Novel Instruction Set Architecture Based Side Channels in popular SSL/TLS Implementations

Student Name: Vaishali Garg
IIIT-D-MTech-CS-IS-13-018

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Sambuddho Chakravarty (Thesis Advisor)
Hrishikesh B. Acharya
Mohan Dhawan

Submitted in partial fulfilment of the requirements for the Degree of M.Tech. in Computer Science, with specialization in Information Security
Abstract

Various Open Source Cryptographic Libraries are being used these days to implement the general purpose cryptographic functions and to provide a secure communication channel over the internet. These libraries, that implement SSL/TLS, have been targeted by various side channel attacks in the past that result in leakage of sensitive information flowing over the network. Side channel attacks rely on inadvertent leakage of information from devices through observable attributes of online communication. Some of the common side channel attacks discovered so far rely on packet arrival and departure times (Timing Attacks), power usage and packet sizes. Our research explores novel side channel attack that relies on CPU architecture and instruction sets. In this research, we explored such side channel vectors against popular SSL/TLS implementations which were previously believed to be patched against padding oracle attacks, like the POODLE attack. We were able to successfully extract the plaintext bits in the information exchanged using the APIs of two popular SSL/TLS libraries.
Acknowledgments

I wish to express my gratitude to Mrityunjay Gautam for his unparallel guidance and sharing his expertise throughout the course of this work. The efforts he puts in work always inspire me to work hard. It is my pleasure to work with him in Citrix R&D India Pvt. Ltd.

I sincerely thank Dr. Sambuddho for his support and motivation.

I would also like to thank the entire Product Security Team especially Manasi Sachdeva from Citrix R&D India Pvt. Ltd. for the valuable suggestions.

A special thanks to all my friends for critically reviewing and commenting on the work and helping to make it what it is.
Contents

1. Introduction  
   1.1. Problem Description  
   1.2. Our Contribution  

2. Related Work  

3. Background Work  
   3.1. Padding  
      3.1.1. PKCS5 Padding  
      3.1.2. PKCS7 Padding  
   3.2. Attacks  
      3.2.1. Padding Oracle Attack  
      3.2.2. POODLE Attack  

4. Our Approach  
   4.1. Simulations involving OpenSSL Library  
   4.2. Simulation involving wolfSSL Library  

5. Experimental Results  
   5.1. OpenSSL Results  
   5.2. wolfSSL Results  

6. Discussion And Future Work  

7. Conclusion
1. **INTRODUCTION**

SSL/TLS is a framework used for establishing secret communication channel. They make use of cryptographic primitives to provide confidentiality, integrity & authentication between two hosts. These protocols ensure the establishment of an encrypted link between the server & the client and hence, allow secure transmission of sensitive information.

Various Open Source Toolkits Cryptographic Libraries like OpenSSL, wolfSSL and PolarSSL implement the SSL / TLS protocols as well as the general purpose cryptographic functions to provide encrypted communication over the Internet. These libraries are widely used in industry. OpenSSL is platform independent & according to a report [24] it is used by some of the top websites of the world, WolfSSL which is having 20 times smaller memory footprint than OpenSSL is used in many types of network devices such as smart devices on automobiles, IP phones, mobile phones, printers, credit card scanners, and PolarSSL is being used in wide variety of embedded devices. WolfSSL is particularly useful in an environment where footprint size is critical or memory usage per connection makes a big impact on the performance.

Various vulnerabilities of these libraries are being exposed every year, which result in leakage of sensitive data effecting millions of users. These vulnerabilities are exposed by various observable attributes of an encrypted online communication for instance packet size and timing. Such attributes are often referred to as *side–channel information* of an encrypted traffic which can provide us some insights about the transmitted data.

Various attacks have targeted the popular SSL/TLS implementations in the past. For instance, in 2010, Juliano discovered a practical padding oracle attack on web applications[9] which exposed the vulnerability caused by CBC mode & the padding scheme used while decrypting a plaintext using any block cipher algorithm. In 2011, he again discovered a new side channel attack named as BEAST [22] which combined crypto and browser weaknesses and effected major software vendors. In 2012, padding oracle attack revived on cryptographic hardware. Also, a new side channel attack named as CRIME [23] was explored which was based on compression in packet size of HTTP request. This attack effected all the SSL/TLS implementations which use inbuilt compression schemes to reduce network congestion. Again, in the year 2013, CRIME revived in the form of BREACH [16] and the year 2014 saw the revival of padding oracle attack in the form of its new variant named as POODLE [13]. Thus, it can be seen that a trend has been followed in past where the base attack strategy remains the same but the attacks are revived every year by different side channels and there can be more possibility of existence of such unknown side channels. So, through our research we have explored such possible side channels which can make these libraries vulnerable in future.
1.1 Problem Description

Till now the researchers have looked into side channels based on packet arrival time[1], [5], packet size [2], [3], [4] and electromagnetic radiations [1] and various mitigations have been implemented to protect these libraries against these attacks [1], [21]. However, none of the research efforts till now have looked into CPU architecture oriented side channels. Through our research we have explored such side channels where we focused on popular SSL/TLS implementations such as OpenSSL and wolfSSL and we discovered that these libraries are vulnerable to such side channel attacks. Our research involved testing such side channel attacks on two architectures viz. x86 and ARM. We implemented the two popular side channel attacks, viz. padding oracle attack[2] and POODLE[13] and tested them against OpenSSL and wolfSSL respectively in a simulated environment. Thereafter, we discovered that novel side channels exist which reveal sensitive information sent over the network.

1.2 Our Contribution

Through this work we have

a) **Explored Novel Side Channels**: New side channel vectors against popular SSL/TLS implementations that rely on CPU architecture and instruction set have been discovered. Some of these side channels which leak sensitive information are number of busy CPU cycles, number of memory accesses, number of register reads and writes etc. Approximately twenty such new attributes have been discovered through this work.

b) **Successful decryption of cipher text using aforementioned side channels**: By performing a statistical analysis, it was discovered that CPU architecture related attributes such as CPU cycles, memory accesses, register reads and writes depict a change in behaviour while going through the block cipher decryption algorithm. This change is clearly observable for the case when the decryption process results in correct plaintext value. Using this information leaked by such side channels we were able to decipher all the bytes of the cipher text successfully.
2. RELATED WORK

At Eurocrypt 2002 Vaudenay presented an attack in [6] where he described the attack on CBC mode encryption when a particular padding method is used. He claims that while decrypting a message using a block cipher algorithm the padding format is always checked. The attack requires an oracle which on receipt of a cipher text, decrypts it and replies to the sender whether the padding is valid or not. This reply from the oracle is either in the form of an error message or an acknowledgment which he claims to be the side channel and can be exploited to decrypt the entire cipher text without knowing the key. This attack is known by the name of padding oracle attack and in this research he explained the attack strategy through which the entire message sent over the network can be decrypted.

In a new research [7], Paterson examined the security of various types of padding schemes that can be used along with CBC mode and described that ISO CBC-mode standards can also be exploited by padding oracle attacks. Further research done by Yau[8], claimed that padding oracle attacks as described by Vaudenay in [6] were still effective for CBC mode encryption having secret and random IVs. He also provided with some mitigation steps that can be followed in order to avoid such vulnerabilities. Black and Urtobía came up with a new research in [5] where they generalised Vaudenay's attack to other padding schemes and modes of operations. They demonstrated the attack against various symmetric encryption schemes and argued that side-channels are bound to crop up again and again as long as the adversary is allowed to freely produce valid cipher texts having predictable relationship with the underlying plaintext. They presented that the combination of chosen-plaintext security and integrity of cipher texts should be adopted to protect the applications against such attacks.

Various side channel attacks based on server response time, power consumption and server error messages have been described in brief in [1]. Juliano in [9] described the practical use of padding oracle attacks on web applications where they demonstrated how to break CAPTCHA systems and decrypt JSF view state. At Ekoparty 2011, he again demonstrated a new attack viz. BEAST - Browser Exploit Against SSL/TLS [22] on the most famous website PayPal in which he decrypted HTTP requests and obtained secret cookies. This attack combined crypto and browser weaknesses. It was also notified that some of the major software vendors like Mozilla and even Google were at risk. A new side channel attack viz. CRIME was again explored by Juliano in 2012 [23] where the compression size of the HTTP request was exploited. All SSL/TLS implementations were effected by this attack because they use inbuilt compression mechanisms to reduce network congestion. At Black Hat 2013, Yoel Gluck come up with an attack viz. BREACH [16] which was a revival of CRIME attack. Here the base attack strategy was similar to CRIME attack but this attack took advantage of HTTP level compression instead of TLS layer compression. This attack exploited the compression size of HTTP response instead of HTTP request as done in case of CRIME attack. This attack can be mitigated by adding length - hiding to TLS.
Padding Oracle Attack also revived in the form of POODLE [13] in the year 2014. POODLE is a new variant of padding oracle attack which exploits PKCS7 which is a different type of padding scheme. This attack created an impact on all the SSL/TLS implementations giving support for SSL version 3.0. The worst part about this attack was that it can't be mitigated except by completely disabling SSL v3.0 on all the servers. In later 2014 it was again found that POODLE not only bites SSL v3 but it also effects TLS v1.

Thus, it can be observed that new attacks keep on target various SSL/TLS implementations via different side channels and multiple such attacks have been discovered so far.
3. BACKGROUND WORK

3.1. PADDING

Symmetric key algorithms like 3DES, AES etc. operate on blocks of input data. For this to happen, the length of the input data must be exactly equal to the multiple of the block length for that algorithm. In order to achieve this, the end of the message is filled with some data in order to complete the last block which is called as padding. For example, let us take AES 128-bit encryption which has a block length of 128-bits or 16-bytes. Suppose, the input data (plaintext) to be encrypted has size 28-bytes. As the size of the input data is not a multiple of block length thus, we need to pad it. In this case,

28 bytes will require (16-(28-16)) = 4 bytes of padding.

Various kind of padding schemes can be used by an algorithm some of which are described below.

3.1.1 PKCS5 PADDING

In PKCS5 padding scheme the message is padded with bytes whose value show the number of padding bytes. For instance, for AES algorithm with a block size of 16 bytes ,PKCS5 padding will add bytes as follows :

![PKCS5 Padding Diagram](image)

Initial message is of length 11 bytes

```
0x32 0x21 0x90 0xde 0x28 0x57 0xcf 0x41 0x02 0x31
```

Padding is done to complete a block of 16 bytes

```
0x32 0x21 0x90 0xde 0x28 0x57 0xcf 0x41 0x02 0x31 0x05 0x05 0x05 0x05 0x05
```

Figure 3.1 :PKCS5 Padding

3.1.2 PKCS7 PADDING

For an algorithm with block length as ‘L’ and no. of padded bytes required to be added to complete a block length is ‘M’ then, the last byte of the message will be filled with a value M & rest M-1 bytes will be filled with a random value. For instance, for 3DES algorithm with a block size of 8 bytes, PKCS7 padding will add bytes as follows :
3.2. THE ATTACKS

3.2.1 PADDING ORACLE ATTACK

The padding oracle attack queries an oracle for different values of the cipher text & then tries to determine the plaintext based on the response from the oracle. This attack works by detecting the response from the server that tells the client whether padding is valid or not. The response of the server can be just the difference in time i.e. the time taken by the server when the padding is valid is longer than the time taken by the server when the padding is invalid or the number of ALU accesses can be more for valid padding as compared to invalid padding case.

Oracle can be anything which responds as true or false, yes or no or other conditions for different brute forced values of the cipher text. In a web application, the server acts as ‘the oracle’ when the client can tell from the server response when the padding is valid or invalid.
The working of the padding oracle attack is described in the figure below.

![Diagram of Padding Oracle Attack](image)

(a) Padding Invalid  
(b) Padding Valid

Figure 3.3: Padding Oracle Attack for different brute forced values of cipher text

The attack works under the assumption that the attacker can intercept the padded messages encrypted in CBC mode and he has access to the aforementioned oracle. The attacker then performs brute force by sending different values of the cipher text. The oracle responds invalid for all the values except for one where the padding is valid.

**Example Scenario**

Suppose there is a web application which stores the encrypted session token of the client in the URL parameter ‘token’ as follows:


Suppose, the block cipher algorithm used on the server side is 3DES which has a block length of 8 bytes. Now, to attack the oracle server, we first need to break up the cipher text into blocks. Now, as the algorithm used has a block length of 8 bytes & the length of the parameter ‘token’ is 16 bytes in hexadecimal so we divide it into 2 blocks such that, 

\[ C_1 = 'a55a15d599a1673e' \text{ and } C_2 = '6e62a997052847fd' \]

Now, we will send two cipher text blocks into ‘the oracle’:

1) The first block is a block of custom made cipher text which could contain anything (random or null byte). The important thing is the last byte that affects padding should be sought by means of brute force to create padding to be valid.

2) The target block will be \( C_2 = '6e62a997052847fd' \) which is fixed in each request.
The block cipher that is sent to the server (oracle) is shown below:

![Image of block cipher output]

Now, as we are interested in last byte of P2, so we have to find the value at which P2 becomes 01 (valid padding) which is shown by the figure below.
means, $A \text{xor} B = 01$

Figure 3.4: CBC Mode Decryption
Brute Force The Last Byte

Let us look into the deeper process where we start brute forcing the last byte of C1 to make the last byte of P2 as 01 so that the padding is valid.

Figure 3.5: Oracle response for different values
As shown in the figure 3.5, we send different variants of the cipher text to the oracle where last byte of second last block varies from 00 – ff. We keep track of two things i.e. server response and server response time for all the brute forced values and check if there is any difference between the brute forced values and the correct value. For example, in figure 3.5 (a) we are interrogating the oracle with the question “A xor0x00 = 0x01?”, and check server response & server response time. Similarly, we try with different combinations from 00-ff. There will be only one value between 00-ff which for which the padding status will be valid which is 0x39 here (as given by the experiment). The server response will be same in all the cases i.e. error 404 server not found but the server response time for this correct value will be a little more as compared to other values. The server takes more time to give response for this particular value because as soon as the server finds the padding to be correct, it starts decrypting the other bytes of the cipher text which involves some cryptographic operations. As the cryptographic operations are expensive & take some time to complete hence, the server takes a little more time to give response.

So, now the oracle has taken more time to respond for the equation A xor 0x39 = 0x01 hence, we can obtain the value of ‘A’ by solving this equation i.e. A = 0x39 xor 0x01 = 0x38. Thus, in figure 3.4 we are able to obtain the last byte of “Intermediate block” of C2. But, we are interested in last byte of P2. Now, according to CBC mode decryption, $P_2 = (\text{Intermediate block of } C_2) \ xor \ (C_1)$. Because, the last byte of C1 is 0x3e therefore, $P_2 = (0x38) \ xor \ (0x3e) = 0x06$
Brute Force the Second Last Byte

Now, after obtaining the last byte of P2 we are interested in the second last byte of P2. We apply the same process as previous but this time the desired conditions of valid padding are 02-02 which is shown by the figure below.

![CBC Mode Decryption Diagram]

Figure 3.6: CBC Mode Decryption

Here, the last byte of first cipher text block is set to 0x3a because now the correct padding will be at 0x02 and hence, 0x38 xor 0x02 = 0x3a.

Now again, we start brute forcing the second last byte of C1 to make the second last byte of P2 as 02 so that the padding is valid. We try with different variants of cipher text with second
last byte varying between 00-ff. This time we interrogate the oracle with the question “A xor B = 0x02?”

After completing the brute force process we found that this time the oracle takes more time to give the response at the value B = 0x63 and hence, A = 0x63 xor 0x02 = 0x61 which is the second last byte of intermediate block of C2. Now to obtain the second last byte of P2 we know that, P2 = (Intermediate block of C2) xor (C1). Because, the last byte of C1 is 0x67 therefore, P2 = (0x61) xor (0x67) = 0x06.

Similarly, we proceed with all the other bytes till we reach the first byte and in this manner we are able to decrypt the entire cipher text without knowing the key & the algorithm used.

### 3.2.2 POODLE ATTACK

POODLE (Padding Oracle On Downgrade Legacy Encryption) takes advantage of PKCS7 padding where the block cipher padding is not deterministic. This attack works in a chosen plaintext context where the attacker is interested in data that gets protected with SSL such as cookies, session tokens etc. For this attack to work the attacker must be able to:

- a) Inject data of his own before & after the secret value that he wants to obtain;
- b) Inspect, inject & modify the resulting bytes on the wire.

A real world scenario where an attacker can have such privileges is a Web context. Say, a user is logged into bank.com (some bank website or an e-commerce site) where his session is authenticated by a session cookie saved in his browser. Now, an attacker who is the owner of the site “evil.com” makes a request to “bank.com” either by HTML image tag or a JavaScript image object and carries out a CSRF (Cross Site Request Forgery) by forcing the user (victim in this case) to click on it. Because of the browser’s usual way of working where whenever a request is sent to a specific domain the cookies associated to that domain are also sent across, so as soon as the user clicks on the image object which redirects him to the site “bank.com” , the cookies stored in the user’s browser are also sent along with the request. In this way, the attacker is able to obtain the secret token of the user in an encrypted form which he can further use to carry out the POODLE attack.

We can carry out the attack on a POST request which looks as follows:

```
POST /path Cookie: name=value...\n\r\n\r\nbody ǁ‖20byteMAC ǁ‖padding
```

Here, we can control both the request path and the request body so we can induce requests such that the **two conditions** hold as follows:

- a) Cookie’s first unknown byte appears as the final byte in one of the previous encrypted blocks (encrypted as Ci)
- b) Total request length should be such that a full block of padding is required to be added at the end (encrypted into Cn)
For example, suppose the plaintext is as follows:

**Plaintext:** POST/gp/redirect/indi.htmlCookie:token=wdHZJaxBe .......... (48 bytes)

Here, the cookie’s first unknown byte appears as the last byte of one of the 8-byte blocks (highlighted in red) which fulfils the first condition of POODLE attack. This block is called as Ci. Also, as these are 48 bytes so, a full – block of padding called as Cn will be added at the end which fulfils the second condition.

The Cipher text corresponding to the above plaintext is as follows:

```
Cipher text:
5f06fc6cc5ff7b230fdcf40b1d0603ca0067df2ec6a8f3be0067df2ec6a8f3be5c48fe59
6cca889f
```

Here, the text highlighted in red corresponds to the plaintext block having cookie’s first unknown byte as the last byte and the text highlighted in green corresponds to the last block of plaintext where the padding will be added.

We then carry out the attack by arranging the request in such a manner that fulfils these conditions in different iterations and replace the last block of the encrypted record (Cn) with a copy of the block that contains the cookie’s first unknown byte (Ci) as follows:

```
Cipher text:
5f06fc6cc5ff7b230fdcf40b1d0603ca0067df2ec6a8f3be0067df2ec6a8f3be5c48fe59
6cca889f
```

Then we apply the process similar to padding oracle as described in Section 3.1. We send the cipher text blocks into ‘the oracle’ such that:

1) The second last block Cn-1 is a block of custom made cipher text which could contain anything (random or null byte). The important thing is the last byte that affects padding should be sought by means of brute force to create padding to be equal to 0x00.

2) The target block will be Cn which is replaced by Ci as described above and it will remain fixed during each request when the last byte of Cn-1 will be brute forced.

The record that is sent to the oracle after changing the second last block and replacing the last block Cn with Ci as follows:

```
5f06fc6cc5ff7b230fdcf40b1d0603ca0067df2ec6a8f3be0067df2ec6a8f3be5c48fe59
6cca889f
```
Then, we start brute forcing the last byte of the second last block starting from 0x00 – 0xff until we get a positive response from oracle. This time we expect the oracle to respond positive when Pn becomes 0x00. This is shown by the figure below:

means, \( A \ xor \ B = 0x00 \)
As shown in the figure above, we send different variants of the cipher text to the oracle until we get a response from the oracle as valid padding. For example, in figure 3.8 (a) we are interrogating the oracle with the question “A xor 0x00 = 0x00?” , & it turns out the answer is ‘no’ so, we try with different combinations from 00-ff. There will be only one value between 00-ff which will yield the padding status to be valid which is 0xc9 here (as given by the experiment).

So, now the oracle has given ‘valid padding’ response for the equation A xor 0xc9 = 0x00 and hence, we can obtain the value of ‘A’ by solving this equation i.e. A = 0xc9 xor 0x00 =
0xc9. Thus, in figure 3.7 we are able to obtain the last byte of “Intermediate block” of Cn/Ci. But, we are interested in last byte of Pi. Now, according to CBC mode decryption, Pi = (Intermediate block of Ci) xor (Ci-1). Because, the last byte of Ci-1 is 0xbe therefore, Pi = (0xc9) xor (0xbe) = 0x77 which is the hex equivalent to ‘w’ which was the first unknown byte of cookie as shown in the plaintext before.

Similarly, we proceed further by injecting our own data iteratively before or after the session cookie in order to fulfil the conditions of POODLE attack. This is shown by the figure below:

Unknown byte appears as Final byte of a 8 byte block

| POST/gp/ redirect /indi.ht mlCookie Cookie: token= | dHZJaxBe | .......
| POST/gp/ redirect /ind.htm lCookie: token=wd | HZJaxBe. | .......
| POST/gp/ redirect /in.html Cookie:token=wdH | ZJaxBe.. | .......
| POST/gp/ redirect /i.htmlCookies= | J.......... | .......

CB C B C B C B C B
| 0 1 2 3 4 5 6 |
| 8 16 24 32 40 48 56 |
| Bytes Bytes Bytes Bytes Bytes Bytes Bytes |

Encrypts into Ci-1 Encrypts into Ci

Fig. 3.8 : Iterations of plaintext for POODLE attack

In this manner, we are able to decrypt the complete value of the secret session cookie.
4. APPROACH

Various side channel attacks such as Padding Oracle Attack [6], BEAST [22], CRIME [23], BREACH [16], POODLE [13] targeted various SSL/TLS implementations in the past where it was observed that even if the attack strategy remains the same, new attacks expose vulnerabilities of these libraries via different channels. Thus, the aim of this research was to explore such new side channel attacks that involve peculiarities of CPU architecture which lead to the leakage of data in OpenSSL and wolfSSL libraries which are some of the popular SSL/TLS implementations.

This research has been carried out using a computer system simulation platform which is designed for use in computer architecture research. Any such simulation platform can be used for this purpose. However, we have used GEM5 [14]. It supports various Instruction Set Architectures (ISAs) such as Alpha, ARM, MIPS, Power, SPARC and x86. In this research, GEM5 is being used in system emulation mode to gather various computer architecture related data such as CPU cycles, ALU accesses, register reads etc. Through this research it has been observed that there exist various computer architecture related side channels which can be exploited in future to gather sensitive data, making these libraries vulnerable to these side channels.

The complete research was divided into two phases. In the first phase, the library OpenSSL is taken into consideration and the second phase focus upon the library wolfSSL.

4.1. Simulations involving OpenSSL Library

There are various implementations of OpenSSL library namely the stock version, the mobile version and the desktop version. This research has been carried out on CPAN implementation of Perl’s OpenSSL library.

The initial phase involves, programs carrying out the encryption and decryption of a plain user input and encrypted text respectively were written in Perl language. The encryption algorithm used for this purpose was AES in CBC mode with 128 bit key and standard padding where, inbuilt support for this algorithm was given by CPAN. These programs are similar to the programs hosted on client side or server side to carry out the encryption & decryption of user input.

In the next step, these programs were required to be analysed to determine any new side channels if exist via which sensitive information can be leaked. For this purpose, Gem5 simulator has been studied and it was found that it is the most widely used tool to carry out the research related to computer architecture. There are two modes in which this platform can be used.

First, System Emulation Mode, is used for running and studying individual applications or set of applications. Second, Full System Mode, is used to study how does the OS effects our application or devices.
The GEM5 tool takes the static binary of an application as input, analyses it and provides data related to computer architecture such as number of CPU cycles consumed, number of times ALU has been accessed, number of register reads and writes happened and much more. A total of 430 such application specific attributes can be obtained from this simulator for one static binary fed into it. This simulator platform supports various Instruction Set Architectures (ISAs) such x86, ARM, MIPS, Power etc. Thus, applications related to multiple ISAs can be easily analysed with this tool and hence, it also provides support to carry out architecture specific research. As Gem5 only takes static binary of an application thus, static version of Perl viz. App-StaticPerl 1.4 was configured to build static binaries for our encryption & decryption program.

Our research approach has been described by the figure.

In the Figure 4.1, user input as plaintext was fed to the encryption program which in return provided us a cipher text. The length of the plaintext was chosen in such a manner that the last block is always less than the AES block size of 16 bytes so that this block is eventually padded in order to complete the block size of 16 bytes. Then the static binary of decryption program was loaded into the GEM5 simulator. After this, the cipher text obtained from the encryption program was modified in order to carry out the padding oracle attack. To extract the last byte of the original plaintext, the last byte of second last block of cipher text was changed from 0x00 – 0xff as described in Section II and these 255 variants of cipher text were fed to the decryption binary under GEM5 and approximately 430 attributes’ data such as number of CPU cycles, memory reads, memory writes etc. corresponding to each variant of cipher text was obtained. A statistical analysis of this data was then performed and it was observed that a correlation exists between the measured attributes’ data and the 255 variants of the cipher text where an anomaly could be seen corresponding to the correct value for around 20 such attributes which leaked sensitive information. The list of such side channels is as follows:

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 – bit AES encryption</td>
<td>Ciphertext</td>
</tr>
<tr>
<td>255 cipher text variants</td>
<td>GEM5 simulator</td>
</tr>
<tr>
<td>128 – bit AES decryption static binary</td>
<td>Statistical Analysis</td>
</tr>
</tbody>
</table>

Figure 4.1. : OpenSSL Approach
1. Number of busy CPU cycles
2. Number of times Integer Registers were read
3. Number of times Integer Registers were written
4. Number of times CC Registers were read
5. Number of times CC Registers were written
6. Number of Integer ALU accesses
7. No. of instructions committed
8. No. of micro operations committed
9. No. of Bytes read from memory
10. No. of Bytes written to the memory
11. Number of read requests responded to by memory
12. Number of write requests responded to by memory
13. Number of memory references
14. Packet count per connected master and slave (bytes)
15. Packet size per connected master and slave (bytes)
16. Number of times a function call or return occurred
17. Number of instructions that are conditional controls
18. Number of load instructions
19. Number of store instructions
20. Number of branches fetched

The same process as described in Figure 4.1, was carried out iteratively in order to extract all the other bytes of the plaintext starting from the second last byte till the first byte. For extracting the next bytes the attack strategy followed was the same as described in the padding oracle attack in Section 3.2.1 but this time the block size was equal to 16 bytes.

After analysing the source code of OpenSSL for standard padding it was found that first the length of the padding is determined by looking at the last byte of the plaintext. Say, this value is ‘L’. Then, number of bytes starting from the end of plaintext and equal to the length of padding are checked. If all the bytes turn out to be equal to the value ‘L’ which means the padding is correct then further processing is done else the processing stops & the function returns a null value as soon as first mismatch occurs. Thus, it can be seen that when the padding is correct then further processing takes more CPU cycles, memory accesses and register reads. This, correct padding results into 20 different side channels which can be used to extract correct value of the plaintext.

Thus, positive results were obtained for OpenSSL Library and new side channels involving CPU architecture were explored which are explained in Section 5.
4.2. Simulations involving wolfSSL Library

wolfSSL is another popular library providing support for SSL/TLS as well as other cryptographic operations. It is 20 times smaller than OpenSSL and completely written in C language. It has been optimized to minimize size and maximize speed. It is specifically designed for use in embedded devices which require small footprint size and low runtime memory usage in order to improve the performance. Thus, it is an optimal SSL and cryptography solution.

Initially our research approach followed for wolfSSL Library was the same as described in figure. The encryption and decryption programs using wolfSSL API involving AES-128 bit block cipher algorithm with CBC mode were written in C language. Then, a plaintext was given as user input to the encryption program. The plaintext was kept in such a manner that it is always a non-multiple of AES block size of 16 bytes so that some amount of padding is added implicitly in order to make it a multiple of 16 bytes. However, after analyzing the source code of AES in wolfSSL API it was discovered that the AES algorithm in wolfSSL only accepts plaintext which is a multiple of 16 bytes else it completely rejects the entire plaintext. Hence, no padding is added implicitly by AES algorithm and thus padding oracle attack cannot work on AES algorithm of wolfSSL.

Therefore, we carried out our further research on 3-DES which is also a FIPS recognised block cipher algorithm and wolfSSL provides implicit padding support for this algorithm. Here, we started with an encryption and decryption program written in C language where a plaintext was given as user input to the encryption program which in turn returns a cipher text. Now, the first iteration of Padding Oracle Attack as described in Section II was carried out where 255 variants of cipher text were obtained by changing the last byte of second last block from 0x00 – 0xff. These 255 variants of cipher text were fed to the decryption binary loaded in GEM5 simulator and 430 attributes’ data was obtained corresponding to each variant of cipher text. Then, this data is merged such that for each attribute , values corresponding to each cipher text variant are placed under one file. In this manner 430 such files were formed which were then analysed statistically. However, it was realised that there exists no correlation between the measured attributes and the 255 variants of cipher text. We also carried out the second and third iteration of Padding Oracle Attack in order to see any difference but the results obtained were not conclusive. However, for the first iteration some of the results show that the last byte of the plaintext can be determined when the padding is 0x00.

Hence, we modified our attack and we switched to POODLE attack. In order to carry out the poodle attack we had to fulfil the two conditions of POODLE as described in Section 3.2.2.
Our attack strategy is described by the figure below:

![Diagram showing the wolfSSL approach](image)

**Figure 4.2: wolfSSL Approach**

To fulfil the two conditions of POODLE attack, we modified our plaintext such that it is a multiple of 8 bytes so that a complete block of padding of 8 bytes would be added at the end say Xn. Also, the first byte of the secret message in which we were interested to extract was kept as the last byte of one of the blocks of plaintext say Xi. This plaintext was given to the encryption program which in turn provides us with a cipher text. Let’s say the cipher text block corresponding to the plaintext block Xi and Xn is Yi and Yn respectively. After this in order to carry out the padding oracle attack, the cipher text block Yn was then replaced by the block Yi. This cipher text was then fed to decryption binary running under GEM5 and padding oracle attack was carried out as described in Section 3.2.2. After completing 255 iterations of padding oracle attack for first byte we obtained attributes’ data from GEM5 which was then analysed statistically and it was observed that approx 20 such attributes reveal the byte of the second last cipher text block when the padding is 0x00. With this value we can obtain the byte of the plaintext we are interested in as follows:

Let’s say the byte value revealed by the attributes is Cn-1.

Now,

\[
Pi = [D_k(C_n)] \text{xor} [C_{n-1}] \quad (1)
\]

where, \( D_k(C_i) \) is intermediary block called as Cn’ or Ci’

As the results showed

\[
[D_k(C_n)] \text{xor} [C_{n-1}] = 0x00
\]

But, as mentioned earlier Cn was replaced by Ci therefore, the intermediary block is Cn’ or Ci’

It implies,

\[
[D_k(C_i)] \text{xor} [C_{n-1}] = 0x00
\]

\[
D_k(C_i) = [C_{n-1}] \text{xor} [0x00]
\] (2)

Therefore, after combining (1) and (2)

\[
P_i = [C_{n-1}] \text{xor} [0x00] \text{xor} [C_{i-1}] \quad (3)
\]
As, the attacker has access to the complete cipher text therefore, using (3) determining the last byte of the plaintext block is easy.

This procedure was repeated in the similar manner iteratively as described in Section II in order to extract the other unknown bytes of the secret message. Remember, small modifications are required to be made in the plaintext in order to fulfil the conditions of POODLE to carry out the attack.

The results obtained are described in Section 5.
5. EXPERIMENTAL RESULTS

5.1. OpenSSL RESULTS

In order to carry out the research on OpenSSL we first wrote two scripts to perform the encryption & decryption using AES algorithm in CBC mode with 128 bit key and standard padding whose inbuilt support is provided by Perl's CPAN implementation.

Our research approach is described in Figure 4.1. A plaintext was first fed to the encryption program. We tried different values of plaintext having different lengths given as input to the encryption program. Let’s say our one of the plaintext values is as follows which is of 24 bytes:

Plaintext : zmx4wker02g6HELLOHOWSUqA

Here, as the plaintext was of 24 bytes which is not a multiple of AES block size of 16 bytes therefore, some kind of padding was added at the end implicitly. Here, the no. of bytes of padding added were \((16 - [24\%16] = 8)\). Therefore, the plaintext was changed to:

Plaintext : zmx4wker02g6HELLOHOWSUqAx08x08x08x08x08x08x08x08

The corresponding cipher text in hexadecimal obtained from the encryption program was as follows:

Cipher text :
52cb50912065ac52a55a15d599a16930de95342e176fe8f26e62a997052847fd

Which made two blocks of cipher text as follows:

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>52cb50912065ac52a55a15d599a16930</td>
<td>dc95342e176fe8f26e62a997052847fd</td>
</tr>
</tbody>
</table>

Now, a static binary of decryption program was built using App-StaticPerl and as described in Figure 4.1, static decryption binary was run under GEM5. In order to carry out the padding oracle attack for last byte of plaintext, 255 variants of cipher text were given as input to this static decryption binary.
Padding Oracle Attack - First Iteration

In the first iteration of padding oracle attack, 255 variants of cipher text were given as input to the decryption binary under GEM5 as follows:

![GEM5 simulator](128-bit AES decryption static binary)

![Figure 5.1: 255 Iterations of Padding Oracle Attack on OpenSSL](images)

Every run on GEM5 corresponding to each variant of cipher text provided us with some data related to different attributes of CPU architecture for which statistical analysis was done. The statistical analysis corresponding to every attribute and every variant of cipher text was then performed and it was observed that a correlation exists between the measured attributes’ data and the 255 variants of the cipher text where an anomaly could be seen corresponding to the correct value for around 20 such attributes which leaked sensitive information. Some of the graphs obtained on x86 architecture were as follows:

![Instruction bytes read from memory](images)

At last byte 0x39
Here, as we could see from the graph there was an anomaly corresponding to the last byte of cipher text at 0x39 i.e. $C_{i-1}[16] = 0x39$.

Using this value, we were able to extract the exact value of the last byte of the plaintext. In order to extract the plaintext byte,

According to CBC mode,

$$P_i = [D_k(C_i)] \oplus [C_{i-1}]$$

Or,

$$P_i = [C'_i] \oplus [C_{i-1}]$$

For last byte,

$$P_{[16]} = C'_{[16]} \oplus C_{i-1}[16] \quad (1)$$

Also, from figure 3.3

$$C'_i = [P_i] \oplus [C_{i-1}]$$

For last byte,

$$C'_{[16]} = P_{[16]} \oplus C_{i-1}[16] \quad (2)$$

Now, for First iteration of Padding Oracle Attack it is expected that,

$$P_{[16]} = 0x01 \quad (3)$$

And, from the statistical analysis we know that,

$$C_{i-1}[16] = 0x39 \quad (4)$$

Therefore, from (2), (3) and (4)

$$C'_{[16]} = 0x01 \oplus 0x39$$

$$C'_{[16]} = 0x38 \quad (5)$$

We also know that last byte of the original cipher text was, $C_{i-1}[16] = 0x30 \quad (6)$

Therefore, from (1), (5) and (6)

$$P_{[16]} = 0x38 \oplus 0x30$$

$$= 0x08$$

which is the original value of last byte of the plaintext

Thus, we were able to extract the last byte of the plaintext successfully.

**Padding Oracle Attack - Second Iteration**

The padding oracle was carried out in the similar manner as described above to extract the second last byte of the plaintext.

From (5) above, we knew that $C'_{[16]} = 0x38$ and for the second iteration the expected value of the last two bytes of plaintext are 0x02 i.e. $P_{[16]} = 0x02$ and $P_{[15]} = 0x02$.

We know that,

$$P_{[16]} = C'_{[16]} \oplus C_{i-1}[16]$$

Implies,

$$C_{i-1}[16] = P_{[16]} \oplus C'_{[16]}$$
Therefore, \[ C_{i-1}[16] = 0x02 \text{xor} 0x38 \]
\[ = 0x3a \]

Hence, we change the last byte of second last block as 0x3a and input 255 variants of cipher text to the decryption binary as follows:

![GEM5 simulator and 128-bit AES decryption static binary]

Figure 5.2 : 255 Iterations of Padding Oracle Attack on OpenSSL

The graph obtained for some of the attributes which show us an anomaly in x86 architecture are as follows:

![Integer ALU Accesses graph]

Here, an anomaly can be seen corresponding to the second last byte of cipher text at 0x63 i.e. \( C_{i-1}[15] = 0x63 \).

Using this value, we were able to extract the exact value of the second last byte of the plaintext as follows:

According to CBC mode,

\[ P_i = [D_k(C_i)] \text{xor} [C_{i-1}] \]
Or,
\[ P_i = [C'_i] \text{xor} [C_{i-1}] \]

For last byte,
\[ P_{i[15]} = C'_{i[15]} \text{xor} C_{i-1[15]} \]  \hspace{1cm} (1)

implies,
\[ C'_{i[15]} = P_{i[15]} \text{xor} C_{i-1[15]} \]  \hspace{1cm} (2)

Now, for Second iteration of Padding Oracle Attack it is expected that,
\[ P_{i[15]} = 0x02 \]  \hspace{1cm} (3)

And, from the statistical analysis we know that,
\[ C_{i-1[15]} = 0x63 \]  \hspace{1cm} (4)

Therefore, from (2), (3) and (4)
\[ C'_{i[15]} = 0x02 \text{xor} 0x63 \]
\[ C'_{i[15]} = 0x61 \]  \hspace{1cm} (5)

We also know that last byte of the original cipher text was, \( C_{i-1[15]} = 0x69 \)  \hspace{1cm} (6)

Therefore, from (1), (5) and (6)
\[ P_{i[15]} = 0x61 \text{xor} 0x69 \]
\[ P_{i[15]} = 0x08 \] \hspace{1cm} which is the original value of the second last byte of plaintext

Thus, we were able to extract the second last byte of the plaintext successfully.

In this manner, we can run padding oracle attack iteratively to extract all the bytes of the plaintext.

**Results on ARM Architecture**

The experiment was also carried out on ARM architecture and the results obtained in that case were similar to x86 architecture. Some of the graphs obtained for first and second iteration of Padding Oracle Attack are:

**For First Iteration**
For Second Iteration

However, it was determined that the running time on ARM was much more, typically in the same of 2.5 to 3.5 times than that on the x86 architecture. This suggests that architecture difference can take an overall extra time to run the code due to some factors — CPU speed, memory, thread scheduling, kernel, or any other unknown factors but the leakage of data on a side channel would stay the same and are not dependent on architectures.

5.2 wolfSSL Results

(a) In order to carry out the research on wolfSSL we first wrote two scripts to carry out the encryption & decryption using AES algorithm in CBC mode with 128 bit key using wolfSSL API’s.

Our research approach is described in Figure 4.2. A plaintext was first fed to the encryption program. We tried different values of plaintext having different lengths given as input to the encryption program. Let’s say our one of the plaintext values is as follows which is of 42 bytes:

**Plaintext**: POST/gp/redirect/indi.htmlCookie:token=wdH

Here, as the plaintext was of 42 bytes which is not a multiple of AES block size of 16 bytes therefore, some kind of padding was added at the end implicitly. Here, the no. of bytes of padding added were (16 – [42%16] = 6). Therefore, the plaintext was changed to:

**Plaintext**: POST/gp/redirect/indi.htmlCookie:token=wdHx06x06x06x06x06x06

The corresponding cipher text in hexadecimal obtained from the encryption program was as follows:

**Cipher text**: 
c5421b878ec2d36a052f45796ad1bdea11d501312a305f586369cd8e19136db095145ea93c3b647860c13228d8dc1e90c
Which made three blocks of cipher text as follows:

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c5421b878c2d36a052f457 96ad1bdea1</td>
<td>1d501312a305f586369cd8 e19136db09</td>
</tr>
<tr>
<td>5145ea93c3b647860c1322 8d8dc1e90c</td>
<td></td>
</tr>
</tbody>
</table>

Now, a static binary of decryption program was built and as described in figure 4.2, static decryption binary was run under GEM5. In order to carry out the padding oracle attack for last byte of plaintext, 255 variants of cipher text were given as input to this static decryption binary as follows:

```
00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000002
```

Figure 5.3: 255 Iterations of Padding Oracle Attack on wolfSSL

The graphs obtained in this case is as follows:

```
00000000000000000000000000000000
```

This graph did not give us any conclusive results and hence, we modified our attack.

(b) We modified the encryption and decryption codes such that they support 3DES algorithm in CBC mode. This time we modified our attack and carried out POODLE. According to Figure 4.2 we gave the plaintext to the encryption program which in
turn provided us with a cipher text. Let’s say one of the plaintext values were as follows which is of 48 bytes:

**Plaintext**: POST/gp/redirect/indi.htmlCookie:**token=wdHZJaxBe

This plaintext was of 48 bytes which is a multiple of 3DES block size of 8 bytes thus, a complete block of padding was added at the end of this plaintext which fulfills the first condition of POODLE attack as described in Section 3.2.2.

The plaintext will be taken by CBC Mode into blocks of 8 bytes as follows:

<table>
<thead>
<tr>
<th>POST/gp/</th>
<th>Redirect</th>
<th>/indi.ht</th>
<th>mCookie</th>
<th><strong>token=w</strong></th>
<th>dHZJaxBe</th>
<th>0x08 0x08 0x08 0x08 0x08 0x08 0x08 0x08</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>Block 2</td>
<td>Block 4</td>
<td>Block 5</td>
<td>Block 6</td>
<td>Block 7</td>
<td></td>
</tr>
</tbody>
</table>

Also, the length of the plaintext was kept in such a manner that secret byte in which we were interested to extract was kept as the last byte of one of the blocks. For instance, here the first byte of the secret token i.e. ‘w’ is kept as the last byte of the Block 5. In this manner the second condition of POODLE attack as described in Section 3.2.2 was also fulfilled.

The cipher text in hexadecimal given by the encryption program corresponding to the above plaintext was:

**Cipher text**: 5f06fc6cc5ff7b230fdcf40b1d0603ca0067df22ec6a8f3be0067df22ec6a8f3be5c48fe596ca889f54e07a3571ac7ad26b8d80ac480904bb075c3bf233b4d6298bf5816a3d3dca5fa7d5261e44b0ce4a545f137188de3a49446a30343a09e38a73b3e45bd96efbb

Here, the cipher text block in red say Ci corresponds to the plaintext block 5 which was the plaintext block having the unknown secret byte was kept as the last byte. The block in green say Cn corresponds to the plaintext block 7 where complete block was filled with some padding.

We then carried out the attack by replacing the last block of the cipher text (Cn) with a copy of the block that contains the cookie’s first unknown byte (Ci) as follows:

**Cipher text:** 5f06fc6cc5ff7b230fdcf40b1d0603ca0067df22ec6a8f3be0067df22ec6a8f3be5c48fe596ca889f54e07a3571ac7ad26b8d80ac480904bb075c3bf233b4d6298bf5816a3d3dca5fa7d5261e44b0ce4a545f137188de3a49446a30343a09e385c48fe596ca889f

Now, a static binary of decryption program was built in C language and as described in Figure 4.2, static decryption binary was run under GEM5. In order to carry out the POODLE attack, 255 variants of cipher text were given as input to this static decryption binary.
POODLE Attack - First Iteration

In the first iteration of POODLE attack, 255 variants of cipher text were given as input to the decryption binary under GEM5 as follows:

```
0000000000000000 5c48fe596cca889f
0000000000000001 5c48fe596cca889f
0000000000000002 5c48fe596cca889f
0000000000000003 5c48fe596cca889f
```

Figure 5.4: 255 Iterations of PODDLE Attack on wolfSSL

The entire cipher text was input to GEM5 but here, only the last two blocks of cipher text have been depicted as the modification was done only to these last two blocks in order to carry out the POODLE attack.

Every run on GEM5 corresponding to each variant of cipher text provided us with some data related to different attributes of CPU architecture for which statistical analysis was done. The statistical analysis corresponding to every attribute and every variant of cipher text was then performed and it was observed that a correlation exists between the measured attributes’ data and the 255 variants of the cipher text where an anomaly could be seen corresponding to the correct value for around 20 such attributes which leaked sensitive information. Some of the graphs obtained were as follows:

```
No. of Integer register Reads

<table>
<thead>
<tr>
<th>Last Byte</th>
<th>00.log</th>
<th>0a.log</th>
<th>14.log</th>
<th>1e.log</th>
<th>28.log</th>
<th>32.log</th>
<th>3c.log</th>
<th>46.log</th>
<th>50.log</th>
<th>5a.log</th>
<th>64.log</th>
<th>6e.log</th>
<th>78.log</th>
<th>82.log</th>
<th>8c.log</th>
<th>96.log</th>
<th>a0.log</th>
<th>a1.log</th>
<th>b4.log</th>
<th>be.log</th>
<th>c8.log</th>
<th>d2.log</th>
<th>dc.log</th>
<th>e6.log</th>
<th>f0.log</th>
<th>fa.log</th>
</tr>
</thead>
<tbody>
<tr>
<td>452160000</td>
<td>452180000</td>
<td>452200000</td>
<td>452220000</td>
<td>452240000</td>
<td>452260000</td>
<td>452280000</td>
<td>452300000</td>
<td>452320000</td>
<td>452340000</td>
<td>452360000</td>
<td>452380000</td>
<td>452400000</td>
<td>452420000</td>
<td>452440000</td>
<td>452460000</td>
<td>452480000</td>
<td>452500000</td>
<td>452520000</td>
<td>452540000</td>
<td>452560000</td>
<td>452580000</td>
<td>452600000</td>
<td>452620000</td>
<td>452640000</td>
<td>452660000</td>
<td>452680000</td>
</tr>
</tbody>
</table>
```

Integer Register Reads

Maximum value at byte = 0xc9

After analysis it was determined that the anomaly was shown by the graph at the point where the last byte value of plaintext becomes 0x00. For instance, here the highest value is shown by the graph at 0xc9 i.e. \( C_{n-1}[8] = 0xc9 \)

Using equation (3) of Section 1.2 we know that,

\[
P_i = [C_{n-1}] \text{ xor } [0x00] \text{ xor } [C_{i-1}]
\]

To determine the last byte of plaintext,

\[
P_{i}[8] = C_{n-1}[8] \text{ xor } [0x00] \text{ xor } C_{i-1}[8]
\]

Now, from the cipher text it can be determined that \( C_{i-1}[8] = 0xbe \) and from the graph it is analysed that \( C_{n-1}[8] = 0xc9 \).

Therefore,

\[
P_{i}[8] = [0xc9] \text{ xor } [0x00] \text{ xor } [0xbe] = 0x77
\]

which is the hex equivalent to ‘w’

Thus, in this manner we were able to extract the first byte of the plaintext successfully.

**POODLE Attack - Second Iteration**

In order to extract the second byte of the secret token, POODLE attack was again carried out but this time a small variation was done in the plaintext such that conditions of POODLE attack are fulfilled. This time the plaintext given to the encryption binary was as follows:

<table>
<thead>
<tr>
<th>POST/gp/</th>
<th>Redirect</th>
<th>/ind.htm</th>
<th>lCookie:</th>
<th><strong>token=wd</strong></th>
<th>HZJaxBey</th>
<th>0x08 0x08 0x08 0x08 0x08 0x08 0x08 0x08</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>Block 2</td>
<td>Block 3</td>
<td>Block 4</td>
<td>Block 5</td>
<td>Block 6</td>
<td>Block 7</td>
</tr>
</tbody>
</table>

Here, the next unknown byte of the secret token i.e. ‘d’ is kept as the last byte of one of the blocks.

The cipher text in hexadecimal given by the encryption program corresponding to the above plaintext was:

**Cipher text:**

5f06fc6cc5f7b230fdef40b1d0603ca0067df2ec6a8f3be082427bedd116e95866c75eb0953fe92be4e07a3571aac7ad26bf8d80ac480904bb075cbbf233bd6298bf5816a3d3dca5fa7d5261e44b0ce4a545f137188deb3a49446a30343a09e38a73b3e45bd96efbb

Here, the cipher text block in red say \( C_i \) corresponds to the plaintext block 5 which was the plaintext block having the unknown secret byte was kept as the last byte. The block in green say \( C_n \) corresponds to the plaintext block 7 where complete block was filled with some padding.
We then carried out the attack by replacing the last block of the cipher text (Cn) with a copy of the block that contains the cookie’s second unknown byte (Ci) as follows:

**Cipher text:**

```
5f06fc6cc5f7b230f0cf40b1d0603a0067df2ec6a8f3be082427bedd116e95866c75eb0953fe92b4e07a3571aac7ad26b8d80ac480904bb075cbf233b4d6298bf5816a3d3dca5fa7d5261e44b0ce4a545f137188deb3a49446a30343a09e38866c75eb0953fe92
```

The same experiment as described above to determine the last byte of the plaintext was performed where 255 variants of this cipher text were given as input to the static decryption binary. The statistical analysis of the attributes’ data obtained from GEM5 provided us with some attributes through which sensitive information was leaked. Some of the graphs obtained are as follows:

![Busy CPU Cycles](image)

Here, the value obtained for \( C_{n-1}[8] = 0xf1 \) which further helped in determining the unknown value of the plaintext i.e. ‘d’ in this case.

Thus, this experiment can be performed iteratively to determine the unknown bytes of any secret token

**Results on ARM Architecture**

Similar to the case of openSSL, wolfSSL library was also tested on x86 as well as ARM architecture. The results of first and second iteration were similar to x86 architecture as follows:
For First Iteration

**Integer Register Reads**

- Maximum value at byte = 0xe9

For Second Iteration

**Busy CPU Cycles**

- Maximum value at byte = 0xf1

Similar to x86, it was determined that the running time on ARM was much more, than that on the x86 architecture.

### 5.3 Results for different message and padding lengths

Different message lengths were experimented upon and there was no difference which was observed in the experiments. On analysis of the code for both the libraries and the theory of the side channel attacks which were tried in the research, we came to a conclusion that the only thing which matters for any change in time is the actual size of padding which stays upper bounded to a block size irrespective of the message length.

Also, we performed experiment on openSSL and wolfSSL for different padding lengths starting from 0x01 to the block size of the block cipher algorithm used. The typical run time and analysis time for getting results on one kind of padding length is around 14 hours for CyaSSL and around 10 hours for OpenSSL. Any possible changes which could have occurred in time for different padding block size was something that was not measurable in any significant value.

However, the anomaly could be seen in different side channel parameters for all types of padding lengths. Thus, it was concluded that side channels exist for every type of padding length. The only difference that could be seen was in the values of
different side parameters. For example, the no. of busy CPU cycles will be higher in the case when there are two bytes of padding as compared to the case when there is one byte of padding. Hence, padding length doesn’t make any significant difference in the anomalies observed in the behaviour of different side channel parameters.

6. DISCUSSION AND FUTURE WORK

The research carried out on OpenSSL and wolfSSL libraries provided us with some attributes related to CPU architecture which lead to leakage of sensitive information. As mentioned earlier GEM5 provides data related to around 430 such attributes so, we tried to understand the reason why only 20 of these provide us an anomaly on statistically analysing the attributes’ data against every variant of cipher text. We started exploring the source code of OpenSSL API. The snippet of the source code which leads to leakage of sensitive information is given below:

```perl
sub _standard_padding ($$$) {
  my ($b,$bs,$decrypt) = @_;  
  $b = length $b ? $b : '';  
  if ($decrypt eq 'd') {
    my $pad_length = unpack("C",substr($b,-1));

    # sanity check for implementations that don't pad correctly
    return $b unless $pad_length >= 0 && $pad_length <= $bs;
    my $pad_chars = unpack("C*",substr($b,-$pad_length));
    return $b if grep { $pad_length != -1 } @pad_chars;

    return substr($b,0,$bs-$pad_length);
  }
  my $pad = $bs - length($b) + $bs;
  return $b . pack("C*",($pad)x$pad);
}
```

According to this source code, valid padding check is being done by the lines 7-10. After the decryption of the incoming cipher text if the padding of the obtained plaintext is correct then the server proceeds further and completes the further processing else the plaintext is rejected as soon as incorrect padding is observed. This extra processing done by the server for the case when the padding is correct results into the attributes leaking information as described in section 4.1.

Similar attributes leaked the information in case of wolfSSL library as well. The source code snippet responsible for this leakage of data is given below:
According to this, the length of the padding given by variable 'padLen' is determined by line 7. After this the bytes equal to the length of the padding are removed from the end of the plaintext & rest of the bytes are copied to the memory (line 10).

As observed from the graphs the anomaly is shown at 0x00 which implied that the length of the padding is zero. Thus, in this case the number of bytes copied to the memory will be equal to the length of the plaintext i.e. the maximum number of bytes will be copied in this case and thus the maximum number of CPU cycles, memory accesses, register reads etc. will be done. The list of the attributes which leaked sensitive information for wolfSSL library was the same as OpenSSL as described in section 4.1.

Through this research we were able to determine approximately 20 attributes through which sensitive data can be leaked. Thus, any attacker who can sniff into the packets carrying sensitive information over the network can apply the padding oracle attack strategy to obtain the entire message sent over the network. Similarly, an attacker can take control of the request sent to the server. He can craft the request in such a manner so that the two conditions of POODLE attack are satisfied. After this he can carry out the POODLE attack strategy iteratively to obtain the secret message which he desire to know.

Different message lengths were experimented upon and there was no difference which was observed in the experiments. On analysis of the code for both the libraries and the theory of the side channel attacks which were tried in the research, we came to a conclusion that the only thing which matters for any change in time is the actual size of padding which stays upper bounded to a block size irrespective of the message.
length. The typical run time to launch these attacks and analysis time taken for getting the results on one kind of padding length is approximately 10 hours for OpenSSL library and 14 hours for wolfSSL library. Any possible changes which could have occurred in time for different padding block size was not measurable in any significant value.

We carried out this research in a simulated environment on GEM5 simulator. GEM5 being a CPU simulator has its own performance bottlenecks and thus, it may not be possible to decrypt large amount of data in real life scenario. However, in future if some tool which can give faster results gets discovered then perhaps these attacks can work faster.

In our experiment we can’t say anything about whether the decryption thread is the only responsible factor for the different number of CPU instructions. There can be a possibility that OpenSSL and wolfSSL are multi-threaded libraries which may effect the results.

We have used one kind of API for encrypting and decrypting data in OpenSSL and wolfSSL. However, different API calls exist in both the libraries to encrypt and decrypt any data using one kind of block cipher algorithm. Other APIs can also be explored in future to determine the effect of these attacks on them. In addition to this, many operating systems such as windows, Macintosh etc. use their own implementations to support SSL/TLS protocols. Such popular SSL/TLS cryptographic libraries can be a potential area to explore in future. These cryptographic libraries can be explored to determine the extent up to which these libraries are vulnerable to the attacks as we described above and whether the attributes revealing the sensitive information remain the same or there is addition of any new attributes.

In addition to this, some important real life applications using OpenSSL and wolfSSL making impact on large number of users can also be studied to identify what information can be derived from such applications and up to what extent.
7. CONCLUSION

Through this research we have discovered new side channels related to CPU architecture in the most widely used SSL/TLS libraries viz. OpenSSL and wolfSSL. These libraries have been exposed to various vulnerabilities which results in leakage of sensitive information sent over the network. Many side channel attacks have left these libraries vulnerable in the recent past but these attacks have taken advantage of some of the observable attributes of an encrypted online communication such as packet size and timing information. Many such side channel attacks have come into existence and it is realised that these side channel attacks keep on reviving very often. However, none of the research till now have tried to explore the leakage of data through side channels based on CPU architecture such as CPU cycle information, memory accesses etc. Through this research we have tried to explore such side channels on the various SSL/TLS implementations. Though, it has been observed that these libraries have been fixed against the previous side channel attacks but it is observed that these libraries are still vulnerable to CPU architecture based side channel attacks as described above. Thus, all the real world applications which have been patched against side channel attacks such as BEAST, CRIME, BREACH, POODLE etc. are still vulnerable to newer side channel attacks like ours. The current research has been carried out in a simulated environment using the GEM5 simulator but in future we would like to try it on real or virtualised environment which will not have performance bottlenecks like GEM5.
REFERENCES

2. Skorobogatov, Sergei. "Side-channel attacks."
16. Gluck, Yoel, Neal Harris, and Angelo Prado. "BREACH: reviving the CRIME attack."
25. Sarkar, Fitzgerald."Attacks on SSL"