FLORIDA TECH
FLORIDA'S STEM UNIVERSITY™

Department of Computer Science

CSE 4820: Wireless and Mobile Security

11. Bluetooth Security Ct'd

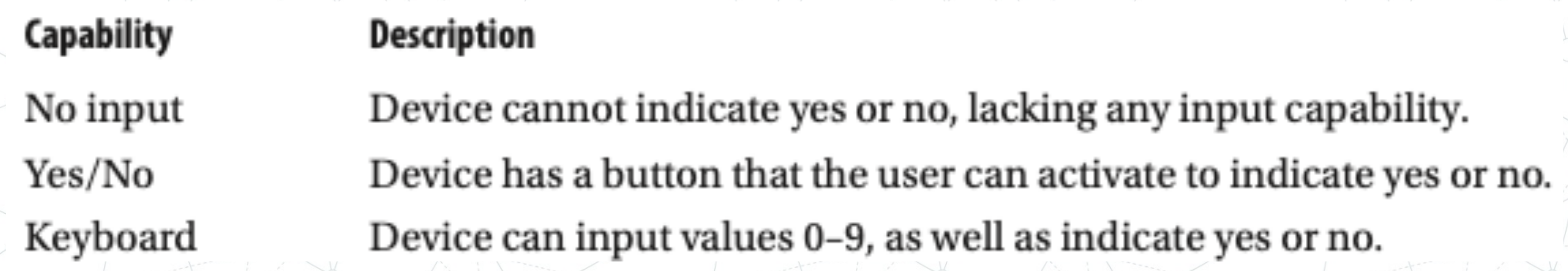
Dr. Abdullah Aydeger
Location: Harris Inst #310
Email: aaydeger@fit.edu

Outline

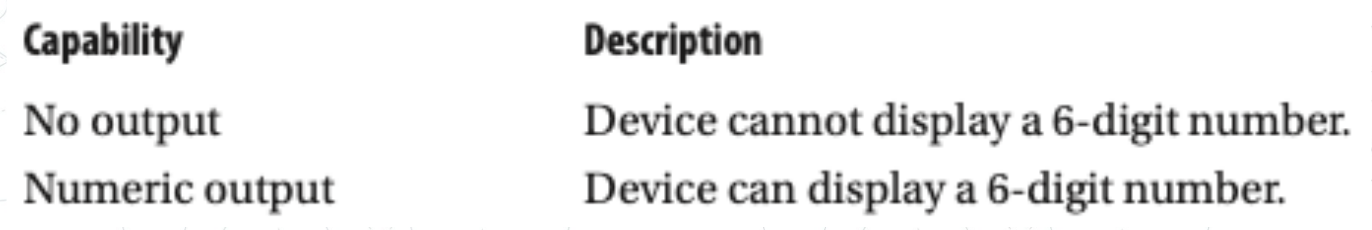
- Bluetooth
  - Attacking Passphrase Authentication
  - SSP Nino Attack

Recall: Secure Simple Pairing: Capabilities Exchange

- Tables summarizing the capability information exchanged by devices:



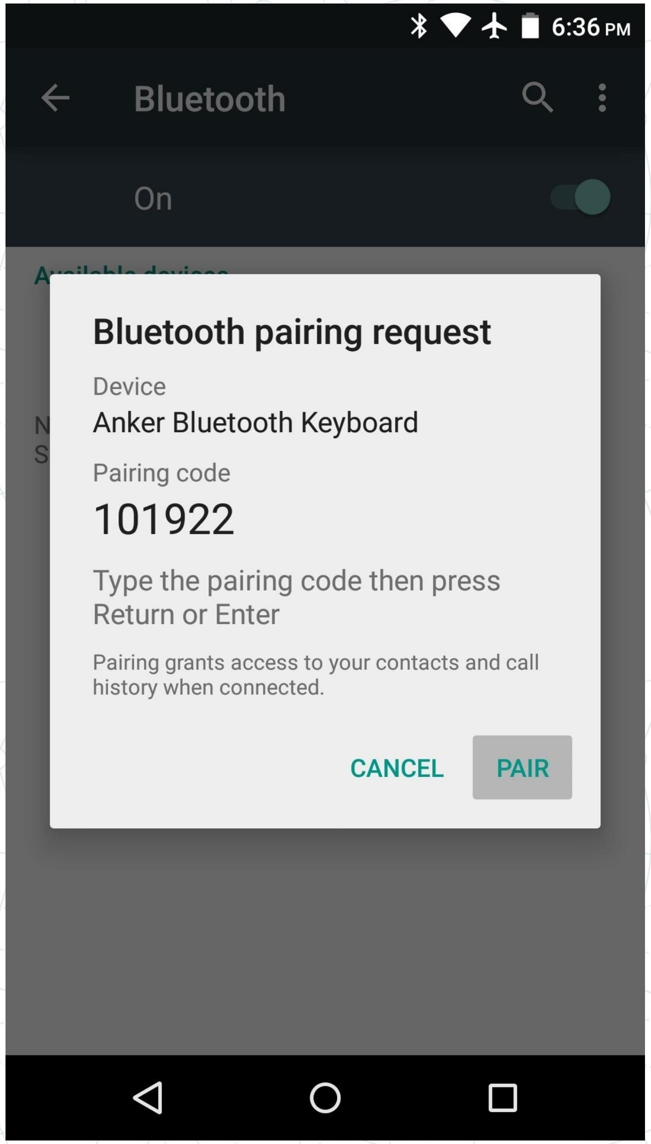
Capability	Description
No input	Device cannot indicate yes or no, lacking any input capability.
Yes/No	Device has a button that the user can activate to indicate yes or no.
Keyboard	Device can input values 0–9, as well as indicate yes or no.



Capability	Description
No output	Device cannot display a 6-digit number.
Numeric output	Device can display a 6-digit number.

Recall: Authentication: Passkey Entry

- The goal of the protocol is to verify the passphrase with the other device, <u>one bit at a time</u>
- For every bit in the passphrase, the devices will compute a hash including the bit and <u>transmit the hash to the other side</u>
  - Both sides will transmit eight hashes



Bluetooth pairing request

Device
Anker Bluetooth Keyboard

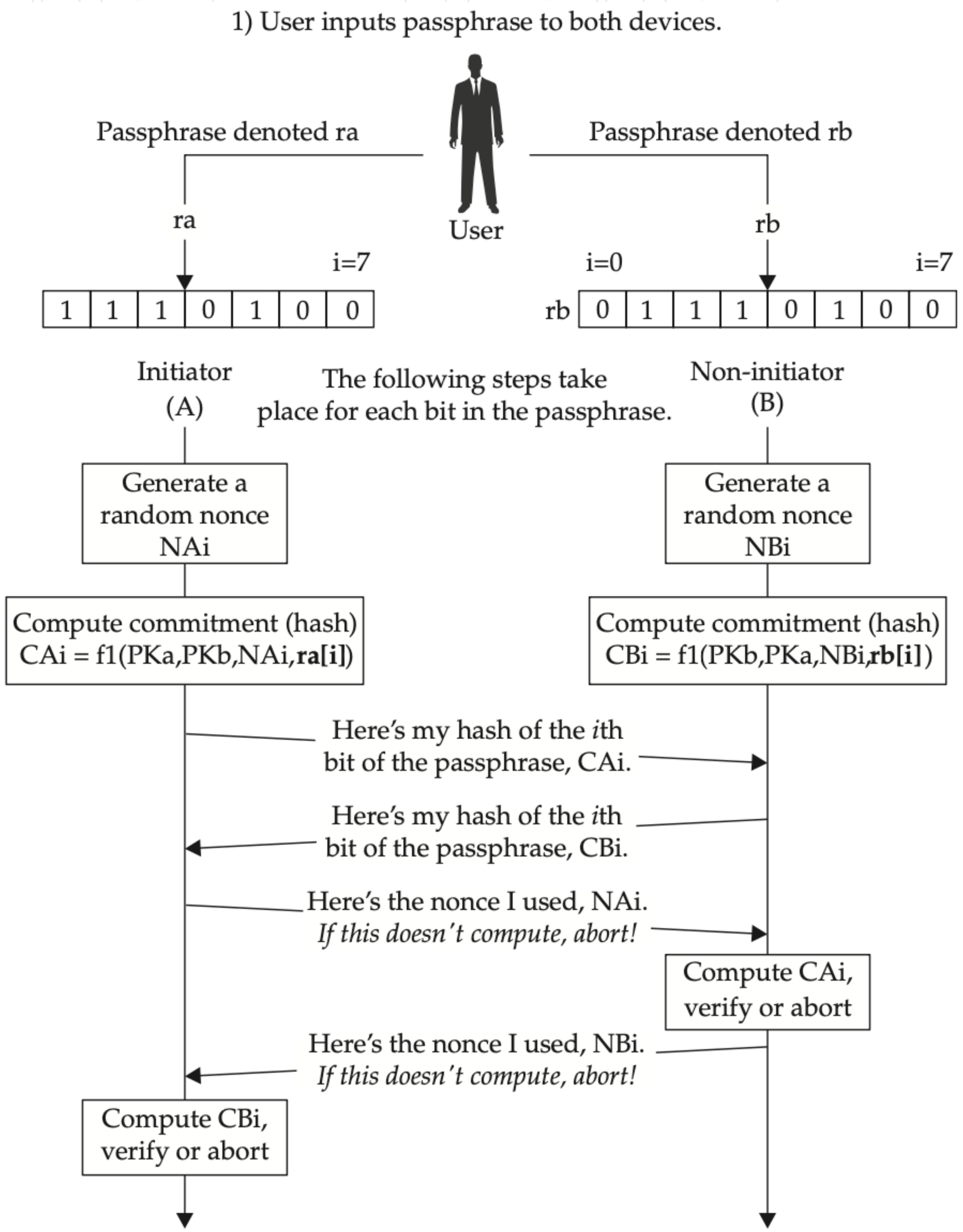
Pairing code
101922

Type the pairing code then press Return or Enter

Pairing grants access to your contacts and call history when connected.

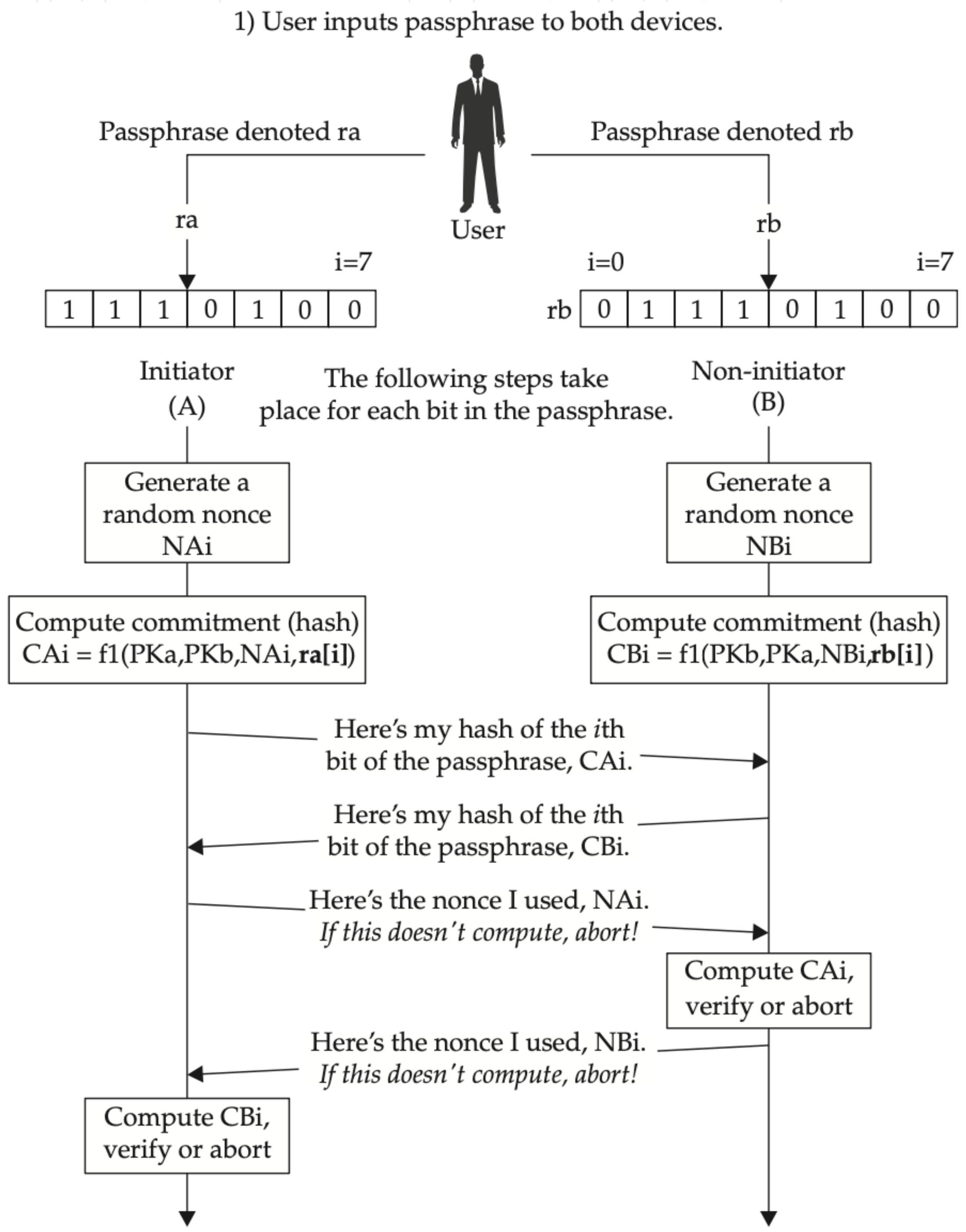
Authentication: Passkey Entry

- The Bluetooth specification calls this protocol a “gradual release” procedure of the passphrase, and it is rationalized in the following manner
  - Device A reveals ra[i] before device B does
  - B, however, has already transmitted his commitment hash, before A reveals the bit
  - This means that B cannot change his choice at this point (in other words, he is committed)


1) User inputs passphrase to both devices.
Passphrase denoted ra
Passphrase denoted rb
User
ra
rb
i=7
i=0
i=7
1 1 1 0 1 0 0
rb 0 1 1 1 0 1 0 0
Initiator (A)
The following steps take place for each bit in the passphrase.
Non-initiator (B)
Generate a random nonce NAi
Generate a random nonce NBi
Compute commitment (hash) CAi = f1(PKa,PKb,NAi,ra[i])
Compute commitment (hash) CBi = f1(PKb,PKa,NBi,rb[i])
Here's my hash of the ith bit of the passphrase, CAi.
Here's my hash of the ith bit of the passphrase, CBi.
Here's the nonce I used, NAi. If this doesn't compute, abort!
Compute CAi, verify or abort
Here's the nonce I used, NBi. If this doesn't compute, abort!
Compute CBi, verify or abort

Authentication: Passkey Entry

- This process prevents a MITM attack because the commitment hash also has the public keys used in the *DHKey* exchange as input
  - A potential MITM attacker will only be able to guess with 50 percent accuracy
  - The odds that an attacker will successfully guess a randomly generated passphrase in such a matter are 1/2i, where i is the length of the passphrase in bits


1) User inputs passphrase to both devices.
Passphrase denoted ra
Passphrase denoted rb
User
ra
rb
i=7
i=0
i=7
1 1 1 0 1 0 0
rb 0 1 1 1 0 1 0 0
Initiator (A)
The following steps take place for each bit in the passphrase.
Non-initiator (B)
Generate a random nonce NAi
Generate a random nonce NBi
Compute commitment (hash) CAi = f1(PKa,PKb,NAi,ra[i])
Compute commitment (hash) CBi = f1(PKb,PKa,NBi,rb[i])
Here's my hash of the ith bit of the passphrase, CAi.
Here's my hash of the ith bit of the passphrase, CBi.
Here's the nonce I used, NAi. If this doesn't compute, abort!
Compute CAi, verify or abort
Here's the nonce I used, NBi. If this doesn't compute, abort!
Compute CBi, verify or abort

Passively Attacking Passphrase Authentication

- An attacker who can observe the entire exchange can trivially recover the passphrase
- For every bit in the passphrase, the attacker will know the following values:
  - *PKa, PKb:* Public keys observed during the initial public key exchange
  - *CAi, Cbi:* The commitment hashes for the ith bit
  - *NAi, Nbi:* The nonces used as input to the above commitment hashes

Passively Attacking Passphrase Authentication

- Given these values, an attacker has what he needs to compute the ith bit himself
  - Remember, there are only two possible values for ra[i]: 1 or 0
- The attacker simply has to compute f1(PKa, PKb, Nai, 0)
  - If the result equals CAi, then ra[i] = 0
  - If that doesn’t pan out, the attacker can compute f1(PKa, PKb, Nai, 1), which will reveal ra[i] is, in fact, 1
- The gist of this is that an attacker can retrieve the passphrase in its entirety, and all he has to do is observe the pairing and compute a few SHA256 hashes

Passively Attacking Passphrase Authentication

- In this regard, SSP is actually worse than the PIN-based scheme used in traditional pairing
  - Under the old system, if the user inputs a 32-bit pin, an attacker will need to compute 232 hashes (worst case) before finding it
  - In the current system, the attacker will need to compute at worst 32 hashes
- Yet, an attacker who learns the passphrase still does not know the link key, because it will be derived from the *DHKey*, which an attacker cannot derive
  - Once the attacker has recovered the passphrase, he can try to convince the devices to pair with him

Actively Attacking Passphrase-Protected Devices

- Another attack can be levied against a passphrase-protected device
  - Assume that a device has a 32-bit passphrase (for simplicity)
  - Also assume that this device can be placed in pairing mode repeatedly
  - Finally, assume that you would like access to this device, and you don't have the link key or passphrase
- All you need to do is randomly choose a bit for ra[i], starting with ra[0] and working your way up
  - If the device continues, then you chose correctly
  - If the device aborts, you know that you chose incorrectly and, therefore, will choose correctly the next time

Actively Attacking Passphrase-Protected Devices

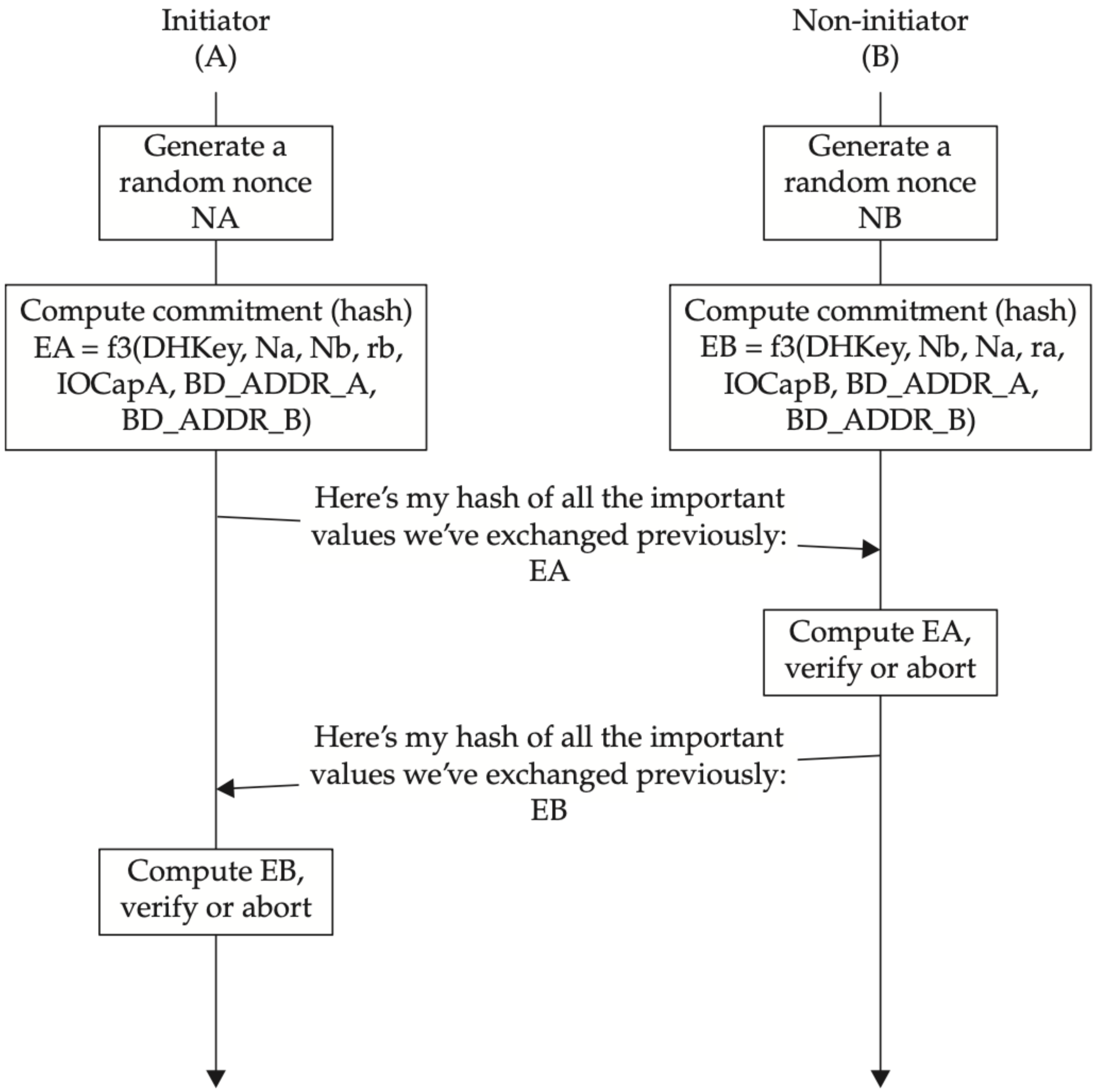
- Assuming the device has a 32-bit passphrase, you will be able to guess the entire passphrase correctly after 16 pairing attempts (on average)
- The problem is that the protocol reveals a bit of the passphrase to the attacker at every step, regardless of her choice
- The specification says that exponential back-off times should be used to slow down attacks such as this
  - But when such a small number of attempts are needed, it seems unlikely to help

Countermeasure

- The moral of this story is that devices using passphrase authentication should <u>never</u> be configured to <u>reuse a static passphrase</u>
- If the passphrase is <u>randomly</u> generated on every attempt, you <u>will not be able to carry the partial information across attempts</u>

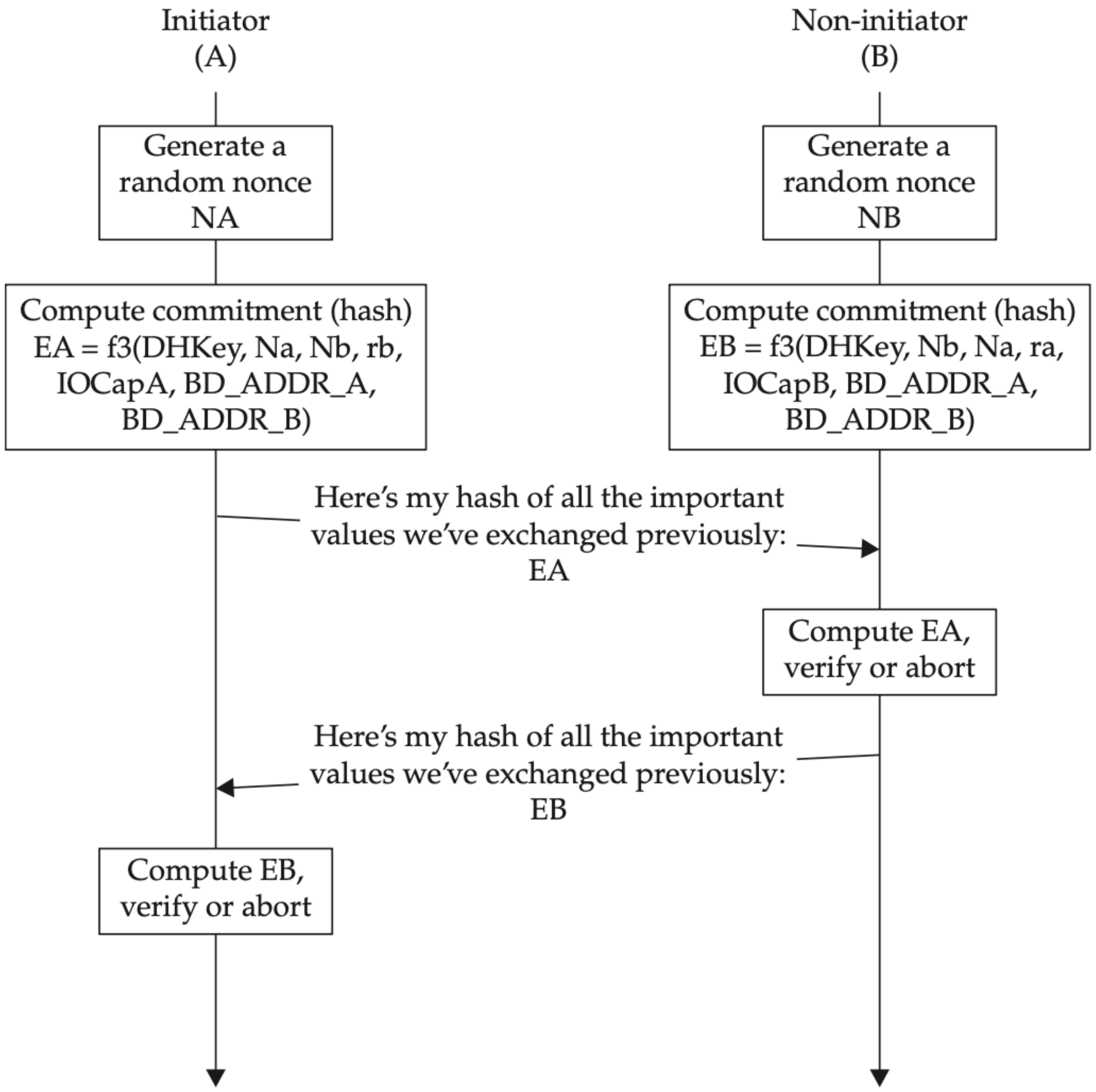
Authentication Phase 2

- Regardless of the authentication technique used, once authentication has completed, the following exchange takes place:
  - Device A computes a hash over all of the important values used previously (the DHKey, IOCapabilities, BD_ADDRs, etc.) and transmits the hash to device B


Initiator (A)
Non-initiator (B)
Generate a random nonce NA
Generate a random nonce NB
Compute commitment (hash) EA = f3(DHKey, Na, Nb, rb, IOCapA, BD_ADDR_A, BD_ADDR_B)
Compute commitment (hash) EB = f3(DHKey, Nb, Na, ra, IOCapB, BD_ADDR_A, BD_ADDR_B)
Here's my hash of all the important values we've exchanged previously: EA
Compute EA, verify or abort
Here's my hash of all the important values we've exchanged previously: EB
Compute EB, verify or abort

Authentication Phase 2

- This is device B's <u>last chance</u> to verify that they have agreed on everything so far
- Device B should <u>verify</u> this hash, and if it checks out, send a similar hash of his own
- At this point, the <u>authentication</u> phase is <u>complete</u>



Initiator (A)
Non-initiator (B)
Generate a random nonce NA
Generate a random nonce NB
Compute commitment (hash) EA = f3(DHKey, Na, Nb, rb, IOCapA, BD_ADDR_A, BD_ADDR_B)
Compute commitment (hash) EB = f3(DHKey, Nb, Na, ra, IOCapB, BD_ADDR_A, BD_ADDR_B)
Here's my hash of all the important values we've exchanged previously: EA
Compute EA, verify or abort
Here's my hash of all the important values we've exchanged previously: EB
Compute EB, verify or abort

Link-Key Derivation

- Once the authentication phase is complete, the devices are convinced that the DHKey was negotiated with the desired party
    - Now it's time to use it for creating the link key
- Link-key derivation is simple at this point because all of the authentication is out of the way
- The link key is derived from the DHKey using the following hash:
    - *LinkKeyAB=f2(DHKey, Nmaster, Nslave, "btlk", BDADDR_master, BD_ADDR_slave)*

Link-Key Derivation

- Once the devices compute the link key, they will store it along with the peer's BD_ADDR
- Then they can use it for future authentication sessions, without having to re-create it
- At this point, SSP becomes identical to traditional pairing
- Encryption keys will be derived using the same hashing values

Link-Key Derivation

- The same encryption system is used to protect the link key
- Note that an attacker who observes any of these exchanges will not be able to compute the link key because it was derived from a Diffie-Hellman key exchange

SSP Summary

- Despite attacks on passphrase authentication just outlined, on average Bluetooth security is enhanced via the use of SSP
- Unless there is a serious cryptographic breakthrough, a passive attacker should not be able to recover the link key since this would require a passive attack against the Diffie-Hellman key exchange
- The best passive attack known to date involves the misuse of the passphrase authentication scheme with a static key
  - An attacker who observes this can trivially compute the passphrase used for the pairing session

SSP Summary

- This is a serious flaw, because the attacker can potentially pair with the compromised devices herself, but she still cannot impersonate one device to the other, or decrypt intercepted traffic
- In order for an attacker to recover the link key between two devices, she must actively attack them during the pairing process
- A successful attack will result in the compromised devices pairing with the attacker, instead of themselves
  - The attacker can then read/write data to both devices

SSP Niño (NoInputNoOutput) Attack

- When two devices pair, they negotiate the IOCapability information to identify a suitable authentication mechanism that is supported by both devices
  - This exchange happens before the link key is derived and, as such, is not a protected exchange
- This attack leverages this SSP IOCapabilities exchange deficiency to manipulate one or more devices, forcing the victim device to "dumb-down" its selected authentication mechanism to the Just Works technique
  - Once one or more devices are forced to this weaker authentication method, the attacker can eavesdrop on any data sent between devices

SSP Niño Attack

- First, the attacker must force two devices to re-pair
  - One option is to launch a denial-of- service (DoS) attack against the Bluetooth devices in the area, designing a transmitter that hops along with the piconet master and jamming on each channel during the frequency- hopping exchange
  - A second option is to leverage a wide-band jammer that can jam all
- 79 Bluetooth channels simultaneously, ceasing all Bluetooth communication within range of the attacker
  - Once the end-user becomes frustrated with the lack of communication between two Bluetooth devices, the attacker would hope that the user attributes the failure to the devices themselves and attempts to repair the devices to resolve the issue, at which time the attacker would stop the DoS attack

SSP Niño Attack

- Immediately before the devices re-pair, the attacker would impersonate the BD_ ADDR and friendly name of both devices and implement a MITM attack, brokering the authentication exchange between the victim devices
- Instead of allowing the legitimate advertised IOCapabilities to pass between devices, however, the attacker would indicate to the responder that the initiator only supports the NoInputNoOutput capability, and vice versa, effectively dumbing-down the connection exchange and leaving the Just Works authentication method as the only plausible method

SSP Niño Attack

- Because the Just Works method does not provide protection against MITM attacks, the attacker can complete the authentication process with each device to conclude the pairing exchange and finish the connection establishment
- With knowledge of the link key derived by both devices, the attacker can now eavesdrop on the traffic between the devices, optionally manipulating the content as desired

Nino: Countermeasure

- The Niño attack leverages a design concession in the SSP specification to accommodate devices with no man-machine interface
- Due to the lack of cryptographic integrity protecting the IOCapabilities exchange, an attacker could leverage this weakness to manipulate one or more victim devices into creating a connection with the attacker
- There are no available exploit tools to implement the Niño attack
  - With integrated support in the BlueZ stack for SSP, however, it would be possible to impersonate both devices in anticipation of a pairing exchange and force the use of the Just Works authentication mechanism
- Additional research and experimentation is needed to evaluate the practicality of this attack against real-world implementations

Nino: Countermeasure

- One significant requirement for the Niño attack is to force a Bluetooth user to delete the prior pairing information and re-pair devices, or to actively catch a user pairing two devices for the first time
- You can leverage this attack requirement as a countermeasure, by not pairing Bluetooth devices in any location that is susceptible to traffic sniffing attacks
    - If one or more of your Bluetooth devices prompts you to spuriously repair, disable Bluetooth on both devices temporarily until you can return to a safe location and re-pair

FLORIDA TECH
FLORIDA'S **STEM** UNIVERSITY™

Thank you. Questions?

Dr. Abdullah Aydeger