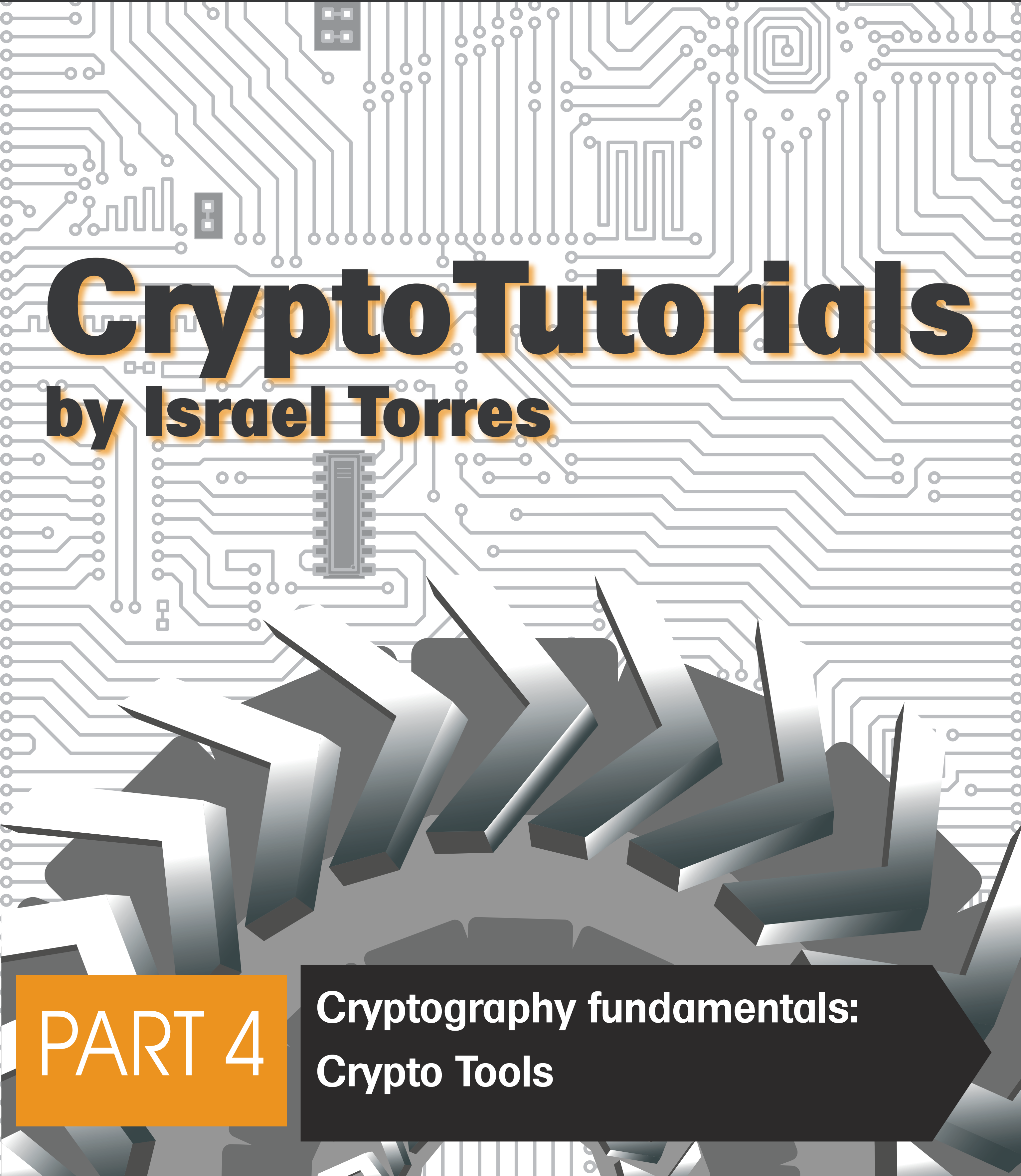
Hakin9
TUTORIALS

CryptoTutorials

by Israel Torres

PART 4

Cryptography fundamentals: Crypto Tools



CryptoTutorials

By: Israel Torres

What you will learn...
crypto fundamentals, building and testing algorithms quickly

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

Creating algorithms for encrypting/encoding/enciphering can be a timely task. Especially if creating them from scratch and implementing them on the fly. Sometimes what may seem like a fine idea ends up being a waste of time due to implementation flaws/limits unforeseen during the design phase. This tutorial demonstrates a handy set of tools to help quickly test cipher systems you've implemented.

We'll be using a simple demo C program created just for this tutorial (Listing 1). As the comments explain once compiled this application will encrypt/decrypt text strings respectively using the `-e/-d` parameters.

Most basic ciphers use a symmetric algorithm (deciphering the encipherment is simply the reverse process of enciphering without use of external keys). This specific one uses the uppercase limitation/filter of A-Z (ASCII 65 – 90). This means if the input is not in between the 26 letter alphabet of A through Z it is skipped. Otherwise the letter position is incremented by a value of one. For example if the input letter is A the result will be B, B becomes C, C becomes D. As you've probably noticed this is a simple Caesar shift cipher. Thus inversely to decipher C becomes B, B becomes A.

This implementation is purposely flawed so there is no *wrap-around*; however as some folks getting into programming and crypto may not realize there needs to be a modulo (%) component (shown in the code below) implemented for what happens when the alphabet needs to wrap around. This case becomes evident when the letter Z is shifted up by one (or more). Adding 1 to the ASCII value of Z (90) turns it into 91 which in turn is represented by the left square bracket (not a letter in the alphabet at all). The intended shift in this cipher is actually the letter A (ASCII value 65). Here's a way to handle it in C:

```
printf(„IN\tS+1\t(S+1)MOD\n");
char string[]="XYZ", tmp; int x = 0;
for (;x<strlen(string);x++){
    if (string[x] >= 'Z')
```

Listing 1. *Source demo algorithm implementation*

```
// demo.c cc demo.c -o demo && ./demo -e test
// Mac OS X 10.7.1 Lion using (GCC) 4.2.1
#include <stdio.h>
#include <string.h>
void e(char test[20]){ // -e encrypt
    int x=0; for (;x<strlen(test);x++)
        if ((test[x]>=65) && (test[x]<=90))
            printf("%c",test[x]+1);
}
void d(char test[20]){ // -d decrypt
    int x=0; for (;x<strlen(test);x++)
        if ((test[x]>=65) && (test[x]<=90))
            printf("%c",test[x]-1);
} // this demo app shifts string 1 char right
int main (int argc, char * argv[]){
    if (argc == 3) {
        if (strcmp(argv[1],"-e")==0)
            e(argv[2]);
        if (strcmp(argv[1],"-d")==0)
            d(argv[2]);
    }else{
        printf("usage: demo [-e|-d] 'text'");
        return -1;
    }
    return 0;
} // EOF
```

```
        tmp=(string[x])%'Z'+'A';
    else
        tmp=string[x]+1;

    printf(„%c\t%c\t%c\n",string[x],string[x]+1,tmp);
}
```

which when compiled and ran runs the following result:

```
IN  S+1 (S+1)MOD
X   Y   Y
Y   Z   Z
Z   [   A
```

... in the initial output (S+1) demonstrates what demo.c will output and is expected to fail the test for a successful algorithm as it does not meet the criteria of only allowing uppercase letters for input. The secondary output (S+1)MOD handles the wraparound to match correctly when the tolerance is met/exceeded.

Someone not savvy in this may miss it and only start to notice anomalies if not tested correctly. Homebrew test strings may not reach this test case and may go into production unnoticed until it is too late.

In the past I would run line by line tests then gather the output and rerun the tests using the gathered output and

Listing 2. *Source ctestio*

```
#!/bin/bash
# ./ctestio.sh
# Israel Torres <hakin9@israeltorres.org>
# Tue Aug 23 18:16:59 PDT 2011
# "CryptoTutorials 4: Crypto Tools"
# this shell script tests the encoder/decoder
                    functions
# - displays input and output character ratio
# - verfies that the hashes match
# ./ctestio.sh 'ALGORITHM' 'PLAINTEXT'
# created and tested on:
# Mac OS X 10.7.1 Lion using GNU bash 3.2.48(1)-
                    release
#
if [ ! $# -ne 2 ]; then
TESTAPP=$1; PLAIN=$2; TSTMP=$(DATE +%s)
PSHA1=$(echo -n $PLAIN | shasum | cut -d ' ' -f 1)
ENCODE=$($TESTAPP -e "$PLAIN")
DECODE=$($TESTAPP -d "$ENCODE")
DSHA1=$(echo -n $DECODE | shasum | cut -d ' ' -f 1)
PLAINLEN=$(echo -n $PLAIN | wc -c)
ENCODELEN=$(echo -n $ENCODE | wc -c)
DECODELEN=$(echo -n $DECODE | wc -c)

echo -ne "PLAIN:\t$PLAIN\nENCOD:\t$ENCODE\nDECOD:
                    \t$DECODE\nRATIO:\t"
echo $PLAINLEN$ENCODELEN$DECODELEN
if [ "$PSHA1" = "$DSHA1" ]; then
    echo -e "MATCH:\t[PASS]"
else
    echo -e "MATCH:\t[FAIL]"
fi
else
    echo "usage: $0 'ALGORITHM' 'PLAINTEXT'"
    echo "example: $0 ./2011-08-12-4 HELLOWORLD"
fi
#EOF
```

Listing 3. *Source ctestio-tests*

```
#!/bin/bash
# ./ctestio-tests.sh
# Israel Torres <hakin9@israeltorres.org>
# Wed Sep  7 07:52:29 PDT 2011
# "CryptoTutorials 4: Crypto Tools"
# this shell script runs a serial of ctestios
# add test lines below and run w/ ./ctestio-tests.sh
# created and tested on: (GCC) 4.2.1
# Mac OS X 10.7.1 Lion using GNU bash 3.2.48(1)-
                    release
function hashline { # 80 column
for i in {1..79}; do echo -n "#"; done; echo "#"
}
# compile demo algo
gcc demo.c -o demo # compile algorithm
if [ $? -eq 0 ]; then # check if compile good
hashline # first hashline - next append with
                    ;hashline
### PUT TESTS HERE BEGIN ###
./ctestio.sh ./demo ABCDEF; hashline # simple test 1
./ctestio.sh ./demo GHIJKL; hashline # simple test 2
./ctestio.sh ./demo TUWXYZ; hashline # simple test 3
### PUT TESTS HERE END!! ###
else # message to fix it
echo something went horribly wrong - fix it!
fi
#EOF
```

Notes

All source code created and tested on:
Mac OS X 10.7.1 11B26
Darwin Kernel Version 11.1.0
GNU bash, version 3.2.48(1)-release
(GCC) 4.2.1

Web Links and References

- *http://en.wikipedia.org/wiki/Caesar_cipher*
- *http://en.wikipedia.org/wiki/ASCII*

then manually validate the strings matched (eyeballing). During the long hours things may have been missed only to reappear later (and usually in compromising circumstances).

Enter ctestio (short for crypto test input output). This bash script runs a few features to help validate that the algorithm is running as expected and brings notice when failures occur during the test line. Specific features of the output log for ctestio (Listing 2) are as follows:

- unique timestamp per test
- visualization of encoded result
- visualization of decoded result
- visualization of character count (plain, encoded, decoded)
- visualization of sha1 hash result (match:PASS/FAIL)

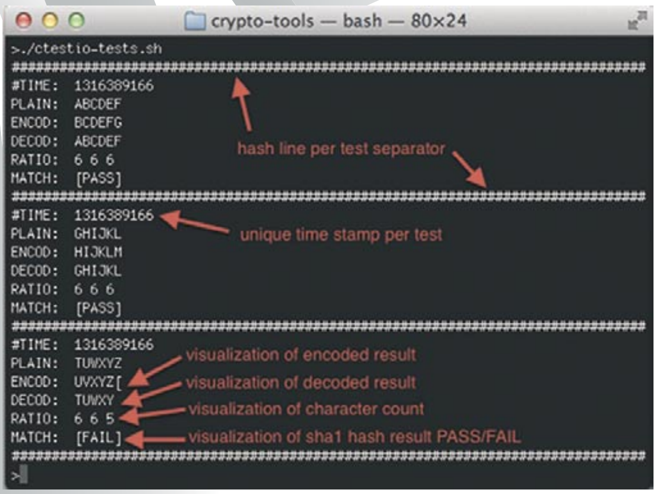
Such various methods of validations include visual inspection as well as hash comparison. The only input necessary is either the plaintext or the encrypted text. ctestio runs the string against the algorithm and feeds itself (via variable handling) the results to which it runs the comparison against and gives the results in the output. This really handy for testing *what-if* scenarios as well as building test cases for past and future algorithms which helps in finding algorithmic weak points.

The variables also hold the output as-intended so even if the output contains non-printable characters /multiline output it will feed them back during the encoding/decoding phase. You'll need to be aware of this as this normally wouldn't be able to done via the console input without special care. It also allows you to make filters for this within your algorithm implementation so such cases aren't met (on purpose).

For the batch processing part ctestio is wrapped by another shell script aptly named ctestio-tests. (Listing 3) This is what runs the test cases in batch showing each test separated by an 80 column hex line (Figure 1).

After the batch file has run and logged to file via redirector (>) it is simple enough to search for the string '[FAIL]' via grep. If all went well there will be no such string. Finding one is just another turn into finding the string that triggered the error and understanding what changes are necessary before running the batch process again.

As CryptoTutorials develops we'll be adding more tools to help build, analyze, solve and break algorithms. Such tools are best used with the full understanding of how they work so they can be rewritten in other languages other than bash and expanded upon for useful purposes.


crypto-tools — bash — 80×24
hash line per test separator
unique time stamp per test
visualization of encoded result
visualization of decoded result
visualization of character count
visualization of sha1 hash result PASS/FAIL

```
>./ctestio-tests.sh
################################################################################
#TIME:  1316389166
PLAIN:  ABCDEF
ENCOD:  BCDEFG
DECOD:  ABCDEF
RATIO:  6 6 6
MATCH:  [PASS]
################################################################################
#TIME:  1316389166
PLAIN:  GHIJKL
ENCOD:  HIJKLM
DECOD:  GHIJKL
RATIO:  6 6 6
MATCH:  [PASS]
################################################################################
#TIME:  1316389166
PLAIN:  TUWXYZ
ENCOD:  UVXYZ[
DECOD:  TUWXY
RATIO:  6 6 5
MATCH:  [FAIL]
################################################################################
>
```

Figure 1. *ctestio-tests script running test*

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