

6.1800 Spring 2025

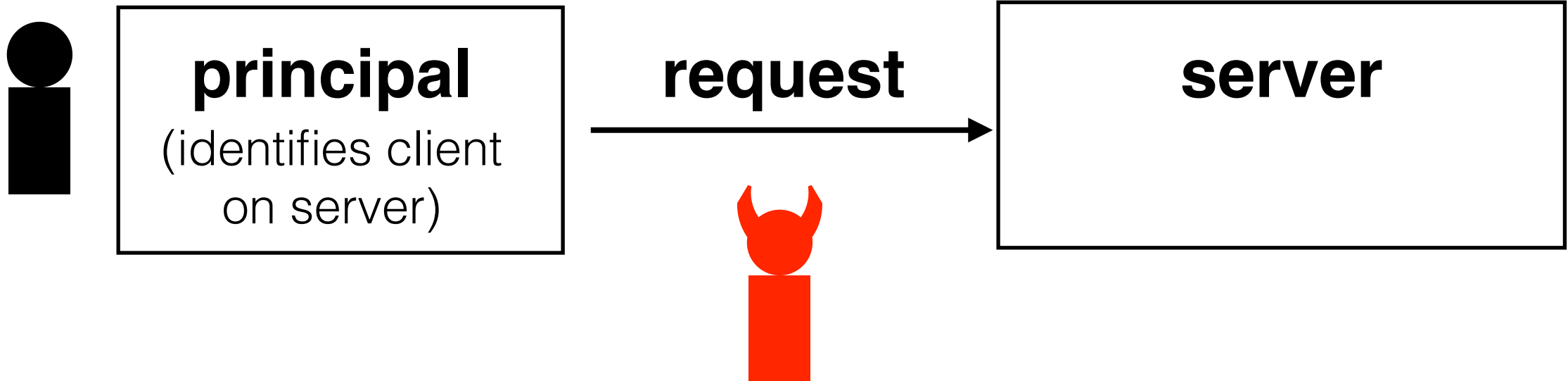
Lecture #23: Secure Channels

confidentiality and integrity through the magic of cryptography

**so far, we've dealt with
adversaries that were trying
to access data on a server**



this week, we're going to turn
to adversaries that are
observing data on the **network**



**some network traffic is
difficult to interpret**

e.g., IP addresses are private or resolve to
Akamai or Amazon servers

```
14:05:31.983557 34392425us tsft -62dB signal -98dB noise antenna 1 5785 MHz 11a
ht/20 [bit 20] CF +QoS IP 184.28.89.95.443 > 10.189.86.146.41204: Flags [P.], seq
1643649202:1643649233, ack 1215791031, win 285, options [nop,nop,TS val 2235675295
ecr 95087166], length 31
0x0000:  aaaa 0300 0000 0800 4548 0053 b11e 4000  ....EH.S..@.
0x0010:  3506 2174 b81c 595f 0abd 5692 01bb a0f4  5.!t..Y_..V....
0x0020:  61f8 18b2 4877 7fb7 8018 011d 835f 0000  a...Hw....._..
0x0030:  0101 080a 8541 b29f 05aa ea3e 1503 0300  ....A.....>....
0x0040:  1ac6 d28d 46ab 64f6 36a3 4efb edd1 f693  ....F.d.6.N....
0x0050:  5cf0 0132 65f2 0b0d 21dd 66                \..2e...!.f
```

```
[katrina ~] dig -x 184.28.89.95

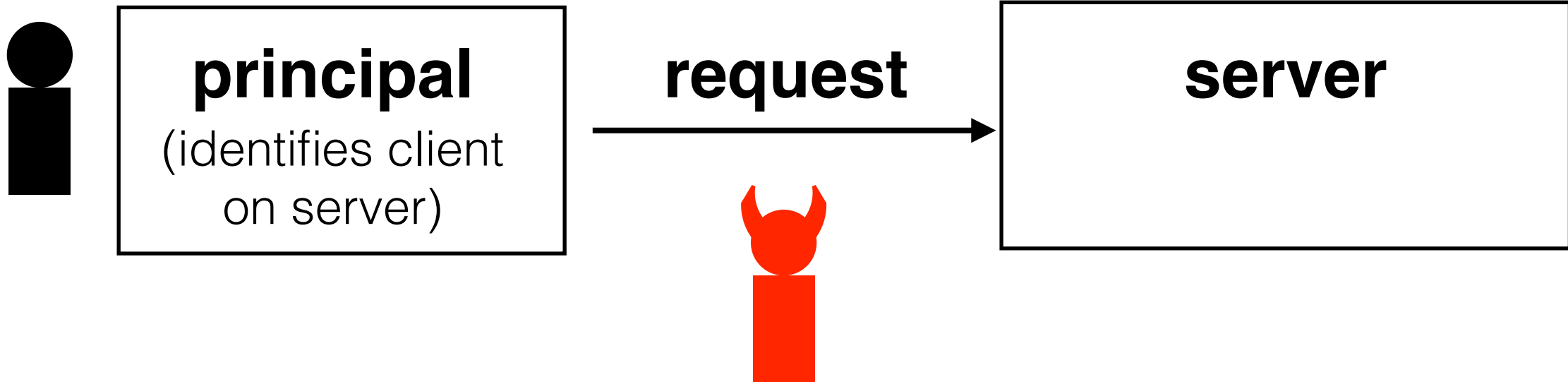
; <<>> DiG 9.8.3-P1 <<>> -x 184.28.89.95
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47850
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 8, ADDITIONAL: 8

;; QUESTION SECTION:
;95.89.28.184.in-addr.arpa.      IN      PTR

;; ANSWER SECTION:
95.89.28.184.in-addr.arpa. 43125 IN      PTR      a184-28-89-95.deploy.static.akamaitechnologies.com.
```

this week, we're going to turn
to adversaries that are
observing data on the network

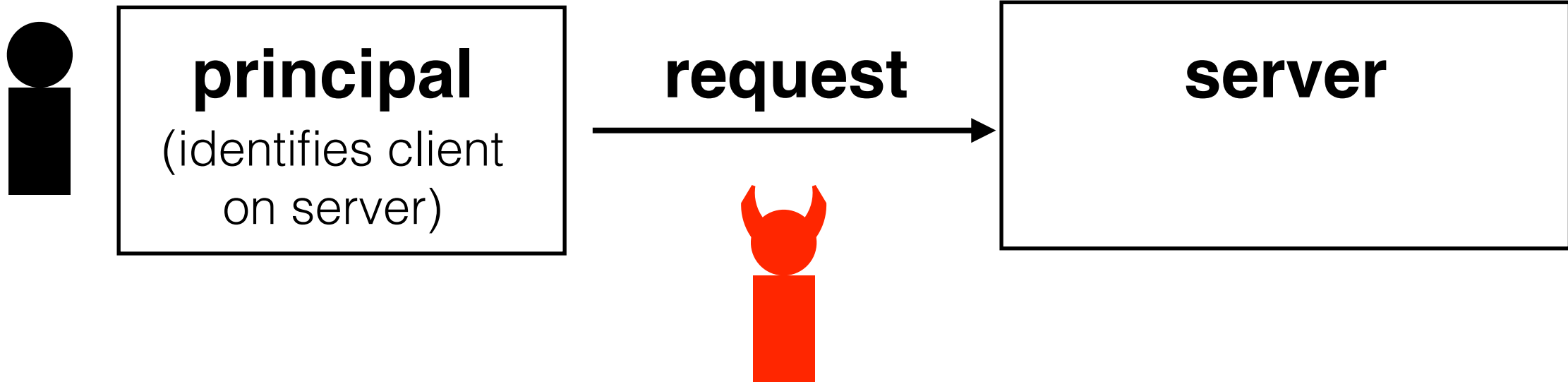
some packet data can reveal
what you're doing even
if the packet headers are difficult
to interpret



0x0130:	5f47	414d	455f	4556	454e	5425	3236	6a73	_GAME_EVENT%26js
0x0140:	6f6e	5f76	616c	2533	4425	3742	2532	3261	on_val%3D%7B%22a
0x0150:	7070	496e	666f	2532	3225	3341	2537	4225	ppInfo%22%3A%7B%
0x0160:	3232	6170	7069	6425	3232	2533	4125	3232	22appid%22%3A%22
0x0170:	636f	6d2e	7469	6e79	636f	7270	2e70	6f74	com.tinycorp.pot
0x0180:	7465	7225	3232	2532	4325	3232	636f	7265	ter%22%2C%22core
0x0190:	7325	3232	2533	4132	2532	4325	3232	6465	s%22%3A%22%2C%22de
0x01a0:	7669	6365	5f69	6425	3232	2533	4125	3232	vice_id%22%3A%22
0x01b0:	4533	3346	3230	3642	2d33	3336	302d	3444	E33F206B-3360-4D
0x01c0:	3736	2d42	4236	422d	3742	4144	3043	4130	76-BB6B-7BAD0CA0
0x01d0:	3746	4541	2532	3225	3243	2532	3264	6576	7FEA%22%2C%22dev
0x01e0:	6963	655f	6d6f	6465	6c25	3232	2533	4125	ice_model%22%3A%
0x01f0:	3232	6950	686f	6e65	3925	3243	3225	3232	22iPhone9%2C%22
0x0200:	2532	4325	3232	6875	6d61	6e5f	6964	2532	%2C%22human_id%2
0x0210:	3225	3341	2532	3225	3232	2532	4325	3232	2%3A%22%22%2C%22
0x0220:	6964	6661	2532	3225	3341	2532	3231	4237	idfa%22%3A%221B7
0x0230:	3646	4643	362d	4130	3432	2d34	4530	312d	6FFC6-A042-4E01-
0x0240:	4239	3934	2d42	4245	3135	3443	3738	4645	B994-BBE154C78FE
0x0250:	3625	3232	2532	4325	3232	696e	7374	616c	6%22%2C%22instal
0x0260:	6c5f	6964	2532	3225	3341	2d36	3135	3437	l_id%22%3A-61547
0x0270:	3635	3033	2532	4325	3232	6c61	6e67	7561	6503%2C%22langua
0x0280:	6765	2532	3225	3341	2532	3265	6e2d	5553	ge%22%3A%22en-US
0x0290:	2532	3225	3243	2532	326c	6f63	616c	6525	%22%2C%22locale%
0x02a0:	3232	2533	4125	3232	656e	5f55	5325	3232	22%3A%22en_US%22
0x02b0:	2532	4325	3232	6f73	5f74	7970	6525	3232	%2C%22os_type%22
0x02c0:	2533	4125	3232	6950	686f	6e65	2b4f	5325	%3A%22iPhone+OS%
0x02d0:	3232	2532	4325	3232	6f73	5f76	6572	7369	22%2C%22os_versi

this week, we're going to turn
to adversaries that are
observing data on the network

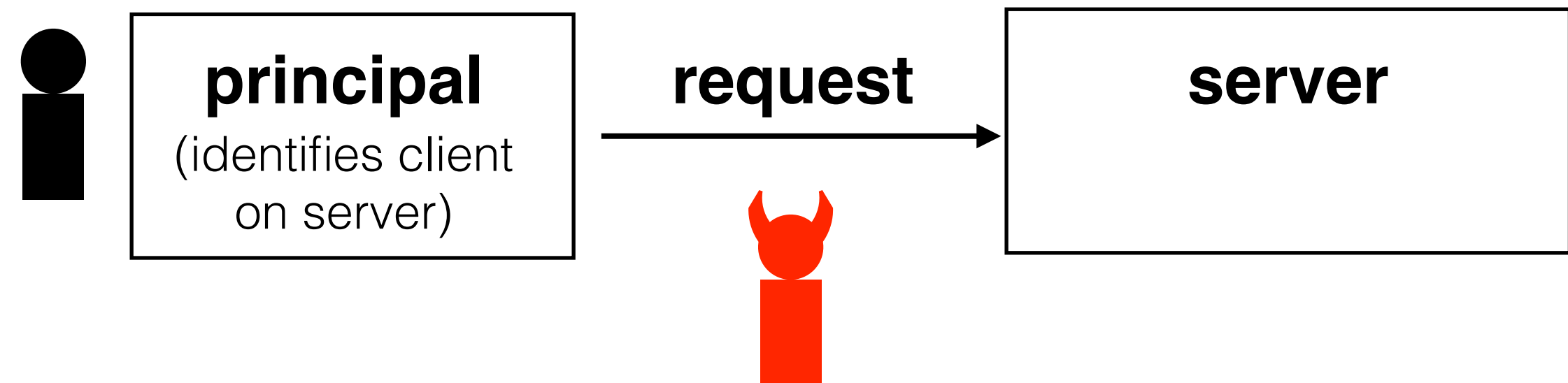
some packet data can reveal
what you're doing even
if the packet headers are difficult
to interpret



0x0400:	2532	3225	3243	2532	3261	7474	5f63	6f75	%22%2C%22att_cou
0x0410:	7261	6765	2532	3225	3341	3131	2532	4325	rage%22%3A11%2C%
0x0420:	3232	6174	745f	656d	7061	7468	7925	3232	22att_empathy%22
0x0430:	2533	4131	3125	3243	2532	3261	7474	5f6b	%3A11%2C%22att_k
0x0440:	6e6f	776c	6564	6765	2532	3225	3341	3132	nowledge%22%3A12
0x0450:	2532	4325	3232	6176	6174	6172	5f68	6f75	%2C%22avatar_hou
0x0460:	7365	2532	3225	3341	2532	3273	6c79	7425	se%22%3A%22slyt%
0x0470:	3232	2532	4325	3232	6176	6174	6172	5f79	22%2C%22avatar_y
0x0480:	6561	7225	3232	2533	4132	2532	4325	3232	ear%22%3A2%2C%22
0x0490:	6563	686f	2532	3225	3341	2537	4225	3232	echo%22%3A%7B%22
0x04a0:	6625	3232	2533	4125	3232	636f	6d2e	7469	f%22%3A%22com.ti
0x04b0:	6e79	636f	7270	2e70	6f74	7465	7225	3232	nycorp.potter%22
0x04c0:	2532	4325	3232	7025	3232	2533	4166	616c	%2C%22p%22%3Afal
0x04d0:	7365	2532	4325	3232	7225	3232	2533	4174	se%2C%22r%22%3At
0x04e0:	7275	6525	3744	2532	4325	3232	656e	6572	rue%7D%2C%22ener
0x04f0:	6779	5f62	616c	616e	6365	2532	3225	3341	gy_balance%22%3A
0x0500:	3025	3243	2532	3265	7665	6e74	5f74	7970	0%2C%22event_typ
0x0510:	6525	3232	2533	4125	3232	6261	636b	6772	e%22%3A%22backgr
0x0520:	6f75	6e64	5365	7373	696f	6e25	3232	2532	oundSession%22%2
0x0530:	4325	3232	6576	656e	745f	756e	6978	5f74	C%22event_unix_t
0x0540:	6d25	3232	2533	4131	3535	3635	3631	3131	m%22%3A155656111
0x0550:	3225	3243	2532	3267	7569	6425	3232	2533	2%2C%22guid%22%3
0x0560:	4125	3232	3263	6433	6433	3336	2d35	3463	A%222cd3d336-54c
0x0570:	642d	3433	6538	2d39	3539	332d	3961	6537	d-43e8-9593-9ae7
0x0580:	3563	6430	3433	3938	2532	3225	3243	2532	5cd04398%22%2C%2
0x0590:	3268	635f	6261	6c61	6e63	6525	3232	2533	2hc_balance%22%3
0x05a0:	4131	3131	2532	4325	3232	6875	6d61	6e5f	A111%2C%22human_

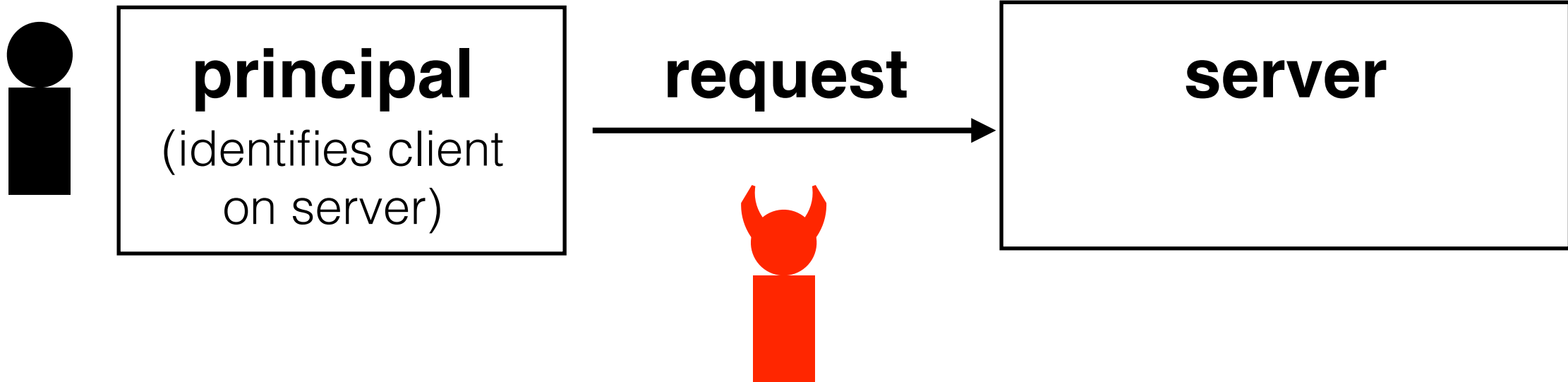
this week, we're going to turn to adversaries that are observing data on the network

some packet data can reveal what you're doing even if the packet headers are difficult to interpret



```
14:10:28.658392 331061605us tsft -98dB noise antenna 1 5785 MHz 11a ht/20 [bit 20]
+QoS IP 18.4.86.46.80 > 18.21.134.133.59071: Flags [..], seq 9009:10457, ack 1, win
options [nop,nop,TS val 1469784939 ecr 1030694527], length 1448: HTTP
0x0040: 0d0a 0a09 0909 3c6f 7074 696f 6e20 7661 .....<option.va
0x0050: 6c75 653d 2234 3439 223e 266e 6273 703b lue="449">&nbsp;
0x0060: 2026 6e62 7370 3b54 6f77 6e20 5371 7561 .&nbsp;Town.Squa
0x0070: 7265 3c2f 6f70 7469 6f6e 3e0a 0909 0a09 re</option>.....
0x0080: 0909 3c6f 7074 696f 6e20 7661 6c75 653d ..<option.value=
0x0090: 2234 3430 223e 4426 616d 703b 4420 4d79 "440">D&D.My
0x00a0: 7374 6572 7920 4d61 6669 613c 2f6f 7074 stery.Mafia</opt
0x00b0: 696f 6e3e 0a09 090a 0909 093c 6f70 7469 ion>.....<opti
0x00c0: 6f6e 2076 616c 7565 3d22 3434 3122 3e26 on.value="441">&
0x00d0: 6e62 7370 3b20 266e 6273 703b 4d6f 6e73 nbsp;.&nbsp;Mons
0x00e0: 7465 7220 4d61 6e75 616c 3c2f 6f70 7469 ter.Manual</opti
0x00f0: 6f6e 3e0a 0909 0a09 0909 3c6f 7074 696f on>.....<optio
0x0100: 6e20 7661 6c75 653d 2234 3432 223e 266e n.value="442">&n
0x0110: 6273 703b 2026 6e62 7370 3b50 6c61 7965 bsp;.&nbsp;Playe
0x0120: 7227 7320 4861 6e64 626f 6f6b 3c2f 6f70 r's.Handbook</op
0x0130: 7469 6f6e 3e0a 0909 0a09 0909 3c6f 7074 tion>.....<opt
0x0140: 696f 6e20 7661 6c75 653d 2234 3433 223e ion.value="443">
0x0150: 266e 6273 703b 2026 6e62 7370 3b44 756e &nbsp;.&nbsp;Dun
0x0160: 6765 6f6e 204d 6173 7465 7227 7320 4775 geon.Master's.Gu
0x0170: 6964 653c 2f6f 7074 696f 6e3e 0a09 090a ide</option>....
```

this week, we're going to turn
to adversaries that are
observing data on the network



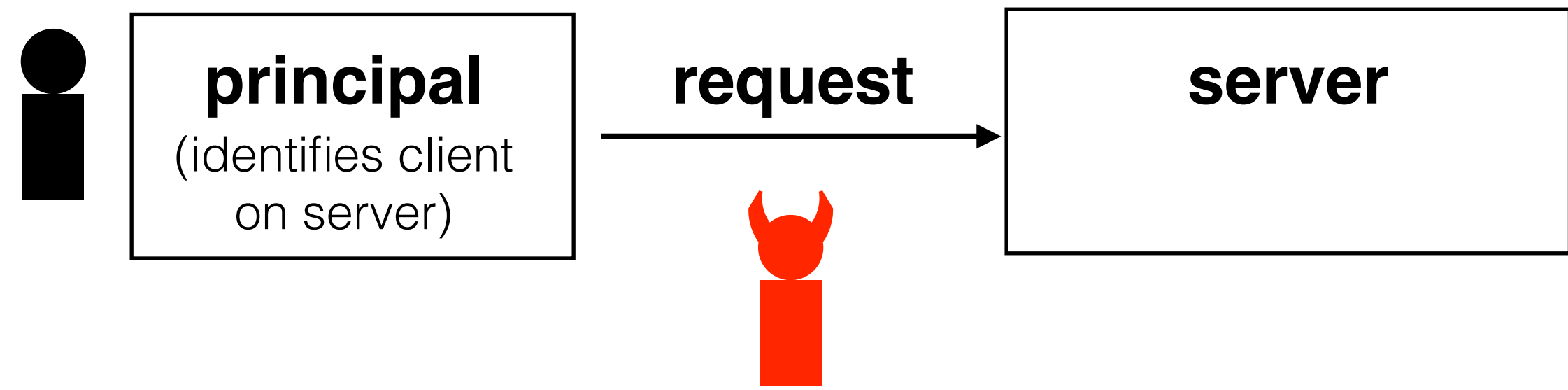
sometimes traffic can be easily tied
to individuals

either in packet headers or packet data

```
14:05:29.947459 104653458us tsft -70dB signal -92dB noise antenna 0 2412 MHz 11g
ht/20 39.0 Mb/s MCS 10 20 MHz lon GI mixed BCC FEC [bit 20] CF +QoS IP
10.189.6.135.5353 > 224.0.0.251.5353: 0*- [0q] 2/0/3 (Cache flush) PTR Bobs-
iPhone.local., (Cache flush) PTR Bobs-iPhone.local. (217)
```

0x0000:	aaaa	0300	0000	0800	4500	00f5	2053	0000	E....S..
0x0010:	ff11	a865	0abd	0687	e000	00fb	14e9	14e9	...e.....
0x0020:	00e1	5867	0000	8400	0000	0002	0000	0003	..Xg.....
0x0030:	0137	0135	0144	0133	0139	0130	0138	0133	.7.5.D.3.9.0.8.3
0x0040:	0135	0135	0139	0144	0144	0141	0143	0130	.5.5.9.D.D.A.C.0
0x0050:	0130	0130	0130	0130	0130	0130	0130	0130	.0.0.0.0.0.0.0
0x0060:	0130	0130	0130	0130	0130	0138	0145	0146	.0.0.0.0.0.8.E.F
0x0070:	0369	7036	0461	7270	6100	000c	8001	0000	.ip6.arpa.....
0x0080:	0078	0015	0d44	3139	8b64	432d	6950	686f	.x..... Bobs-iPho
0x0090:	6e65	056c	6f63	616c	0003	3133	3501	3603	ne.local. ..135.6.
0x00a0:	3138	3902	3130	0769	6e2d	6164	6472	c050	189.10.in-addr.P
0x00b0:	000c	8001	0000	0078	0002	c060	c00c	002f	x...`.../
0x00c0:	8001	0000	0078	0006	c00c	0002	0008	c075	x.....u
0x00d0:	002f	8001	0000	0078	0006	c075	0002	0008	./.....x...u....
0x00e0:	0000	2905	a000	0011	9400	1200	0400	0e00	..).....
0x00f0:	256e	8dc1	7d01	b16c	8dc1	7d01	b1		%n..}.1..}..

this week, we're going to turn to adversaries that are observing data on the network



sometimes traffic can be easily tied to individuals

either in packet headers or packet data

today we're going to focus on how to protect packet data from an adversary

next time, we'll talk about how you can protect meta-information (e.g., packet headers) from an adversary

0x0000:	aaaa	0300	0000	0800	4500	009b	2acb	0000	E...*...
0x0010:	ff11	d8b2	1215	c4c3	e000	00fb	14e9	14e9	
0x0020:	0087	a623	0000	0000	0002	0000	0000	0001	...#.....
0x0030:	184d	6174	74e2	8099	7320	4d61	6342	6f6f	.XXXX...s.MacBoo
0x0040:	6b20	4169	7220	2833	290f	5f63	6f6d	7061	k.Air.(3)._compa
0x0050:	6e69	6f6e	2d6c	696e	6b04	5f74	6370	056c	nion-link._tcp.l
0x0060:	6f63	616c	0000	1000	0116	5468	6f6d	6173	ocal.....XXXXXX
0x0070:	e280	9973	204d	6163	426f	6f6b	2041	6972	...s.MacBook.Air
0x0080:	c025	0010	8001	0000	2905	a000	0011	9400	.%.....).....
0x0090:	1200	0400	0e00	81a6	4167	2f68	dc84	4167	Ag/h..Ag
0x00a0:	2f68	dc							/h.
0x0000:	aaaa	0300	0000	0800	4500	00e2	338a	0000	E...3...
0x0010:	ff11	cfac	1215	c4c3	e000	00fb	14e9	14e9	
0x0020:	00ce	5a25	0000	0000	0005	0000	0000	0001	..Z%.....
0x0030:	114d	6f68	616e	e280	9973	204d	6163	2050	.XXXXX...s.Mac.P
0x0040:	726f	0f5f	636f	6d70	616e	696f	6e2d	6c69	ro._companion-li
0x0050:	6e6b	045f	7463	7005	6c6f	6361	6c00	0010	nk._tcp.local...
0x0060:	0001	154d	6f68	616e	e280	9973	204d	6163	...XXXXX...s.Mac
0x0070:	2050	726f	2028	3229	c01e	0010	0001	1566	.Pro.(2).....X
0x0080:	6572	6761	736f	6ee2	8099	7320	4375	7465	XXXXXXXX...s..Mac
0x0090:	426f	6f6b	c01e	0010	0001	184d	6174	74e2	Book.....XXXX.
0x00a0:	8099	7320	4d61	6342	6f6f	6b20	4169	7220	..s.MacBook.Air.
0x00b0:	2833	29c0	1e00	1000	010d	4d61	7961	e280	(3).....XXXX..
0x00c0:	9973	2069	5061	64c0	1e00	1000	0100	0029	.s.iPad.....)
0x00d0:	05a0	0000	1194	0012	0004	000e	0081	a641	A
0x00e0:	672f	68dc	8441	672f	68dc				g/h..Ag/h.
0x0000:	aaaa	0300	0000	0800	4500	007a	3ea9	0000	E...z>...

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

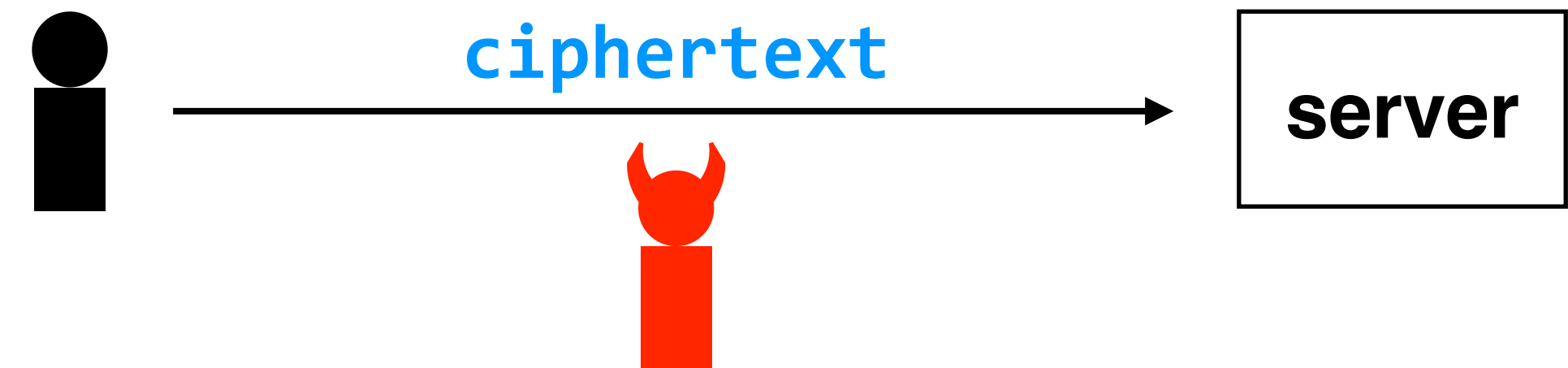
`encrypt(key, message) → ciphertext`

`decrypt(key, ciphertext) → message`

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679  
26cd393d4b93c58f78c
```

```
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7  
8c") = hello, world
```

property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**



adversary can't determine **message**, *but* might be able to cleverly alter **ciphertext** so that it decrypts to a different message

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

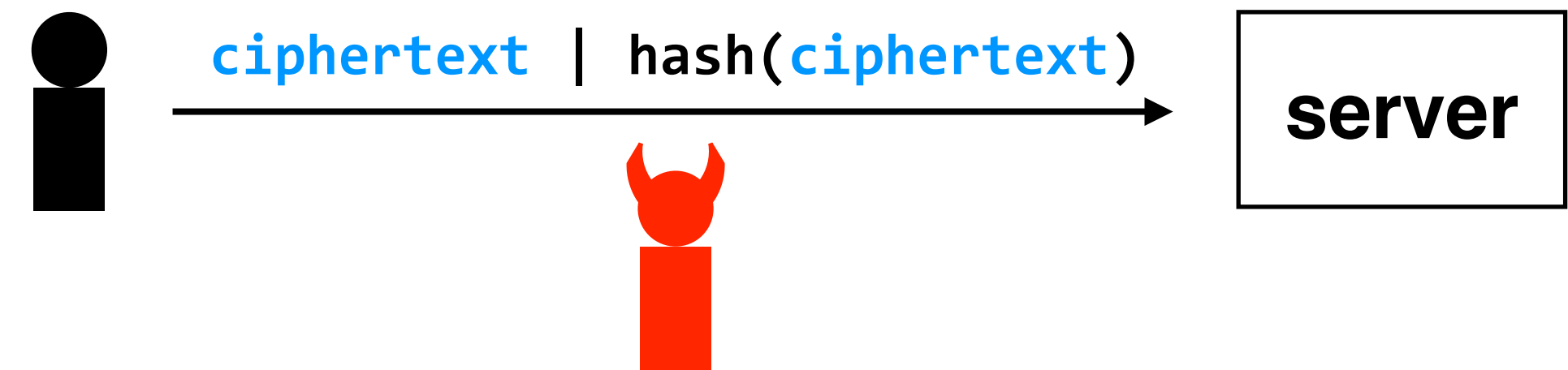
`encrypt(key, message) → ciphertext`

`decrypt(key, ciphertext) → message`

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679  
26cd393d4b93c58f78c
```

```
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7  
8c") = hello, world
```

property: given the **ciphertext**, it is
(virtually) impossible to obtain the
message without knowing the **key**



no good — if the adversary changes **ciphertext**, it can also
(correctly) update the hash

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

encrypt(**key**, **message**) → **ciphertext**

decrypt(**key**, **ciphertext**) → **message**

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

MAC(**key**, **message**) → **token**

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**

it is also impossible to go in the reverse direction: given **token**, you can't get **message** even with the **key**



policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

encrypt(**key**, **message**) → **ciphertext**

decrypt(**key**, **ciphertext**) → **message**

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

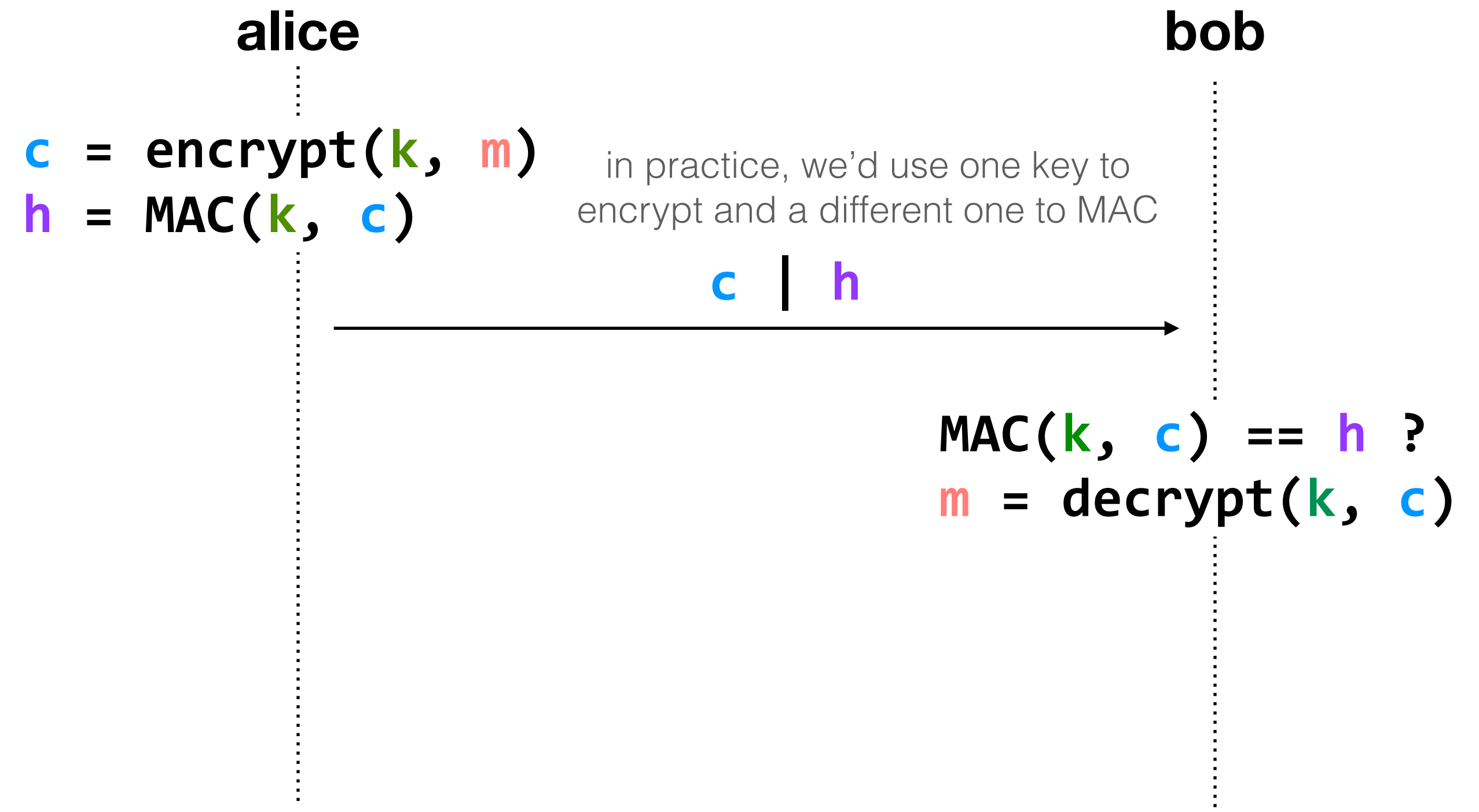
property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

MAC(**key**, **message**) → **token**

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**

it is also impossible to go in the reverse direction: given **token**, you can't get **message** even with the **key**



policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

encrypt(**key**, **message**) → **ciphertext**

decrypt(**key**, **ciphertext**) → **message**

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

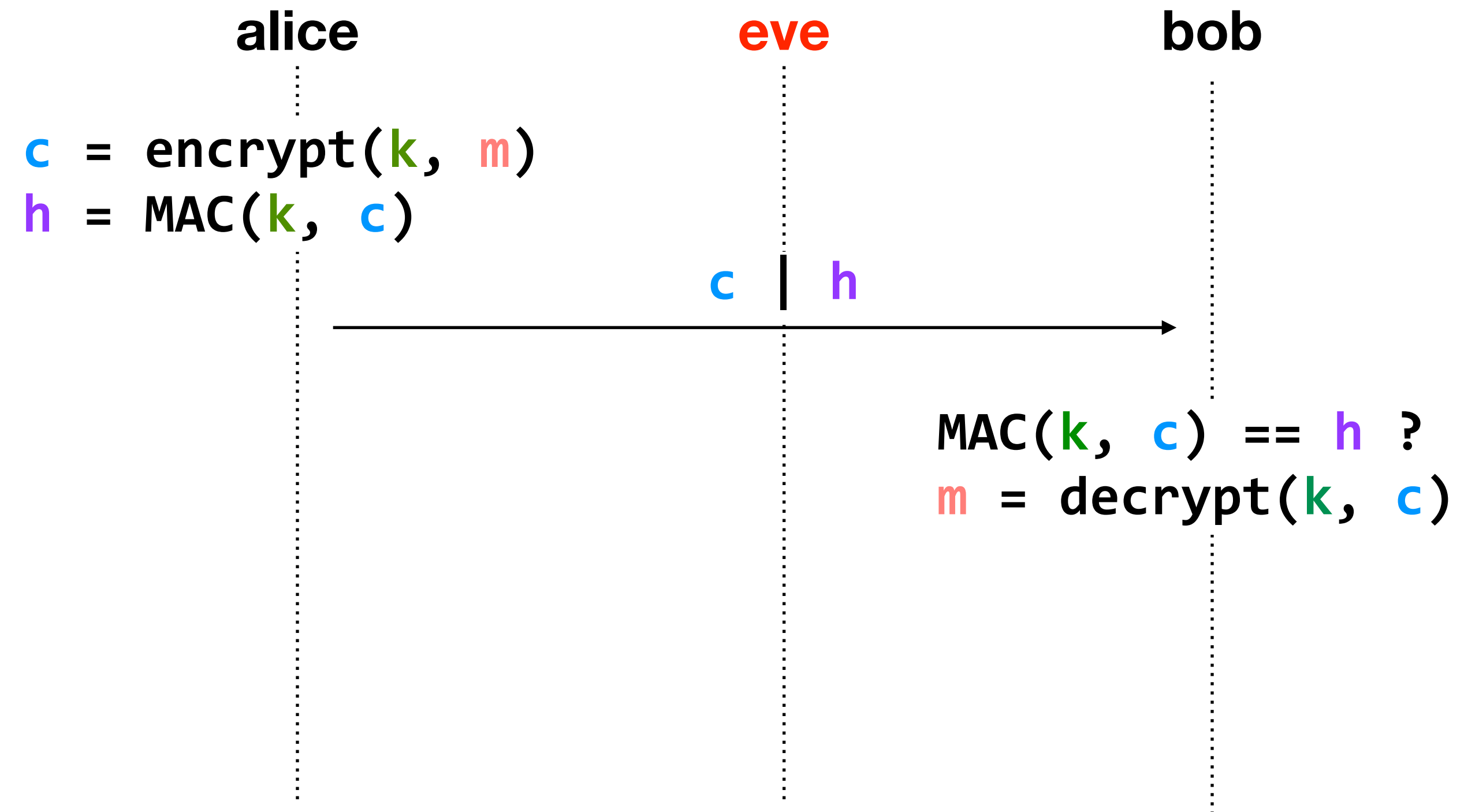
property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

MAC(**key**, **message**) → **token**

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**

it is also impossible to go in the reverse direction: given **token**, you can't get **message** even with the **key**



eve can neither read **m** nor tamper with **c**
(without going unnoticed)

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

encrypt(**key**, **message**) → **ciphertext**

decrypt(**key**, **ciphertext**) → **message**

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

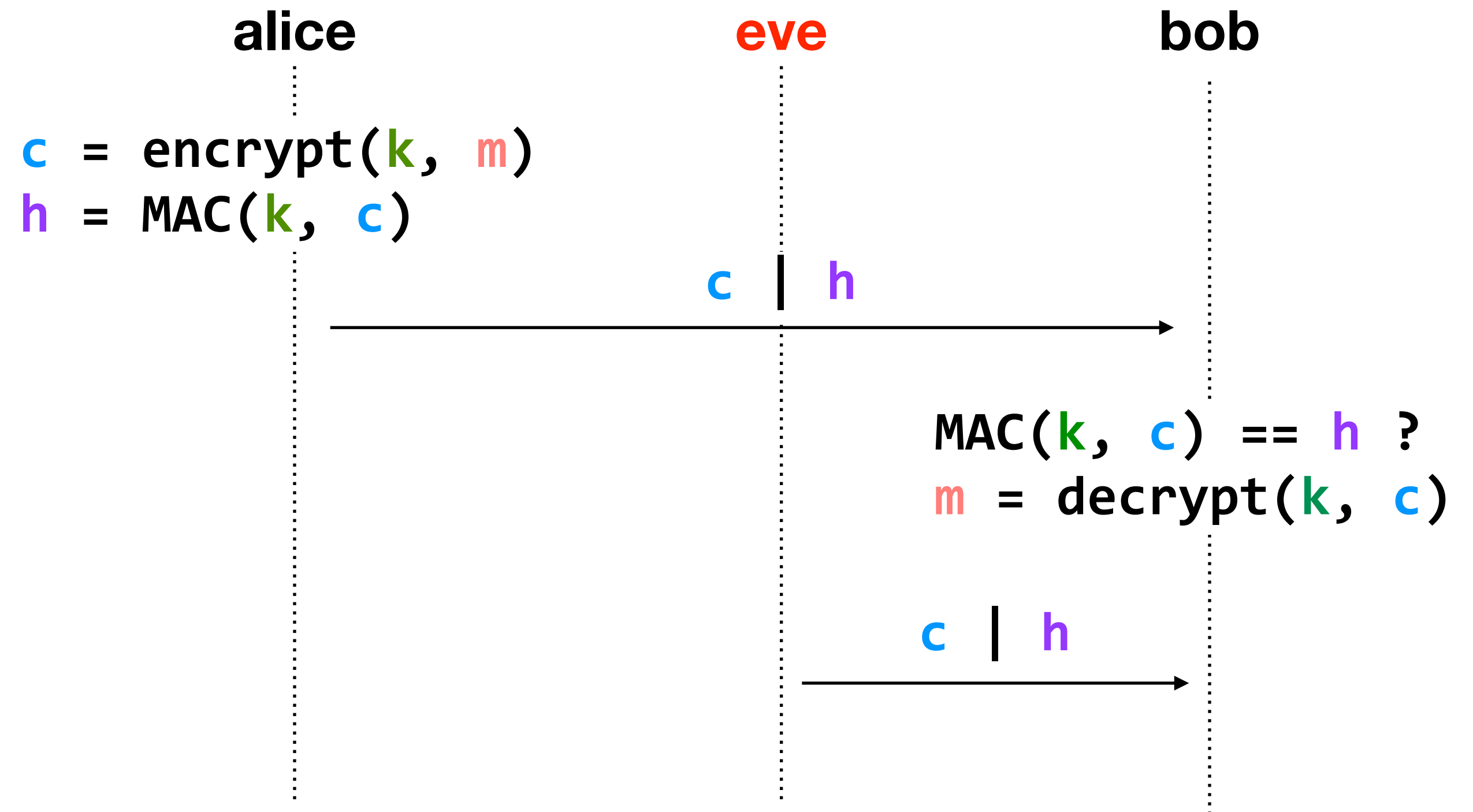
property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

MAC(**key**, **message**) → **token**

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**

it is also impossible to go in the reverse direction: given **token**, you can't get **message** even with the **key**



problem: replay attacks

eve could intercept a message, re-send it at a later time

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

encrypt(**key**, **message**) → **ciphertext**

decrypt(**key**, **ciphertext**) → **message**

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

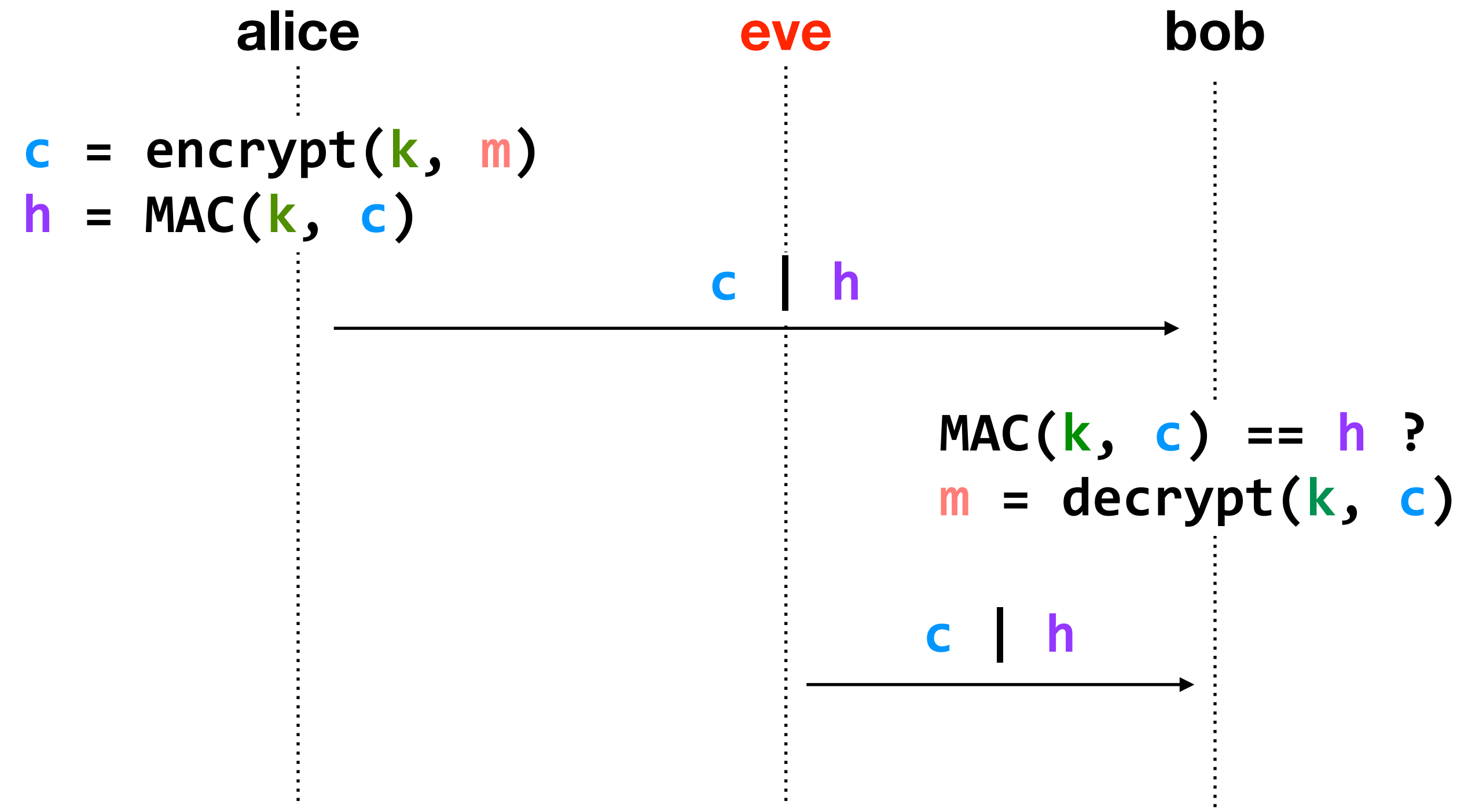
property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

MAC(**key**, **message**) → **token**

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**

it is also impossible to go in the reverse direction: given **token**, you can't get **message** even with the **key**



question: why would **eve** do this?

can you think of times when re-sending a message would cause damage?

bonus question: do you know any techniques to mitigate this attack?

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

encrypt(**key**, **message**) → **ciphertext**

decrypt(**key**, **ciphertext**) → **message**

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

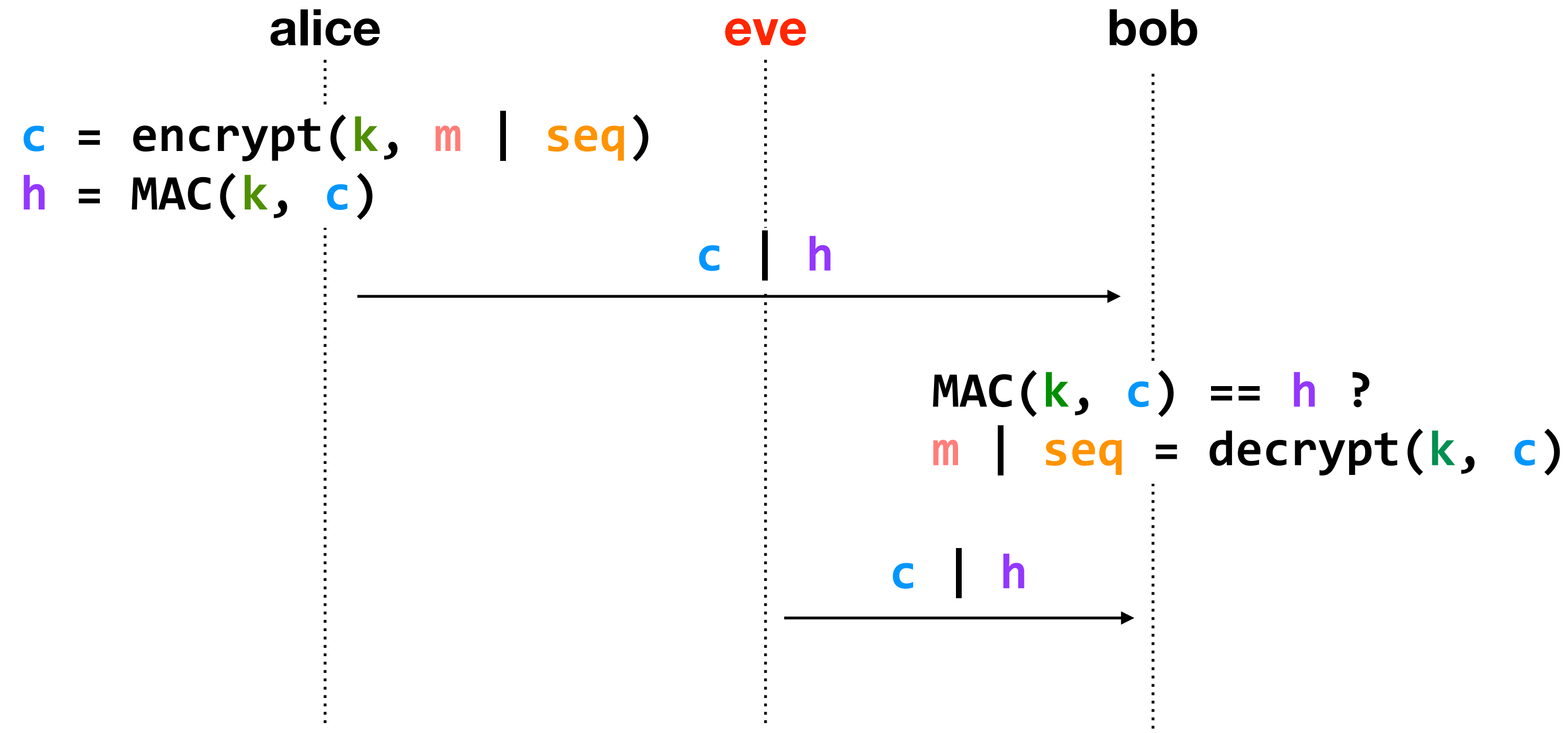
property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

MAC(**key**, **message**) → **token**

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**

it is also impossible to go in the reverse direction: given **token**, you can't get **message** even with the **key**



if eve replays the message, bob will notice
because bob has already seen this sequence number

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

encrypt(key, message) → ciphertext

decrypt(key, ciphertext) → message

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

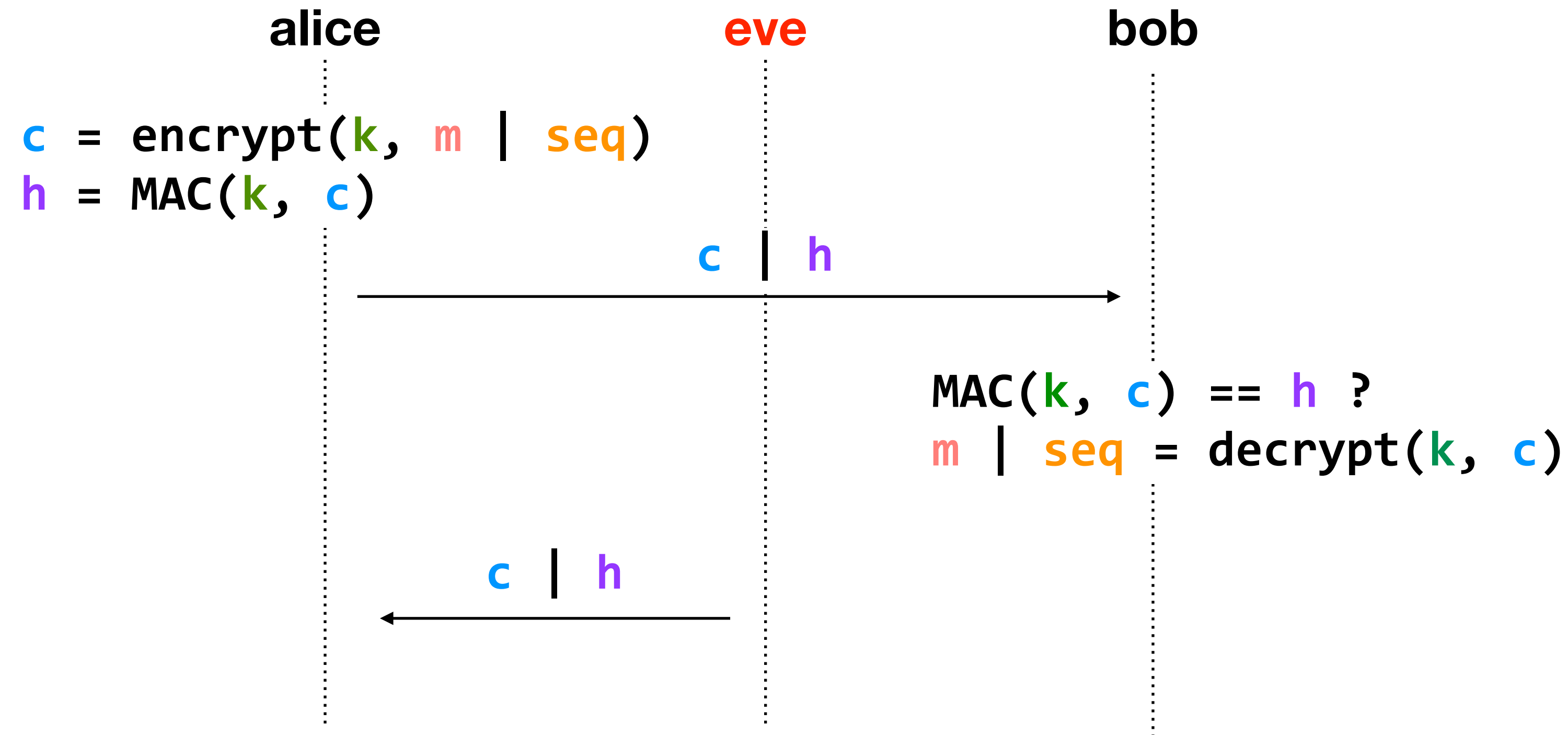
property: given the ciphertext, it is (virtually) impossible to obtain the message without knowing the key

MAC(key, message) → token

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the message, it is (virtually) impossible to obtain the token without knowing the key

it is also impossible to go in the reverse direction: given token, you can't get message even with the key



problem: reflection attacks

eve could intercept a message, re-send it at a later time in the opposite direction

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

`encrypt(key, message) → ciphertext`
`decrypt(key, ciphertext) → message`

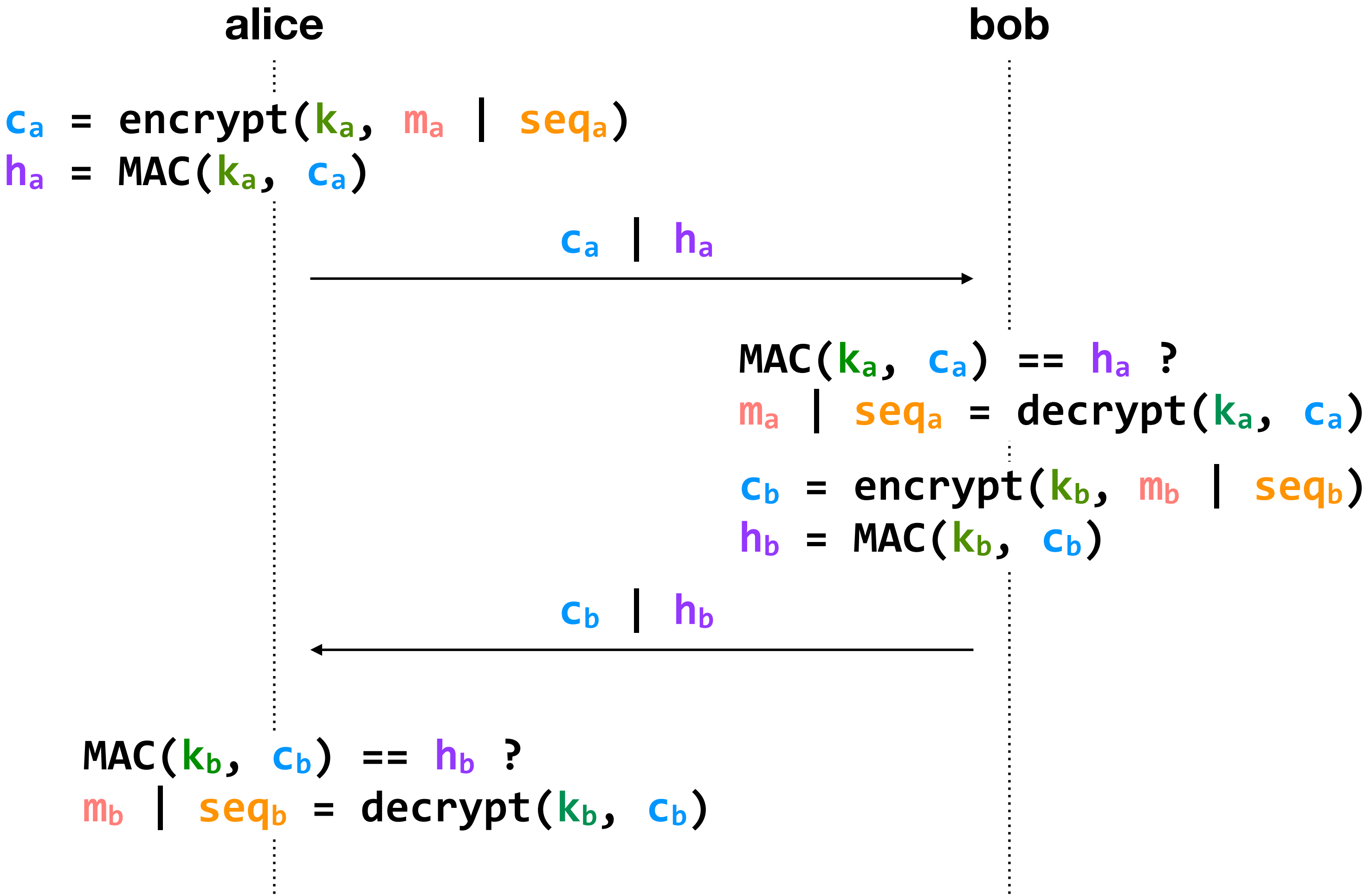
`encrypt(34fbcbd1, "hello, world") = 0x47348f63a67926cd393d4b93c58f78c`
`decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f78c") = hello, world`

property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

`MAC(key, message) → token`
`MAC(34fbcbd1, "hello, world") = 0x59cccc95723737f777e62bc756c8da5c`

property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**

it is also impossible to go in the reverse direction: given **token**, you can't get **message** even with the **key**



problem: how do the parties know the keys?

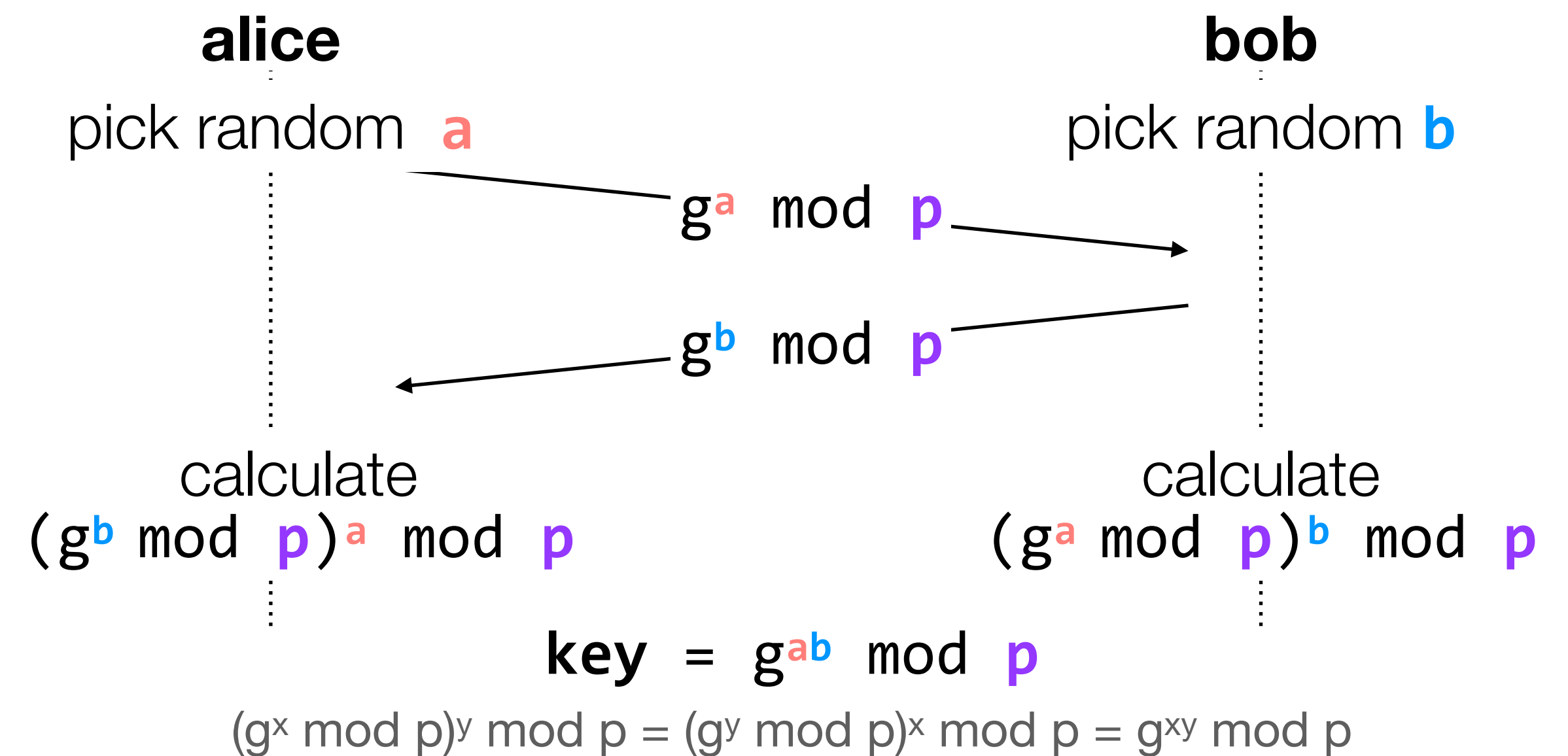
policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

$x \bmod y$ is the remainder when **x**
is divided by **y**
e.g., $10 \bmod 8 = 2$; $23 \bmod 10 = 3$

known to everyone: **p** (prime), **g**
 g and p are related mathematically (g is a
“primitive root” mod p). this relationship makes
the next property possible.

property: given **$g^r \bmod p$** , it is
(virtually) impossible to determine **r**
even if you know g and p



an observer on the network knows p , g , $g^a \bmod p$, and $g^b \bmod p$, but cannot use that information to learn a or b
and thus cannot calculate the key

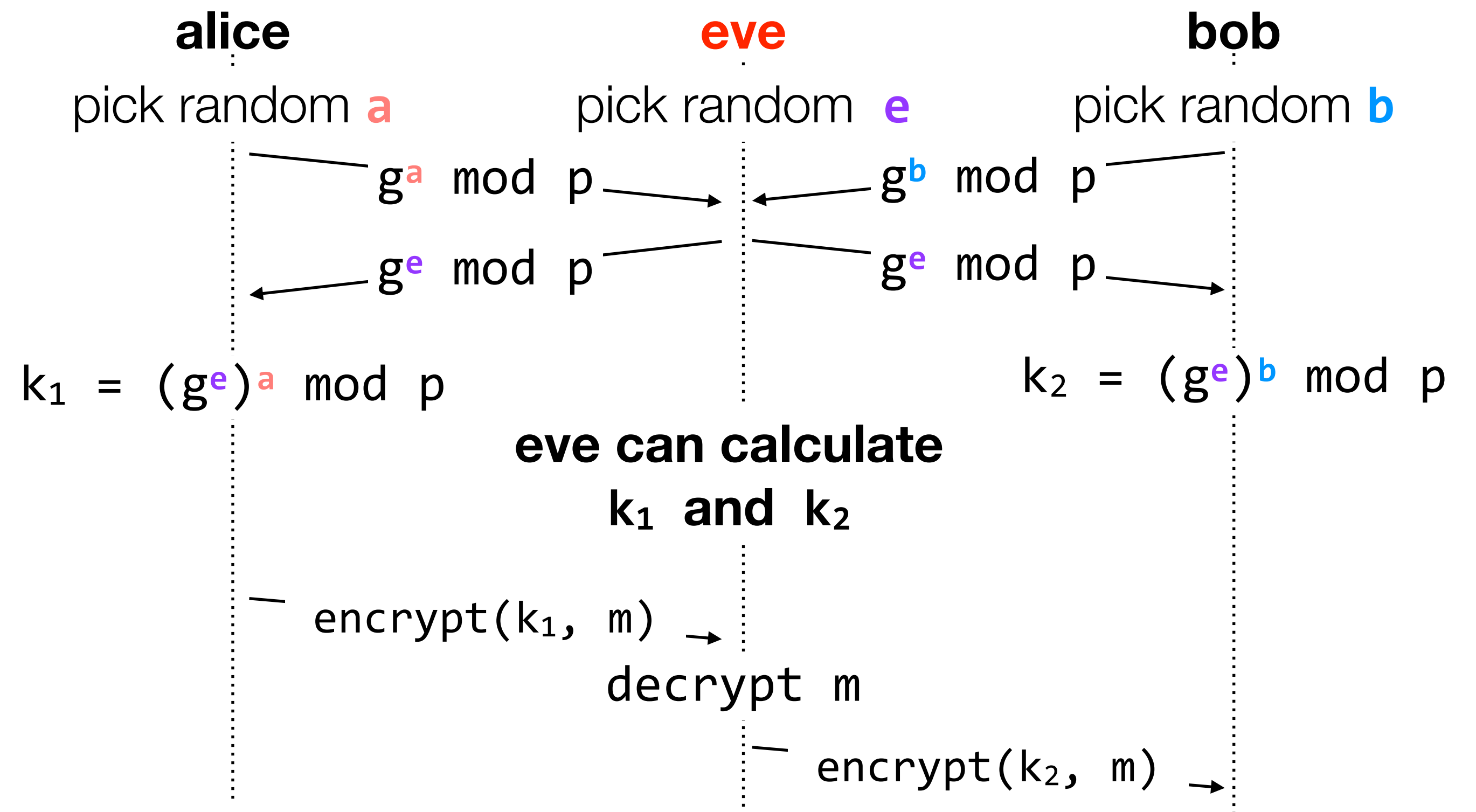
policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

$x \bmod y$ is the remainder when **x**
is divided by **y**
e.g., $10 \bmod 8 = 2$; $23 \bmod 10 = 3$

known to everyone: **p** (prime), **g**
 g and p are related mathematically (g is a
“primitive root” mod p). this relationship makes
the next property possible.

property: given **$g^r \bmod p$** , it is
(virtually) impossible to determine **r**
*even if you know **g** and **p***



problem: alice and bob don't know they're not communicating directly

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

cryptographic signatures allow users to verify identities using public-key cryptography

users generate **key pairs**; the two keys in the pair are related mathematically

{public_key, secret_key}

`sign(secret_key, message) → sig`
`verify(public_key, message, sig) → yes/no`

property: it is (virtually) impossible to compute **sig** without **secret_key**



this is a *rough outline* of how to think about public signatures in the context of this lecture. in reality, things work a bit differently; you'll see an example in a few minutes

how do we distribute public keys?

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

cryptographic signatures allow users to verify identities using public-key cryptography

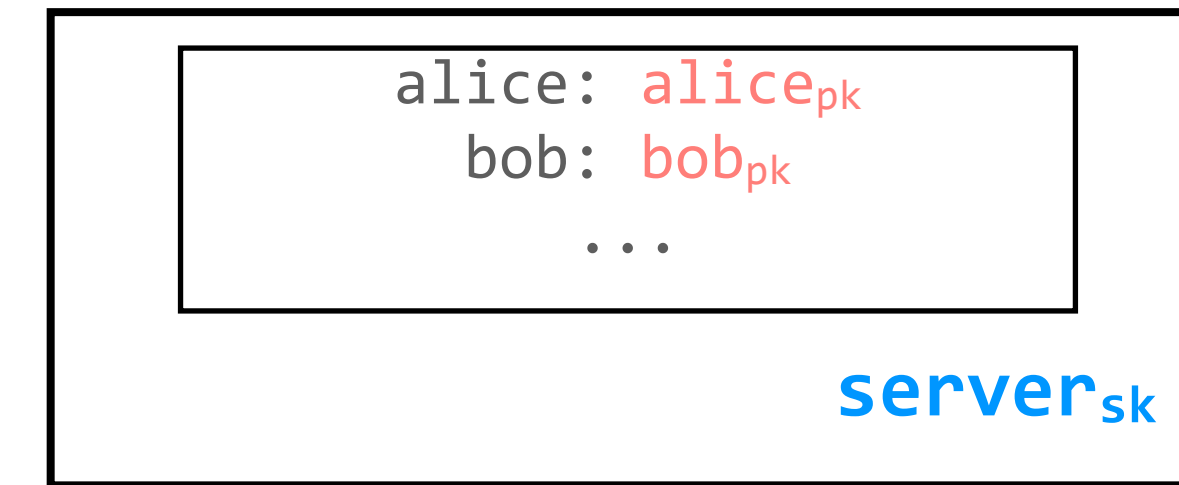
users generate **key pairs**; the two keys in the pair are related mathematically

{**public_key**, **secret_key**}

sign(**secret_key**, message) → **sig**
verify(**public_key**, message, **sig**) → yes/no

property: it is (virtually) impossible to compute **sig** without **secret_key**

alice
alice_{sk}



server_{pk}

bob
bob_{sk}

alice and bob could ask the server for any public keys they need, but that doesn't scale, and we also have to figure out how to distribute the server's public key

x_{pk} = x's public key

x_{sk} = x's secret key (known only to x)

how do we distribute public keys?

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

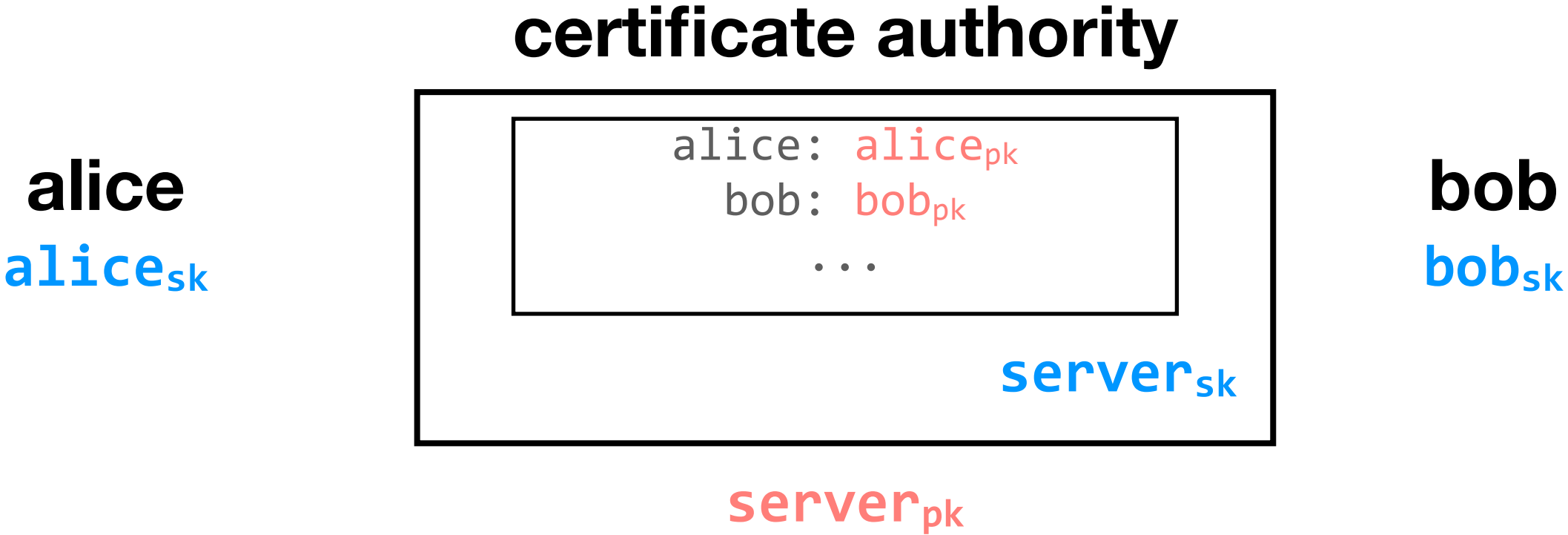
cryptographic signatures allow users to verify identities using public-key cryptography

users generate **key pairs**; the two keys in the pair are related mathematically

$\{\text{public_key}, \text{secret_key}\}$

$\text{sign}(\text{secret_key}, \text{message}) \rightarrow \text{sig}$
 $\text{verify}(\text{public_key}, \text{message}, \text{sig}) \rightarrow \text{yes/no}$

property: it is (virtually) impossible to compute **sig** without **secret_key**



server pre-computes **signed** messages that map names to their public keys

$\text{sign}(\text{server}_{sk}, \text{"alice: } \text{alice}_{pk}\text{"}) \rightarrow \text{sig}$

$\text{alice, } \text{alice}_{pk}, \text{sig}$

certificate

anyone can verify that the authority signed this message given server_{pk} , but the server itself doesn't have to distribute the signed messages

policy: provide **confidentiality** (adversary cannot learn message contents) and **integrity** (adversary cannot tamper with packets and go undetected)

threat model: adversary can observe network data, tamper with packets, and insert its own packets

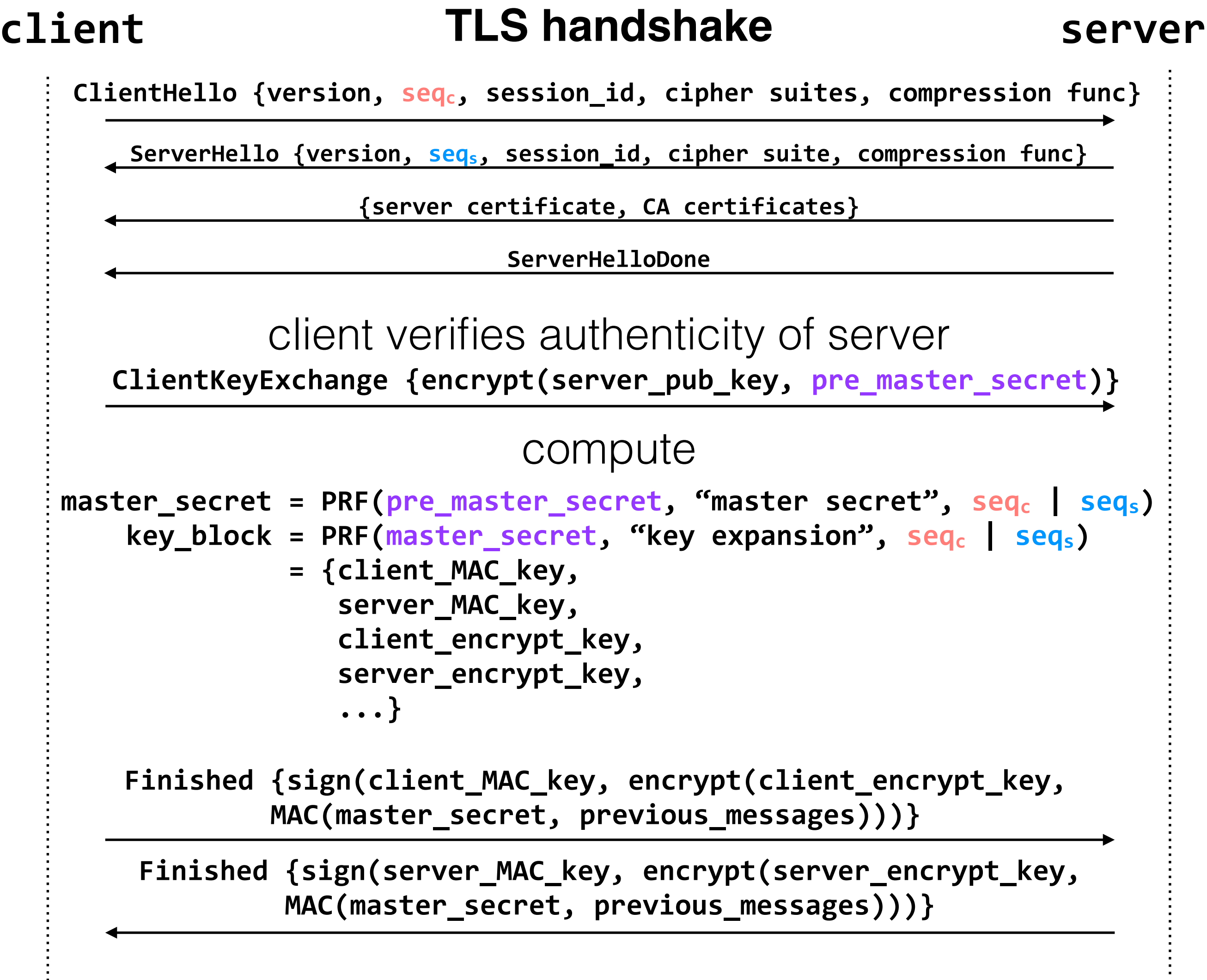
cryptographic signatures allow users to verify identities using public-key cryptography

users generate **key pairs**; the two keys in the pair are related mathematically

{**public_key**, **secret_key**}

sign(**secret_key**, message) → **sig**
verify(**public_key**, message, **sig**) → yes/no

property: it is (virtually) impossible to compute **sig** without **secret_key**



6.1800 in the news

OOPS —

Major cryptography blunder in Java enables “psychic paper” forgeries

A failure to sanity check signatures for division-by-zero flaws makes forgeries easy.

DAN GOODIN - 4/20/2022, 3:28 PM

ECDSA signatures rely on a pseudo-random number, typically notated as K , that's used to derive two additional numbers, R and S . To verify a signature as valid, a party must check the equation involving R and S , the signer's public key, and a cryptographic hash of the message. When both sides of the equation are equal, the signature is valid.

In a **writeup published Wednesday**, security firm Sophos further explained the process:

“

- S1. Select a cryptographically sound random integer K between 1 and $N-1$ inclusive.
- S2. Compute R from K using Elliptic Curve multiplication.
- S3. In the unlikely event that R is zero, go back to step 1 and start over.
- S4. Compute S from K , R , the hash to be signed, and the private key.
- S5. In the unlikely event that S is zero, go back to step 1 and start over.

6.1800 in the news

OOPS —

Major cryptography blunder in Java enables “psychic paper” forgeries

A failure to sanity check signatures for division-by-zero flaws makes forgeries easy.

DAN GOODIN - 4/20/2022, 3:28 PM

Madden wrote:

“

Guess which check Java forgot?

That’s right. Java’s implementation of ECDSA signature verification didn’t check if R or S were zero, so you could produce a signature value in which they are both 0 (appropriately encoded) and Java would accept it as a valid signature for any message and for any public key. The digital equivalent of a blank ID card.

encryption provides confidentiality

here, we are using symmetric-key encryption: the same key is used to encrypt *and* decrypt

encrypt(key, message) → ciphertext

decrypt(key, ciphertext) → message

```
encrypt(34fbcbd1, "hello, world") = 0x47348f63a679
26cd393d4b93c58f78c
decrypt(34fbcbd1, "0x47348f63a67926cd393d4b93c58f7
8c") = hello, world
```

property: given the ciphertext, it is (virtually) impossible to obtain the message without knowing the key

MACs provides integrity

MAC(key, message) → token

```
MAC(34fbcbd1, "hello, world") =
0x59cccc95723737f777e62bc756c8da5c
```

property: given the message, it is (virtually) impossible to obtain the token without knowing the key

it is also impossible to go in the reverse direction: given token, you can't get message even with the key

in the next lecture, we are going to use a different style of encryption — public-key encryption — to provide confidentiality in a different system

cryptographic signatures allow users to verify identities using public-key cryptography

users generate **key pairs**; the two keys in the pair are related mathematically

{public_key, secret_key}

sign(secret_key, message) → sig

verify(public_key, message, sig) → yes/no

property: it is (virtually) impossible to compute sig without secret_key

secure channels protect us from adversaries that can observe and tamper with packets in the network

because a secure channel requires an agreement between the client and the server, system designers must think about whether to provide this abstraction, and who is **impacted** if they do (or do not) provide it

encrypting with **symmetric keys** provides confidentiality, and using **MACs** provides integrity. **Diffie-Hellman key exchange** lets us exchange the symmetric key securely

to verify identities, we use **public-key cryptography** and cryptographic **signatures**. we often distributed public keys via **certificate authorities**, though this method is not perfect