PMAC Algorithm

Course of Security in Networked Computing Systems
MSc. Computer Engineering, University of Pisa

Damiano Barone, Lorenzo Diana

A.Y 2014/2015

1 Introduction

1.1 Description

We implemented the PMAC algorithm in a distributed system. We used the C language. This algorithm describes a new message authentication code. Unlike customary modes for message authentication, the construction here is fully parallelizable. This will result in a faster authentication in a variety of settings.

1.2 Library

We used the openmp library that allows to execute parallel code, it creates threads as many as the cores/threads in the pc. We didn't use the pthread libary because it does not give us certainty to execute in parallels. The OpenMP API is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from embedded systems and accelerator devices to multicore systems and shared-memory systems. We can use this library by adding the parameter -fopenmp, example:

```
gcc worker.c -o worker -lcrypto -fopenmp
```

2 Algorithm

The PMAC construction is stingy in its use of block-cipher calls, employing just $\lceil |M|/n \rceil$ block-cipher invocations to MAC a non empty string M using n-bit block cipher. A MAC computed by PMAC can have any length up to n bits.
Unlike the CBC MAC (in its basic form), PMAC can be applied to any message M, in particular |M| need not be a positive multiple of n. Likewise, messages being MACed do not needs to be of one fixed length, messages of varying lengths can be safely MACed. When using PMAC with AES, the key for PMAC is a single AES key, and all AES invocations are under that key. We have n=128 bits and key=128, 192, 256.

Algorithm PMAC

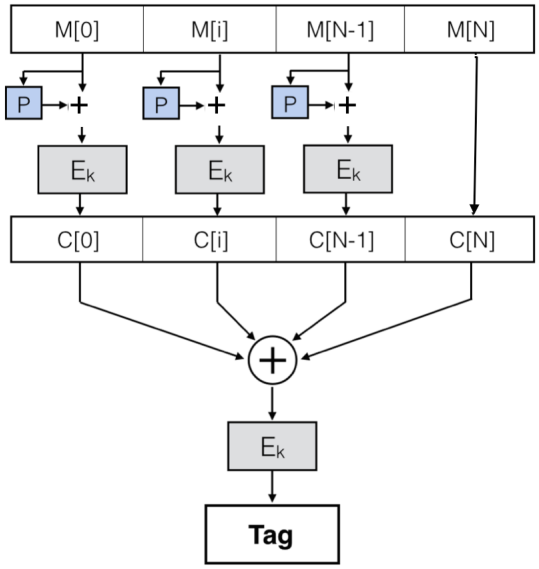
Let $m = \max\{1, \lceil |M|/n \rceil\}$
Let $M[1], \ldots, M[m]$ be strings s.t. $M[1] \cdots M[m] = M$ and $|M[i]| = n$ for $1 \leq i < m$

for $i = 1$ to $m - 1$ **do**
$C[i] = E_K(M[i] \ + \ iL)$

if $|M[m]| = n$ **then** $preTag = C[1] \oplus C[2] \oplus \cdots \oplus C[m-1] \oplus M[m]$
$Tag = E_K(preTag \ + \ \text{Final}(L))$
else $W = pad_n(M[m])$
$preTag = C[1] \oplus C[2] \oplus \cdots \oplus C[m-1] \oplus W$
$Tag = E_K(preTag)$

$T = Tag[\text{bits 1 to } tagLen]$
return T

Given a k-bit key, we can compute a key L by way of $L = E_K(0^n) \vee 0^{n\text{-}1}1$. Instantiate by computer addition of n-bit words (ignoring any carry) and instantiate iL, for i $\geq$ 1, by repeated addition. Let be Final(L) , the bitwise complement of L.
The *padding* is the string $\{0^{n\text{-}|A|\text{-}1},1, A\}$, where A are the bits of the last block, that is prepend 0-bits and then a 1-bit so as to get to length n.
Another rappresentation of the algorithm is:
P=iL



3 System Architecture

Our scenario is in a distributed system where the Server upload the job and the Worker execute a part of the job.

3.1 Server

The server receives the connection by the worker, therefore the Server know the number of available core of the Workers. When the Server upload the file, it can executes the job by command*run*. It splits the messagge in block of 128 bits, if the message is not a multiple of 128 it will add padding and send messagges to workers in proportion to the number of cores of each worker. The Server prefers use to worker with many cores. When the server receives all messagges will xor between them and do another ecryption. The final result is the MAC at 128 bits.

Commands

The server has many commands, for example upload the file, obviously the Server itemize the key (we have obbliged the Server to upload the lenght of key 256 bits) for example:

```
!upload /Users/Damiano/prova.zip 12345678901234567890123456789012
```

another command is *run*, that execute the first job uploaded:

```
!run
```

Alternatively we can use *!runall* for execute all job in the Server.
Another command is *!nclients*, that rappresent the number of worker connected.

3.2 Worker

The Worker compute the encryption of a messagge. When it connects to a Server, itl sends the number of avalaible core, then it wait until it receves a job. When it receive a messagge, it split the data in blocks of 128 bit each. It calculates the P = iL and will sum it to each i-th block and then encrypts, it will do the xor with all the blocks and then send the message of 128 bits to the server.

3.3 Worker crash

If a Worker crashes, the Server deletes the worker and reassigns the job when another worker is free. We have a list of urgent job to manage thi case.

3.4 Message exchange

M1 W $\rightarrow$ **S**: number of available cores
M2 S $\rightarrow$ **W**: E_{shared_key}(key, SHA1{key, first block index, data length, data}), first block index, data length, data
M3 W $\rightarrow$ **S**: E_{key}(data)

In the system there are 3 messages.
In the first the client connect to the server and sends his number of cores. In the second the server sends a job to the client, the job consists in:

- key, used to encrypt data of the job;
- index of the first block of the data;
- data length;
- data.

In the last message client reply to the server with the encrypted data, needed to perform PMAC. At the end client waits for new job.

4 Scenarios

4.1 Our scenario

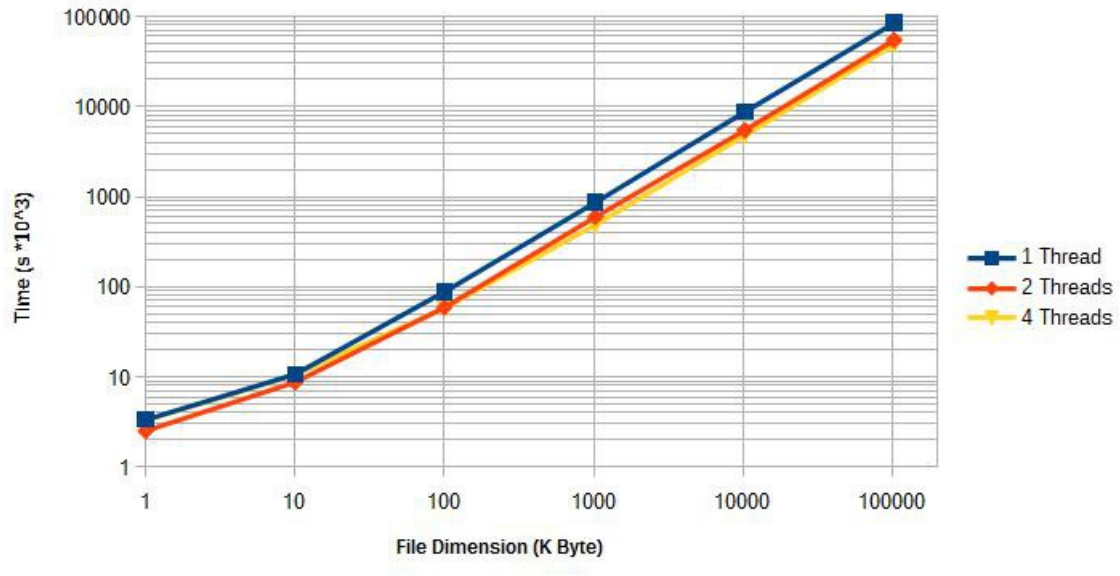
In our scenario server and workers are on different computers and comunicate via tcp connection. The channel is assumed as not secure, so in order to guarantee integrity for M2 and confidenziality of the key we encrypt the key and tha hash of key, and other data of M2, by shared_key. shared_key is a shared secret between Server and all the Workers. This last step can be avoided if we are sure that the channel is reliable and no interference can occur by third parts. The file that we want to MAC is uploaded at server side with the corresponding password.

4.2 Other scenario

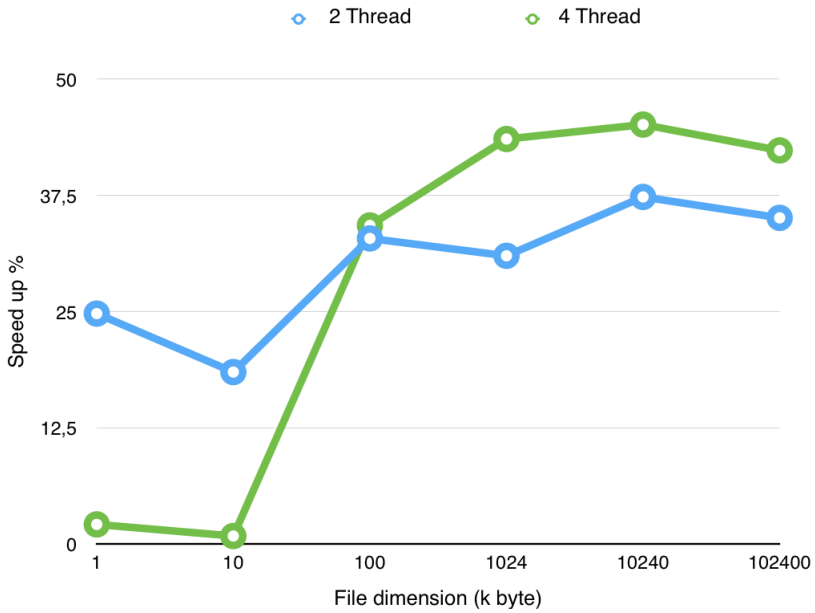
In an another scenario we have another entity: the Client. It uploads file and key, for example both encrypt by public-key cryptography. The server computes PMAC by workers and sends the reply to the client, this solution is for a client with a low cpu performance. Regarding communication channel in this case we can apply the same considerations made in the previously exposed case.
Another possible scenario is an extension for library openssl without workers but with only a server that computes the PMAC locally by exploiting parallelism of the available CPUs. In this case no communication channel is needed and we don't have to wait the corresponding overhead.

5 Performance

We show here the performance obtained by executing the Worker imposing the number of threads that have to be used.
The performance are calculated since the Server sends the blocks of the job to the worker, until the last Server encryption to obtain the MAC, so we measure the total time need to perform PMAC see by the server, including overhead produced by underlying communication channel.
The test has been made by using only one Worker executed with the Server both on the same machine. The test have been mada with a Macbook pro with Intel Core i5 dual-core at 2,7GHz, 8GB Ram, os: OSX Yosemite.



The speedup was made as a percentage respect to 1 thread.



As we can see relevant improvments are achieved passing from 1 to 2 Threads for the client, while we don't notice evident differences passing from 2 to 4 threads.

The numerical result of the test are:

1 THREAD	2 THREAD	4 THREAD	
0.000335	0.000252	0.000328	**1k**
0.001071	0.000873	0.001062	**10k**
0.008808	0.005913	0.005790	**100k**
0.086299	0.059547	0.048701	**1M**
0.870313	0.545391	0.477607	**10M**
8.4392182	5.478527	4.866202	**100M**

6 References

PMAC: A Parallelizable Message Authentication Code, Phillip Rogaway, University of California at Davis (USA) and Chiang Mai University (Thailand)