Lightweight Cryptography

Edinburgh Napier University

CSN11102

e-Security

Author:
O. Thornewill von Essen

Student Number:
40210534

Date: 16 May 2020

Abstract

IoT devices are often constrained in CPU power, RAM size, and are often powered by battery. Traditional cryptographic methods are unsuitable for constrained devices, and lightweight cryptographic methods must be implemented accordingly. Traditional cryptographic methods have the drawback of being too resource intensive for constrained devices. The type of code must also be considered when implementing lightweight crypgography. The experiment demonstrates that compiled code executes more quickly and is more efficient when compared to both just-in-time and interpreted code languages.

1 Introduction

The Internet of Things (IoT) is an ever growing area as they become more widespread in both enterprise and consumer environments. The IoT sphere is often focussed in connecting physical objects without human interaction (Naru, Saini, & Sharma, 2017). IoT devices allow objects to be connected at any time (Goyal, Sahula, & Kumawat, 2019). Examples of IoT devices could include smart home sensors which may perform temperature monitoring, or may exist on other household appliances (Naru et al., 2017). The computational processing power requirement is often low for IoT devices, which allow them to be relatively constrained in their hardware. Constraints could be battery power, CPU architecture and operating speed, size of memory, among others (Omrani, Rhouma, & Sliman, 2018).

The reason that IoT device usage is increasing is due to their low cost and efficiency at the tasks for which they are intended. However, this does not come without predicaments. As IoT devices can transfer data over public networks, it is essential that communications are confidential and that the data holds integrity. (Goyal et al., 2019; Lee, Lin, & Huang, 2014). Adversaries can always be listening; therefore, it is important to consider these aspects of security before deploying an IoT network (Sharif & Mansoor, 2010; Lee et al., 2014; Naru et al., 2017).

While traditional cryptographic algorithms such as AES and RSA exist, these are considered to require an undesirably high amount of computational power for IoT devices. There are two challenges with regards to IoT device cryptography. The first challenge is that new lightweight cryptographic algorithms must be developed, which can keep confidentiality and integrity while suiting IoT device computational restrictions (Goyal et al., 2019; Buchanan, Li, & Asif, 2017). The second challenge is that IoT devices must be able to perform their primary functions without being greatly hindered by the lightweight cryptographic algorithms (Beaulieu et al., 2015; Buchanan et al., 2017).

2 Literature Review

2.1 IoT devices and embedded systems

The constrained resources on IoT devices make it such that running processes must be as efficient as possible as to save battery power (Omrani et al., 2018). There are various existing CPU architectures; however, there are two primary standards. The first is x86 which was developed by Intel, and the second is ARM. Servers, desktop computers, and laptops all typically make use of x86 architecture CPUs, whereas low powered devices such as smartphones, RFID cards use ARM architecture (Blem, Menon, & Sankaralingam, 2013).

The x86 architecture CPUs focussed on increasing processing speed, decreasing processing time while using more energy. Increased computational power at the cost of increased energy usage is not a problem when there is consistent power supply as is with servers or desktop machines. On the other hand, IoT devices often are powered by a battery, and therefore the x86 architecture is less suitable. ARM architecture CPUs can be a solution in its much more efficient architecture which has been focussed on conserving battery power compared to processing speed (Blem et al., 2013; Jaggar, 1997).

The recent increase in ARM performance and the introduction of 64-bit ARM CPUs have led to the more widespread usage of this architecture (Cloutier, Paradis, & Weaver, 2016). Therefore, it is vital to create lightweight cryptographic algorithms which suit ARM CPUs and reduce the impact on battery consumption.

2.2 Lightweight Cryptography

There are two types of cryptography. Firstly there is conventional cryptography intended for use on servers or desktop. Secondly, there is lightweight cryptography that is designed to be used on IoT devices and embedded systems (Omrani et al., 2018). Lightweight crypto seeks to address the unsuitable application of conventional cryptograpgy on IoT devices (Beaulieu et al., 2015). There are already existing lightweight cryptography algorithms, which include PRESENT, SPECK, SIMON, PRINCE, and RECTANGLE among others (Omrani et al., 2018). However, the different algorithms have their use cases, and therefore it must be analyzed which algorithm is most suitable when deployed. For example, SPECK is most efficient in terms of its memory usage; RECTANGLE is best in terms of processing speed and PRESENT is best in terms of the security that it provides (Omrani et al., 2018).

At one point, one must question the depth of security required for the use case. Moores law declares that computing power will double every two years. The growth of computing power is exponential, and parallelization can further increase computing power significantly. The risk therefore is that algorithms can be rendered insecure in less time than anticipated. However, one should take into consideration the impact of the loss of confidentiality of data in the future. Moreover, there is a high cost associated in preventing theoretical brute force attacks that perhaps may never happen in practice (Beaulieu et al., 2015).

A primary requirement of lightweight cryptography is that the chosen algorithm should not impact the performance of the main task too significantly (Buchanan et al., 2017). Naturally, any additional computational requirements will change device usage. This requirement is crucial because otherwise, time-sensitive operations may not be executed quickly enough. A second vital requirement of lightweight cryptography is that algorithms should be practical across many different families of IoT devices (Beaulieu et al., 2015). There will be little use in an algorithm which is only efficient on one type of hardware as IoT devices and embedded systems all have different specifications such as CPU clock speed, amount of memory, and battery capacity.

Selecting a lightweight cryptographic algorithm also entails a compromise between high security, high performance speed and low memory usage (Omrani et al., 2018). While there is a benefit of higher security, there is the drawback that extra computing power is required. On the other hand, if the algorithm throughput must be prompt, then the protection against adversaries could be negatively impacted if a shorter key size is used.

SIMON and SPECK are both block ciphers released by the NSA in 2013 and have been accepted by the National Institute of Standards and Technology (Beaulieu et al., 2015). SIMON and SPECK are both designed to offer security on constrained devices. The International Organization for Standardization/International

Electrotechnical Commission (ISO/IEC) have accepted the PRESENT cipher in ISO/IEC 29192-2:2012 (Goyal et al., 2019).

All of the ciphers mentioned above are block ciphers which means that input data is sectioned into chunks before being ciphered with the key. Also, these are symmetric key algorithms which means that the same key is used to encrypt and decrypt the data. The symmetric key has the advantage that of being more battery efficient as the keys are often of shorter length.

2.3 PRESENT Cipher

The PRESENT cipher functions in a similar format to the Advanced Encryption Standard (AES) cipher, which has been unbreakable so far (Goyal et al., 2019). AES is a traditional cryptographic method used on non constrained devices. PRESENT challenged AES in that there had not yet been any alternative block ciphers aimed at constrained devices (Bogdanov et al., 2007). In figure 1, it can be seen that AES uses larger block sizes and key sizes than PRESENT. While the potential level of security is decreased for PRESENT, this is less of a concern when the focus is data integrity, or when data is not sensitive (Bogdanov et al., 2007).

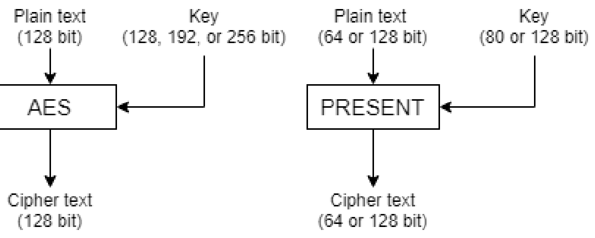
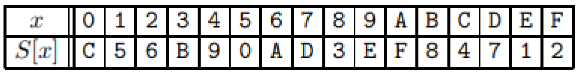

Plain text (128 bit)
Key (128, 192, or 256 bit)
AES
Cipher text (128 bit)
Plain text (64 or 128 bit)
Key (80 or 128 bit)
PRESENT
Cipher text (64 or 128 bit)

Figure 1: AES vs. Present

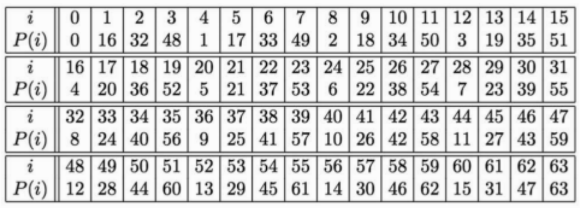
PRESENT is built on a substitution-permutation network (SPN) consisting of 31 rounds of operation (Goyal et al., 2019; Bogdanov et al., 2007). The rounds include round key operation, substitution box (S-Box), and permutation box (P-Box) - as seen in figure 4. The block size of the PRESENT cipher is 64 bits, and there are two key variants; 80 and 128 bit. In a further attempt to optimize the cipher for hardware, it implements a 4-bit S-Box compared to the traditional 8-bit or 16-bit S-Box used in traditional cryptography.

The function of the 4-bit S-Box seen in figure 2 (Bogdanov et al., 2007), is to mix the data in an attempt to obscure the relationship between the key and ciphertext. For example, 0xA would turn into 0xF.



x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Figure 2: 4-bit Substitution Box



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Figure 3: Permutation box

The function of the P-Box is to perform bitwise operations such as permute or transpose bits. For example, The 6th position bit would shift to the position of the 33rd bit. The full P-Box can be seen in figure 3 (Bogdanov et al., 2007).

2.4 Code available to implement PRESENT

The PRESENT cipher can be implemented using various programming languages. Examples include C, Golang or Python among other languages, and are made up of two families of code. The first family is compiled code which is converted by a compiler to machine-executable code. Languages such as C, Haskell, Golang are compiled languages. Coding languages that need to be compiled can often allow a deeper level of control, for example, control of memory management or CPU usage. The second family is interpreted languages, which are executed on a line-by-line basis. Coding languages such as PHP, Python, JavaScript are interpreted languages. The substantial advantages which compiled languages have over interpreted languages are the speed of execution and the efficiency of execution. On the other hand, the interpreted languages often have the advantage of being more versatile and can be executed on different platforms without having to re-write the code.

Golang is an interesting language as it contains the advantages of both compiled and interpreted code. Firstly, Golang can be executed on different platforms without having to adjust the code. Secondly, Golang is efficient due to the nature of the compiled code. A second language with similar advantages is PyPy, which is an alternative implementation of the standard Python language. PyPy operates by interpreting python code and executes as a just-in-time (JIT) compiler (Yegulalp, 2019). The JIT compilation method is beneficial concerning execution speed and

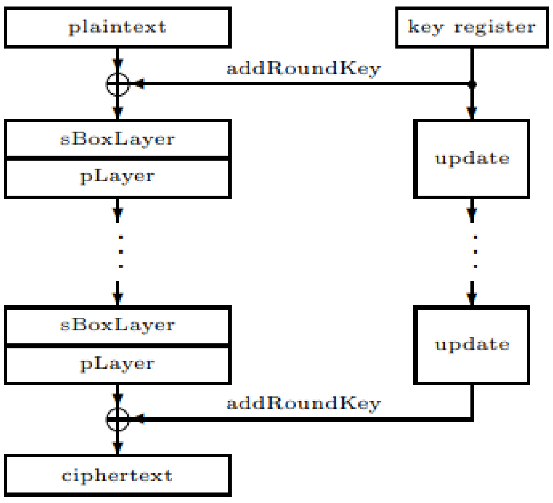


Figure 4: Present cipher

efficiency.

In the upcoming section, the PRESENT cipher will be implemented in Golang and Python. The speed of operation will be investigated between Golang, Python and the JIT compiled version of Python, PyPy.

3 Implementation

The full code used for implementation can be seen in the GitHub repository submitted with this paper.

3.1 Python

Python 3 was used for this project. There are already existing PRESENT cipher implementations in Python (Serhiichuk, 2015), allowing one to call the cipher functions. Therefore the focus of implementation of the Python code was a method of measuring the speed of execution. To time the speed of execution, a timer class was created. A more basic version of the implemented timer class can be seen in code listing 1. It is seen in code listing 1 that the elapsed time is returned upon stopping the timer.

```python
import time

class TimerError(Exception):
    """A custom exception used to report errors in use of Timer
    class"""

class Timer:
    def start(self):
        self._start_time = time.perf_counter()

    def stop(self):
        elapsed_time = time.perf_counter() - self._start_time
        return elapsed_time
```

Listing 1: Timer class

In code listing 2, the sample plaintext and 80/128 bit keys can be seen. Directly after starting the timer, the PRESENT cipher function is called, and is executed a specified number of times. The number of executions is defined as a command-line argument when executing the python script. Upon completing the execution of the PRESENT cipher, the timer is stopped and the elapsed time is reported. Using this information, it can derive the duration of the operation. A sample of the benchmark can be seen in listing 3.

```python
def benchmark():
    plain = "AAAAAAAA"
    key_80 = "AAAAAAAAAAAAAAAAAAAA"
    key_128 = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

    p_80 = Timer()
    p_80.start()
    for _ in range(0,args.benchmark):
        present_80(plain, key_80)
    print(f"Present_80\t{p_80.stop():0.6f} seconds")

    p_128 = Timer()
    p_128.start()
    for _ in range(0,args.benchmark):
        present_128(plain, key_128)
    print(f"Present_128\t{p_128.stop():0.6f} seconds")
```

Listing 2: Benchmark function

```
$ python3 main.py --benchmark 1000
[+] Performing 1000 rounds
Present_80  1.002271 total seconds
            0.001002 s/operation
Present_128 1.016137 total seconds
            0.001016 s/operation
```

Listing 3: Python benchmarking PRESENT cipher

The python3 code can be used using PyPy without having to adjust the code. A benchmark test can be seen in listing 4.

```
$ pypy3 main.py -b 1000
[+] Performing 1000 rounds
Present_80  0.469334 total seconds
            0.000469 s/op
Present_128 0.473651 total seconds
            0.000474 s/op
```

Listing 4: PyPy benchmarking PRESENT cipher

3.2 Golang

The Golang version 1.13.8 was used. There is an existing implementation of the PRESENT cipher on Github in the yi-jiayu/PRESENT.go repository. It is possible to import the repository files into a Golang build file.

Golang has an in-built method to benchmark the performance of its code. To achieve this, a test file must be created that makes use of test cases. The purpose of the test case is to verify that the code is functioning correctly additionally. Some example cases can be seen in listing 5, using both `0x00` byte arrays, and `0xFF` byte arrays. Using the individual cases, a function encrypts the data, and a different function decrypts the ciphertext.

```
var cases = []struct {
    Key        string
    Plaintext  string
    Ciphertext string
}{
    {
        Key:        "00000000000000000000",
        Plaintext:  "0000000000000000",
        Ciphertext: "5579C1387B228445",
    },
    {
        Key:        "FFFFFFFFFFFFFFFFFFFF",
        Plaintext:  "FFFFFFFFFFFFFFFF",
        Ciphertext: "3333DCD3213210D2",
    }
}
```

Listing 5: Test cases

The result of the benchmark can be seen in listing 6

```
$ go test present_test.go -bench=.
goos: linux
goarch: amd64
BenchmarkBlock_Enc/80-bit_key      1000000000   0.000016 ns/op
BenchmarkBlock_Enc/128-bit_key     1000000000   0.000012 ns/op
BenchmarkBlock_Dec/80-bit_key      1000000000   0.000019 ns/op
BenchmarkBlock_Dec/128-bit_key     1000000000   0.000020 ns/op
```

Listing 6: Golang benchmark

4 Evaluation

After implementing the PRESENT cipher in both Python and Golang code, the next stage is to measure the performance of the different languages. It was first predicted in section 2.4 that Golang would perform more quickly when compared to traditional Python. The second prediction is that PyPy would perform more rapidly than conventional Python, however not as quickly as a natively compiled language.

To measure the average time required for the Python implementation of encryption and decryption 500,000 rounds were executed and the duration in seconds measured. Using the duration, it is possible to derive the combined duration per encrypt and decrypt operation - a sample can be seen in listing 3. The same method was used to measure the performance of PyPy without adjusting the code - an example can be seen in listing 4. The Golang benchmark functions automatically executed 1,000,000,000 rounds of each encryption and decryption and reported the duration per operation in nanoseconds - which can be seen in listing 6.

It was found that Python performs encryption and decryption of one block of plaintext in 0.001002 seconds for both 80-bit key length, and 0.001016 seconds for 128-bit

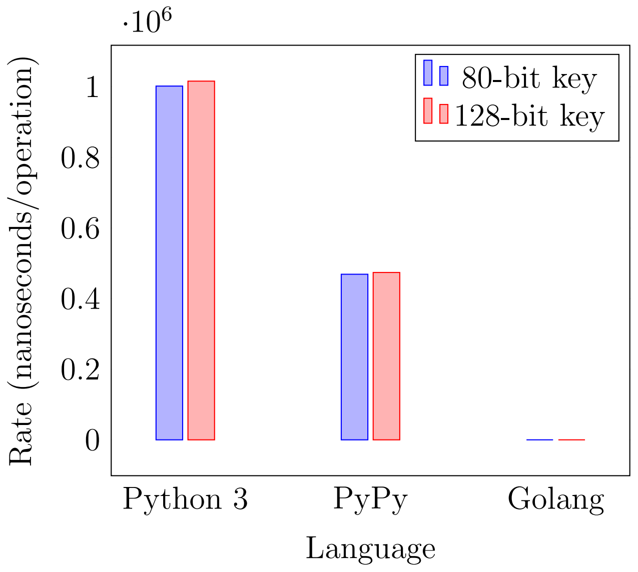


Figure 5: Execution Rate of Code

key length. As predicted, PyPy performed more quickly, where one block of plaintext could be encrypted and decrypted in 0.000469 seconds for an 80-bit key, and 0.000474 seconds for a 129-bit key. The increase in performance from traditional Python 3 to PyPy is 52.7%. Golang executed more quickly, at 0.000035 nanoseconds combined for single block encryption and decryption using 80-bit and 128-bit keys. Figure 5 visually displays the performance of Python 3, PyPy, and Golang in nanoseconds. It is seen that Golang performs more quickly compared to both Python3 and PyPy.

There are two limitations with the evaluation method. Firstly, only the duration of execution has been measured, and the power was not taken into acocunt. While PyPy performed more quickly than traditional Python, it is possible that PyPy consumed more power. However, it can be assumed that when the Python, PyPy and Golang operations are performed that they use the maximum power available. Therefore while the power was not measured it is still possible to assume that Golang is more power efficient than Python or PyPy.

Secondly, due to hardware limitations, the performance was measured on x86 architecture rather than an ARM architecture CPU. It cannot be stated how much the performance of Golang is quicker than traditional Python or PyPy. However, due to the nature of compiled code, it can be assumed that Golang would perform more quickly than Python or PyPy on the ARM architecture CPU as well.

5 Conclusions

IoT devices are often limited in their computing power having slower clock speeds, less amount of RAM, and are often powered by a battery. The CPU architecture moved away from x86 to ARM architecture as ARM is more suitable when considering power consumption.

Traditional cryptographic methods are not ideal for hardware constrained devices for two reasons. Firstly when looking at AES for example, the block size could be too large for the small communication messages of IoT devices. Secondly, the encryption/ decryption process of AES is more robust than a lightweight cipher at the cost of computing power. The problem when using cryptography that is so robust, one could be protecting against an attack which may never happen on data which could be considered low impact when confidentiality is lost. The purpose of the cryptography is not only to gain privacy, but also the integrity of transferred data. For this reason, IoT devices require lightweight cryptographic methods; examples include SIMON and SPECK, RECTANGLE, and PRESENT, among others.

There are many coding languages available to implement lightweight ciphers. Python is often considered as it does not need to be compiled, and the code is flexible to perform on different systems without having to adjust the code. Nevertheless, the downside of Python is that it is an interpreted code language, thereby performing slower than a compiled language such as C or C++. However, Golang is a compiled language which is flexible to run on different platforms without having to adjust the code. Flexible execution is beneficial for constrained devices as the implementation does not focus on one targetted device. It was measured that PyPy - a just in time compiled version of Python - is 52% faster than traditional Python. However, Golang is the most efficient implementation, and accordingly, can be recommended for use on devices with constrained hardware.

References

Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., & Wingers, L. (2015). The simon and speck lightweight block ciphers. In *Proceedings of the 52nd annual design automation conference* (pp. 1–6).

Blem, E., Menon, J., & Sankaralingam, K. (2013). Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 ieee 19th international symposium on high performance computer architecture (hpca)* (pp. 1–12).

Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J., ... Vikkelsoe, C. (2007). Present: An ultra-lightweight block cipher. In *International workshop on cryptographic hardware and embedded systems* (pp. 450–466).

Buchanan, W. J., Li, S., & Asif, R. (2017). Lightweight cryptography methods. *Journal of Cyber Security Technology*, *1*(3-4), 187–201.

Cloutier, M. F., Paradis, C., & Weaver, V. M. (2016). A raspberry pi cluster instrumented for fine-grained power measurement. *Electronics*, *5*(4), 61.

Goyal, T. K., Sahula, V., & Kumawat, D. (2019). Energy efficient lightweight cryptography algorithms for iot devices. *IETE Journal of Research*, 1–14.

Jaggar, D. (1997). Arm architecture and systems. *IEEE micro*, *17*(4), 9–11.

Lee, J.-Y., Lin, W.-C., & Huang, Y.-H. (2014). A lightweight authentication protocol for internet of things. In *2014 international symposium on next-generation electronics (isne)* (pp. 1–2).

Naru, E. R., Saini, H., & Sharma, M. (2017). A recent review on lightweight cryptography in iot. In *2017 international conference on i-smac (iot in social, mobile, analytics and cloud)(i-smac)* (pp. 887–890).

Omrani, T., Rhouma, R., & Sliman, L. (2018). Lightweight cryptography for resource-constrained devices: a comparative study and rectangle cryptanalysis. In *International conference on digital economy* (pp. 107–118).

Serhiichuk, Y. (2015). *Present-cipher.* `https://github.com/xSAVIKx/PRESENT-cipher`. GitHub. ([Git commit: 5f01dfe23e751ff2838c1df93e0b974c6d143cfc])

Sharif, S. O., & Mansoor, S. (2010). Performance analysis of stream and block cipher algorithms. In *2010 3rd international conference on advanced computer theory and engineering (icacte)* (Vol. 1, pp. V1–522).

Yegulalp, S. (2019). *What is pypy? faster python without pain.*

https://www.infoworld.com/article/3385127/what-is-pypy-faster-python-without-pain.html. ([Online; accessed 05-May-2020])