Port Number Assignments

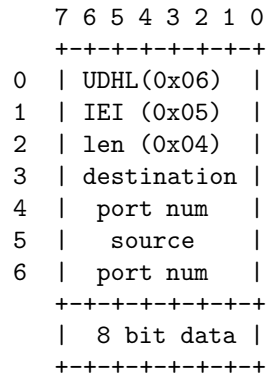

Figure B.3: User Data Header Layout