An Analysis of OpenSSL's Random Number Generator
Eurocrypt 2016

Falko Strenzke

cryptosource

cryptosource GmbH,
Darmstadt
fstrenzke@cryptosource.de

September 14, 2016

Pseudo Random Number Generation

- Software-based RNG's use pseudo random number generators (PRNGs)
- but are not PRNGs

seed

RAND_add (buffer, estimated entopy)

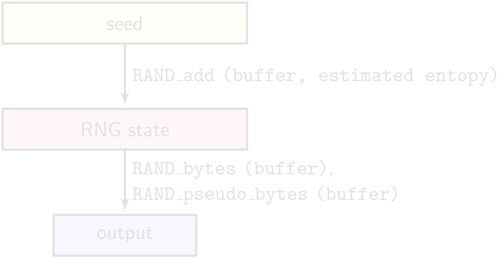
RNG state

RAND_bytes (buffer),
RAND_pseudo_bytes (buffer)

output

Pseudo Random Number Generation

- Software-based RNG's use pseudo random number generators (PRNGs)
- but are not PRNGs



seed

RAND_add (buffer, estimated entopy)

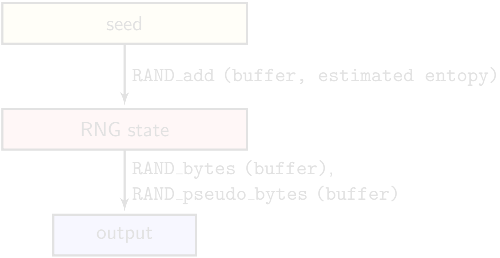
RNG state

RAND_bytes (buffer),
RAND_pseudo_bytes (buffer)

output

Pseudo Random Number Generation

- Software-based RNG's use pseudo random number generators (PRNGs)
- but are not PRNGs



seed

RAND_add (buffer, estimated entopy)

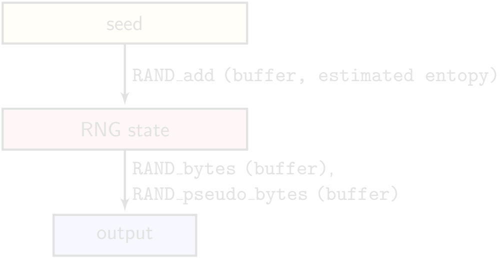
RNG state

RAND_bytes (buffer),
RAND_pseudo_bytes (buffer)

output

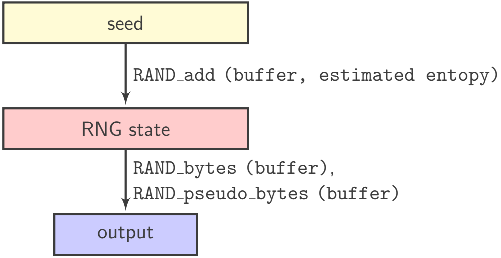
Pseudo Random Number Generation

- Software-based RNG's use pseudo random number generators (PRNGs)
- but are not PRNGs

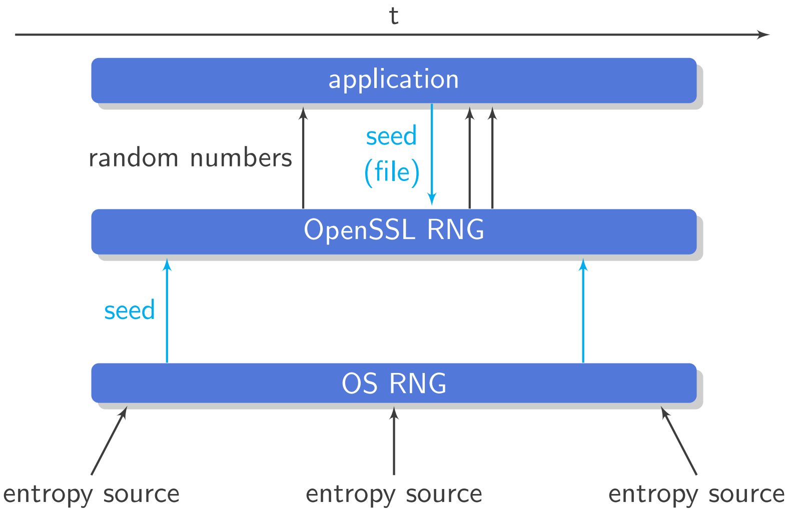

seed
RAND_add (buffer, estimated entopy)
RNG state
RAND_bytes (buffer),
RAND_pseudo_bytes (buffer)
output

Pseudo Random Number Generation

- Software-based RNG's use pseudo random number generators (PRNGs)
- but are not PRNGs

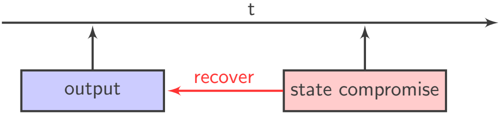

seed
RAND_add (buffer, estimated entopy)
RNG state
RAND_bytes (buffer),
RAND_pseudo_bytes (buffer)
output

Random Number Generation in Cryptographic Libraries

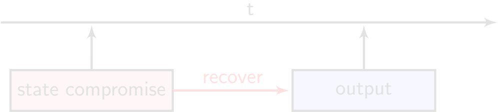


Security Notions for RNGs

- forward security



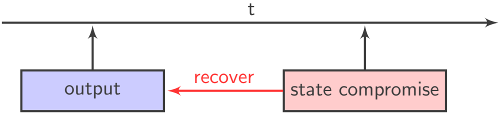
- backward security



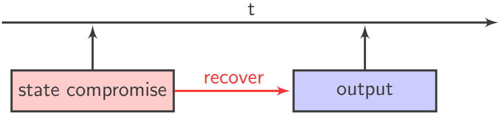
- don't leak any information about state in output

Security Notions for RNGs

- forward security



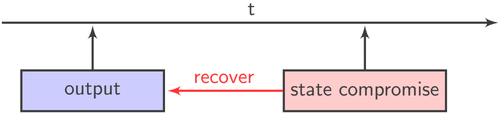
- backward security



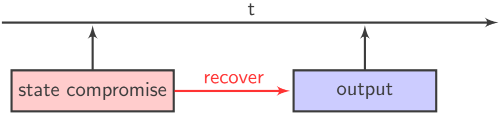
- don't leak any information about state in output

Security Notions for RNGs

- forward security



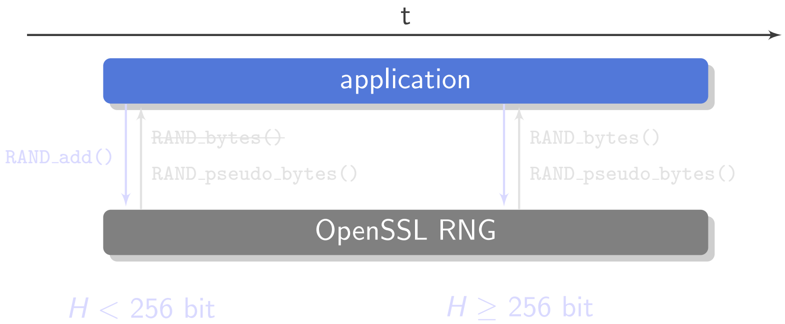
- backward security



- don't leak any information about state in output

Low Entropy Secret Leakage

Seeding by the Application



t

application

RAND_add()

~~RAND_bytes()~~

RAND_pseudo_bytes()

RAND_bytes()

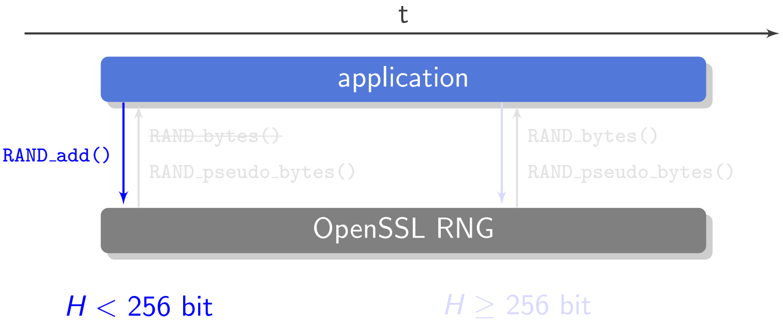
RAND_pseudo_bytes()

OpenSSL RNG

$H < 256$ bit

$H \geq 256$ bit

Seeding by the Application



t

application

RAND_add()

~~RAND_bytes()~~

RAND_pseudo_bytes()

RAND_bytes()

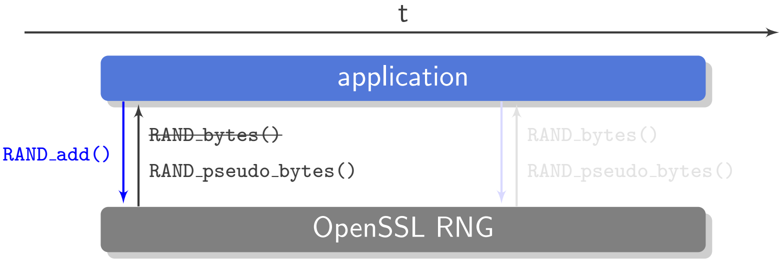
RAND_pseudo_bytes()

OpenSSL RNG

$H < 256$ bit

$H \geq 256$ bit

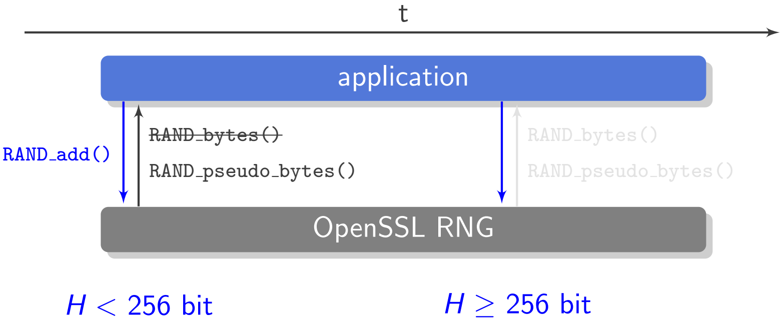
Seeding by the Application



$H < 256$ bit

$H \geq 256$ bit

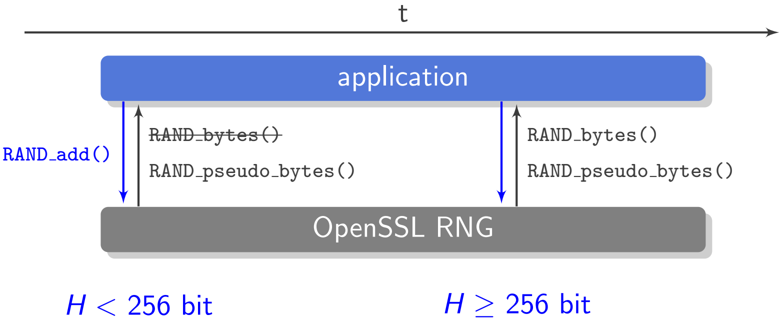
Seeding by the Application



$H < 256$ bit

$H \geq 256$ bit

Seeding by the Application



t

application

RAND_add()

~~RAND_bytes()~~

RAND_pseudo_bytes()

RAND_bytes()

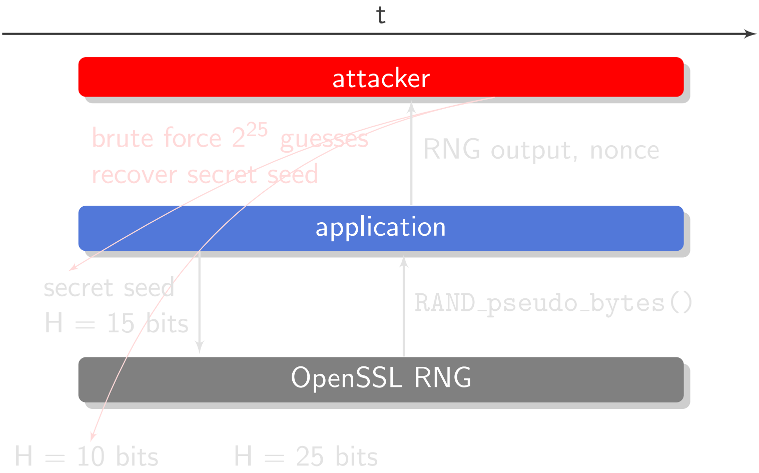
RAND_pseudo_bytes()

OpenSSL RNG

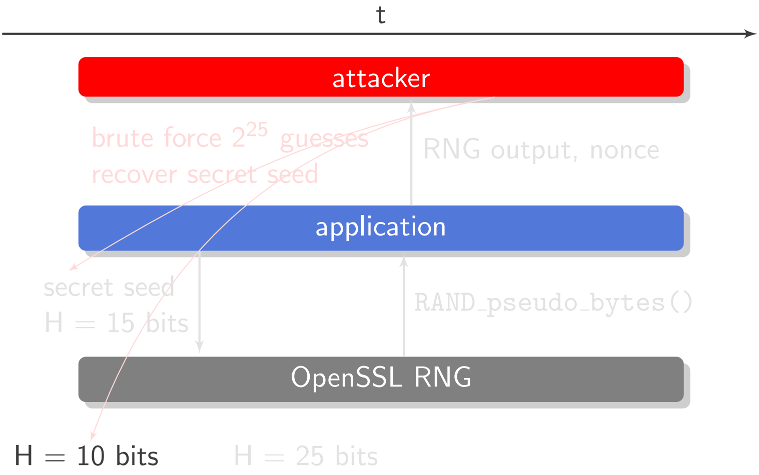
$H < 256$ bit

$H \geq 256$ bit

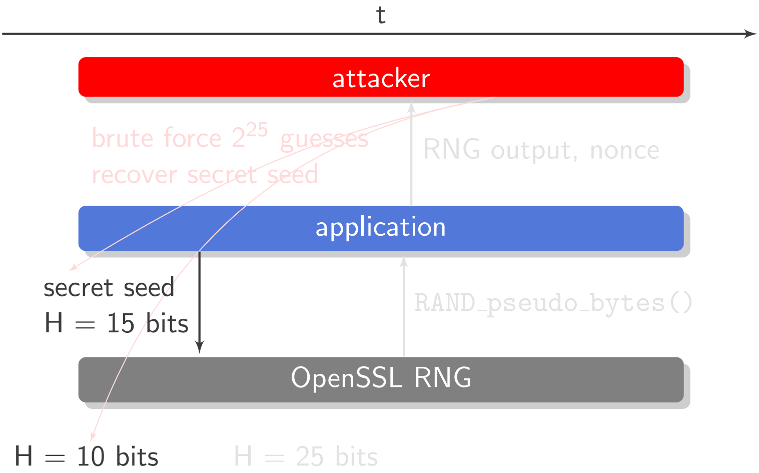
Outputting Random Numbers in Low Entropy States



Outputting Random Numbers in Low Entropy States



Outputting Random Numbers in Low Entropy States



t

attacker

brute force 2^{25} guesses

recover secret seed

RNG output, nonce

application

secret seed

$H = 15$ bits

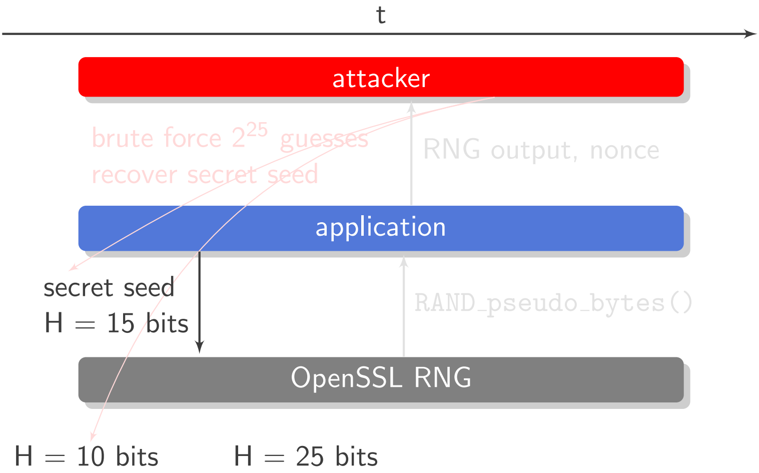
RAND_pseudo_bytes()

OpenSSL RNG

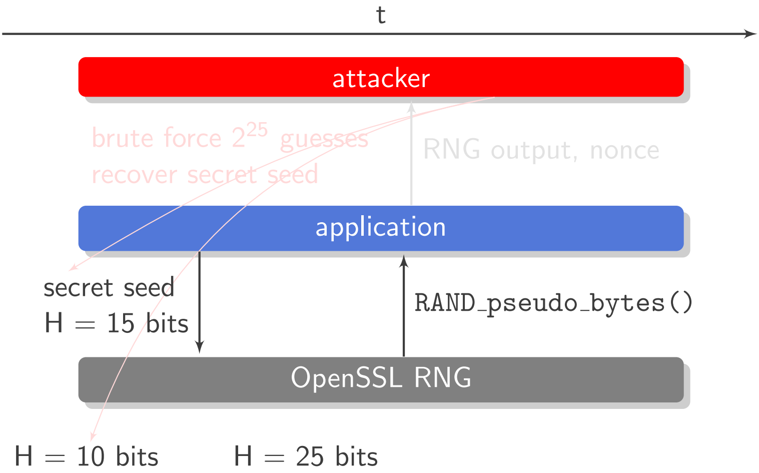
$H = 10$ bits

$H = 25$ bits

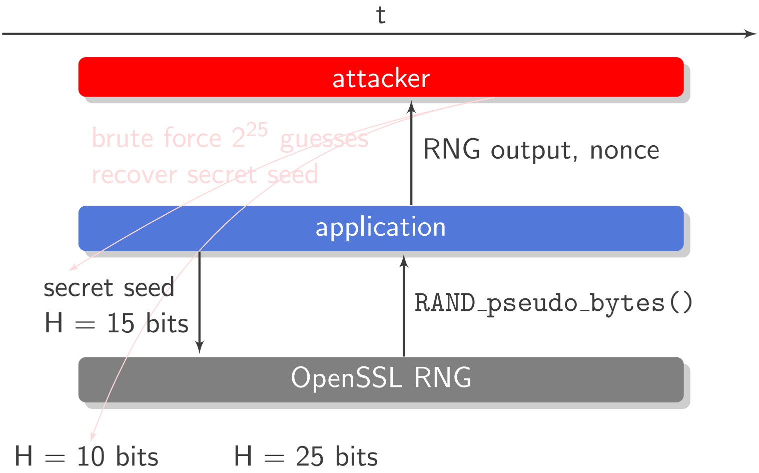
Outputting Random Numbers in Low Entropy States



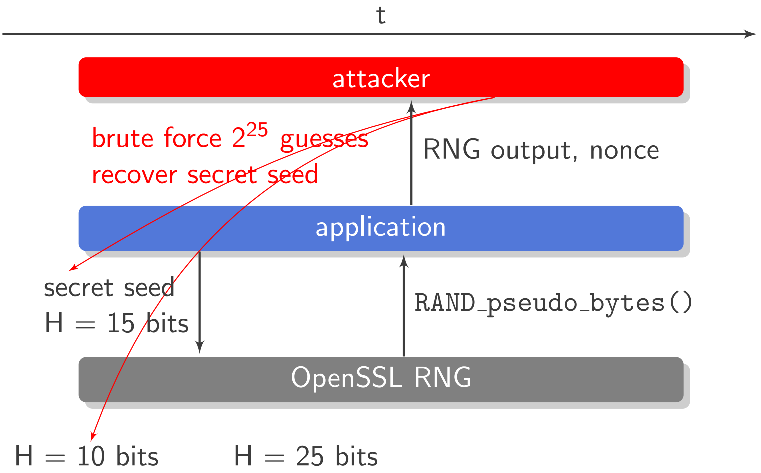
Outputting Random Numbers in Low Entropy States



Outputting Random Numbers in Low Entropy States



Outputting Random Numbers in Low Entropy States



Potentially Leaked Secrets

- RAND_pseudo_bytes generates output in the same way as RAND_bytes
- API documentation suggests to feed low-entropy secrets such passwords
- OpenSSL feeds the previous contents of buffers to be randomized to RNG state (Debian issue in 2008)
- previous contents could contain low entropy secrets by themselves
- overwriting secrets with random numbers is an established practice
- overwritten low entropy secrets may be leaked in output

Potentially Leaked Secrets

- RAND_pseudo_bytes generates output in the same way as RAND_bytes
- API documentation suggests to feed low-entropy secrets such passwords
- OpenSSL feeds the previous contents of buffers to be randomized to RNG state (Debian issue in 2008)
- previous contents could contain low entropy secrets by themselves
- overwriting secrets with random numbers is an established practice
- overwritten low entropy secrets may be leaked in output

Potentially Leaked Secrets

- `RAND_pseudo_bytes` generates output in the same way as `RAND_bytes`
- API documentation suggests to feed low-entropy secrets such passwords
- OpenSSL feeds the previous contents of buffers to be randomized to RNG state (Debian issue in 2008)
- previous contents could contain low entropy secrets by themselves
- overwriting secrets with random numbers is an established practice
- overwritten low entropy secrets may be leaked in output

Potentially Leaked Secrets

- RAND_pseudo_bytes generates output in the same way as RAND_bytes
- API documentation suggests to feed low-entropy secrets such passwords
- OpenSSL feeds the previous contents of buffers to be randomized to RNG state (Debian issue in 2008)
- previous contents could contain low entropy secrets by themselves
- overwriting secrets with random numbers is an established practice
- overwritten low entropy secrets may be leaked in output

Potentially Leaked Secrets

- RAND_pseudo_bytes generates output in the same way as RAND_bytes
- API documentation suggests to feed low-entropy secrets such passwords
- OpenSSL feeds the previous contents of buffers to be randomized to RNG state (Debian issue in 2008)
- previous contents could contain low entropy secrets by themselves
- overwriting secrets with random numbers is an established practice
- overwritten low entropy secrets may be leaked in output

Potentially Leaked Secrets

- `RAND_pseudo_bytes` generates output in the same way as `RAND_bytes`
- API documentation suggests to feed low-entropy secrets such passwords
- OpenSSL feeds the previous contents of buffers to be randomized to RNG state (Debian issue in 2008)
- previous contents could contain low entropy secrets by themselves
- overwriting secrets with random numbers is an established practice
- overwritten low entropy secrets may be leaked in output

Core Cryptographic Function of OpenSSL's RNG

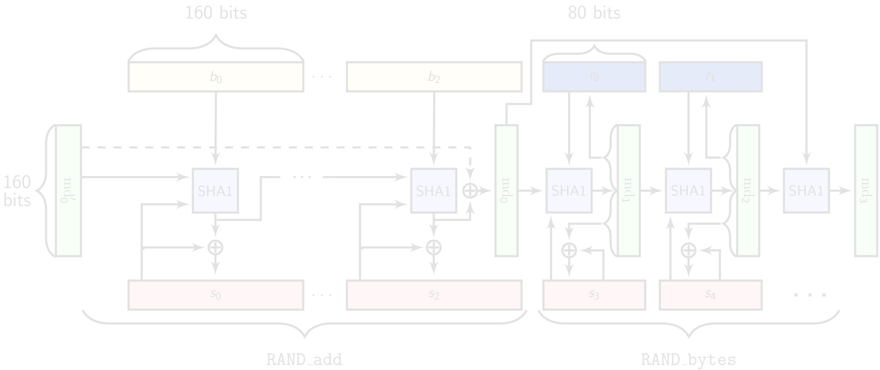
Core Cryptographic Function of OpenSSL's RNG

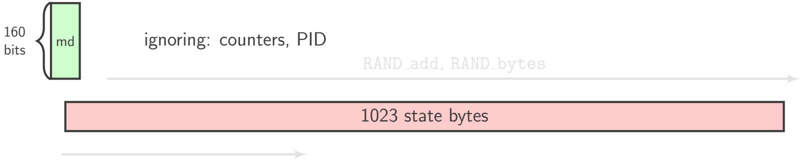
- custom design
- ©1998

Core Cryptographic Function of OpenSSL's RNG

- custom design
- ©1998

Core Cryptographic Function of OpenSSL's RNG

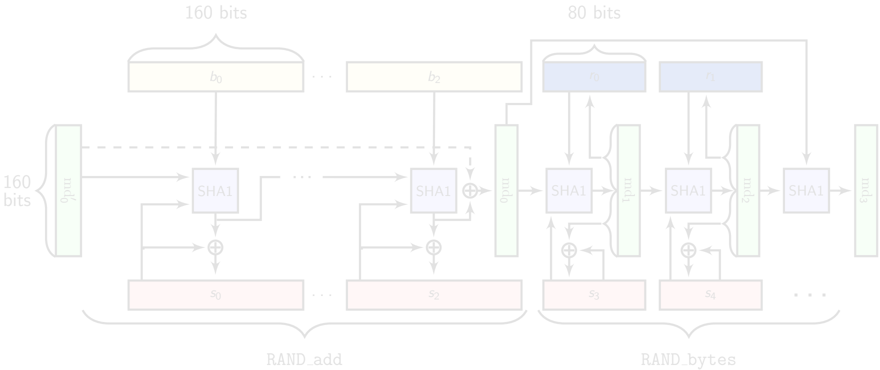




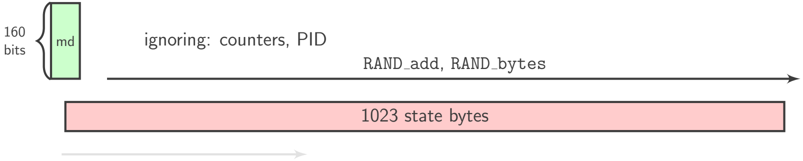
ignoring: counters, PID

1023 state bytes

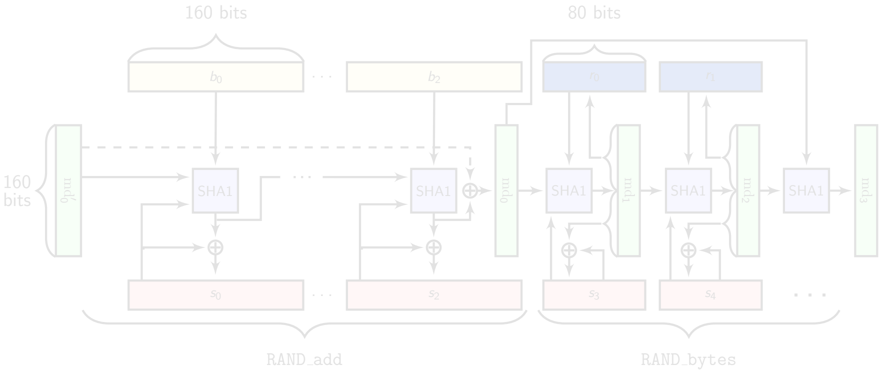
Core Cryptographic Function of OpenSSL's RNG

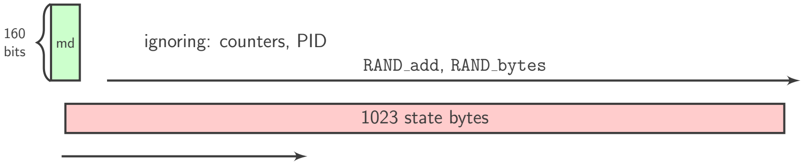


ignoring: counters, PID



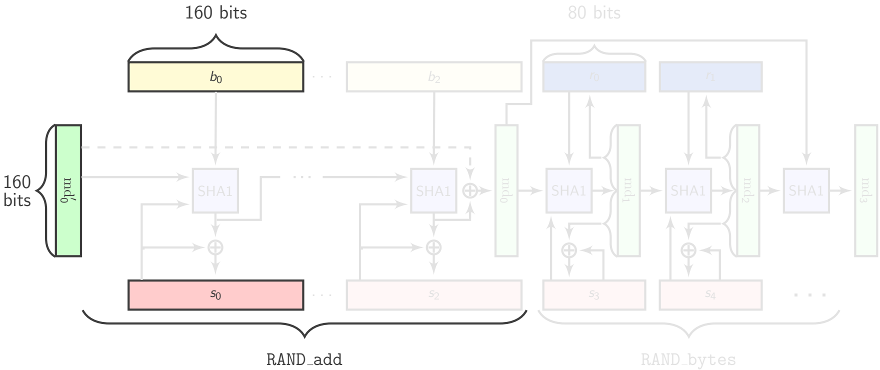
Core Cryptographic Function of OpenSSL's RNG

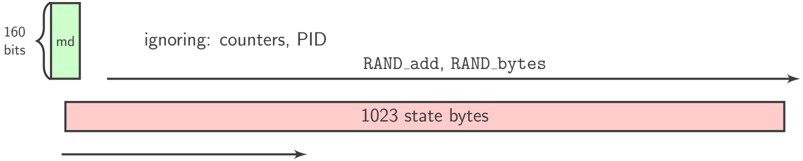




ignoring: counters, PID

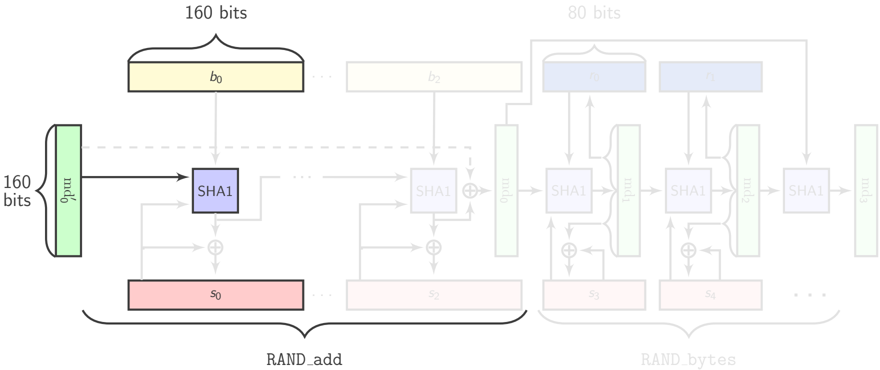
Core Cryptographic Function of OpenSSL's RNG

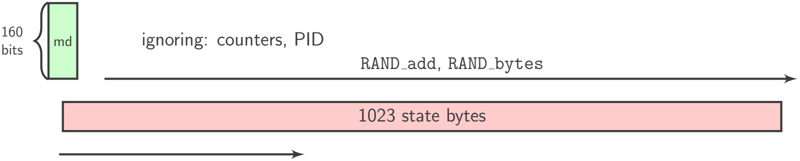




ignoring: counters, PID

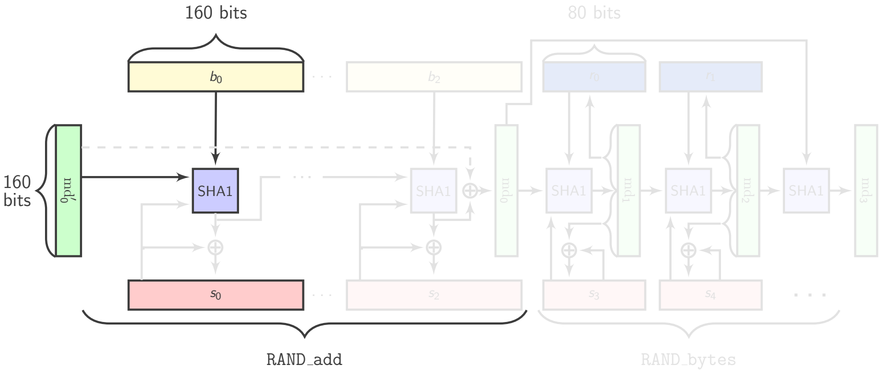
Core Cryptographic Function of OpenSSL's RNG

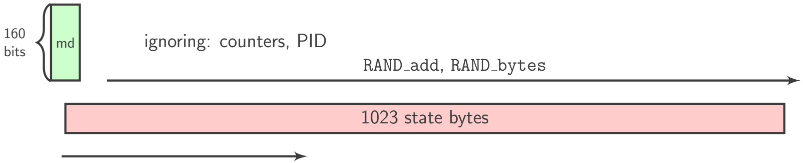




ignoring: counters, PID

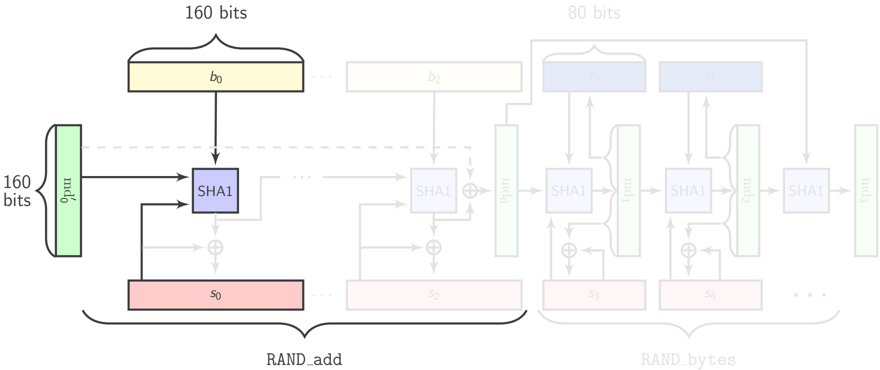
Core Cryptographic Function of OpenSSL's RNG



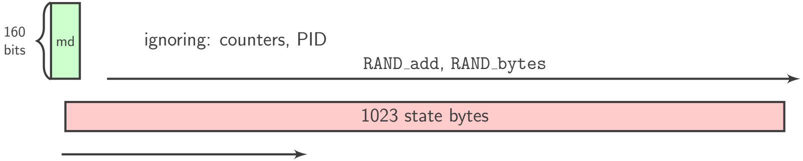


ignoring: counters, PID

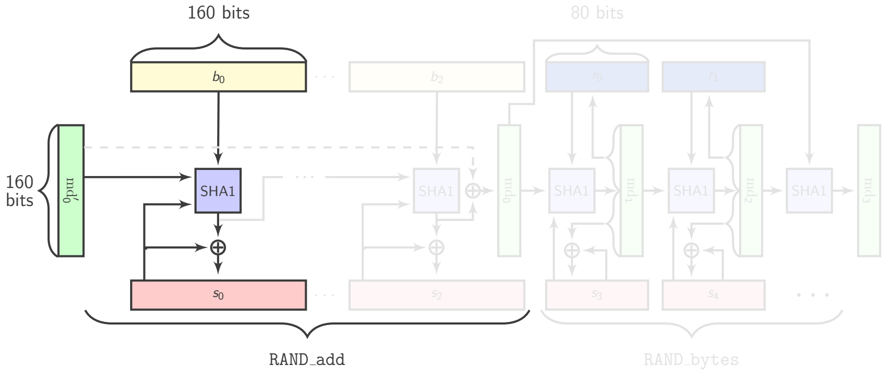
Core Cryptographic Function of OpenSSL's RNG

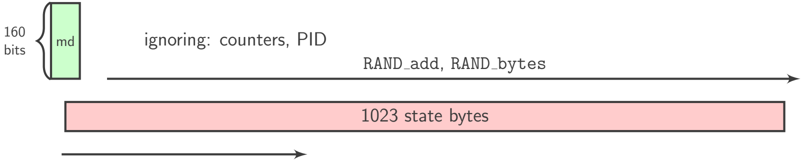


ignoring: counters, PID



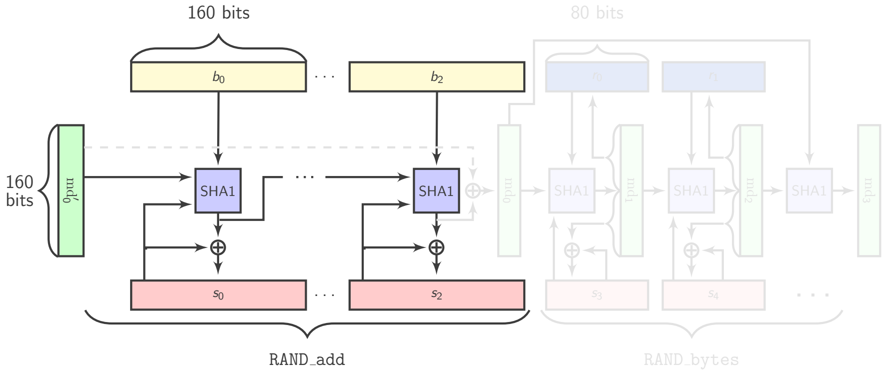
Core Cryptographic Function of OpenSSL's RNG



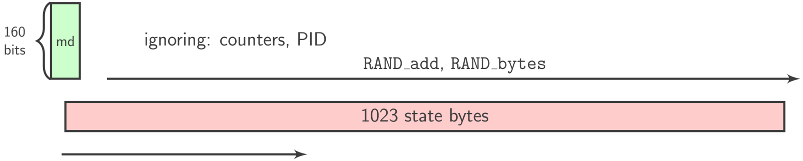


ignoring: counters, PID

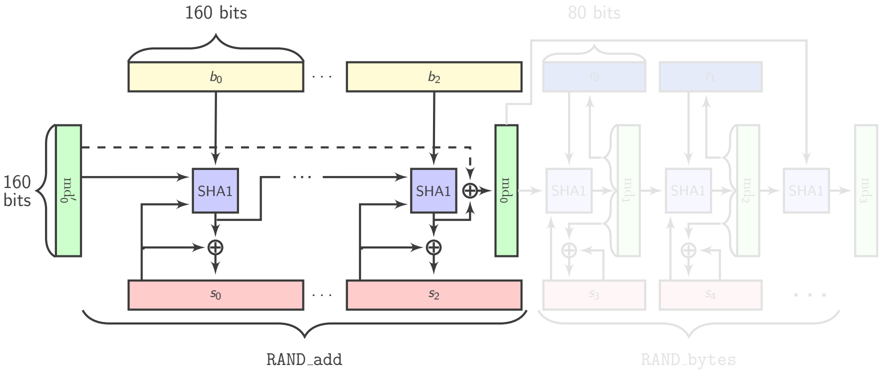
Core Cryptographic Function of OpenSSL's RNG

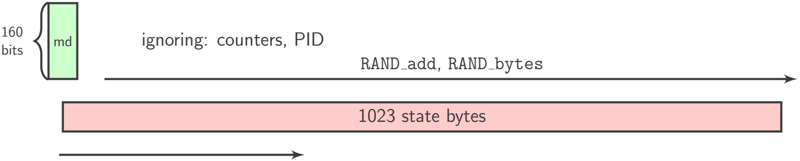


ignoring: counters, PID



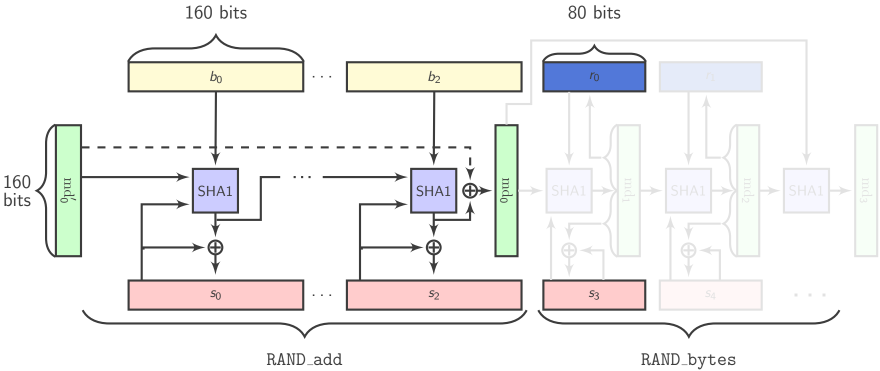
Core Cryptographic Function of OpenSSL's RNG

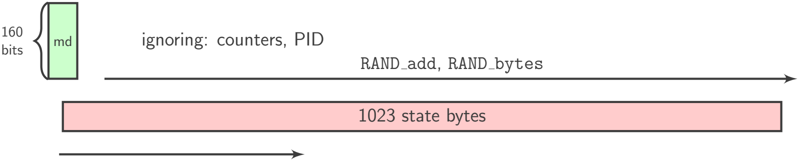




ignoring: counters, PID

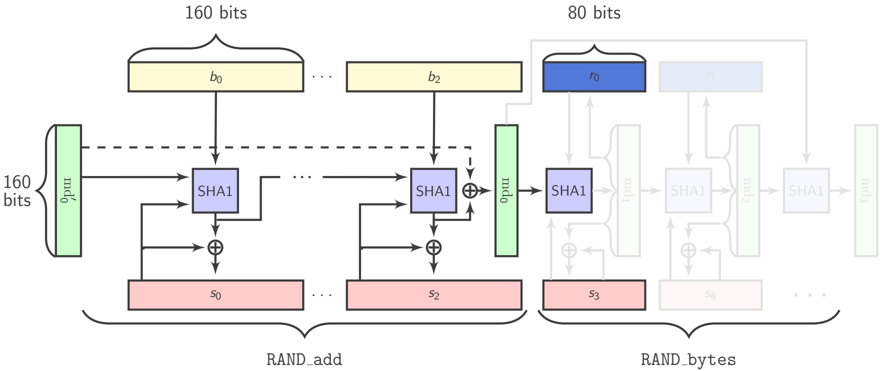
Core Cryptographic Function of OpenSSL's RNG



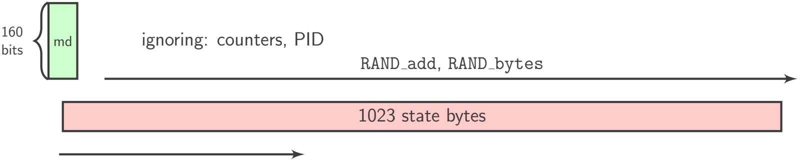


ignoring: counters, PID

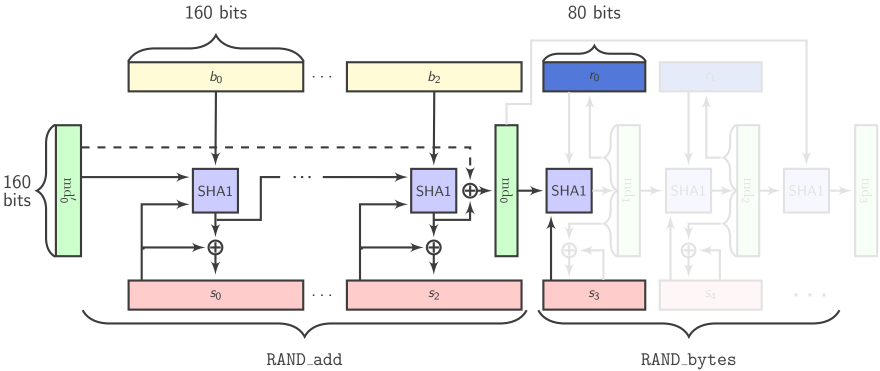
Core Cryptographic Function of OpenSSL's RNG

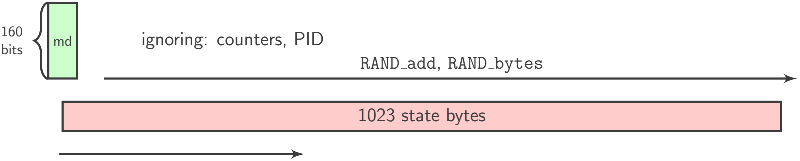


ignoring: counters, PID



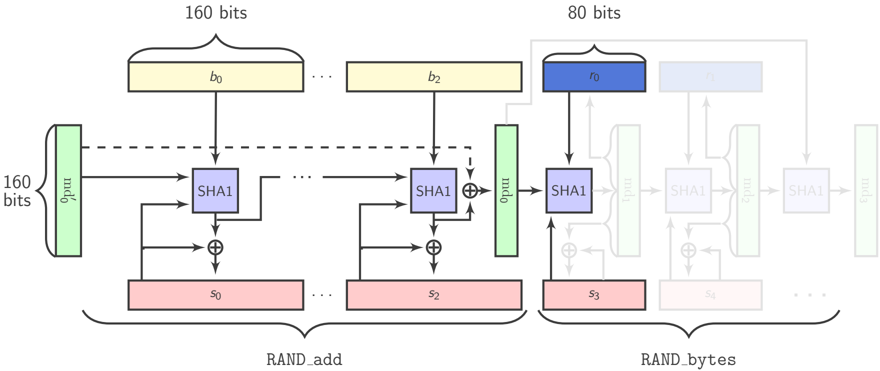
Core Cryptographic Function of OpenSSL's RNG



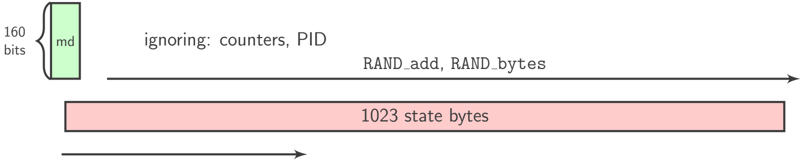


ignoring: counters, PID

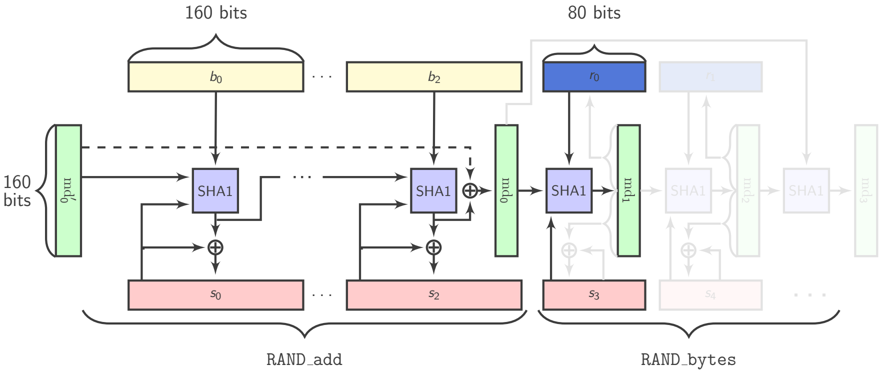
Core Cryptographic Function of OpenSSL's RNG

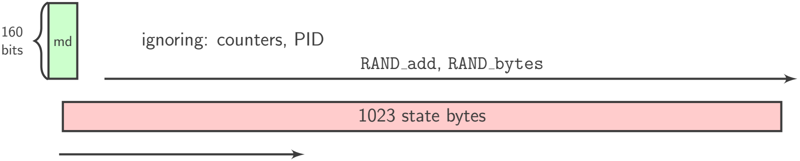


ignoring: counters, PID



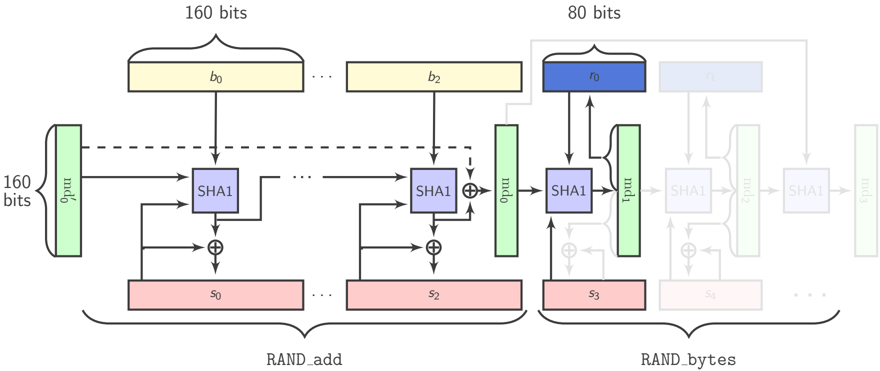
Core Cryptographic Function of OpenSSL's RNG

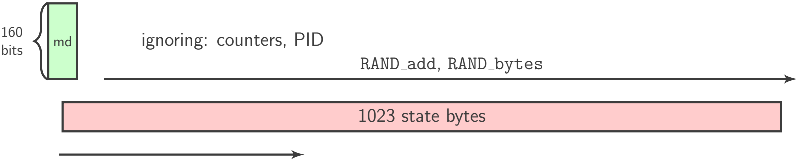




ignoring: counters, PID

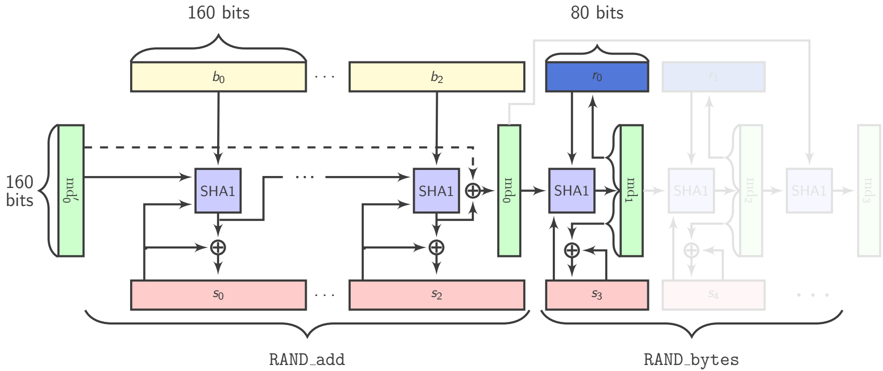
Core Cryptographic Function of OpenSSL's RNG

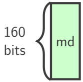




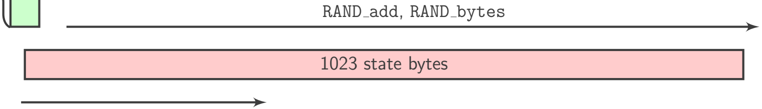
ignoring: counters, PID

Core Cryptographic Function of OpenSSL's RNG

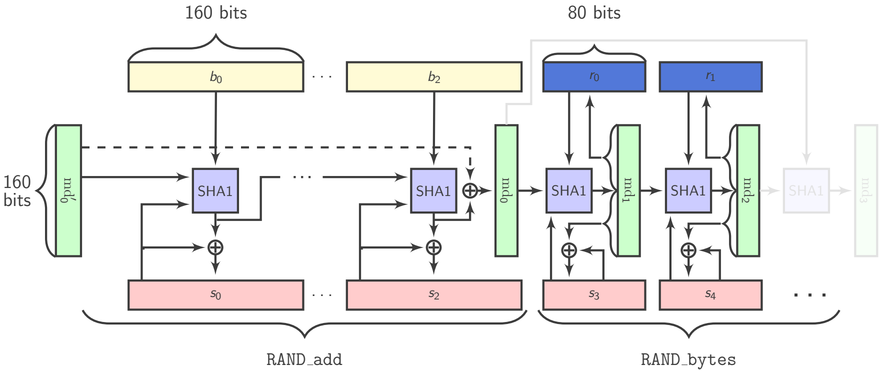


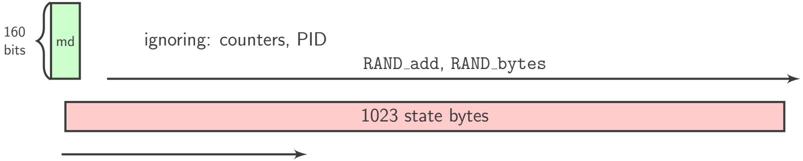


ignoring: counters, PID



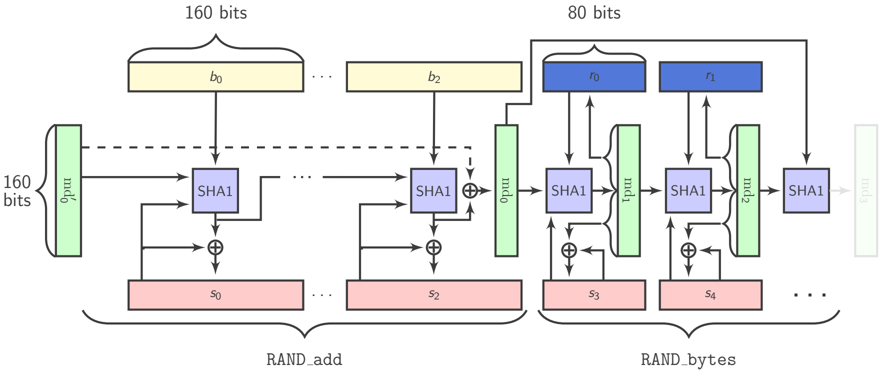
Core Cryptographic Function of OpenSSL's RNG

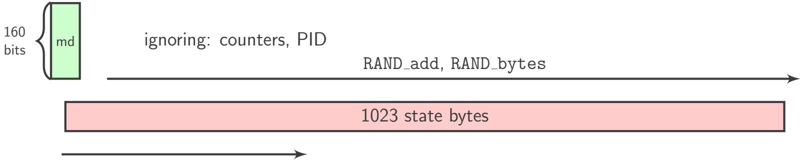




ignoring: counters, PID

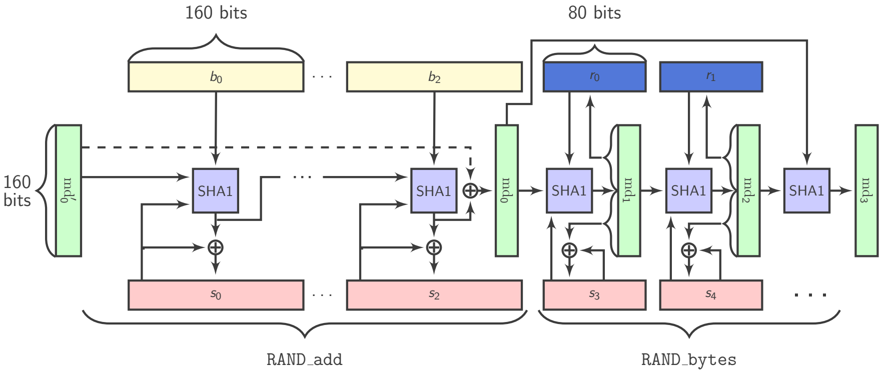
Core Cryptographic Function of OpenSSL's RNG

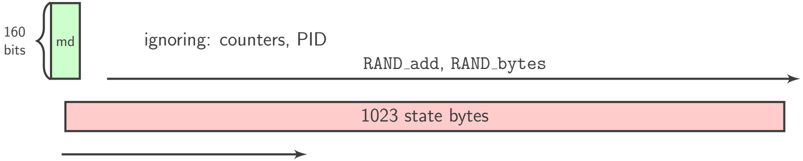




ignoring: counters, PID

Core Cryptographic Function of OpenSSL's RNG

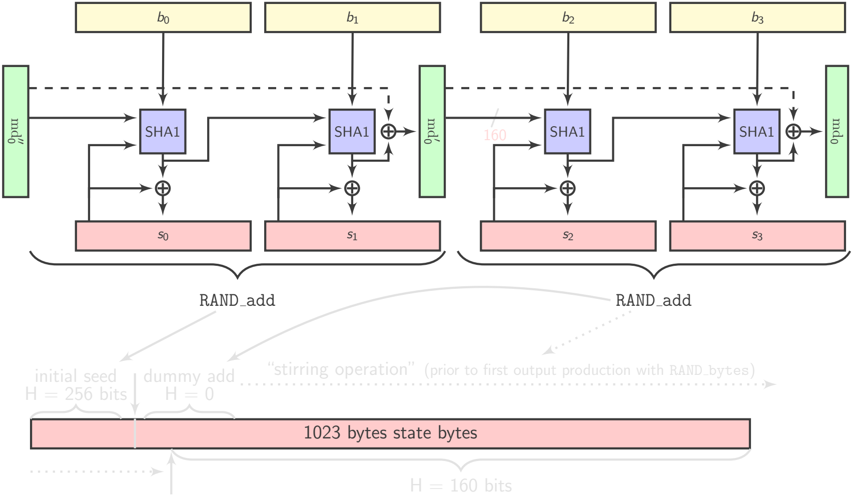




ignoring: counters, PID

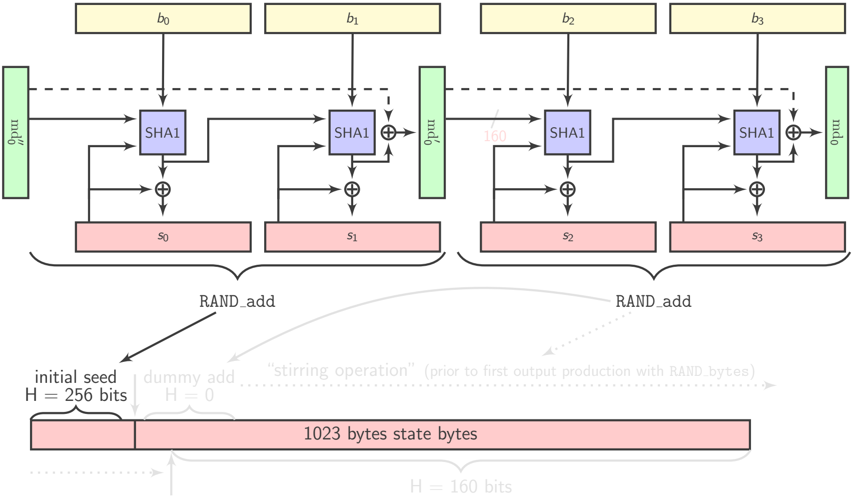
Output Entropy Limitation Vulnerabilities

ELO240



b_0
b_1
b_2
b_3
md_0''
md_0'
md_0
SHA1
SHA1
SHA1
SHA1
160
s_0
s_1
s_2
s_3
RAND_add
RAND_add
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
H = 160 bits

ELO240



RAND_add

RAND_add

initial seed
H = 256 bits

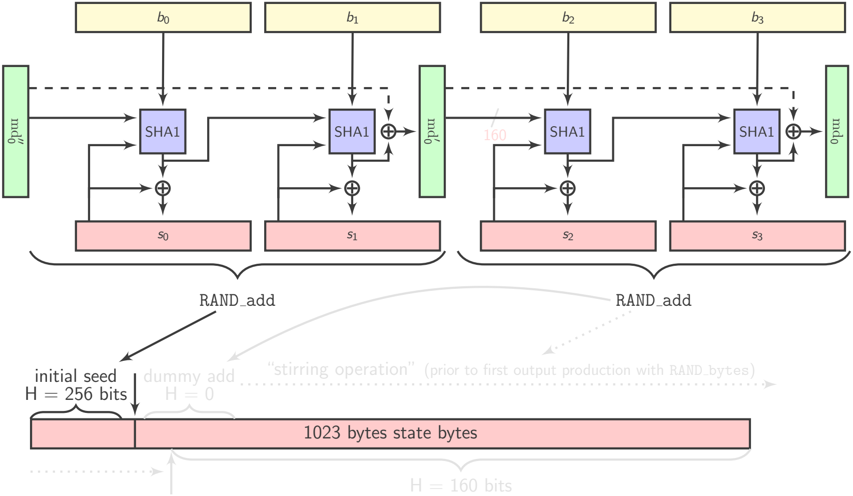
dummy add
H = 0

“stirring operation” (prior to first output production with RAND_bytes)

1023 bytes state bytes

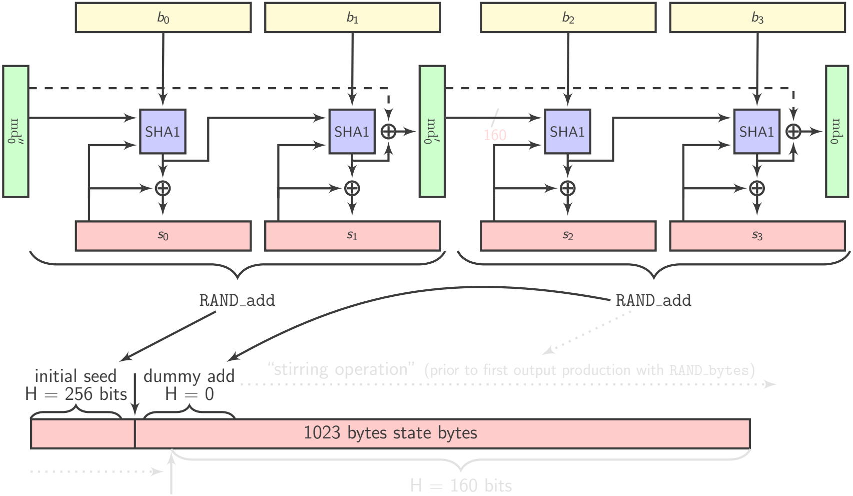
H = 160 bits

ELO240

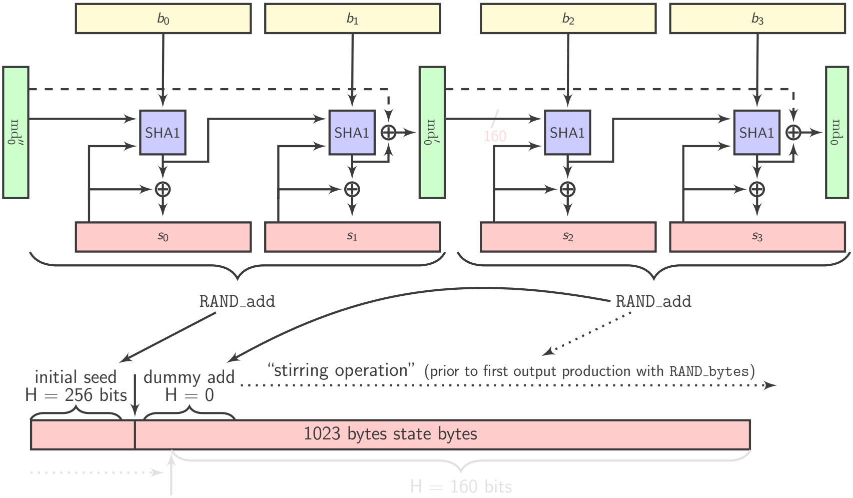


b_0
b_1
b_2
b_3
md_0''
md_0'
md_0
SHA1
SHA1
SHA1
SHA1
160
s_0
s_1
s_2
s_3
RAND_add
RAND_add
initial seed
H = 256 bits
dummy add
H = 0
“stirring operation” (prior to first output production with RAND_bytes)
1023 bytes state bytes
H = 160 bits

ELO240



b_0
b_1
b_2
b_3
md''
md'
md_0
SHA1
SHA1
SHA1
SHA1
160
s_0
s_1
s_2
s_3
RAND_add
RAND_add
"stirring operation" (prior to first output production with RAND_bytes)
initial seed
H = 256 bits
dummy add
H = 0
1023 bytes state bytes
H = 160 bits



RAND_add

RAND_add

initial seed
H = 256 bits

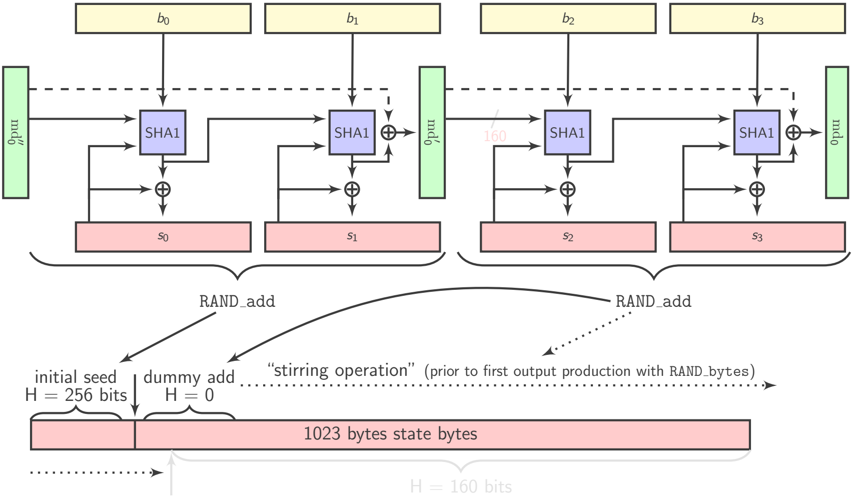
dummy add
H = 0

"stirring operation" (prior to first output production with RAND_bytes)

1023 bytes state bytes

H = 160 bits

ELO240



RAND_add

RAND_add

initial seed
H = 256 bits

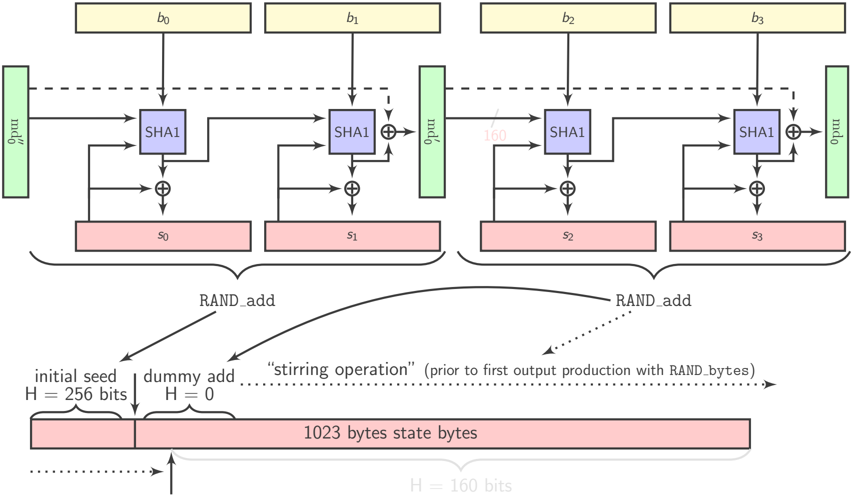
dummy add
H = 0

“stirring operation” (prior to first output production with RAND_bytes)

1023 bytes state bytes

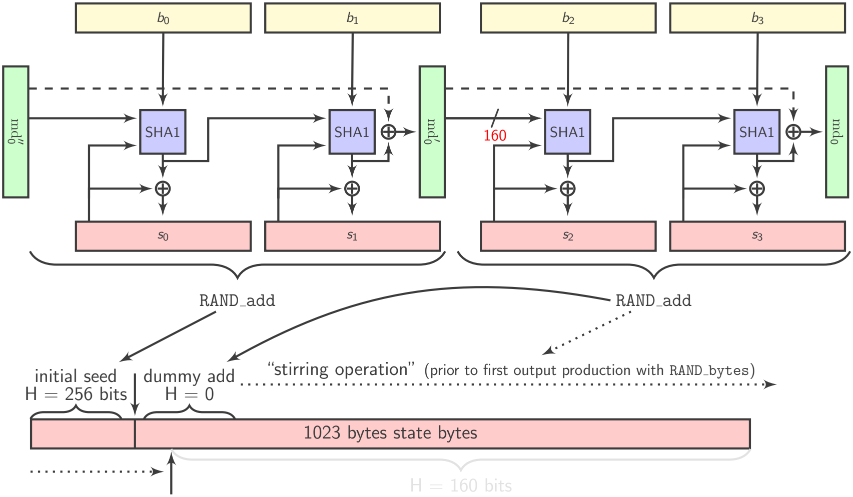
H = 160 bits

ELO240



b_0
b_1
b_2
b_3
md''
SHA1
SHA1
md'
160
SHA1
SHA1
md_0
s_0
s_1
s_2
s_3
RAND_add
RAND_add
initial seed
H = 256 bits
dummy add
H = 0
“stirring operation” (prior to first output production with RAND_bytes)
1023 bytes state bytes
H = 160 bits

ELO240

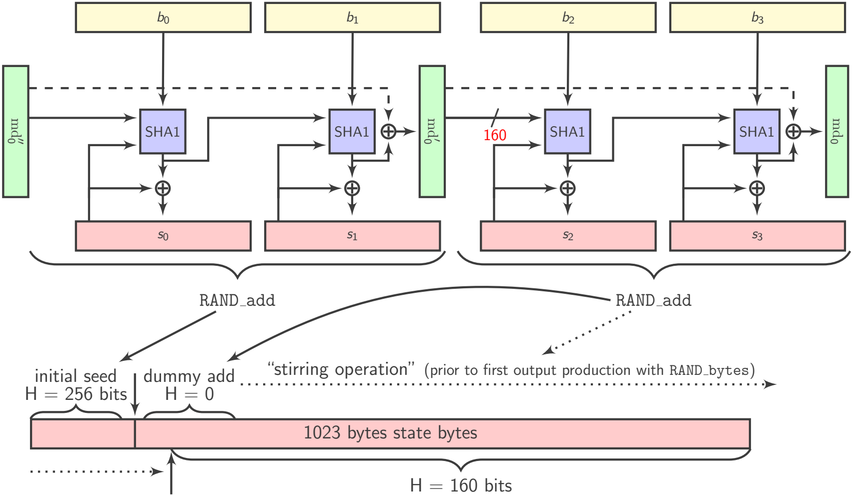


initial seed
H = 256 bits

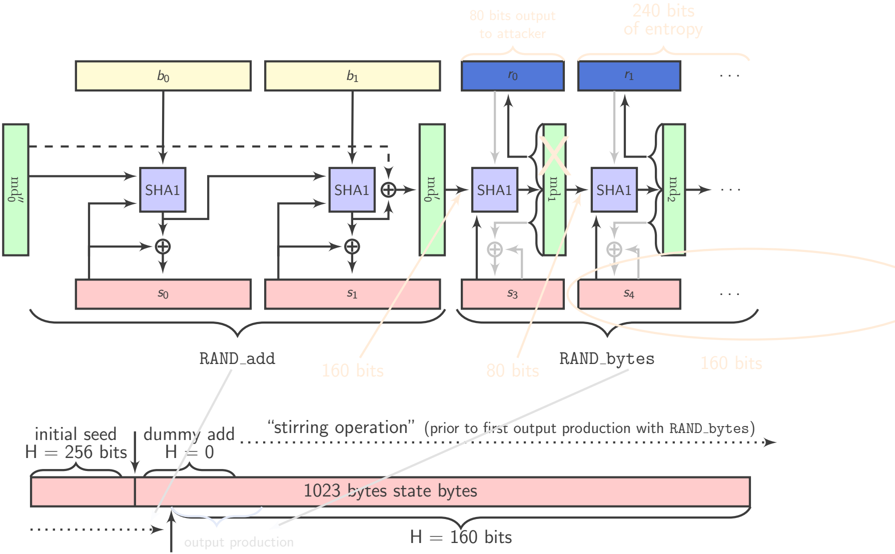
dummy add
H = 0

"stirring operation" (prior to first output production with RAND_bytes)

H = 160 bits


b_0
b_1
b_2
b_3
md''
md'
md
SHA1
SHA1
SHA1
SHA1
160
s_0
s_1
s_2
s_3
RAND_add
RAND_add
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
H = 160 bits

ELO240



80 bits output to attacker

240 bits of entropy

b_0 b_1 r_0 r_1 . . .

md_0'' SHA1 SHA1 ⊕ md_0' SHA1 md_1 SHA1 md_2 . . .

s_0 s_1 s_3 s_4 . . .

RAND_add 160 bits 80 bits RAND_bytes 160 bits

initial seed H = 256 bits

dummy add H = 0

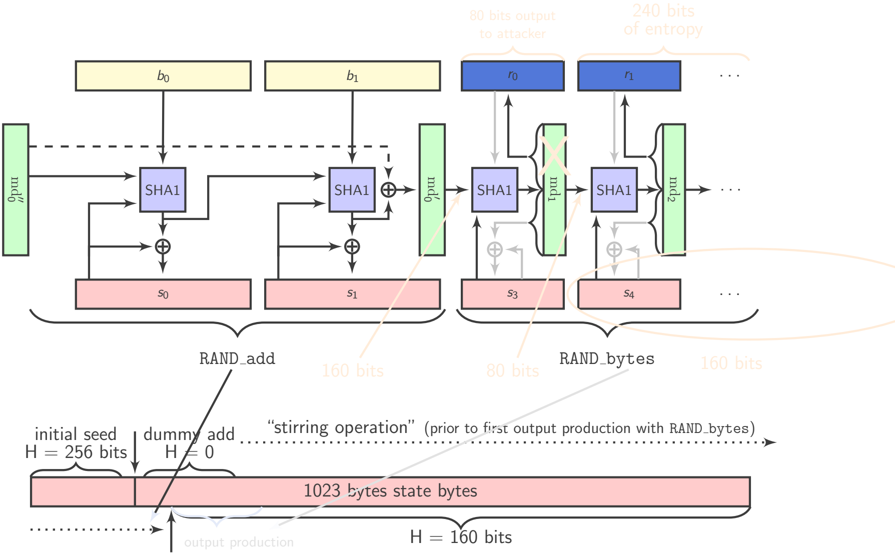
"stirring operation" (prior to first output production with RAND_bytes)

1023 bytes state bytes

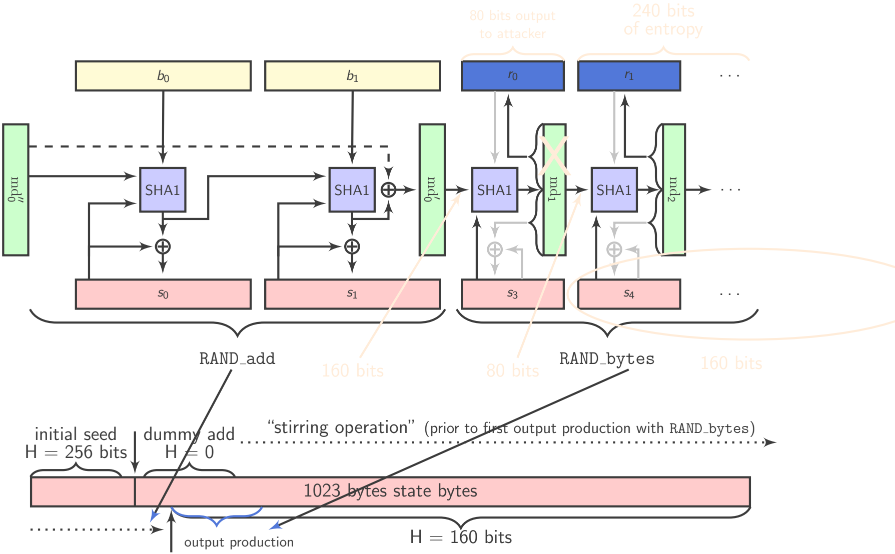
output production

H = 160 bits

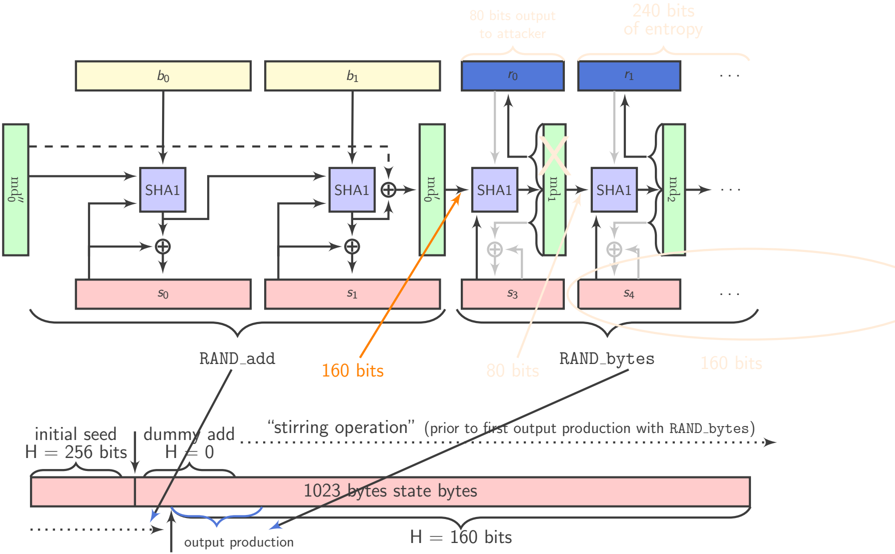
ELO240


80 bits output
to attacker
240 bits
of entropy
b_0
b_1
r_0
r_1
. . .
md$''_0$
SHA1
SHA1
md$'_0$
SHA1
md$_1$
SHA1
md$_2$
. . .
s_0
s_1
s_3
s_4
. . .
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

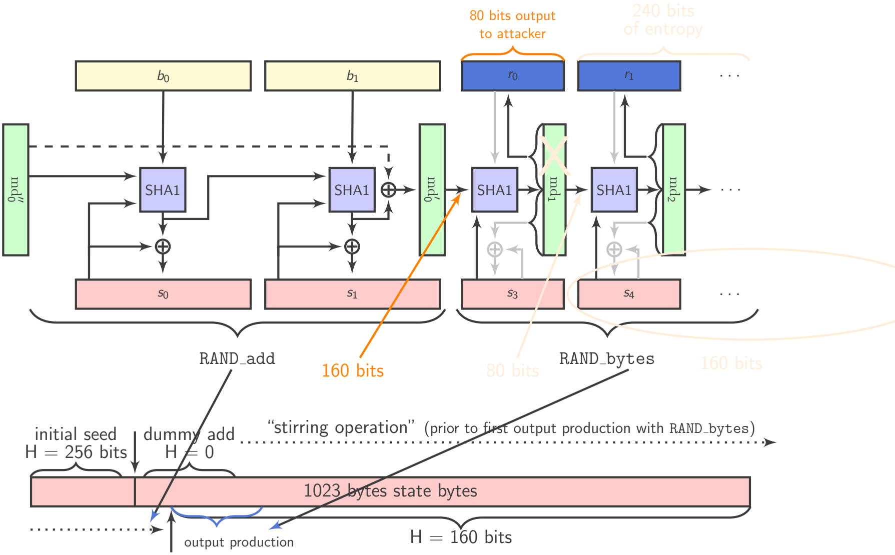
ELO240


80 bits output
to attacker
240 bits
of entropy
b_0
b_1
r_0
r_1
. . .
md_0''
SHA1
SHA1
$\oplus$
md_0'
SHA1
md_1
SHA1
md_2
. . .
$\oplus$
$\oplus$
$\oplus$
$\oplus$
s_0
s_1
s_3
s_4
. . .
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

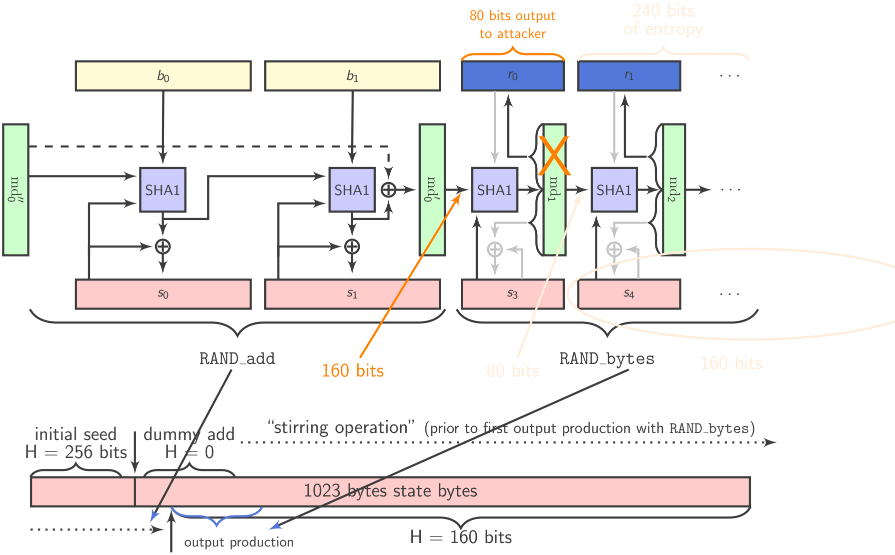
ELO240


80 bits output
to attacker
240 bits
of entropy
b_0
b_1
r_0
r_1
. . .
md_0''
SHA1
SHA1
⊕
md_0'
SHA1
md_1
SHA1
md_2
. . .
⊕
⊕
⊕
⊕
s_0
s_1
s_3
s_4
. . .
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

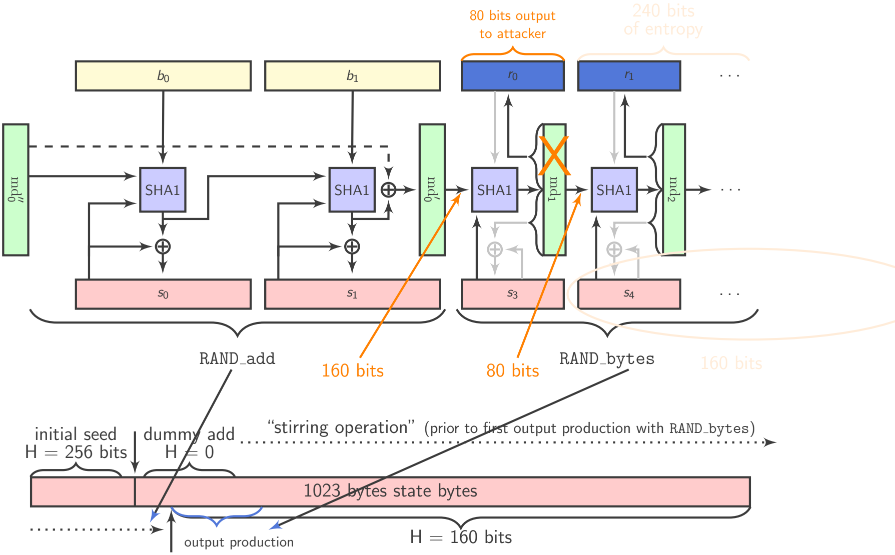
ELO240


80 bits output
to attacker
240 bits
of entropy
b_0
b_1
r_0
r_1
. . .
md_0''
SHA1
SHA1
md_0'
SHA1
md_1
SHA1
md_2
. . .
s_0
s_1
s_3
s_4
. . .
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

ELO240

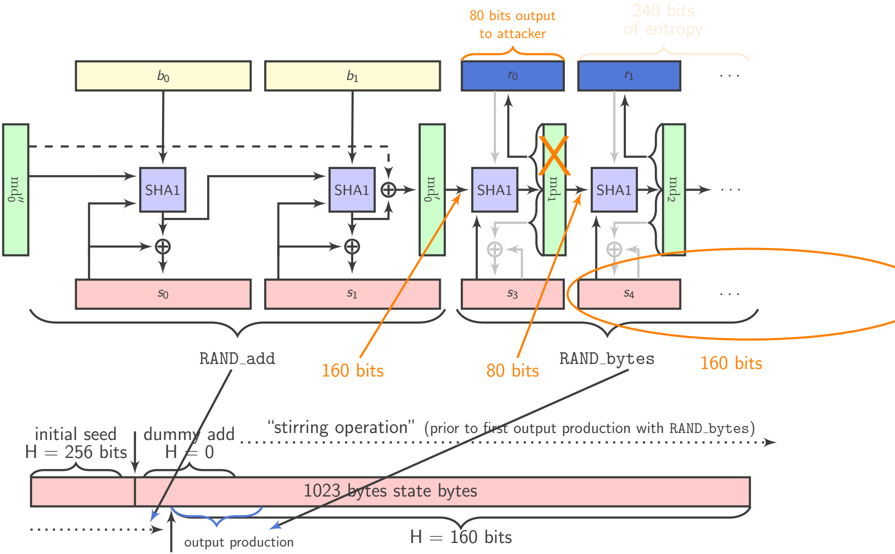

80 bits output
to attacker
240 bits
of entropy
b_0
b_1
r_0
r_1
. . .
md$''_0$
SHA1
SHA1
⊕
⊕
⊕
md$'_0$
SHA1
md$_1$
SHA1
md$_2$
. . .
s_0
s_1
s_3
s_4
. . .
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
“stirring operation” (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

ELO240



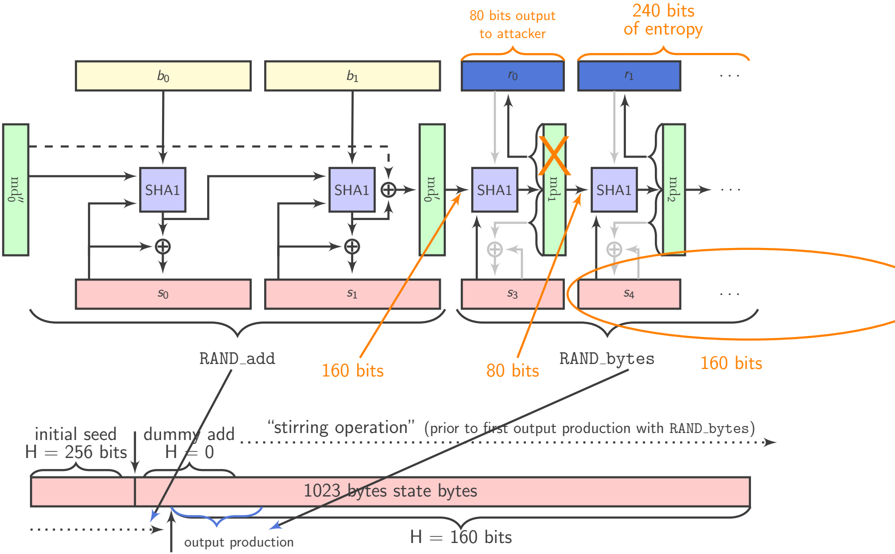
80 bits output
to attacker
240 bits
of entropy
b_0
b_1
r_0
r_1
. . .
md_0''
SHA1
SHA1
md_0'
SHA1
md_1
SHA1
md_2
. . .
s_0
s_1
s_3
s_4
. . .
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

ELO240



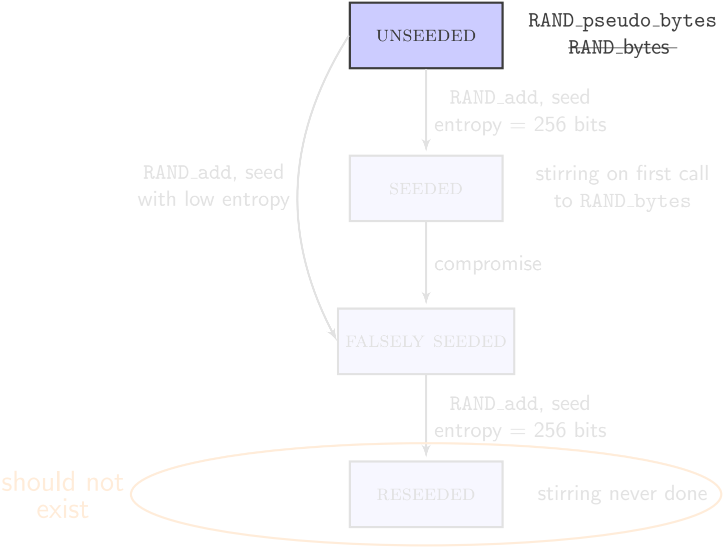
80 bits output
to attacker
240 bits
of entropy
b_0
b_1
r_0
r_1
. . .
md$''_0$
SHA1
SHA1
md$'_0$
SHA1
md$_1$
SHA1
md$_2$
. . .
s_0
s_1
s_3
s_4
. . .
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

ELO240

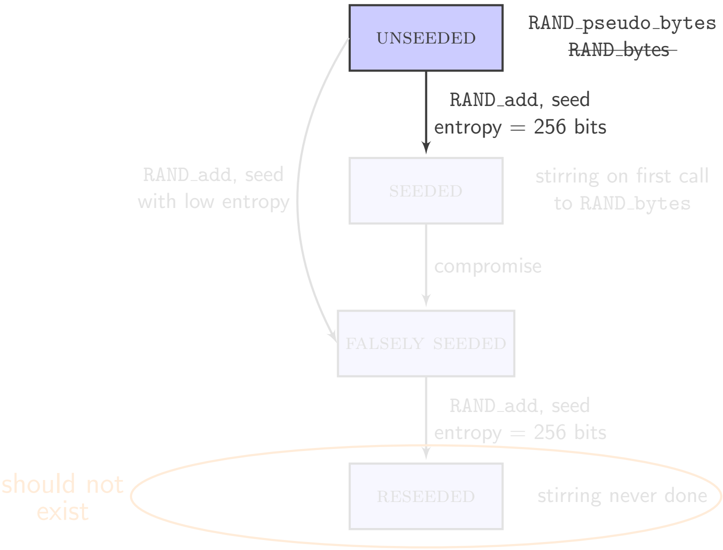


80 bits output to attacker
240 bits of entropy
b_0
b_1
r_0
r_1
. . .
md''_0
SHA1
SHA1
md'_0
SHA1
md_1
SHA1
md_2
X
s_0
s_1
s_3
s_4
RAND_add
160 bits
80 bits
RAND_bytes
160 bits
initial seed
H = 256 bits
dummy add
H = 0
"stirring operation" (prior to first output production with RAND_bytes)
1023 bytes state bytes
output production
H = 160 bits

Life cycle


UNSEEDED
RAND_pseudo_bytes
~~RAND_bytes~~
RAND_add, seed
entropy = 256 bits
RAND_add, seed
with low entropy
SEEDED
stirring on first call
to RAND_bytes
compromise
FALSELY SEEDED
RAND_add, seed
entropy = 256 bits
should not
exist
RESEEDED
stirring never done

Life cycle



UNSEEDED

RAND_pseudo_bytes
~~RAND_bytes~~

RAND_add, seed
entropy = 256 bits

RAND_add, seed
with low entropy

SEEDED

stirring on first call
to RAND_bytes

compromise

FALSELY SEEDED

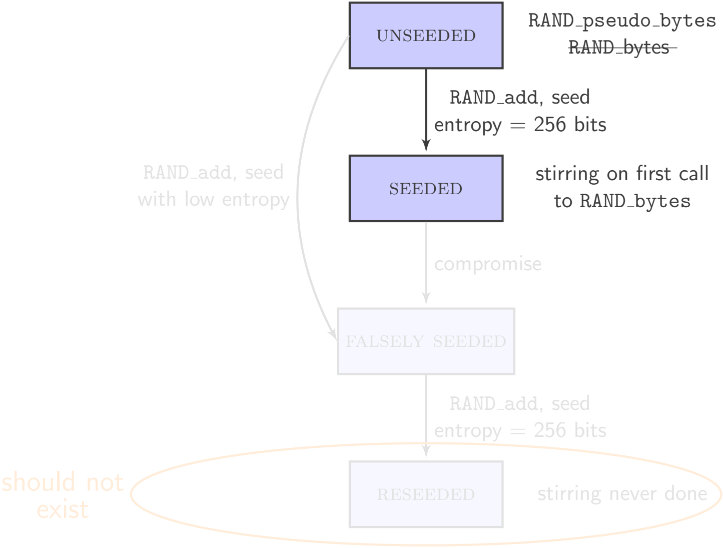
RAND_add, seed
entropy = 256 bits

should not
exist

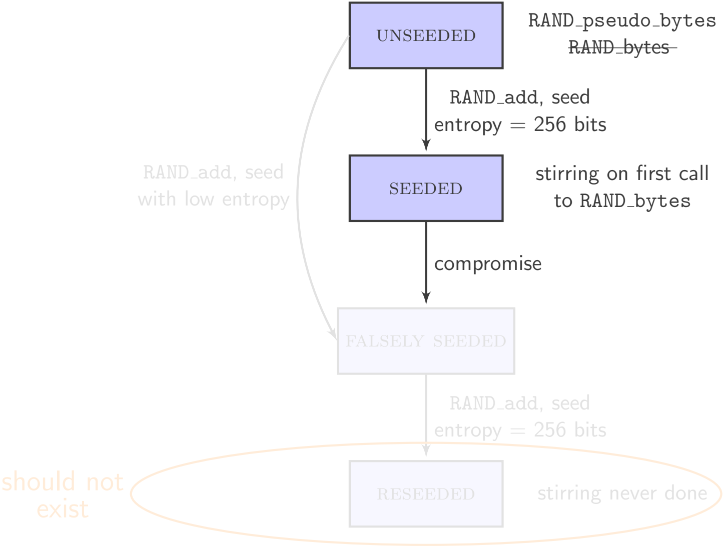
RESEEDED

stirring never done

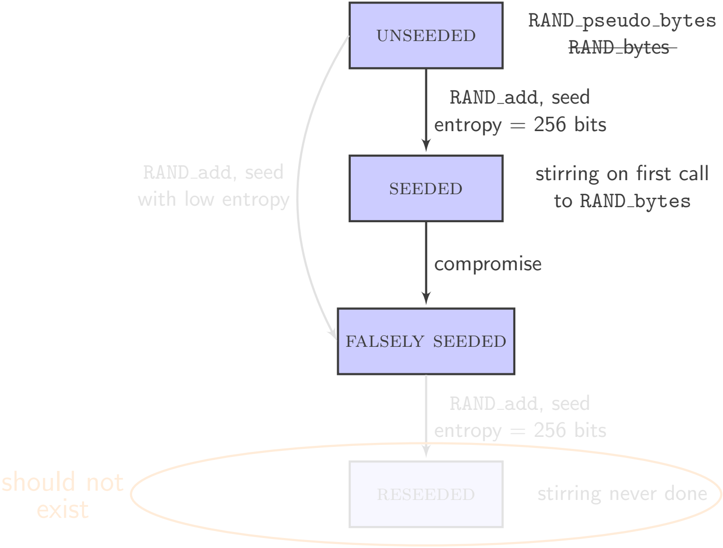
Life cycle


UNSEEDED
RAND_pseudo_bytes
~~RAND_bytes~~
RAND_add, seed
entropy = 256 bits
SEEDED
stirring on first call
to RAND_bytes
RAND_add, seed
with low entropy
compromise
FALSELY SEEDED
RAND_add, seed
entropy = 256 bits
RESEEDED
stirring never done
should not
exist

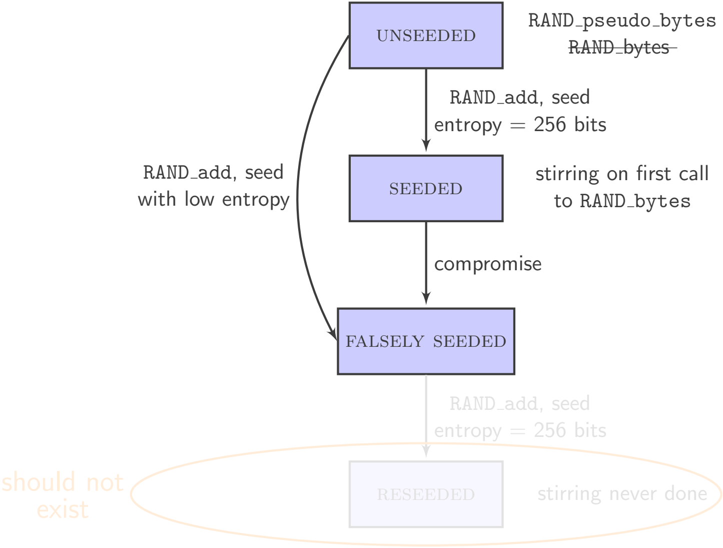
Life cycle


UNSEEDED
RAND_pseudo_bytes
~~RAND_bytes~~
RAND_add, seed
entropy = 256 bits
SEEDED
stirring on first call
to RAND_bytes
compromise
RAND_add, seed
with low entropy
FALSELY SEEDED
RAND_add, seed
entropy = 256 bits
RESEEDED
stirring never done
should not
exist

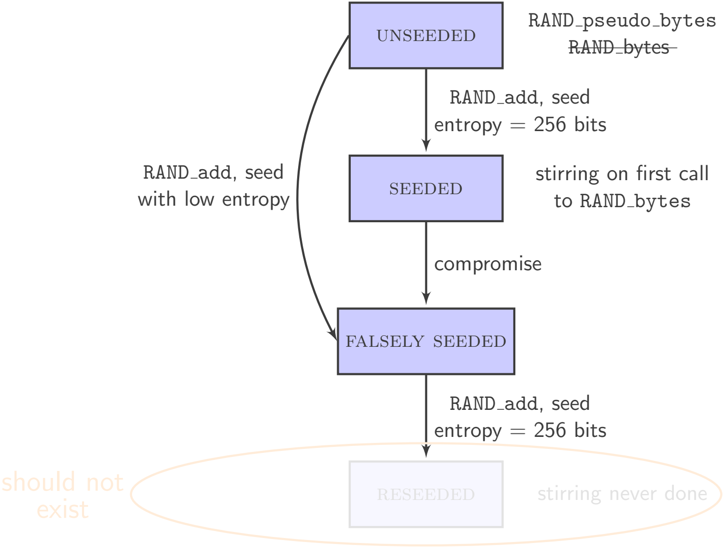
Life cycle


UNSEEDED
RAND_pseudo_bytes
~~RAND_bytes~~
RAND_add, seed
entropy = 256 bits
SEEDED
stirring on first call
to RAND_bytes
RAND_add, seed
with low entropy
compromise
FALSELY SEEDED
RAND_add, seed
entropy = 256 bits
should not
exist
RESEEDED
stirring never done

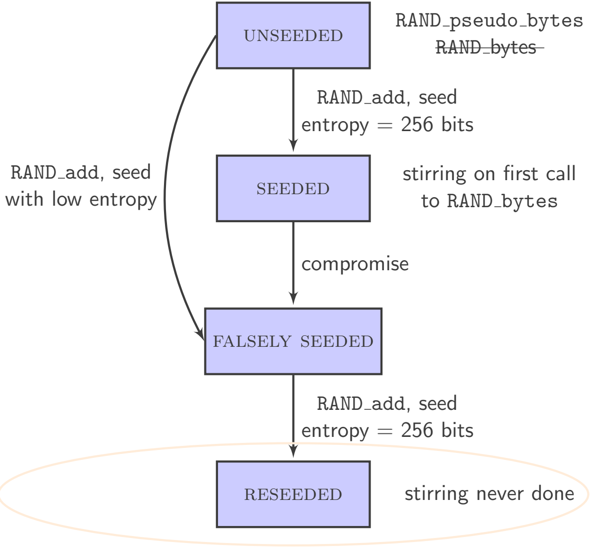
Life cycle


UNSEEDED
RAND_pseudo_bytes
~~RAND_bytes~~
RAND_add, seed
entropy = 256 bits
SEEDED
stirring on first call
to RAND_bytes
RAND_add, seed
with low entropy
compromise
FALSELY SEEDED
RAND_add, seed
entropy = 256 bits
should not
exist
RESEEDED
stirring never done

Life cycle


UNSEEDED
RAND_pseudo_bytes
~~RAND_bytes~~
RAND_add, seed
entropy = 256 bits
SEEDED
stirring on first call
to RAND_bytes
RAND_add, seed
with low entropy
compromise
FALSELY SEEDED
RAND_add, seed
entropy = 256 bits
RESEEDED
stirring never done
should not
exist

Life cycle



UNSEEDED

RAND_pseudo_bytes
~~RAND_bytes~~

RAND_add, seed
entropy = 256 bits

SEEDED

stirring on first call
to RAND_bytes

RAND_add, seed
with low entropy

compromise

FALSELY SEEDED

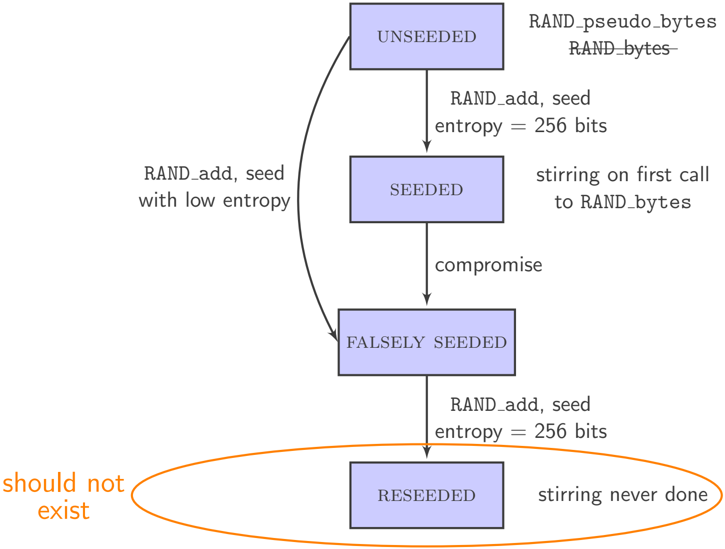
RAND_add, seed
entropy = 256 bits

RESEEDED

stirring never done

should not
exist

Life cycle



UNSEEDED

RAND_pseudo_bytes
~~RAND_bytes~~

RAND_add, seed
entropy = 256 bits

SEEDED

stirring on first call
to RAND_bytes

RAND_add, seed
with low entropy

compromise

FALSELY SEEDED

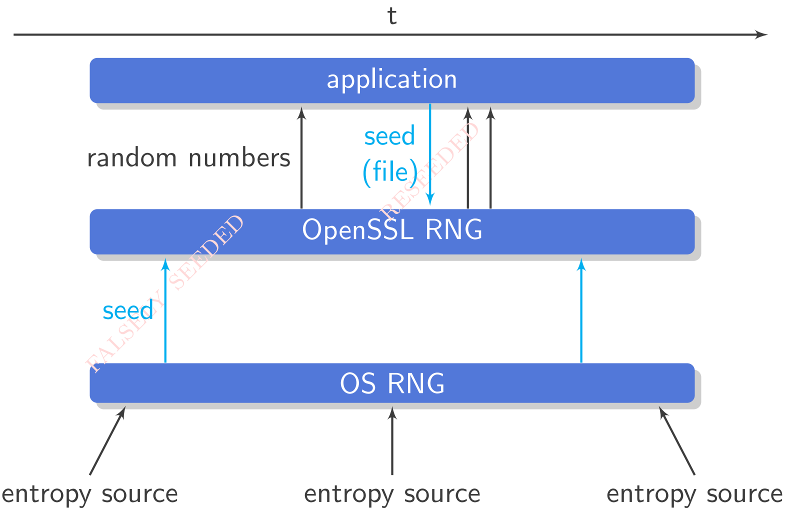
RAND_add, seed
entropy = 256 bits

RESEEDED

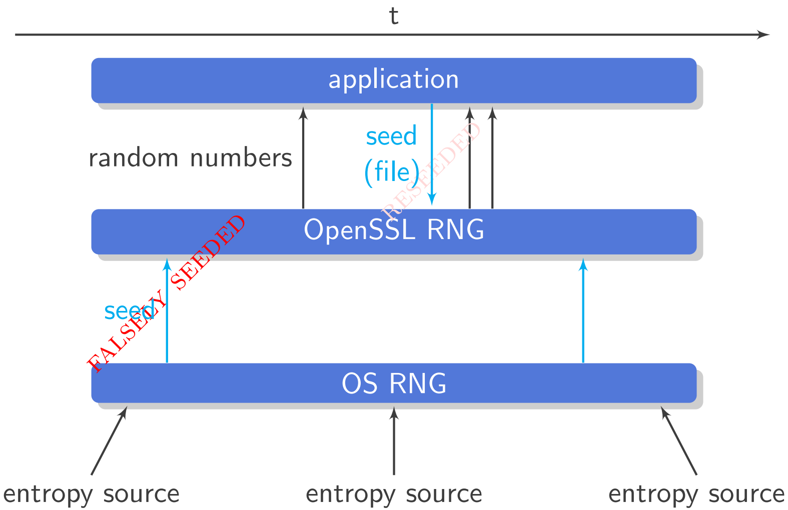
stirring never done

should not
exist

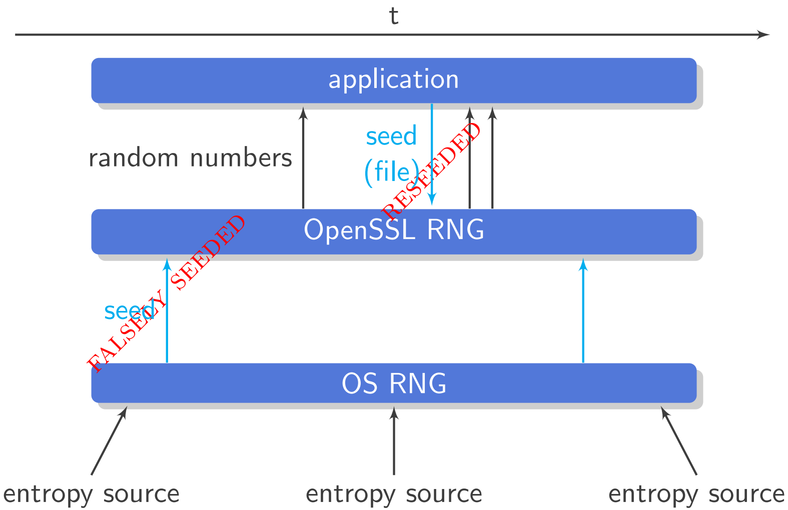
Reseeded State in Practice



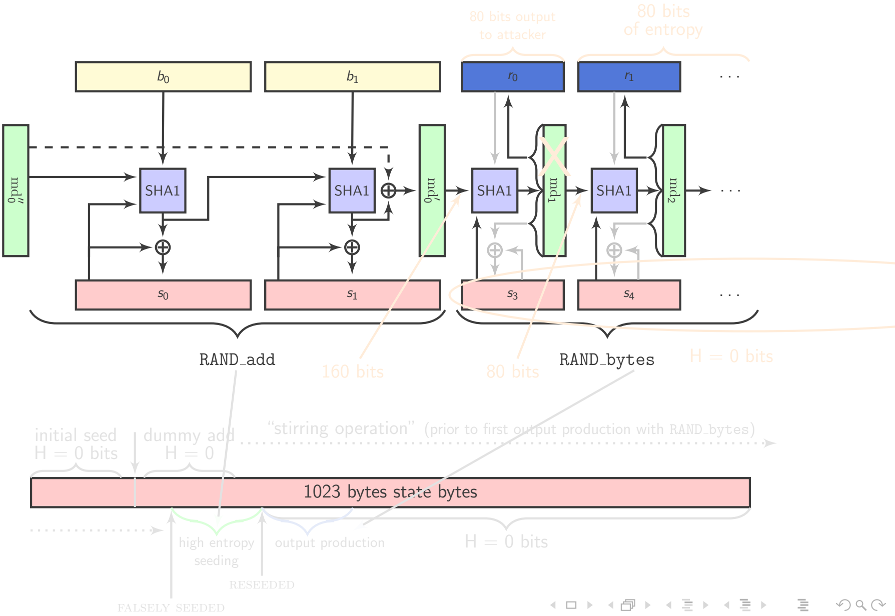
Reseeded State in Practice



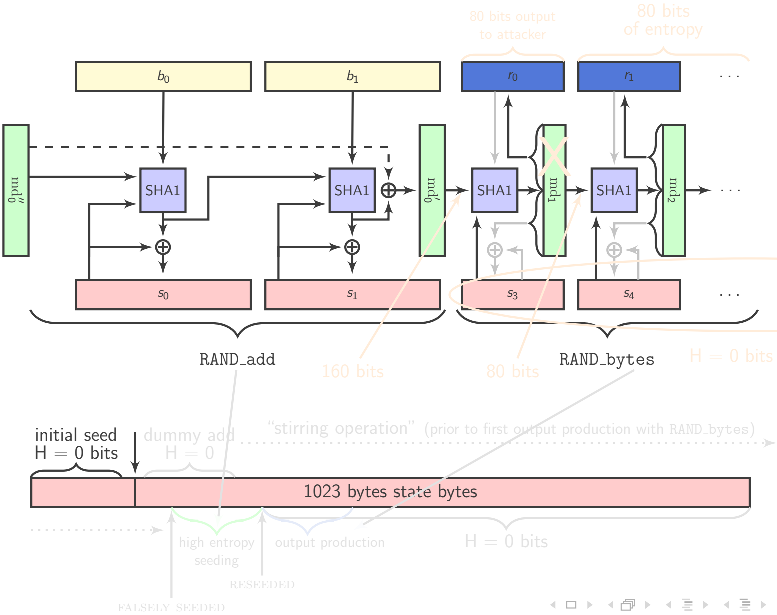
Reseeded State in Practice



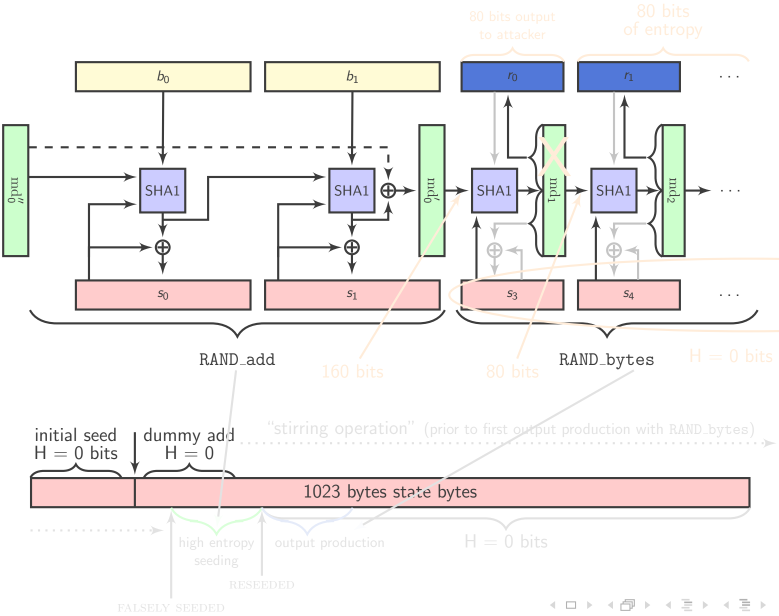
ELO160 / ELO80 in State RESEEDED



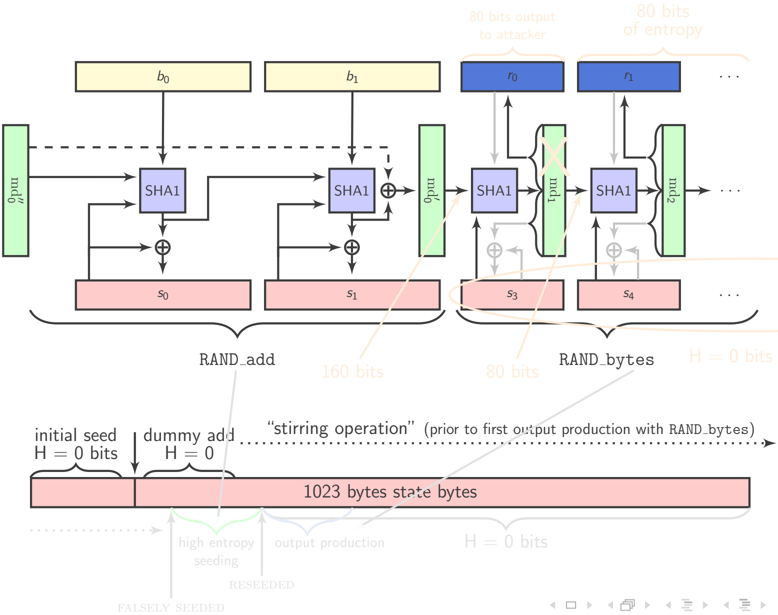
ELO160 / ELO80 in State RESEEDED



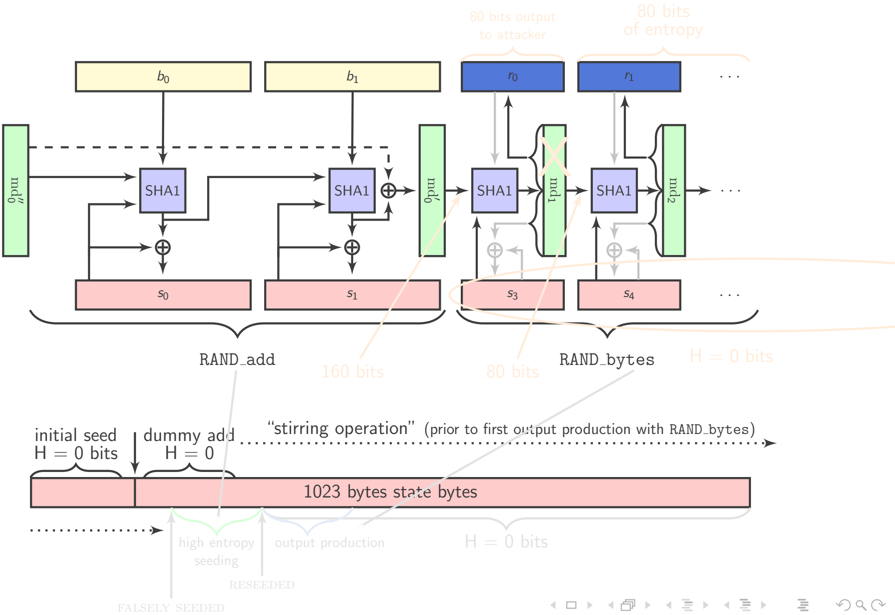
ELO160 / ELO80 in State RESEEDED



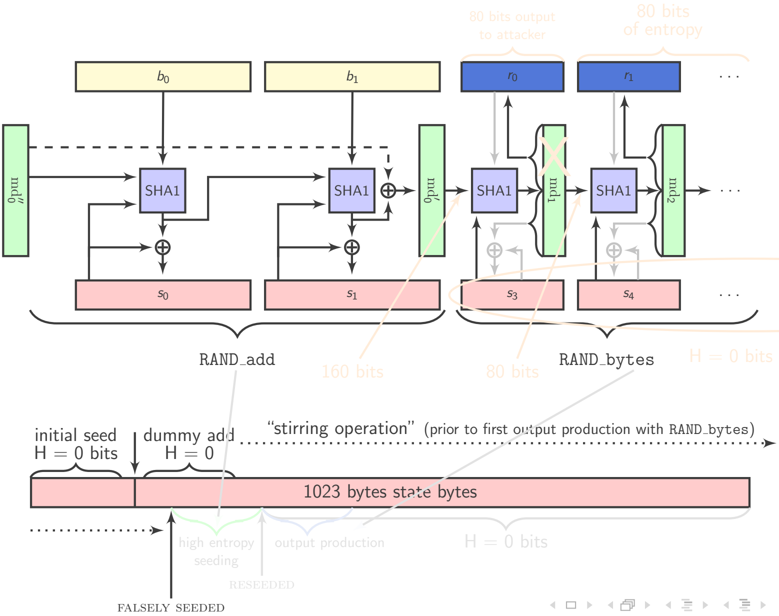
ELO160 / ELO80 in State RESEEDED



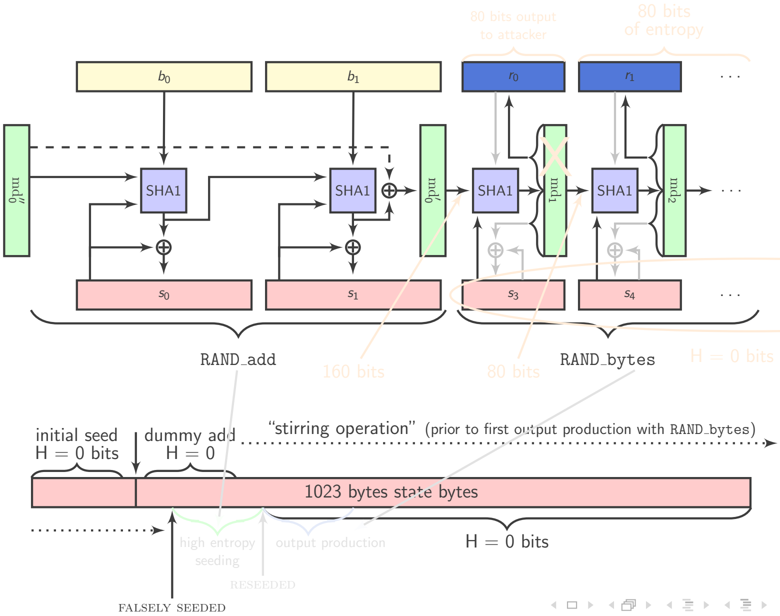
ELO160 / ELO80 in State RESEEDED



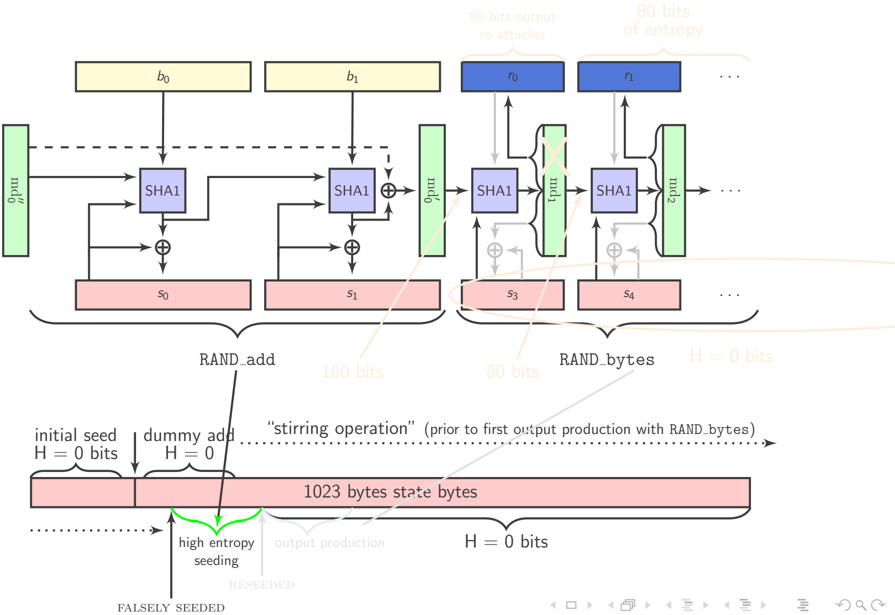
ELO160 / ELO80 in State RESEEDED



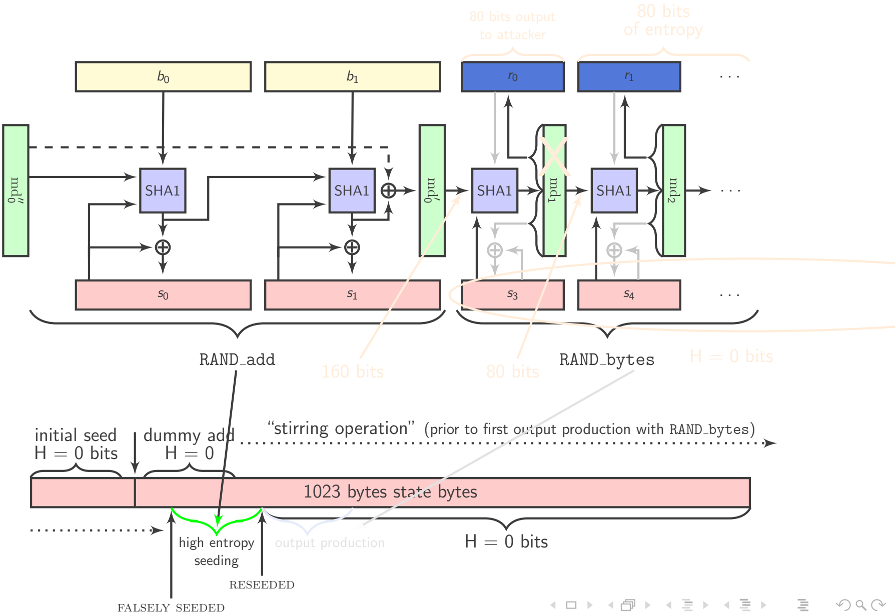
ELO160 / ELO80 in State RESEEDED



ELO160 / ELO80 in State RESEEDED



ELO160 / ELO80 in State RESEEDED



80 bits output to attacker

80 bits of entropy

b_0

b_1

r_0

r_1

. . .

md_0''

SHA1

SHA1

md_0'

SHA1

md_1

SHA1

md_2

. . .

s_0

s_1

s_3

s_4

. . .

RAND_add

160 bits

80 bits

RAND_bytes

H = 0 bits

initial seed
H = 0 bits

dummy add
H = 0

“stirring operation” (prior to first output production with RAND_bytes)

1023 bytes state bytes

high entropy seeding

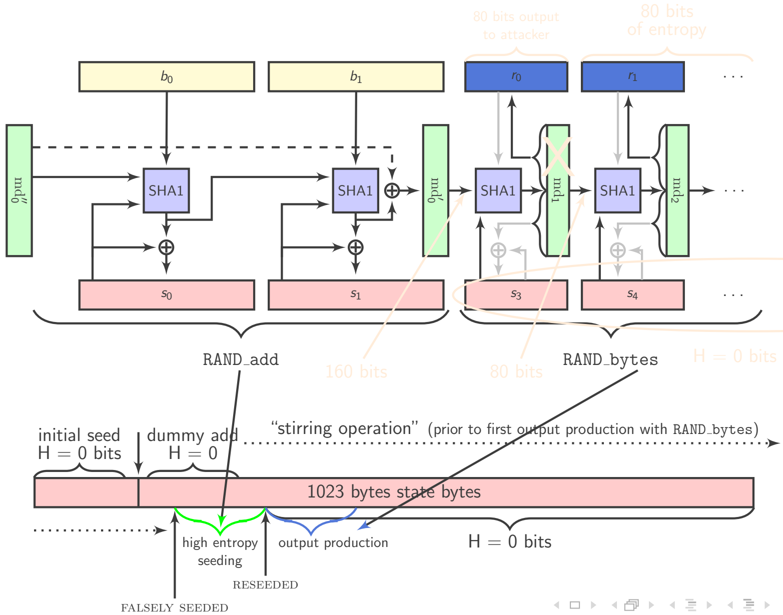
output production

H = 0 bits

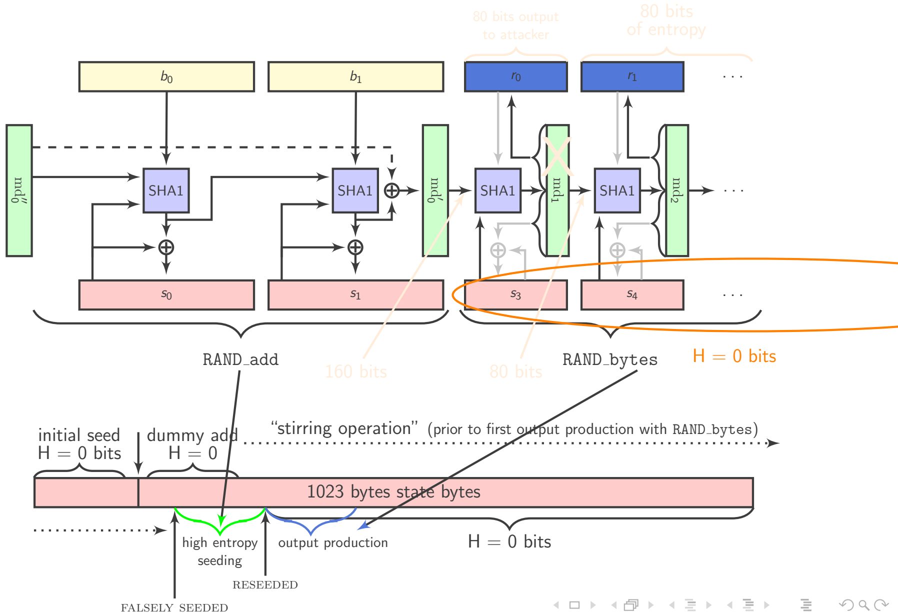
RESEEDED

FALSELY SEEDED

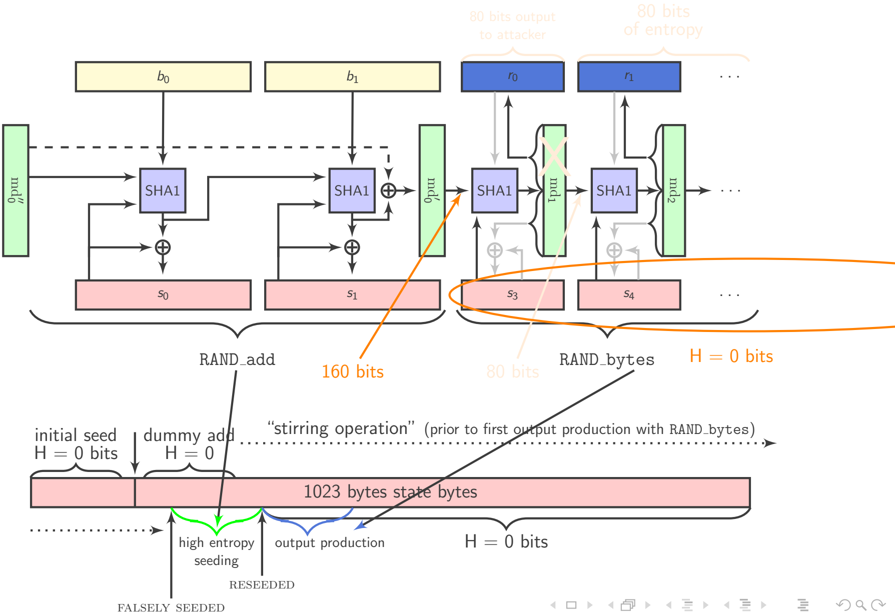
ELO160 / ELO80 in State RESEEDED



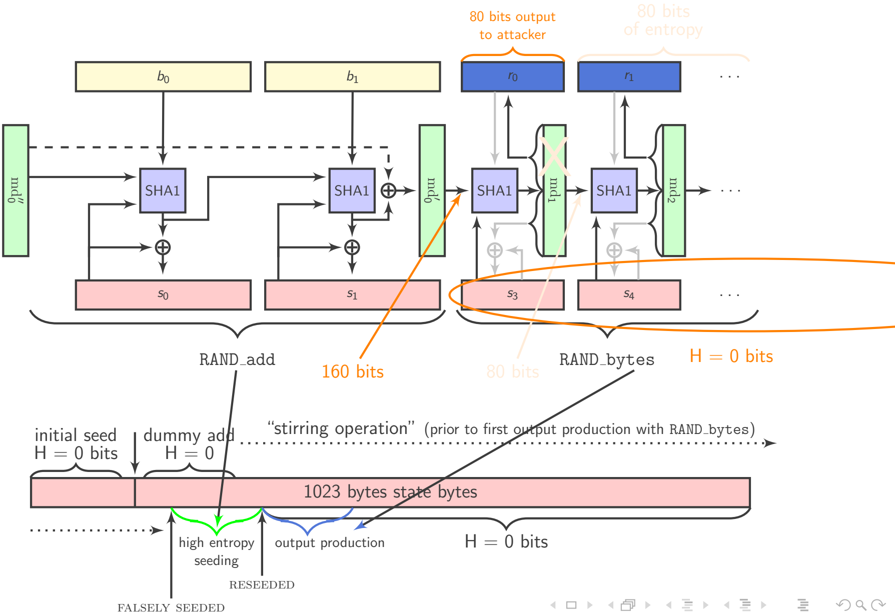
ELO160 / ELO80 in State RESEEDED



ELO160 / ELO80 in State RESEEDED



ELO160 / ELO80 in State RESEEDED



80 bits output to attacker

80 bits of entropy

RAND_add

160 bits

80 bits

RAND_bytes

H = 0 bits

initial seed H = 0 bits

dummy add H = 0

“stirring operation” (prior to first output production with RAND_bytes)

1023 bytes state bytes

high entropy seeding

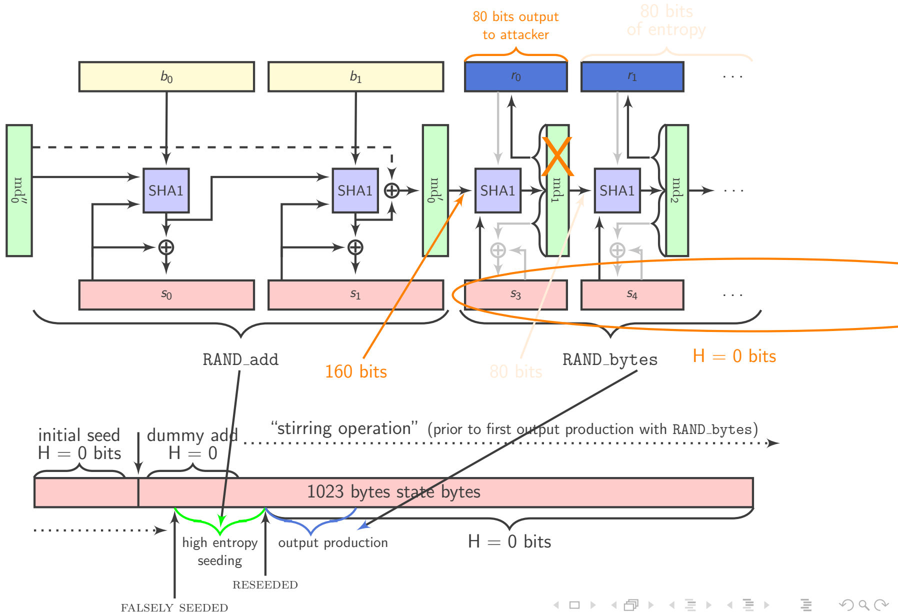
output production

H = 0 bits

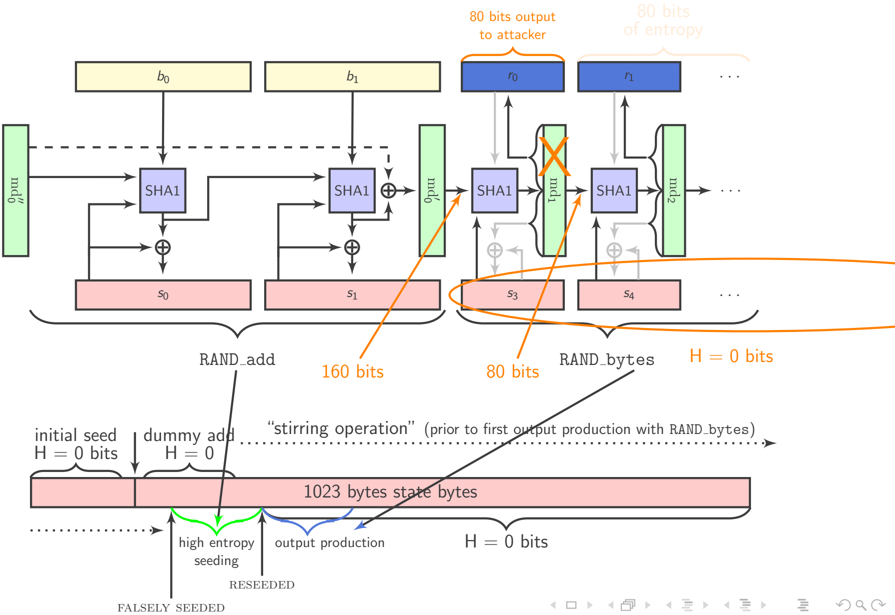
RESEEDED

FALSELY SEEDED

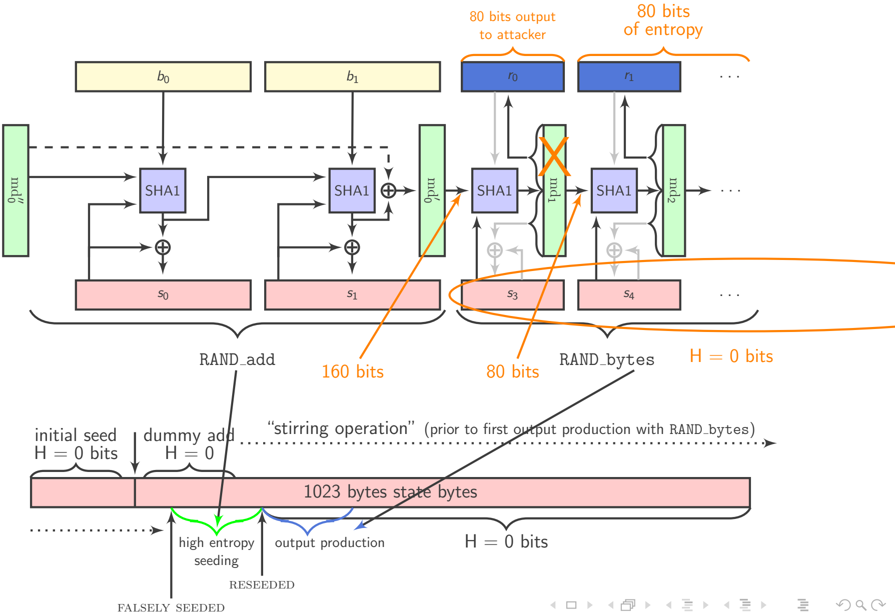
ELO160 / ELO80 in State RESEEDED



ELO160 / ELO80 in State RESEEDED



ELO160 / ELO80 in State RESEEDED



Attacks so far

- ELO-240: purely cosmetic
- ELO-160: not exploitable
- ELO-80: only predict output from same call to `RAND_bytes`
- can we do better? $\approx 2^{80}$ and more realistic conditions?

Attacks so far

- ELO-240: purely cosmetic
- ELO-160: not exploitable
- ELO-80: only predict output from same call to `RAND_bytes`
- can we do better? $\approx 2^{80}$ and more realistic conditions?

Attacks so far

- ELO-240: purely cosmetic
- ELO-160: not exploitable
- ELO-80: only predict output from same call to `RAND_bytes`
- can we do better? $\approx 2^{80}$ and more realistic conditions?

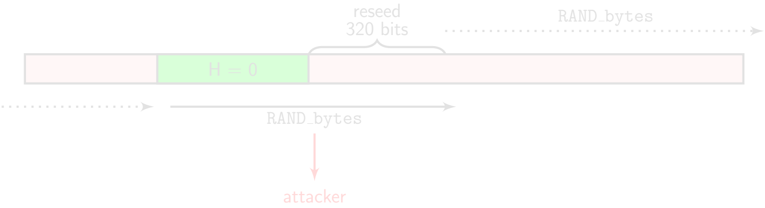
Attacks so far

- ELO-240: purely cosmetic
- ELO-160: not exploitable
- ELO-80: only predict output from same call to `RAND_bytes`
- can we do better? $\approx 2^{80}$ and more realistic conditions?

State Recovery Attacks

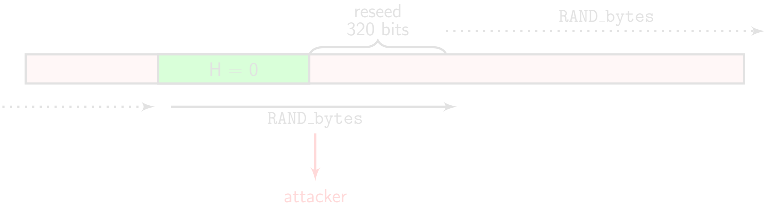
State Recovery Attack: DEJA-STATE

- RNG in low entropy state
- high entropy reseeding
- in RESEEDED state
- goal: recover RNG state after reseeding



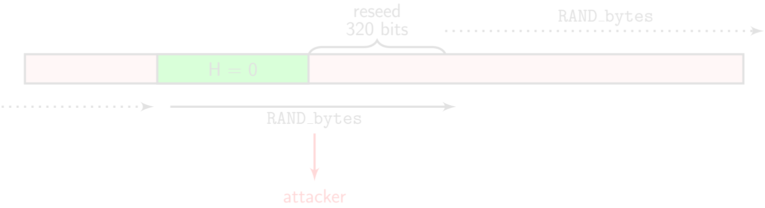
State Recovery Attack: DEJA-STATE

- RNG in low entropy state
- high entropy reseeding
- in RESEEDED state
- goal: recover RNG state after reseeding



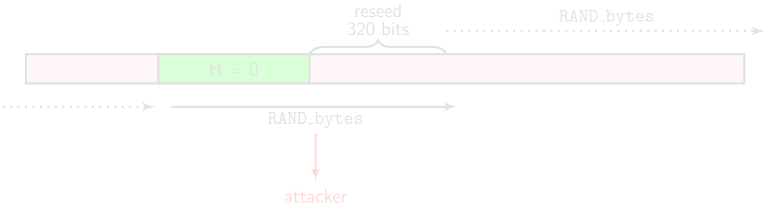
State Recovery Attack: DEJA-STATE

- RNG in low entropy state
- high entropy reseeding
- in RESEEDED state
- goal: recover RNG state after reseeding



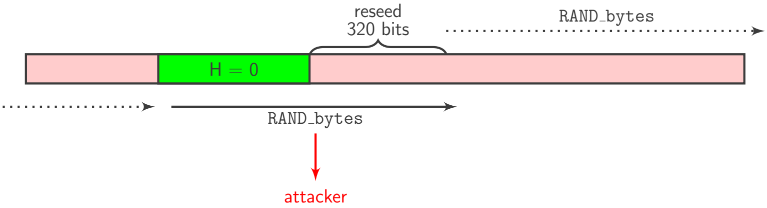
State Recovery Attack: DEJA-STATE

- RNG in low entropy state
- high entropy reseeding
- in RESEEDED state
- goal: recover RNG state after reseeding

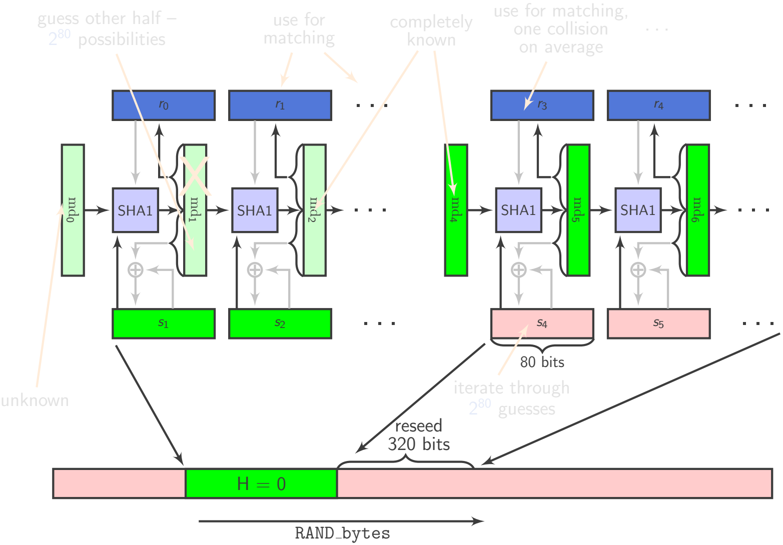


State Recovery Attack: DEJA-STATE

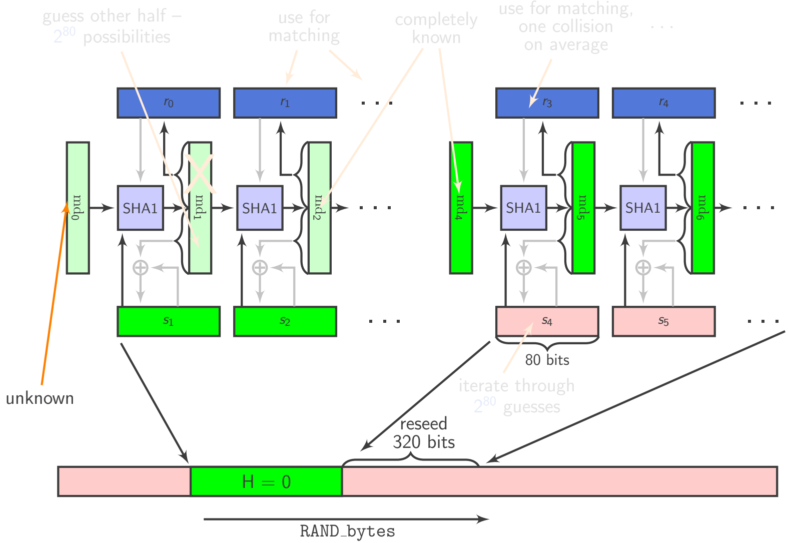
- RNG in low entropy state
- high entropy reseeding
- in RESEEDED state
- goal: recover RNG state after reseeding



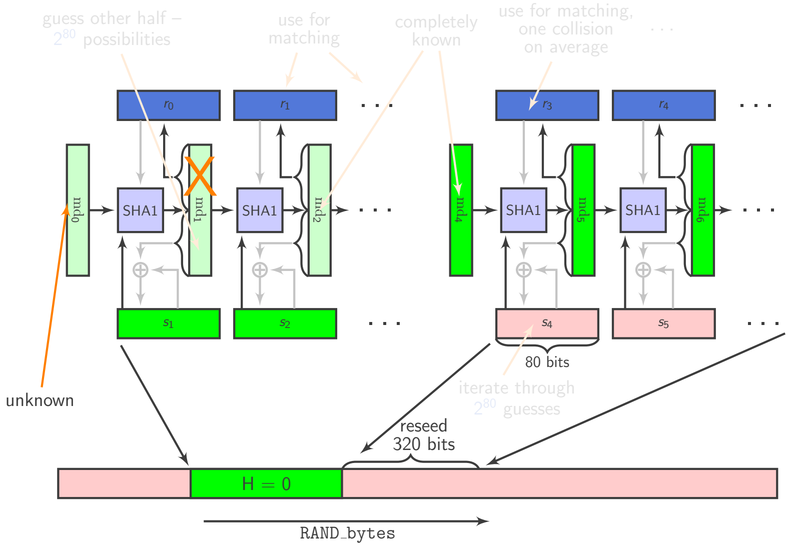
State Recovery Attack: DEJA-STATE



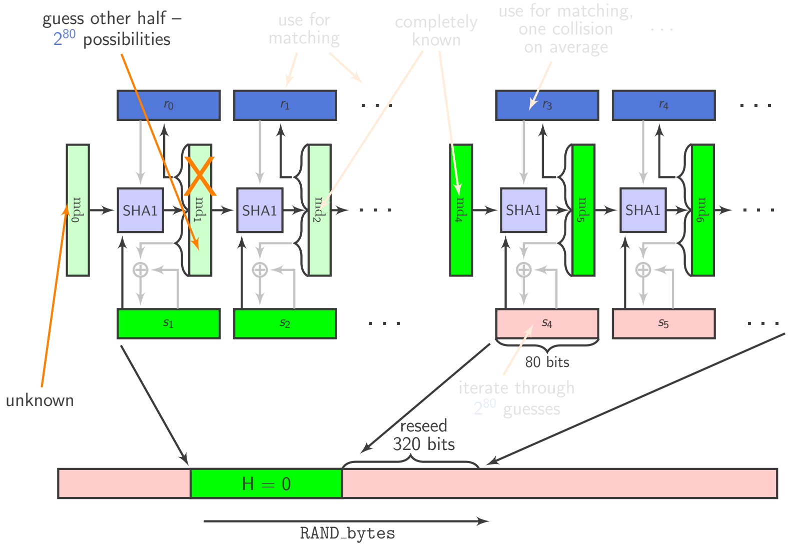
State Recovery Attack: DEJA-STATE



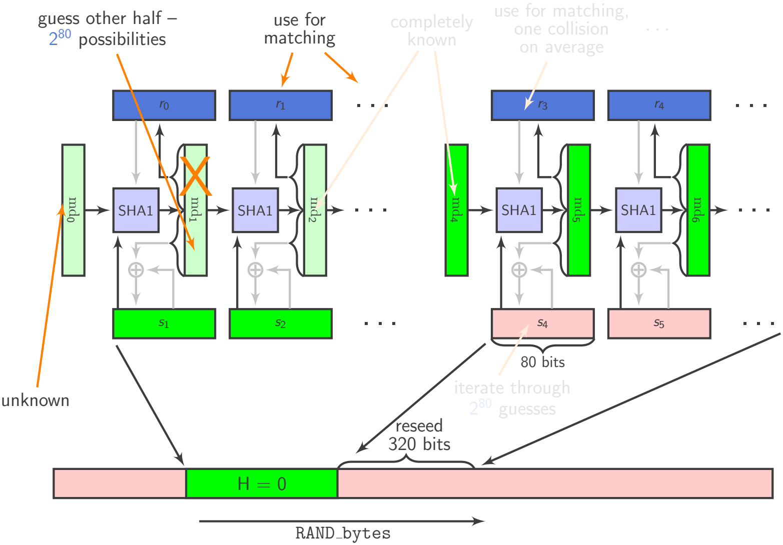
State Recovery Attack: DEJA-STATE



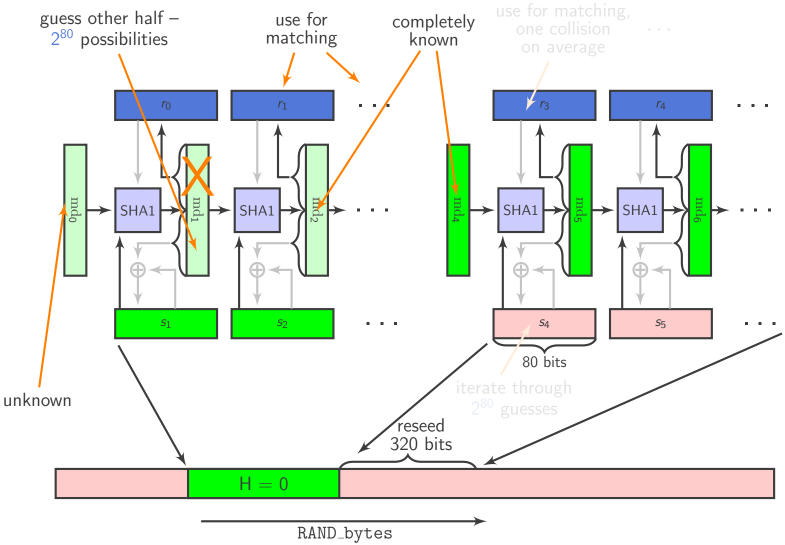
State Recovery Attack: DEJA-STATE



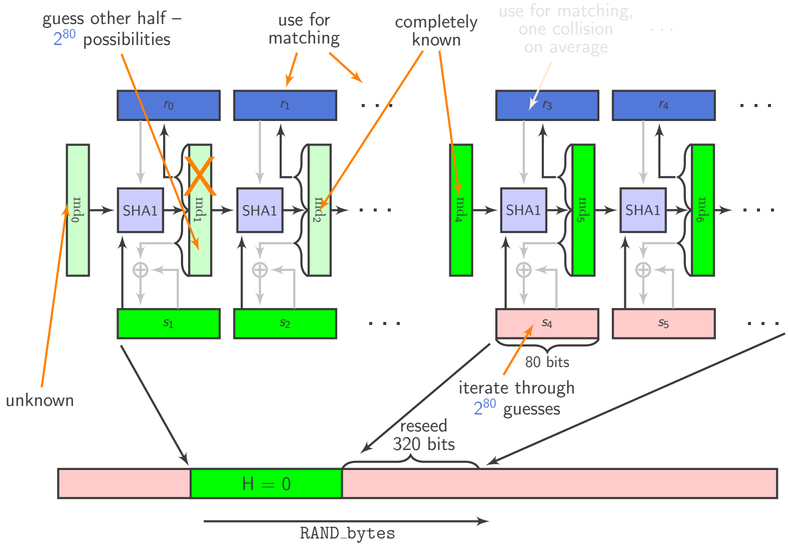
State Recovery Attack: DEJA-STATE



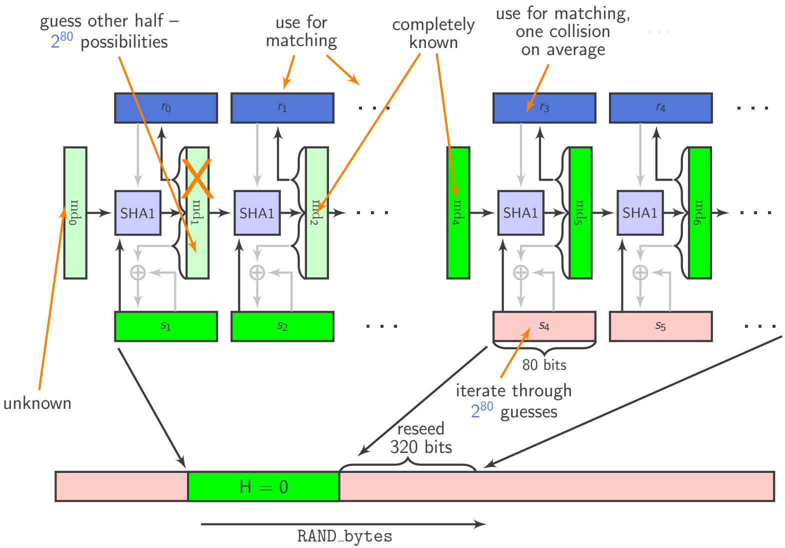
State Recovery Attack: DEJA-STATE



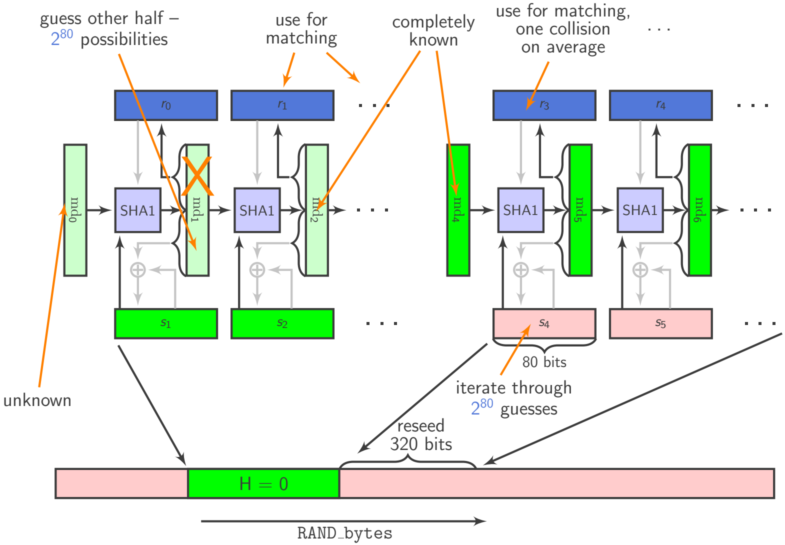
State Recovery Attack: DEJA-STATE



State Recovery Attack: DEJA-STATE

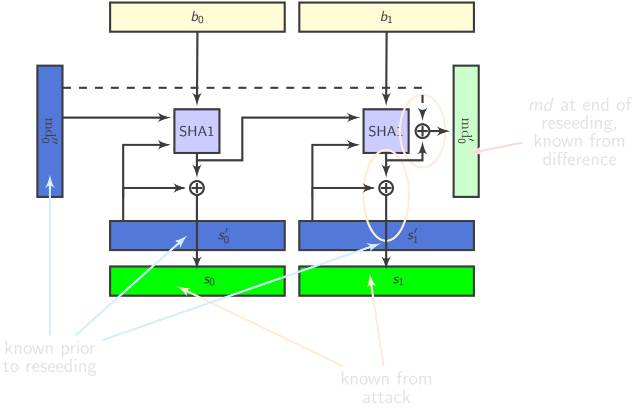


State Recovery Attack: DEJA-STATE



State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



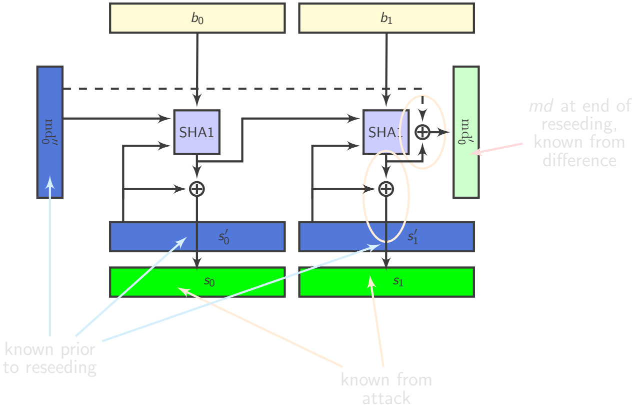
md at end of reseeding, known from difference

known prior to reseeding

known from attack

State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



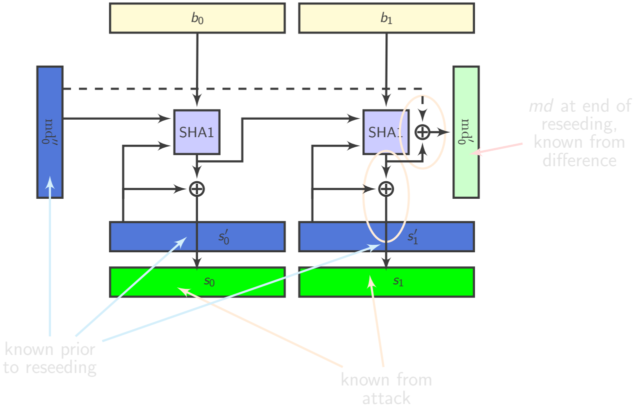
md at end of reseeding, known from difference

known prior to reseeding

known from attack

State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



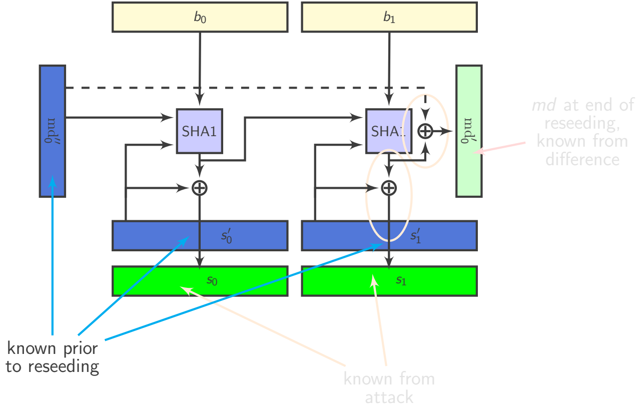
md at end of reseeding, known from difference

known prior to reseeding

known from attack

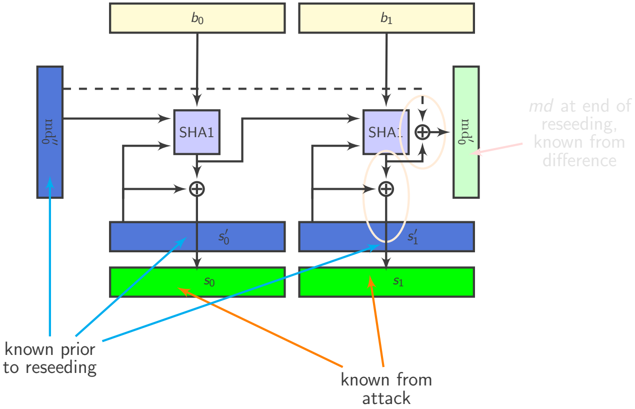
State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



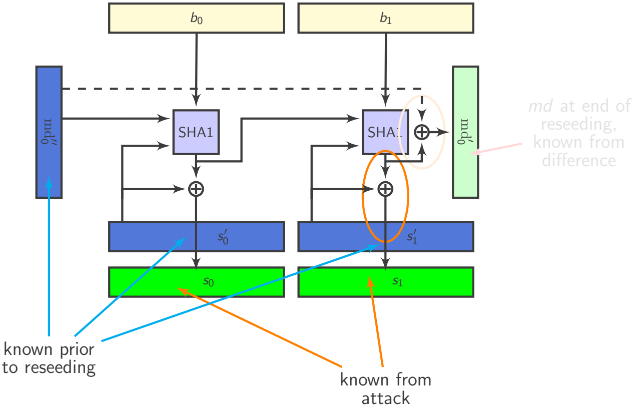
State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



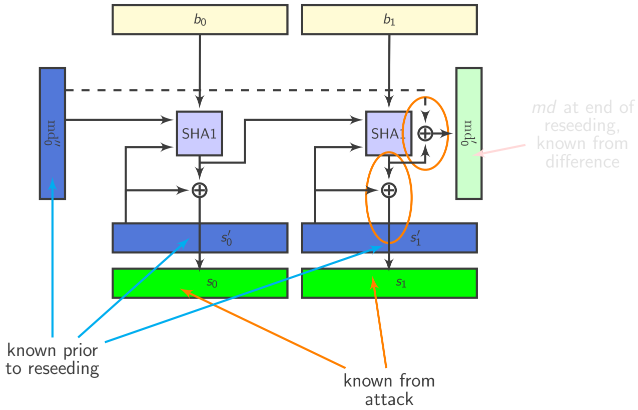
md at end of reseeding, known from difference

known prior to reseeding

known from attack

State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



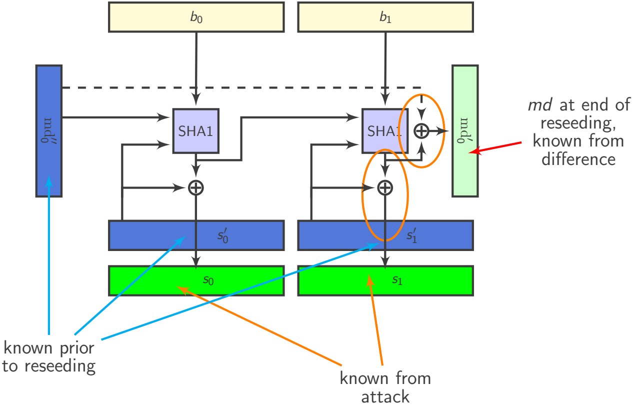
md at end of reseeding, known from difference

known prior to reseeding

known from attack

State Recovery Attack: DEJA-STATE

- state bytes recovered
- now: recovery of *md* after the seeding
- revisit the attacked seeding:



md at end of reseeding, known from difference

known prior to reseeding

known from attack

Dealing with Non-Zero Initial Entropy

Strategies to deal with non-zero initial entropy

- determine state prior to seeding from output
- determine additional entropy during the recovery of md
  - computational effort 2^{80+x}

Dealing with Non-Zero Initial Entropy

Strategies to deal with non-zero initial entropy

- determine state prior to seeding from output
- determine additional entropy during the recovery of md
    - computational effort 2^{80+x}

Dealing with Non-Zero Initial Entropy

Strategies to deal with non-zero initial entropy

- determine state prior to seeding from output
- determine additional entropy during the recovery of md
    - computational effort 2^{80+x}

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered “light-weight security”
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered “light-weight security”
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered “light-weight security”
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered "light-weight security"
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered "light-weight security"
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered "light-weight security"
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered "light-weight security"
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered "light-weight security"
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

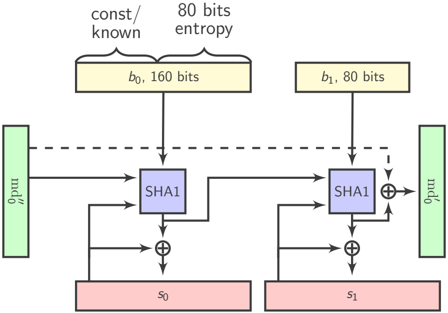
- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered "light-weight security"
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-STATE

- state after reseeding completely recovered
- condition: attacker receives longer portion of output at specific offset after reseeding
- effort for a 320-bit seed: 2^{84} hash evaluations
- (some tens of bytes in each hash invocation)
- also possible for seed not a multiple of 80 bits
- 2^{80} considered "light-weight security"
  - $\approx$ RSA-1024
  - PRESENT light-weight block cipher for RFID applications
  - must be feared to be breakable within a decade (?)
  - will incur considerable costs for a long time

State Recovery Attack: DEJA-SEED

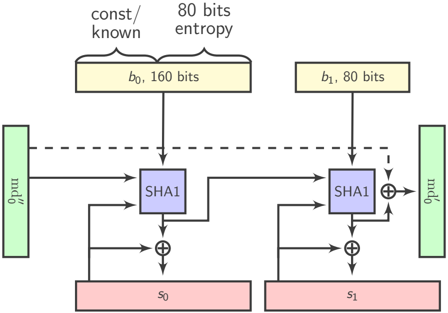
- similar attack, recover also the seed



- synching to *md* like in DEJA-STATE
- then iterate through the possible seed values

State Recovery Attack: DEJA-SEED

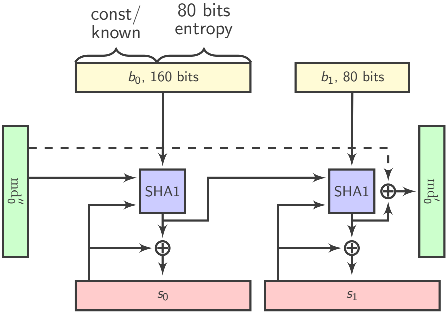
- similar attack, recover also the seed



- synching to *md* like in DEJA-STATE
- then iterate through the possible seed values

State Recovery Attack: DEJA-SEED

- similar attack, recover also the seed



- synching to *md* like in DEJA-STATE
- then iterate through the possible seed values

Forward Security of Seed Data

Forward Security of Seed Data

- forward security of seed data not a recognized notion
- OpenSSL's RNG: even high entropy seed data potentially recoverable
- block-wise hashing in `RAND_add` is a mistake
- correct: hashing state together with new input
- always inefficient for large RNG states

Forward Security of Seed Data

- forward security of seed data not a recognized notion
- OpenSSL's RNG: even high entropy seed data potentially recoverable
- block-wise hashing in `RAND_add` is a mistake
- correct: hashing state together with new input
- always inefficient for large RNG states

Forward Security of Seed Data

- forward security of seed data not a recognized notion
- OpenSSL's RNG: even high entropy seed data potentially recoverable
- block-wise hashing in `RAND_add` is a mistake
- correct: hashing state together with new input
- always inefficient for large RNG states

Forward Security of Seed Data

- forward security of seed data not a recognized notion
- OpenSSL's RNG: even high entropy seed data potentially recoverable
- block-wise hashing in `RAND_add` is a mistake
- correct: hashing state together with new input
- always inefficient for large RNG states

Forward Security of Seed Data

- forward security of seed data not a recognized notion
- OpenSSL's RNG: even high entropy seed data potentially recoverable
- block-wise hashing in `RAND_add` is a mistake
- correct: hashing state together with new input
- always inefficient for large RNG states

Theoretical Considerations

Notion of Mixing Function

$$H(f(I,S)) \geq H(S) \text{ and } H(f(I,S) \geq H(I)$$

- input I, state S
- `RAND_add` fulfills this notion formally
- but not effectively
- only useful if whole state is used in output production in a **symmetric way**
- need definition which considers **entropy of subsequent output** instead of that of the state

Notion of Mixing Function

$$H(f(I,S)) \geq H(S) \text{ and } H(f(I,S) \geq H(I)$$

- input I, state S
- `RAND_add` fulfills this notion formally
- but not effectively
- only useful if whole state is used in output production in a **symmetric way**
- need definition which considers **entropy of subsequent output** instead of that of the state

Notion of Mixing Function

$$H(f(I,S)) \geq H(S) \text{ and } H(f(I,S) \geq H(I)$$

- input I, state S
- `RAND_add` fulfills this notion formally
- but not effectively
- only useful if whole state is used in output production in a **symmetric way**
- need definition which considers **entropy of subsequent output** instead of that of the state

Notion of Mixing Function

$$H(f(I,S)) \geq H(S) \text{ and } H(f(I,S) \geq H(I)$$

- input I, state S
- `RAND_add` fulfills this notion formally
- but not effectively
- only useful if whole state is used in output production in a **symmetric way**
- need definition which considers **entropy of subsequent output** instead of that of the state

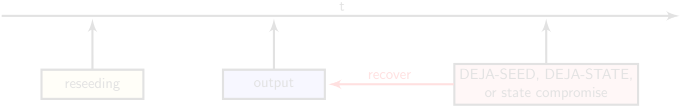
Notion of Mixing Function

$$H(f(I,S)) \geq H(S) \text{ and } H(f(I,S) \geq H(I)$$

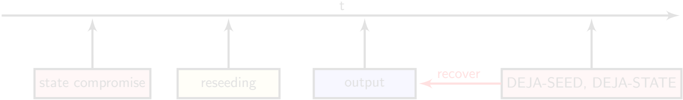
- input I, state S
- `RAND_add` fulfills this notion formally
- but not effectively
- only useful if whole state is used in output production in a **symmetric way**
- need definition which considers **entropy of subsequent output** instead of that of the state

Formal Vulnerabilities of OpenSSL's RNG

- impaired forward security



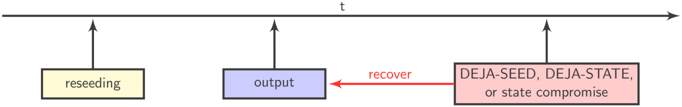
- backward security not attempted by RNG itself
    - but when attempted by application, suffers from our attacks



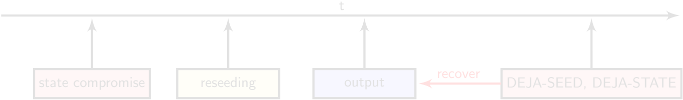
- new notion: forward security of seed data
    - not achieved by OpenSSL's RNG

Formal Vulnerabilities of OpenSSL's RNG

- impaired forward security



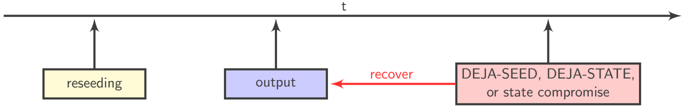
- backward security not attempted by RNG itself
  - but when attempted by application, suffers from our attacks



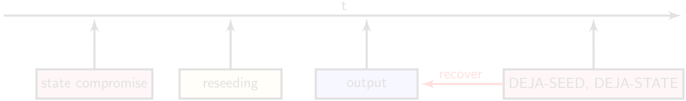
- new notion: forward security of seed data
  - not achieved by OpenSSL's RNG

Formal Vulnerabilities of OpenSSL's RNG

- impaired forward security



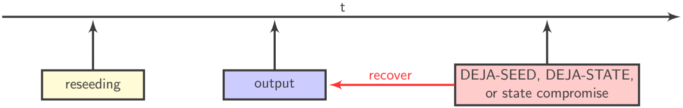
- backward security not attempted by RNG itself
  - but when attempted by application, suffers from our attacks



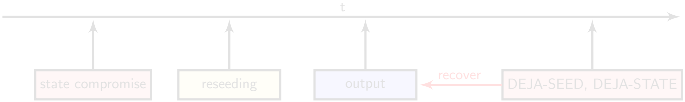
- new notion: forward security of seed data
  - not achieved by OpenSSL's RNG

Formal Vulnerabilities of OpenSSL's RNG

- impaired forward security



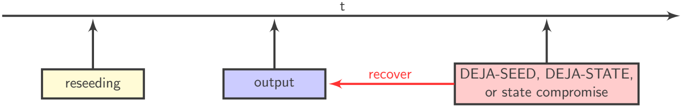
- backward security not attempted by RNG itself
  - but when attempted by application, suffers from our attacks



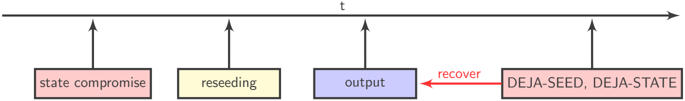
- new notion: forward security of seed data
  - not achieved by OpenSSL's RNG

Formal Vulnerabilities of OpenSSL's RNG

- impaired forward security



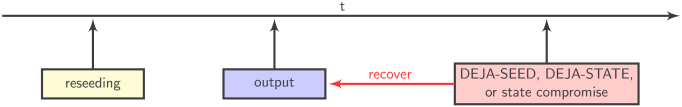
- backward security not attempted by RNG itself
  - but when attempted by application, suffers from our attacks



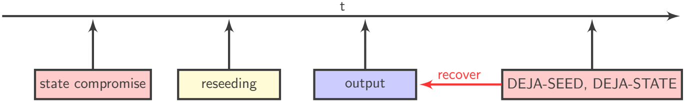
- new notion: forward security of seed data
  - not achieved by OpenSSL's RNG

Formal Vulnerabilities of OpenSSL's RNG

- impaired forward security



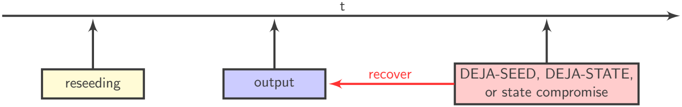
- backward security not attempted by RNG itself
  - but when attempted by application, suffers from our attacks



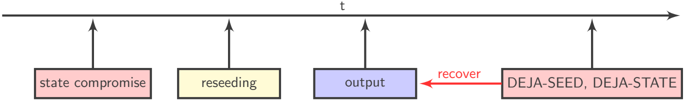
- new notion: forward security of seed data
  - not achieved by OpenSSL's RNG

Formal Vulnerabilities of OpenSSL's RNG

- impaired forward security



- backward security not attempted by RNG itself
    - but when attempted by application, suffers from our attacks



- new notion: forward security of seed data
    - not achieved by OpenSSL's RNG

Repairing OpenSSL's RNG

Repairing the RNG

- RAND_pseudo_bytes must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist

    e.g. AES / counter mode generators
    as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the “entropy flow” beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- RAND_pseudo_bytes must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the “entropy flow” beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the “entropy flow” beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the "entropy flow" beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Repairing the RNG

- `RAND_pseudo_bytes` must use different state (LESLI)
- use cipher-based generator
  - approved and efficients designs exist
    - e.g. AES / counter mode generators
    - as realized in the FIPS version of the library!
  - more efficient than hash-based, due to hardware support
- ad-hoc repair
  - increase the “entropy flow” beyond 160 bits
  - remove the leakage of half of md
  - forward security of seed-data cannot be efficiently addressed
- so far **no** repair in OpenSSL
- secure wrapper functions (→ paper)
- Note: the forks LibreSSL and BoringSSL are even worse

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Conclusion

- multiple design errors in OpenSSL's RNG
  - LESLI
  - ELO240,ELO160,ELO80
  - DEJA-STATE, DEJA-SEED
    - effort around 2^{80} hash evaluations
- impact
  - attacks highly application specific
  - relevant for embedded systems
- theoretic insights
  - applicability of the notion of mixing function
  - forward security of seed data
- repairs suggested

Thank you!