Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 17
October 3, 2002

Problem Set 2 Solutions

Problem 1-1. One-Time Pads

We received a lot of nice and different solutions to this problem. We had a hard time picking the nicest.

The two main approaches to breaking the messages of one group of messages sharing the same One-Time Pad were:

(a) Guessing a totally blank message and trying to recognize English patterns in the other messages of the same group, and reiterate for the other messages to get even more chunks of the plaintexts. This method provides a fast start, but does not scale that well.

(b) Making smart guesses. While this method makes the decoding of the first message more tedious, it scales better.

There were as many variants as languages/tools used to solve this problem: Perl, C, C++, C#, Java, Maple, Emacs and Excel.

We provide here an excellent solution to this problem combining both approaches, and their respective advantages: fast start from the all-blank messages trials, and scalability from the educated guesses. Also, the coding is really neat.

This solution has been brought to you by: Jim Paris, David Mellis, Chris Elledge, and Hernan Lombardo.

It is easy to figure out when two messages use the same OTP. Note that, due to the strange message format, for any two messages x and y,

$$0 \leq x_i \oplus y_i < 16$$

for all individual bytes $i \in [0, 191]$. Therefore, by taking two encoded messages $x' = x \oplus p$ and $y' = y \oplus q$ and XORing them together, we get $x' \oplus y' = (x \oplus y) \oplus (p \oplus q)$, which in turn equals $x \oplus y$ if they used the same pad $p = q$. So, if all bits of $x' \oplus y'$ are less than 16, we can be reasonably certain that the same OTP was used; otherwise, we can be fully certain that different OTPs were used.

At this point, we wrote a program to figure this out, as well as decode all of the messages. It was written iteratively, repeatedly looking at the output, deciding what other useful stuff it could do to figure out the contents, and modifying it accordingly. The basic idea is to XOR together two messages with the same OTP, then XOR that again with a guess at one of the messages – if the guess is correct, then the result is another valid message. The iterative process we went through was something like this:

- Figure out which messages have the same OTP
- For each pair, guess that one of the messages is all spaces and XOR that
- Since many messages *do* contain many spaces, this gave us enough information to just look at the results and figure out the message format (which is now placed in the "goal" variable)

- Let the program use "goal" as guesses and iteratively refine the guesses until we decode as much as possible
- Add the "smartfillin" function to fix up "obvious" parts of the data (completing the word "TOMORROW", for example).
- Run the program again. Now it decodes everything.

The source code is available at (it has neat comments about what the various functions do, and more details about the steps):

`http://web.mit.edu/6.857/www/handouts/H17/solve.c`

(a) It turns out that only 7 pads are used for the 24 messages:

```
messages with pad 1: 02 03 09 16
messages with pad 2: 06 07 11
messages with pad 3: 04 10 12
messages with pad 4: 05 13
messages with pad 5: 01 14 19 21
messages with pad 6: 08 17 18 22 23
messages with pad 7: 15 20 24
```

Running the code, we get:

```
Now decoding messages with pad 1: 02 03 09 16
TO: RACHEL .FROM: PAUL   .WHEN: TODAY   .TIME: 11:00PM.WHERE: VIENNA     .DATE: WED 03 JUL 2002.
TO: DAVID  .FROM: RACHEL .WHEN: TODAY   .TIME: 12:00PM.WHERE: BUDAPEST   .DATE: THU 08 AUG 2002.
TO: BOB    .FROM: CHARLIE.WHEN:         .TIME:        .WHERE:            .DATE: THU 05 SEP 2002.
TO: ALICE  .FROM: BOB    .WHEN:         .TIME:        .WHERE:            .DATE: SUN 22 SEP 2002.
Now decoding messages with pad 2: 06 07 11
TO: PAUL   .FROM: MARY   .WHEN:         .TIME:        .WHERE:            .DATE: SAT 14 SEP 2002.
TO: ALICE  .FROM: BOB    .WHEN: TOMORROW.TIME: 04:15PM.WHERE: PARIS      .DATE: WED 25 SEP 2002.
TO: JOHN   .FROM: KELLY  .WHEN:         .TIME:        .WHERE:            .DATE: THU 15 AUG 2002.
Now decoding messages with pad 3: 04 10 12
TO: PAUL   .FROM: ALICE  .WHEN: TOMORROW.TIME: 09:20AM.WHERE: ATHENS     .DATE: FRI 31 MAY 2002.
TO: CHARLIE.FROM: KELLY  .WHEN:         .TIME:        .WHERE:            .DATE: SUN 01 SEP 2002.
TO: BOB    .FROM: CHARLIE.WHEN: TODAY   .TIME: 04:20PM.WHERE: ROME       .DATE: FRI 13 SEP 2002.
Now decoding messages with pad 4: 05 13
TO: DAVID  .FROM: SUSAN  .WHEN:         .TIME:        .WHERE:            .DATE: TUE 10 SEP 2002.
TO: KELLY  .FROM: SUSAN  .WHEN: TOMORROW.TIME: 10:30AM.WHERE: WARSAW     .DATE: SUN 01 SEP 2002.
Now decoding messages with pad 5: 01 14 19 21
TO: DAVID  .FROM: RACHEL .WHEN:         .TIME:        .WHERE:            .DATE: WED 07 AUG 2002.
TO: CHARLIE.FROM: KELLY  .WHEN: TOMORROW.TIME: 03:50PM.WHERE: STOCKHOLM  .DATE: SAT 07 SEP 2002.
TO: KELLY  .FROM: SUSAN  .WHEN:         .TIME:        .WHERE:            .DATE: FRI 30 AUG 2002.
TO: MARY   .FROM: ALICE  .WHEN:         .TIME:        .WHERE:            .DATE: MON 02 SEP 2002.
Now decoding messages with pad 6: 08 17 18 22 23
TO: JOHN   .FROM: KELLY  .WHEN: TODAY   .TIME: 05:25PM.WHERE: MADRID     .DATE: FRI 30 AUG 2002.
TO: PAUL   .FROM: ALICE  .WHEN:         .TIME:        .WHERE:            .DATE: MON 27 MAY 2002.
TO: MARY   .FROM: ALICE  .WHEN: TOMORROW.TIME: 08:30AM.WHERE: COPENHAGEN .DATE: TUE 10 SEP 2002.
TO: SUSAN  .FROM: JOHN   .WHEN:         .TIME:        .WHERE:            .DATE: FRI 16 AUG 2002.
TO: PAUL   .FROM: MARY   .WHEN: TODAY   .TIME: 06:10PM.WHERE: HELSINKI   .DATE: THU 19 SEP 2002.
Now decoding messages with pad 7: 15 20 24
TO: RACHEL .FROM: PAUL   .WHEN:         .TIME:        .WHERE:            .DATE: SAT 30 JUN 2002.
TO: DAVID  .FROM: SUSAN  .WHEN: TOMORROW.TIME: 01:45PM.WHERE: LONDON     .DATE: MON 12 SEP 2002.
TO: SUSAN  .FROM: JOHN   .WHEN: TOMORROW.TIME: 02:55PM.WHERE: PRAGUE     .DATE: MON 19 AUG 2002.
```

(b) From the decoded messages, we see that the only time "Paul" meets "Mary" is on September 19 in "Helsinki" at 06:10PM. We've missed the meeting, so we can't intercept them.

(c) From the decoded messages, we see that "Alice" and "Bob" plan on meeting on September 26 at 04:15PM in "Paris". Our Secret Agent Man^{TM} is on his way!

(Jim Paris wrote most of the code and this writeup. David Mellis had most of the ideas about pattern matching and filling in what we know, and also figured out the message format. Chris Elledge and Hernan Lombardo gave additional ideas and support during the whole process. Secret Agent Man^{TM} just watched.)

For your reference, we used the following code to design and try to break the messages:

`http://web.mit.edu/6.857/www/handouts/H17/otp.jar`

Using the second approach exclusively, starting by breaking the group of 5 messages (as suggested as a hint by the problem) takes about 5 to 10 minutes by first guessing "Copenhagen", while breaking the other groups should take less than a couple of minutes per group if you first try to guess the chunk with all the blank fields:
"`.when:␣␣␣␣␣␣␣␣␣.time:␣␣␣␣␣␣␣␣.where:␣␣␣␣␣␣␣␣␣␣␣␣.date:␣`".

Problem 1-2. Hashes do grow in trees, you know

Here is one of the excellent solutions, written by Kenny C. K. Fong, Jonathan M. Hunt, Soyini D. Liburd, and Eduardo I. McLean. We have annotated their solution a little bit.

(a) Proving a leaf

Several solutions said that direct ancestors (the path to the root itself) must appear in the signature. This is not necessary because the signature verifier must recompute the path anyway. The verifier cannot trust that the path is what the signer claims without recomputing it. This led to signature sizes off by a factor of 2.

We accepted answers of 560 or 580 bytes, depending if you considered the leaf itself as part of the signature. Usually we do not consider the value being authenticated as part of the signature though. A few teams made off-by-one errors. As defined by the problem set, a tree with depth t has 2^t leaves. Therefore the path length is t, not $t-1$. Consider the case of $t = 2$ and draw the tree and signature...

The signer should publish the sibling of x_s and the siblings of all ancestors of x_s in order to reveal x_s and prove it is part of the tree rooted at X (the *ancestors* of x_s are the tree nodes along the path between x_s and the root X). Since the depth of the Merkle tree is t, there are t tree nodes (i.e. the siblings) in total in the proof, and so t hash function evaluations are required on the part of the verifier to check the proof. The sibling of x_s is itself a leaf and so its length is 160 bits; furthermore, the siblings of the ancestors of x_s are hash values of h and so are 160 bits long. Hence, the length of every tree node in the proof is 160 bits, whereas the length of the proof is $160t$ bits, i.e., $20t$ bytes.

(b) Why do we force w_i to end with a zero bit?

The best way to answer this question is to ask yourself what happens if you instead define $w_i = i||v_i$. In this case, the signatures are malleable as shown below. In fact, we could make the last bit random or even make the parent of x_{w_i} the hash of a single leaf instead of two leaves. That is, we don't really need an $x_{i||v_i||1}$ node.

If we do not force w_i to end with a zero bit, an eavesdropping adversary can make an existential forgery. To illustrate this attack, assume that w_i does not end with the zero bit, i.e., $w_i = i||v_i$. Suppose, without loss of generality, that the signer Alice publishes a proof of $v_i = 1$ as the i-th

value of the hash digest; the proof consists of i, $v_i = 1$, and the proof that x_{w_i} is in the tree rooted at X (note that $w_i = i||1$ because $v_i = 1$). From part (a), we know that the proof that x_{w_i} is in the Merkle tree consists of the sibling of x_{w_i} and the siblings of all ancestors of x_{w_i}. Now, suppose that the adversary Eve eavesdrops this proof of $v_i = 1$. Then Eve can forge a proof of $v_i = 0$ as the i-th value of the hash digest as follows. The proof of $v_i = 0$ consists of i, $v_i = 0$, and the proof that $x_{w'_i}$ is in the Merkle tree rooted at X, where $w'_i = i||0$. The proof that $x_{w'_i}$ is in the Merkle tree consists of x_{w_i} (which is the sibling of $x_{w'_i}$) and the siblings of all ancestors of x_{w_i} (which are also the siblings of all ancestors of $x_{w'_i}$). The implication of Eve's ability to forge a proof of $v_i = 0$ is that she can forge a message whose hash digest is the same as Alice's except the i-th bit.

(c) How long is the signature? How many hash function evaluations are required to check the signature? (Give upper and lower bounds.)

There were several correct answers for this question, but all had the same form: you had to realize that signatures of various bits may overlap in the Merkle hash tree. Some students only included the bits for the proofs of leaves – neglecting the space needed to represent i and v_i. However, it's not necessary to include i and v_i if both parties have an established protocol such that the placement in the tree is implied. So we accepted both answers.

One question we did not ask is how long it takes a signer to generate a signature. Or how much space it takes to store the tree. In fact, the signer will have to run approximately $2^{t+1} - 1$ hash operations because the root node cannot be computed without all the children nodes known!

The signature consists of a proof of v_i as the i-th value of the hash digest of the message, for all $i \in \{0, \ldots, 159\}$. The proof of each v_i consists of i (26 bits), v_i (1 bit), and the proof that x_{w_i} is in the tree rooted at X ($160t$ bits from part (a)). Therefore, the proof of each v_i is $(160t + 27)$ bits long. There are 160 such bits in the hash digest to be authenticated, so the length of the signature is $160(160t + 27)$ bits, i.e., $20(160t + 27)$ bytes.

A lower bound for the number of hash function evaluations required to check the signature is yielded in the case that the 160 leaves to be authenticated are as close to each other as possible in the Merkle tree so as to maximize the degree of overlapping among those immediate hash values obtained during verification. The definition of w_i forces that for any full subtree of depth 2 (with 4 leaves) rooted at a node at level $t - 2$ of the Merkle tree, one can find at most one leaf to be authenticated by the signature. Hence, a scenario in which the 160 leaves to be authenticated are as close to each other as possible is that they are located within the leftmost $4 \times 160 = 640$ leaves in the Merkle tree (see Figure 1). We can now divide the Merkle tree into three areas. Within the area between levels $t - 2$ and t, $160 \times 2 = 320$ hash function evaluations are required. Within the area between levels $t - 10$ and $t - 2$ (note that $\lceil \lg 160 \rceil = 8$ and $t - 10 = (t - 2) - 8$), $80 + 40 + 20 + 10 + 5 + 3 + 2 + 1 = 161$ hash function evaluations are required, because all the 160 grandparents (at level $t-2$) of the 160 leaves to be authenticated are consecutive. Within the area above level $t - 10$, only $t - 10$ hash function evaluations are required, because all branches leading to the leftmost 640 leaves merge to a single node at level $t - 10$ and so now we just need to head towards the root by following a single path. To conclude, a lower bound for the number of hash function evaluations required to check the signature is given by $320 + 161 + (t - 10) = 471 + t$.

On the contrary, an upper bound for the number of hash function evaluations required to check the signature is yielded in the case that the 160 leaves to be authenticated is as far away from each other as possible in the Merkle tree. In other words, the 160 leaves should be evenly spread out at the

bottom level (see Figure 2). In this case, we count the number of hash function evaluations required using a top-down approach rather than a bottom-up approach. We divide the Merkle tree into 2 areas. Within the area above level 8 ($= \lceil \lg 160 \rceil$), immediate hash values for verification of different leaves overlap , and so $128+64+32+16+8+4+2+1 = 255$ hash function evaluations are required. Within the area below level 8, signature verification for each of the 160 leaves follows a single path between the leaf and one unique node at level 8, and these 160 paths do not intersect each other (because these paths have to spread out evenly in the huge Merkle tree so that their ends, i.e. the 160 leaves to be authenticated, are also evenly spread out at the bottom level). Therefore, in this area, $160(t-8)$ hash function evaluations are required. Hence, an upper bound for the number of hash function evaluations required to check the signature is given by $255+160(t-8) = 160t-1025$.

(d) What properties should the hash function have?

The hash function should have the following properties:

1. It should be One-Way. Given one node, it should be infeasible for someone to compute the values of its children. For example, people should not be able to compute the values of the secret children from the value of the public root.
2. It should be Collision Resistant. It should be infeasible to find two different inputs that hash to the same output value. Otherwise, for example, it would be possible to sign one message, and have an adversary claim that you signed another message (that has the same proof).

(e) How do signature size and verification speed compare with 1,024-bit RSA with small public exponent size? (You can find suitable information on the Web...)

Because the hash function is 160 bits, we had hoped you would evaluate the only 160-bit hash function we discussed in lecture: SHA-1. We also accepted comparisons to MD5 performance (about twice as fast as SHA-1) and parameterized performance (e.g., hash H can process X bytes/sec). The verification performance of the RSA and Merkle methods are surprisingly close. The RSA verification method is slow, but the small public exponent makes verification must faster than the average modular exponentiation. The Merkle hash tree verification requires several hundred hash operations, but each operation is very fast.

According to slides 21–22 on `http://www.cs.wpi.edu/~rek/Adv_nets/Spring2002/Kerberos.ppt`, a modern machine achieves a SHA-1 throughput of 59.369 MB/sec and a 1,024-bit RSA verification (presumably low-exponent) of 0.23 ms/verification. Neglecting startup overheads, we can assume SHA-1 can make 1,556,322 hash operations/sec when the input is always 320 bits (each Merkle tree node is the hash of two 160-bit values). Then a single SHA-1 hash of 320 bits should take about 0.6425μsec. That means we can do 358 SHA-1 operations for each RSA signature verification. Since verification in the Merkle scheme varies between 499 and 3,455 hash operations (according to part (c)), the Merkle verification is slightly slower (0.32 ms minimum, 2.2 ms maximum). However, there may be start up overheads in SHA-1 (hashing 320 bits probably doesn't reach the asymptotic running time). So the verification is likely no faster than determined above.

From part (c), we know that the signature size of the proposed authentication scheme is $20(160t+27)$ bytes. If $t = 28$, then the signature size is about 88 Kbytes, which is too large. However, a 1,024-bit RSA signature has size 1,024 bits only [1].

RSA signature scheme with small public exponent size is characterized by its fast verification speed. If the public exponent is small, say 17, then RSA verification involves 5 modular multiplications only.

From [2], we know that 1,024-bit RSA verification (with a small public exponent) has a throughput of 7.4 Kbits/sec on a 90 MHz Pentium. However, modular exponentiation is much slower than a hash function evaluation. From [3], we can see that the throughput of MD5 is 136.7 Mbits/sec! Recall from part (c) that the upper bound for the number of hash function evaluations required for verification in the proposed authentication scheme is $160t - 1025$. If $t = 28$, the verification speed of the proposed scheme is just 3456 hash function evaluations (one more hash function evaluation is needed to hash the message with h), which is not much, taking the extreme high speed of a hash function evaluation into account. Hence, the verification speed of the proposed scheme is faster than that of 1,024-bit RSA.

Contributors
Solution Design: (entire group effort) - Kenny Fong, Jonathan Hunt, Soyini Liburd, Eduardo McLean.
Write Up: Kenny Fong, Soyini Liburd.
Proof Reading: (entire group effort) - Kenny Fong, Jonathan Hunt, Soyini Liburd, Eduardo McLean.

References
1. Robert Zuccherato, Elliptic Curve Cryptography Support in Entrust. May 9, 2000.
2. How fast is RSA? http://www.x5.net/faqs/crypto/q9.html
3. The hash function RIPEMD-160. http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html

Problem 1-3. Block cipher security

(a) Key size

Essays covered a wide array of issues. A few essays misunderstood Triple-DES (3DES) though. Triple-DES is not three times as strong as single DES. Rather, it is at most twice as strong with its 112-bit key. The encrypt-decrypt-encrypt chain consists of encryption with key 1, decryption with key 2, and encryption with key 3. This is useful for preventing meet-in-the-middle attacks while also allowing backwards compatibility. Setting key 1 = key 2 makes the chain exactly single DES.

Here is an excellent essay from Mike Hamler, Richard Hu, Flora Lee, and Neil Sanchala. It addresses the major issues of selecting symmetric key sizes.

Recently, the National Institute for Standards and Technology adopted a new encryption algorithm, AES, as the new standard encryption. The standard previously had been DES which was created by IBM in 1976.[1] However, when the 56-bit key size for DES was judged to be insufficient, a search for a new encryption scheme began. In the meantime, people utilized Triple-DES. Triple-DES uses 112-bit keys and is therefore believed to be much more secure than DES. However, NIST decided to implement and disseminate an entirely new algorithm rather than continuing the trend of re-encrypting something multiple times in order to achieve longer key lengths. This decision was wise because certain algorithms do not scale well, a new algorithm cuts off old attacks, and also because of the change of the attitude toward encryption.

Certain algorithms and technology do not scale well. It is not necessarily efficient to merely continually encrypt multiple times in order to achieve longer key lengths. For example, in order to achieve a 112-bit key length with DES, we cannot simply encrypt a plaintext message twice. Such

[1] www.nist.gov

a scheme is thwarted by a meet-in-the-middle attack. Therefore in order to double the key size, the user must actually run DES three times, encrypt-decrypt-encrypt.[2] As we wish to increase the key size more and more, we may have to run DES a disproportionate number of times. This method is extremely inefficient. Instead, if we implement a new encryption scheme, we can keep our current efficiency while attempting to maintain the level of security. In addition, this repeated encryption does not always scale well with the key length either. Continuing the DES example, in Triple-DES, certain keys cannot be used because if the decrypt key is the same as either of the encrypt keys then the end result is just DES-level encryption. This artifact cuts down the space of available keys down and does not give us true 112-bit key length protection.[3]

Furthermore, merely re-encrypting the plaintext with an established cipher does not solve the original problem. Even though established ciphers may have stood the test of time, it does not mean that they still do not contain risks. For example, the EFF built a DES cracking machine. It recovers the DES key under a variety of attack models.[2] This information about the implementation of the cracking machine implies that it uses an approach to crack DES that is slightly better than brute-force. There still exists slight flaws in DES that can be exploited to recover the key in better than brute-force time. These flaws are not solved when the key length is expanded and in fact may lull many people into a false sense of security. Only by implementing an entirely new encryption scheme can we remove these flaws. While the new encryption scheme may have its own shortcomings, it at least forces hackers to try and find them.

Lastly, the risk of AES having a fatal flaw that is not discovered within the first month of its inception is very small. There has been a major change in attitude toward encryption as was demonstrated by NIST. NIST called for a public competition of encryption schemes which was helpful as people tried to crack other people's schemes [4]. In the past, this was not the case as certain agencies such as the NSA tried to dictate the limits and tried to restrict access to encryption schemes. Especially since the inception of the Internet and distributed computing programs such as SETI, more and more people are getting involved in helping review and test encryption schemes. For example, in a recent issue of Crypto-gram, a free monthly newsletter released by Bruce Schneier, it describes a recent claim of a flaw in AES.[5] Before the Internet, this type of information would not have been able to spread so quickly and to so many people. This makes battle-testing new encryption schemes less risky than in the past.

Because of the change in attitude toward encryption and the spread of knowledge, thanks in part to the Internet, developing and testing new encryption schemes is not as risky as it once was. This revelation makes it more attractive for the NIST to establish new standards for encryption instead of trying to patch the old ones through a tactic such as repeated encryption. The new standards come with flaws that are unknown to both hacker and to client, as opposed to old standards that may not scale well and may have slight flaws that compound themselves as we try to enlarge their key space.

(b) $\mathcal{P} = \mathcal{NP}$ or $\neq \mathcal{NP}$. That is the question.

There were several great essays arguing for or against Ben. This excellent essay comes from Brent Buddensee, Ariel Segall, and David Wilson.

[2] 6.857 Problem Set #2
[3] www.tropsoft.com
[4] 6.857 Lecture 5
[5] CRYPTO-GRAM, September 15, 2002

Ben Bitdiddle's assertion that 128-bit keys are too large because the probability of one being found in a known plaintext attack is less than the probability that we will discover that $\mathcal{P} = \mathcal{NP}$ is flawed. This assertion does not address the heart of the matter: whether a 128-bit key is appropriate for good security. We claim that a 128-bit key is not too large, and that the probability of discovering that $\mathcal{P} = \mathcal{NP}$ is irrelevant.

If $\mathcal{P} = \mathcal{NP}$, several of the assumptions upon which modern cryptography is based will fall apart. Assigning a probability to whether or not it will be proved in a given time period, however, is meaningless. Even with a useful probability, it is irrelevant for this question. Either $\mathcal{P} = \mathcal{NP}$, or it does not. If it does, then we don't need to worry about key size, since the relevant algorithms will become useless; if it does not, then the probability of proving that it does is effectively zero, and the effectiveness of our key size can be determined normally.

A more useful inspection of the usefulness of a 128-bit key would be to see how it holds up against likely modern and future adversaries. According to Handout 8, a conservative estimate of the key length necessary to protect information from an organization with the resources of an intelligence agency in 1995 was 75 bits. Applying Moore's Law, we can guess that the equivalent key length necessary for security in 2002 is 80 bits. Twenty years from now, 98 bits will be the minimum necessary for thorough protection from brute force attacks if these guidelines continue to hold.

What does it mean for a key to be too small? If a key is too short, an adversary can trivially brute-force the key values to decrypt a message. A key being too large is much harder to define. According to these numbers, a 128-bit key is significantly larger than is necessary for even moderately long-term solution. On the other hand, the numbers mentioned are guidelines, and assume that the algorithms are perfect and that Moore's Law continues to hold. The difference in processing power necessary to perform encryption using a 128-bit key is not significantly greater than that required to encrypt using a 100-bit key. Each extra bit doubles the work required on an adversary's part, however. If we are trying to design a secure algorithm, extra bits in the key certainly won't hurt, and can mean a dramatic increase in effectiveness.

Ben Bitdiddle may be correct that the 128-bit key is not necessary for good security, but the difference in computation on the user's part between minimum security and overly strong security is so small that we can err on the side of safety without even worrying much about legacy systems. 128-bit keys should provide secrecy even from well-budgeted intelligence agencies for decades as long as the algorithm is not broken. Average individual users and small corporations may not require the level of cryptographic protection offered by such a large key, but a security standard should allow for the most significant protection possible without causing undue trouble to the user. Although perhaps somewhat larger than minimally necessary, a 128-bit key standard fits this specification nicely.