Cryptanalysis of hash-based lookup table in Python

Project Report

CSC 529 : Cryptography

Prepared by :
Min Xiang,
Jyoti Sheoran,
Prashant Chhabra

Department of Computer Science

University of Victoria

INDEX

ABSTRACT

Many HTTP Gateway frameworks are based on python. These gateway frameworks process the HTTP POST requests by first storing them into a dictionary before passing them to the application. Python dictionary is implemented using python hash() function, which is a non-cryptographic hash function. Even though the current version of python (2.7.5) has randomized its hash() by introducing a random seed when python environment starts, it is still a bad randomization. The seed can be easily recovered given only two outputs of the hash(). The hash() is vulnerable to meet-in-the-middle pre-image attack. This attack method can be used by an attacker to find multicollisions, which can be send to the web-server in the form of a large HTTP POST request. This will result in hash collisions and worst-case lookup for insert, add, etc operations. A significant amount of such data can keep the server busy in handling only that one request and thereby denying service to other requests/clients. Such an attack is called Denial-of-Service(DoS) attack on web-servers. In this project, we attempt to analyze the current python hash() function used by python dictionary. We also implemented a SipHash function for python. SipHash is a cryptographic hash function that has the best match of speed and security for small input. We also analysed the performance of Siphash compared to current hash() and other cryptographic hash functions.

INTRODUCTION

A hash function is a function that converts arbitrary length string into fixed length output. It can be considered as a lookup table which stores indexes for input strings. The hash function has complexity O(n) in best/average case scenario. When two or more input strings results in same output value, then a linked list is created for that bucket (or index) and each input string is added to the linked list. Hence, the worst-case of lookup table is O(n^2).

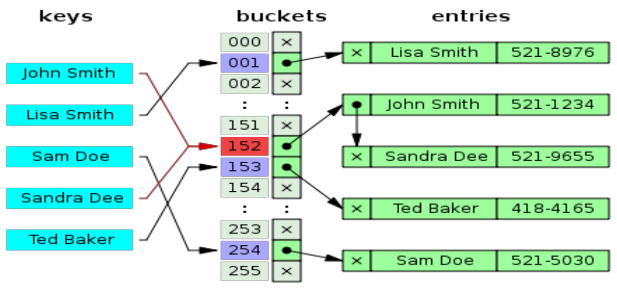

Figure 1: Hash function [1]

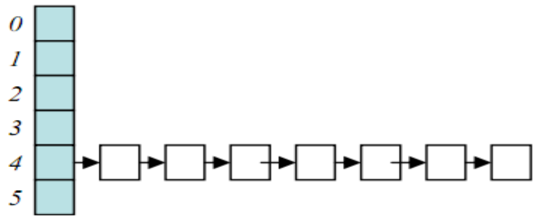

Figure 2: Worst-case hash table collisions.

HTTP gateway frameworks process HTTP POST requests by storing key-value pairs in a dictionary before passing to the application. Dictionaries are usually implemented using a hash function. A weak hash function used for implementing dictionary can make the gateway framework vulnerable to DoS attack. The goal of DoS attack is to make server unavailable to other user's request. Hash flooding is one of the method to achieve DoS. Hash flooding occurs when an attacker generates many strings that result in same hash value which need to be stored in linked list leading to an increase in server

CPU load. A sufficiently large HTTP POST data containing colliding strings can keep a CPU core busy for enough time to deny service to other clients. This will ultimately lead to Denial-of-Service.

Python uses a non-cryptographic hash function for implementing dictionary [2]. This hash() function is not collision resistant nor it is a one-way function. The hash() uses a salt value, called seed. As python hash() implementation/definition is public, it is very easy to find the seed given only two output values of the hash function.

A generic solution to this problem would be to change hash() implementation of Python to a secure hash function. We need a cryptographic hash-function, i.e. a strong pseudo-random function (PRF) and possibly a one-way function. This will ensure that even if an attacker finds two colliding keys (say, by brute-force attack), he will be unable to distinguish between these keys from any other randomly generated key and hence cannot generate more colliding keys using the existing ones. The new hash function should be fast for short-input, otherwise it cannot be used for daily tasks like (use in dictionary). The non-cryptographic hash functions, like DJBX33X, MurmurHash, CityHash, etc. also have similar weakness which can be exploited to create DoS attack. Cryptographic hash functions like SHA-1, SHA-2, SHA-3, provide good security but are quite slow. SipHash [3] is a strong PRF optimized for small input.

BACKGROUND :

Many languages have tried to implement a better hash function by adding a secret seed to the hash function. Non-cryptographic hash functions are vulnerable to hash collision attacks. These functions are neither collision resistant nor one-way (i.e. These functions are invertible). Meet-in-the-middle preimage attack can be used to compute multicollisions for such hash functions. In dec 2011, Alexander [4] presented DoS based on hash-flooding on many popular languages (including Python, PHP, Java, etc.). The primary weakness in these languages was the deterministic nature of their hash function. Alexander [4] suggested to use a random hash function. Python changed its hash function to use a randomized seed [2]. In 2012, Aumasson et al. [3], presented SipHash as a short-input fast and strong pseudo-random function that can be used as a hash-function for dictionary. They also showed that python seed can be easily recovered. Many languages has now changed their hash function to SipHash including Ruby, Perl, etc.

Algorithmic complexity attack:

The average time of a server processing a HTTP POST request is O(n) (hash-based lookup table). But if hash collision occurs, then this time increases and becomes $O(n^2)$. Hence by carefully monitoring the response time of the server, we can determine which strings are creating collisions [5]. This method can be used by an attacker to find the secret seed of a hash function that is initialized only once.

Meet-in-the-middle preimage attack:
The hash function is divided into two sub-functions. Sub-function 1 (sf1) works in forward direction (same as hash function) and subfunction 2 (sf2) works in backward direction (i.e. inverse of hash function). These two sub-functions are defined in such a way that some bits, b1 of input string are dependent only on sf1() and are independent of sf2() and some bits, b2 are independent of sf1() and depend only on sf2(). The idea is to calculate preimages that result in same hash value.
The backward function sf2 starts with a fixed hash value, h and then calculates inverse of hash for random n1-sized suffixes. The intermediate hash h1 and suffix s is then stored in a table.
The forward function sf1 work on random (n - n1)-sized prefixes, p and calculates same intermediate hash value h1. If the intermediate hash_value matches the one in the table, then we have found a colliding string for hash h. The string is (p + s).

Protection against DoS based on hash flooding :
There are many ways to provide protection against DoS hash flooding. Following are the different ways to protect against DoS hash-flooding :
1. Count the number of collisions and raise an exception if the number of collisions exceeds a certain threshold. This approach may be good for web-based gateway frameworks as they can just throw away excessive keys. However, many applications need to store all keys. (e.g. kernel)
2. Limit the POST/GET request size. This approach is also specific to web gateway frameworks only.
3. Provide a way to control CPU time. For e.g using timers or signals to raise an exception if some operation is taking too much of cpu time.
4. Check the input data for face validity before processing it. This is time-consuming and not scalable.
5. Change the hash function to a strong pseudo-random function. This is a scalable approach as it handles the problem at its root.

We studied below mentioned hash functions:
1) DJX33XA is a simple and fast hash function. It is very similar to Python hash function and is vulnerable to meet-in-the-middle preimage attack to find hash collisions.
2) MurmurHash is a non-cryptographic hash functions which gives good speed for small input. But is is vulnerable to differential attacks.
3) CityHash is also a non-cryptographic hash function. It is based on MurmurHash and is vulnerable to same differential attack.
4) Cryptographic Hash functions are strong PRF but they are very slow for small input and hence can not be used for daily purposes.
5) SipHash [3] is a cryptographic hash function. It is a strong PRF and provides speed comparable to current python hash function and other non-cryptographic hash functions.

SipHash :

SipHash is a cryptographic hash function. It takes 64-bit input string and 128-bit seed. It is specifically designed for short-input strings. Hence, it’s speed is comparable to other non-cryptographic hash functions and the current python hash function. SipHash is a strong psuedo-random hash function.

SipHash function Definition:

SipHash function consists of 3 stages - initialization, SipRounds and Finalization. SipHash-c-d means c rounds for each message block and d means d rounds at final stage.

Initialization Stage :

4 64-bit internal states v0 to v3 are initialized with a constant string xored with the key. Key is divided into two parts k0 and k1.

v0 = k0 ⊕ “str1”

v1 = k1 ⊕ “str2”

v2 = k0 ⊕ “str3”

v3 = k1 ⊕ “str4”

Here str1 to str4 are 4 parts of the string “ somepsuedorandomstring”. This string is chosen randomly.

64-bit messages m0, m1,... go through following routine :

1) v3 ⊕= m0
2) c iterations of SipRound
3) v0 ⊕= m0

Finalization :

1) v2 ⊕= ff
2) d iterations of SipRound
3) v0 ⊕ v1 ⊕ v2 ⊕ v3

SipRound :

```
v0 += v1          v2 += v3
v1 <<= 13         v3 <<= 16
v1 ⊕= v0          v3 ⊕= v2
v0 <<= 32
v2 += v1          v0 += v3
v1 <<= 17         v3 <<= 21
v1 ⊕= v2          v3 ⊕= v0
v2 <<= 32
```

Security Claims :

Key Recovery : brute-force : 2^{127}

State Recovery : 2^{191}

Internal Collisions : 2^{128}

Differential analysis : difficult after 3 rounds

THE PROBLEM :

Python Hash Function :

Python uses a modified Fowler-Noll-Vo (FNV) function as its default hash function. Below is the code for Python hash function. It is written in C language :

From objects/stringobject.c : line 1258

```
static long string_hash(PyStringObject *a)
{
    register Py_ssize_t len;
    register unsigned char *p;
    register long x;

    #ifdef Py_DEBUG
          assert(_Py_HashSecret_Initialized);
    #endif
    if (a->ob_shash != -1)
          return a->ob_shash;
    len = Py_SIZE(a);
    /*
          We make the hash of the empty string be 0, rather than using
          (prefix ^ suffix), since this slightly obfuscates the hash secret
    */
    if (len == 0) {
          a->ob_shash = 0;
          return 0;
     }
    p = (unsigned char *) a->ob_sval;
    x = _Py_HashSecret.prefix;                  /* Secret Seed */
    x ^= *p << 7;
    while (--len >= 0)
          x = ( 1000003 * x ) ^ *p++;
    x ^= Py_SIZE(a);
    x ^= _Py_HashSecret.suffix;             /* Secret Seed */
    if (x == -1)
```

```
            x = -2;
        a->ob_shash = x;
        return x;
    }
```

In Python 2.7.3, by default the secret seed is a constant value , i.e. hash function was deterministic. The seed can be randomized seed by using -R option [2]. In Python 2.7.5, seed is randomized by default [6]. The seed is chosen randomly when the python engine starts. The seed remains same for that session of python. Hence, it is equivalent to a deterministic hash function with secret seed.

Hash-flooding attack on Python :

Finding collisions using Meet-in-the-middle preimage attack :

Python hash function is deterministic. Also, its not a one-way function, i.e. it is invertible. Hence, we can use meet-in-the-middle preimage attack to find multi-collisions. This attack assumes that we already know the secret seed of the hash function.

Meet-in-the-middle attack on Python hash function :
Lets first divide the Python hash function into two sub-functions, forward-hash() and backward-hash().

1) Run the backward-hash() function and store the intermediate-hash and suffix to a table T1.
2) Run the forward-hash() function. For each intermediate-hash generated,
i) Search it in table T1, if the intermediate-hash value matches, then store prefix in corresponding row of table T1
3) This will result in multiple prefix-suffix pair. Each such pair will create a collision.

Step 1:

```
/*  Precomputation : filling the lookup table  */
repeat n times do
        s = random_suffix
        h = backward_function(s, target)
        precomp[h] := s
end
```

Step 2:

```
/*  Finding preimages   */
repeat n times do
        s = random_prefix
        h = forward_function(s)
        if h in precomp then
                print s+precomp[h]    /*  prefix+suffix = preimage  */
        end
end
```

Inversion of Python Hash Function :

```
h = end ;
h ^=  _Py_HashSecret.suffix ;
h ^=  Py_SIZE(msg) ;
len = Py_SIZE(msg) ;
for (; len>0; len - -)
    h = (h ^ msg[len-1]) * 2021759595   ;
         /* As (1000003 * 2021759595)  is congruent to 1 mod 2^32 */
```

Finding Secret Seed :
Ways to determine secret key of Python hash function :
1) Definition of python hash function is public. Secret seed can be determined easily given two key-value pairs.
2) Alternatively, if one has access to the python machine, the script provided by Aumasson et al. [3] can be used to find the seed.
3) In case of HTTP Gateway frameworks, when can use algorithmic complexity attack to find the secret seed.
The hash value of python hash function depends only on last 8 bits of prefix of secret seed. There are only 2^8 = 256 different hash functions (one for each value of 8 LSB bits of prefix). One can easily find seed using these steps :

i) Calculate colliding keys/strings for each of 256 different hash functions and pass to the server as HTTP POST request.

ii) The request which has significantly higher response time is creating hash collisions. We already know the prefix (8 LSBs) for this request. We do not need to find the full seed value, we can set any arbitrary value of the rest of the bits of the secret seed.

DoS Attack :
To perform DoS attack, first step is to find the secret seed of the python hash function. Once we know the secret seed, we can calculate more colliding keys using meet-in-the-middle preimage attack. A large set of such colliding keys sent as a HTTP POST request (e.g. form submission), can keep the server busy.

Implementation of SipHash for Python :

High-level Overview :

To implement an efficient SipHash for Python, we decided to use C programming language to implement the hash function and embed the hash function in python so that a secure lookup table module can access the hash function. Implementation in C programming language makes this solution more efficient than using python to implement SipHash, since Python store all numbers in objects, the computations of XOR, addition and bit shifting suffer more overhead in python than in C.

The work is divided into two parts: python code and C code. We use python code to build the secured lookup table module, this lookup table module reuse most functions from the default python lookup table module except hash related functions, which are rewritten to call SipHash function implemented in C through the Python C API. As shown in Fig. 1.

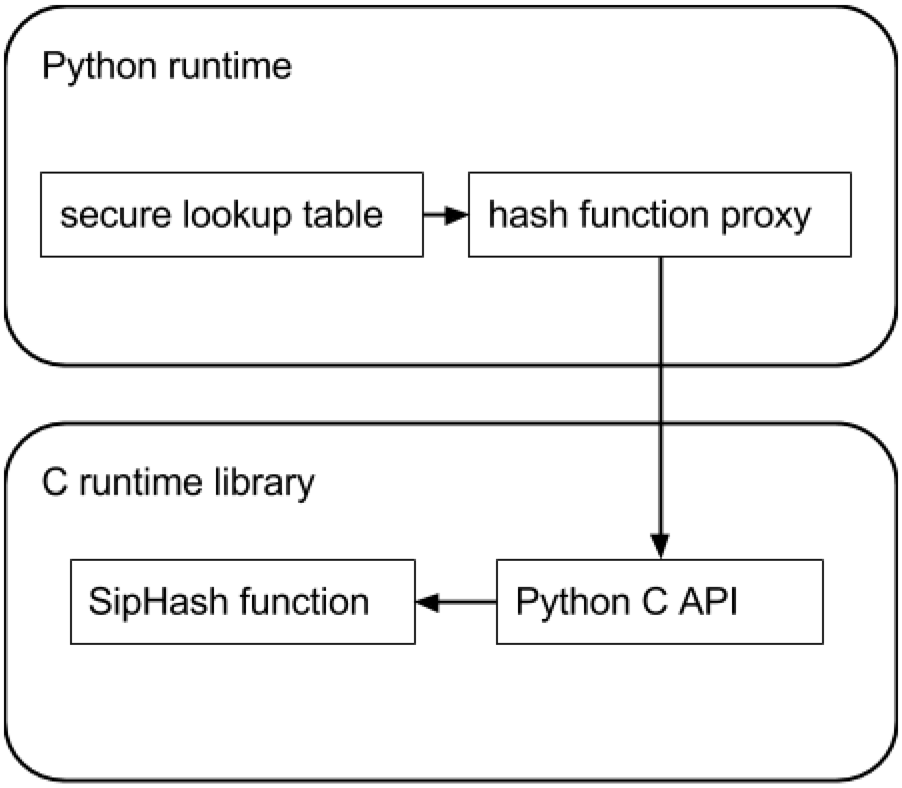

Fig. 1

In the implementation of the secure lookup table, we rewrite the function for getting and setting value. When the new (*key*, *value*) pair add the lookup table, we calculate the SipHash of the key, which is a 64-bit integer, and use this SipHash value as new key and the tuple (*key*, *value*) as new value put in to default python dictionary (lookup table). Since in python hash value of integer is itself, we achieve the goal to use SipHash value as the lookup key, thus preventing potential attacks.

Performance testing :

High-level Overview :
For performance test, we focused on the comparison of efficiency of SipHash, default hash function of Python2.7.5 and cryptographic hash function MD5 and SHA1.
Since default hash function has built-in caching feature in python runtime, we only compared MD5, SHA1 and SipHash, that we implemented in python runtime. And we abstracted the python hash function from its source code and compared it with SipHash in C runtime.

Results :
We collect the running time of hashing string of various length 2^{20} time on dual core 64-bit CPU @ 1.7GHz with 8GB dual channel RAM @ Clock Speed: 1600 MHz:

Table 1 compare SipHash and python default hash function

string length	8	32	256	1024
python2.7.5	0.084s	0.276s	1.904s	7.516s
SipHash-2-4	0.404s	0.672s	3.352s	12.429s

The result shows that speed of SipHash is comparable with python default hash function. For longer strings the SipHash took less than double of the time python default hash took.

Table 2 compare SipHash, MD5 and SHA1

string length	8	32	256	1024
SipHash-2-4	1.067s	1.106s	1.638s	3.426s
MD5	3.000s	3.015s	3.789s	6.139s
SHA1	3.409s	3.465s	5.418s	11.428s

This shows that SipHash is very efficient compared to other cryptographic hash functions.

CONCLUSION :

The current hash function used by Python is a non-cryptographic hash function. It is an invertible function and the secret seed is easy to recover. Hence, this hash function is vulnerable to hash collisions and can be exploited by attackers for DoS hash-flooding attack on web-services based on python. Python hash function must be changed to some strong pseudo-random function, like SipHash. We implemented SipHash for Python and the test results showed speed comparable to current hash function. Hence, SipHash provides both speed and security. Our implementation of SipHash can be used for hashing/dictionary instead of using the default hash function.

RELATED WORK :

Python has accepted Python Enhancement Proposal (PEP) 456 : Secure and interchangeable hash algorithm [7]. As per the latest update on 20-Nov-2013, Python has decided to use Siphash-2-4 as the default hash function for v3.4. The plan is to still keep existing FNV-based hash function for backward compatibility. Also, they are be uniting hash function definition for all objects into single definition.

REFERENCES :

1. http://en.wikipedia.org/wiki/Hash_function
2. http://bugs.python.org/issue13703 : Hash collision security issue
3. Aumasson, Jean-Philippe, and Daniel J. Bernstein. "SipHash: a fast short-input PRF." Progress in Cryptology-INDOCRYPT 2012. Springer Berlin Heidelberg, 2012. 489-508.
4. Alexander “alech”Klink n.runs AG, Julian “zeri”Wälde TU Darmstadt. "Efficient Denial of Service Attacks on Web Application Platforms" December 28th, 2011. 28th Chaos Communication Congress. Berlin, Germany.
5. Crosby, Scott A., and Dan S. Wallach. "Denial of service via algorithmic complexity attacks." Proceedings of the 12th USENIX Security Symposium. Washington: USENIX, 2003.
6. http://bugs.python.org/issue14621 : Hash function is not randomized properly
7. http://www.python.org/dev/peps/pep-0456/

APPENDIX :

A1 : SipHash Code :

```
/*****************************************************************************************************/
/* python siphash module:
       siphash24.h     siphash implementation from https://131002.net/siphash
       _siphash.c      provide siphash as a python function
       setup.py        compile setup for _siphash.c
       compile command:
                    python setup.py build
                  pick up the dynamic library under 'build/lib.XXX/'
                  XXX should be platform related.
*/
/*****************************************************************************************************/

/************************************************ _siphash.c *********************************************/
#include "python2.7/Python.h"
#include "siphash24.h"

static PyObject *
_siphash_str(PyObject *self, PyObject *args)
{
  unsigned char *str;
  int len;
  unsigned long long k[2], hash;
  if (!PyArg_ParseTuple(args, "s#KK", &str, &len, k, k + 1))
    return NULL;
  crypto_auth((unsigned char*)(&hash),
            str, (unsigned long long)len,
            (unsigned char*)k);

  return Py_BuildValue("K", hash);
}


static PyMethodDef MethodsTable[] = {
  {"siphash24", _siphash_str, METH_VARARGS, "test func"},
  {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC
init_siphash(void)
{
  (void) Py_InitModule("_siphash", MethodsTable);
}

/*
int main()
{
  printf("A\n");

  return 0;
}
*/
/****************************************************************************************************/


/*******************************************setup.py***************************************************/
from distutils.core import setup, Extension

module1 = Extension('_siphash',
                sources = ['_siphash.c'])

setup (name = '_siphash',
      version = '1.0',
      description = 'This is a demo package',
      ext_modules = [module1])

/****************************************************************************************************/
```

A2 : Secure Lookup Table:

```
/********************************************securedict.py*********************************************/
```

```
import collections
import siphash
class SecureDict(collections.MutableMapping):
  def __init__(self, *args, **kwargs):
    self.store = dict()
    for key, value in args:
      self[key] = value
    self.update(**kwargs)
    _siphash = siphash.SipHash()
    self.set_hash_function(_siphash.hash)

  def set_hash_function(self, hash_function):
    self._hash = hash_function

  def __getitem__(self, key):
    for k, v in self.store[self._hash(key)]:
      if key == k:
        return v
    raise KeyError

  def __setitem__(self, key, value):
    v = self.store.get(self._hash(key), [])
    try:
      pos = [pair[0] for pair in v].index(key)
      v[pos] = (key, value)
    except ValueError:
      v.append((key, value))
    self.store[self._hash(key)] = v

  def __delitem__(self, key):
    v = self.store.get(self._hash(key), [])
    try:
      pos = [pair[0] for pair in v].index(key)
      del v[pos]
    except ValueError:
      pass

    if v:
      self.store[self._hash(key)] = v
    else:
      del self.store[self._hash(key)]

  def __len__(self):
    l = 0
    for key, values in self.store.items():
       l += len(values)
    return l
  def __iter__(self):
    for key, values in self.store.items():
      for k, v in values:
        yield k
```

/***/

A3 : Testing Code :

/***/

```
Testing code:
        pyhash.h        python2.7 hash function isolated from python source code
        test.py         testing script in python environment
        test.c          testing script in C code
                        execute parameters:
                          test <'python' or 'siphash'> <length of string>
                        example of execution time measuring:
                          time test python 256
```

/***/

```
/****************************************pyhash.c************************************************/

#include "stdlib.h"
#include "time.h"
long prefix;
long suffix;

static long
string_hash(char *p, long len)
{
    register long x;
    register long l;
    if (len == 0) {
        return 0;
    }
    l = len;
    x = prefix;
    x ^= *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= l;
    x ^= suffix;
    if (x == -1)
        x = -2;
    return x;
}

static long rand_long()
{
  long r1 = rand();
  long r2 = rand();
  long r3 = rand();
  long r4 = rand();
  long r5 = rand();

  return r1 ^ (r2<<15) ^ (r3<<30) ^ (r4 << 45) ^ (r5 << 60);
}

void init_rand()
{
  srand(time(0));
  prefix = rand_long();
  suffix = rand_long();
}

/*****************************************************************************************************/


/*********************************************test.py*************************************************/
from _siphash import siphash24
import hashlib

from random import randint, choice
k0 = randint(0, (1 << 64) - 1)
k1 = randint(0, (1 << 64) - 1)

def siphash(s):
  return siphash24(s, k0, k1)

def md5(s):
  return hashlib.md5(s).hexdigest()

def sha1(s):
  return hashlib.sha1(s).hexdigest()

import string, time
char = string.ascii_letters + string.digits
text = ''.join([choice(char) for x in range(64)])
def test_time(name, function):
  start = time.time()
  for i in range(1, 1 << 20):
    a = function(text)
  end = time.time()
  print name, ':', (end - start), 's'

def test_group(size):
  global text
  print 'testing with string size of %d' % size
  text = ''.join([choice(char) for x in range(size)])
  test_time('py_hash', hash)
  test_time('siphash', siphash)
  test_time('md5', md5)
  test_time('sha1', sha1)
  print ''

if __name__ == '__main__':
  for i in [8, 32, 256, 1024]: #[8, 16, 32, 64, 128]:
    test_group(i)

/********************************************************************************************************/



/********************************************test.c****************************************************/
#include "stdio.h"
#include "pyhash.h"
#include "siphash24.h"

int main(int argc, char* argv[])
{
  init_rand();
  char text[256];
  char key[16];
  char out[8];
  int i;
  int len;
  for (i = 0;i < 8; i++)
    key[i] = rand() & 0xff;
  for (i = 0;i < 256; i++)
    text[i] = (rand() % ('z' - 'a' + 1) ) + 'a';

  len = atoi(argv[2]);
  if (!strcmp(argv[1],"python"))
    for (i = 0; i < (1<<20); i++)
      string_hash(text, len);
  else
    for (i = 0; i < (1<<20); i++)
      crypto_auth(out, text, len, key);
}

/********************************************************************************************************/
```