THE UNIVERSITY OF ILLINOIS AT CHICAGO

A Systematic Analysis of the Juniper Dual EC Incident

Stephen Checkoway

With Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney,
Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, Hovav Shacham

Juniper's surprising announcement

PROBLEM:
During an internal code review, two security issues were identified.

Administrative Access (CVE-2015-7755) allows **unauthorized remote administrative access** to the device. Exploitation of this vulnerability can lead to complete compromise of the affected device.

VPN Decryption (CVE-2015-7756) may allow a **knowledgeable attacker who can monitor VPN traffic to decrypt that traffic**. It is independent of the first issue.

https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713

Affected devices and firmware

- Juniper's *Secure Services Gateway* firewall/VPN appliances
- Various revisions of *ScreenOS* 6.2 and 6.3

Administrative access backdoor

```
                LDR         R0, =aSCtUUnSSipSDip ; ">>> %s(ct=%u, un='%s', sip=%s, dip=%s, "...
                LDR         R1, =aAuth_admin_int ; "auth_admin_internal"
                BL          log

backdoor_authentication                 ; CODE XREF: auth_admin_internal+2C↑j
                ADD         R0, R5, #0x44
                LDR         R1, =aSUnSU ; "<<< %s(un='%s') = %u"
                BL          strcmp
                CMP         R0, #0
                BNE         loc_13DC78
                MOV         R0, #0xFFFFFFFD
                LDMDB       R11, {R4-R8,R11,SP,PC}
```

- Extra check inserted in `auth_admin_internal` for hardcoded admin password: `<<< %s(un=‘%s’) = %u`
- Works with both SSH and Telnet
- Analysis by HD Moore

VPN decryption

- Juniper's bulletin is a bit vague: ***knowledgeable attacker*** ?
- The first hint comes from a strings diff between an affected version and its corresponding fix

```
 FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
 FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
 5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
 6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
 FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
-9585320EEAF81044F20D55030A035B11BECE81C785E6C933E4A8A131F6578107
+2C55E5E45EDF713DC43475EFFE8813A60326A64D9BA3D2E39CB639B0F3B0AD10
```

- Almost the entire difference

VPN decryption

P-256 parameters in short Weierstrass form
$y^2 = x^3 + ax + b \pmod{p}$ with generator $P = (P_x, P_y)$:
p, $a = -3 \pmod{p}$, b, P_x, and P-256 group order n

```
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
-9585320EEAF81044F20D55030A035B11BECE81C785E6C933E4A8A131F6578107
+2C55E5E45EDF713DC43475EFFE8813A60326A64D9BA3D2E39CB639B0F3B0AD10
```

VPN decryption

P-256 parameters in short Weierstrass form
$y^2 = x^3 + ax + b \pmod{p}$ with generator $P = (P_x, P_y)$:
p, $a = -3 \pmod{p}$, b, P_x, and P-256 group order n

```
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
-9585320EEAF81044F20D55030A035B11BECE81C785E6C933E4A8A131F6578107
+2C55E5E45EDF713DC43475EFFE8813A60326A64D9BA3D2E39CB639B0F3B0AD10
```

Via reverse engineering: nonstandard x-coordinate of Dual EC point Q

Dual EC DRBG timeline

- 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 CSPRNG in BSAFE (for $10MM)
- 2005: Standardized in NIST SP 800-90
- 2007: Shumow and Ferguson demonstrate a theoretical backdoor attack
- 2013: Snowden documents lead to renewed interest in Dual EC
- 2014: Practical attacks on TLS using Dual EC demonstrated
- 2014: NIST removes Dual EC from list of approved PRNGs
- 2016: Practical attacks on IKE using Dual EC (this work)

A backdoored PRNG

s_k — Internal PRNG states

r_k — Outputs

$f(\bullet)$ — State update function

$g(\bullet)$ — Output function

$h(\bullet)$ — Backdoor function

■ — Attacker computation

s_0

A backdoored PRNG

s_k — Internal PRNG states

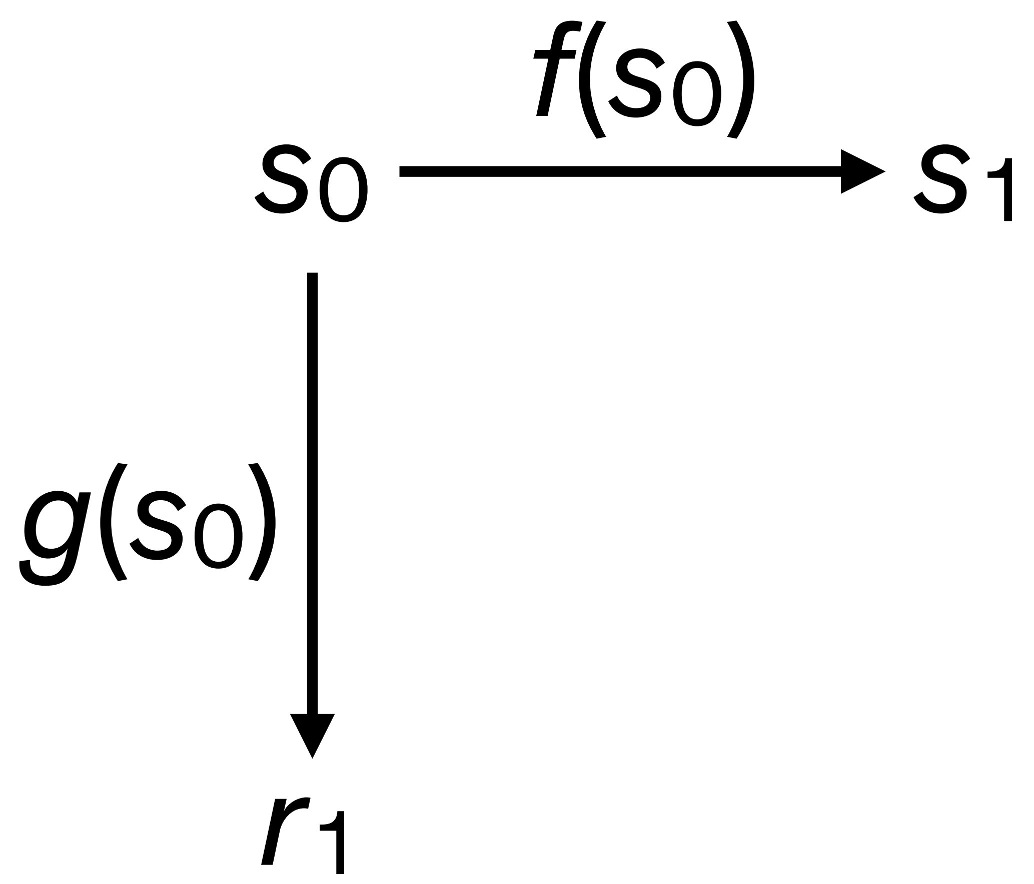
r_k — Outputs

$f(\bullet)$ — State update function

$g(\bullet)$ — Output function

$h(\bullet)$ — Backdoor function

■ — Attacker computation



A backdoored PRNG

s_k — Internal PRNG states

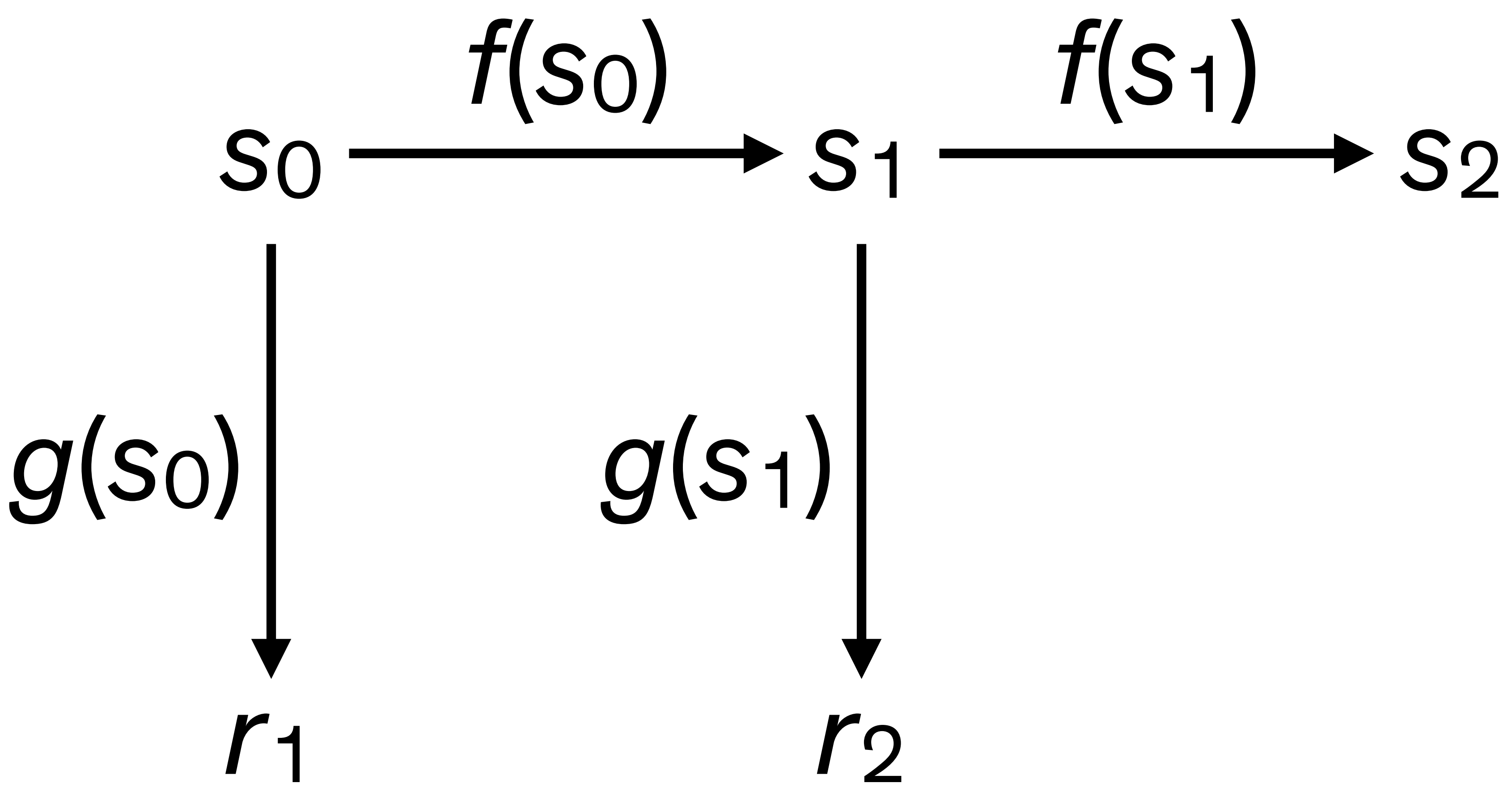
r_k — Outputs

$f(\bullet)$ — State update function

$g(\bullet)$ — Output function

$h(\bullet)$ — Backdoor function

■ — Attacker computation



A backdoored PRNG

s_k — Internal PRNG states

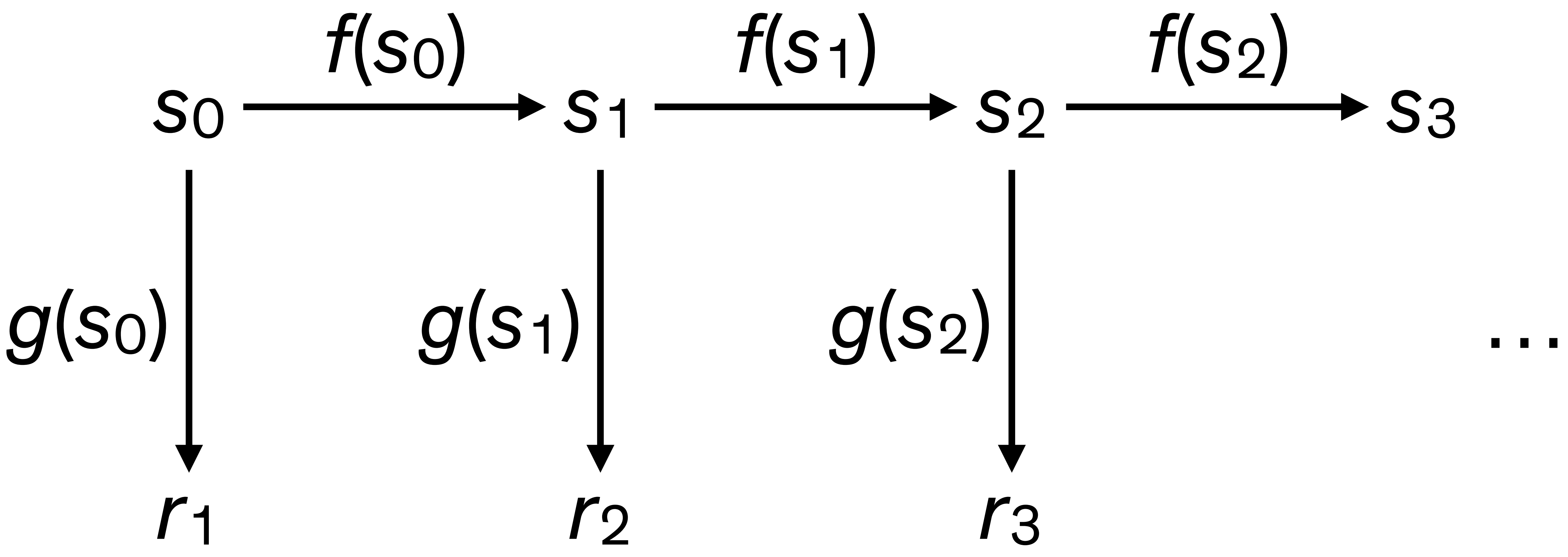
r_k — Outputs

$f(\bullet)$ — State update function

$g(\bullet)$ — Output function

$h(\bullet)$ — Backdoor function

■ — Attacker computation



A backdoored PRNG

s_k — Internal PRNG states

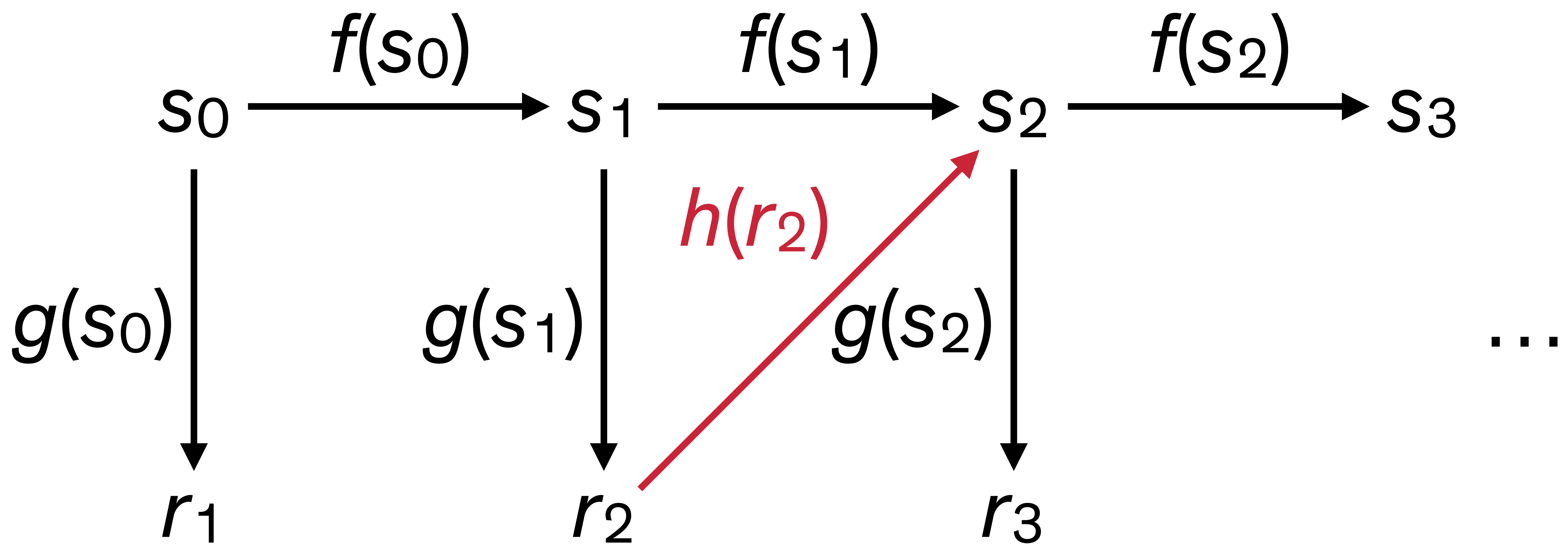
r_k — Outputs

$f(\bullet)$ — State update function

$g(\bullet)$ — Output function

$h(\bullet)$ — Backdoor function

■ — Attacker computation



A backdoored PRNG

s_k — Internal PRNG states

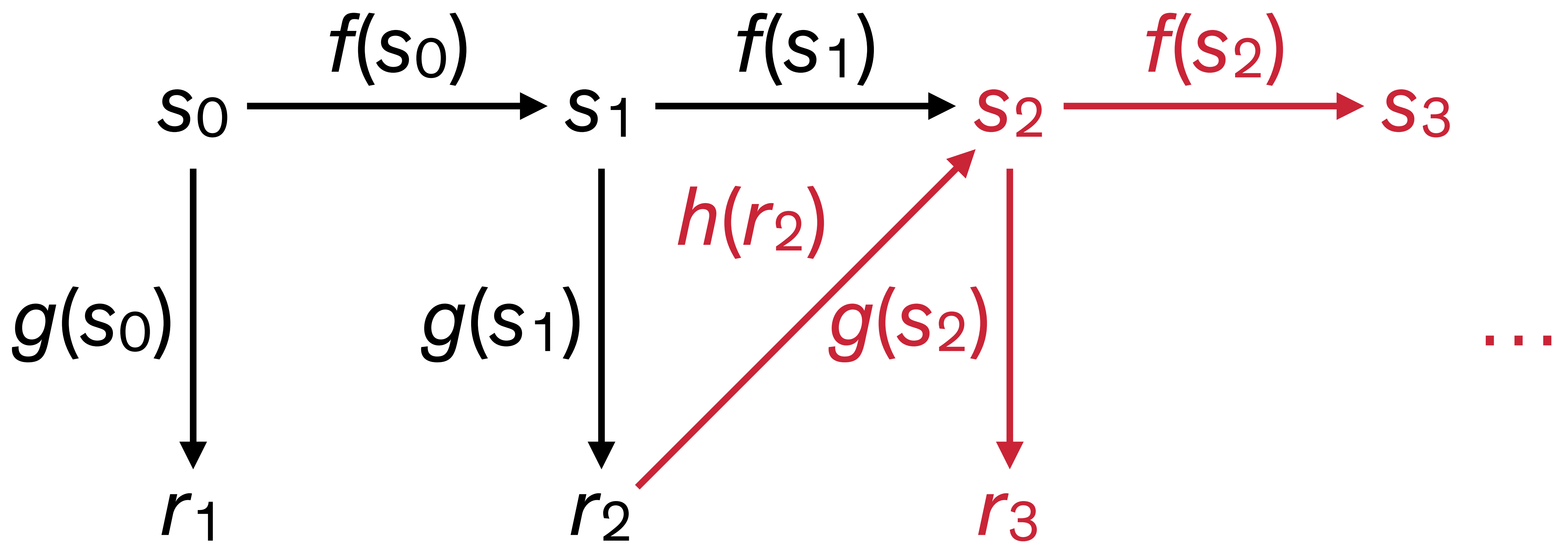
r_k — Outputs

$f(\bullet)$ — State update function

$g(\bullet)$ — Output function

$h(\bullet)$ — Backdoor function

■ — Attacker computation



Elliptic curve primer

- Points on an elliptic curve are pairs (x, y)
- x and y are 32-byte integers (for the curve we care about here)
- Points can be added together to get another point on the curve
- Scalar multiplication: Given integer n and point P,
  $nP = P + P + \ldots + P$ is easy to compute
- Given points P and nP, n is hard to compute (elliptic curve discrete logarithm problem)

Dual EC operation (simplified)

s_0

32-byte states

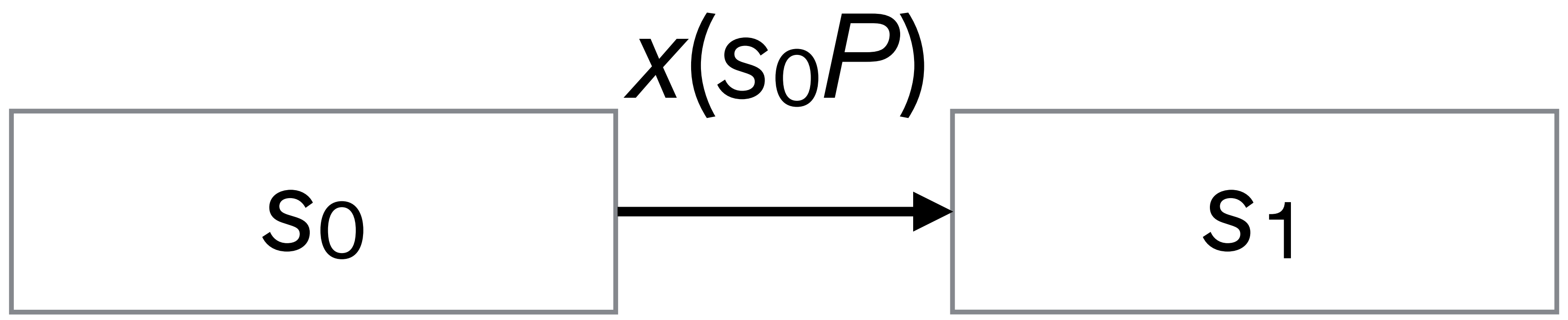
P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*

output

Dual EC operation (simplified)

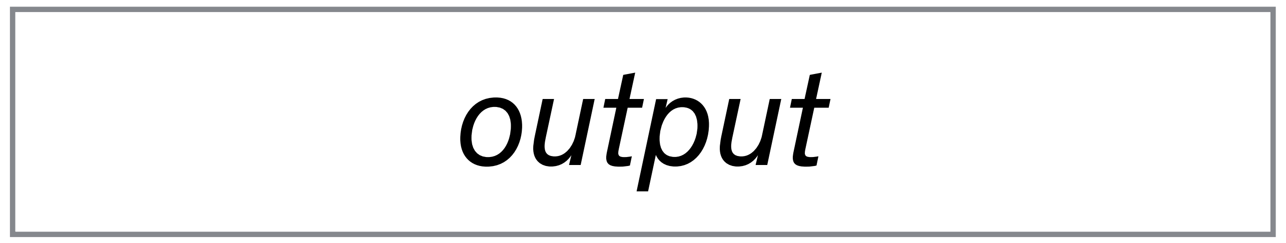


32-byte states

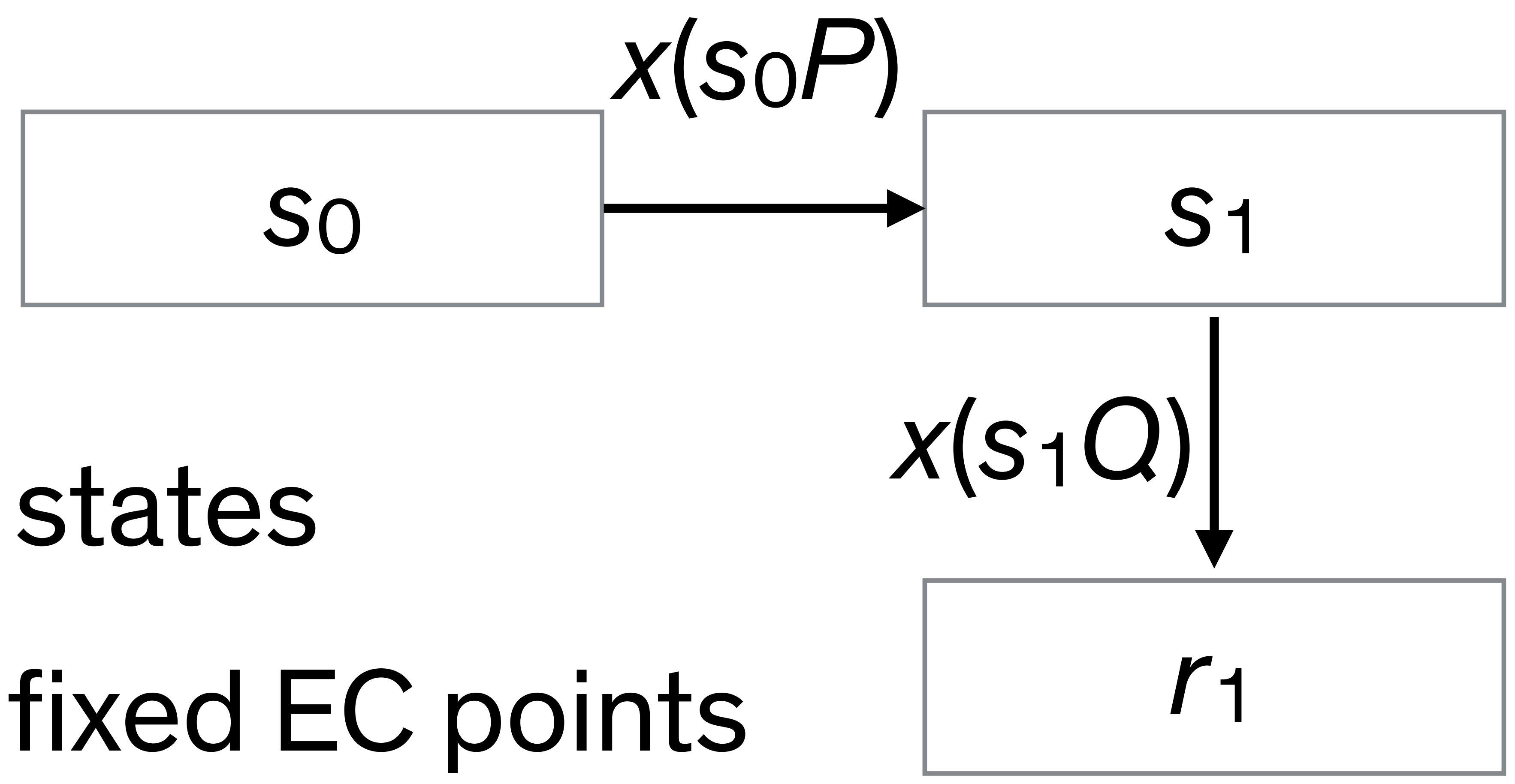
P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*



Dual EC operation (simplified)

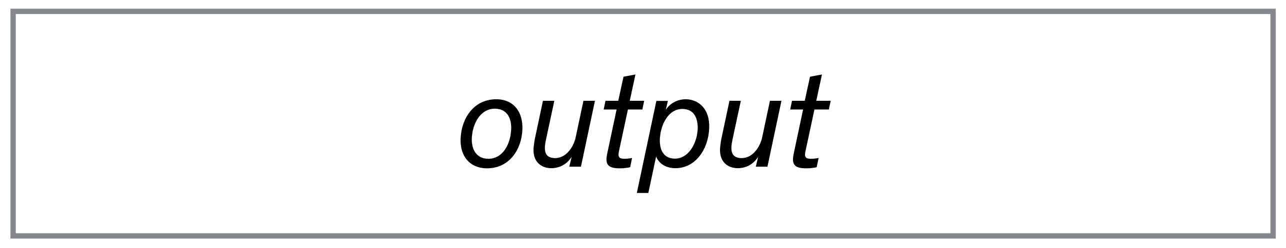


32-byte states

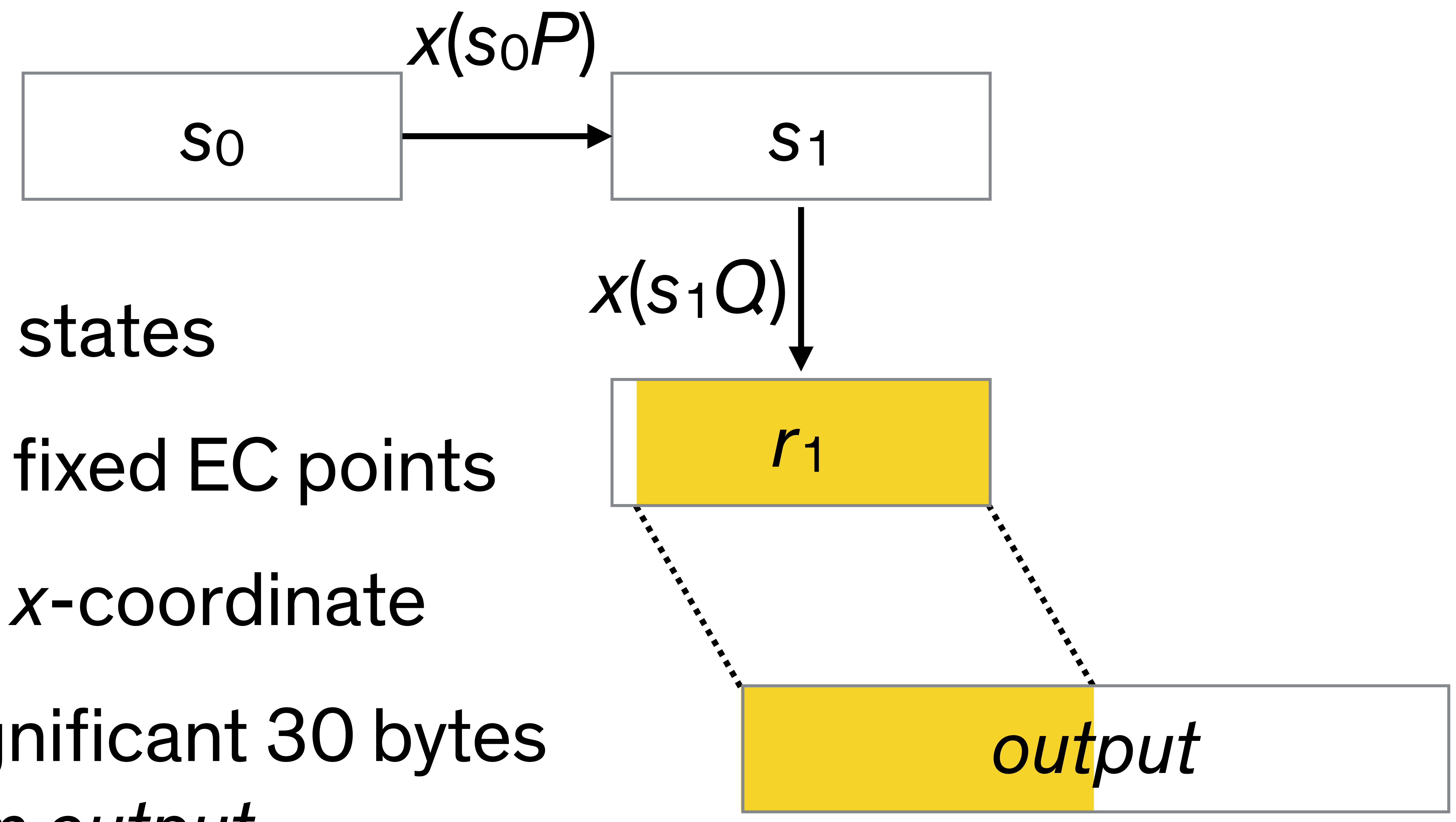
P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*



Dual EC operation (simplified)



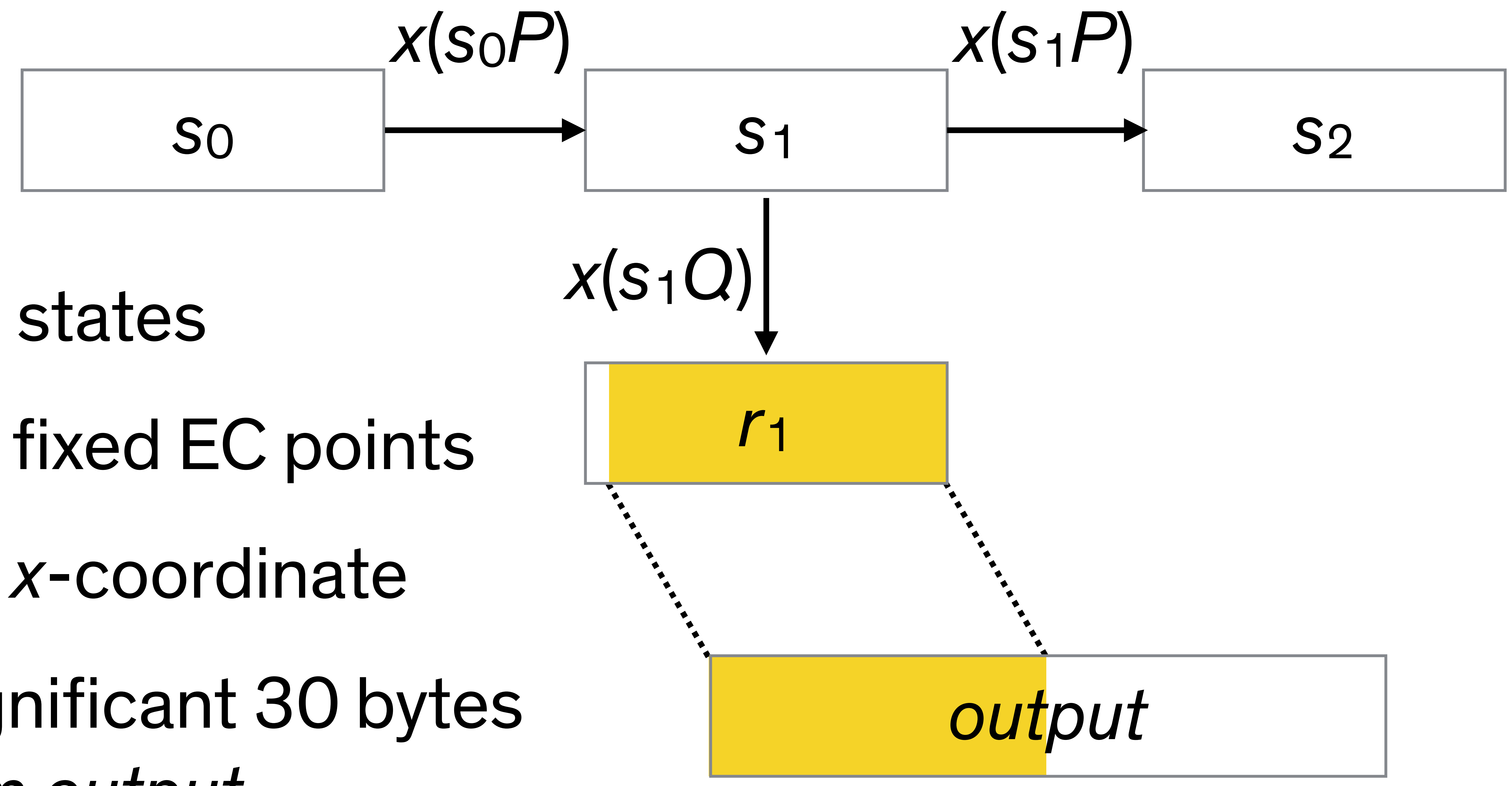
32-byte states

P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*

Dual EC operation (simplified)



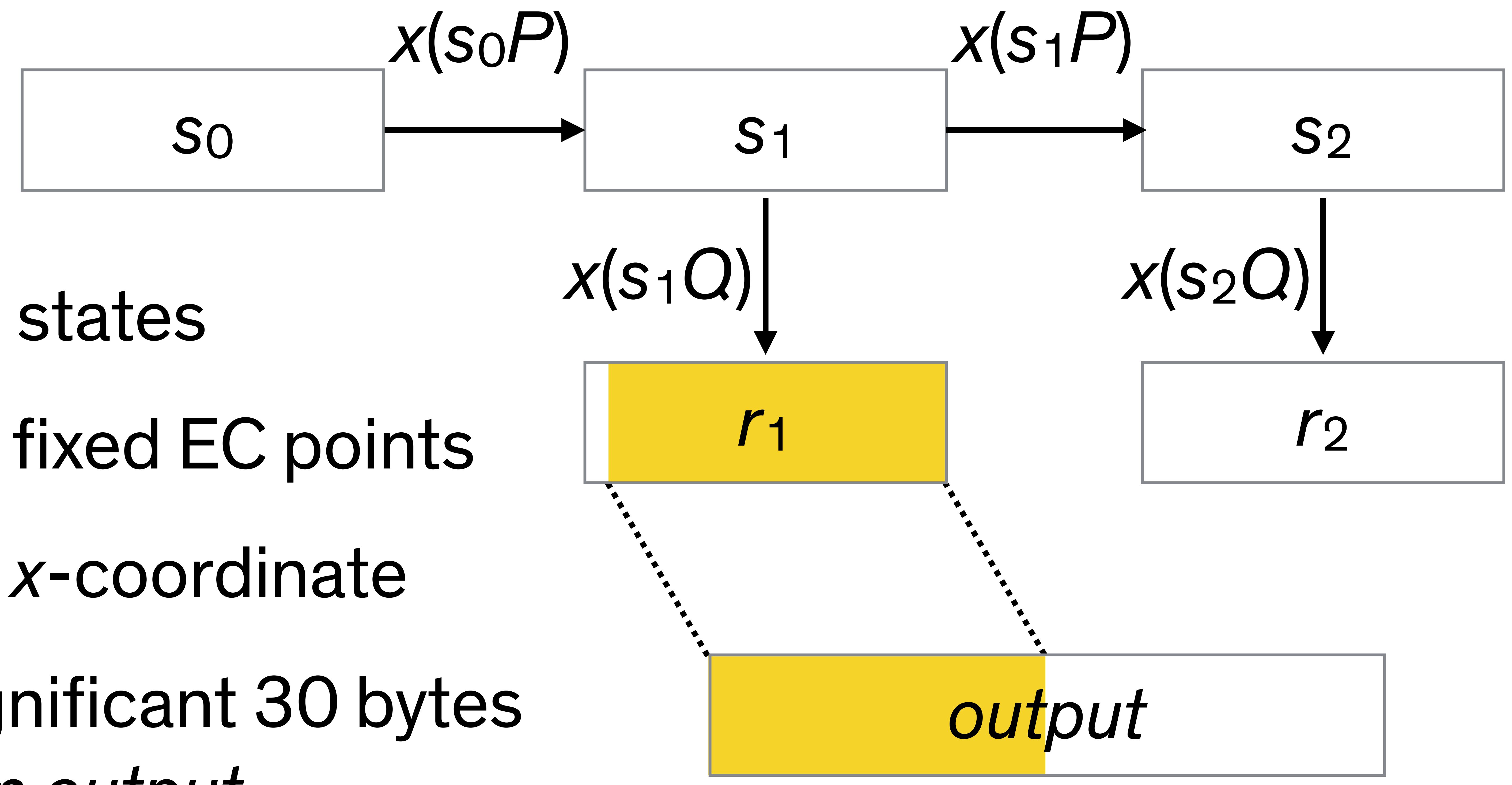
32-byte states

P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*

Dual EC operation (simplified)



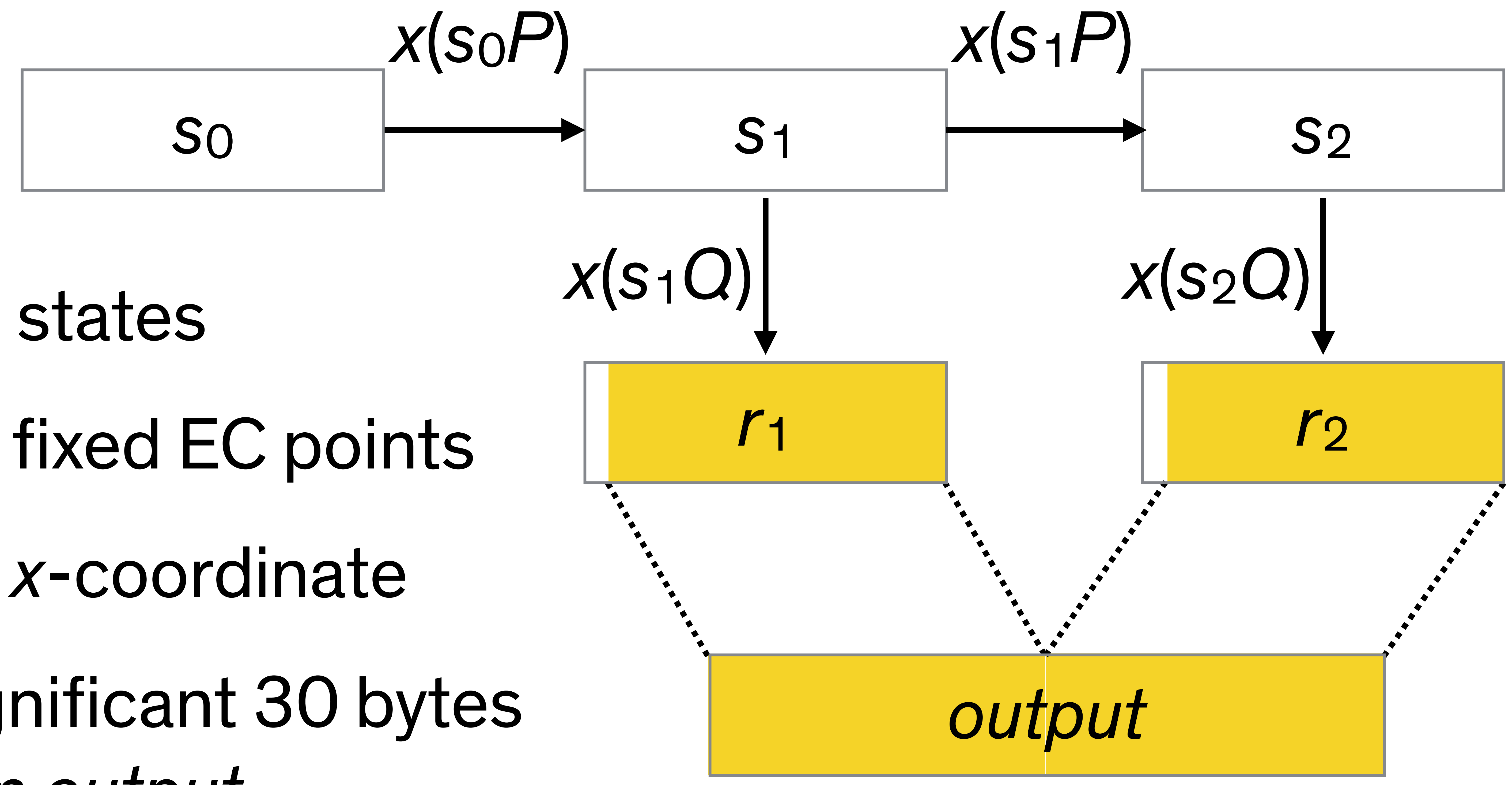
32-byte states

P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*

Dual EC operation (simplified)



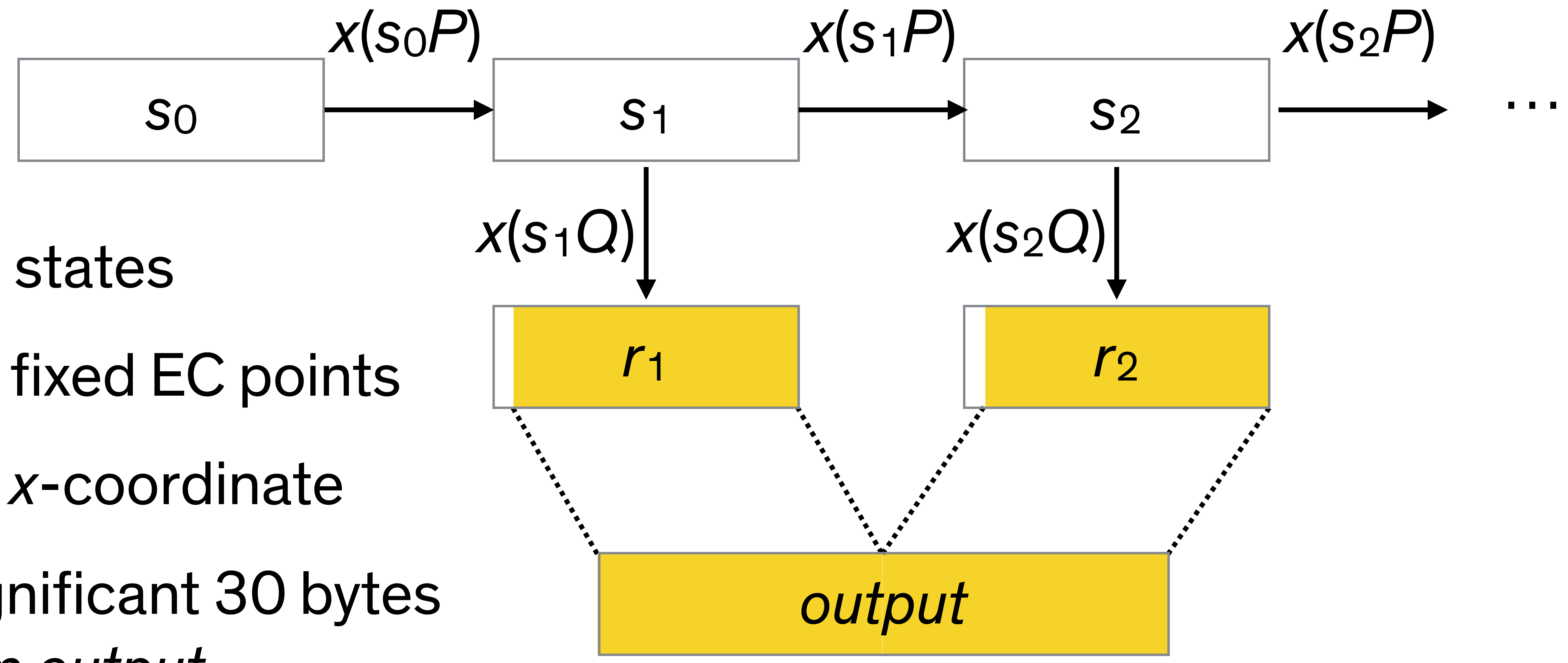
32-byte states

P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*

Dual EC operation (simplified)



32-byte states

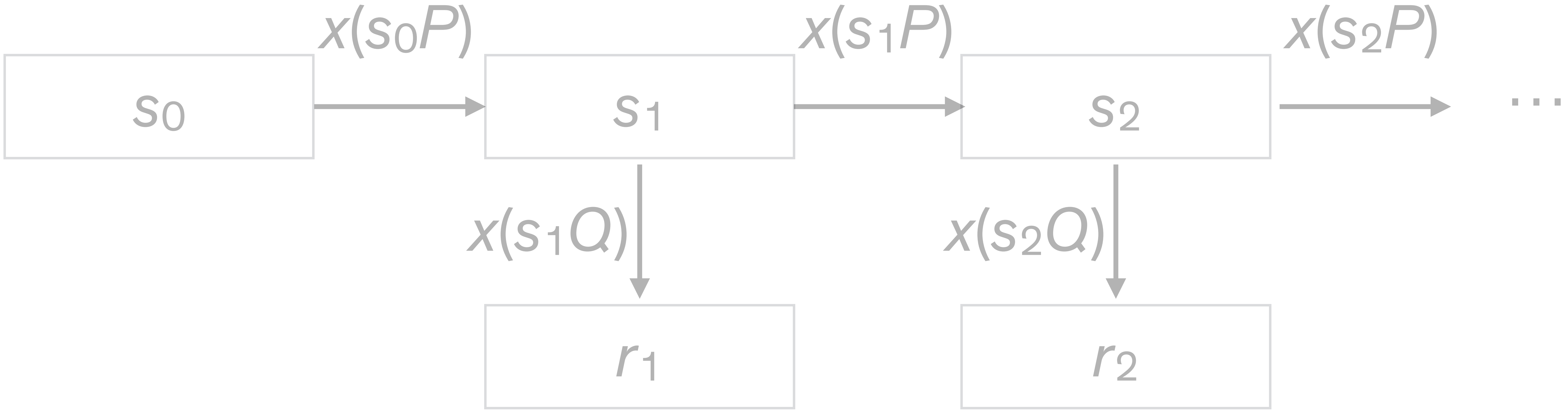
P, Q — fixed EC points

$x(\bullet)$ — x-coordinate

least significant 30 bytes of r_i form *output*

Shumow–Ferguson attack

Assumes attacker knows the integer d such that $P = dQ$

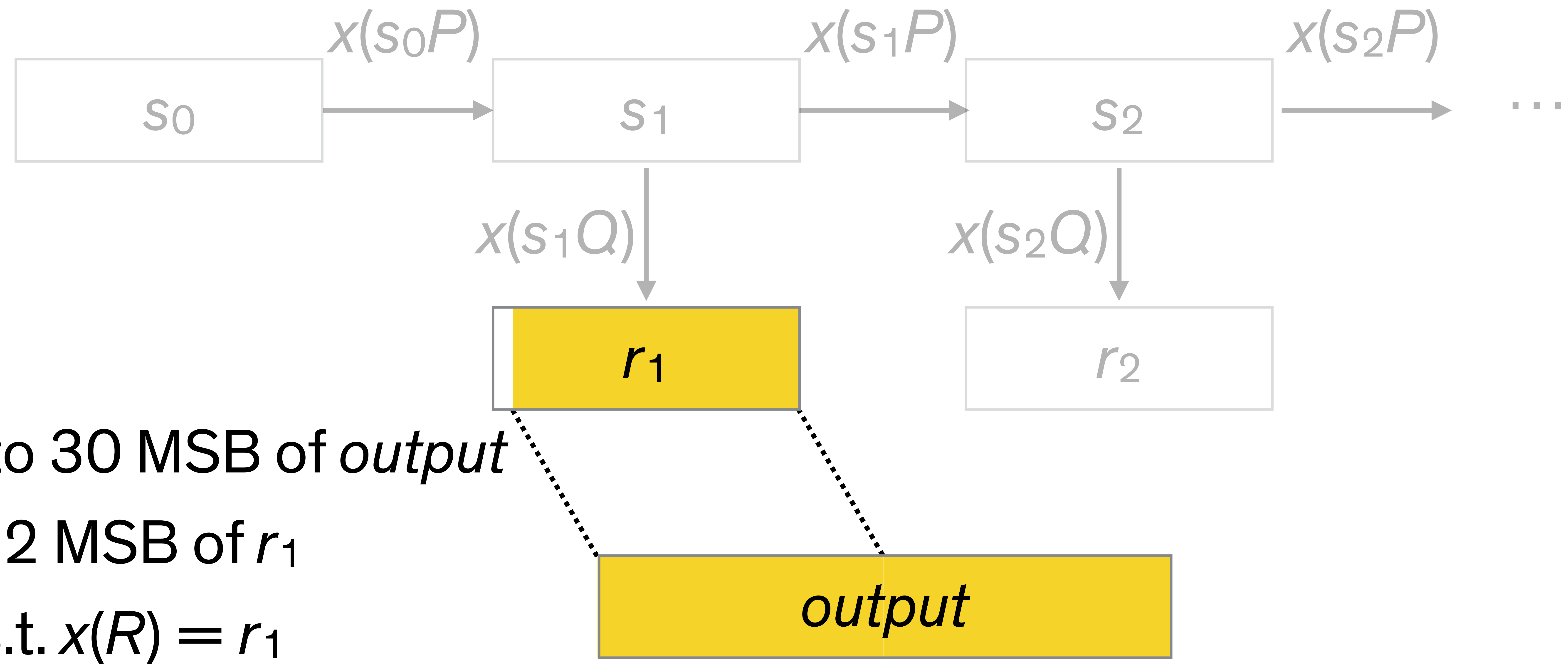


1. Set r_1 to 30 MSB of *output*
2. Guess 2 MSB of r_1
3. Let R s.t. $x(R) = r_1$
4. Compute $s_2 = x(s_1P) = x(s_1dQ) = x(ds_1Q) = x(dR)$
5. Compute r_2 and compare with *output*; goto 2 if they differ

output

Shumow–Ferguson attack

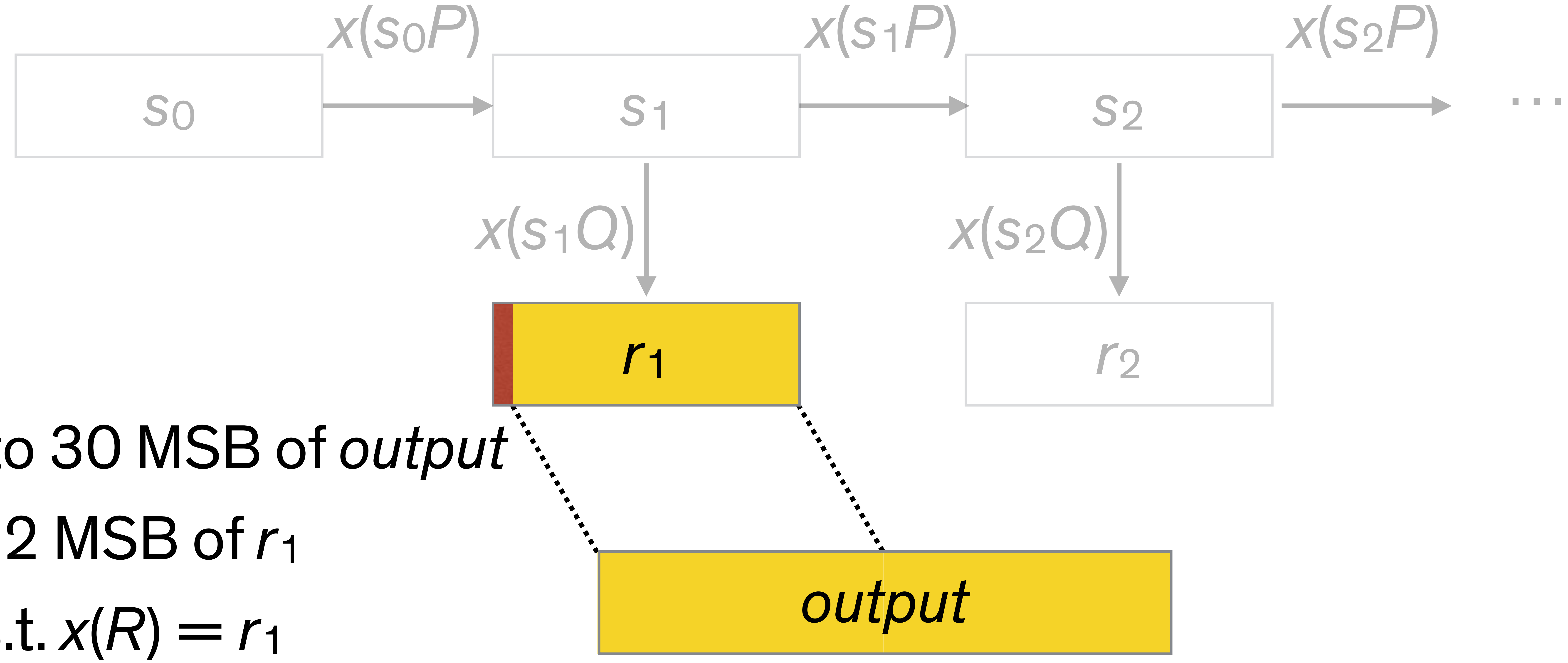
Assumes attacker knows the integer d such that $P = dQ$



1. Set r_1 to 30 MSB of *output*
2. Guess 2 MSB of r_1
3. Let R s.t. $x(R) = r_1$
4. Compute $s_2 = x(s_1P) = x(s_1dQ) = x(ds_1Q) = x(dR)$
5. Compute r_2 and compare with *output*; goto 2 if they differ

Shumow–Ferguson attack

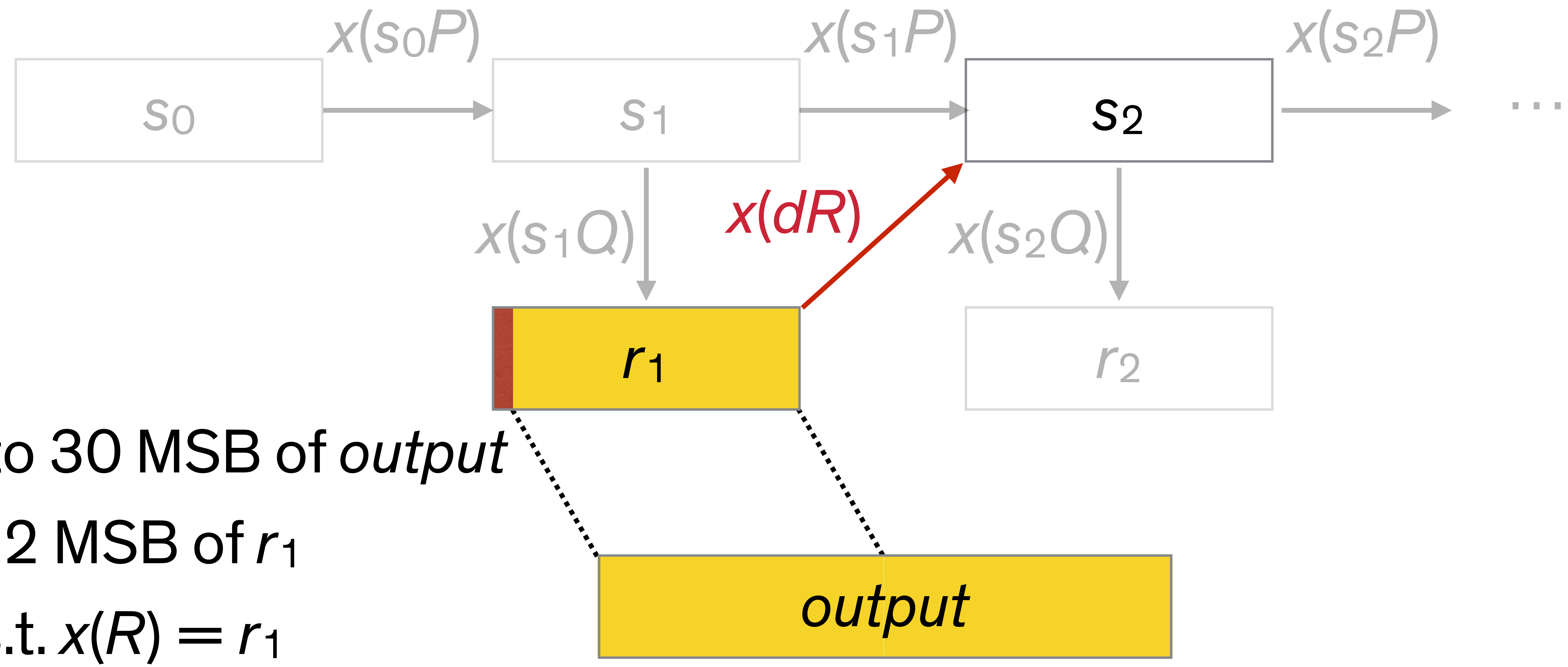
Assumes attacker knows the integer d such that $P = dQ$



1. Set r_1 to 30 MSB of *output*
2. Guess 2 MSB of r_1
3. Let R s.t. $x(R) = r_1$
4. Compute $s_2 = x(s_1P) = x(s_1dQ) = x(ds_1Q) = x(dR)$
5. Compute r_2 and compare with *output*; goto 2 if they differ

Shumow–Ferguson attack

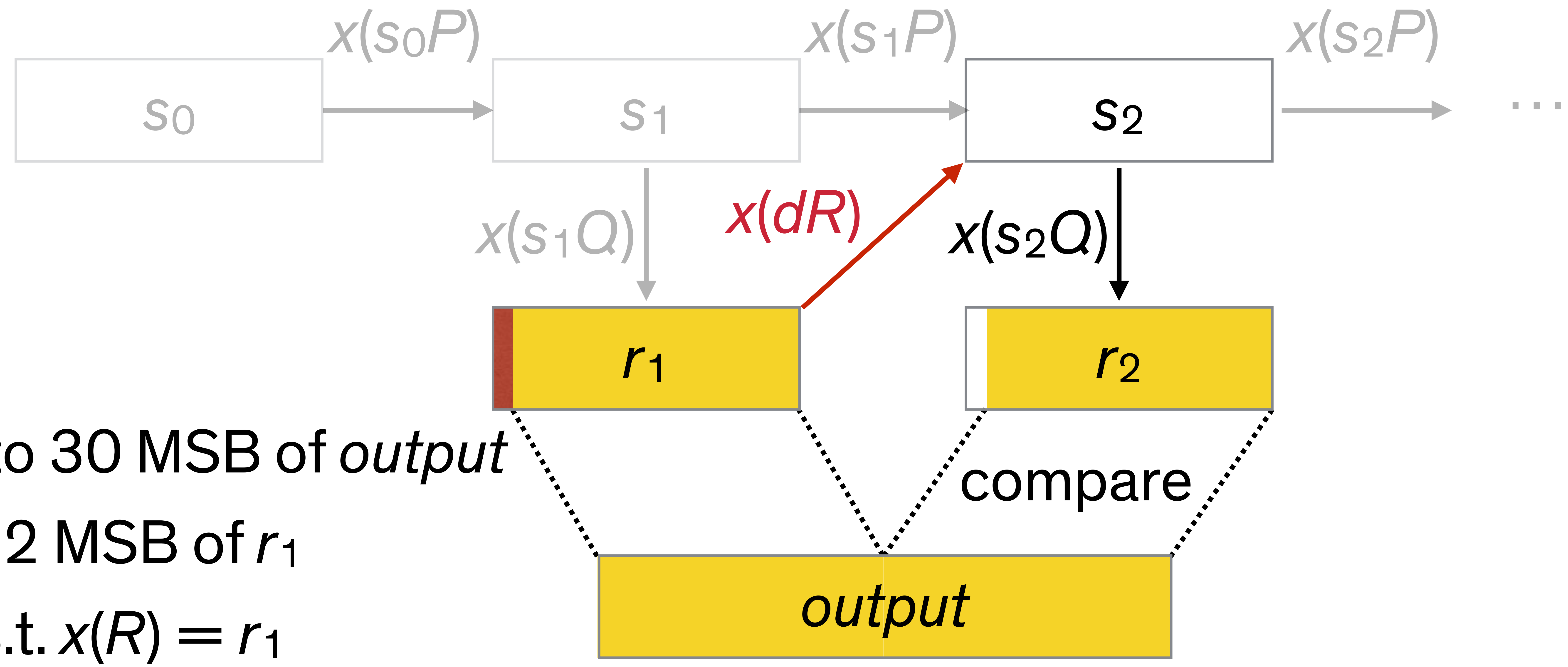
Assumes attacker knows the integer d such that $P = dQ$



1. Set r_1 to 30 MSB of *output*
2. Guess 2 MSB of r_1
3. Let R s.t. $x(R) = r_1$
4. Compute $s_2 = x(s_1P) = x(s_1dQ) = x(ds_1Q) = x(dR)$
5. Compute r_2 and compare with *output*; goto 2 if they differ

Shumow–Ferguson attack

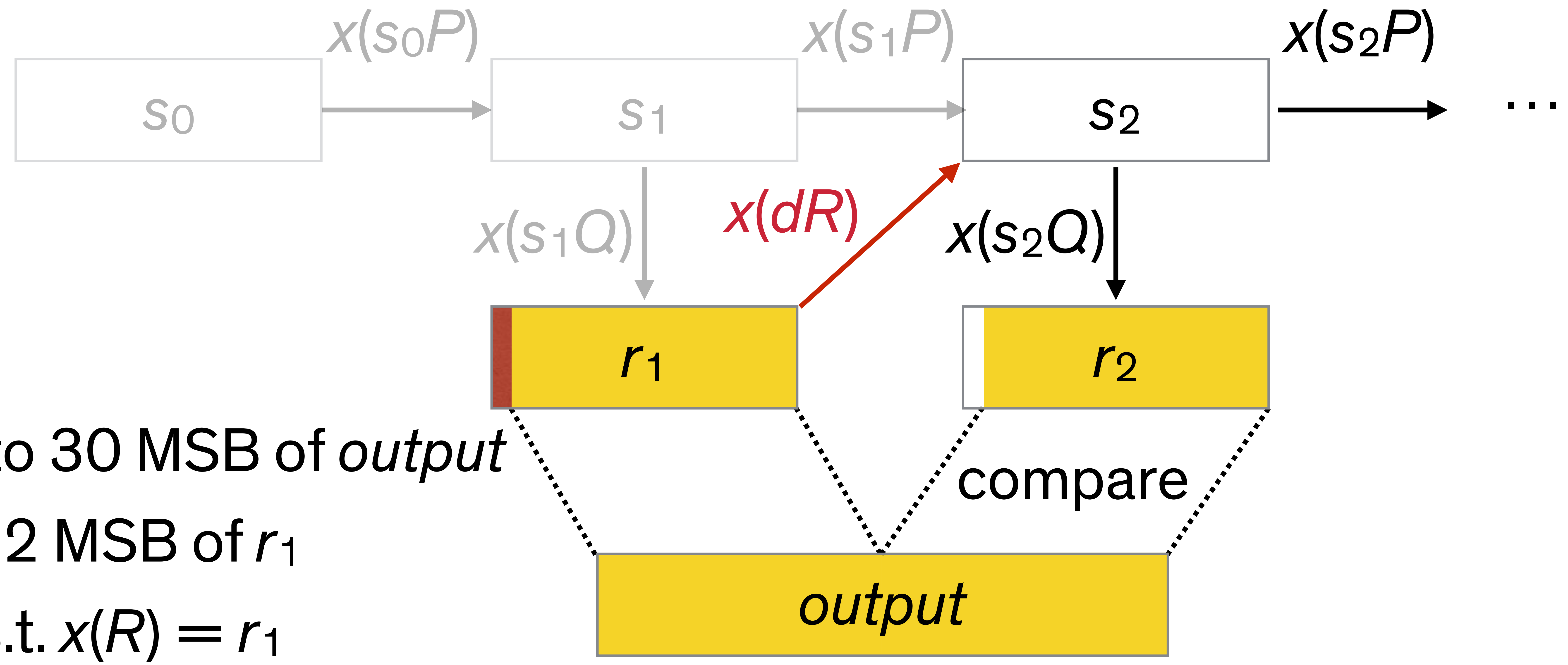
Assumes attacker knows the integer d such that $P = dQ$



1. Set r_1 to 30 MSB of *output*
2. Guess 2 MSB of r_1
3. Let R s.t. $x(R) = r_1$
4. Compute $s_2 = x(s_1P) = x(s_1dQ) = x(ds_1Q) = x(dR)$
5. Compute r_2 and compare with *output*; goto 2 if they differ

Shumow–Ferguson attack

Assumes attacker knows the integer d such that $P = dQ$



1. Set r_1 to 30 MSB of *output*
2. Guess 2 MSB of r_1
3. Let R s.t. $x(R) = r_1$
4. Compute $s_2 = x(s_1 P) = x(s_1 d Q) = x(d s_1 Q) = x(dR)$
5. Compute r_2 and compare with *output*; goto 2 if they differ

Shumow–Ferguson attack prereqs

Attacker needs to see

1. Most (e.g., $\geq$ 28 bytes) of r_k for some k
2. Some public function of "enough" of the following output

For example, a network protocol that sends

1. a $\geq$ 28-byte *nonce*; and
2. a Diffie–Hellman public key g^x

over the wire where the *nonce* is generated before x is vulnerable

Methods of learning $d = \log_Q P$

Reminder: The backdoor function involves a multiplication by $d = \log_Q P$

Methods:

1. Solve the discrete logarithm problem
2. Pick official point Q by selecting a large integer e and set $Q = eP$
   Then $d = e^{-1}$ (mod group order n)
3. Use nonstandard point Q' generated as in 2
4. Gain access to third party source code and substitute your own nonstandard Q' generated as in 2

Methods of learning $d = \log_Q P$

Reminder: The backdoor function involves a multiplication by $d = \log_Q P$

Methods:

1. Solve the discrete logarithm problem **(too hard)**
2. Pick official point Q by selecting a large integer e and set $Q = eP$
   Then $d = e^{-1}$ (mod group order n) **(NSA picked Q, but how?)**
3. Use nonstandard point Q' generated as in 2 **(ScreenOS does this)**
4. Gain access to third party source code and substitute your own nonstandard Q' generated as in 2 **(Juniper incident)**

What did Juniper's ***knowledgable* attacker** know? The discrete log d

Oct. 2013 Knowledge Base article

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.

https://web.archive.org/web/20150220051616/https://kb.juniper.net/InfoCenter/index?page=content&id=KB28205

Research questions

1. Why doesn't the use of X9.31 defend against a compromised *Q*?
2. Why does a change in *Q* result in passive VPN decryption?
3. What is the history of the ScreenOS PRNG code?
4. Are the versions of ScreenOS with Juniper's *Q* vulnerable to attack?
5. How was Juniper's *Q* generated?

Forensic reverse engineering

- We draw on a body of released firmware revisions to answer some research questions
  1. ANSI X9.31 doesn't help
  2. Changing $Q \implies$ VPN decryption
  3. History of ScreenOS PRNG
- Need other materials to answer
  4. Is Juniper's Q vulnerable
  5. How Juniper's Q is generated

Device series	Architecture	Version	Revisions
SSG-500	x86	6.3.0	12b
SSG-5/ SSG-20	ARM-BE	5.4.0	1–3, 3a, 4–16
		6.0.0	1–5, 5a, 6–8, 8a
		6.1.0	1–7
		6.2.0	1–8, 19
		6.3.0	1–6

UIC

ScreenOS 6.2 PRNG

```c
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```c
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

ScreenOS 6.2 PRNG

```c
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```c
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

ScreenOS 6.2 PRNG

```
char output[32];      // PRNG output buffer
int  index;           // Index into output
char seed[8];         // X9.31 seed
char key[24];         // X9.31 key
char block[8];        // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

Conditional reseed

ScreenOS 6.2 PRNG

```
char output[32];      // PRNG output buffer
int  index;           // Index into output
char seed[8];         // X9.31 seed
char key[24];         // X9.31 key
char block[8];        // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

Conditional reseed

Generate 32 bytes, 8 bytes at a time, via X9.31; store in output

ScreenOS 6.2 PRNG

```
char output[32];    // PRNG output buffer
int  index;         // Index into output
char seed[8];       // X9.31 seed
char key[24];       // X9.31 key
char block[8];      // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

Conditional reseed

Generate 32 bytes, 8 bytes at a time, via X9.31; store in output

ScreenOS 6.2 PRNG

```c
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

Generate 32 bytes, via Dual EC; store in output

```c
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
      reseed();
      ndex < 32; index += 8) {
      S checks removed for clarity
      gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

Conditional reseed

Generate 32 bytes, 8 bytes at a time, via X9.31; store in output

ScreenOS 6.2 PRNG

```
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
      eseed();
      ndex < 32; index += 8) {
      S checks removed for clarity
      gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

Generate 32 bytes, via Dual EC; store in output

First 8 bytes become new X9.31 seed; remaining 24 become new X9.31 key

Conditional reseed

Generate 32 bytes, 8 bytes at a time, via X9.31; store in output

UIC

ScreenOS 6.2 PRNG

```
char output[32];    // PRNG output buffer
int  index;         // Index into output
char seed[8];       // X9.31 seed
char key[24];       // X9.31 key
char block[8];      // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

ScreenOS 6.2 PRNG

```
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

index set to 0

```
void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

ScreenOS 6.2 PRNG

```c
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```c
void prng_generate(void) {
  int time[2] = { 0, g
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

index set to 0

Always returns false*; reseed on every call

★ Can be disabled via undocumented configuration command

ScreenOS 6.2 PRNG

```c
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void)
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

32 bytes from Dual EC stored in output

```c
void prng_generate(void) {
  int time[2] = { 0, g
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

index set to 0

Always returns false*; reseed on every call

★ Can be disabled via undocumented configuration command

ScreenOS 6.2 PRNG

```
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void)
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

32 bytes from Dual EC stored in output

index set to 32

```
void prng_generate(void) {
  int time[2] = { 0, g
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

index set to 0

Always returns false*; reseed on every call

★ Can be disabled via undocumented configuration command

ScreenOS 6.2 PRNG

```
char output[32];      // PRNG output buffer
int  index;           // Index into output
char seed[8];         // X9.31 seed
char key[24];         // X9.31 key
char block[8];        // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void)
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

32 bytes from Dual EC stored in output

index set to 32

```
void prng_generate(void) {
  int time[2] = { 0, g
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

index set to 0

Always returns false*; reseed on every call

Loop never executes!

★ Can be disabled via undocumented configuration command

ScreenOS 6.2 PRNG

```c
char output[32];    // PRNG output buffer
int  index;         // Index into output
char seed[8];       // X9.31 seed
char key[24];       // X9.31 key
char block[8];      // X9.31 output block
int  reseed_counter;

void x9_31_reseed(void) {
  reseed_counter = 0;
  if (dualec_generate(output, 32) != 32)
    error("[...]PRNG failure[...]", 11);
  memcpy(seed, output, 8);
  index = 8;
  memcpy(key, &output[index], 24);
  index = 32;
}
```

```c
void prng_generate(void) {
  int time[2] = { 0, g
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed();
  for (; index < 32; index += 8) {
    // FIPS checks removed for clarity
    x9_31_gen(time, seed, key, block);
    // FIPS checks removed for clarity
    memcpy(&output[index], block, 8);
  }
}
```

32 bytes from Dual EC stored in output

index set to 32

index set to 0

Always returns false*; reseed on every call

Loop never executes!

output still contains 32 bytes from Dual EC

★ Can be disabled via undocumented configuration command

What the heck is going on?

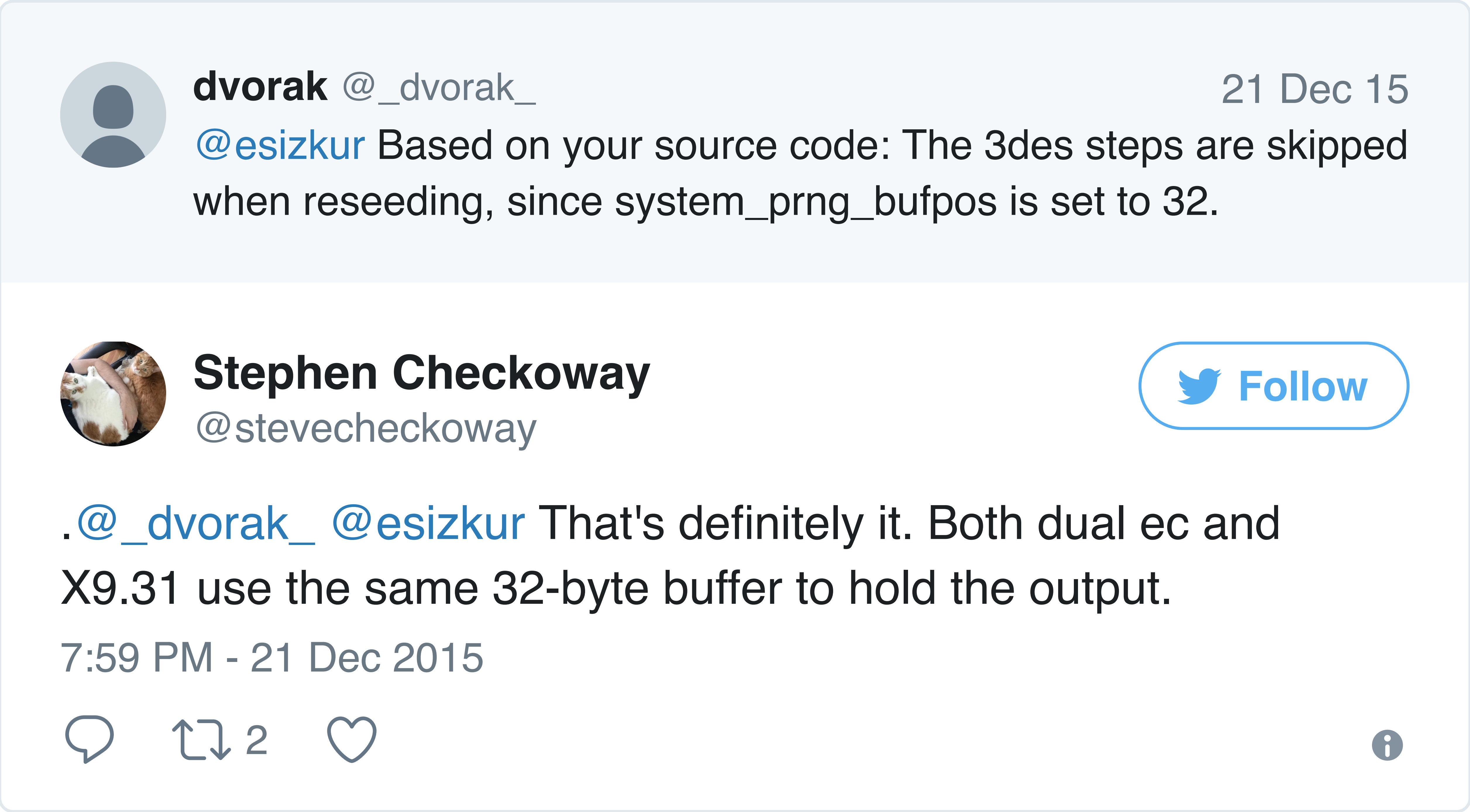
Global output buffer used as both

1. Reseed temporary buffer
2. Output of prng_generate

Index var is global…*for some reason*

Index reuse first publicly noted by Willem Pinckaers (@_dvorak_) on Twitter

```
char output[32];     // PRNG output buffer
int  index;          // Index into output
```



dvorak @_dvorak_ 21 Dec 15

@esizkur Based on your source code: The 3des steps are skipped when reseeding, since system_prng_bufpos is set to 32.

Stephen Checkoway @stevecheckoway Follow

.@_dvorak_ @esizkur That's definitely it. Both dual ec and X9.31 use the same 32-byte buffer to hold the output.

7:59 PM - 21 Dec 2015

2

First research question

Why doesn't the use of X9.31 defend against a compromised Q?

Contrary to Juniper's assertion, X9.31 is never used due to the reuse of the output buffer and the global index variable.

Internet Key Exchange (IKE)

- Used to establish keys for VPN session
- Two major versions, IKEv1 and IKEv2
- Both use two phases:
  - Phase 1 establishes keys to encrypt the phase 2 handshake
  - Phase 2 establishes keys for IPSec (or other encapsulated protocol)
- Classic Diffie–Hellman key exchange between peers

IKE Phase 1 packet

Header

Payload: Security Association

Contains details about which cipher suites to use

Payload: Key Exchange

Contains DH public key, g^x

Payload: Nonce

Contains 8–128 byte random value

Other payloads: Vendor info, identification, etc.

IKE Phase 1 packet

Header

Payload: Security Association

Contains details about which cipher suites to use

Payload: Key Exchange

Contains DH public key, g^x

> ScreenOS: 20-byte private key x generated via Dual EC

Payload: Nonce

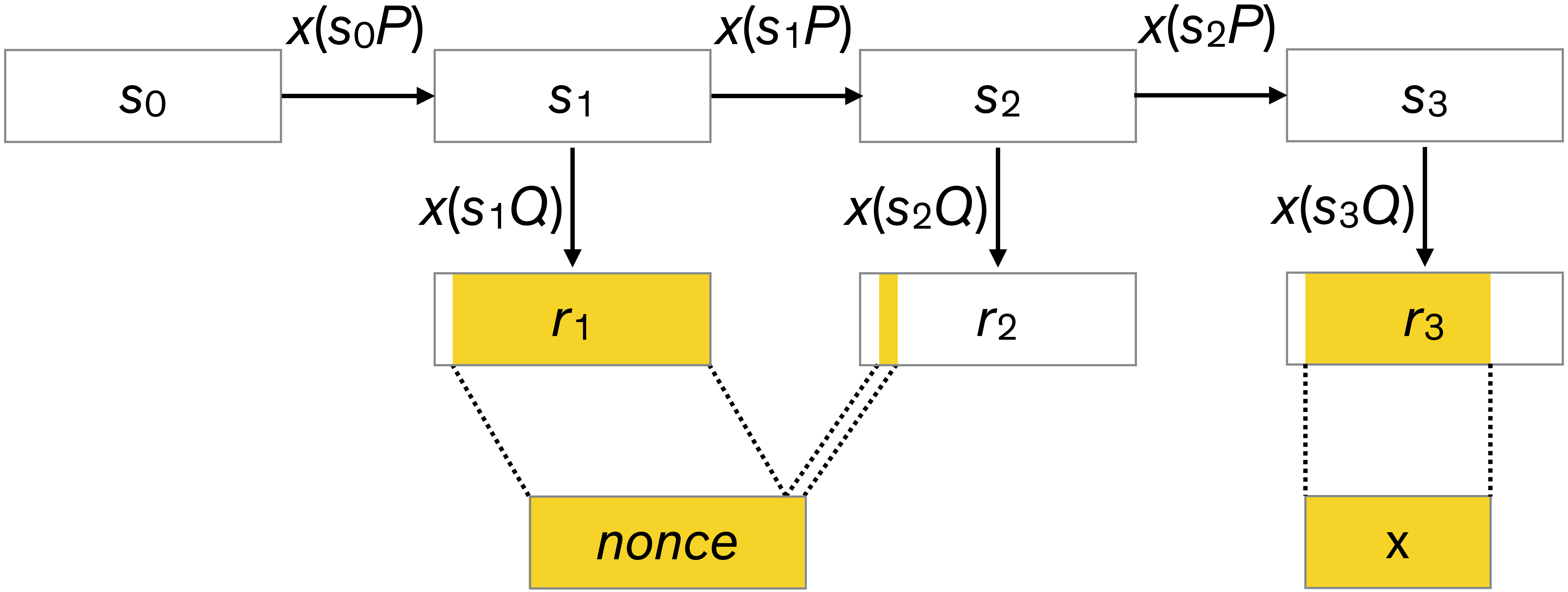
Contains 8–128 byte random value

> ScreenOS: 32 bytes, generated via Dual EC

Other payloads: Vendor info, identification, etc.

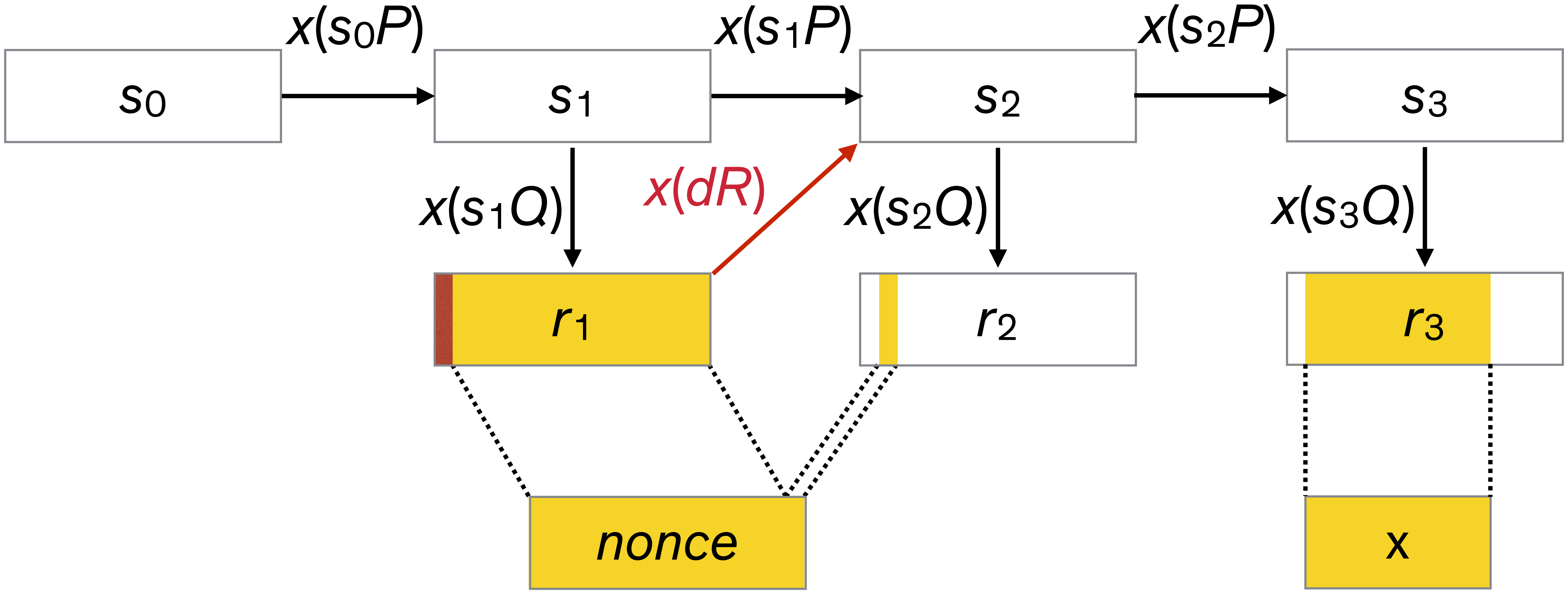
Attacking IKE phase 1 (ideal)

- Nonce generated before Diffie–Hellman private exponent x
- Use Shumow–Ferguson attack on nonce to recover PRNG state s_2
- Predict private exponent x, compare g^x with Diffie–Hellman public key



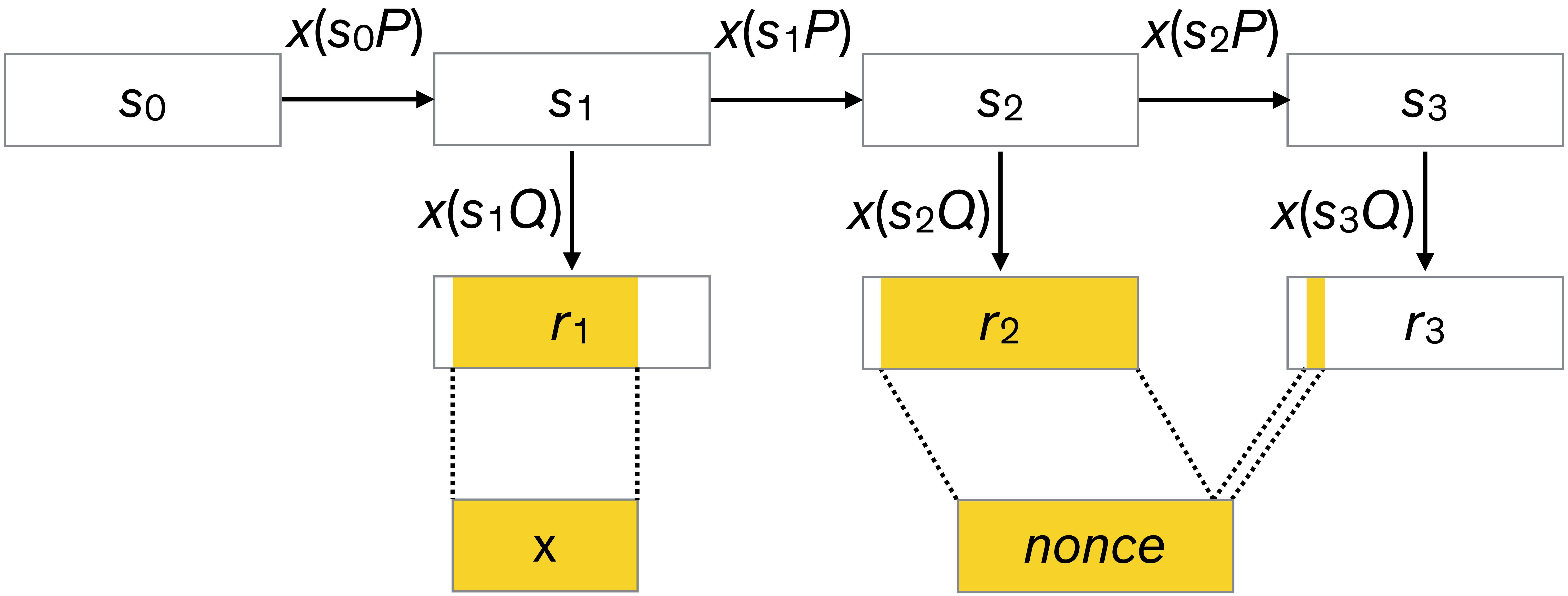
Attacking IKE phase 1 (ideal)

- Nonce generated before Diffie–Hellman private exponent x
- Use Shumow–Ferguson attack on nonce to recover PRNG state s_2
- Predict private exponent x, compare g^x with Diffie–Hellman public key



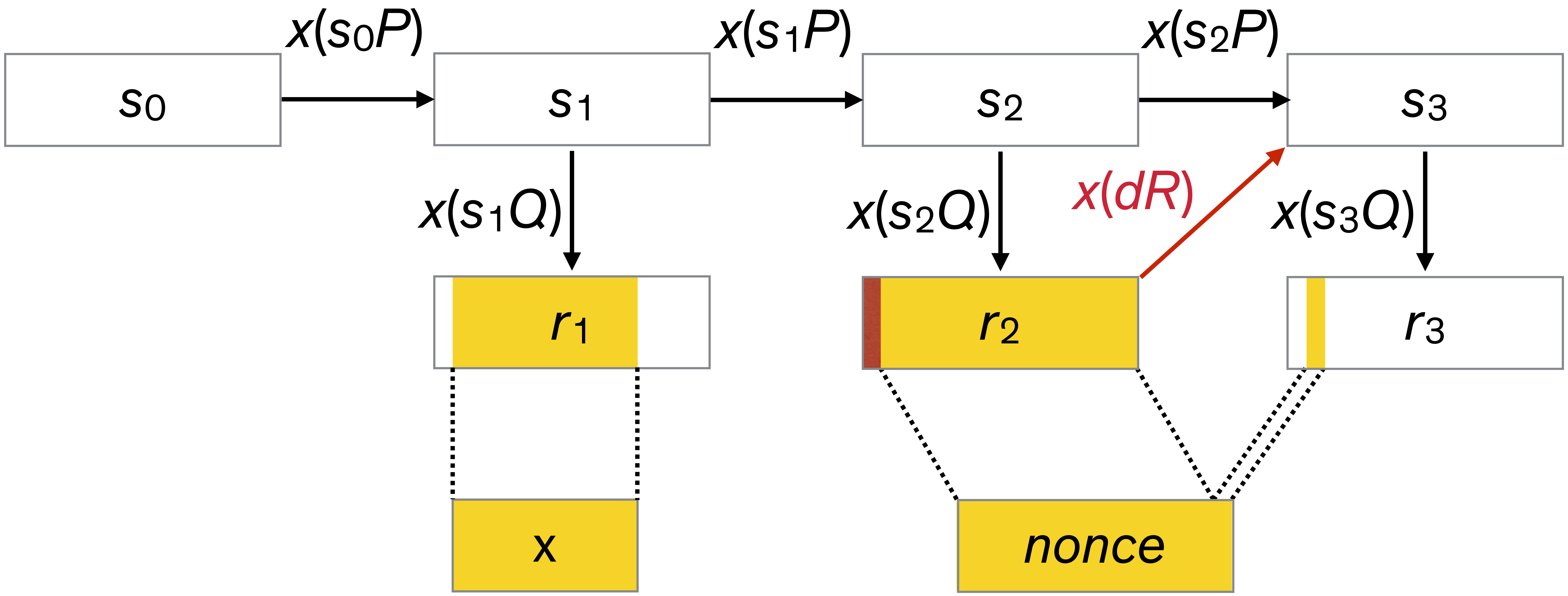
Attacking IKE phase 1 (apparent)

- In protocol and code, nonce apparently generated *after* exponent
- Shumow–Ferguson attack doesn't recover *x*



Attacking IKE phase 1 (apparent)

- In protocol and code, nonce apparently generated *after* exponent
- Shumow–Ferguson attack doesn't recover x



Attacking IKE phase 1 (reality)

- ScreenOS contains queues of pre-generated nonces and DH key pairs
- Queues filled one element per second, *nonces first*
- In many cases ideal attack succeeds: Each VPN connection can be decrypted individually
- It's possible for *x* to be generated before *nonce* which necessitates a multi-connection attack (see paper for details)

IKE phase 1 authentication modes

IKEv1

- Digital signatures: Attack works!
- Preshared keys: Attack works but attacker needs to know the key
- Public key encryption (2 modes): Attack fails due to encrypted nonces

IKEv2

- Key derivation independent of authentication modes: Attack works!

Attacking IKE phase 2

Phase 2

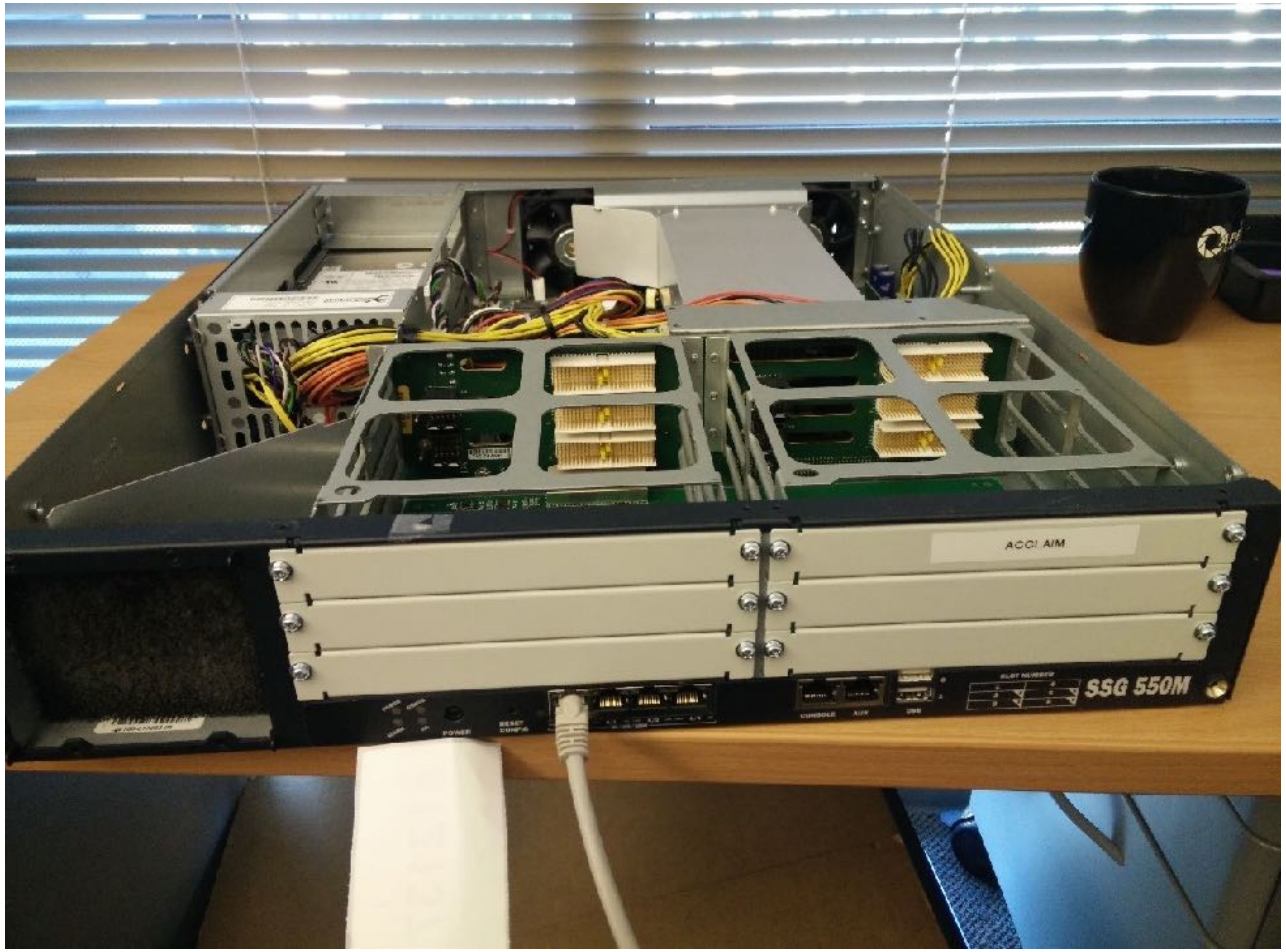
- New nonces are exchanged
- Optional second Diffie–Hellman exchange

Attack possibilities with a second Diffie–Hellman exchange

- Rerun Shumow–Ferguson attack
- Run Dual EC forward from the state recovered for phase 1

Proof of concept

- Bought a NetScreen SSG 550M
- Created modified firmware with our own *Q* (for which we know the discrete log *d*)
- Attacked VPN configurations
  - IKEv1 with PSK (required PSK)
  - IKEv1 with RSA cert
  - IKEv2 with PSK
  - IKEv2 with RSA cert



Second research question

Why does a change in Q result in passive VPN decryption?

Dual EC output is directly used to create the IKE nonces and Diffie–Hellman private exponents so the Shumow–Ferguson attack applies, at least for some VPN configurations.

Third research question

What is the history of the ScreenOS PRNG code?

ScreenOS 6.1.0r7 (last 6.1 revision)

- ANSI X9.31
  - Reseeded every 10k calls
- 20-byte IKE nonces
- DH pre-generation queues

ScreenOS 6.2.0r0 (first 6.2 revision)

- Dual EC → ANSI X9.31 cascade
  - Reseeded every call
  - Reseed “bug” exposes Dual EC
- 32-byte IKE nonces
- DH & nonce pre-generation queues

Raises a number “why” questions

1. Introduction of Dual EC

Dual EC was added to seed ANSI X9.31. Why?

- No engineering reason I can think of
    - Required the introduction of a lot of custom elliptic curve code to their embedded copy of OpenSSL
- No standardization reason
    - ScreenOS was already FIPS certified for X9.31
    - ScreenOS was never FIPS certified for Dual EC

2. Reseed on every call

ScreenOS 6.1 (without FIPS checks)

```
char seed[8];       // X9.31 seed
char key[24];       // X9.31 key
char block[8];      // X9.31 output block
int  reseed_counter;

void prng_generate(char *output) {
  int index = 0;
  if (reseed_counter++ > 9999)
    x9_31_reseed();
  int time[2] = { 0, get_cycles() };
  do {
    x9_31_gen(time, seed, key, block);
    int size = min(20-index, 8);
    memcpy(&output[index], block, size);
    index += size;
  } while (index < 20);
}
```

ScreenOS 6.2 (without FIPS checks)

```
char output[32];    // PRNG output buffer
int  index;         // Index into output
char seed[8];       // X9.31 seed
char key[24];       // X9.31 key
char block[8];      // X9.31 output block
int  reseed_counter;

void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed(); // Sets index to 32
  for (; index < 32; index += 8) {
    x9_31_gen(time, seed, key, block);
    memcpy(&output[index], block, 8);
  }
}
```

2. Reseed on every call

X9.31 PRNG reseeded on every call. Why?

- No engineering reason I can think of
- Maybe for X9.31 backtracking resistance?
- Could just be another bug

3. Reseed "bug"

ScreenOS 6.1 (without FIPS checks)

```
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void prng_generate(char *output) {
  int index = 0;
  if (reseed_counter++ > 9999)
    x9_31_reseed();
  int time[2] = { 0, get_cycles() };
  do {
    x9_31_gen(time, seed, key, block);
    int size = min(20-index, 8);
    memcpy(&output[index], block, size);
    index += size;
  } while (index < 20);
}
```

ScreenOS 6.2 (without FIPS checks)

```
char output[32];     // PRNG output buffer
int  index;          // Index into output
char seed[8];        // X9.31 seed
char key[24];        // X9.31 key
char block[8];       // X9.31 output block
int  reseed_counter;

void prng_generate(void) {
  int time[2] = { 0, get_cycles() };
  index = 0;
  ++reseed_counter;
  if (!one_stage_rng())
    x9_31_reseed(); // Sets index to 32
  for (; index < 32; index += 8) {
    x9_31_gen(time, seed, key, block);
    memcpy(&output[index], block, 8);
  }
}
```

3. Reseed “bug”

Both output and index became global variables and are reused by the reseed procedure in ScreenOS 6.2. Why?

- No (good*) engineering reason I can think of
- Could just be a bug, but it’s a very strange one

* Sharing a global 32-byte buffer may be reasonable for some classes of *extremely* space-constrained devices. The NetScreen family doesn’t belong to such a class.

4. IKE nonce size increase

ScreenOS 6.3 increases the IKE nonce size from 20 bytes to 32 bytes. Why?

- No engineering reason I can think of
- No (good*) cryptographic reason I can think of
- At 20 bytes, the Shumow–Ferguson attack takes ≈ 2^{96} scalar multiplications, at 32 bytes, it takes ≈ 2^{16}

* US Department of Defense apparently claimed “the public randomness for each side [in TLS] should be at least twice as long as the security level for cryptographic parity” — *Extended Random Values for TLS*.

5. IKE nonce pre-generation queue

ScreenOS 6.2 has pre-generated Diffie–Hellman key pairs

- Reasonable. Computing $g^x \pmod{p}$ is computationally expensive

ScreenOS 6.3 adds pre-generated nonces. Why?

- Dual EC is about 125× slower than X9.31 (4 elliptic curve point multiplications for 32 bytes)
- Engineering reason: Adding Dual EC likely noticeably slowed down VPN connections

ScreenOS PRNG changes

ScreenOS 6.1.0r7 (last 6.1 revision)

- ANSI X9.31
  - Reseeded every 10k calls
- 20-byte IKE nonces
- DH pre-generation queues

ScreenOS 6.2.0r0 (first 6.2 revision)

- Dual EC → ANSI X9.31 cascade
  - Reseeded every call
  - Reseed “bug” exposes Dual EC
- 32-byte IKE nonces
- DH & nonce pre-generation queues

Required for passive VPN decryption

Enables single connection decryption

Research questions revisited

1. Why doesn't the use of X9.31 defend against a compromised *Q*?
   X9.31 is not used.
2. Why does a change in *Q* result in passive VPN decryption?
   Shumow–Ferguson attack on IKE.
3. What is the history of the ScreenOS PRNG code?
   Many attack-enabling changes in one point release
4. Are the versions of ScreenOS with Juniper's *Q* vulnerable to attack?
   Maybe. It depends on how *Q* was generated and who knows *d*
5. How was Juniper's *Q* generated?
   Impossible to say with the data we have

UIC

Lessons learned

Pseudorandom numbers are critical; be wary of exposing raw output

- Consider hashing output before putting it on the wire
- Scrutinize any PRNG changes, including output length changes, closely
- Use separate PRNG instances for public and secret data

Don't allow nonces to vary in length or be longer than necessary

- E.g., IKE's 256-byte nonces are unnecessarily long
- Long/variable length nonces provide implementations the opportunity to expose secrets
- Variable length enables implementation fingerprinting

Stephen Checkoway
sfc@uic.edu
@stevecheckoway

Lessons learned

Include even low-entropy secrets into key derivation

- IKEv1 PSK more secure than IKEv2 PSK because the PSK influences key derivation in IKEv1

NOBUS (NObody But US) need not remain so

- Dual EC is (indistinguishable from) a building block of a NOBUS exceptional access mechanism
- This incident is a clear warning of the dangers of exceptional access