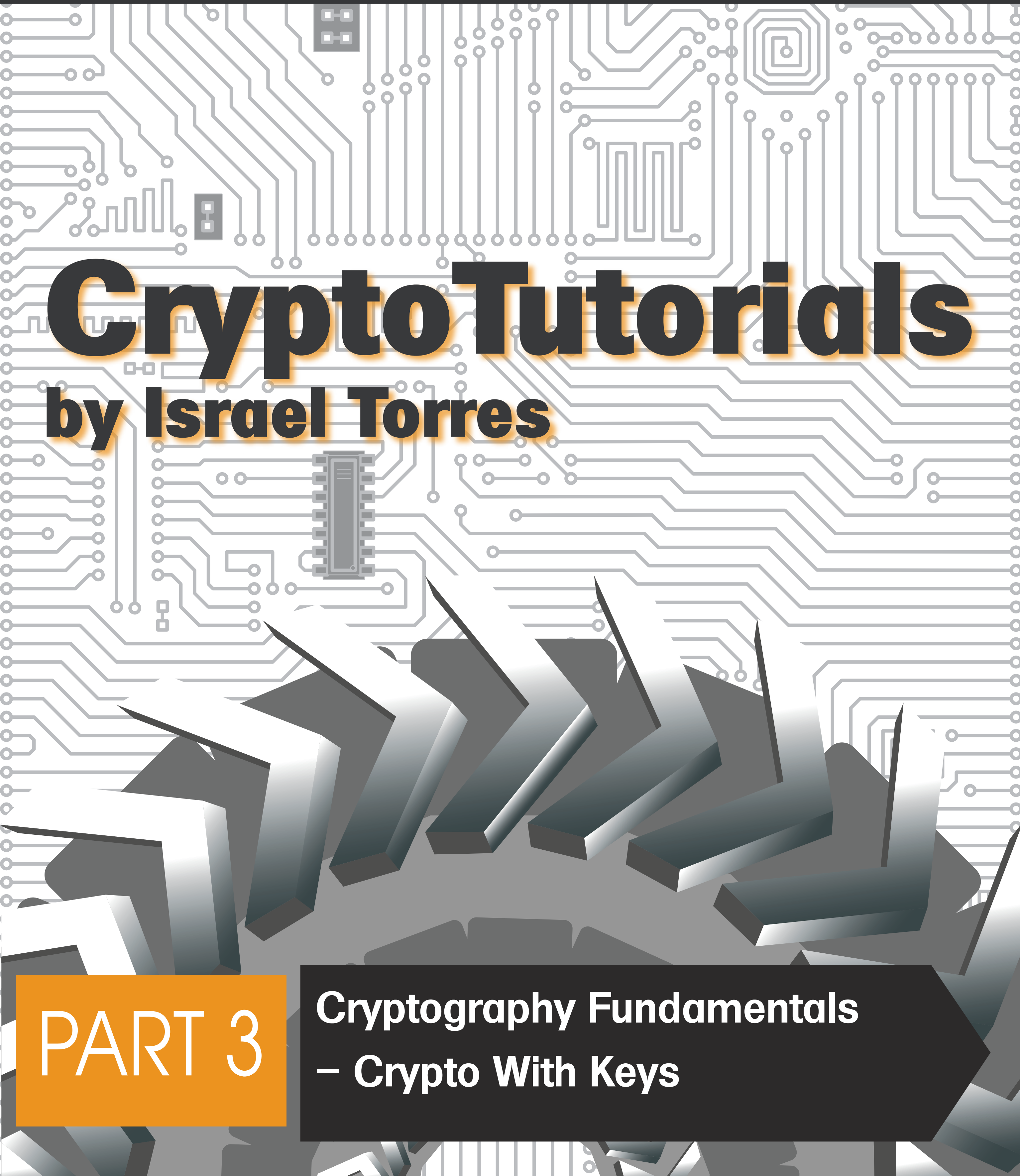
Hakin9
TUTORIALS

CryptoTutorials

by Israel Torres



PART 3

Cryptography Fundamentals – Crypto With Keys

CryptoTutorials

By: Israel Torres

What you will learn...
crypto fundamentals, how to use a key to encrypt your communications

What you should know...
basic C programming for programmatic implementation

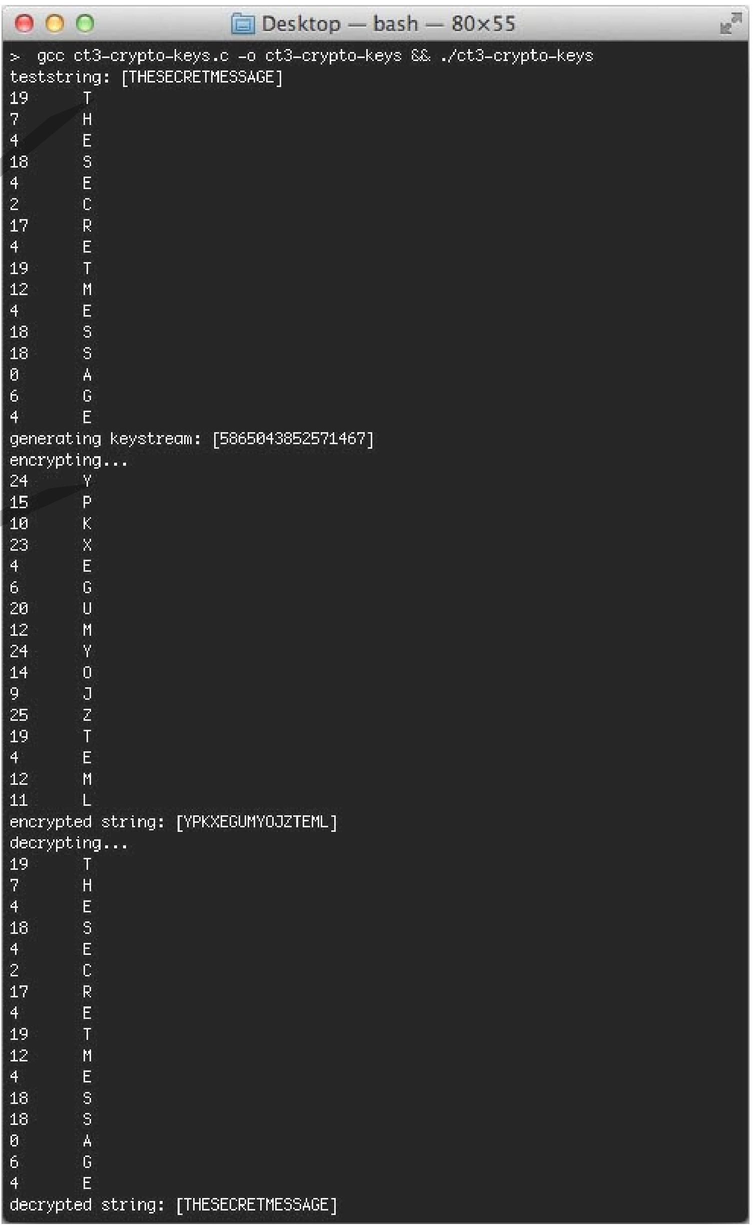
There are quite a number of cryptographic implementations out there that allow you to encrypt/encode/encipher your communications (PM (*Private Messages*), IM (*Instant Messages*), Tweets, IRC (*Internet Relay Chat*), e-mail (electronic mail), snail-mail (postal), SMS/MMS, voice, etc). In your encrypted communications you are often lead to believe your communications are secure; if you are lucky you get to know the algorithm used (whether it is public/private), and if you are even luckier you get to play with the open source code to make your affirmations sound. Most of the time you are kept guessing as to what you are sending on that cool 99 cent app is secure or not.

A lot of homegrown implementations out there are some type of text rotation (shifting letters from left to right) or XORing (bitwise-eXclusive-OR). This topic/demo will cover ASCII text messages – results you can copy and paste into text-based clients and how to protect them using a simple technique. The difference here is you'll understand how it works and how to make it better.

Note

As this is a tutorial it is certainly encouraged that you play with the source code provided as you like however it is important to understand that if you really need to make sure no one other than yourself or others you trust sees a communique you should really use a known algorithm/implementation such as AES-256, blowfish, twofish.

I've provided the source C file `ct3-crypto-keys.c` (Figure 1) which demonstrates a few of the exercises we'll be going over in this tutorial. (if you can't locate the source code contact me



```
>  gcc ct3-crypto-keys.c -o ct3-crypto-keys && ./ct3-crypto-keys
teststring: [THESECRETMESSAGE]
19      T
7       H
4       E
18      S
4       E
2       C
17      R
4       E
19      T
12      M
4       E
18      S
18      S
0       A
6       G
4       E
generating keystream: [5865043852571467]
encrypting...
24      Y
15      P
10      K
23      X
4       E
6       G
20      U
12      M
24      Y
14      O
9       J
25      Z
19      T
4       E
12      M
11      L
encrypted string: [YPKXEGUMYOJZTEML]
decrypting...
19      T
7       H
4       E
18      S
4       E
2       C
17      R
4       E
19      T
12      M
4       E
18      S
18      S
0       A
6       G
4       E
decrypted string: [THESECRETMESSAGE]
```

Figure 1. *ct3-crypto-keys console output*

– info below) Compile and run it with this one-liner in terminal:

```
gcc ct3-crypto-keys.c -o ct3-crypto-keys && .
 /ct3-crypto-keys
```

If you don't understand C it's ok I've named the variables, documented and chunked up the code so it is easy to understand even by someone that doesn't code.

For brevity let's state that our test string will aptly be `THESECRETMESSAGE` however instead of relying on the ASCII Decimal values of 65-90 for A-Z we are going to convert them to something more familiar 1-26. It's a simple matter of subtracting the value of 65 – here's an example:

```
char alpha[ALPHABET]="\0"; int counter=0, numlist=0;
for (;counter < ALPHABET; counter++){
    alpha[counter]=(counter+ALPHAUPP);
    printf(„%c\t%d\n",alpha[counter], counter+1);
}
```

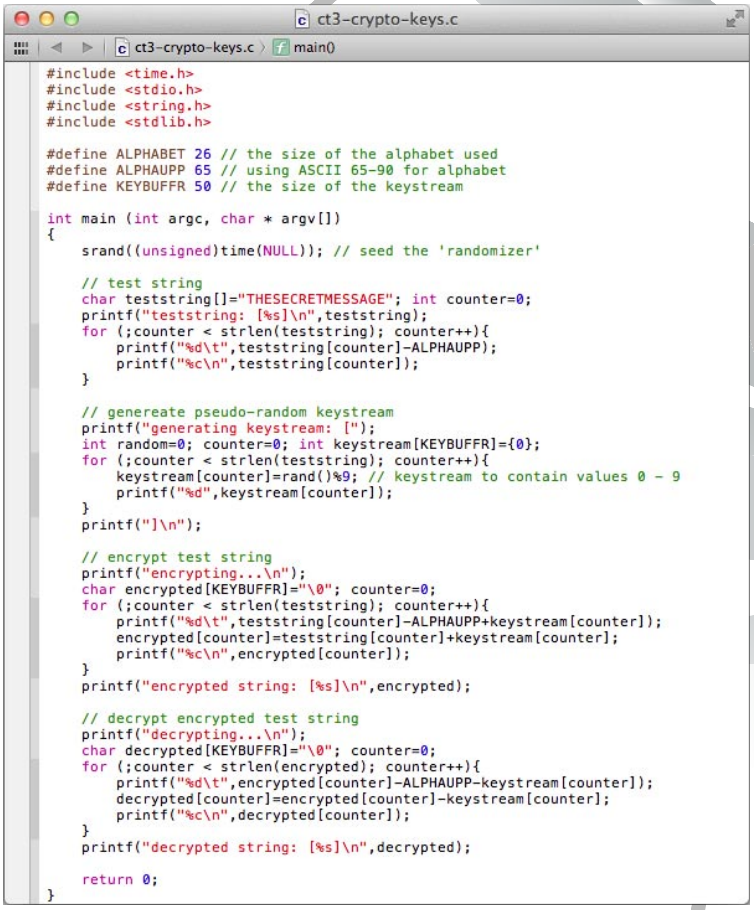
which outputs 26 lines: `A = 1 ... Z = 26`. This is handy when using familiar number systems (before the days of using personal computers to do all the dirty work). In another tutorial we'll go over how to use a similar system to handwrite a key system using these same values and then some but do it all by hand instead of using a computer. (note in this particular block we've made counter to be 1-based instead of 0-based as the rest of the code uses).

In the first block we are declaring the secret message and saving it in a string (character array) labeled *teststring* and printing it out to the console. (Figure 2). We'll be using *teststring* a couple of more times so whether we print it to the console or not we'll still need to declare it. (Later you can convert this to a more dynamic input handler but that is out of the focus of this tutorial).

In the second block we use the length of *teststring* (so we match index value for index value) to generate a pseudo-random value (in this case 0-9) and populate the values in a numeric (integer) array labeled *keystream*. *keystream* will save the key for encrypting and decrypting against. *keystream* is also the value you want to guard as it is the key to your secret message. Without knowing the value of *keystream* it will take longer to attack your message.

In the third block we use *teststring* and add the values of *keystream* so for example the value of *teststring*[0] is T which is ASCII Decimal 84 which minus 65 is 19 which is the alphabetic number 20 on a 0-based index (which we are using in our for loops) and 20 is the 20th letter in the alphabet known as T. Our *keystream*[0] in turn rolls out an integer value of 5. Adding 19 with 5 we get a value of 24 which again 0-based is the 25th letter in the alphabet being Y. To reverse this symmetrically we subtract the same value (5) to get us back to where we were. We go through this routine for the entire length of *teststring* until it is complete and show the transformed message from `[THESECRETMESSAGE]` using the *keystream* of `[5865043852571467]` to our *encrypted* array values of `[YPKXEGUMYOJZTEML]`. In this simple example we are using the *keystream* as a *one-time-pad* (OTP) as long as we only use it once. Done correctly (with a better randomizer) the cryptanalyst (the attacker in this case) will have a very difficult time restoring the message correctly as the *keystream* is the other half needed to ensure that the communication is decrypted properly.

The fourth and final block puts the *encrypted* values against the *keystream* into a *decrypted* array by simply



```
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define ALPHABET 26 // the size of the alphabet used
#define ALPHAUPP 65 // using ASCII 65-90 for alphabet
#define KEYBUFFR 50 // the size of the keystream

int main (int argc, char * argv[])
{
    srand((unsigned)time(NULL)); // seed the 'randomizer'

    // test string
    char teststring[]="THESECRETMESSAGE"; int counter=0;
    printf("teststring: [%s]\n",teststring);
    for (;counter < strlen(teststring); counter++){
        printf("%d\t",teststring[counter]-ALPHAUPP);
        printf("%c\n",teststring[counter]);
    }

    // genereate pseudo-random keystream
    printf("generating keystream: [");
    int random=0; counter=0; int keystream[KEYBUFFR]={0};
    for (;counter < strlen(teststring); counter++){
        keystream[counter]=rand()%9; // keystream to contain values 0 - 9
        printf("%d",keystream[counter]);
    }
    printf("]\n");

    // encrypt test string
    printf("encrypting...\n");
    char encrypted[KEYBUFFR]="\0"; counter=0;
    for (;counter < strlen(teststring); counter++){
        printf("%d\t",teststring[counter]-ALPHAUPP+keystream[counter]);
        encrypted[counter]=teststring[counter]+keystream[counter];
        printf("%c\n",encrypted[counter]);
    }
    printf("encrypted string: [%s]\n",encrypted);

    // decrypt encrypted test string
    printf("decrypting...\n");
    char decrypted[KEYBUFFR]="\0"; counter=0;
    for (;counter < strlen(encrypted); counter++){
        printf("%d\t",encrypted[counter]-ALPHAUPP-keystream[counter]);
        decrypted[counter]=encrypted[counter]-keystream[counter];
        printf("%c\n",decrypted[counter]);
    }
    printf("decrypted string: [%s]\n",decrypted);

    return 0;
}
```

Figure 2. *ct3-crypto-keys C source code*

Notes

All source code created and tested on:
Mac OS X 10.7 11A511
Darwin Kernel Version 11.0.0
gcc version 4.2.1

Web Links and References

- *http://en.wikipedia.org/wiki/AES*
- *http://en.wikipedia.org/wiki/Blowfish_(cipher)*
- *http://en.wikipedia.org/wiki/Twofish*
- *http://en.wikipedia.org/wiki/ASCII*
- *http://en.wikipedia.org/wiki/One-time_pad*

reversing the process as mentioned above to be able to read the message as intended.

The *keystream* can be anything from randomly generated numbers, timestamps, rss feeds, anything that is constantly changing and non-repeating is recommended. The bad part with public streams is that the attacker also has access to them. It becomes quite dangerous if the communications use things like headers (which is usually a lot how patterns are found and eventually decrypted by unintended parties). The lesson learned there is don't use headers in your messages or anything that creates a template and lastly never repeat words in the same places. One of the reasons that OTPs are frowned upon is the next problem: key distribution (how to get your secret key to your partners in crime). Since you have to have synchronized keys to match your encrypted messages getting the keys to parties not local to you can be quite challenging in more ways than one. We'll have to cover that another time. :)

Source: ct3-crypto-keys.c

ISRAEL TORRES
Israel Torres is a hacker at large with interests in the hacking realm.
hakin9@israeltorres.org http://twitter.com/israel_torres

Got More Time Than Money?
Try this month's crypto challenge:
http://hakin9.israeltorres.org