

Final

Instructor: Matthew Green

Due: 11:59pm on 12/15

Name: _____

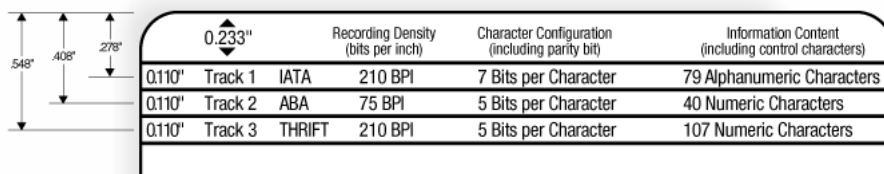
This exam is to be turned in on Blackboard. As with any exam, please do not collaborate or otherwise share information with any other person. You *are* permitted to use books, notes and online resources.

Problem 1: Short Answer (21 points)

1. **Shor's Algorithm.** Explain what Shor's algorithm is and what it is used for. What implications might the existence of this algorithm have for our decisions about which cryptosystems to use right now.
2. **Subliminal Channels.** In Chapter 11 of Security Engineering,¹ Ross Anderson describes the concept of "subliminal channels". Explain this concept and give an example of a signature scheme besides DSA that could be (mis)used to provide such a channel.
3. **Broadcast encryption.** Explain what was wrong with DVD CSS and how AACS broadcast encryption tried to rectify this problem.

Problem 2: Secure J-cards (20 points)

Alishah and David discussed the inadequate security of the current J-card system. Your goal is to improve the security of the magnetic swipe J-card system (let's ignore the contactless version for the moment.) Recall the track design shown in class:



The diagram shows a magnetic card with three tracks. Dimensions are indicated on the left: 548" for the total height, 408" for the track area, and 278" for the track width. A recording density of 0.233" is shown above the tracks.

			Recording Density (bits per inch)	Character Configuration (including parity bit)	Information Content (including control characters)
0.110"	Track 1	IATA	210 BPI	7 Bits per Character	79 Alphanumeric Characters
0.110"	Track 2	ABA	75 BPI	5 Bits per Character	40 Numeric Characters
0.110"	Track 3	THRIFT	210 BPI	5 Bits per Character	107 Numeric Characters

Figure 1: Magnetic card tracks.

1. A problem with the design presented in class is that current J-card numbers have entirely too little entropy (that is, they are too short and predictable). If we used every single symbol on all three tracks above (and ignored specific formatting conventions within the tracks), how many *total bits* of data could we use for a J-card number?

¹Available on the Course Syllabus page.

2. Assume we want to perform J-card validation without having to connect card readers to a live back-end database. How could we do this securely using cryptography? How do we handle things like stolen and replacement cards without a live database?
3. Assume you want to achieve a 128-bit security level for your system, *i.e.*, the total number of operations required to break a cryptographic primitive is about 2^{128} . What cryptographic primitives would you use, considering the data storage limits described above? How would you do key distribution, and what are the tradeoffs?

Problem 3: Source code review and reverse-engineering (50 points)

In August 2016 the Turkish intelligence services (MIT) announced that they had decrypted messages sent by a group of individuals who were alleged to have plotted the July 2016 attempted military coup. According to these reports, the individuals used an encrypted messaging application called “ByLock” to communicate securely.² While many aspects of this reporting are unverifiable, we do have the ability to examine the application to determine whether this story is plausible. Your goal in this assignment is to reverse-engineer and examine the Java code of the ByLock Android client application (APK), in order to derive an understanding of the operation of the system and its resilience to compromise.

To get started, download the files located at:

<https://github.com/matthewdgreen/practicalcrypto/tree/master/FinalExam>

Important safety warnings: The Bylock APK is an Android application that was downloaded from an untrusted software archiving website. You should assume that it could be infected with malware. *Thus: use caution if you intend to install this application on an Android device and/or manipulate the APK extensively on a trusted system.* To simplify your work, I have provided a ZIP file ([source.zip](#)) containing the decompiled source code from the APK, which I obtained using an online APK decompiler. It is not necessary to run the application in order to complete this assignment, and indeed I have not attempted to do so. If you choose to try this it is your responsibility to use careful isolation procedures. *Proceed at your own risk.*

If you plan to travel to Turkey in the near future, I recommend that you not carry this code on your devices.

Technical Background: Bylock is a client/server application that supports email, “chat” and phone calling. For mail and chat messages, the operation is similar to JMessage. The ByLock server appears to have been taken down and the code is no longer available. Users are identified by a username and a password. There is some circumstantial evidence that the ByLock server may have been under attack,³ so for this analysis you should assume that the server could have been at least temporarily compromised at some point.

Suggestions on your review. The application I retrieved has been stripped of most function names, which makes it difficult to read. To review the source code you can use any

²See e.g., <http://www.reuters.com/article/us-turkey-security-app-idUSKCN10E1UP> and many similar reports.

³See e.g., <https://github.com/matthewdgreen/practicalcrypto/blob/master/FinalExam/Wordpress.pdf>

text editor and search utility (*e.g.*, `grep`), although I *highly* recommend the use of an IDE like Eclipse or Android Studio.⁴ A rough summary of some key code regions is described below:

1. `net.client.by.lock.f` contains network communications routines.
2. `net.client.by.lock.e` contains document formatting and parsing routines for messages sent to and from the server.
3. `net.client.by.lock.d` contains user state-keeping and cryptographic routines.
4. An important class you should know about is in the file `r.java` in the directory `net.client.by.lock.d`. This class appears to store information such as cryptographic keys and usernames for known users. It also stores information for the account that is currently logged into the app, such as secret keys and other cryptographic state. The *static* method `r.i()` returns an instance of the `r` class corresponding to the currently logged-in user. Thus, if you see a reference to `r.i().d()` that would correspond to a call to method `d()` on the class `r` corresponding to the logged-in user. Similarly, a call to `r.i().l()` would be a call to the method `l()` in the same instance. You will see this convention used throughout the entire codebase.

Questions. Using the provided source code in `source.zip`, answer the following questions. For each answer, please provide source code filenames, function names *and/or line numbers*⁵ identifying specific pieces of code that support your interpretation.

1. Can you find the IP address of the (now defunct) ByLock server? Look in the sub-directory containing the network code. Describe how ByLock sent commands to the server, *e.g.*, what communication mechanism it used.
2. Did ByLock use TLS to communicate with the server? If so, how does the client validate the server's certificate?⁶
3. In the directory `net.client.by.lock.f`, what do you think the classes in `k.java` and `l.java` are doing?
4. ByLock uses public key encryption to send messages to users. Using some basic knowledge of how Java crypto APIs work, try to find the location where the client generates the RSA public and private keys it uses for encryption.
5. Now from this point, try to identify how the RSA private key is stored when the user is not running the app. Hint: as a beginning, start from the function you identified in question 4 above. Now look at the call to a method named `c()` in the same class, and determine what it does (and what inputs it uses), then determine what happens to the resulting output. Be strong.⁷

⁴In particular, if you import the source tree into Eclipse, you will be able to use the "Open Declaration" function in order to rapidly find declarations in the code.

⁵It's ok if you decompile your own code, but the questions below are based on my version of the decompiled code. If you analyze your own decompiled code, please specify function names and code excerpts in the event that they are different from the provided code.

⁶Hint: this is in the same directory as the previous question.

⁷Hint: see my note above about what the prefix `r.i()` means.

6. How does ByLock encrypt, decrypt and verify chat messages? Note: give me the cryptographic primitives used and the order. Hint: to find the right file, grep for the error string "Error while verifying chat."
7. As mentioned above, the class `r.java` in the directory `net.client.by.lock.d` stores account-specific state, including secret state for the logged-in user. In this class you will find a member variable of type `SecureRandom`. How is this random number generator class seeded? What is it used for? ⁸
8. Given everything you learned above, imagine that you hacked into the ByLock server and made a copy of its database. Would you be able to decrypt past encrypted traffic that was stored in this database? How? Feel free to give multiple theories if you want.

Problem 4: CBC-MAC and Proofs (30 points)

A common approach to building a Message Authentication Code (MAC) is to use a block cipher in CBC mode. This MAC, known as CBC-MAC, is computed by running CBC mode encryption on a message using a fixed IV (typically 0) and outputting the final block of ciphertext as the MAC. Let $M = m1 || m2 || \dots || m_x$, where each m_i is the size of the cipher's block length. The following diagram illustrates the computation of CBC-MAC:

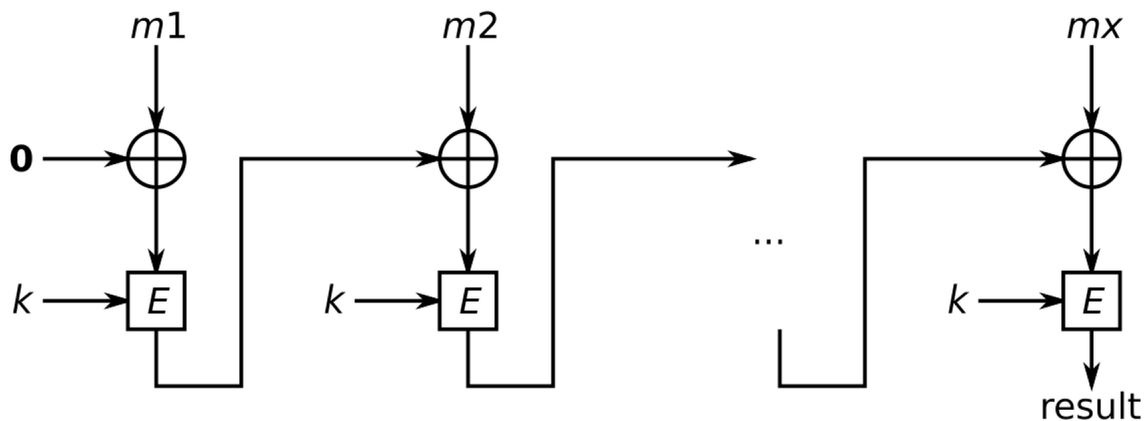


Figure 2: Description of CBC-MAC, courtesy Wikipedia.

A secure MAC should satisfy the definition of *Existential Unforgeability under Chosen-Message Attack* (EU-CMA). Briefly, this means that no Adversary should win the following game played against an honest Challenger except with negligible probability:

1. The Challenger picks a random λ -bit key k .
2. As many times as the Adversary likes, s/he can send a "query" M_i to the Challenger. The Challenger computes $T_i = \text{CBCMAC}(k, M_i)$ and returns T_i to the Adversary.

⁸Hint: the acronym PRNG means "pseudo random number generator".

3. The Adversary wins if s/he can output any pair (M^*, T^*) such that $T^* = \text{CBCMAC}(k, M^*)$ and for all i , $(M^*, T^*) \neq (M_i, T_i)$ (*i.e.*, the adversary has not previously issued a query on M^* and received T^* in response).

Consider the following questions:

1. For this question, *assume that all messages are of a pre-determined fixed length* (exactly N blocks long, for some arbitrary N).

Let E be an ideal cipher with a λ -bit key and ℓ -bit input/output. Using the ideal cipher model heuristic, give an informal argument as to why CBC-MAC might be a secure MAC under the EU-CMA definition above. Note that this does not need to be a proof, but it should be complete enough to convince me that CBC-MAC satisfies the informal definition above.

(Hint: The structure of your argument should be as follows. First, consider an Adversary playing the game with a Challenger. Assume that both parties are calling out to an ideal cipher oracle [the “gnome”] each time they call the E function on some key and plaintext. Explain how the Challenger will produce each MAC requested by the Adversary. For the pair (M^, T^*) output in step 3, explain the properties of the tag T^* and convince me of the Adversary’s probability of finding such a tag.)*

2. CBC-MAC is not secure when the messages are of *variable-length*. To forge a message, the adversary can query on two messages M, M' to obtain MAC tags T, T' respectively. Let us represent M' as $\{m'_1, \dots, m'_N\}$. The Adversary can now compute a third message M'' as follows:

$$M'' = M || (m'_1 \oplus T) || m'_2 || \dots || m'_N$$

What is the correct MAC on the message M'' . How does the Adversary find it?

3. Bonus (5 points): Some CBC-MAC variants prepend the *length* of the message M to the message before computing the MAC. Explain how this foils the attack described above.