University of Waterloo

Software Engineering

Signing Messages for Mobile Devices

Lookout

San Francisco

Prepared by:
Stephen Li
Student ID: 0123456789
User ID: sli
3B Software Engineering
Sept 8, 2014

Stephen Li
200 University Ave. West
Waterloo, ON
N2L 3G1

Sept 8, 2014

Dr. Andrew Morton, Director
Software Engineering
University of Waterloo
200 University Ave. West
Waterloo, ON
N2L 3G1

Dear Dr. Morton:

This report, *Signing Messages for Mobile Devices*, is my second work term report. It details a design decision that I participated in while working on the iOS team at Lookout Inc. for my 4th work term.

Lookout is a mobile security company that provides its users with virus detection, data backup, and theft alerts for both iOS and Android. One of the projects that I worked on was investigating the feasibility of implementing digital signature verification on the iOS platform with a third party library.

This report describes the concept of digital signature schemes and investigates how feasible it is to implement and use them in our client applications and server. This report's target audience are the security, client, and server teams who ensure the security and stability of our platform, as well as any software engineer who are curious about applied cryptography.

I would like to thank all of my co-workers at Lookout for their support and advice throughout my work term. I would also like to thank the maintainer of the 3rd party library and my roommate for providing feedback on my report. I hereby confirm that I have received no help, other than what is mentioned above, in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

[Insert Signature Here]

Stephen Li
Student ID: 0123456789

Executive Summary

During Lookout's transition from a monolithic back-end to a service oriented architecture, the server team created the Micropush service to provide a uniform interface for communicating with iOS and Android devices. In order to ensure the authenticity of Lookout's messages sent to users' devices, the team had to choose to use either RSA or ECDSA signatures.

This report evaluates the two schemes based on their user impact, server impact, and ease of implementation. Although the benchmark results and projected costs mostly favour ECDSA, the ultimate decision was to use RSA. The primary reason for choosing RSA is because it is supported natively on both iOS and Android and thus is trivial to implement. In order to use ECDSA on Android, the team would need to implement and maintain a wrapper around a 3rd party C library.

In the future, it is recommended to re-evaluate the benchmarks and estimate the costs to see if it justifies the additional work of implementing ECDSA on Android.

Table of Contents

Executive Summary . . . . **iii**

List of Figures . . . . **vi**

List of Tables . . . . **vii**

1 Introduction **1**

2 Background **2**

2.1 Problem 3

2.2 Digital Signatures 4

2.2.1 RSA Signatures 5

2.2.2 DSA Signatures 6

2.3 Security Level 6

3 Evaluation **8**

3.1 Key Size 8

3.2 Signature Size 9

3.3 Speed 9

3.4 Server Costs 10

3.5 Ease of Implementation 10

3.6 Additional File Size 11

4 Conclusion **12**

5 Recommendations **13**

References **14**

Acknowledgements **15**

Appendix A Additional Details **16**

A.1 RSA 16

A.1.1 Key Generation 16
A.1.2 Signing Messages 16
A.1.3 Verifying Messages 17
A.2 ECDSA 17
A.2.1 Key Generation 17
A.2.2 Signing Messages 18
A.2.3 Verifying Messages 18
Appendix B Calculations **20**
B.1 Comparing Relative Security Levels 20
B.2 Signatures per Second 21
Appendix C Benchmark Details **22**
C.1 Signature Verification Benchmarks 22
C.2 Signature Generation Benchmarks 22
C.3 File Size of Library 22

List of Figures

2.1 Architecture of Micropush Service 3

2.2 Generating Digital Signatures 5

2.3 Verifying Digital Signatures 5

List of Tables

2.1 Comparing Relative Security Levels 7

3.1 Public Key Sizes and Their Corresponding Security Level 8

3.2 Signature Verification Benchmark Results (μs) 9

3.3 Average Number of Messages Signed Per Second 10

3.4 Additional File Size for Including `libsodium` (KB) 11

B.1 Time to Break Scheme 20

B.2 Time to Sign Messages (s) 21

B.3 Messages Signed Per Second (rounded down) 21

C.1 Size of Compiled `libsodium` Library for Android Architectures (KB) 23

C.2 Size of Compiled `libsodium` Library for iOS Architectures (KB) 23

1. Introduction

Lookout is a mobile security company that provides solutions for both consumers and enterprises. On the consumer side, Lookout has iOS and Android applications that protect their users from malware, data loss, and phone thefts.

Presently, there are many ways to communicate with mobile devices such as using Short Message Service (SMS), Apple Push Notification (APN), Google Cloud Messaging (GCM), or simple socket connections. In order to ease development for the back-end team and improve Lookout's ability to scale, a service called "Micropush" is developed. The goal for Micropush is to abstract away the details for the individual channels and provide back-end developers with an uniform interface to communicate with users' devices.

Some of the messages that a device receives from Micropush are commands such as asking for the device's current location or making the device play a loud alarm. Due to the sensitive nature of these commands, there needs to be a way to prevent others from posing as Lookout. Messages that are sent through APN for iOS and GCM for Android are already validated by Apple [1] and Google [2], respectively. However, the team is not comfortable in trusting Apple and Google with unrestricted access to these sensitive commands. With no protection, Apple, Google, or anyone with access to their servers can theoretically pose as Lookout and send these commands. In addition, these services are not always reliable as they require the clients to have cellular data or be connected to Wi-Fi. In the future, Lookout may also want to support other devices that do not have secure communication channels like APN or GCM. In order to maximize the reachability of Lookout's servers, Micropush must be able to send messages through other channels that are not as secure such as SMS or direct sockets. Therefore, messages sent by Micropush must be accomplied by what is known as a "digital signature" in the field of cryptography. A digital signature, like a hand signature, allows the recipient to check the authenticity of the received message.

This report will first explain Micropush in more detail. Next, this report will introduce and evaluate the digital signature schemes used in practice. Finally, this report will conclude with the team's decision and recommendations for the future.

It is expected that the reader is familiar with basic security and programming concepts. To fully understand the mathematics behind the signature schemes, it is recommended that the reader is also familiar with algebra concepts such as modulus arithmetic and prime numbers.

2. Background

Lookout started five years ago with a monolithic Ruby on Rails back-end where every consumer feature was running as a single unit. Currently, the back-end is very expensive to maintain in terms of both server costs and developer time because it is running on Lookout's own hardware and can easily fail. Some common issues that the server team have encountered in the past include networking issues at the data center, traffic spikes, and corrupt software updates.

As Lookout continues to gain more users and build more features, the team realized that a monolithic software architecture is unsustainable in the long run. As a result, there has been a major push in the engineering department to refactor the back-end into a service oriented architecture. The comparison of service oriented architecture versus monolithic architecture is beyond the scope of this report; however to summarize, a service oriented architecture simply decomposes the monolithic software into independent modules that communicate with each other through a high level interface.

From this initiative, the team planed to develop a service called Micropush for messaging users' devices. A simplified diagram of the control flow from Lookout's website and threat network to users' devices is shown in Figure 2.1. Essentially, Micropush is a queue that provides a uniform interface for Lookout's internal services to communicate with devices. It abstracts away the communication channels' details and allows developers to focus on their own work.

1. Lookout will send messages to a user's device when the user wants to locate or scream their device from the website control panel. Lookout will also send messages to everyone when Lookout's threat network publishes security updates such as when a new virus is detected or when a new software update is available.

2. These messages are sent to the Micropush service and are then queued up.

3. Micropush processes the messages and then dispatches them to the appropriate third-party service depending on what device the user have.

4. Third-party services such as APN and GCM then make a best-attempt to send the message to the user's device. If the device is turned off or is unreachable, then these services will store the message and retry later or give up after a certain time period.

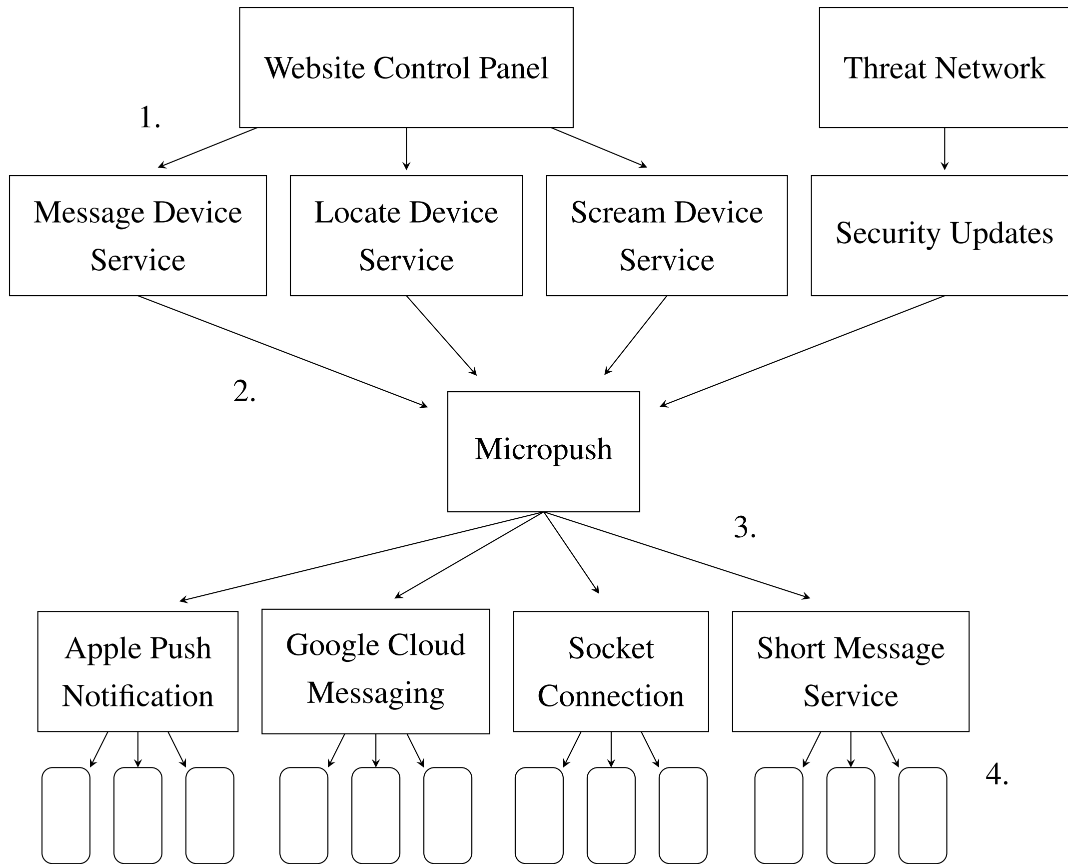

Figure 2.1: Architecture of Micropush Service

2.1 Problem

Since Lookout's servers send sensitive commands to users' devices, the Lookout application is a highly attractive target for attackers. For example, if there is no security then a sophisticated attacker can send a fake locate command to a user's device and get it to respond back with its location. With the recent revelations of the National Security Agency's (NSA) activities [3], it is vital for Lookout's international partnerships that users' devices can only accept commands from Lookout. Before formally defining the team's problem, the reader should be familiar with the four fundamental goals of cryptography [4]:

Confidentiality: To ensure that the transmitted message cannot be read by anyone except the intended recipient(s).

Data Integrity: To ensure that the original message has not been altered by an adversary or during transmission.

Data Origin Authentication: To ensure that the received message came from who the recipient thinks it came from, i.e. the recipient knows exactly who sent the message.

Non-repudiation: To ensure the sender cannot deny sending the message.

Of the four properties listed above, it is apparent that at the minimum Micropush needs to establish data integrity, data origin authentication, and non-repudiation. Firstly, the team wishes to establish data integrity because it is important that any commands the application accept are unmodified. Secondly, the team wishes to establish data origin authentication because it is important that only commands from Lookout's servers are accepted and processed by users' devices. Thirdly, the team wishes to establish non-repudiation because although only Lookout should be sending messages to users' devices, a competitor may wish to damage Lookout's reputation by maliciously sending *Scream Device* to themselves and claim it as a malfunction.

Note that Micropush does not necessarily require confidentiality because the messages that it will be transmitting may already be encrypted by other services before they are queued up to Micropush. The main requirement for Micropush is to guarantee that users' devices will only accept commands from Lookout.

2.2 Digital Signatures

Based on the formal requirements, it is apparent that Micropush needs to use a ”digital signature scheme” to satisfy all three requirements [4]. A signature scheme is essentially just an algorithm that takes in a binary input and a key, performs arithmetic on those inputs, and then outputs another binary sequence. When signing a message, the algorithm takes in the message and private key and outputs a signature; when verifying a signature, the algorithm takes in the signature and public key and outputs the message. The signing process is illustrated in Figure 2.2 and the verification process is illustrated in Figure 2.3.

In practice, there are two widely studied and used signature algorithms that can the team can choose from: RSA (named after its designers Ron Rivest, Adi Shamir, and Leonard Adleman) and DSA (Digital Signature Algorithm). Also note that, these signature algorithms have a size limit on the binary input so messages are generally first hashed[1] before it gets fed into the algorithm.

[1] It's important that the hashing algorithm used is ”cryptographically secure” such as those in the `SHA2` family.

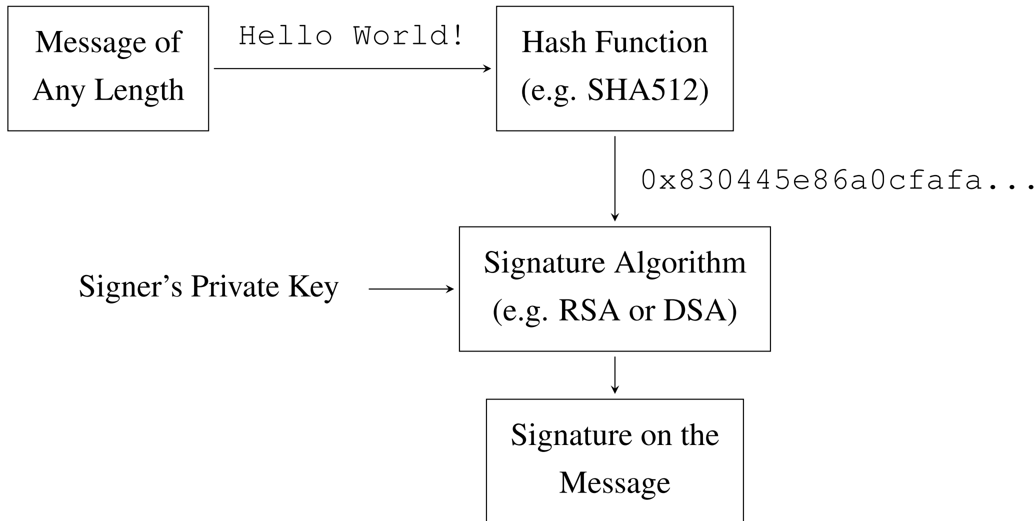

Figure 2.2: Generating Digital Signatures

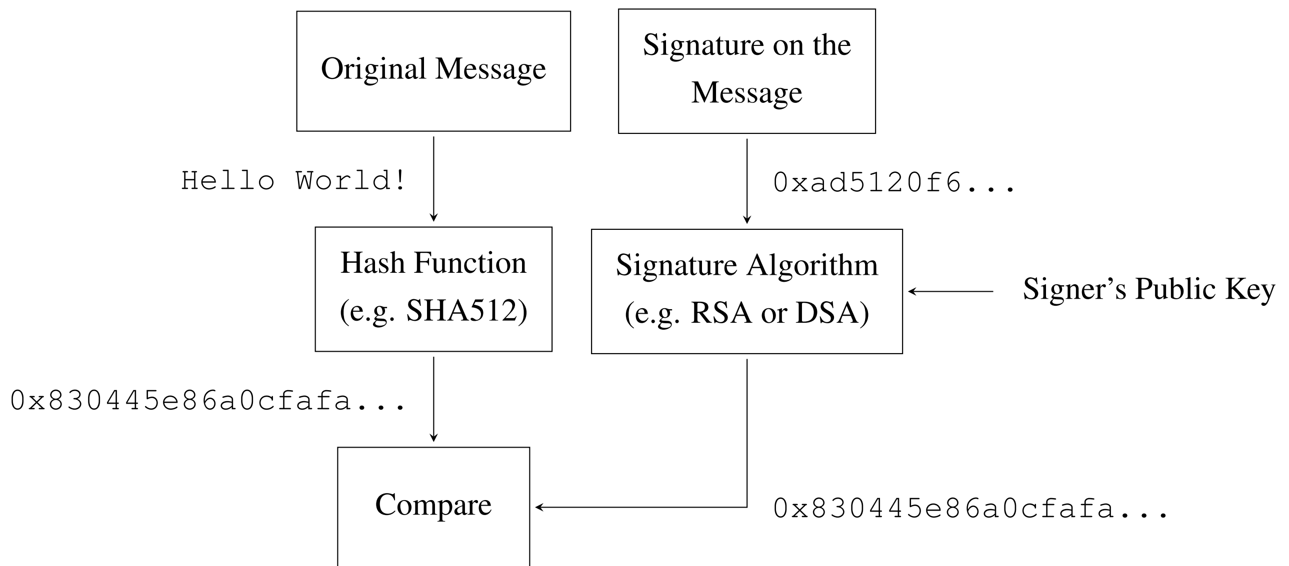

Figure 2.3: Verifying Digital Signatures

See Appendix A for more details about both schemes' key generation, signature generation, and signature verification steps.

2.2.1 RSA Signatures

The security of RSA is based on the Integer Factorization Problem (IFP): given the public key $n = pq$ where p and q are its prime factors, there are no efficient algorithms that can factor

n. If an attacker is able to factor n and obtain p and q, then it will be trivial for them to obtain the private key. In practice, RSA is natively supported on both iOS and Android so there is no additional work needed beyond calling the native library methods.

2.2.2 DSA Signatures

The security of DSA is based on the Discrete Log Problem (DLP): given the public parameters g and p and the public key $y = g^x \mod p$, there are no efficient algorithms that can obtain the private key x [5]. A variant of DSA is Elliptic Curve DSA (ECDSA) whose security is based on the Elliptic Curve DLP (ECDLP): given the public parameter Q, a point on an *elliptic curve*, adding it to itself x times (denoted xQ) is synonymous to calculating $g^x \mod p$ in regular DLP. The advantage of using ECDSA over DSA is that ECDSA uses smaller keys to achieve the same amount of security.

Although both DSA and ECDSA are supported by various open-source libraries for both iOS and Android, the team wants to avoid using any libraries that are using government published specifications because they may have NSA backdoors. After doing some research, the security team recommended to the team to use `libsodium` which implements ECDSA with `Curve25519`. This elliptic curve was designed by a famous and independent cryptography professor, Daniel J. Bernstein and is currently used in notable applications such as the iOS operating system[2] [6].

Note that for signing messages, `libsodium` actually uses the `Ed25519` curve whose ”birational equivalence” relationship with `Curve25519` [7] is beyond the scope of this report. Due to the exact definition of an *elliptic curve*, `Ed25519` falls under the category of ”EdDSA” instead of ”ECDSA”. Beyond semantics, there should be no difference for the reader. Therefore for the sake of simplicity, the remainder of this report will continue to use the term ”ECDSA”.

2.3 Security Level

One of the most important criteria to consider when choosing any key-based cryptography scheme, such as a digital signature scheme, is the size of the keys needed to achieve a desired security level. In the field of cryptography, an n-bit security level means that an attacker must perform approximately 2^n computations to ”break” the scheme, e.g. forging a valid signature or

[2] It is used internally so regular developers cannot access the library.

recovering the private key, etc. Table 2.1 lists the feasibility of attempting to run 2^n computations on a supercomputer (See Appendix B.1 for calculations).

Table 2.1: Comparing Relative Security Levels

Security Level	Feasibility	Time to break on a supercomputer
64	Feasible	Less than a year
80	Barely Infeasible	Less than a decade
128	Infeasible	When all of the stars in the universe exhaust their fuel supplies [8]
256	Extremely infeasible	When the universe only have black holes left

Although a 256-bit security level may seem impressive, the performance trade-off of achieving that security level is generally not worth it when a 128-bit security level is just as unbreakable. Since `libsodium` only supports the 128-bit security level and RSA's security level can easily be changed, the remainder of this report will only consider the two schemes based on the 128-bit security level.

3. Evaluation

When the team is choosing between RSA and ECDSA signatures, there are six criteria to consider. Since these two schemes' security are based on different mathematical problems, there are a couple of notable differences as described in the following sections to consider.

3.1 Key Size

For both RSA and ECDSA signature schemes, there is a private and public key pair. A private key, as its name implies, must remain private but a public key is known by everyone, including attackers. If an attacker can calculate the private key based on publicly available information, then they can forge a signature on any message they want. Therefore, it is important to choose a public key that is large enough to prevent attackers from computing the private key but is small enough to avoid wasting excessive resources.

Due to advances in mathematics, both the IFP and ECDLP have been weakened significantly from the expected strength of a brute-force attack. Table 3.1 lists the necessary public key sizes to achieve specific security levels [9]. These key sizes are based on the strongest algorithms for solving the generic IFP and ECDLP. There are much faster attacks on both problems that are tailored to specific prime numbers and elliptic curves that are not used in practice.

Table 3.1: Public Key Sizes and Their Corresponding Security Level

Security Level	RSA Key Size (bits)	ECDSA Key Size (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Since Micropush needs to establish a 128-bit security level, it needs to use a 3072-bit public key for RSA or a 256-bit public key for ECDSA. This also indicates the additional space the applications need to store the public keys.

3.2 Signature Size

By design, the size of RSA signatures is the same size as the RSA public key; and the size of ECDSA signatures is twice the size of the ECDSA public key. Note that the signature size remains constant regardless of the message length. In order to achieve the 128-bit security level, Micropush will need to send either a 3 KB RSA signature or a 0.5 KB ECDSA signature along with the original message. This will directly impact the servers' bandwidth costs as well as the users' data usage.

3.3 Speed

Since a device usually receive messages in the background, the iOS team was initially worried about the amount of time it takes to verify a signature. If there are several messages queued up, then verifying all of them may use up a significant portion of the 10 minute background time on iOS. Although Android allows unlimited background time, the Android team also wants to keep the background time down in the long run in order to save battery life.

By design, RSA is fast to verify but slow to sign; conversely, ECDSA is slow to verify but fast to sign. Since iOS and Android devices do not have similar hardware and runtime environments (compiled C code versus Java runtime), the performance of the two platforms cannot be directly compared. Instead, the iOS and Android teams benchmarked and evaluated the two schemes' verification speeds independently. The results are shown in Table 3.2:

Table 3.2: Signature Verification Benchmark Results (μs)

	Android (Nexus 4)	iOS (iPhone 5c)
RSA (3072-bit key)	617.50	63.30
ECDSA (256-bit key)	1139.35	26.85

On Android, the results match the hypothesis that RSA is faster to verify than ECDSA. However on iOS, the results seem to contradict the hypothesis. Without dissecting the generated assembly which is beyond the scope of this report, there are a few possible explanations for the disparity. It is possible that iOS imposes stricter penalties on data-dependencies, i.e. RSA performs arithmetic on 3 KB numbers whereas ECDSA performs arithmetic on 256-bit numbers. In addition, `libsodium`'s implementation of ECDSA is designed and optimized for 64-bit architecture so there may only be a slight penalty for running on the 32-bit architecture of iPhone 5c compared to

running the unoptimized RSA algorithm.

Regardless of the disparities, these results are significantly lower than what the client teams originally expected. Therefore in terms of verification speeds, the scheme that the team chooses should be inconsequential.

3.4 Server Costs

Although signing messages scale linearly with system resources (i.e. signatures are generated independently so the task can easily be parallelized), it is still important to minimize the server costs. On top of regular messages, adding signatures will impose an additional payload size (as described in Section 3.2) and computation cost. To investigate the impact on the server, the team benchmarked the schemes' signature verification speeds. The results are shown in Table 3.3 (See Appendix B.2 for calculations and see Section 3.3 for explanations on the disparity).

Table 3.3: Average Number of Messages Signed Per Second

RSA (3072-bit key)	209
ECDSA (256-bit key)	14030

From this benchmark, the team infers that generating ECDSA signatures is at least over 50 times faster than generating RSA signatures. This also implies that Lookout will have to invest at least 50 times more server resources for generating RSA signatures compared to generating ECDSA signatures.

3.5 Ease of Implementation

Since RSA is implemented natively for both iOS and Android, it is trivial to implement as there are plenty of tutorials and code samples online. However to use ECDSA, every team needs to use `libsodium`. On iOS, it is also simple to implement because the library, written in C, can be compiled directly to the platform to run. On Android, it is not as simple because the team needs to build a Java Native Interface (JNI) around the library to be able to access its C methods in Java code. Due to the sheer size of the library, the team decided to use a tool called SWIG[1] to

[1] `http://www.swig.org`

automatically generate the wrapper. However, it is possible that this generated code may not be as secure and will require a security audit before it can be used in production.

3.6 Additional File Size

Since users are more aversive to installing large applications, the client teams are very conservative when including third party libraries. Fortunately, RSA is natively supported on both iOS (in `CommonCrypto` library) and Android (in `java.security` library). If the team chooses RSA, the applications will not have a noticeable file size increase. However if the team chooses ECDSA implemented in `libsodium`, then the client teams will need to package the library into Lookout's applications. Table 3.4 lists the additional file size after including the compiled `libsodium` binaries on iOS and Android.

Table 3.4: Additional File Size for Including `libsodium` (KB)

Android	360
iOS	884

4. Conclusion

The present and future security of the signature scheme is the most important criteria to consider because the primary goal of signing Micropush's messages is to ensure their authenticity. The ease of implementation and maintainability is also very important because security related code must be well maintained and understood by all of Lookout's developers. The signing speed is somewhat important because although the number of messages that Micropush can sign scale linearly with hardware, the team still wants to minimize the future server costs as the number of users grows exponentially.

It is not necessary to consider the verification speed because all of the times are too low to be of concern. In addition, users' devices should only receive a few messages per day so the computation time and battery drain will be barely noticeable.

The additional file size for including `libsodium` is not very important because although it could mean an increase of up to 10% in application size, it is more important to focus on security and costs than to let this criteria hinder the team from making the optimal choice.

In the end, Micropush was set up to sign its messages with RSA primarily because of its ease of implementation. Due to the difficulty of implementing `libsodium` and maintaining it on Android, the Android team is strongly opposed to using ECDSA. In terms of security, RSA has been studied by cryptographers much longer than ECDSA has so it should be considered weaker despite having the same theoretical security level. Nonetheless, no cryptography scheme is bulletproof as both will continue to get weaker over time. Since it is possible for either schemes to succumb to breakthroughs overnight, it is still better to use RSA because its key size and security can easily be increased whereas `libsodium`'s key is fixed at 256 bits. Finally, despite the significant future server costs in both bandwidth and computation time, current server costs are still negligible compared to developer time - especially at Lookout's current stage as a company.

5. Recommendations

While there are sufficient server resources to sign messages with RSA today, the magnitude of the performance difference as shown in Table 3.3 may be problematic in the future. Although RSA is hypothesised to be faster to verify, Table 3.2 proves otherwise for the iOS platform. And with the optimized version of `libsodium` for embedded systems coming soon [10], it is possible that ECDSA will be faster than RSA for both signing messages and verifying signatures. This is especially important because with new embedded hardware entering the mainstream market, Lookout may have to support more platforms due to partnerships or projected growth. In the future, it is recommended to rerun the benchmarks and re-evaluate the available cryptography libraries based on the above criterion with market-dominant devices.

In addition, it is recommended for any key-based cryptography algorithm to generate new keys at least once every two years. Although it is unlikely for an extremely powerful and well-funded organization such as foreign governments to target Lookout, the team still want to be able to avoid the worst case scenario where attackers are able to forge a valid signature faster than what Table 2.1 predicts.

References

[1] Apple Inc., “Apple Push Notification Service,” https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html, Feb 2014, (Accessed: 2014-07-30).

[2] Google Inc., “GCM HTTP Connection Server,” http://developer.android.com/google/gcm/http.html, (Accessed: 2014-07-30).

[3] Electronic Frontier Foundation, “NSA Spying,” https://www.eff.org/nsa-spying, (Accessed: 2014-08-11).

[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, 5th ed. CRC Press, 2001.

[5] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. New York: Wiley, 1996.

[6] Apple Inc., “iOS Security,” https://www.apple.com/iphone/business/docs/iOS_Security_Feb14.pdf, Feb 2014, (Accessed: 2014-09-04).

[7] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Y. Yang, “High-speed high-security signatures,” http://ed25519.cr.yp.to/ed25519-20110926.pdf, (Accessed: 2014-08-13).

[8] F. C. Adams and G. Laughlin, “A Dying Universe: The Long Term Fate and Evolution of Astrophysical Objects,” *Reviews of Modern Physics*, 1997.

[9] E. B. Barker, W. C. Barker, W. E. Burr, and W. T. Polk, “Recommendation for Key Management - Part 1: General (Revision 3),” National Institute of Standards and Technology, NIST Special Publication 800-57, July 2012.

[10] Frank Denis, “Comment on ’More work...’ 5f53252,” https://github.com/Trinovantes/Work-Term-Report-3B/commit/5f53252749d9bcf44a81e03d3ec64fc4af0ca116#commitcomment-7384548, Accessed: 2014-09-04.

Acknowledgements

Firstly, I would like to thank my mentor Daren Desjardins for assigning me the original task of investigating the feasibility of implementing `libsodium` on iOS; secondly, I would like to thank Alex Shoykhet from the Android team for advising me on their investigations; and thirdly, I would like to thank Marc Chung for helping me understand Micropush's architecture and interactions with the rest of Lookout's back-end. I would also like to thank the maintainer of `libsodium` Frank Denis and my roommate Michael Chang for providing early feedback on this report. Finally, I would like to thank my company Lookout for providing me with the necessary equipment for writing this report and performing the benchmarks.

A. Additional Details

The following sections provide an overview of the mathematics behind RSA and ECDSA signature schemes. These sections ignore implementation details associated with the algorithms such as message padding or truncations in order to provide a simple overview. Note that other references may use different notation to represent the same values.

A.1 RSA

A.1.1 Key Generation

1. Generate two unique prime numbers p and q that are roughly the same size, i.e. they should have approximately the same number of bits.
2. Compute:

$$n = pq \tag{A.1}$$

$$\phi = (p-1)(q-1) \tag{A.2}$$

3. Select a random integer e such that $1 < e < \phi$ and $\gcd(e, \phi) = 1$. In practice, e is choosen to have a small bit length such as $3 = 0\text{b}11$ or $65537 = 0\text{x}10001$.
4. Compute using the extended Euclidean algorithm:

$$d \equiv e^{-1} \pmod{\phi} \tag{A.3}$$

5. The public key is the pair (n, e) and the private key is d.

A.1.2 Signing Messages

1. Given the a message m to sign, generate its hash M with a hashing function H.

$$M = H(m) \tag{A.4}$$

2. Compute:

$$s = M^d \pmod{n} \tag{A.5}$$

   In practice, the value of d is very large so this step will take a while; thus explaining why it is slow to generate RSA signatures.

3. The signature for m is s.

A.1.3 Verifying Messages

1. Given a message m and its signature s, first generate the message's hash M using the same hashing function H used to sign it.

$$M = H(m) \tag{A.6}$$

2. Compute:

$$M' = s^e \mod n \tag{A.7}$$

   Since the value of e was chosen to be small with minimal number of bits equal to one, RSA signatures are fast to verify.

3. The signature is valid if and only if $M = M'$. The proof for correctness can be found in the original RSA paper by R. L. Rivest, A. Shamir, and L. Adleman[1].

A.2 ECDSA

A.2.1 Key Generation

Without going too deep into the field of elliptic curve cryptography, it is sufficient to know that the public parameters include an ”elliptic curve field” F which can be thought of as a set of q points $\{G, 2G, 3G, ..., (q-1)G\} \bigcup \{\infty\}$ where $G = (x_G, y_G)$ is known as the ”base point” and ∞ is a specially defined ”point at infinity”. Moreover, every point's x y coordinates except the ∞ point are modulo over a prime number p. Note that point addition and multiplication such

[1] http://people.csail.mit.edu/rivest/pubs/RSA78.pdf

as $2G = G + G$ is not as simple as adding the x and y coordinates together. Instead it is a complex formula that is simple going forward but difficult to reverse for *secure* elliptic curves; thus providing the security for ECDSA (this is known as the ECDLP).

In summary, the public parameters for ECDSA are p, F, G, and q. The private key is a randomly selected integer $a \in [1, q-1]$ and the public key is $Y = aG$ in the field F.

A.2.2 Signing Messages

1. Given the a message m to sign, generate its hash M with a hashing function H.

$$M = H(m) \tag{A.8}$$

2. Choose an unique and random per-message secret $k \in [1, q-1]$. It is important that k is only used once and cannot be easily predicted otherwise it will be trivial to obtain the private key.

3. Compute:

$$(x_1, y_1) = kG \tag{A.9}$$

4. Compute:

$$r = x_1 \mod q \tag{A.10}$$

$$s = k^{-1}(M + ar) \mod q \tag{A.11}$$

   If $r = 0$, i.e. when $kG = \infty$, then choose a different k and try again. Notice that a lot of the values such as k, $k^{-1} \mod q$, x_1, r, and ar can be precomputed before the message m is known; thus explaining why it is fast to generate ECDSA signatures.

5. The signature for m is the pair (r, s)

A.2.3 Verifying Messages

1. Given a message m and its signature (r, s), first generate the message's hash M using the same hashing function H used to sign it.

$$M = H(m) \tag{A.12}$$

2. Compute:

$$u_1 = s^{-1}M \mod q \tag{A.13}$$
$$u_2 = s^{-1}r \mod q \tag{A.14}$$

3. Compute:

$$(x_2, y_2) = (u_1 G + u_2 Y) \mod q \tag{A.15}$$

4. The signature is valid if and only if $x_2 = r$. The proof for correctness is shown below:

$$\begin{aligned}
s &\equiv k^{-1}(M + ar) \text{ (mod } q) && \text{From (A.11)} && \text{(A.16)}\\
k &\equiv s^{-1}(M + ar) \text{ (mod } q) && && \text{(A.17)}\\
kG &= (s^{-1}(M + ar) \mod q)G && && \text{(A.18)}\\
(x_1, y_1) &= (s^{-1}M \mod q)G + (s^{-1}ar \mod q)G && && \text{(A.19)}\\
&= (s^{-1}M \mod q)G + (s^{-1}r \mod q)Y && && \text{(A.20)}\\
&= u_1 G + u_2 Y && && \text{(A.21)}\\
&= (x_2, y_2) && && \text{(A.22)}
\end{aligned}$$

B. Calculations

B.1 Comparing Relative Security Levels

According to the June 2014 TOP500 list of supercomputers[1], the fastest supercomputer in the world (Tianhe-2) can perform 33.86 Pflop/s. For the sake of simplicity, the following calculations assume that one flop (floating point operation) is equivalent to performing one operation to break the cryptography scheme. Since each operation usually require more than one flop, these calculations will be overestimating the supercomputer's capabilities.

$$33.86 \text{ quadrillion guesses per second} \approx 2^{56} \text{ guesses per second} \quad \text{(B.1)}$$

$$\approx 2^{80} \text{ guesses per year} \quad \text{(B.2)}$$

Assuming that a supercomputer that can perform 33.86 quadrillion operations actually exists, then the time to break 64-bit, 80-bit, 128-bit, and 256-bit security levels can be summarized in Table B.1.

Table B.1: Time to Break Scheme

Security Level	Operations	Time
64	2^{64}	Less than a year
80	2^{80}	Less than a decade
128	2^{128}	$2^{48} \approx 2.8 \times 10^{14}$ years
256	2^{256}	$2^{172} \approx 6.0 \times 10^{51}$ years

[1] http://www.top500.org/lists/2014/06/

B.2 Signatures per Second

To determine the how many signatures can be generated per second, it is much easier to first calculate how many seconds it takes to sign a fixed number of messages and then inverse that result - assuming that signing messages scales linearly. The results of benchmarking how long it takes to sign n messages are show Table B.2.

Table B.2: Time to Sign Messages (s)

Number of Signatures	RSA (3072-bit key)	ECDSA (256-bit key)
1000	4.7938	0.0726
2000	9.4438	0.1423
3000	14.3038	0.2105
4000	19.0899	0.2832
5000	23.7339	0.3585

From the table above, it is clear that time scales linearly with the number of messages to sign. Therefore, it is a valid assumption that the number of messages signed per second is the inverse of the above table multiplied by the number of messages signed:

Table B.3: Messages Signed Per Second (rounded down)

Time (s)	RSA (3072-bit key)	ECDSA (256-bit key)
1	208	13774
2	211	14054
3	209	14251
4	209	14124
5	210	13947

C. Benchmark Details

For each benchmark's working source code, see `https://github.com/Trinovantes/Work-Term-Report-3B`.

C.1 Signature Verification Benchmarks

For the RSA benchmarks, the team used the test devices' native implementations: Nexus 4 running on `Android 4.4.4` and iPhone 5c running on `iOS 7.1.2`. For the ECDSA benchmarks, the team used `libsodium 0.6.1` compiled with `-Os` and `--enable-minimal` flags optimizing for file size instead of speed (See Section 3.6 for explanation).

The benchmarks first run the verification steps 80 times to make sure any runtime optimizations are completed and system caches are filled with the relevant data. They then run the verification steps another 20 times and averages the results.

C.2 Signature Generation Benchmarks

As mentioned in Section 3.4, signing messages on the server scale linearly with hardware so it is sufficient to run this benchmark on any environment and scale accordingly to match Lookout's production servers. Therefore, the team ran this benchmark on a mid-range laptop with a `2.3 GHz Intel Core i5 2410M` processor. For the RSA benchmarks, the team used `OpenSSL 1.0.1f` because Linux does not have native RSA support. For the ECDSA benchmarks, the team used `libsodium 0.6.1` compiled with `-O3` flag optimizing for speed instead of file size.

Similar to the benchmark in Section 3.3, this benchmark also runs the verification step 80 times before averaging the next 20 runs to ensure runtime optimizations have completed and caches are filled.

C.3 File Size of Library

To minimize the additional file size of the iOS and Android applications when packaged with `libsodium`, the team compiled with the `--enable-minimal` flag. When submitting to the Google Play Store or to the Apple App Store, the uploaded application must support every architecture of those platforms. Table C.1 and Table C.2 lists the file size of the compiled binaries

for each platform's supported architectures.

Table C.1: Size of Compiled `libsodium` Library for Android Architectures (KB)

armeabi	360
Total	360

Table C.2: Size of Compiled `libsodium` Library for iOS Architectures (KB)

arm64	260
armv7	312
armv7s	312
Total	884