12 Weird cryptography; or, How to resist brute-force attacks.

by Philippe Teuwen

> *"Unbreakable, sir?" she said uneasily. "What about the Bergofsky Principle?"*
>
> *Susan had learned about the Bergofsky Principle early in her career. It was a cornerstone of brute-force technology. It was also Strathmore's inspiration for building TRANSLTR. The principle clearly stated that if a computer tried enough keys, it was mathematically guaranteed to find the right one. A code's security was not that its pass-key was unfindable but rather that most people didn't have the time or equipment to try.*
>
> *Strathmore shook his head. "This code's different."*
>
> *"Different?" Susan eyed him askance. An unbreakable code is a mathematical impossibility! He knows that!*
>
> *Strathmore ran a hand across his sweaty scalp. "This code is the product of a brand new encryption algorithm—one we've never seen before."*
>
> [...]
>
> *"Yes, Susan, TRANSLTR will* always *find the key—even if it's huge." He paused a long moment. "Unless..."*
>
> *Susan wanted to speak, but it was clear Strathmore was about to drop his bomb.* Unless what?
>
> *"Unless the computer doesn't know when it's broken the code."*
>
> *Susan almost fell out of her chair. "What!"*
>
> *"Unless the computer guesses the correct key but just keeps guessing because it doesn't realize it found the right key." Strathmore looked bleak. "I think this algorithm has got a rotating cleartext."*
>
> *Susan gaped.*
>
> *The notion of a rotating cleartext function was first put forth in an obscure, 1987 paper by a Hungarian mathematician, Josef Harne. Because brute-force computers broke codes by examining cleartext for identifiable word patterns, Harne proposed an encryption algorithm that, in addition to encrypting, shifted decrypted cleartext over a time variant. In theory, the perpetual mutation would ensure that the attacking computer would never locate recognizable word patterns and thus never know when it had found the proper key.*

Yes, we are in a pure sci-fi techno-thriller. Some of you may have recognized this excerpt from the *Digital Fortress* by Dan Brown, published in 1998. Not surprisingly, there is no such thing as the concept of rotating cleartext or Bergofsky Principle, and Josef Harne never existed.

There is still a germ of an interesting idea: What if "the computer guesses the correct key but just keeps guessing because it doesn't realize it found the right key"? Instead of trying to conceal plaintext in yet another layer of who-knows-what, let's try to make the actual plaintext indistinguishable from incorrectly decoded ciphertext. It would be a bit similar to format-preserving encryption (FPE)[59] where ciphertext looks similar to plaintext and honey encryption,[60] which both share the motivation to resist brute-force. But beyond single words and passwords, I want to encrypt full sentences... into other grammatically correct sentences! Now if Eve wants to brute-force such an encrypted message, every single wrong key would produce a somehow plausible sentence. She would have to choose amongst all "decrypted" plaintext candidates for the one that was my initial sentence.

So starts a war of natural language models... Anything the cryptanalyst can find to discard a candidate can be used in turn to tune the initial grammar model to create more plausible candidates. The problem

[59] `https://en.wikipedia.org/wiki/Format-preserving_encryption`
[60] `http://pages.cs.wisc.edu/ rist/papers/HoneyEncryptionpre.pdf`

for the cryptanalyst C can be expressed as a variation of the Turing test, where the test procedure is not a dialog but consists of presenting n texts, of which $n-1$ were produced by a machine A, and only one was written by a human B (cf. Fig. 22.)

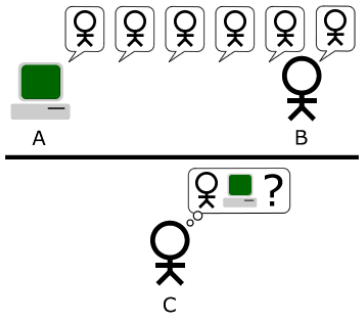

Figure 22: Turing test, our way.

We'll start with a mapping between sentences and their numerical representations. Let's represent a language by a graph. Each sentence is one path through the language graph. Taking another random path will lead to another grammatically correct sentence. To encrypt a message, the first step is to encode it as a description of the path through the grammar graph. This path has to be identified numerically (enumerated) among the possible paths. Ideally, the enumeration must be balanced by the frequency of common grammatical constructions and vocabulary, something you get more or less for free if you manage to map some Huffman coding onto it. If there is a complete map between all the paths up to a given length and a bounded set of integers, then we have the guarantee that any random pick in the set will be accepted by the deciphering routine and will lead to a grammatically correct sentence. So the numerical representation can now be ciphered by any classic symmetric cipher.

A complete solution has to follow a few additional rules. It must not include any metadata that would confirm the right key when brute-forced, so e.g., it shouldn't introduce any checksum over the plaintext that could be used by an attacker to validate candidates! And any wrong key should lead to a proper deciphering and a valid sentence, no exception.

Such encoding method covering a balanced language graph could serve as a basis for a pretty cool natural language text compressor, which works a bit like ordering the numbers 3, 10, and 12 in a Chinese restaurant. (I recommend the 12.)

In practice, some junk can be tolerated in the brute-forced candidates; in fact, even a lot of junk could be fine! For example, 99% of detectable junk would lead to a loss of just 6.6 bits of key material.

12.1 Enough talk. Show me a PoC or you-know-what!

Fair enough.

We need to parse English sentences, so a good starting point may be grammar checkers:

```
$ apt-cache show link-grammar
Description-en: Carnegie Mellon University's link grammar parser
 In Selator, D. and Temperly, D. "Parsing English with a Link
Grammar" (1991), the authors defined a new formal grammatical system
called a "link grammar". A sequence of words is in the language of a
link grammar if there is a way to draw "links" between words in such a
way that the local requirements of each word are satisfied, the links
do not cross, and the words form a connected graph. The authors
encoded English grammar into such a system, and wrote this program to
parse English using this grammar.
```

`link-grammar` sounds like a good tool to play with.

Here is, for example, how it parses a quote from Jesse Jackson: *"I take my role seriously as a pastor"*.

```
       +-----------MVp-----------+
       +-------MVa-------+       |
       +----Os---+       |       +---Js---+
 +-Sp*i+    +-Ds-+       |       |  +--Ds-+
 |     |    |    |       |       |  |     |
I.p take.v my role.n seriously as.p a pastor.n
```

The difficulty is the enumeration of paths that would cover the key space if we want to map one path to another one. So, for a first attempt, let's keep the grammatical structure of the plaintext, and we will replace every word by another that respects the same structure. After wrapping some Bash scripting around `link-grammar` and its dictionaries, here's what we can get:

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode
 @23:2 n.1:2865 v.4.2:1050 a n.1:4908 to v.4.1:1352 a adj.1:720 n.1:7124 adv.1:369
```

This is one possible encoding of the input: every word is replaced by a reference to a wordlist and its position in the list. Hopefully, another script allows us to reverse this process:

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode|./decode
my example illustrates a means to obfuscate a complex sentence easily
```

So far, so good. Now we will encode the positions using a secret key (123 in this example) with a very very stupid 16-bit numeric cipher.

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123
 @23:1 n.1:7695 v.4.2:2054 a n.1:2759 to v.4.1:2070 a adj.1:2518 n.1:5439 adv.1:123

$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123|./decode 123
my example illustrates a means to obfuscate a complex sentence easily

$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123|./decode 124
its storey siphons a blink to terrify a sublime filbert irretrievably
```

Using any wrong key would lead to another grammatically correct sentence. So we managed to build an (admittedly stupid) crypto system that is pretty hard to bruteforce, as all attempts would lead to grammatically correct sentences, giving no clue to the bruteforcing attacker. It is nevertheless only moderately hard to break, because one could, for example, classify the results by frequency of those words or word groups in English text to keep the best candidates. But the same reasoning can be used to enhance the PoC and get better statistical results, harder for an attacker to disqualify.

Actually, we can do better: let's send one of those weird sentences instead of the encoded path. This gives plausible deniability: you can even deny it is a message encoded with this method, and claim that you wrote it after partaking of a few Laphroaig Quarter Cask ;-) British neighbors are advised, however, that if this leads to the UK banning Laphroaig Quarter Cask for public safety reasons, the Pastor might no longer be their friend.

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode |./decode 123
your search cements a tannery to escort a unrelieved clause exuberantly
```

This can be deciphered by whoever knows the key:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./encode 123|./decode
my example illustrates a means to obfuscate a complex sentence easily
```

And an attempt to decipher it with a wrong key gives another grammatically correct sentence:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./encode 124|./decode
your scab slakes a bluffer to integrate a introspective hamburger provocatively
```

If someone attempts to brute-force it, she would end up with something like this:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./bruteforce
...
22366:their presentiment reprehends a saxophone to irk a topless mind perennially
22367:your cry compounds a examiner to shoulder a massive bootlegger unconsciously
22368:our handcart renounces a lamplighter to imprint a outbound doorcase weakly
22369:my neurologist fascinates a plenipotentiary to butcher a psychedelic imprint automatically
22370:their safecracker vents a spoonerism to refurnish a shaggy parodist complacently
22371:your epicure extols a governor to belittle a indecorous clip heatedly
22372:our kilt usurps a monger to punish a loud foothold indirectly
22373:my piranha mugs a resistor to evict a obstetric malaise laconically
22374:its controller unsettles a duchess to ponder a diversionary beggar riotously
22375:your glen mollifies a interjection to embezzle a forgetful decibel speciously
22376:our misdeal countermands a pedant to typify a imperturbable heyday topically
22377:their bower misstates a colloquialism to disorientate a apoplectic warrantee courteously
22378:its downpour copies a frolic to sweeten a circumspect cavalcade dispiritedly
22379:your infidel resurrects a masseuse to manufacture a differential fairway famously
22380:my abstract contaminates a birthplace to squire a unaltered subsection lukewarmly
22381:their co-op resents a deuce to inveigle a unsubtle attendant objectionably
^C
```

The scripts are available in this issue's PDF/ZIP, but the PDF itself can be used to secure your communications—because *why not?*

```
$ chmod +x pocorgtfo08.pdf
$ echo "encrypt this sentence !" | ./pocorgtfo08.pdf -e 12345
 besmirch this carat !
$ echo "besmirch this carat !" | ./pocorgtfo08.pdf -d 12345
 encrypt this sentence !
```

The PDF includes an ELF x86-64 version of `link-grammar`, so you will need to *execute* the PDF on a matching platform. Any 64-bit Debian-like distro with `libaspell15` installed should do.

For extra credit, you may construct a meaningful sentence that encodes to Chomsky's famously meaningless but grammatical example, "Colorless green ideas sleep furiously."

Ideas presented in this little essay were first discussed by the author at Hack.lu 2007 HackCamp.

Have fun!