

On the Practical Exploitability of Dual EC in TLS Implementations

Stephen Checkoway^{*}, Matthew Fredrikson[†], Ruben Niederhagen[‡]

Matthew Green^{*}, Tanja Lange[‡], Thomas Ristenpart[†]

Daniel J. Bernstein^{‡,§}, Jake Maskiewicz[¶], and Hovav Shacham[¶]

^{*}Johns Hopkins University, [†]University of Wisconsin, [‡]Technische Universiteit Eindhoven,
[§]University of Illinois at Chicago, [¶]UC San Diego

Abstract

This paper analyzes the actual cost of attacking TLS implementations that use NIST’s Dual EC pseudorandom number generator, assuming that the attacker generated the constants used in Dual EC. It has been known for several years that an attacker generating these constants and seeing a long enough stretch of Dual EC output bits can predict all future outputs; but TLS does not naturally provide a long enough stretch of output bits, and the cost of an attack turns out to depend heavily on choices made in implementing the RNG and on choices made in implementing other parts of TLS.

Specifically, this paper investigates OpenSSL-FIPS, Windows’ SChannel, and the C/C++ and Java versions of the RSA BSAFE library. This paper shows that Dual EC exploitability is fragile, and in particular is stopped by an outright bug in the certified Dual EC implementation in OpenSSL. On the other hand, this paper also shows that Dual EC exploitability benefits from two obscure TLS options, one of which is implemented in BSAFE; from a modification made to the Dual EC standard in 2007; and from several attack optimizations introduced here. The paper’s attacks are implemented; benchmarked; tested against libraries modified to use new Dual EC constants; and verified to successfully recover TLS plaintext.

1 Introduction

On September 5, 2013, the New York Times [18], the Guardian [2] and ProPublica [12] reported the existence of a secret National Security Agency SIGINT Enabling Project with the mission to “actively [engage] the US and foreign IT industries to covertly influence and/or overtly leverage their commercial products’ designs.” The revealed source documents describe a US \$250 million/year program designed to “make [systems] exploitable through SIGINT collection” by inserting vulnerabilities, collecting target network data, and influencing policies, standards and specifications for commercial public key technologies. Named targets include protocols for “TLS/SSL, https (e.g. webmail), SSH, encrypted chat, VPNs and encrypted VOIP.”

The documents also make specific reference to a set of pseudorandom number generator (PRNG) algo-

rithms adopted as part of the National Institute of Standards and Technology (NIST) Special Publication 800-90 [17] in 2006, and also standardized as part of ISO 18031 [11]. These standards include an algorithm called the Dual Elliptic Curve Deterministic Random Bit Generator (Dual EC). As a result of these revelations, NIST reopened the public comment period for SP 800-90.

Known weaknesses in Dual EC. Long before 2013, Dual EC had been identified by the security community as biased [22, 24], extremely slow, and backdoorable.

SP 800-90 had already noted that “elliptic curve arithmetic” makes Dual EC generate “pseudorandom bits more slowly than the other DRBG mechanisms in this Recommendation” [17, p. 177] but had claimed that the Dual EC design “allows for certain performance-enhancing possibilities.” In fact, Dual EC with all known optimizations is two orders of magnitude slower than the other PRNGs, because it uses scalar multiplications on an elliptic curve where the other PRNGs use a hash function or cipher call.

The back door is a less obvious issue, first brought to public attention by Shumow and Ferguson [23] in 2007. What Shumow and Ferguson showed was that an attacker specifying Dual EC, and inspecting some Dual EC output bits from an unknown seed, had the power to predict all subsequent output bits.

Specifically, the description of Dual EC standardizes three parameter sets, each specifying an elliptic curve E over a finite field \mathbf{F}_p , together with points P and Q on E . The back door is knowledge of $d = \log_Q P$, the discrete logarithm of P to the base Q ; an attacker creating P and Q can be assumed to know d . Shumow and Ferguson showed that knowledge of d , together with about $\log_2 p$ consecutive output bits,¹ makes it feasible to predict all subsequent Dual EC output.

Shumow and Ferguson suggested as countermeasures to vary P and Q and to reduce the number of bits output per iteration of the PRNG. However, SP 800-90 requires a particular number of bits per iteration, and states that the standard P and Q “shall be used in applications requiring certification under FIPS 140-2”; this stops use of alternative points in certified implementations.

¹256 bits were sufficient in all their P-256 experiments.

Table 1: Summary of our results for Dual EC using NIST P-256.

Library	Default PRNG	Cache Output	Ext. Random	Bytes per Session	Adin Entropy	Attack Complexity	Time (minutes)
BSAFE-C v1.1	✓	✓	✓ [†]	31–60	—	$30 \cdot 2^{15}(C_v + C_f)$	0.04
BSAFE-Java v1.1	✓		✓ [†]	28	—	$2^{31}(C_v + 5C_f)$	63.96
SChannel I [‡]				28	—	$2^{31}(C_v + 4C_f)$	62.97
SChannel II [‡]				30	—	$2^{33}(C_v + C_f) + 2^{17}(5C_f)$	182.64
OpenSSL-fixed I [*]				32	2^{20}	$2^{15}(C_v + 3C_f) + 2^{20}(2C_f)$	0.02
OpenSSL-fixed III ^{**}				32	2^{35+k}	$2^{15}(C_v + 3C_f) + 2^{35+k}(2C_f)$	$2^k \cdot 83.32$

* Assuming process ID and counter known. ** Assuming 15 bits of entropy in process ID, maximum counter of 2^k . See Section 4.3.
[†] With a library-compile-time flag. [‡] Versions tested: Windows 7 64-bit Service Pack 1 and Windows Server 2010 R2.

The entries in the table are for normal TLS connections. In particular, we exclude all forms of session resumption. A ✓ in the Default PRNG column indicates whether Dual EC is the default PRNG used by the library. A ✓ in the Cache Output column indicates that the unused Dual EC output is cached for use in a subsequent call. A ✓ in the Extended Random column indicates that the proposed TLS extension Extended Random [20] is supported in some configuration of BSAFE. Reported attack times do not rely on use of extended random. Bytes per Session indicates how many contiguous, useful output bytes from Dual EC a TLS server’s handshake message reveals. For SChannel II this is an average value of useful bits, see Section 4.2. Adin Entropy indicates how many bits of additional input are added to each Dual EC generate call. The Attack Complexity is the computational cost in terms of a scalar multiplication with a variable base point, C_v , and a fixed base point, C_f in the worst case. With our optimizations (see Section 5), C_f is roughly two orders of magnitude faster than C_v ; the exact speedup depends on context. The Time column gives our measured worst case time for the attack on a four-node, quad-socket AMD Opteron 6276 cluster; the time for OpenSSL-fixed III is measured using $k = 0$.

Risk assessment for this back door depends on the probability that the creator of P and Q is an attacker. Shumow and Ferguson wrote “WHAT WE ARE NOT SAYING: NIST intentionally put a back door in this PRNG”; but the September 2013 news indicates that NSA may have deliberately engineered Dual EC with a back door. Our concern in this paper is not with this probability assessment, but rather with impact assessment, especially for the use of Dual EC in TLS.

Use of Dual EC in products. Despite the known weaknesses in Dual EC, several vendors have implemented Dual EC in their products [16]. For example, OpenSSL-FIPS v2 and Microsoft’s SChannel include Dual EC, and RSA’s crypto libraries use Dual EC by default. RSA Executive Chairman Art Coviello, responding to news that NSA had paid RSA to use Dual EC [14], stated during the opening speech of RSA Conference 2014: “Given that RSA’s market for encryption tools was increasingly limited to the US Federal government and organizations selling applications to the federal government, use of this algorithm as a default in many of our toolkits allowed us to meet government certification requirements” [4].

Practical attacks on TLS using Dual EC. This paper studies to which extent *deployed* cryptographic systems that use Dual EC are vulnerable to the back door, assuming that an attacker knows $d = \log_Q P$. Specifically, we perform a case study of Dual EC use in TLS, arguably the most important potential target for attacks. The basic attack described by Shumow and Ferguson [23] (and independently, quietly, by Brown and Vanstone in a patent application [3]) turns out to be highly oversimplified: it

does not consider critical limitations and variations in the amount of PRNG output actually exposed in TLS, additional inputs to the PRNG, PRNG reseeding, alignment of PRNG outputs, and outright bugs in Dual EC implementations.

We present not just a theoretical evaluation of TLS vulnerability but an in-depth analysis of Dual EC in four recent implementations of TLS: RSA BSAFE Share for C/C++ (henceforth BSAFE-C), RSA BSAFE Share for Java (henceforth BSAFE-Java), Windows SChannel, and OpenSSL. To experimentally verify the actual performance of our attacks, we replace the NIST-specified constants with ones we generate; for BSAFE and Windows this required extensive reverse-engineering of binaries to find not just P and Q but many implementation-specific constants and run-time test vectors derived from P and Q (see Section 4.4). Our major findings are as follows:

- The BSAFE implementations of TLS makes the Dual EC back door particularly easy to exploit in two ways. The Java version of BSAFE includes fingerprints in connections, making them easy to identify. The C version of BSAFE allows a drastic speedup in the attack by broadcasting longer strings of random bits than one would at first imagine to be possible given the TLS standards.
- Windows SChannel does not implement the current Dual EC standard: it omits an important computation. We show that this does not prevent attacks; in fact, it allows slightly faster attacks.
- We discovered in OpenSSL a previously unknown bug that prevented the library from running when

Dual EC is enabled. It is still conceivable that someone is using Dual EC in OpenSSL, since the bug has an obvious and very easy fix, so we applied this fix and evaluated the resulting version of OpenSSL, which we call OpenSSL-fixed. OpenSSL-fixed turns out to use additional inputs in a way that under some circumstances makes attacks significantly more expensive than for the other libraries.

When a TLS server uses DSA or ECDSA to sign its DH/ECDH public key, a single known nonce reveals the long-lived signing key which enables future active attacks. Our attacks reveal the inner state of Dual EC which generates the nonces and we have successfully recovered the long-term signing keys.

We also perform a brief measurement study of the IPv4 address space to assess the prevalence of these libraries on the Internet.

We summarize our results in Table 1. The BSAFE-C attack is practically instantaneous, even on an old laptop. The BSAFE-Java and SChannel attacks require more processing power to recover missing bits of Dual EC output. The OpenSSL-fixed attack cost depends fundamentally on how much information on the additional input is available. All of these attacks are practical for a motivated attacker, even when the attacks are repeated against a large number of targets.

2 Dual EC attack theory

Review of Dual EC. We focus on Dual EC using the NIST P-256 elliptic curve. For the curve equation and the standardized base points P and Q see the appendix. The state of Dual EC is a 32-byte string s , which the user initializes as a secret random seed. The user then calls Dual EC any number of times; each call implicitly reads and writes the state, optionally reads *additional input* from the user, and produces any number of random bytes requested by the user.

Internally, each call works as follows. Additional input, if provided, is hashed and xored into the state. The state is then updated as follows: compute sP and then overwrite s with the x -coordinate $x(sP)$. A 30-byte block of output is then generated as follows: compute sQ , take the x -coordinate $x(sQ)$, and discard the most significant 2 bytes. The resulting 30 bytes are output. If more random bytes were requested, the state is updated again and another 30-byte block of output is generated; this repeats until enough blocks have been generated. Any excess bytes in the final block are discarded. The state is updated one final time in preparation for the next call, and the generator returns the requested number of bytes.

Review of the basic attack. The attack from Shumow and Ferguson works as follows. The attacker is assumed

to control the initial standardization of P . The attacker takes advantage of this by generating a random secret integer d and generating P as dQ . Alternatively, if the attacker controls the initial standardization of Q rather than P , the attacker generates Q as $(1/d)P$. Either way $P = dQ$, with d secretly known to the attacker.

The idea of the attack is to reconstruct sQ from an output block (see the next paragraph) and then multiply by d , obtaining dsQ , i.e., sP . The x -coordinate $x(sP)$ is the user's next PRNG state. The attacker then computes all subsequent outputs the same way that the user does.

Recall that an output block reveals 30 out of the 32 bytes of the x -coordinate of sQ . The attacker tries all possibilities for the 2 missing bytes, obtaining 2^{16} possibilities for the x -coordinate, and then for each x -coordinate uses the curve equation to reconstruct at most 2 possibilities for the y -coordinate, for a total of at most 2^{17} possibilities for sQ . About half of the x -coordinates will not have any corresponding y -coordinate, and the other half will produce two points (x, y) and $(x, -y)$ that behave identically for the attack, because $x(s(x, y)) = x(s(x, -y))$, so the attacker keeps only one of those points and ends up with about 2^{15} possibilities for sQ . For each of these possibilities, the attacker computes the corresponding possibility for $dsQ = sP$ and for the next Dual EC output. The attacker pinpoints the correct guess by checking the next actual Dual EC output.

Attacks with additional input. Shumow and Ferguson did not analyze the case where a user provides additional input to a Dual EC call. We point out that the analysis of this case depends heavily on whether one considers Dual EC in the June 2006 version of SP 800-90A, which we call Dual EC 2006, or Dual EC in the March 2007 version of SP 800-90A, which we call Dual EC 2007.

Additional input is not used after the beginning of a call and therefore does not stop the attacker from using the first 30 bytes output by a call to predict subsequent bytes output by the same call. The remaining question is whether the attacker can predict the first 30 bytes from this call given the last 30 bytes from the previous call.

If the additional input has enough entropy unknown to the attacker then the answer is no: the first 30 bytes are unpredictable. However, in the applications that we have studied, additional input is either nonexistent or guessable. This is where Dual EC 2006 and Dual EC 2007 produce different answers.

What we have described so far is Dual EC 2007. The previous call outputs most of sQ and produces $s' = x(sP)$ as the new state. This call updates s' to $s'' = x((s' \oplus H(\text{adin}))P)$, where H is a hash function, and then outputs most of $x(s''Q)$. Given sQ the attacker computes $dsQ = sP$, computes s' , and then for each possible adin computes s'' and $s''Q$.

The 2006 version of Dual EC differs slightly from the 2007 version: Dual EC 2006 is missing the final step which updates the seed s at the end of each call. The previous call outputs most of sQ but leaves s untouched. If there were no additional input, then this call would update s to $s' = x(sP)$ and output most of $x(s'Q)$. Given sQ , the attacker computes $sP = dsQ$, s' , and $s'Q$. However, with additional input, the state s is updated to $s' = x((s \oplus H(\text{adin}))P)$ and then most of $x(s'Q)$ is output. Given sQ , the attacker can compute $sP = dsQ$ and $x(sP)$ as before but there is no obvious way to obtain $(s \oplus H(\text{adin}))P$ from sP , even when adin is known.

Although, the addition of the final update step in Dual EC 2007 has the effect of making the basic attack possible in the rare case where the attacker has enough consecutive output bytes to make the attack feasible — but at most 30 bytes — from a single call to the generator, it plays no role in any of our results. In some cases, it actually makes our attacks slightly slower due to the need to compute an extra state update.

Open questions. This theoretical analysis of Dual EC exploitability leaves open several obvious questions regarding the practical exploitability of Dual EC by network attackers who know the secret d . Do implementations of cryptographic protocols such as TLS actually expose enough Dual EC output to carry out the basic attack? How expensive are the computations required to compensate for missing output, and can these computations be made less expensive? Is additional input actually used, and if so is it hard to guess? Are certified implementations of Dual EC in fact implementing Dual EC 2006, Dual EC 2007, or something different?

The answers turn out to include several surprises, and in particular to rely on several implementation details and protocol details that have not been previously observed to be related to Dual EC. Our analysis strongly suggests that, from an attacker’s perspective, backdooring a PRNG should be combined not merely with influencing implementations to use the PRNG but also with influencing other details that secretly improve the exploitability of the PRNG. This paper does not attempt to determine whether this is what happened with Dual EC, and does not explore the difficult topic of defending against such attacks, beyond the obvious advice of not using Dual EC.

3 Attack target: TLS

TLS is the most widely used protocol for securing Internet communications [5]. TLS consists of several sub-protocols, including a *record* protocol and *handshake* protocol. The handshake protocol is most relevant to the attacks discussed in this paper, so for simplicity we will describe only the relevant aspects of the handshake sub-protocol used in TLS version 1.2; further details are available in the RFC [5].

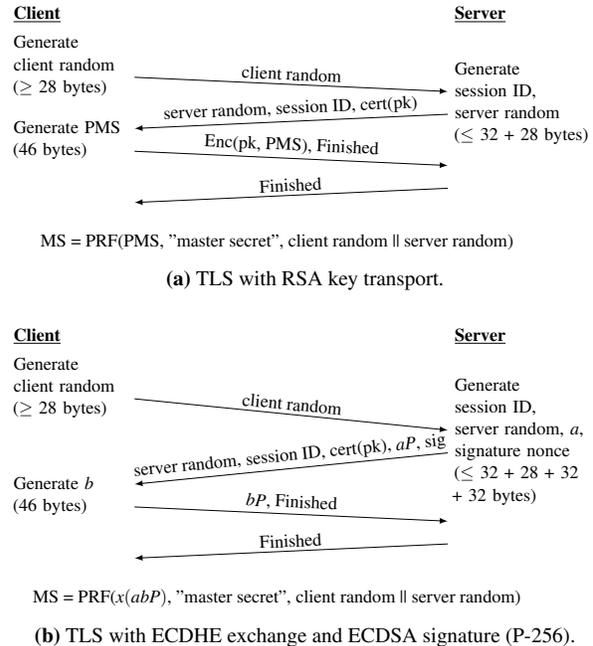


Figure 1: Simplified view of TLS handshakes.

The handshake sub-protocol produces a fresh set of session keys with which application-layer data is encrypted and authenticated using the record protocol. Figure 1 depicts (simplified versions of) the two main handshakes for TLS: RSA key transport and ephemeral Diffie-Hellman key exchange (DHE). Ephemeral DHE uses either elliptic curve groups (ECDHE) or another suitable group. In either handshake, the client initiates a connection by sending a ClientHello message that includes a client nonce and a list of supported cipher suites. The server replies with a server nonce, a session ID, a certificate, an ephemeral DHE public key (for DHE), and a signature over the random nonces and public key. Signatures are either RSA or DSA. The specification mandates 32-byte client and server nonces, each consisting of 28 random bytes and a 4-byte timestamp. The construction of a session ID is arbitrary (i.e., up to the server implementation), though as we will see many implementations use the same PRNG that generates the ServerHello nonce and other cryptographic secrets. The client’s next flow includes a client ephemeral DH public key (for DHE) or an RSA PKCS #1.5 encryption of a premaster secret (for RSA key transport). The premaster secret consists of either a 2-byte version number followed by a 46 byte random value (for RSA key transport), or the DHE secret defined by the DH public keys. Session keys are derived from the premaster secret and other values sent in the clear during the handshake, so learning the premaster secret is sufficient to break all subsequent encryption for a given session.

TLS extensions. There are many extensions to TLS, but we draw attention to three particular proposed — but not standardized — extensions. Each of these extensions has the side effect of removing the most obvious difficulty in exploiting Dual EC in TLS, namely the limited amount of randomness broadcast to the attacker. One might guess that these extensions make P-256 less expensive to exploit in TLS by a factor of 65,536 (and make P-384 and P-521 feasible to exploit), if they are actually implemented; our analysis in Section 4.1 shows that one of these extensions is in fact implemented in BSAFE, although the actual effect on exploitability is more complicated. Neither of these extensions has been previously described in connection with Dual EC.

One proposed extension, authored by Rescorla and Salter [20] in 2008, supports “extended random” client and server nonces. This extension is negotiable using the normal ClientHello extension mechanism, and includes up to $2^{16} - 1$ bytes of data from a suitable PRNG. The server replies with its own extended random that must be of the same length as the client’s extended random. The document states that this extension was requested by the United States Department of Defense with the claim that nonces “should be at least twice as long as the security level” (e.g., 256-bit nonces for 128-bit security). The other extension, “Opaque PRF” proposed by the same authors [19] in 2006, is nearly identical to “extended random” but does not require the data to be random. A third proposed extension, “additional random” by Hoffman [10] in 2010 is essentially the same as “extended random.”

None of the three proposed extensions was ever adopted as a standard by the IETF and the “Internet-Drafts” describing them have all since expired.

Attack goals. We assume that the adversary’s goal is to decrypt TLS packets to learn confidential material, or to steal long-lived secret keys. In the second case the secret keys need not be generated with Dual EC. We consider both small-scale *targeted* attacks and larger-scale *dragnet surveillance* attacks across broad swaths of the Internet.

Attack resources. Most of the attacks that we analyze are purely passive, relying solely on interception of TLS traffic sent through the network by the client and by the server. Usually seeing only one direction of TLS traffic is enough, and the attack can be mounted long after the fact using recorded connections. Occasionally an active attack is more powerful: for example, the range of μ secs in Section 4.3 becomes narrower if the attacker uses carefully timed connections to pin down the server’s clock.

The attacker is assumed to know the Dual EC back door d with $P = dQ$. All of the attacks rely on the client or server using Dual EC, but this is not an assumption; rather,

it is something that we evaluate, by reverse-engineering several TLS implementations and also experimentally assessing the deployment of those implementations.

Our measurements of attack cost assume that the attacker knows the TLS software in use; otherwise the attacker has to try several of the attacks, increasing cost somewhat. See Section 6 for fingerprinting mechanisms.

The computer power required for attacking *one* Dual EC instance is very small by cryptanalytic standards: our optimized attacks (see Section 5) typically consume between \$0.00001 and \$1 of electricity, depending on the TLS implementation being attacked. (The exception to “typical” is OpenSSL; see Sections 4.3 and 5.2.) However, presumably the attacker’s actual goal is to repeat the attack *many* times, especially in the dragnet-surveillance scenario. Our measurements allow straightforward extrapolations of the computer resources required for large-scale attacks.

4 Exploiting Dual EC in implementations

To attack each of the implementations discussed below, the attacker follows three basic steps: (1) recover Dual EC state from the session ID and/or server random fields in the TLS handshake; (2) compute the DHE or ECDHE shared secret which enables computing the 48-byte “master secret” from which all session keys are derived; and optionally (3) recover the long-lived DSA or ECDSA signing key used to sign the server’s DHE or ECDHE public key.

Step (1) is an application of the basic attack which combines information exchanged in the handshake protocol messages to determine the correct Dual EC state from candidate states. Step (2) requires generating the DHE or ECDHE secret key by following the exact generation process used by the TLS implementation. Like Step (2), Step (3) duplicates the implementation’s process for generating the nonce used in the signature of the public key. From the nonce, the signature, and the public key, it is straightforward to recover the signing key.

It is important to note that when a server uses DSA or ECDSA signatures, a *single* broken connection by a passive adversary is sufficient to recover the long-lived signing key which is used to authenticate the server’s (EC)DHE public key. In contrast to RSA long-lived keys, recovering a server’s (EC)DSA signing key does not enable future passive eavesdropping; it does allow impersonation of the server under active attack.

4.1 RSA BSAFE

Description. RSA’s BSAFE family of libraries come in four flavors: Share for Java, Share for C and C++, Micro Edition Suite, and Crypto-J/SSL-J. We examined Share for Java and Share for C and C++. Although the two ver-

sions share a somewhat similar API, the implementation details differ, leading to different attacks.

The BSAFE family of libraries contains a number of options which can be configured at runtime. In order to avoid a combinatorial explosion in the number of configurations to test and attack, we focus our attention on the default configurations and the most secure cipher suites that lead to the use of the P-256 curve in Dual EC and, where applicable, ECDHE and ECDSA.²

Both BSAFE libraries we examined support both prediction resistance whereby the generator is reseeded on each call and output caching so that unused bytes from one call to generate can be used in subsequent calls rather than discarded. By default, neither option is enabled.

BSAFE-C. We examined the RSA BSAFE Share for C and C++ library (BSAFE-C) version 1.1 for Microsoft Windows. The library consists of two libraries, `sharecrypto.lib` which implements the core cryptographic primitives, including Dual EC and `sharesltpki.lib` which implements TLS. Unlike the Micro Edition Suite, BSAFE-C is distributed only as static libraries with associated header files. This necessitated a minor reverse engineering effort to discover how BSAFE-C uses Dual EC in its TLS implementation.

Unlike the other TLS implementations we examined, BSAFE-C v. 1.1 does not support TLS 1.2. As a result, it does not support elliptic curve cryptography for either key exchange or digital signatures. By default, the preferred cipher suites are `TLS_DHE_DSS_WITH_AES_128_CBC_SHA` and `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` so we focused our efforts on these two.

A TLS server implemented using BSAFE-C generates several pseudorandom values used during the TLS handshake to establish session keys. In order, it generates (1) a 32-byte session identifier, (2) 28 bytes for the server random, (3) a 20-byte ephemeral Diffie–Hellman (DH) secret key, and, when using DSA, (4) a 20-byte nonce. 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 server’s first flight of messages to the client during the handshake.

Although BSAFE-C’s Dual EC interface does not cache unused output bytes by default, a separate, internal interface to produce pseudorandom values wraps Dual EC and provides its own layer of caching by only requesting multiples of 30 bytes from the Dual EC interface. This internal interface is used by all of the higher-level functionality, such as generating a DH secret key and a DSA nonce. Due to a quirk of the implementation, if a request to generate n bytes of output cannot be satisfied com-

pletely from the cached bytes, $\lfloor (n + 29)/30 \rfloor \cdot 30$ bytes are generated in a single call to Dual EC, even if most of the n bytes will be taken from the cached bytes.

Caching output bytes means that when a new TLS session is started, an attacker who has not seen all prior connections has no way of knowing if the first value generated by the server—the session id—begins with a full output block or if it contains bytes cached from a previous call to Dual EC. However, due to the use of the requested number of bytes rather than the number of remaining bytes after pulling from the cache, the concatenation of the 32-byte session ID and the 28 pseudorandom bytes in the server random always contains a full 30-byte output block and between one and 30 bytes of a subsequent block where both blocks are generated in the call to Dual EC for 60 bytes made while generating the session ID.

A passive network attacker can easily recover the session keys and the server’s long-lived DSA secret key used to sign the ephemeral DH parameters and public key. The attacker uses the publicly exchanged values in the connection through the ClientKeyExchange handshake message. At this point, the attacker knows, the session ID, the client and server randoms, the DH parameters and client and server public keys, and the signature. This contains everything needed for the attack. The session keys are computed from the public values and the DH shared secret.

To recover the inner state of Dual EC a long string of consecutive output bytes is required. First, the session ID and the pseudorandom 28 bytes of the server random are concatenated into a 60-byte value B . Since up to 29 bytes of B can come from a previous call to Dual EC, one of $B[0..29], B[1..30], \dots, B[29..58]$ must be a full output block. The basic attack is run on each in turn until the Dual EC state for the next output block is recovered. The attacker knows that the correct state has been found by (1) generating the next output block and comparing the corresponding bytes with the remaining bytes in B and then (2) generating more bytes as needed to produce a DH secret key and comparing the corresponding public key to the server’s public key. If the public keys agree, then the DH shared secret can be computed hence the session keys can be computed.

Once the session keys for a single session have been recovered, more bytes can be generated to produce the DSA nonce. The DSA secret key a can be computed from the nonce k , public key (p, q, g, y) , and the signature (R, S) of the message m as $a = R^{-1} \cdot (S \cdot k - H(m)) \bmod q$.

Recovering the Dual EC internal state requires performing approximately $30 \cdot 2^{15}$ scalar multiplications with a variable base point and an equal number with the fixed point Q , in the worst case. The total attack has a cost of $30 \cdot 2^{15}(C_v + C_f)$ where C_v (resp. C_f) is the cost of performing a single scalar multiplication with a variable

²Share for Java additionally supports P-384 and P-521 for Dual EC, ECDHE, and ECDSA.

(resp. fixed) base point. To generate the DH key, between $3C_f$ and $5C_f$ are needed to produce enough Dual EC output bytes; $1C_f$ is needed to compute server’s DH public key; and, finally $1C_v$ is needed to compute the shared DH secret (using the client’s DH input). Finally, to generate the nonce, at most one more Dual EC output is needed, using $3C_f$.

BSAFE-Java. We examined the RSA BSAFE Share for Java library version 1.1 (BSAFE-Java) and focused on connections using the `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` cipher suite.

Unlike BSAFE-C, the output from Dual EC is not cached so each generated output value is aligned with a block of generator output. Unfortunately (for the attacker), the session ID value produced by the server is not a 32-byte pseudorandom value. Instead, the attacker is forced to rely on the server random.

The values generated by Dual EC are, in order, (1) 28 bytes for server random; (2) 32 bytes for an ECDHE secret key; and (3) 32 bytes for an ECDSA nonce.

As before, a passive network attacker waits until she sees the ClientKeyExchange handshake message. At this point she has all of the information she needs to mount the following, simple attack. The 28 bytes from the server random are treated as bytes 2 through 29 of the 32-byte x-coordinate. She then mounts the basic attack by guessing the remaining most significant 16-bits and least significant 16-bits of the x-coordinate. A guess is checked by generating a 32-byte ECDH secret key, computing the corresponding public key, and comparing to the server’s public key.

Once a match is found, the inner state of Dual EC is known. The session keys can be derived from the ECDH shared secret and the other values sent in the clear. Similarly, the ECDSA nonce can be found by generating another 32-byte value. As with the non-elliptic-curve DSA, the server’s private key can be recovered from the nonce and the signature.

In the worst case, recovering the generator state requires approximately 2^{31} scalar multiplications with a variable base point and five times that number with a fixed base point to generate candidate ECDH secret keys and corresponding public keys. In total, the attack takes $2^{31}(C_v + 5C_f)$ work.

BSAFE connection watermarks and extended random. The documentation for BSAFE-C and BSAFE-Java indicate that they support connection watermarking and the TLS extended random extension described in Section 3.

In our experiments, BSAFE-Java has watermarks enabled by default. The watermark works by setting the first 20 bytes of the session ID to be the first 20 bytes of the server random and the last 12 bytes

are set to the string “`RSA_SSLJ_`.” This watermark can only be disabled by setting the property `com.rsa.ssl.server.watermark=disabled` in the Java security properties file [21].

From reverse engineering the BSAFE-Java share-Crypto.jar library, we determined that it contained code to support or require the proposed TLS extended random extension; however, this code was disabled by means of a single static final boolean variable. We surmise that this code is not “dead” in the traditional sense, but rather the value of the variable can be changed to produce versions of the library with different features.

By changing the value of this variable, we were able to verify that the extended random extension is supported by the server.³ When enabled and an extended random extension is received from the client, the server generates an equal length extended random response consisting of bytes generated by Dual EC concatenated with the same 12-byte watermark. The client extended random is 32-bytes by default. Interestingly, the 28 bytes for the server random and the Dual EC generated bytes for the extended random are generated together in a single call to Dual EC. As a consequence, any BSAFE-Java server which supports extended random exposes a sufficient quantity of contiguous output bytes to enable quick recovery of the session keys. There does not appear to be a mechanism for disabling the watermark in the extended random extension.

The BSAFE-C library documentation indicates that both watermarking and extended random are supported in some versions of the library; however, the version we have appears to have been compiled without this support.⁴

For both the Java and C versions of BSAFE, we have no evidence that versions of the libraries supporting extended random ever shipped and our major findings do not rely on extended random in any way.

We performed an Internet-wide scan of port 443 and found very few servers on this default port that exhibited this 32 byte watermark: only 386 of 8 million servers contacted. Details on this scan are included in section 6.

4.2 Windows SChannel

SChannel (“Secure Channel”) is a security component in the Windows operating system (introduced in Windows 2000) that provides authentication and confidentiality for socket-based communications. Although it supports several protocols, it is most commonly used for SSL/TLS, including by Microsoft’s Internet Information Services (IIS) server and Internet Explorer (IE). We focus on ECDHE/ECDSA handshakes that use P-256 (which

³An analogous variable enables support for the client.

⁴The header files for the version of BSAFE-C we have show that the library was compiled with the command line flags `-DNO_TLS_EXT RAND -DNO_RSA_WATERMARK`.

in turn cause Dual EC to also use this curve), as used by the version of IIS distributed with Windows 7 64-bit Service Pack 1 and Windows Server 2010 R2. All information about the internal workings of SChannel and its implementation of Dual EC discussed in the following was obtained via reverse-engineering.

Description. SChannel uses Microsoft’s FIPS 140-2 validated Cryptography Next Generation (CNG) API, which includes an implementation of Dual EC. CNG is implemented in two modules, one for user-mode callers (bcryptprimitives.dll) and one for kernel mode (cng.sys). Dual EC is used to generate pseudorandom bytes when the `BCryptGenRandom` function is explicitly directed to use it via a function argument or when it is selected as the system-wide default. When using Dual EC, `BCryptGenRandom` generates enough fresh blocks to satisfy the request, and discards any remaining bytes (i.e., there is no caching between requests).

Whenever SChannel requests random bytes, it calls `BCryptGenRandom` using the system-wide default. Our reverse-engineering efforts and experiments indicate that additional input is not provided by SChannel for TLS connections. TLS handshakes are performed by a separate process (lsass.exe) on behalf of IIS, which dispatches one of several worker threads to handle each request. Dual EC in CNG maintains separate state for each thread, so a successful attack on the state of one thread will not carry over to the others. Importantly, SChannel caches ephemeral keys for two hours (this timeout is hard-coded in the configurations we examined), and the cached keys are shared among all worker threads until the timeout expires.

When performing an ECDHE handshake, SChannel requests random bytes in a different order than OpenSSL and BSAFE: (1) 32 bytes for session ID, (2) 40 bytes for ephemeral private key, (3) 32 bytes (not relevant to the attack), (4) 28 bytes for ServerHello nonce, and (5) 32 bytes for the signature (if using ECDSA). Notice the 40-byte request for the private key, even though a P-256 private key is only 32 bytes; this is because SChannel uses FIPS 186-3 B.4.1 (Key Pair Generation Using Extra Random Bits) to generate ECDHE key pairs, which specifies 8 additional bytes to reduce bias from a modulo operation. More importantly, SChannel requests bytes for the private key *before* the ServerHello random field. This means that any attempt to infer the private key must use the session ID, or random fields from previous handshakes.

Deviation from SP-800-90A. The implementation of Dual EC in CNG differs from the current SP-800-90A specification in one noteworthy way. The final state update (see Section 2) is computed but the result is thrown away. This makes it identical to Dual EC 2006. This appears to be a bug.

Interestingly, the code in `bcryptprimitives.dll` that implements Dual EC (a function called `MSCryptDualEcGen`) seems to include step 14—performing a point multiplication and projection on the x -coordinate after generating the necessary blocks. However, our reverse engineering efforts, as well as our experiments, indicate that the result is not copied into the seed state, and thus not used in subsequent calls to Dual EC. In short, although the CNG Dual EC implementation appears to contain code that implements the full current specification, it effectively implements Dual EC 2006 by ignoring the result of step 14 in future calls to generate.

Fingerprint in the session ID. When an SChannel server generates a new session ID, it requests 32 bytes, $S[0, \dots, 31]$ from `BCryptGenRandom`, and interprets the first four bytes $S[0, \dots, 3]$ as an unsigned integer v . It then computes $v' = v \bmod \text{CACHE_LEN}$, and constructs the final session ID by concatenating these values, $\text{session_id} = v'[0, \dots, 3] || S[4, \dots, 31]$. `CACHE_LEN` is the maximum number of entries allowed in SChannel’s session cache, which was hard-coded to 20,000 on the systems we tested. Thus, the presence of zeros in the third and fourth bytes of the session ID is a likely (although imperfect) fingerprint for SChannel implementations.

Attack 1: Using the server’s random nonce. With Dual EC enabled, it is possible to use the 28-byte ServerHello nonce to learn the server’s ECDHE private key, which will allow decryption of all ECDHE sessions within the two-hour window before the private key is refreshed. As previously discussed, these bytes are requested after the private key is generated, so in order to use them for the attack, we must look at previous handshake messages sent from the server. The fact that SChannel uses multiple threads to perform handshakes complicates the attack, as we cannot know which thread was used for a particular handshake unless we have learned the state of all threads and updated them as new handshakes were performed. On observing a handshake with the new server public ephemeral key, denoted h , the attacker works backwards through previous handshakes, using the random field in each ServerHello message to generate candidate Dual EC states using the basic attack. Each candidate state is checked first against the ECDSA public key to determine the state used in that handshake, and then against the session ID in h to determine if the same state was used to generate the new ephemeral key. The 32 bytes for the ECDSA nonce are generated in two calls, first 24 bytes then 8 bytes. These values are concatenated and then byte-wise reversed to obtain the nonce.

When the matching state is found, it is straightforward to generate the ephemeral private key and subsequent

session keys. SChannel uses FIPS 186-3 B.4.1 to generate the private key, which corresponds to drawing 40 bytes of random input c , and computing the key as $(c \pmod{n-1}) + 1$, where n is the curve order. The worst-case complexity of this attack requires approximately 2^{31} scalar multiplications with a variable base point and four times as many with a fixed base point to check the ECDSA public key, totaling $2^{31}(C_v + 4C_f)$.

Attack 2: Using the session ID. The second approach uses the session ID in a handshake containing a new ECDHE public key. Denote the 32-byte session ID in the relevant handshake by S , and v' the unsigned integer corresponding to $S[0, \dots, 4]$. Recall that SChannel modifies the first four bytes of the session ID by replacing it with its value modulo `CACHE_LEN`. All that one must do to recover the private ephemeral key is run the basic attack on a set of inputs generated by enumerating (1) all 4-byte sequences whose unsigned integer representation v satisfies $v' = v \pmod{\text{CACHE_LEN}}$ (for the first four bytes of the block that generated the session ID), and (2) all 2-byte sequences for the last two bytes of the first block that generated the session ID. Candidates are checked by generating the next 40 bytes, using FIPS 186-3 B.4.1 to construct a private key, and comparing the corresponding public key against that provided in the `ServerKeyExchange`.

This attack sidesteps the issues created by threading in SChannel, but because of the way the the session ID is generated it is actually more complex than the previous. Recall that `CACHE_LEN` = 20,000 in both configurations tested, so this attack requires approximately 2^{18} guesses to deduce the first four bytes of the original session ID block, and 2^{16} for the last two bytes, giving approximately 2^{33} candidate curve points. Of these, approximately 2^{17} will agree with the last two bytes of the session ID, and we determine which is correct by generating two additional Dual EC blocks for a P-256 ECDHE private key, then performing a point multiplication to compare with the public key sent in the same handshake. The total complexity is $2^{33}(C_v + C_f) + 2^{17}(5C_f)$.

4.3 OpenSSL

Description. OpenSSL is one of the most widely used TLS libraries, due to its inclusion in many Linux/Apache distributions. While the standard edition of OpenSSL does not contain Dual EC, OpenSSL also ships a separate package called the OpenSSL FIPS Object Module. When this module is combined with OpenSSL, it provides a TLS library containing all four DRBG algorithms defined in NIST SP800-90A, including Dual EC. The Dual EC algorithm is not the default PRNG in OpenSSL, but it can be manually enabled by changing the PRNG settings through an API call at runtime.

Bug. While investigating the OpenSSL-FIPS implementation of Dual EC, we discovered a previously unknown bug that, in fact, prevented it from being run.⁵ The presence of this bug may suggest that nobody has successfully run OpenSSL-FIPS configured to use Dual EC. However, the CMVP validation lists [16] show many “private” validations of the OpenSSL-FIPS module so it is possible that some commercial manufacturer has repaired this bug without propagating the fix back to the open source OpenSSL tree. For this reason, we felt it worthwhile to repair the bug in the FIPS module in order to investigate the feasibility of the attack.

Analysis of OpenSSL-fixed. We examined a repaired version of the OpenSSL FIPS Object Module version 2.0.5 in combination with OpenSSL 1.0.1e (henceforth “OpenSSL-fixed”). The library consists of two components, `libcrypto.a` which implements the core cryptographic routines, including Dual EC, and `libssl.a` which implements TLS. OpenSSL documentation provides guidance on building the library, as well as usage in common scenarios.

OpenSSL-fixed supports TLS 1.2 with the full complement of elliptic curve cryptography for key exchange and digital signatures. By default, the preferred cipher suites use ECDHE key exchange and either RSA or ECDSA signatures. We investigated connections made using the ECDHE handshake.

OpenSSL includes a textbook implementation of Dual EC based on the NIST SP 800-90 March 2007 revision. On the server side of the standard ECDHE handshake, the `generate` function is called repeatedly to generate the following values: (1) a 32-byte session identifier, (2) a 28-byte server random,⁶ (3) a 32-byte ECDHE ephemeral private key,⁷ and, when ECDSA is being used, a 32-byte nonce. OpenSSL’s implementation of Dual EC does not cache unused random bytes at the conclusion of a generator call, hence each sequence of random bytes begins with up to 30 bytes drawn from a single elliptic curve point. Figure 2 illustrates the generation of these values.

OpenSSL’s use of additional input. While analyzing OpenSSL’s implementation of SP 800-90, we discovered an important difference between OpenSSL and the other

⁵The bug involves a flaw in the runtime self-test mechanism that causes OpenSSL-FIPS to shut down the generator immediately upon initializing it. This bug is not triggered while the module is in TEST mode, which explains why unit and Known Answer Tests did not discover the issue. See [13] for details.

⁶Although we do not discuss attacks against the client, a recent fix to the OpenSSL client implementation increases the amount of PRNG output in the client random to 32 bytes (see <http://bit.ly/1ftSQrM>) which may decrease the attack complexity significantly.

⁷OpenSSL generates this key by drawing 32 random bytes and checking whether the result (expressed as an integer) is less than the group order n . If not, the process is repeated.

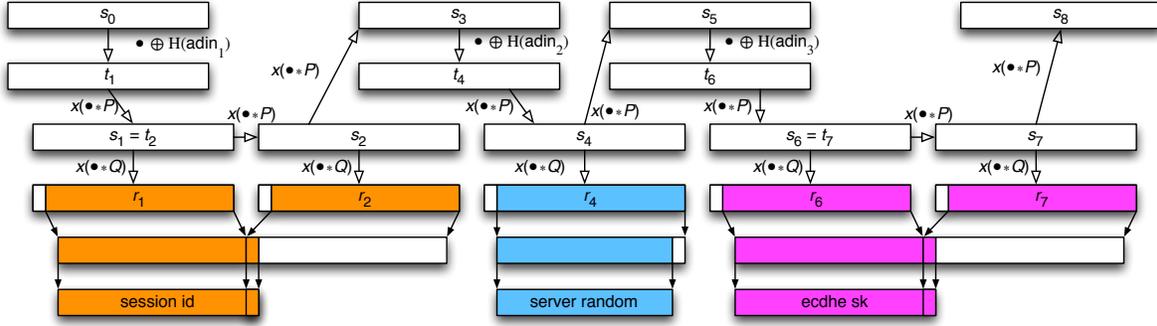


Figure 2: Dual EC usage in OpenSSL-FIPS using ECDHE with P-256.

libraries analyzed in this work. Specifically, OpenSSL provides *additional input* with each call to the generate function. The additional input string is constructed uniquely by the function `FIPS_get_timevec()` prior to each query for random bytes. It comprises 16 bytes with the following structure.

`adin = (time in secs || time in μ secs || counter || pid)`

Each of the component fields in the additional input string is 4 bytes in length. On Unix-based systems the time fields are computed using `gettimeofday()`. The counter is a monotonically increasing global counter that is set to 0 at library initialization, and increments with each call to `FIPS_get_timevec()`. On operating systems where the process IDs are available, pid contains the process ID returned from `getpid()`.

A passive attacker can capture 32 consecutive bytes of Dual EC output by observing the session ID sent to the client by an OpenSSL server. Assuming the generator is instantiated with P-256, the attacker can now execute the initial steps of the basic attack using the first 30 bytes, in order to recover multiple candidate states, and (using the additional two bytes) reduce the number of candidate states to one, or a small number. From this point, the OpenSSL attack differs from the basic attack. Given each candidate state s , the attacker now calculates the step-14 update $s = x(sP)$ and exhaustively guesses the additional input string used in the next call to the generate function as $s' = s \oplus H(\text{adin})$. This requires the attacker to iterate through a set of candidate adin input strings, executing the steps of the generate algorithm to recover a candidate ECDHE private key, and comparing this value to the intercepted ECDHE public key from a real handshake trace.

The complexity of this attack depends on two factors: the number of candidate states remaining at the conclusion of the first portion of the attack, and the number of candidate adin strings. Since we are guessing 16-bits, only about half of all strings give a valid x -coordinate, and are comparing the resultant output against 16 bits,

we expect to see 1 or 2 candidate states that generate the correct first two values. In practice, we never saw more than 3 candidate states.

Since the time in seconds is already transmitted as part of the server random, the first portion of adin is known. Thus it remains to predict the time in μ seconds, process ID and counter. Under reasonable assumptions about the operating system and the number of connections so far handled by the server, this can range from approximately 2^{20} (primarily guessing the μ secs field) to 2^{35} with a typical Unix range of pid values and known counter value, and possibly 2^{45} or more depending on how recently the library was initialized. Notice that once an attacker recovers the adin string for a first TLS connection, it may be relatively easy to predict these values for later connections.

The inclusion of additional input complicates the attack since recovering the Dual EC state when it is most convenient, namely during the generation of the session ID, does not immediately translate into recovering the session keys. There are two cases to consider.

In the first case, the attacker knows nothing about the state of the generator except that the counter value is no bigger than $k \leq 32$ bits. The first step is to recover the generator state (for ease of analysis, assume only one candidate state is possible). As with BSAFE-C, this requires approximately 2^{15} variable base point multiplications and an equal number of fixed base point multiplications. Next, the additional input string needs to be guessed. For each guess, this takes two fixed base point multiplications. There are at most 2^{35+k} additional input strings to try. A guess can be validated by comparing to the server random field. Finally, the ECDHE secret and public keys need to be computed for each guess of the second additional input string. Each guess takes five fixed base point multiplications; however, since the attacker has already determined the pid and the counter value, the attacker has a good estimate of the time and increments the microsecond value from there; this takes about 2^{13} guesses. This gives a total cost of $2^{15}(C_v + C_f) + 2^{35+k}(2C_f) + 2^{13}(5C_f)$. The 2^{13}

is an upper bound for our observations. Usually less than 2^{12} tests were sufficient and on a fast Internet servers even less time passes between two calls of Dual EC.

In the second case, the attacker has already broken a previous connection and so the pid and counter values are known. The cost of performing the whole attack a second time becomes $2^{15}(C_v + C_f) + 2^{20}(2C_f) + 2^{13}(5C_f)$. However, the cost of computing a scalar multiplication with a variable base point is significantly higher than for a fixed base point. It may be in the attacker’s best interest to keep track of the generator’s state throughout each session. This involves keeping track of counter updates and recovering the state after each encrypted TLS record sent and randomness used for ECDSA and IVs. The search space for the time in adin for these values is usually small, similar to that in the ECDHE key.

Then the cost of recovering the state at the beginning of a new connection is at most $2^{20}(2C_f)$ for testing the time (and less if better estimates of the time are known) in place of the $2^{15}(C_v + C_f)$, for a total cost of $2^{20}(2C_f) + 2^{13}(7C_f)$. This is faster if the time update for the server random call requires a smaller search space for the time after the time has been determined for the session ID.

4.4 Attack validation

We implemented each of the attacks against TLS libraries described above to validate that they work as described. Since we do not know the relationship between the NIST-specified points P and Q , we generated our own point Q' by first generating a random value $e \xleftarrow{R} \{0, 1, \dots, n-1\}$ where n is the order of P , and set $Q' = eP$. This gives our trapdoor value $d \equiv e^{-1} \pmod{n}$ such that $dQ' = P$. (Our random e and its corresponding d are given in the Appendix.) We then modified each of the libraries to use our point Q' and captured network traces using the libraries. We ran our attacks against these traces to simulate a passive network attacker.

We would like to stress that anybody who knows the back door for the NIST-specified points can run the same attack on the fielded BSAFE and SChannel implementations without reverse engineering.

We describe the concrete performance results of our attacks in the next section and give details on the libraries here.

RSA BSAFE. The Dual EC implementation in BSAFE-C contains the points P and Q as well as three tables of scalar multiples of each of the points for fast multiplication. The tables contain 65, 517, and 573 multiples. After working out the corresponding scalar factor for each entry in the tables, we computed our own tables and modified the relevant object files in the library. There were no health checks or known answer tests (KATs) to bypass.

BSAFE-Java is distributed as a signed, obfuscated jar file. We reverse engineered the code sufficiently to find

and bypass the checks that prevent modification and replaced the jar’s signature with our own. BSAFE-Java has a single table of 431 scalar multiples of each of P and Q .

Windows SChannel. Dual EC in SChannel is implemented both in the kernel and a user-mode library. We modified the user-mode library, which performs a KAT when the operating system first loads the module at boot, as well as continuously during operation when FIPS mode is enabled. To sidestep these checks, we disabled FIPS mode, and wrote a system service that (1) replaces Q with Q' in the the address space of the Local Security Authority Subsystem Service (IIS and IE delegate TLS handshakes to this process), and (2) makes Dual EC the system-wide default PRNG.

OpenSSL-fixed. Dual EC in OpenSSL is implemented in the separate OpenSSL-FIPS library. This library contains both runtime KATs and a check of the SHA-1 hash of the object code. Since the hash is computed each time the library is compiled, we simply fixed the bug which prevents Dual EC from being used (described above), bypassed the KATs, and substituted Q' for Q .

5 Implementation

We implemented all attacks for parallel architectures, specifically clusters of multicore CPUs. The attacks are parallelized using OpenMP and MPI, with the search space distributed over all cores of the cluster nodes, using one process per CPU and one thread per (virtual) core. The attacks are “embarrassingly parallel”: there are no data dependencies between the parallel computations and thus no communication overhead and no limit on the scalability of the parallelization, other than the total number of independent computations.

5.1 Algorithmic optimizations

For finite-field arithmetic, we use the Gueron/Krasnov OpenSSL patch for NIST P-256, described in [8] and available at [9]. Square-root computations, to recover the y -coordinates, use that $p \equiv 3 \pmod{4}$ and compute \sqrt{a} as $a^{(p+1)/4}$. We refer to the cost of recovering a y -coordinate as C_y .

The definition of the update function in Dual EC requires all scalar multiplications to result in affine points (to derive a unique x -coordinate). To improve the performance of our implementation we compute all point operations in affine coordinates and batch the inversions using Montgomery’s trick [15] across several parallel computations. We use a batch size of 256 for all experiments; increasing the batch size any further does not have a measurable effect on the runtime.

The most performance critical operations on EC points in the attack logic are:

1. Scalar multiplications using fixed base points P and Q in order to compute the next internal state and to compute the output string respectively; P is also used as base point for ECDHE and ECDSA computations.
2. Scalar multiplications using variable base points and a fixed scalar, the back door d , in order to compute a candidate internal state given an output string.

In Table 1 we refer to the costs of a fixed-base-point scalar multiplication as C_f and those of a variable-base-point one as C_v .

For the fixed-base-point computations we use large pre-computed tables of multiples of the base point. For a given width w we compute the lookup table consisting of $T_{P,i,j} = i2^{jw}P$ for $0 < i < 2^w$, $0 \leq j < 256/w$. A scalar multiplication sP can then be performed as $\lceil 256/w \rceil$ additions of precomputed points from the lookup table using $sP = \sum_{j=0}^{256/w} T_{P,s(j),j}$, where $s = \sum_{j=0}^{256/w} s(j)2^{jw}$. We do the same for Q in place of P . These tables are shared among all threads of each process in the implementation. We choose $w = 16$ for all our experiments for a reasonable balance between performance and lookup table size. This brings C_f down to 16 point additions.

We implemented the scalar multiplications with the fixed scalar d using signed sliding windows with window width 5 and fully unrolled the code. This way C_v takes 253 doublings and 50 additions. Our d was a randomly chosen 256-bit integer; see the appendix. An attacker can choose d to minimize the cost of the fixed-scalar variable-base-point scalar multiplication by choosing d with low Hamming weight or more generally with a short addition chain, although a sufficiently low weight runs the risk that someone will discover d by a discrete-logarithm computation. To put an upper bound on the Dual EC attack time we avoid this optimization.

There is a proof of concept of the general Dual EC attack in a blog post by Adamantiadis [1] using OpenSSL’s libcrypto for curve and large integer arithmetic. Adamantiadis does not implement a complete attack but recovers the state from a 30 byte random output. His proof of concept iterates through all 2^{16} candidates to recover the missing bits of the x -coordinate and computes the corresponding y coordinate. In case he discovers a point on the curve, he applies the back-door computation and computes the next random output. This proof of concept has an expected cost of $2^{16}C_y + 2^{15}(C_v + C_f)$.

On a single core of an Intel Xeon CPU E3-1275 v3, Adamantiadis’s code requires about 18.5 s. To circumvent the bug in the OpenSSL FIPS implantation, Adamantiadis is using an older version of libcrypto. We modified Adamantiadis’s code to run with libcrypto version 1.0.1e; this version requires about 12.1 s. For comparison, we reimplemented Adamantiadis’s proof of concept using our optimized primitives. The optimized version requires

about 3.7 s on a single core. Thus, our optimizations give an improvement by a factor of 3.3 over libcrypto.

In Adamantiadis’s code (using libcrypto version 1.0.1e), the computation of a y coordinate (corresponding to cost C_y) takes about 15 μ s on average. In our optimized version, this computation requires only 6 μ s, which is an improvement by a factor of 2.5. The application of the back-door computation in Adamantiadis’s code (scalar multiplication of a variable point by a fixed factor, cost C_v) requires about 168 μ s on average, our code requires about 98 μ s, which is an improvement by a factor of 1.7. Scalar multiplication with fixed base points P and Q (cost C_f) benefits the most from our optimizations. In Adamantiadis’s code, one scalar multiplication requires about 171 μ s on average. In our optimized code, the computation for fixed base points requires only about 6 μ s on average which is an improvement by a factor of 28. For an actual attack, the proportion of C_f to C_v is significantly larger than in the proof of concept. This increases the impact of our improvements on the attacks.

5.2 Performance measurements and estimates

All our attacks are based on the fact that some fields in the handshake messages (e.g., session ID and server random) contain a bit sequence derived from the x -coordinate of a point R . In order to recover R , we iterate through all possible combinations of the missing bits, check whether each candidate r_i actually is a valid x -coordinate and gives a point candidate R_i , apply the back door by computing dR_i , and follow all the steps (including adin for the attacks on OpenSSL-fixed) to check whether the candidate r_i eventually allows us to recover the (EC)DH secret. As the steps differ for each implementation, a different amount of computation is required for each attack (see Table 1, column “Attack Complexity”).

We measure the cost of the attacks on a reference CPU, an Intel Xeon CPU E3-1275 v3, which has 4 cores and 2 hardware threads per core when enabling Hyper Threading. Table 2 lists measured and estimated performance numbers of the attacks. Turbo Boost and Hyper Threading are enabled; thus, we were using 8 OpenMP threads for the measurements on the reference system.

We measure the runtime of testing 2^{22} candidates (about 2^{21} candidate points). From these measurements, we extrapolate the expected runtime of the attack. From the expected runtime, we compute the cost of the attack as the number of Intel Xeon reference processors that would be required to perform the attack in an expected time of less than one second.

Finally, to verify the efficiency of the attack on multiple nodes, we measure the total worst case runtime of the attack on a four-node, quad-socket AMD Opteron 6276 (Bulldozer) computing cluster. The cluster has an Infiniband interconnect and 256 GB memory per node —

Table 2: Performance measurements and estimates.

Attack	Intel Xeon Reference System			16-CPU AMD Cluster
	2^{22} Candidates (s)	Expected Runtime (min)	Expected Cost	Total Runtime (min)
BSAFE-Java v1.1	75.08*	641	38,500	63.96*
BSAFE-C v1.1	–	0.26	16	0.04*
SChannel I	72.58*	619	37,100	62.97*
SChannel II	62.79*	1,760	106,000	182.64*
OpenSSL-fixed I	–	0.04	3	0.02*
OpenSSL-fixed II	–	707	44,200	83.32*
OpenSSL-fixed III	–	$2^k \cdot 707$	$2^k \cdot 44,200$	$2^k \cdot 83.32$

*measured

however, neither of these is relevant for the attacks because they do not need much communication and require less than 1 GB of RAM per process.

For the timing measurements we ran each case several times to verify that there is no significant variance and finally picked the time from a representative test run.

BSAFE-Java v1.1. In this case, the session ID of the handshake is not derived from Dual EC—so we have to use the 28 bytes of the server random, missing 32 bits of the target x -coordinate. The complexity of the attack on BSAFE-Java is $2^{32}C_y + 2^{31}(C_v + 5C_f)$. We measured a time of 75.08 s to check 2^{22} candidates. In total, this attack requires checking at most 2^{32} candidates, so we expect a runtime of $2^{32-22} \cdot 75.08 \text{ s}/2 \approx 641 \text{ min}$ (rounded to three significant digits) on the reference CPU. Therefore, the expected cost to finish this attack within one second is about 38,500 reference CPUs. We measured a worst-case total runtime of 63.96 min on our cluster.

BSAFE-C v1.1. For the BSAFE-C attack, we can simply concatenate session ID and server random and guess 16 bits of the target x -coordinate for the 30 possible cases. The complexity of the attack on BSAFE-C is $30 \cdot [2^{16}C_y + 2^{15}(C_v + C_f)]$. In the worst case this only requires testing $30 \cdot 2^{16}$ candidates which is less than 2^{22} , so we do not have a measurement for the first column in Table 2. Instead, we measured the worst case time for the whole attack (31.12 seconds) and list half of the worst case time, i.e., $31.12 \text{ s}/2 \approx 0.26 \text{ min}$ as expected runtime. This gives an expected cost of 16 reference CPUs. This attack required 0.04 min on our cluster; most of this time is probably due to initialization overhead.

SChannel I. The SChannel I attack uses the server random from the preceding handshake to hook into the random number stream and to discover the server’s ECDHE private key in the handshake when the private key is refreshed. The complexity of this attack is $2^{32}C_y + 2^{31}(C_v + 4C_f)$. Checking 2^{22} candidates takes 72.58 s. This is

slightly less than the time for BSAFE-Java, because this attack requires only four instead of five multiplications by a fixed base point for each point candidate. The whole attack requires checking at most 2^{32} candidates, so the expected runtime is $2^{32-22} \cdot 72.58 \text{ s}/2 \approx 619 \text{ min}$. Therefore, the expected cost of the attack is 37,100 reference CPUs. The measured worst-case total time on our cluster is 62.97 min.

SChannel II. The SChannel II attack uses just one single handshake to recover the secret keys and therefore relies on the session ID (where the 4 least significant bytes have been replaced by their value modulo 20,000) to recover the state of the PRNG. The complexity of attack is $2^{34}C_y + 2^{33}(C_v + C_f) + 2^{17}(5C_f)$, more precisely $2^{32}/20,000 \cdot 2^{16}[C_y + (C_v + C_f + 5C_f/2^{16})/2]$. The dominant part for each candidate check is $C_y + (C_v + C_f)/2$ which requires a smaller number of multiplications with a fixed base point than SChannel I. Thus, we measured only 62.79 s to check 2^{22} candidates. This attack requires checking up to $2^{32}/20,000 \cdot 2^{16}$ candidates; therefore, this attack has an expected runtime of $2^{32-22}/20,000 \cdot 2^{16} \cdot 62.79 \text{ s}/2 \approx 1,760 \text{ min}$. This gives an expected cost of 106,000 reference CPUs.

OpenSSL-fixed. Due to the use of adin before each random draw, OpenSSL is a special case among the implementations of Dual EC. The attack on OpenSSL takes three steps: First, we find the current state by finding the 16 missing bits for the session ID. This requires checking at most 2^{16} candidates; thus, we do not give a measurement for 2^{22} candidates in the first column of Table 2. Since this step might result in more than one state candidate, we always compute all 2^{16} candidates. If more than one candidate is recovered, the attacker either has to check all candidates (in parallel) or retry with a different handshake if applicable. In the following we investigate the expected case that only candidate is found. In the second step, we need to find the adin used to generate server

random. Here, μs consists of the current system time (including μs), the process ID (pid), and a counter value. In the last step, we need to find the next μs before the call to generate the DH key. The pid and the counter are known from the previous μs ; we only need to find the μs over a very short time span by iteratively incrementing the time counter. We assume a maximum of $2^{13} \mu s$ for this time span. Due to the small workload, the scalability of the parallelization of step one and three is limited.

The complexity of the OpenSSL attack is $2^{16}C_v + 2^{15}(C_v + 3C_f) + 2^{20+k+l}(2C_f) + 2^{13}(5C_f)$ where k is the number of unknown bits of the μs counter and l is the number of unknown bits of the μs pid. We are using a batch size of 256. Therefore, we can split the workload of checking 2^{16} candidates in the first step to at most $2^{16}/256 = 256$ threads without loss of efficiency. The last step requires at most 2^{13} iterations, so we can use at most $2^{13}/256 = 32$ threads. Step one and three contribute to only an insignificant fraction of the total complexity when pid or counter are not known.

We examine three cases:

OpenSSL-fixed I: pid and counter are known, μs of time are unknown,

OpenSSL-fixed II: counter is known, μs of time and pid (15 bits) are unknown,

OpenSSL-fixed III: μs of time, pid (15 bits), and counter (k bits) are unknown.

The system time in seconds is known due to the timestamp in the server-random field of the handshake message. Therefore, only the μs must be found by exhaustive search. The seconds might have clocked since the timestamp was obtained; thus, we need to test up to $1,000,000 + \Delta$ candidates for the μs . An upper limit on Δ is the time between the server receiving the ClientHello message and sending the ServerHello message. We use $1,000,000 \mu s + 48,576 \mu s = 2^{20} \mu s$ as upper limit.

The standard maximum pid on Linux systems is 2^{15} . If the attacker starts listening to the server right after bootup, he can assume the initial counter to be zero; otherwise, he may make an educated guess about the current counter state based on uptime and the average connection number.

To compute the expected runtime of this attack we measured the worst-case runtime of the case OpenSSL-fixed I. The first step to compute state candidates took about 0.96 s; the second step checking all possible $2^{20} \mu s$ for one single state candidate took 2.59 s. The final step checking the next $2^{13} \mu s$ took only 0.05 s. Therefore, the expected runtime of OpenSSL-fixed I is $0.96 s + 2.59 s/2 + 0.05 s/2 \approx 0.04$ min; the expected cost is three reference CPUs. The expected runtime of OpenSSL-fixed II is $0.96 s + 2^{15} \cdot 2.59 s/2 + 0.05 s/2 \approx 707$ min. We are using 8 threads in the reference system; the maximum

number of threads for the first step is 256 threads and 32 threads for the last step. Therefore, the first step can not be faster than $0.96 s/(256/8) \approx 0.03 s$ and the last step requires at least $0.05 s/(32/8) \approx 0.01 s$ in the worst case. To finish the whole attack in one second on average, the second step must take $1 s - 0.03 s - 0.01 s = 0.96 s$ on average which requires $2^{15} \cdot 2.59 s/2/0.96 s \approx 44,200$ reference CPUs. For OpenSSL-fixed III these values are multiplied by 2^k , assuming k unknown bits for the counter of the μs .

We ran OpenSSL-fixed II on the cluster, testing all 2^{35} combinations of μs and pid to obtain the worst-case total runtime. The time of 83.32 min is noticeably less than 2^{15} times the time of 0.02 s taken in the OpenSSL-fixed I scenario. This is because in OpenSSL-fixed I the computations of $2^{15}C_v$ for the first step have a strong impact on the runtime, testing 2^{35} candidates gives more precise timing estimates of C_f . We can extrapolate the costs for OpenSSL-fixed III as 2^k times those of OpenSSL-fixed II because the contribution of the first and of the third step become negligible.

These runtime and cost estimates show that a powerful attacker (in case of BSAFE-C, an arbitrary attacker) is able to break TLS connections that use the Dual EC pseudorandom number generator when he possesses the back-door information. The usability of the attack on OpenSSL-fixed depends on additional knowledge about the μs ; however, computing clusters of around 100,000 CPUs are realistic as of today (for example the Tianhe-2 supercomputer in China has 16,000 computing nodes with 5 CPUs each [25]) and sufficient to break BSAFE and SChannel in less than one second.

6 Passive TLS server fingerprinting

In many contexts, including exploitation of the Dual EC backdoor, it is useful to identify, or fingerprint, the implementation used by a TLS server. Existing tools for TLS fingerprinting use active techniques (requesting a page to get an error message and analyzing the result), but our investigations of TLS implementations suggest that the session ID field, in particular, admits a passive fingerprinting mechanism useful to an attacker observing network traffic or even one attacking recorded connections from years ago.

Data collection. We collected a large dataset of TLS session information from servers listening on port 443 in the IPv4 address space. We executed a ZMap scan [6] of port 443 over the entire IPv4 address space (excluding ZMap’s default blacklist). The ZMap scan netted 38.9 million services responding on port 443. For 37.1 million of these services, we used a modified version of OpenSSL v1.0.1e `s_client` to connect to the service, and attempt to perform a TLS handshake up through receiving the ServerHello message (containing the session ID, server

random value, and TLS server extensions), and then sent a TCP RST to the server. Of these attempts, 21.8M servers responded with a ServerHello message.

We investigated a number of candidate fingerprints based on observable behavior to a passive adversary. For each server that exhibited the RSA BSAFE fingerprint, we made an HTTP GET request on port 443 in an attempt to determine what software the server uses via the self-reported `Server` field of the HTTP header. We repeated this for 1,000 randomly-selected IP's exhibiting the SChannel fingerprint. We consider an observable behavior to be a *selective fingerprint* if $\geq 95\%$ of the servers from which we received HTTP headers identify themselves as the same implementation.

6.1 Fingerprints detected

We detected many different types of fingerprints by examining server random values, session IDs, and TLS server extensions (all unencrypted values to a passive observer). In addition to the fingerprints on BSAFE and SChannel discussed in Sections 4.1 and 4.2, we identified five selective fingerprints from unique combinations of supported extensions, 2 selective fingerprints corresponding to session ID values with fewer than 32 bytes, and seven selective fingerprints corresponding to fixed subsequences in the session ID.

In sum, 4 million of the servers we contacted exhibited selective fingerprints. We discuss our findings for BSAFE and SChannel in more detail below.

RSA BSAFE. As described in Section 4.1, by default, BSAFE-Java has a very prominent fingerprint that is enabled by default, and BSAFE-C has a similar fingerprint that is not enabled by default. We found 720 servers with the BSAFE-Java fingerprint, and none with the BSAFE-C fingerprint. Of these servers, 33% self-reported running Apache Coyote 1.1,⁸ with the remaining two self-reported implementations (“ADP API” and `lighthttpd`) appearing on fewer than ten instances. The remaining servers did not return a `Server` field.

Microsoft SChannel. As described in Section 4.2, SChannel exhibits a fingerprint in the first 4 bytes of the session ID. 2.7 million of the servers we contacted exhibited this fingerprint. We requested HTTP headers from 1,000 of these IPs (randomly selected), and 96% of the responses included the string “Microsoft” in the `server` field, suggesting that this is a selective fingerprint.

7 Conclusions

We provided the first theoretical and practical analysis of the exploitability of Dual EC as used in deployed TLS

⁸Apache Coyote is a front end that forwards requests to Apache Tomcat, which supports Java Servlets and JavaServer pages; running Tomcat with BSAFE-Java may indicate an effort to provide a FIPS-compliant web application.

implementations. We evaluated the viability and performance of recovering TLS session keys for fielded implementations that use Dual EC. Our results demonstrate that otherwise innocuous implementation decisions greatly affect exploitability. For example, RSA BSAFE-C is by far the easiest to exploit due to caching of unused bytes of Dual EC output. On the other end of the spectrum, OpenSSL-fixed uses additional input, which can render attacks significantly more challenging if no or only little information is available about the server.

We developed and successfully tested state-of-the-art parallelized implementations of all attacks against versions of the libraries patched to use Dual EC constants that we generated. Depending on the design choices in the implementations, an attacker can recover TLS session keys within seconds on a single CPU or may require a cluster of more than 100,000 CPUs for the same task if a different library is used. For OpenSSL some parameters might require such a serious cluster for an even longer time.

While there are a number of available mitigations to the vulnerabilities we discuss in this work, the simplest and best is to remove the Dual EC implementation from deployed products. OpenSSL has already initiated the (expensive, due to FIPS certification) process of removing Dual EC from its FIPS version and, in the meantime, is not fixing the bug we discovered that prevents its use [13]. RSA has advised developers to stop using the BSAFE Dual EC implementation [7]. Our work further emphasizes the need to deprecate the algorithm as soon as possible.

Acknowledgements

We would like to thank Bo-Yin Yang’s research group at Academia Sinica, Taiwan, for giving us access to their computing cluster for benchmarking. We also thank Steve Marquess and Dr. Stephen Henson of the OpenSSL Foundation for helpful discussions. We thank Kurt Opsahl and Nate Cardozo of the Electronic Frontier Foundation for their legal advice throughout this project.

This work was supported in part by the European Commission through the ICT program under contract INFSO-ICT-284833 (PUFFIN), by the US National Science Foundation under grants 1018836 and 1314919, and by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005.

References

- [1] Aris Adamantiadis. Dual_EC_DRBG backdoor: a proof of concept, December 2013. Online: <http://blog.0xbadc0de.be/archives/155>.
- [2] James Ball, Julian Borger, and Glenn Greenwald. Revealed: how US and UK spy agencies defeat internet privacy and security. *The Guardian*, September 5

2013. Online: <http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security>.
- [3] Daniel R. L. Brown and Scott A. Vanstone. Elliptic curve random number generation, August 2007. Online: <http://www.freshpatents.com/Elliptic-curve-random-number-generation-dt20070816ptan20070189527.php>.
- [4] Art Coviello. RSA Conference 2014 keynote for Art Coviello, February 2014. Online: <http://www.emc.com/collateral/corporation/rsa-conference-keynote-art-coviello-february-24-2014.pdf>.
- [5] Tim Dierks and Eric Rescorla. RFC 5246: The transport layer security (TLS) protocol version 1.2. The Internet Engineering Task Force, August 2008. Online: <http://www.rfc-archive.org/getrfc.php?rfc=5246>.
- [6] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In *Proceedings USENIX Security 2013*, pages 605–619, 2013.
- [7] Dan Goodin. Stop using NSA-influenced code in our products, RSA tells customers, September 2013. Online: <http://arstechnica.com/security/2013/09/stop-using-nsa-influence-code-in-our-product-rsa-tells-customers/>.
- [8] Shay Gueron and Vlad Krasnov. Fast prime field elliptic curve cryptography with 256 bit primes. *IACR Cryptology ePrint Archive*, 2013:816, December 2013. Online: <http://eprint.iacr.org/2013/816>.
- [9] Shay Gueron and Vlad Krasnov, 2013. Online: <http://rt.openssl.org/Ticket/Display.html?id=2582>.
- [10] Paul Hoffman. Additional random extension to TLS, February 2010. Online: <http://tools.ietf.org/html/draft-hoffman-tls-additional-random-ext-01>. Internet-Draft version 01.
- [11] Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 27, IT Security techniques. ISO/IEC 18031: Information technology – Security techniques – Random bit generation, 2005.
- [12] Jeff Larson, Nicole Perlroth, and Scott Shane. Revealed: The NSA’s secret campaign to crack, undermine Internet security. *ProPublica*, September 2013. Online: <http://www.propublica.org/article/the-nsas-secret-campaign-to-crack-undermine-internet-encryption>.
- [13] Steve Marquess. Flaw in Dual EC DRBG (no, not that one), December 2013. Online: <http://marc.info/?l=openssl-announce&m=138747119822324&w=2>.
- [14] Joseph Menn. Exclusive: Secret contract tied NSA and security industry pioneer. *Reuters*, December 2013. Online: <http://www.reuters.com/article/2013/12/20/us-usa-security-rsa-idUSBRE9BJ1C220131220>.
- [15] Peter Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48: 243–264, 1987.
- [16] National Institute of Standards and Technology. DRBG validation list. February 2014. Online: <http://csrc.nist.gov/groups/STM/cavp/documents/drbg/drbgval.html>.
- [17] National Institutes of Standards and Technology. Special publication 800-90. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, 2012. (first version June 2006, second version March 2007). Online: <http://csrc.nist.gov/publications/PubsSPs.html#800-90A>.
- [18] Nicole Perlroth, Jeff Larson, and Scott Shane. N.S.A. able to foil basic safeguards of privacy on web. *International New York Times*, September 2013. Online: <http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html>.
- [19] Eric Rescorla and Margaret Salter. Opaque PRF inputs for TLS, December 2006. Online: <http://tools.ietf.org/html/draft-rescorla-tls-opaque-prf-input-00>. Internet-Draft version 00.
- [20] Eric Rescorla and Margaret Salter. Extended random values for TLS, March 2009. Online: <http://tools.ietf.org/html/draft-rescorla-tls-extended-random-02>. Internet-Draft version 02.
- [21] RSA Security Inc. The RSA watermark. 2009. Online: https://developercontent.emc.com/docs/rsashare/share_for_java/1.1/dev_guide/group__LEARNJSSE__WATERMARK.html.
- [22] Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. *Cryptology ePrint Archive*, Report 2006/190, 2006. Online: <http://eprint.iacr.org/>.
- [23] Dan Shumow and Niels Ferguson. On the possibility of a back door in the NIST SP800-90 Dual Ec

Prng. CRYPTO 2007 Rump Session, August 2007.
Online: <http://rump2007.cr.yt.to/15-shumow.pdf>.

[24] Kristian Gjøsteen, 2006. Online: <http://www.math.ntnu.no/~kristiag/drafts/dual-ec-drbg-comments.pdf>.

[25] TOP500, June 2013. Online: <http://www.top500.org/lists/2013/06/>.

Appendix

Our experiments used the P-256 curve parameters. This curve is defined over \mathbf{F}_p with $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. The curve is given in short Weierstrass form $E: y^2 = x^3 - 3x + b$, where

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

The base point P has order n , where

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

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

```
n = ffffffff 00000000 ffffffff ffffffff \  
bce6faad a7179e84 f3b9cac2 fc632551.
```

The official second point Q as given in the Dual EC description is

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

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

To implement our attacks we generated the following random constant e to compute a new point $Q' = eP$. The trapdoor is $d \equiv e^{-1} \pmod{n}$.

```
e = facc5582 909e66b3 09b1a3ae 5e4d51fc \  
0edbf57 6ef8bfa9 c233b035 9f7a7b49
```

```
d = 6fc45453 894de99c 661581b0 a12087b8 \  
62667b78 5aaba711 6dcdcb3c b3a79afe
```

```
Q'x = f6c4f766 b3c61f09 e6095822 24cc8ebc \  
cf4cd496 1ef780cc 02e8f09a 0efa7ca5
```

```
Q'y = f212c576 8d46716c 6cac4d23 ff12e8ae \  
89fd9eee c83a0e83 e35db3aa de0ccb5b
```