CSE 291-E: Applied Cryptography

Nadia Heninger

UCSD

Fall 2020 Lecture 13

Legal Notice

The Zoom session for this class will be recorded and made available asynchronously on Canvas to registered students.

Announcements

1. HW 5 is due today!
2. HW 6 is due Tuesday!

Last time:

- Digital signatures

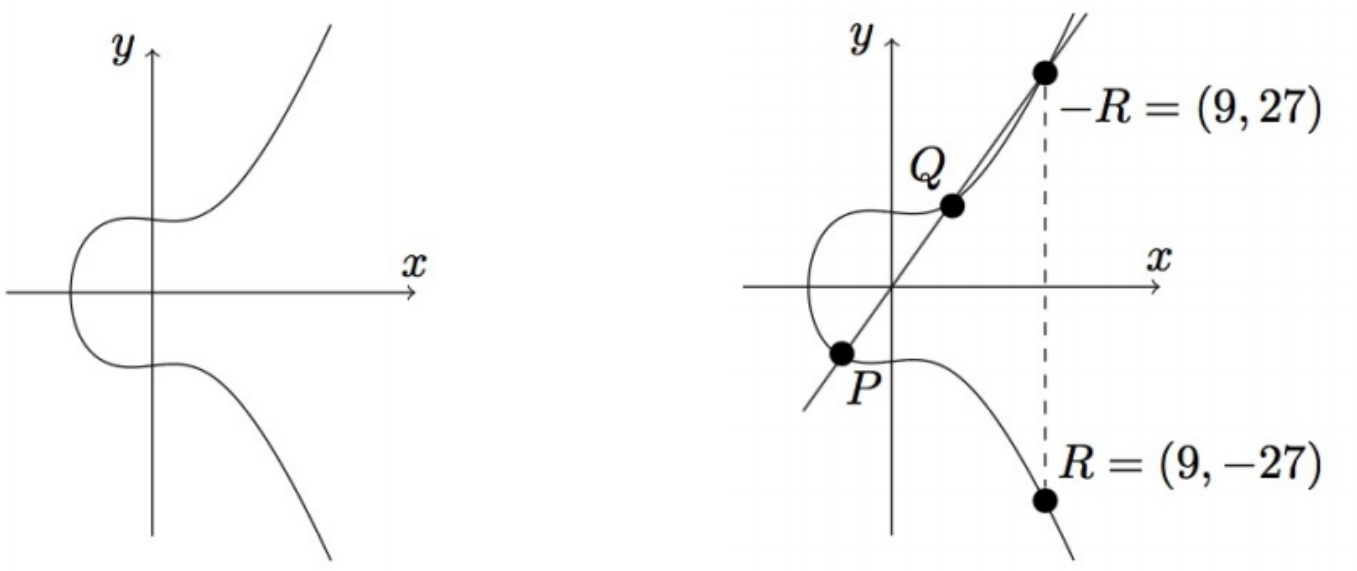
This time:

- Elliptic curve cryptography

Elliptic curves: Motivations

- Factoring and finite field discrete log both have subexponential-time algorithms.
- Current records: 829 bits for factoring, 795 bits for discrete log.
- Current recommendations: use 2048-bit public keys for RSA, Diffie-Hellman, or DSA.
- These are large, so the operations are slow.
- The best cryptanalysis we have for elliptic curves is much weaker, exponential time, so key sizes can be much smaller for the same security level.

Points on an elliptic curve

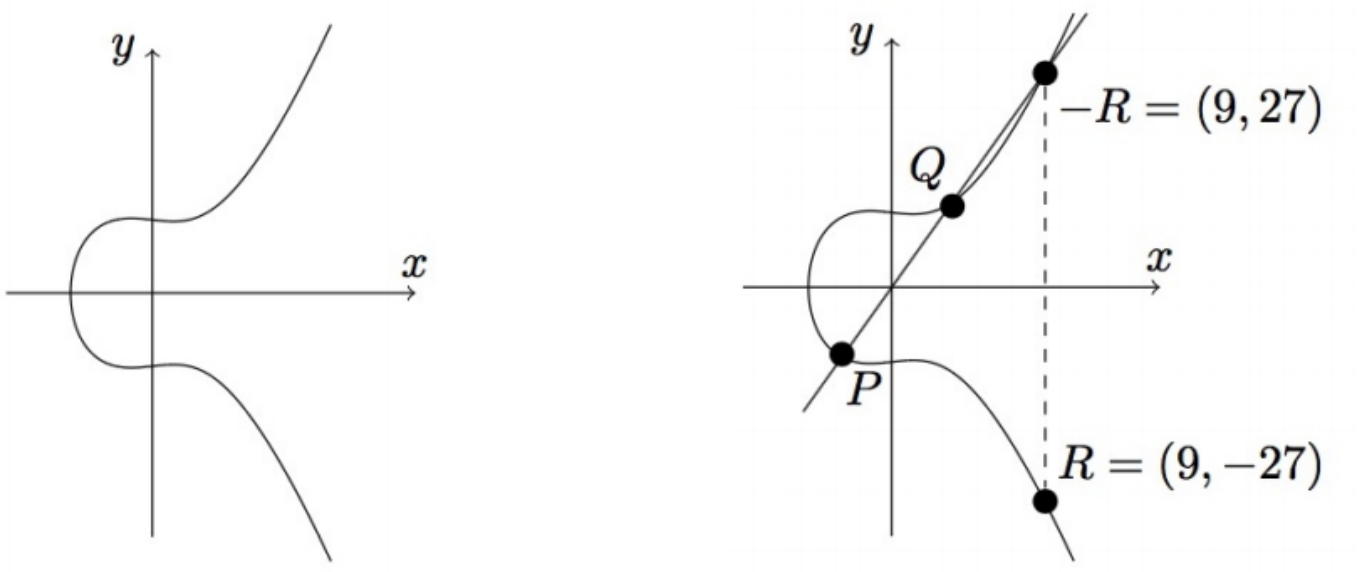


(a) The curve (b) Adding $P = (-1, -3)$ and $Q = (1, 3)$

Figure 15.1: The curve $y^2 = x^3 - x + 9$ over the reals (not drawn to scale)

- Every intro to elliptic curves includes a figure like this.
- The curve is defined by an equation $y^2 = x^3 + ax + b$.
- This picture plots this curve over $\mathbb{R}^2$.
- Classically, Diophantus and Poincaré were interested in rational points $(x, y) \in \mathbb{Q}^2$ on this curve.

Points on an elliptic curve



(a) The curve (b) Adding $P = (-1, -3)$ and $Q = (1, 3)$

Figure 15.1: The curve $y^2 = x^3 - x + 9$ over the reals (not drawn to scale)

- Poincare's method for finding rational points:
  - Take two rational points, define a line (like $y = 3x$ above), and substitute to get a univariate cubic equation.
  - Since it has two rational roots already, the third root is also rational.
  - Get two new points for free: R and $-R$.
- Can define this as a group law: $P + Q = -R$.
- (The operation "+" means apply the above procedure.)

Elliptic curves over finite fields

- For cryptography, we define curves over $\mathbb{F}_p$. Still have curve equation $y^2 = x^3 + ax + b$, with $a, b \in \mathbb{F}_p$, $4a^3 + 27b^2 \neq 0$.
- This is called Weierstrass form. Every elliptic curve can be written in this form.
- Can write down group law in terms of (x, y) coordinates but it's long and has multiple cases.
- Identity element: Special point $\mathcal{O}$ is "point at infinity".
- Group order: Hasse: $|E(\mathbb{F}_{p^e})| = p^e + 1 - t$ for some $-2\sqrt{p^e} \leq t \leq 2\sqrt{p^e}$.

Some curves admit nicer representations with computational benefits:

- Montgomery form speeds up addition.
- Edwards curves have a simpler addition law that also allows faster operations.
- A curve $E/\mathbb{F}_p$ in Edwards form can be written $x^2 + y^2 = 1 + dx^2y^2$, $d \in \mathbb{F}_p$, $d \notin \{0, 1\}$.
- Edwards point addition: $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$

$$P_1 + P_2 = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

Elliptic curve scalar multiplication and discrete log

We have a group law $P + Q = R$, usually written as "addition".

Scalar multiplication of points

- If we iterate the group law a times on a point P, we get a point aP.
- Input: $a \in \mathbb{Z}$, $P \in E(\mathbb{F}_p)$, Output: $aP \in E(\mathbb{F}_p)$
- This can be implemented efficiently with double-and-add, analogous to square-and-multiply that we saw before.

Elliptic curve scalar multiplication and discrete log

We have a group law $P + Q = R$, usually written as "addition".

Scalar multiplication of points

- If we iterate the group law a times on a point P, we get a point aP.
- Input: $a \in \mathbb{Z}$, $P \in E(\mathbb{F}_p)$, Output: $aP \in E(\mathbb{F}_p)$
- This can be implemented efficiently with double-and-add, analogous to square-and-multiply that we saw before.

Elliptic curve discrete log

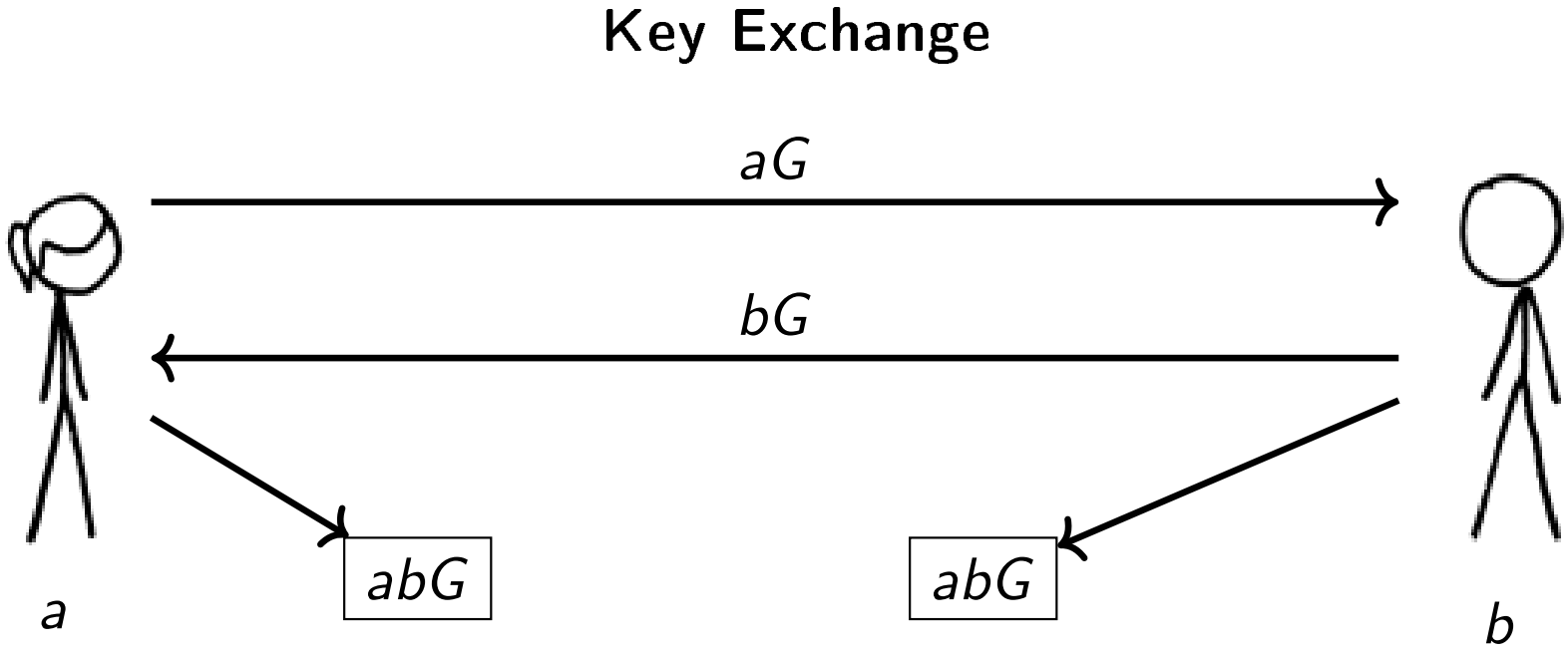
- Input: base point P, target $Q \in E(\mathbb{F}_p)$; output a s.t. $aP = Q$.
- For some families of curves, best algorithms known are generic group algorithms: Pollard rho, baby step giant step, take $O(\sqrt{q})$ time for group order q.
- Broken in polynomial time by a quantum computer.

Elliptic Curve Diffie-Hellman

Public Parameters

E an elliptic curve over $\mathbb{F}_p$

G group generator of order n



ECDSA (Digital Signature Algorithm)

Public Key

E an elliptic curve over $\mathbb{F}_p$

G group generator (base point) on E of order n

$Q = dG$

Private Key

d private key

Verify

$u_1 = H(m)s^{-1} \bmod n$

$u_2 = rs^{-1} \bmod n$

$r \stackrel{?}{=} x\text{-coord of } u_1 G + u_2 Q$

Sign

Generate random k.

$r = x\text{-coordinate of } kG.$

$s = k^{-1}(H(m) + xr) \bmod n$

Output (r, s)

Standardized Elliptic Curves

Curves that withstand all known attacks are more complex to generate than primes, so everyone uses a small number of curves.

NIST P256

- Works over $\mathbb{F}_p$ with $p \approx 2^{256}$.
- Has prime order $q \approx 2^{256}$.
- Best attack: $\sqrt{q}$ time $\approx 2^{128}$.
- Curve parameters generated from deterministic algorithm by opaque seed of unknown generation.

Standardized Elliptic Curves

Curves that withstand all known attacks are more complex to generate than primes, so everyone uses a small number of curves.

NIST P256

- Works over $\mathbb{F}_p$ with $p \approx 2^{256}$.
- Has prime order $q \approx 2^{256}$.
- Best attack: $\sqrt{q}$ time $\approx 2^{128}$.
- Curve parameters generated from deterministic algorithm by opaque seed of unknown generation.

Curve25519

- Edwards curve designed by Daniel J. Bernstein
- Designed to make implementations "secure by default":
  - Simplified group law easier to protect against side-channel attacks
  - Minimal/no point validation required
  - "Twist-secure": Also minimize validation necessary for implementations that only input x-coordinate.

Elliptic curve factoring method

Elliptic curves are useful for factoring too!

Recall $p-1$ method to factor an integer N:

1. Generate composite $M = p_1^{e_1}, \ldots p_k^{e_k}$
2. Check if $\gcd(a^M - 1 \bmod N, N) \neq 1$.

Fermat's little theorem: $a^m \equiv 0 \bmod p$ if $p-1|M$

Finds factors where $p-1|M$, so only works if $p-1$ has only small factors.

Elliptic curve factoring method

Lenstra

1. Generate highly composite $M = p_1^{e_1} \dots p_k^{e_k}$.
2. Choose curve E and point P on E, working modulo N.
3. $Q = MP$.
4. If $\gcd(x(Q), N)$ factors N, done, otherwise repeat for different curves.

Factors N if order of P in $E(\mathbb{F}_p)$ divides M, for factor p of N.

Elliptic curve factoring method

Lenstra

1. Generate highly composite $M = p_1^{e_1} \dots p_k^{e_k}$.
2. Choose curve E and point P on E, working modulo N.
3. $Q = MP$.
4. If $\gcd(x(Q), N)$ factors N, done, otherwise repeat for different curves.

Factors N if order of P in $E(\mathbb{F}_p)$ divides M, for factor p of N.

Theorem

Hasse $p + 1 - 2\sqrt{p} < |E(\mathbb{F}_p)| < p + 1 + 2\sqrt{p}$

In Pollard $p - 1$ method, only one group $\mathbb{Z}_p^*$ of order $p - 1$.

For ECM, can keep choosing random curves with random orders until $|E(\mathbb{F}_p)|$ has small factors.

Running time: $O(e^{\sqrt{2+o(1)}\sqrt{\ln p \ln \ln p}})$. Feasible for 80-bit p.

Extra properties of elliptic curves: Pairings

Some elliptic curve groups have efficiently computable pairings.

We switch to representing the group operation with $\times$ because that is the convention for this application.

A pairing is a map $e : G_0 \times G_1 \to G_T$ that satisfies:

- Bilinear: $e(u \cdot u', v) = e(u, v) \cdot e(u', v)$ and $e(u, v \cdot v') = e(u, v) \cdot e(u, v')$
- Non-degenerate: $e(g_0, g_1) = g_T$ where g_0, g_1, g_T are generators of G_0, G_1, G_T.

- G_0 is an order q subgroup of $E(\mathbb{F}_p)$.
- G_1 is an order q subgroup of $E(\mathbb{F}_{p^d})$.
- G_T is an order q multiplicative subgroup of $\mathbb{F}_{p^d}$.

Neat pairing facts

DDH is false

Let (u, v, w) be our DDH problem: (g_0^a, g_0^b, g_0^c) or (g_0^a, g_0^b, g_0^{ab})

If $e(u, v) = e(g_0, w)$ then

$$e(g_0^a, g_0^b) = e(g_0, g_0)^{ab} \stackrel{?}{=} e(g_0, g_0^c) = e(g_0, g_0)^c.$$

Discrete log in G_0 or G_1 is no harder than in G_T

1. Input $u_0 = g_0^a \in G_0$.
2. Compute $u = e(u_0, g_1)$, $g_T = e(g_0, g_1)$, so $u = g_T^a$.
3. Compute discrete log in G_T, get a.

Application: 3-way Diffie-Hellman

We know how 2-way Diffie-Hellman works:

Alice and Bob exchange messages g^a, g^b and compute g^{ab}.

What about a 3-way exchange?

Application: 3-way Diffie-Hellman

We know how 2-way Diffie-Hellman works:

Alice and Bob exchange messages g^a, g^b and compute g^{ab}.

What about a 3-way exchange?

- Alice, Bob, Charlie exchange g^a, g^b, g^c.
- They want to compute g^{abc}
- Discrete log should remain hard.

If they use pairings:

- Alice computes $e(g^b, g^c)^a = g_T^{bca} = g_T^{abc}$
- Bob computes $e(g^a, g^c)^b = g_T^{acb} = g_T^{abc}$
- Charlie computes $e(g^a, g^b)^c = g_T^{abc}$

Application: 3-way Diffie-Hellman

We know how 2-way Diffie-Hellman works:

Alice and Bob exchange messages g^a, g^b and compute g^{ab}.

What about a 3-way exchange?

- Alice, Bob, Charlie exchange g^a, g^b, g^c.
- They want to compute g^{abc}
- Discrete log should remain hard.

If they use pairings:

- Alice computes $e(g^b, g^c)^a = g_T^{bca} = g_T^{abc}$
- Bob computes $e(g^a, g^c)^b = g_T^{acb} = g_T^{abc}$
- Charlie computes $e(g^a, g^b)^c = g_T^{abc}$

Unsolved: multilinear map/group key exchange for m parties.

Elliptic curves and quantum computers

NSA has been strongly pro elliptic curve cryptography for decades.

In 2015, NSA recommended against transitioning to elliptic curves for people who hadn't already.

Official explanation: Quantum computers are coming, so people should wait for post-quantum crypto.

Plausible explanation:

- Smaller key sizes for ECC mean fewer qubits required to break than 3072-bit factoring.

Dual EC DRBG

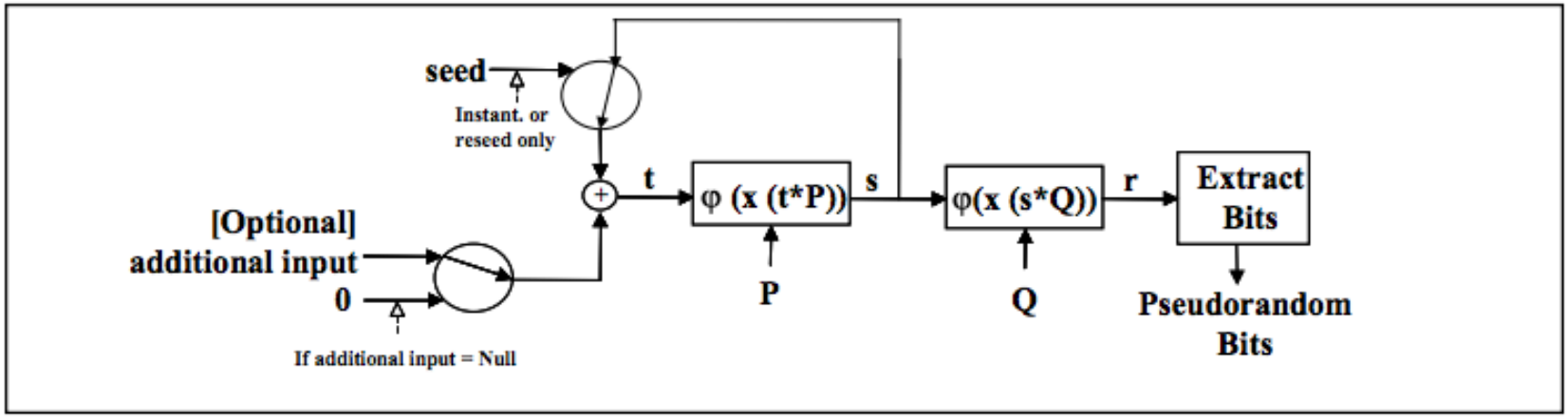


Figure 13: Dual_EC_DRBG

- Parameters: Pre-specified elliptic curve points P and Q.
- Seed: 32-byte integer s
- State: x-coordinate of point sP. ($\phi(x(sP))$ above.)
- Update: $t = s \oplus$ optional additional input. State $s = x(tP)$.
- Output: At state s, compute x-coordinate of point $x(sQ)$, discard top 2 bytes, output 30 bytes.

Dual EC DRBG History

- Early 2000s: Created by the NSA and pushed towards standardization
- 2004: Published as part of ANSI X9.82 part 3 draft
- 2004: RSA makes Dual EC the default PRNG in BSAFE
- 2005: Standardized in NIST SP 800-90 draft
- 2007: Shumow and Ferguson demonstrate theoretical backdoor
- 2013: Snowden documents lead to renewed interest in Dual EC
- 2014: Practical attacks on TLS using Dual EC demonstrated
- 2015: NIST removes Dual EC from list of approved PRNGs

Shumow and Ferguson 2007

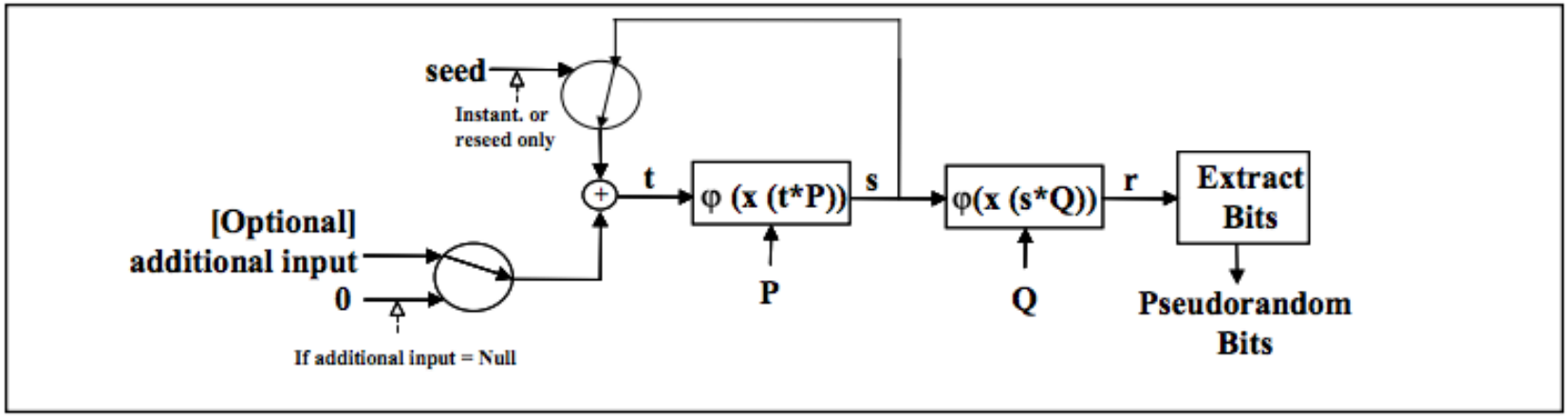


Figure 13: Dual_EC_DRBG

1. Assume attacker controls standard and constructs points with known relationship $P = dQ$.
2. Attacker gets 30 bytes of x-coordinate of sQ. Attacker brute forces 2^{16} MSBs, gets 2^{17} possible y-coordinates, ends up with 2^{15} candidates for sQ.
3. For each candidate sQ attacker computes $dsQ = sP$ and compares to next output.

September 2013: NSA Bullrun in NY Times

- (TS//SI//REL TO USA, FVEY) Insert vulnerabilities into commercial encryption systems, IT systems, networks, and endpoint communications devices used by targets.
- (TS//SI//REL TO USA, FVEY) Collect target network data and metadata via cooperative network carriers and/or increased control over core networks.
- (TS//SI//REL TO USA, FVEY) Leverage commercial capabilities to remotely deliver or receive information to and from target endpoints.
- (TS//SI//REL TO USA, FVEY) Exploit foreign trusted computing platforms and technologies.
- (TS//SI//REL TO USA, FVEY) Influence policies, standards and specification for commercial public key technologies.
- (TS//SI//REL TO USA, FVEY) Make specific and aggressive investments to facilitate the development of a robust exploitation capability against Next-Generation Wireless (NGW) communications.

Dual EC Attack Complexity in TLS Implementations

Checkoway et al. 2014

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	20	$2^{15}(C_v + 3C_f) + 2^{20}(2C_f)$	0.02
OpenSSL-fixed III**				32	$35 + k$	$2^{15}(C_v + 3C_f) + 2^{35+k}(2C_f)$	$2^k \cdot 83.32$

* 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 grugq
@thegrugq

Follow

Woah! Juniper discovers a backdoor to decrypt VPN traffic (and remote admin) has been inserted into their OS source

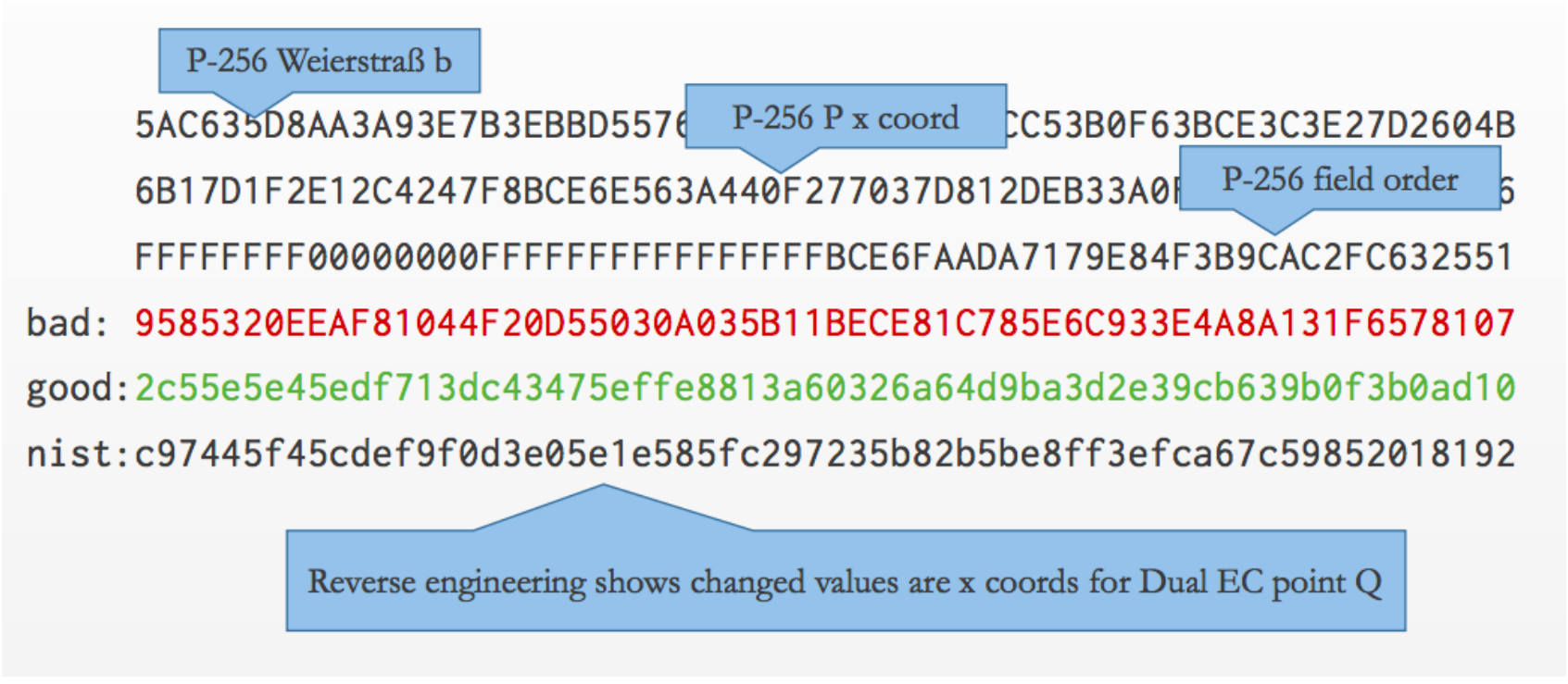
Important Announcement about ScreenOS®

IMPORTANT JUNIPER SECURITY ANNOUNCEMENT CUSTOMER UPDATE: DECEMBER 20, 2015 Administrative Access (CVE-2015-7755) only affects ScreenOS 6.3.0r17 through

forums.juniper.net

Diff of VPN code change

Checkoway et al. 2016



Juniper cascaded Dual EC with ANSI X9.31

- ScreenOS only FIPS validated for ANSI X9.31, not Dual EC
- Juniper used non-default points for Dual EC

The following product families do utilize Dual_EC_DRBG, but do not use the pre-defined points cited by NIST:

1. ScreenOS*

* ScreenOS does make use of the Dual_EC_DRBG standard, but is designed to not use Dual_EC_DRBG as its primary random number generator. ScreenOS uses it in a way that should not be vulnerable to the possible issue that has been brought to light. Instead of using the NIST recommended curve points it uses self-generated basis points and then takes the output as an input to FIPS/ANSI X.9.31 PRNG, which is the random number generator used in ScreenOS cryptographic operations.

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();     // conditional reseed
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)    // generate Dual EC output
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24); // copy output
  prng_output_index = 32;
}
```

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block); // gen output
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;      // global variable
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())       // always true
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)     // global variable
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;   // set to 32
}
```

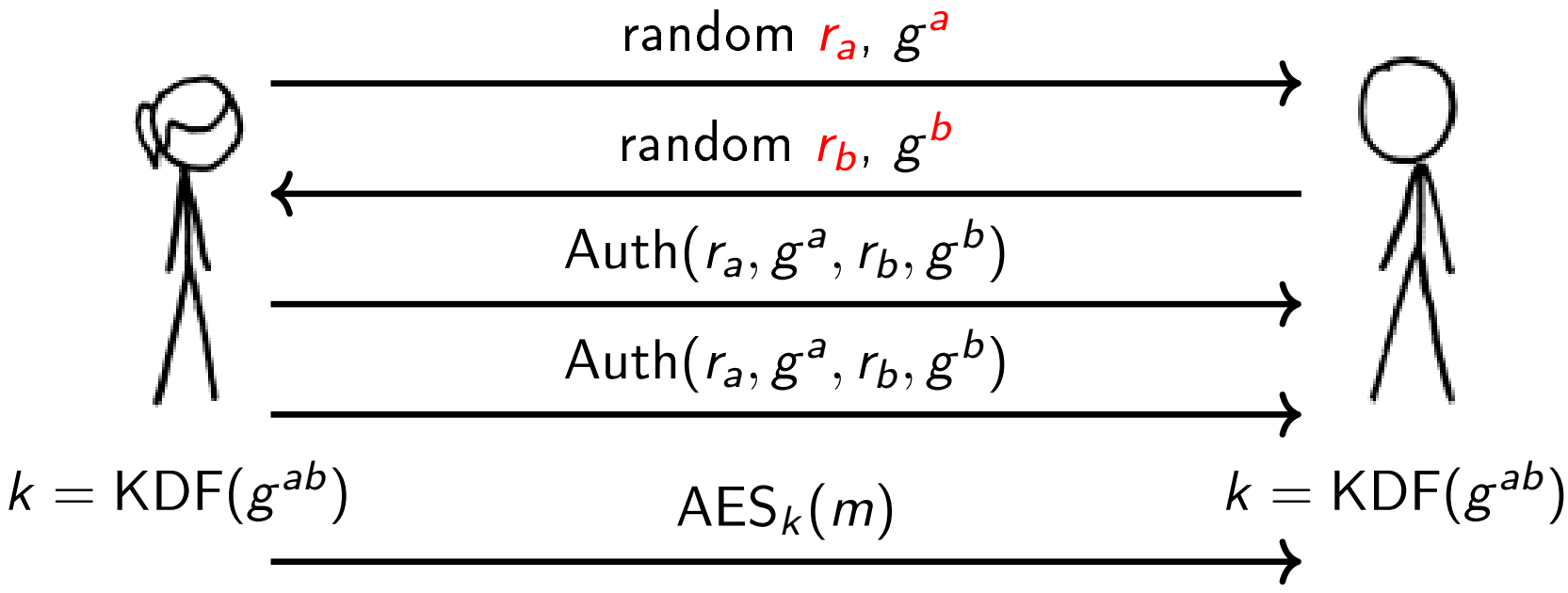
ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) { // never runs
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8); // reuses buffer
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }  // output is raw Dual EC output!
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

Passive state recovery in ScreenOS IPsec



- Use random nonces to carry out state recovery attack.
- ScreenOS used 32-byte nonce $\implies$ efficient attack.
- After state recovered, then recover secret exponents.
- We demonstrated attack with our own backdoored P, Q.

ScreenOS Version History

ScreenOS 6.1.0r7

- ANSI X9.31
- Seeded by interrupts
- Reseed every 10k calls
- 20-byte IKE nonces

ScreenOS 6.2.0r0 (2008)

- Dual EC $\rightarrow$ ANSI X9.31
- Reseed bug exposes raw Dual EC
- Reseed every call
- Nonces generated before keys
- 32-byte IKE nonces

- Attacker changed constant in 6.2.0r15 (2012).
- But passive decryption enabled in earlier release.
- Juniper's "fix" was to reinstate original Q value. After our work they removed Dual EC completely.

Discussion and Lessons

- "NOBUS" backdoors can be repurposed.
- Don't know how Juniper's parameters were generated, or who wrote their Dual EC cascade.
- Juniper wasn't certified for Dual EC, so it wasn't on the radar of researchers who looked for vulnerable implementations. Who else are we missing?
- Could we detect both implementations and bugs automatically?
- How do we prevent backdoors in standards?