

Bob



Alice



# Cryptography

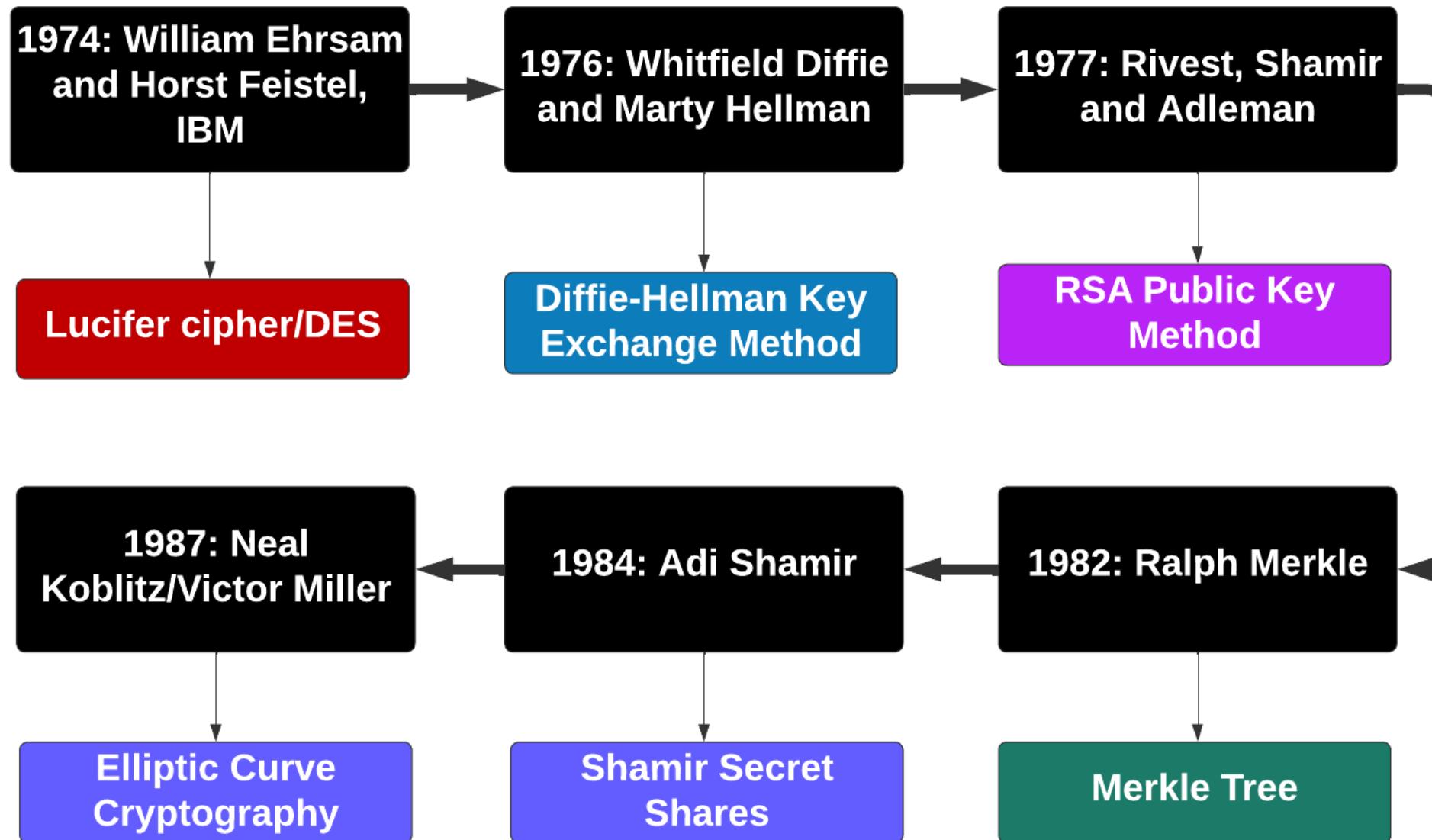
Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com>

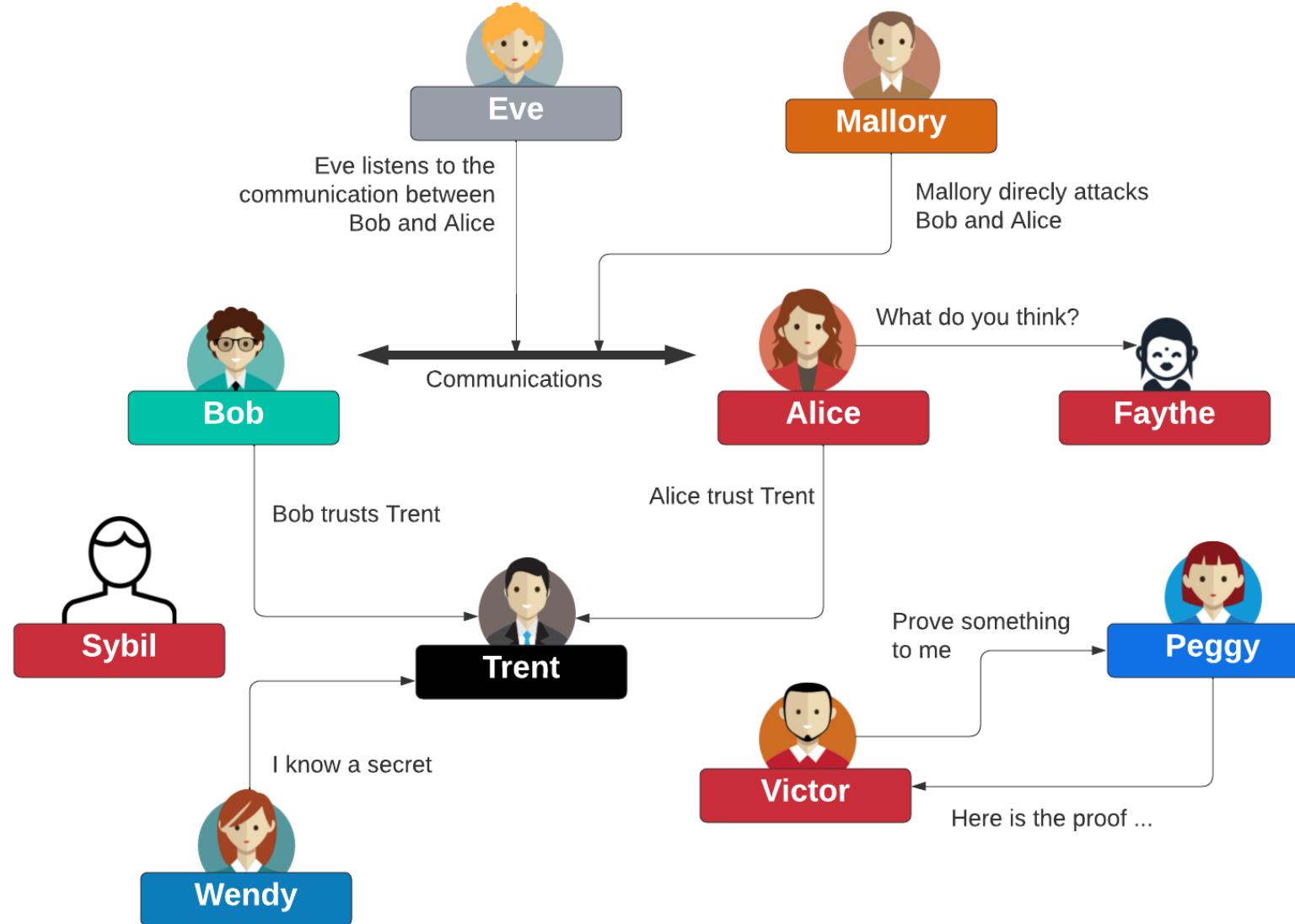
Eve



# A Bit of History



# Some Actors



Bob



Alice



# Cryptography: Fundamentals

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com>

## Fundamentals

Symmetric Key

Hashing

MAC

KDF

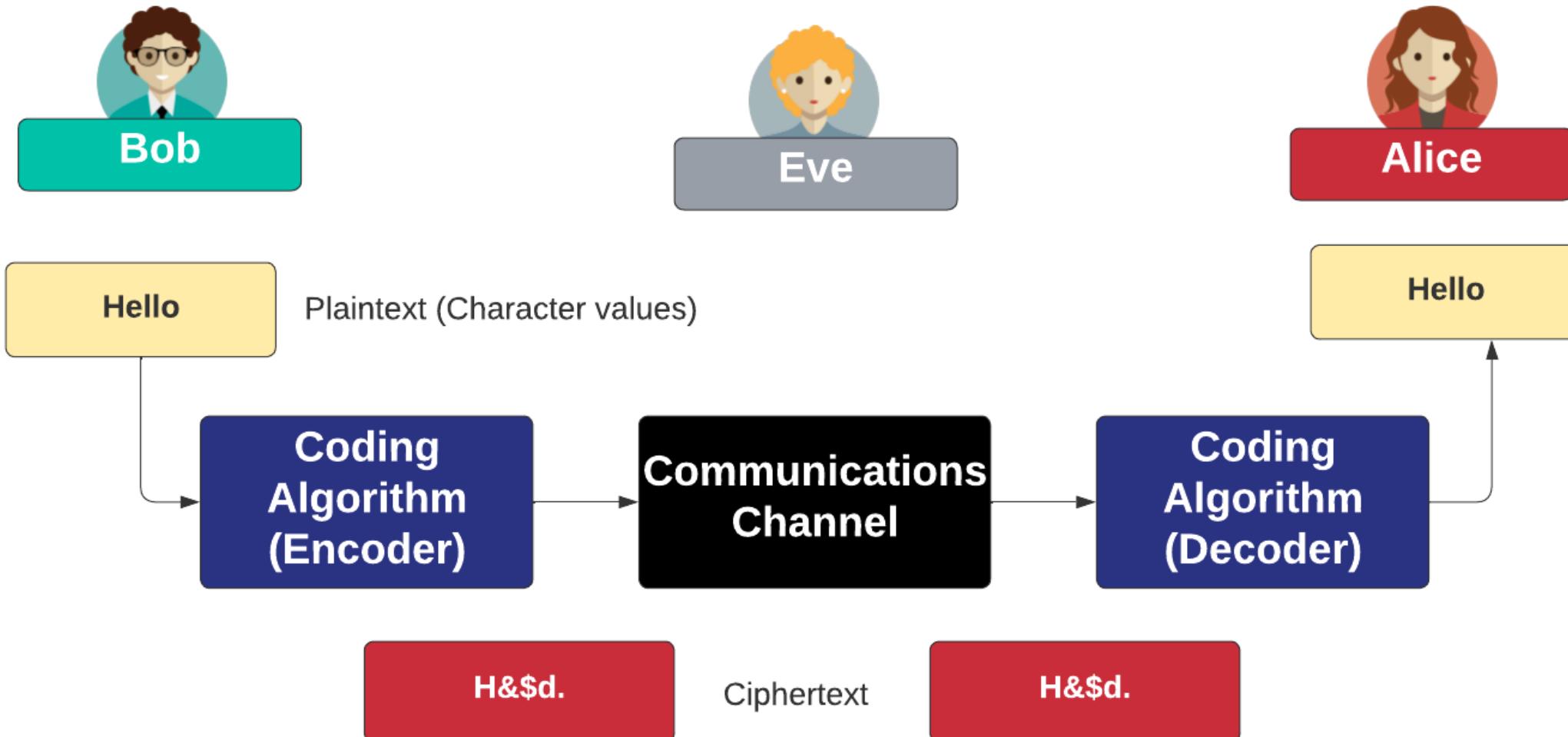
Public Key

Key Exchange

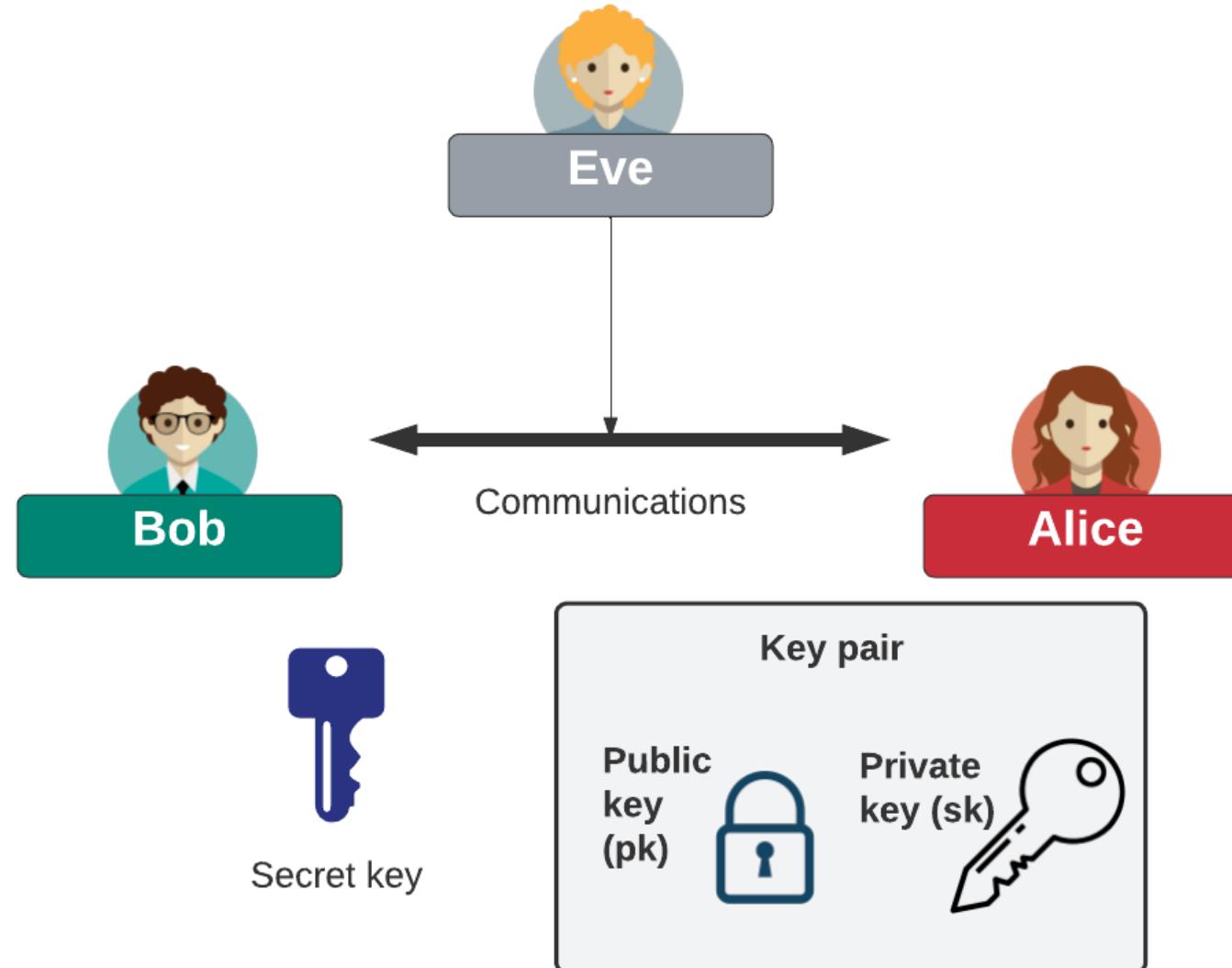
Signatures

Digital Certificates

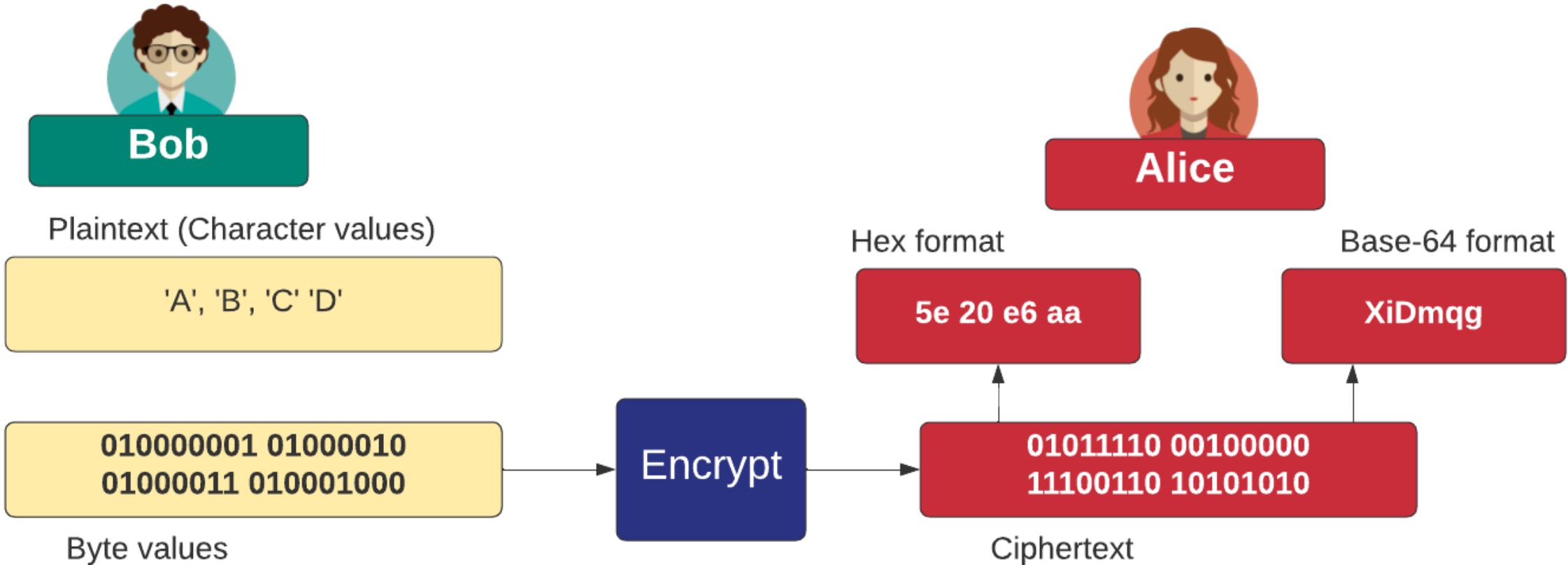
# Encoding and Decoding



# Keys



# Encoding



# Hexadecimal



**Bob**

For hex conversion, take the bit stream, and split into groups of four bits

Bit stream

0101 1110 0010 0000 1110 0110 1010 1010

Hex

5 e 2 0 e 6 a a

5e

20

e6

aa

Bytes

<b>Value</b>	<b>Bit value</b>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

# Base 64



Bob

For Base-64, we take the bit stream and convert into groups of six bits

Bit stream

01100110 01110010 01100101 01100100

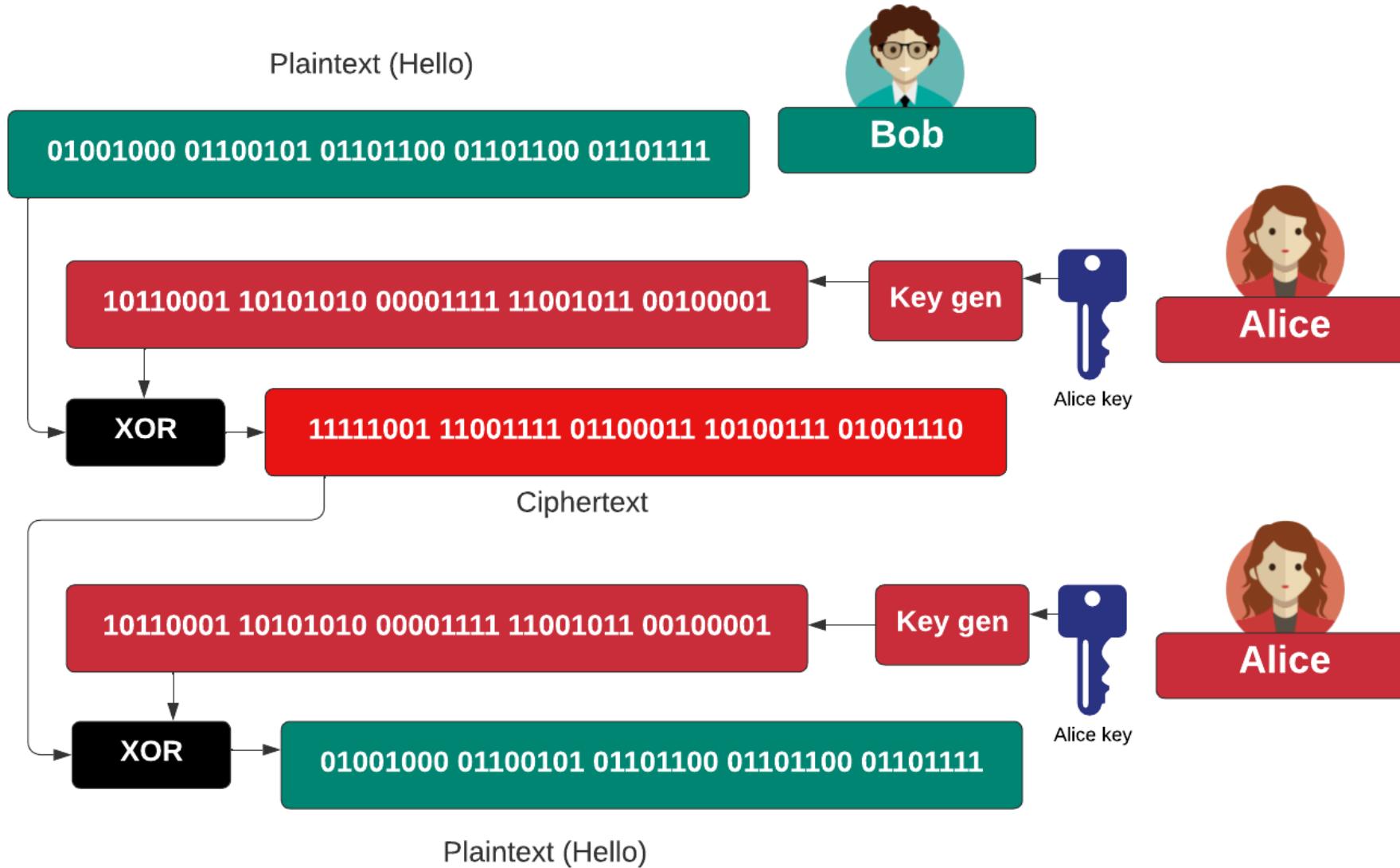
011001 100111 001001 100101 011001 00

Base64

Z n J I Z A ==

Val	Enc	Val	Enc	Val	Enc	Val	Enc
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

# A Basic Operation



Bob



Alice



# Cryptography: Symmetric Key

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com/symmetric>

Fundamentals

Symmetric Key

Hashing

MAC

KDF

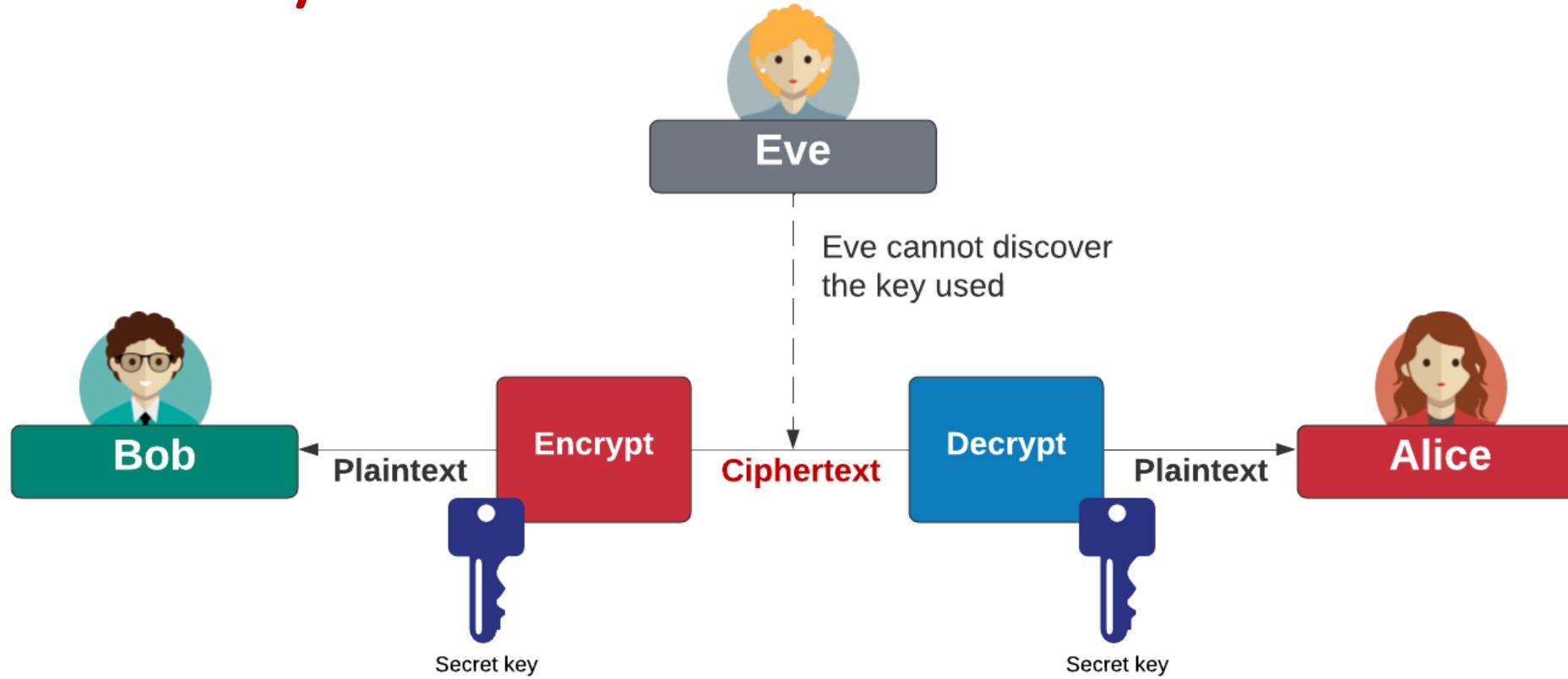
Public Key

Key Exchange

Signatures

Digital Certificates

# Symmetric Key



Block cipher: **AES (128-bit block)**, DES (64-bit block), 3DES (64-bit block), Blowfish, SM4, Camellia, GOST, IDEA, RC2, Serpent, Skipjack, Twofish, and XTEA

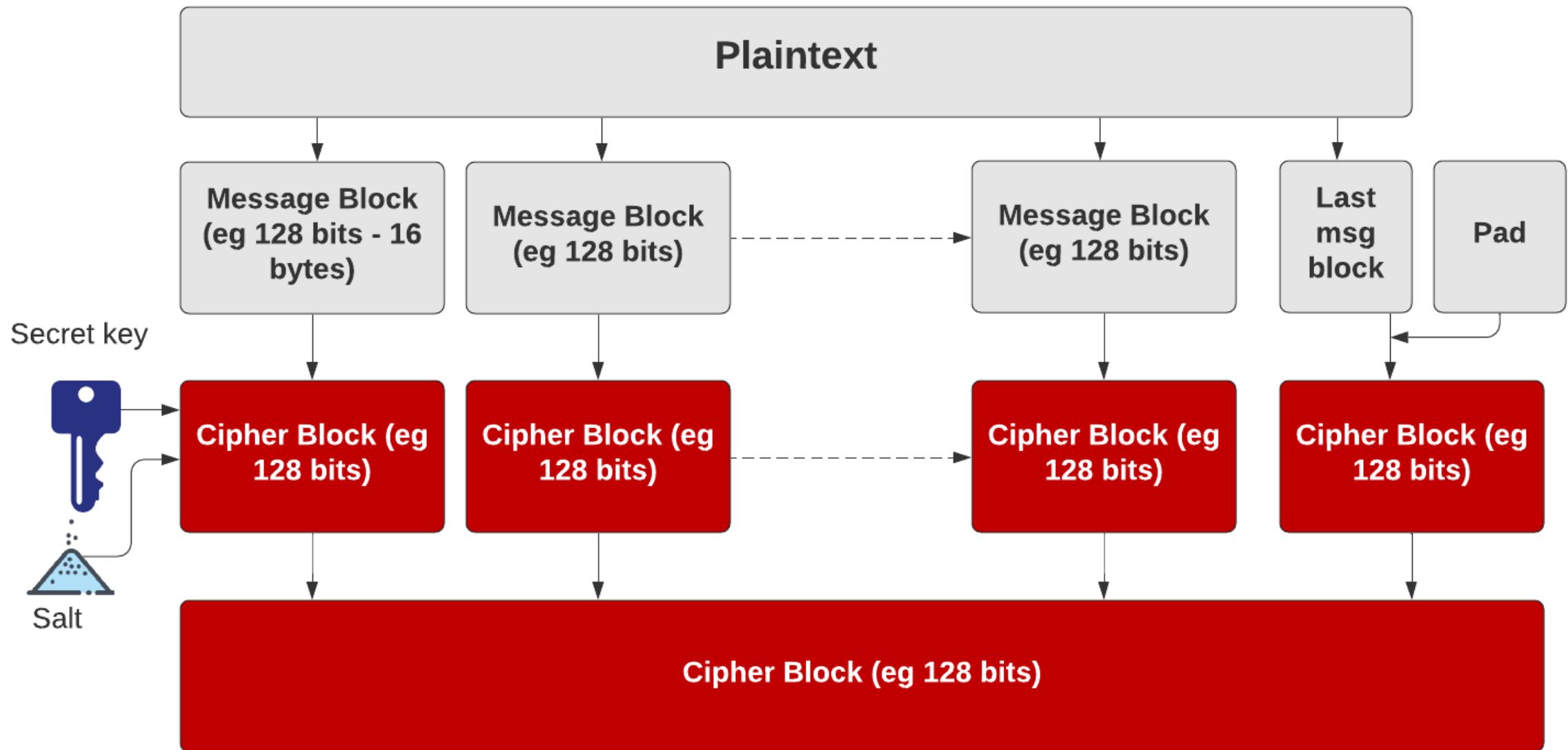
Stream cipher: **ChaCha20**, **AES GCM**, ARIA, Sala20, and RC4.

AES also known as Rijndael (after Joan Daemen and Vincent Rijmen).

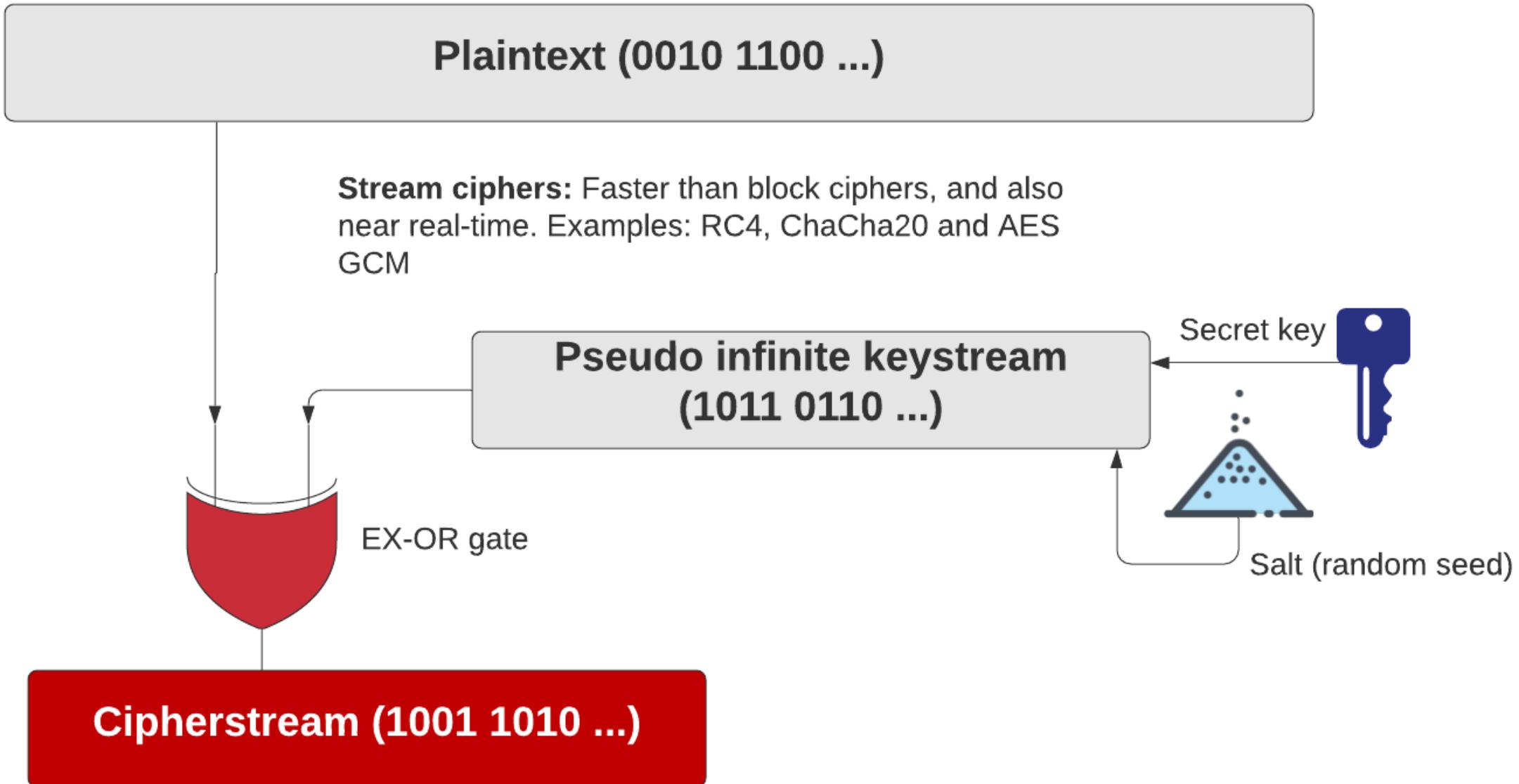
ChaCha20 created by Daniel J Bernstein.

Blowfish and Twofish created by Bruce Schneier.

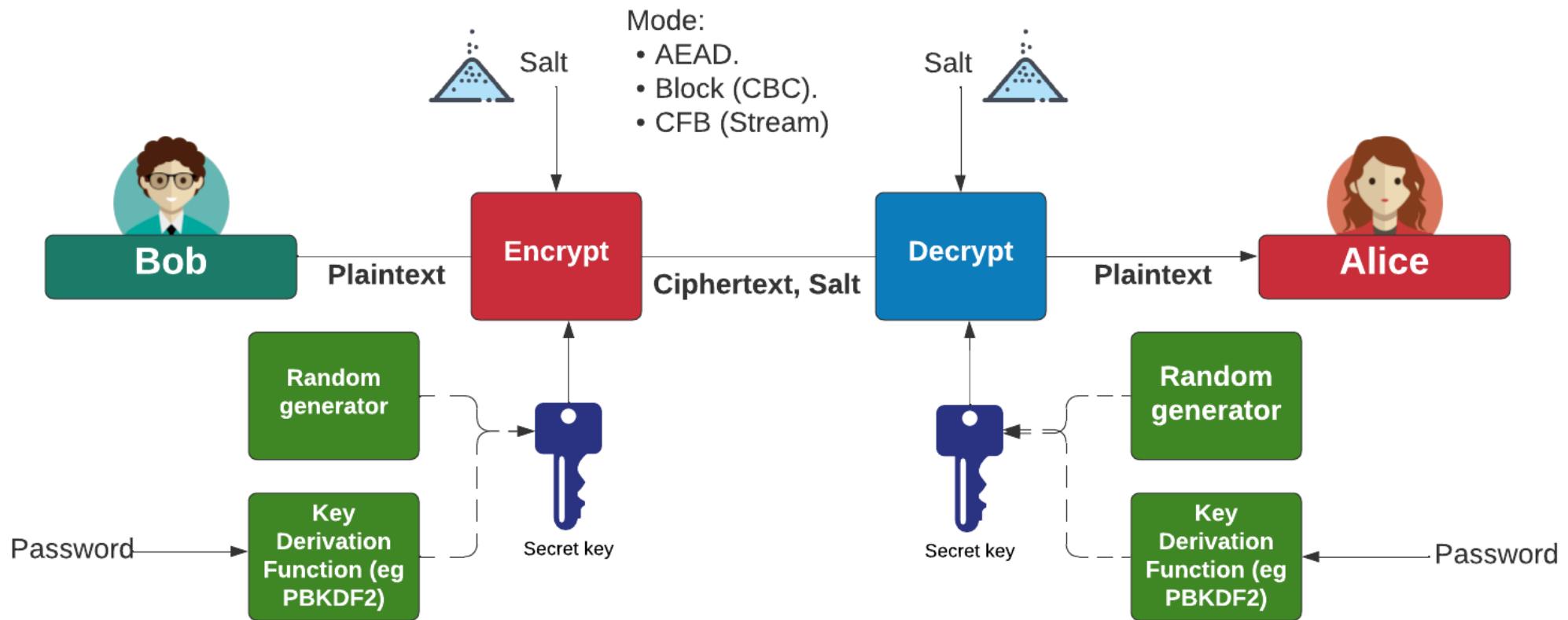
# Symmetric Key – Block cipher



# Symmetric Key – Stream cipher



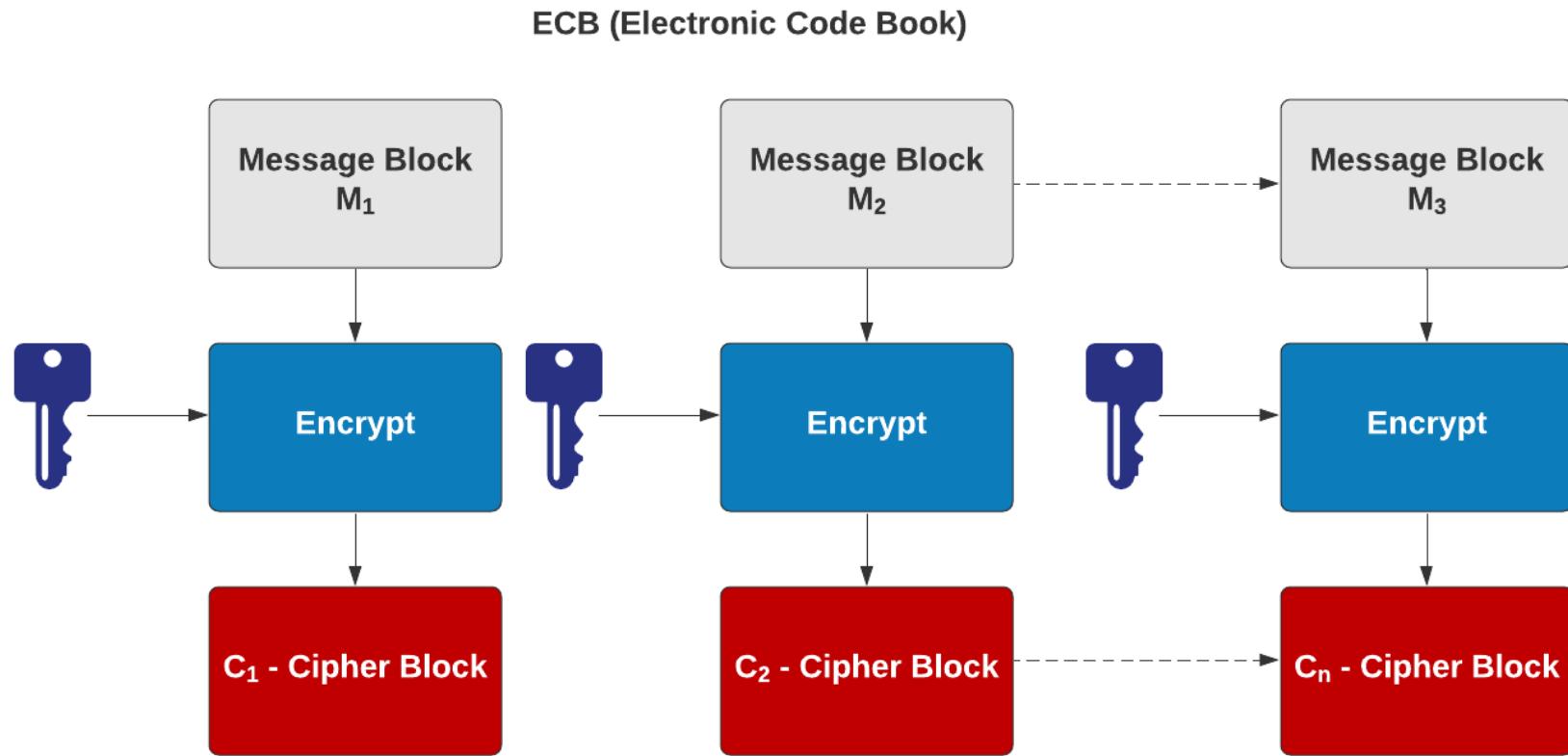
# Symmetric Key - Modes



AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), **GCM** (Galois Counter Mode).

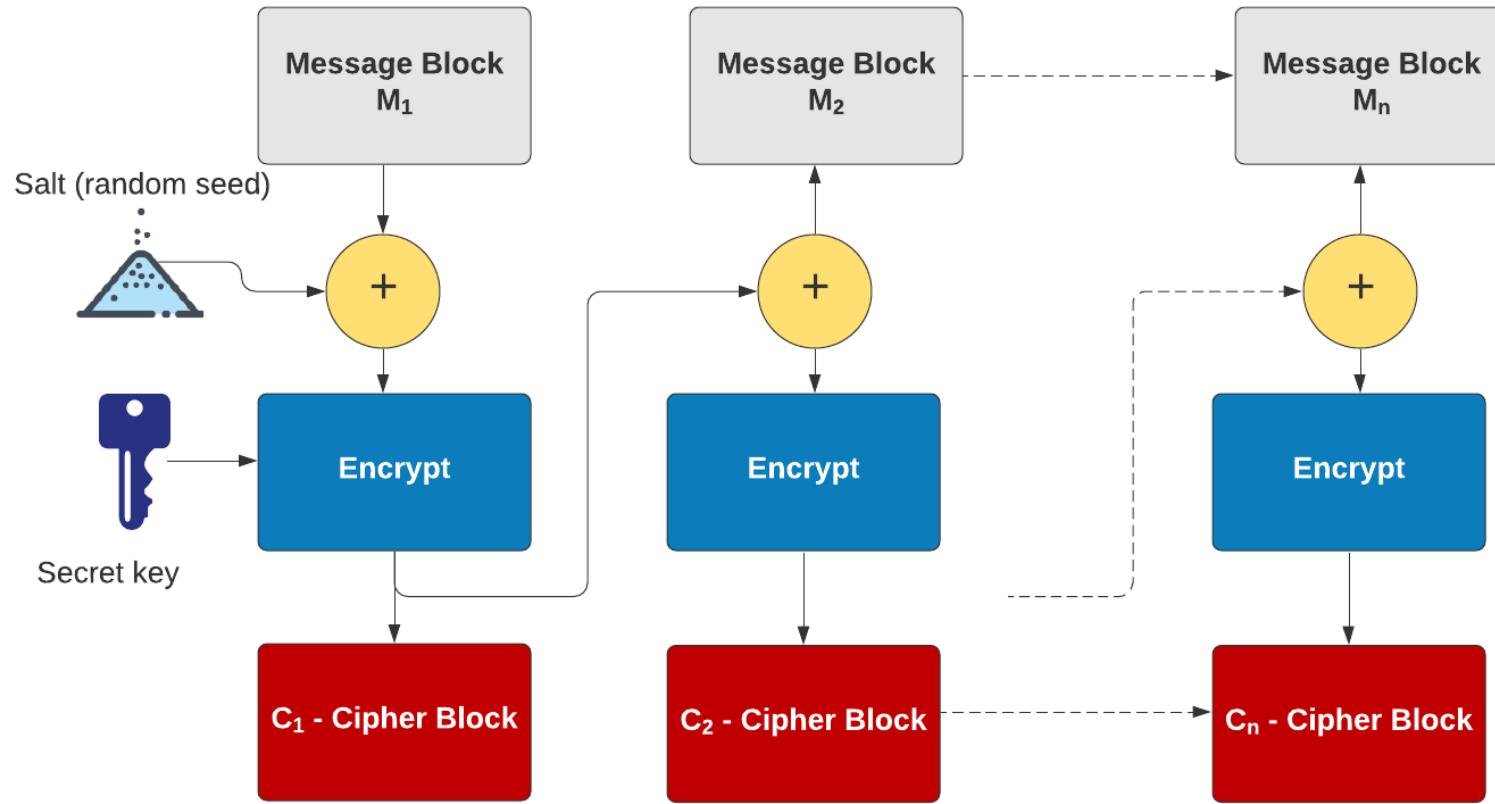
# Symmetric Key – Electronic Code Book (ECB)



AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

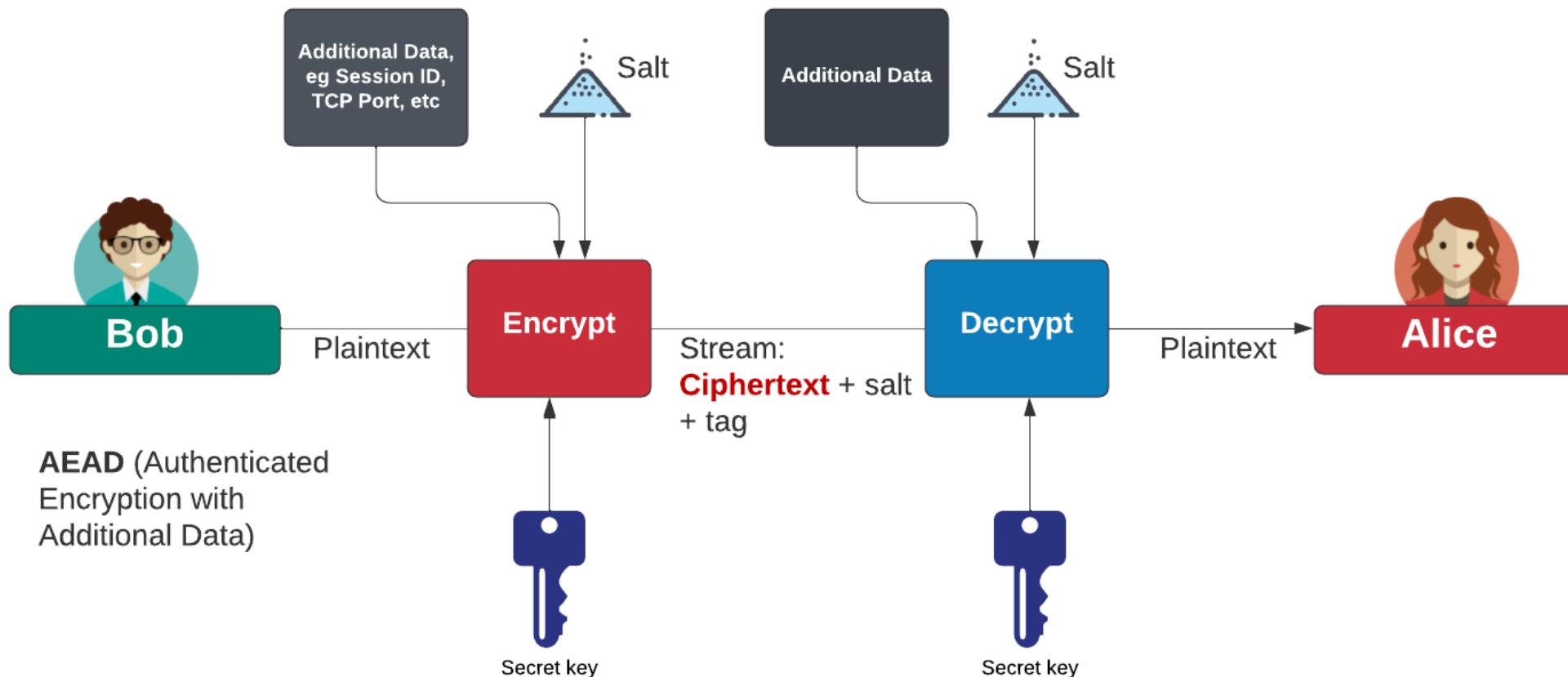
# Symmetric Key – Cipher Block Chaining (CBC)



AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

# Authenticated Encryption with Additional Data (AEAD)



AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

Bob



Alice



# Cryptography: Hashing

Prof Bill Buchanan OBE FRSE

Fundamentals  
Symmetric Key

**Hashing**

MAC

KDF

Public Key

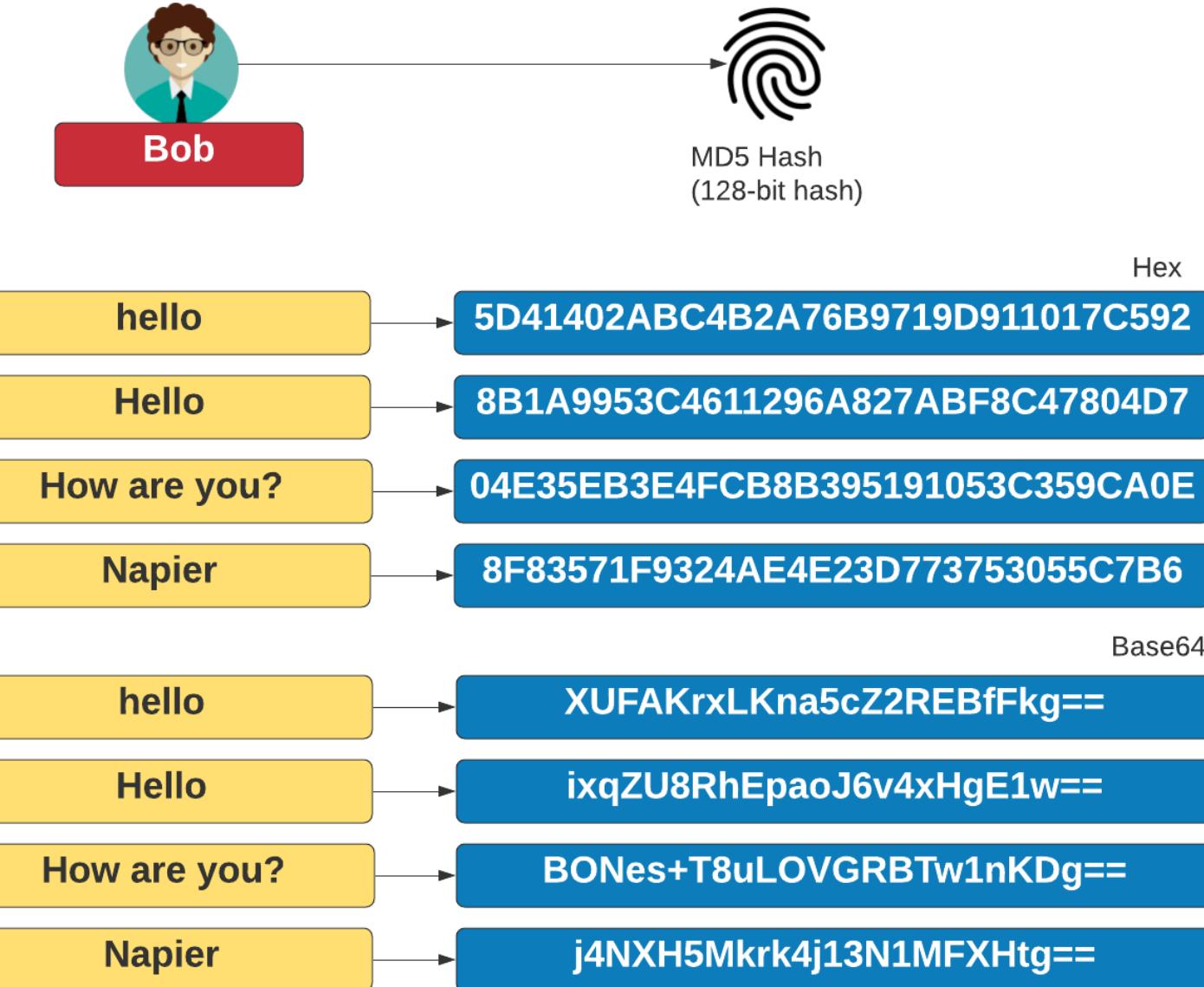
Key Exchange

Signatures

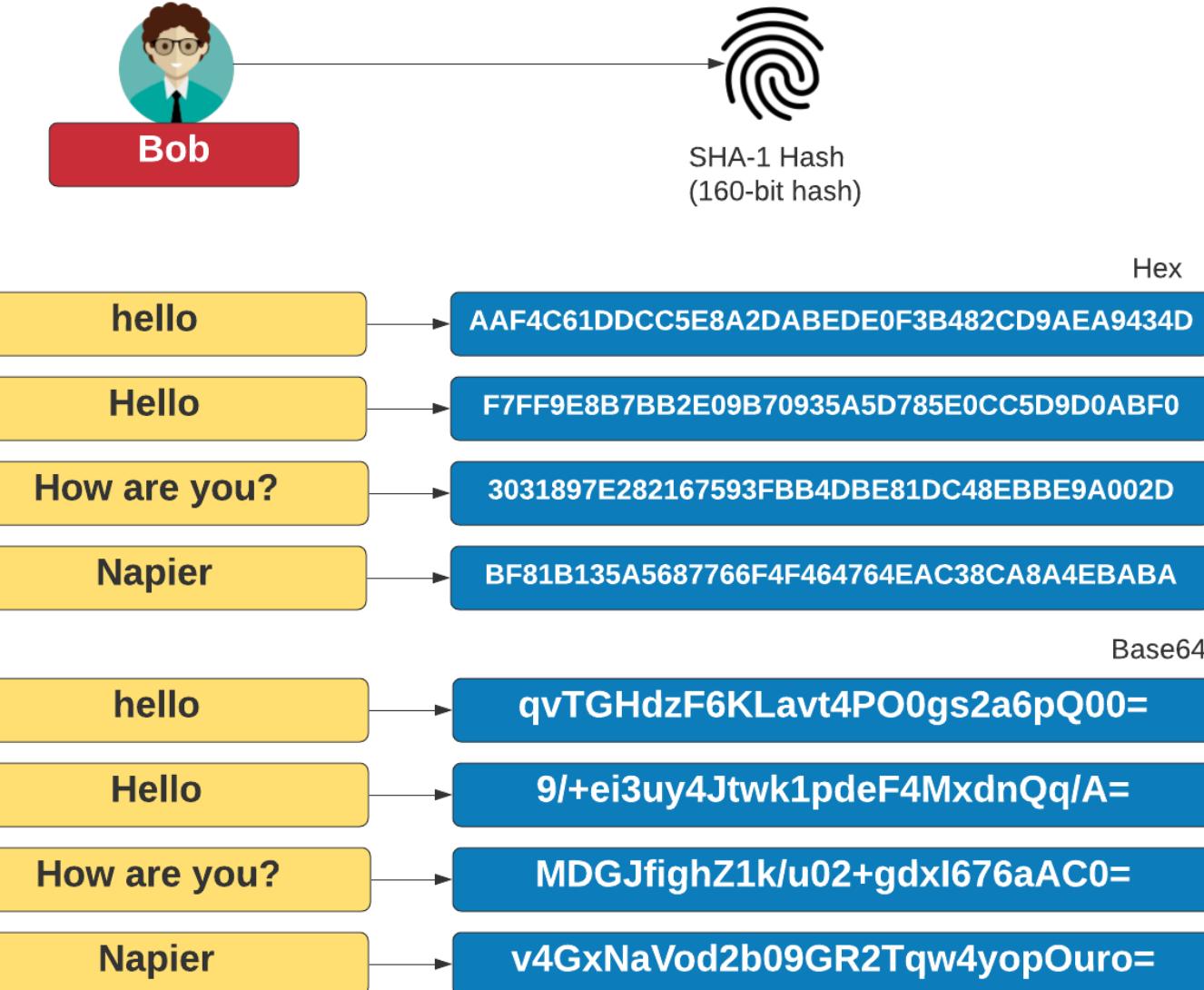
Digital Certificates

<https://asecuritysite.com/hashing>

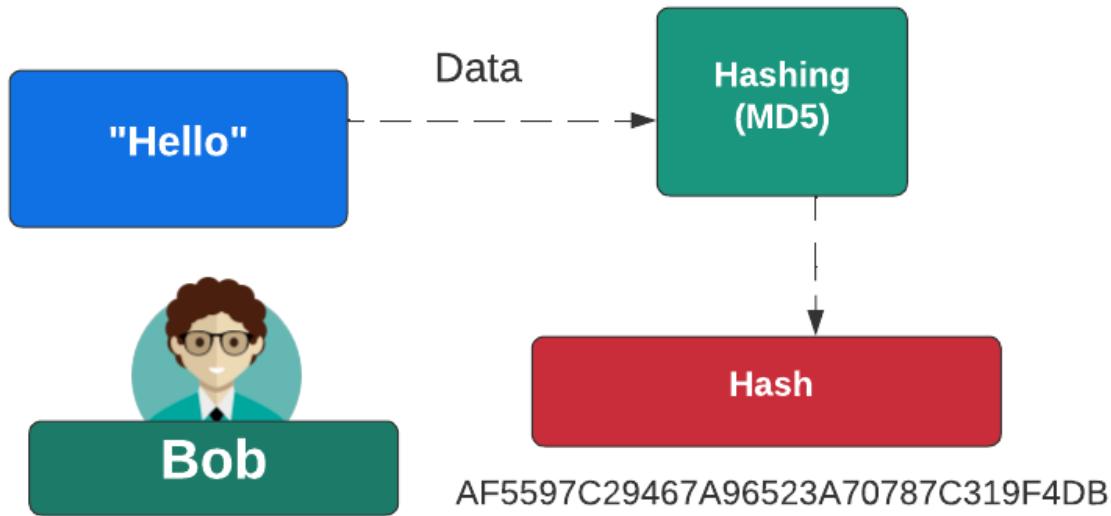
# Hashing – MD5 (128-bit)



# Hashing – SHA-1 (160-bit)



# Password Hashing



Hashed password

Bob: AF5597C29467A96523A70787C319F4DB

Hashed password with salt (**zj8n**hello)

Bob:**zj8n**:51A7C663A3BDCD06D6CE21E2BCB2AD5A

# openssl passwd -1 -s **abc** Hello

Word: Hello

Method: -1

Salt: **abc**

\$1\$**abc**\$0IWzLRntj3IdjLzhKriFx1

Cryptographic Hashes: MD5, SHA-1, SHA-256, SHA-3, RIPEMD, Blake2b

Windows hashes: NTLM

Non-cryptographic hashes: Spooky, Farm, FVN

# Hashing



There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **stop** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

**14375C20F07A9DF2EBDE78F063C5AD7C**

**FDdclPB6nfLr3njwY8WtfA==**

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **top** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

**174F3AB19D8047DAE398CE620407B60B**

**F086sZ2AR9rjmM5iBAe2Cw==**

# Hashing and OpenSSL



There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **stop** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

**14375C20F07A9DF2EBDE78F063C5AD7C**

**FDdclPB6nfLr3njwY8WtfA==**

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **top** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

```
Linux command: echo -n "Hello" | openssl dgst -md5  
Windows command: echo | set /p = "Hello" | openssl dgst -md5
```

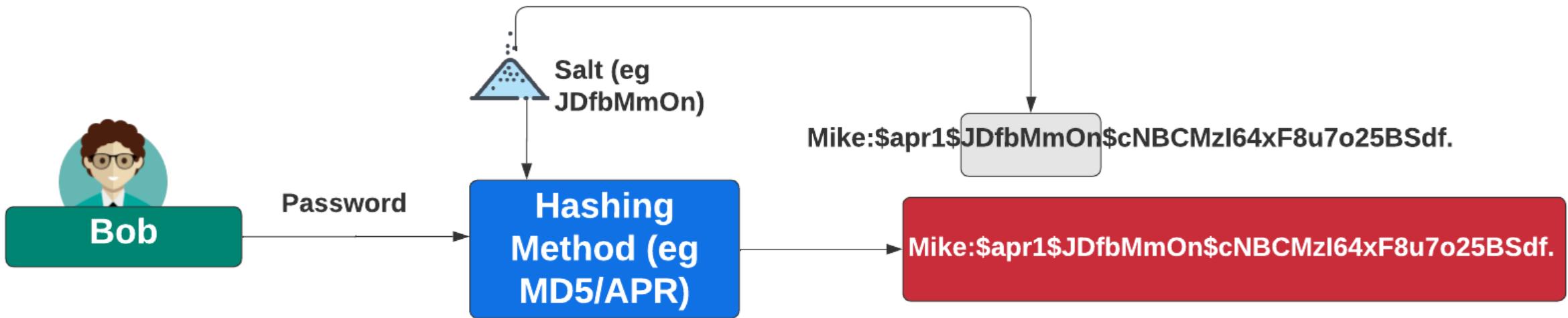
```
Message: Hello
```

```
Mode: md5
```

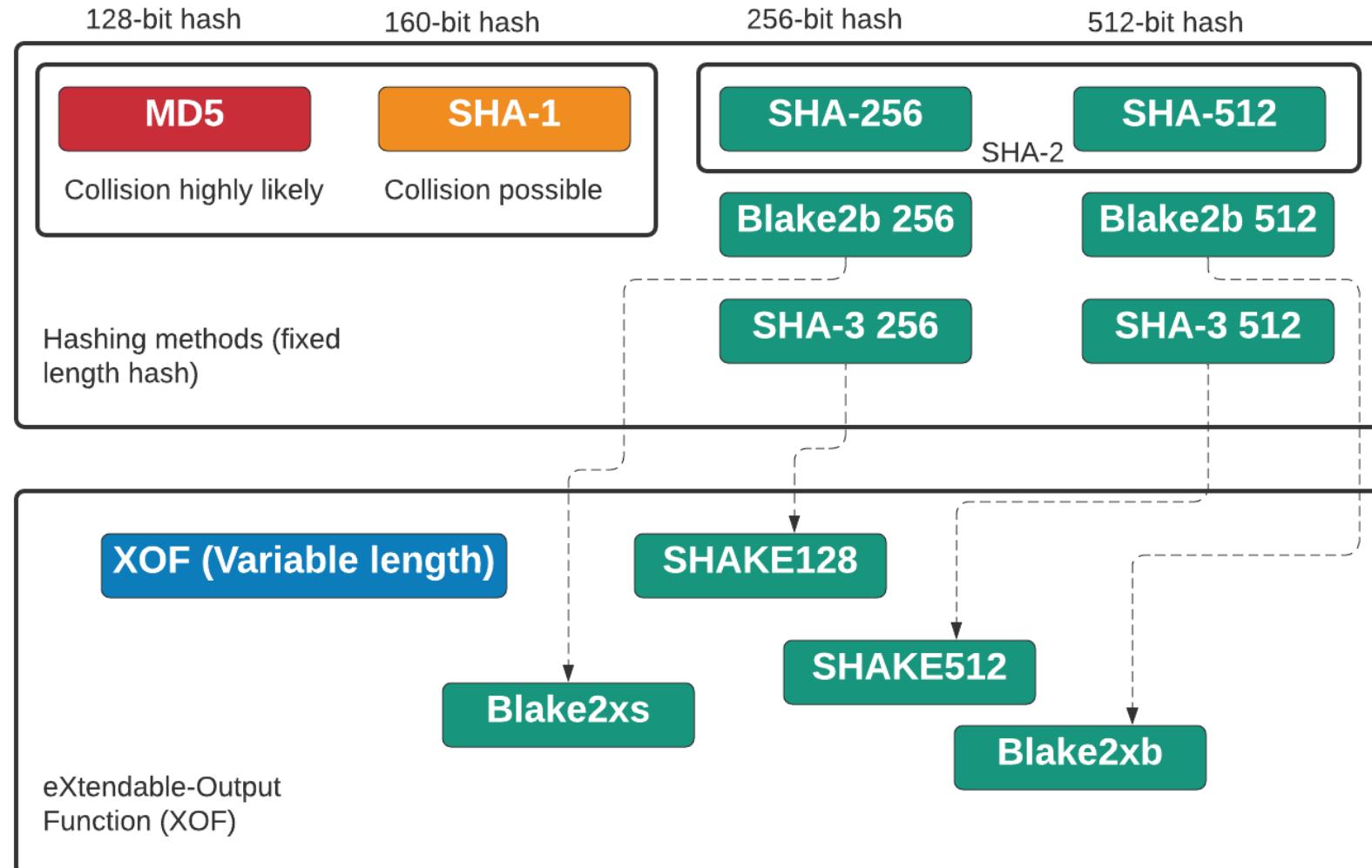
```
=====
```

```
8b1a9953c4611296a827abf8c47804d7
```

# Adding Salt



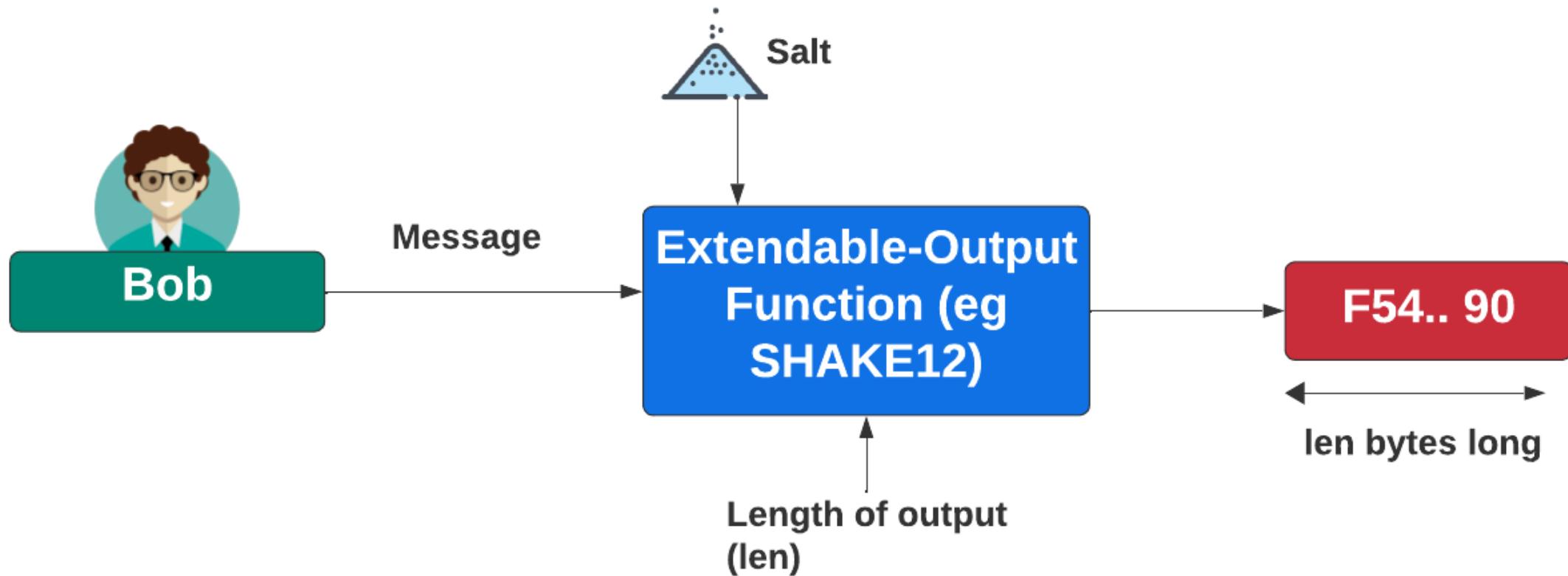
# MD5, SHA-1, SHA-2, SHA-3 and XOF



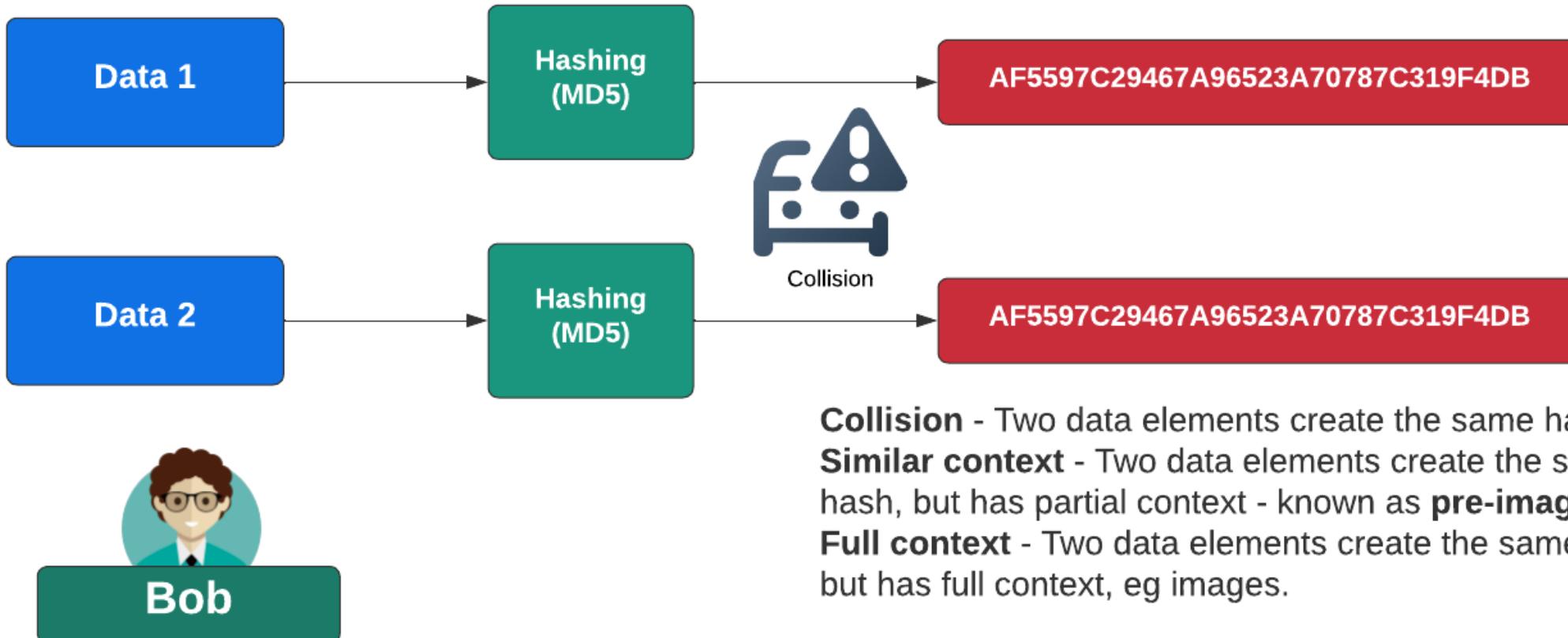
# OpenSSL

The screenshot shows a web browser window with the URL [https://asecuritysite.com/openssl/openssl\\_full2](https://asecuritysite.com/openssl/openssl_full2). The page title is "Hashing Methods (MD5, SHA-1, SHA-256, SHA-512, SHA-3, Blake2b, and so on)". On the left, there is a sidebar with a dropdown menu containing various hashing algorithms: blake2b512 (selected), blake2s256, sha1, sha224, sha256, sha384, sha512, sha3-224, sha3-224, sha3-256, sha3-384, sha3-512, sha512-224, sha512-256, shake128, shake256, and SM3. Below the dropdown is a "Determine" button. The main content area displays the results for the selected algorithm (blake2b512). It shows the Linux command: echo -n "Hello" | openssl dgst -md5, the Windows command: echo | set /p = "Hello" | openssl dgst -md5, the Message: Hello, the Mode: md5, and the resulting hash: MD5d1a7fb5eab1c16cb4f7cf341cf188c3d.

# XOF



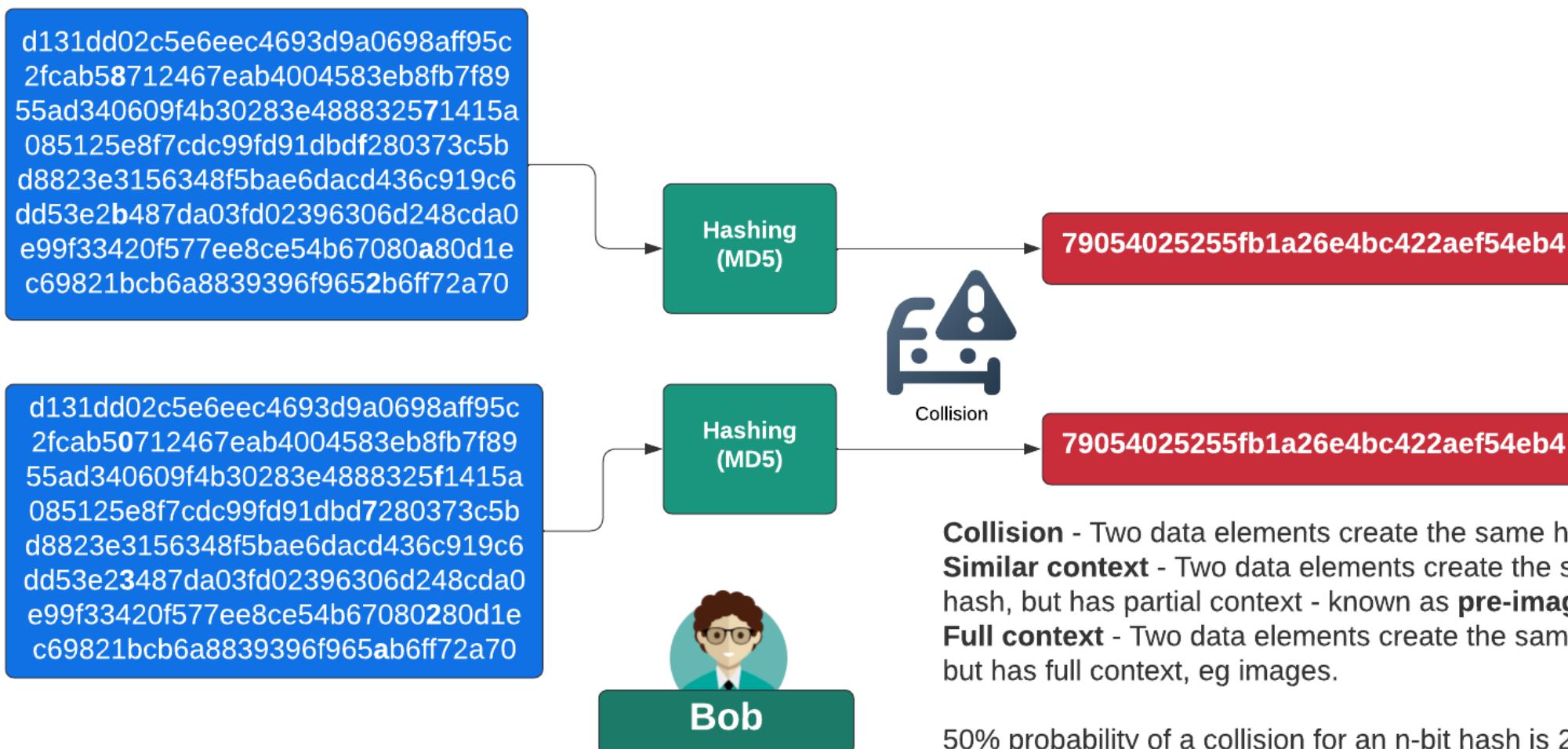
# Hash Collisions



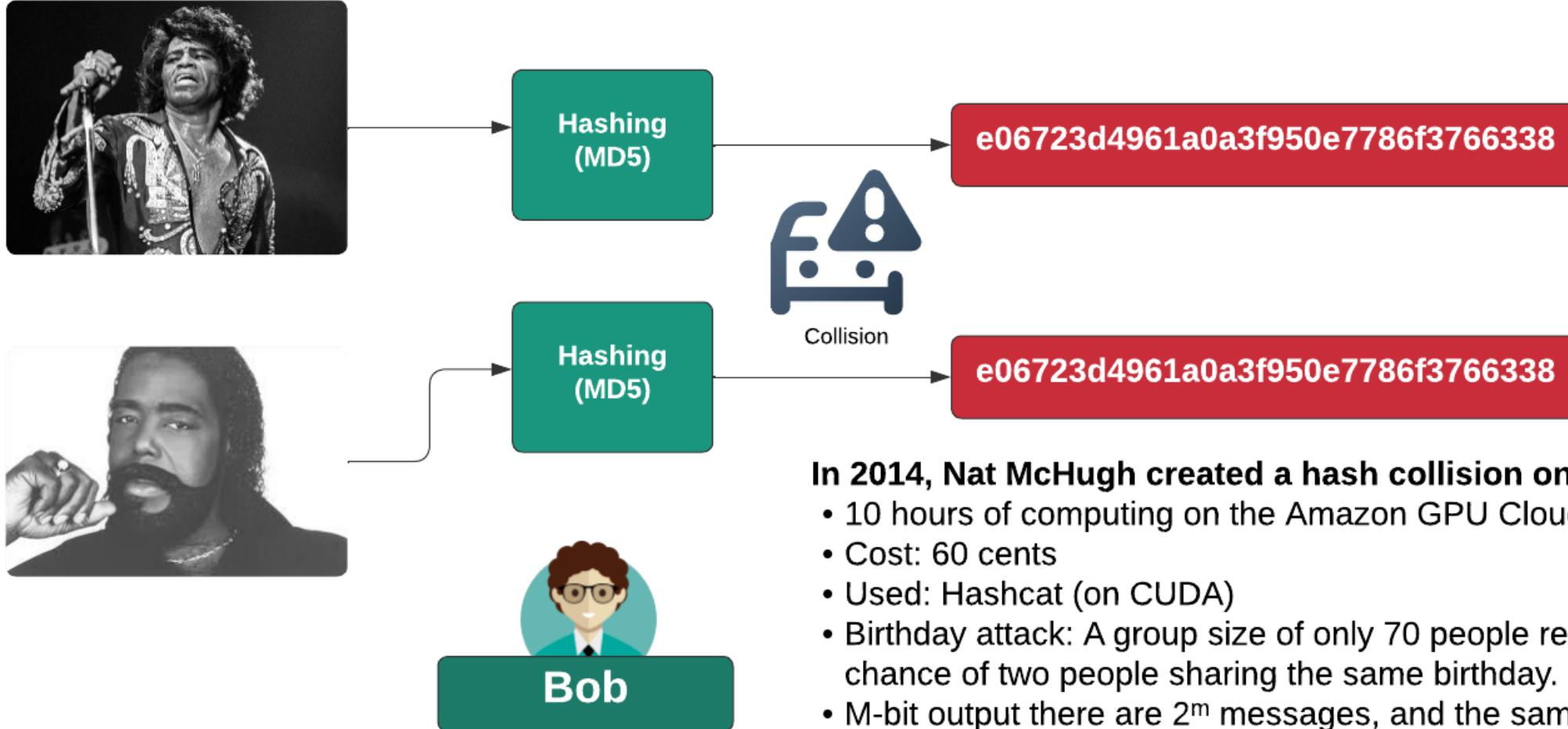
**Collision** - Two data elements create the same hash  
**Similar context** - Two data elements create the same hash, but has partial context - known as **pre-image attack**.  
**Full context** - Two data elements create the same hash, but has full context, eg images.

50% probability of a collision for an n-bit hash is  $2^{n/2}$

# A Real Hash Collision Due To A Flaw



# Hash Collision Using Birthday Attack



In 2014, Nat McHugh created a hash collision on two images

- 10 hours of computing on the Amazon GPU Cloud.
- Cost: 60 cents
- Used: Hashcat (on CUDA)
- Birthday attack: A group size of only 70 people results in a 99.9% chance of two people sharing the same birthday.
- M-bit output there are  $2^m$  messages, and the same hash value would only require  $2^{m/2}$  random messages.

# Hash Benchmarks

Ultra fast:	
Murmur:	545,716 hashes per second
Fast:	
SHA-1:	134,412
SHA-256:	126,323
MD5:	125,741
SHA-512:	76,005
SHA-3 (224-bit):	72,089
Medium speed:	
LDAP (SHA1):	13,718
MS DCC:	9,582
NT Hash:	7,782
MySQL:	7,724
Postgres (MD5):	7,284
Slow:	
PBKDF2 (SHA-256):	5,026
Cisco PIX:	4,402
MS SQL 2000:	4,225
LDAP (MD5):	4,180
Cisco Type 7:	3,775
PBKDF2 (SHA1):	2,348
Ultra-slow:	
LM Hash:	733
APR1:	234
Bcrypt:	103
DES:	88
Oracle 10:	48
Hashes "The quick brown fox jumps over the lazy dog:"	
SHA-1:	2fd4e1c67a2d28fcfd849ee1bb76e7391b93eb12
SHA-256:	d7a8fb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
SHA-512:	07e547d9586f6a73f73fbac0435ed76951218fb7d0c8d788a309d7 85436bbb642e93a252a954f23912547d1e8a3b5ed6e1bfd7097821233fa0538f3db854fee6
MD-5:	9e107d9d372bb6826bd81d3542a419d6
DES:	ZDeS94Lcq/6zg
Bcrypt:	\$2a\$05\$2czCv5GYgkx3aobmEyewB.ejV2hePMdbvTdCyNaSzWtIGPPjB2xx6
APR1:	\$apr1\$ZDzPE45C\$3PvRanPycmNc6c2G9wT9b/
PBKDF2 (SHA1):	\$pbkdf2\$5\$WkR6UEU0NUM\$0RB2bimWrMY.EPYibpaBT2q3HFg
PBKDF2 (SHA-256):	\$pbkdf2-sha256\$5\$WkR6UEU0NUM\$yrJz2oJix7uBJZwZ/50vWUgdE I/iOffqeU4obqC0pk4
LM Hash:	a7b07f9948d8cc7f97c4b0b30cae500f
NT Hash:	4e6a076ae1b04a815fa6332f69e2e231
MS DCC:	efa9778bbc94a7360f664eb7d7144725
LDAP (MD5):	{MD5}9e107d9d372bb6826bd81d3542a419d6
LDAP (SHA1):	{SHA}2fd4e1c67a2d28fcfd849ee1bb76e7391b93eb12
MS SQL 2000:	0x0100BF77CE595DCD1FC87A37B3DEBC27A8C97355CB96B8BAB 63E602662BA5D5D33B913E422499BE72FF3D9BB65DE
MySQL:	*A4E4D26FD0C6455E23E2187C3AABE844332AA1B3
Oracle 10:	4CDA2299FCAD0499
Postgres (MD5):	md5d44c15daa11770f25c5350f7e5408dd1
Cisco PIX:	kGyKN5CqdFQ1qJUs
Cisco Type 7:	15260309443B3E2D2B3875200108010D41505640135E1B0E080 519574156401540035E460B594D1D53020B5C

Benchmark

# Hash Crackers/Bit Coin Miners



## 25 GPU Hash Cracker

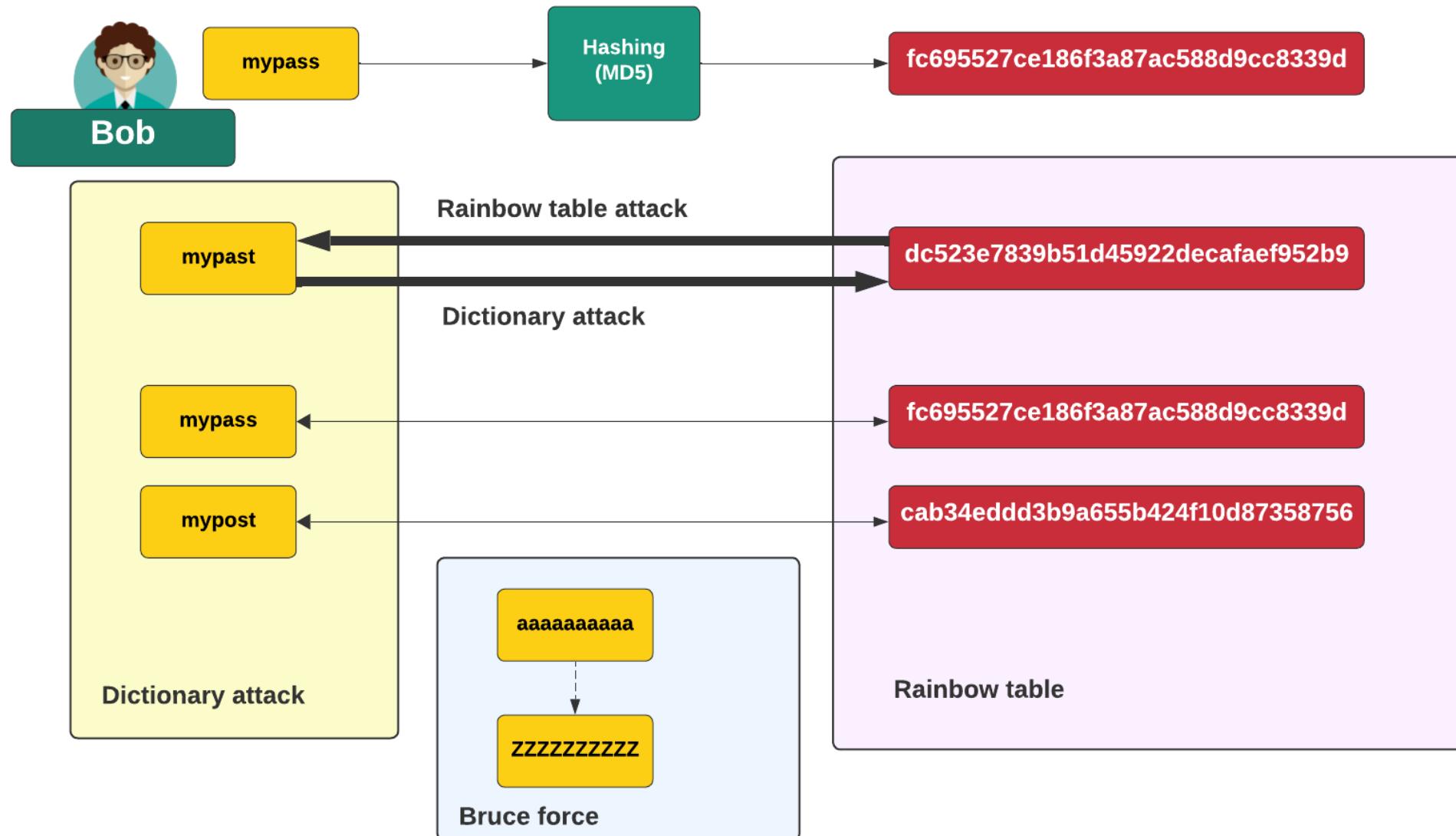
- An eight character NTLM password cracked in 5.5 hours. 14 character LM hash cracked in six minutes. 350 billion hashes per second.

## Fast Hash One

- 1.536TH/s – Cost 3-5,000 dollars.



# Cracking Methods



Bob



Alice



# Cryptography: MAC

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com/mac>

Fundamentals

Symmetric Key

Hashing

**MAC**

KDF

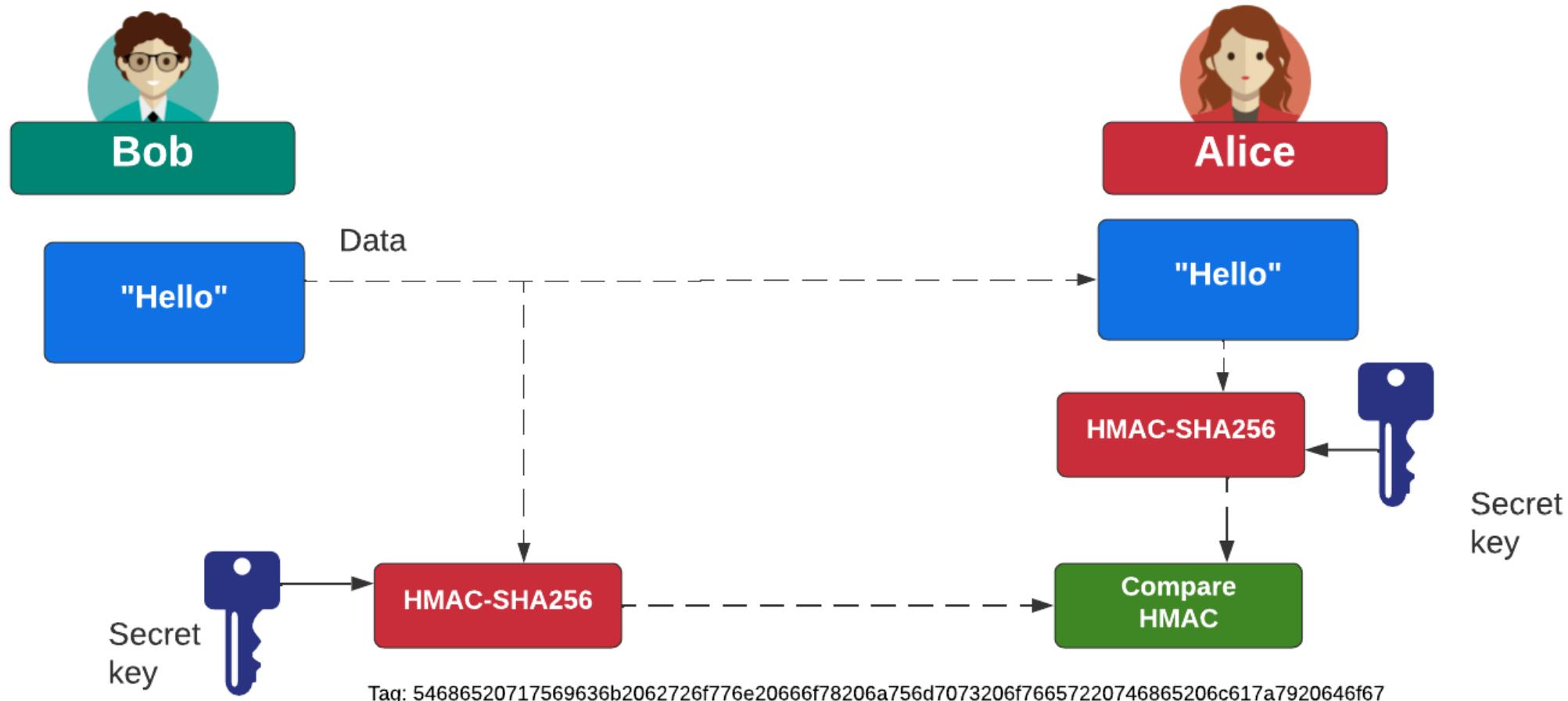
Public Key

Key Exchange

Signatures

Digital Certificates

# HMAC (Key Hash Message Authentication Code)



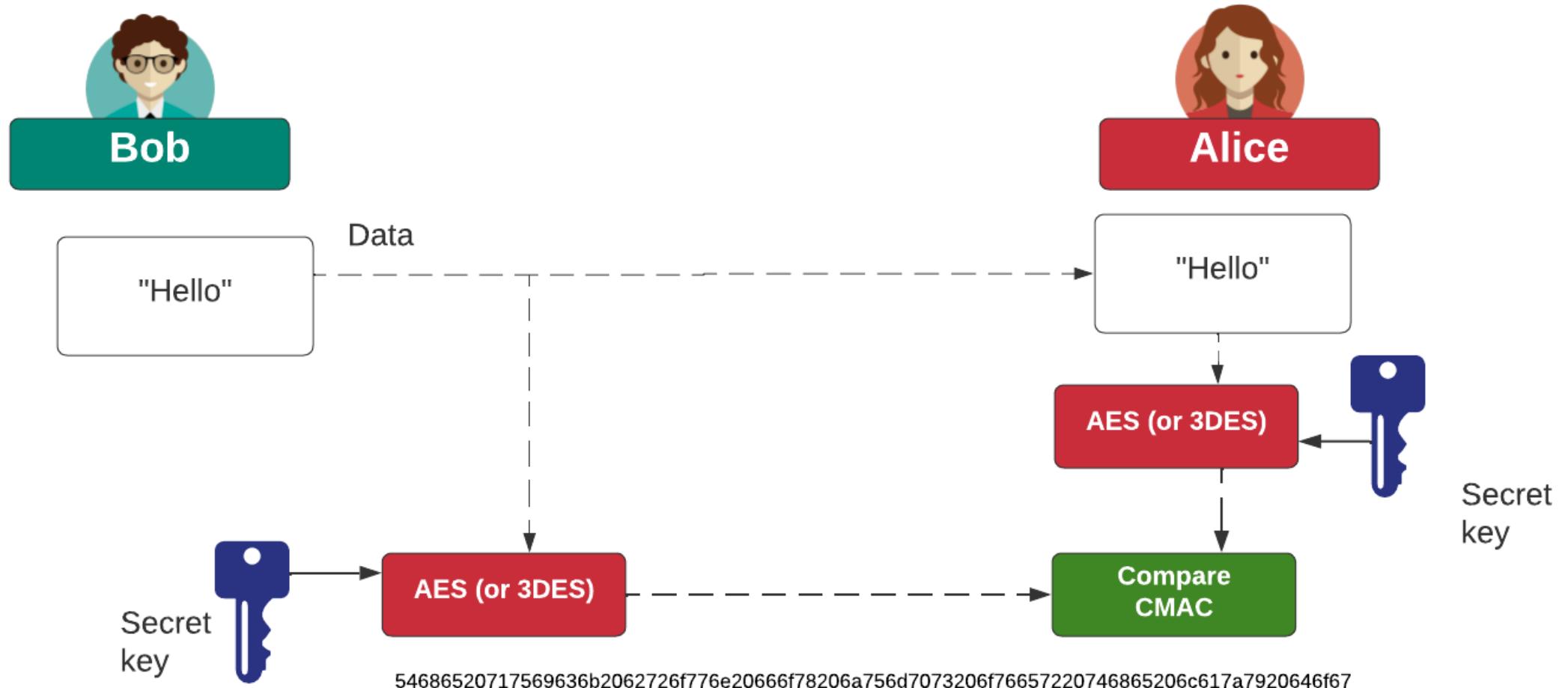
# HMAC (Key Hash Message Authentication Code)

Some AES modes are vulnerable to bit flipping attacks. In this case, we will use AES CTR mode, and flip the bits of the nth character in the ciphertext, and where we specify the character to target (by default, the program will focus on changing the 9th character).

 aka Rijndael  
@asecuritysite.com

Data:  Pay Bob 1 Dollar	Message: Pay Bob 1 Dollar Key: b'333dfcec027543d99aba11bf74262e6763ab65095a0236494a2ef5d527a49b7b' IV: b'7c7833650705f323ee236e33844889e4'  ==== AES CTR ==== Flipping two least significant bits in character 9  Before bitflip: 76368b817f252424d668369378fdae0b After bitflip: 76368b817f252424d568369378fdae0b  Cipher: b'76368b817f252424d568369378fdae0b'
Character to flip:  1st	Determine

# CMAC (Cipher MAC)



# Generating MACs with OpenSSL

[Message Authentication Code (MAC) Home][Home]

In the following we will generate MACs for HMAC, GMAC, CMAC, Blake2bmac, Blake2smac, KMAC128 and KMAC256. Overall, we will use an encryption key defined as a hex value for GMAC and CMAC, and a pass phrase for the others. The message is "hello".



## Parameters

Message:  
hello

Password (for Blake2b/2s  
and KMAC): Message:  
test

Key (used for CMAC/GMAC):  
77A77FAF290C1FA30C  
683DF16BA7A77B

Cipher (for CMAC and  
GMAC):  
AES-128-CBC

Digest (for HMAC):  
sha256

Generate:

```
Message: hello
Password: test
Key: 77A77FAF290C1FA30C683DF16BA7A77B
Cipher: AES-128-CBC
=====
HMAC:
"echo set /p="hello" | openssl dgst -sha256 -hmac test"
f151ea24bda91a18e89b8bb5793ef324b2a02133cce15a28a719acbd2e58
a986
=====
CMAC:
"echo | set /p="hello" | openssl mac -cipher AES-128-CBC -macopt hexkey:77A77FAF290C1FA30C683DF16BA7A77B CMAC"
8CDDB6E42ED45C8A98328ADC739D3E41
=====
GMAC:
"echo | set /p="hello" | openssl mac -cipher AES-128-CBC -macopt hexkey:77A77FAF290C1FA30C683DF16BA7A77B GMAC"
=====
Blake2bmac:
"echo | set /p="hello" | openssl mac -macopt key:test BLAKE2BMAC"
64DB361C140C4A9987AC25A34BF37E5E95ADCA360E2EFF628D9814E295F48403DC5A7FE9B545BD6A3BB7F09DEF1E694D624B8A22922B3CBF295BC63
E6A7CA947
=====
Blake2smac:
"echo | set /p="hello" | openssl mac -macopt key:test BLAKE2SMAC"
4C7961E3C366C776FA23FB9CEF1BA11DDAF310BF403F18F8D9A0C14BF2471D9E
=====
KMAC128:
"echo | set /p="hello" | openssl mac -macopt key:test KMAC128"
A816D9D438C572D382F7AA6EF7529B1C44FFAF8FE528C37EC6CF267F4B195B08
=====
KMAC256:
```

Bob



Alice



# Cryptography: KDF

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com/kdf>

Fundamentals

Symmetric Key

Hashing

MAC

KDF

Public Key

Key Exchange

Signatures

Digital Certificates

# Keys Generated By Passwords With Brute Force

**Characters in password:**

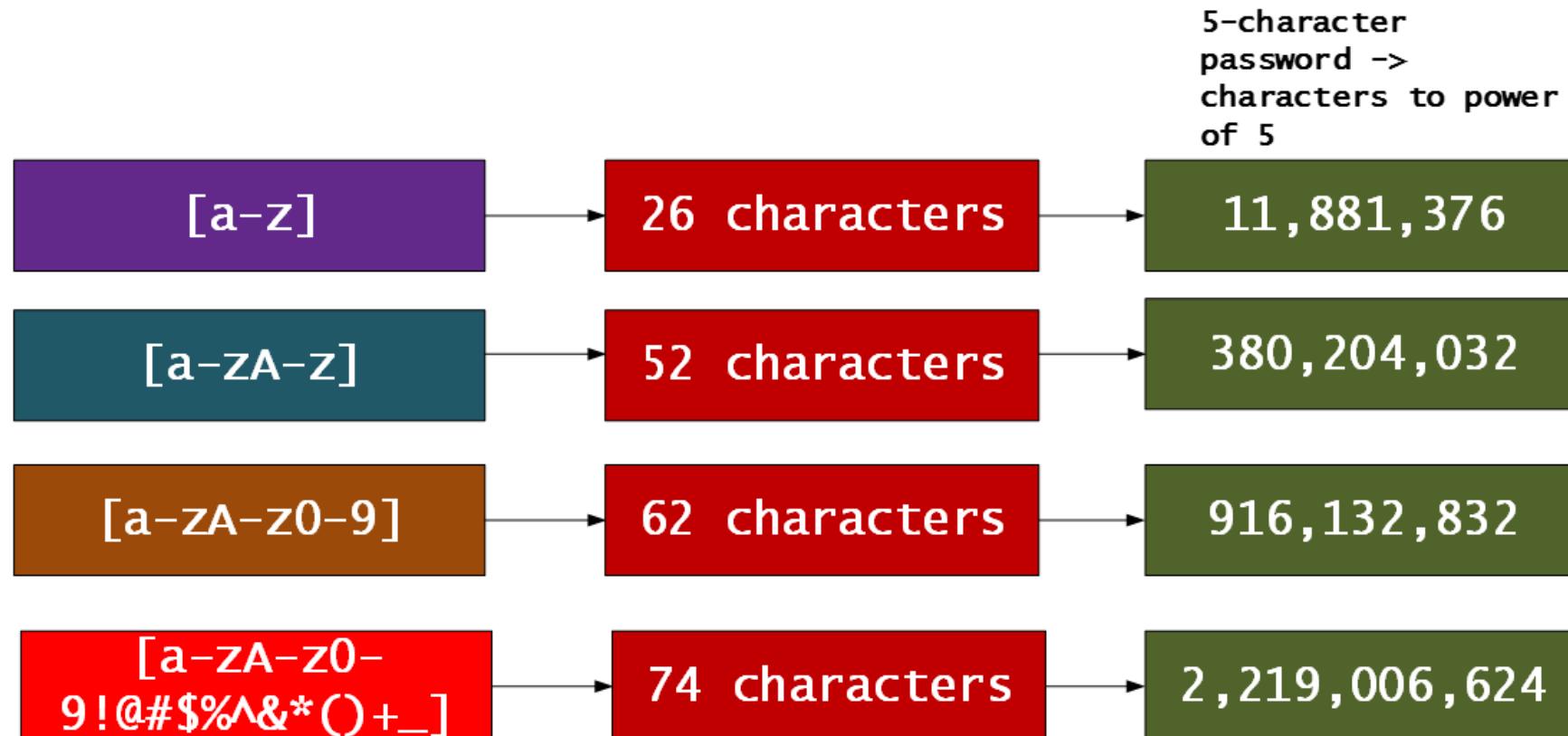
Characters [a-z]:   
Characters [A-Z]:   
Characters [0-9]:   
Characters [!@#\$%^&\*()\_]:   
Characters [~`-\_+=[]{}\\|;:\\",.<>?/]:

**Password cracking speed:** 1 Million per second **Processing elements:** 1

**Results**

No of characters:	52	Time to crack (max)	Key entropy (bits)
No of passwords (5 digits):	380,204,032	6.34 mins	28.5
No of passwords (6 digits):	19,770,609,664	5.49 hours	34.2
No of passwords (7 digits):	1,028,071,702,528	11.90 days	39.9
No of passwords (8 digits):	53,459,728,531,456	1.69 years	45.6
No of passwords (9 digits):	2,779,905,883,635,712	88.09 years	51.3
No of passwords (10 digits):	144,555,105,949,057,024	4580.78 years	57
No of passwords (11 digits):	7,516,865,509,350,965,248	238200.38 years	62.7
No of passwords (12 digits):	390,877,006,486,250,192,896	12386419.66 years	68.41

# Keys Generated By Passwords With Brute Force



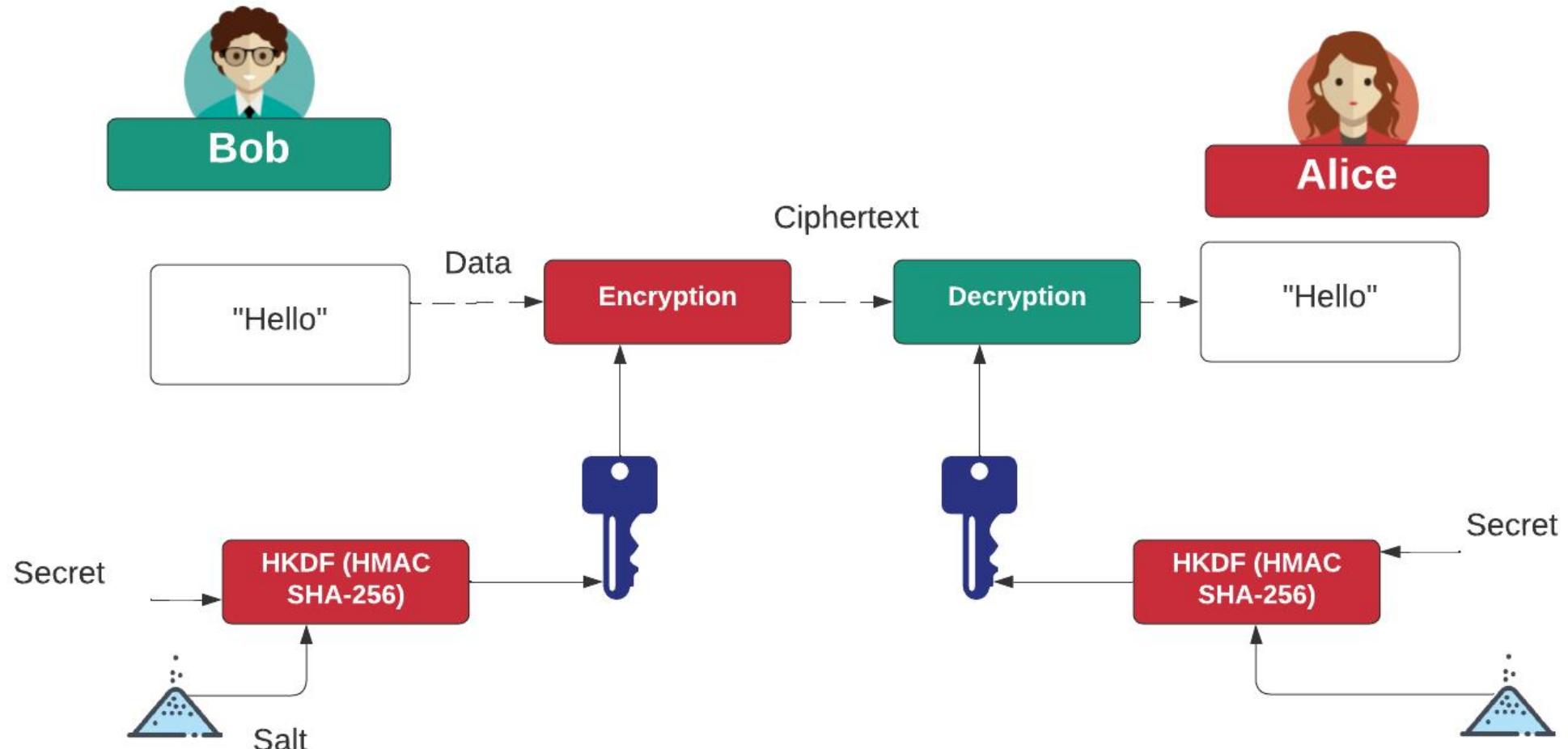
freddy

frEdY

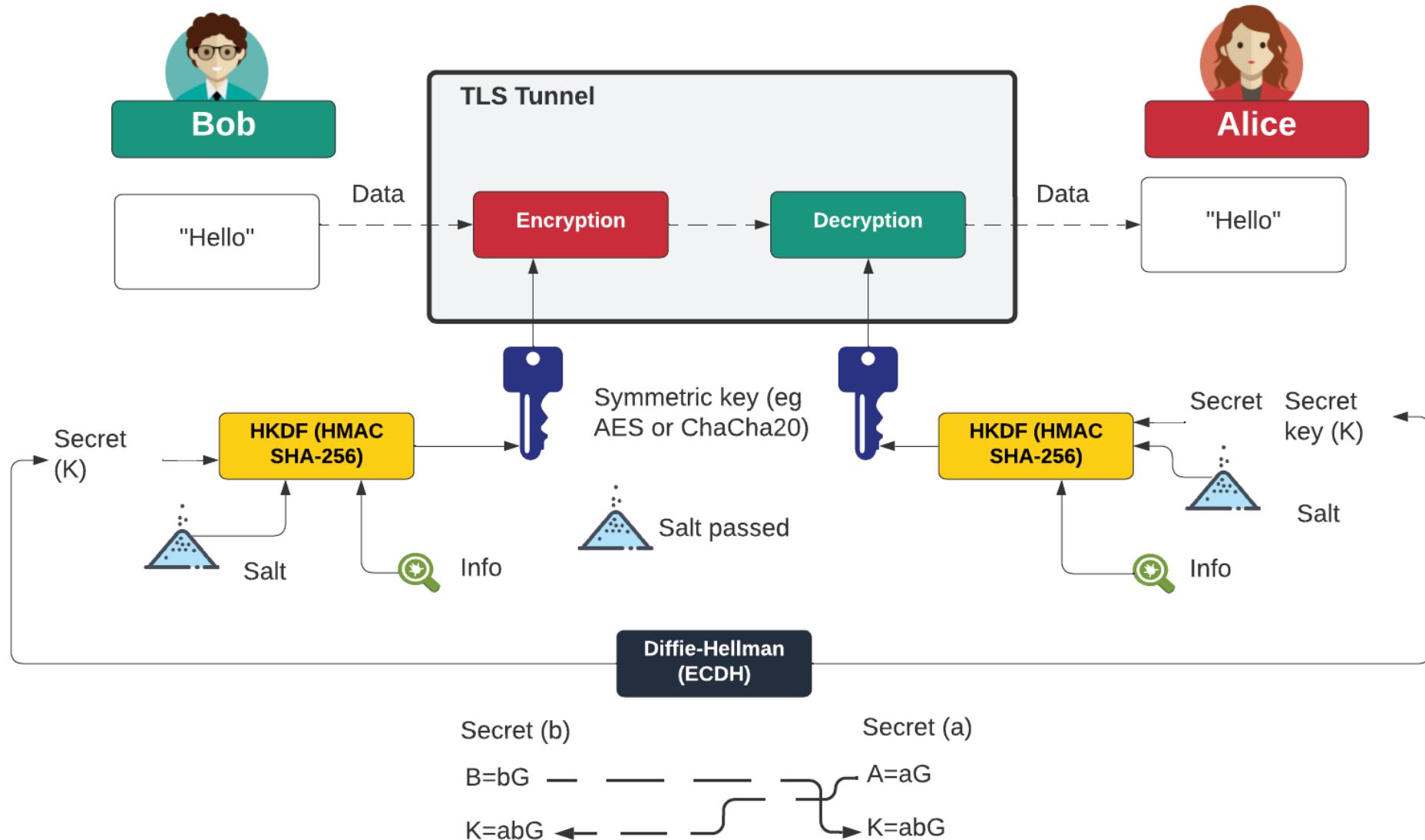
fr3dY

fr3d!

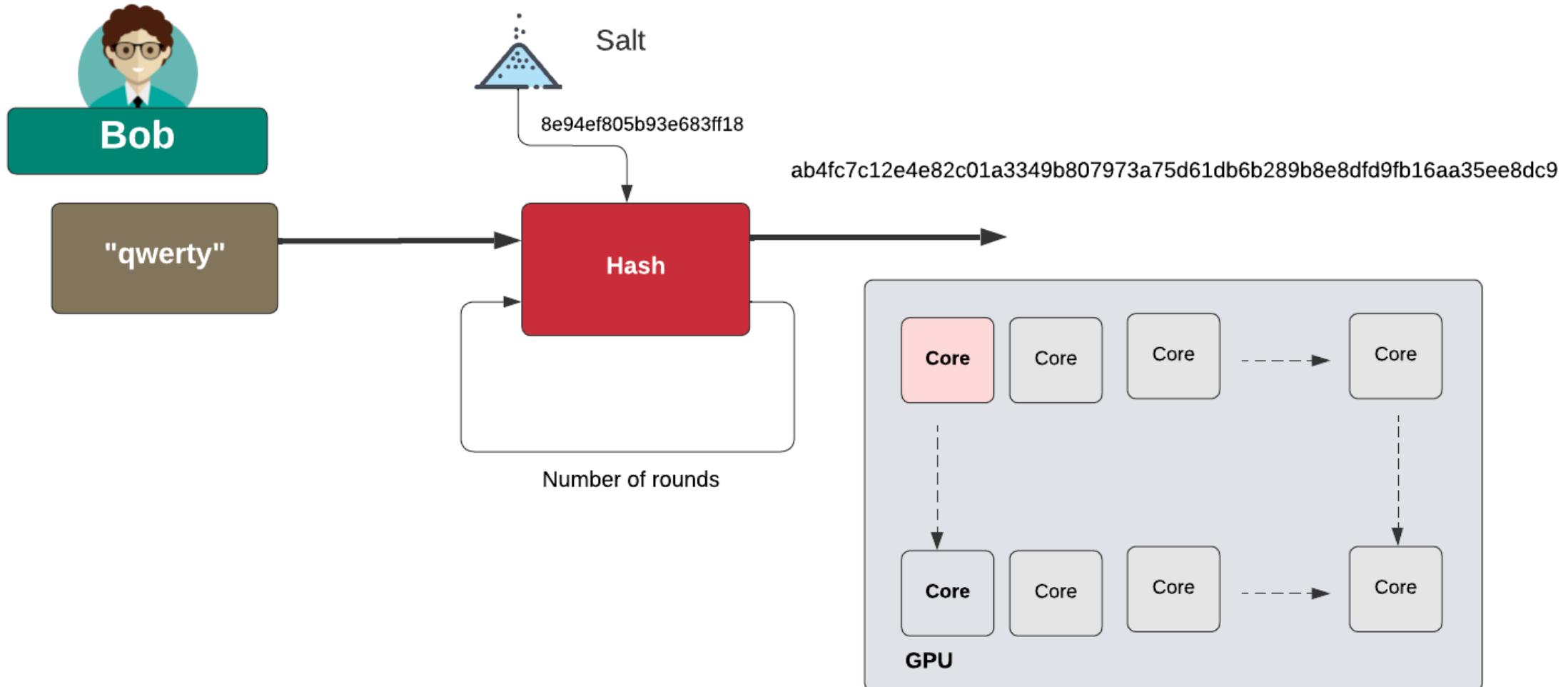
# HKDF (HMAC Key Derivation Function)



# TLS Setup of Symmetric Key

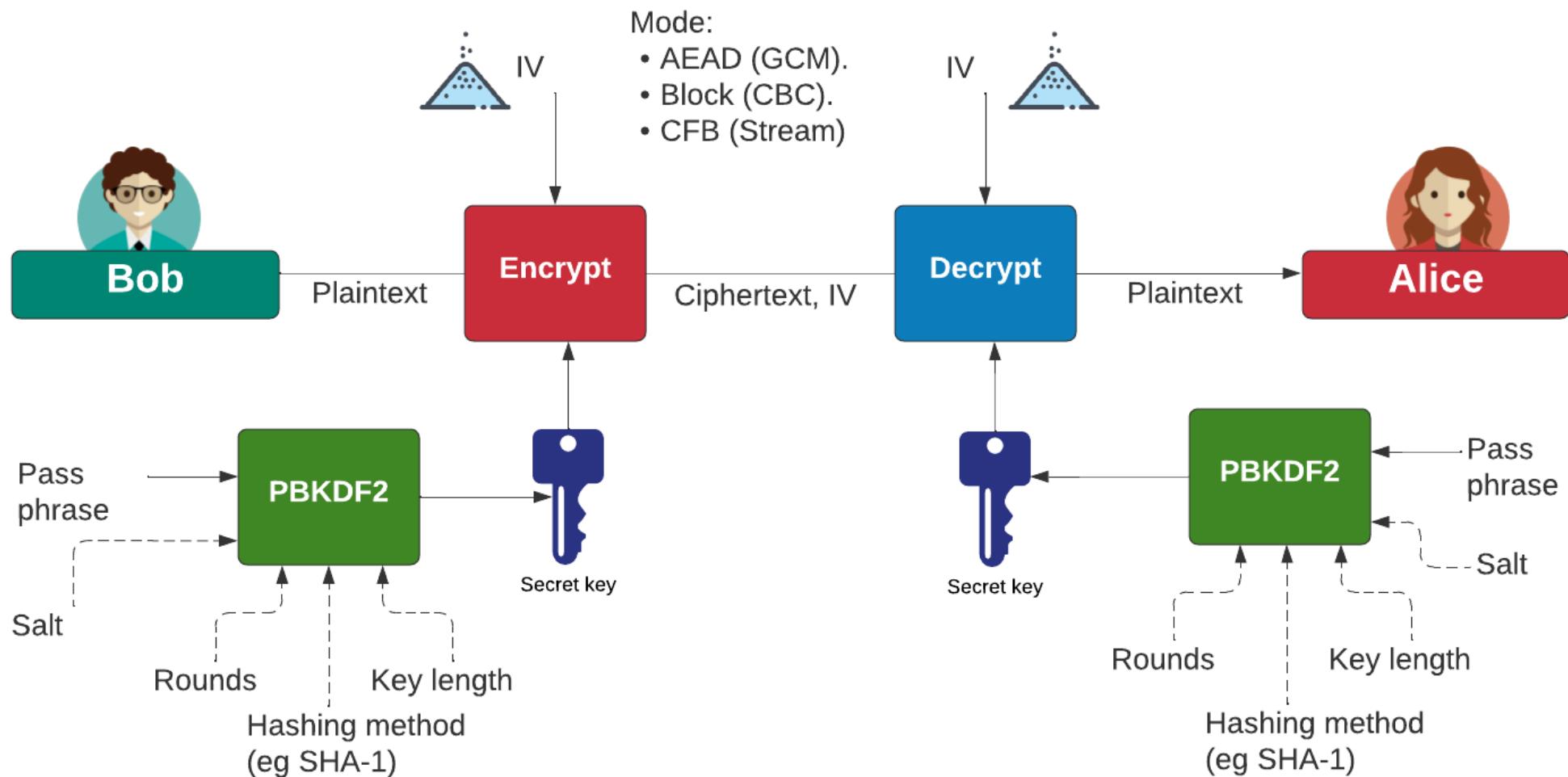


# Slowing down hashing

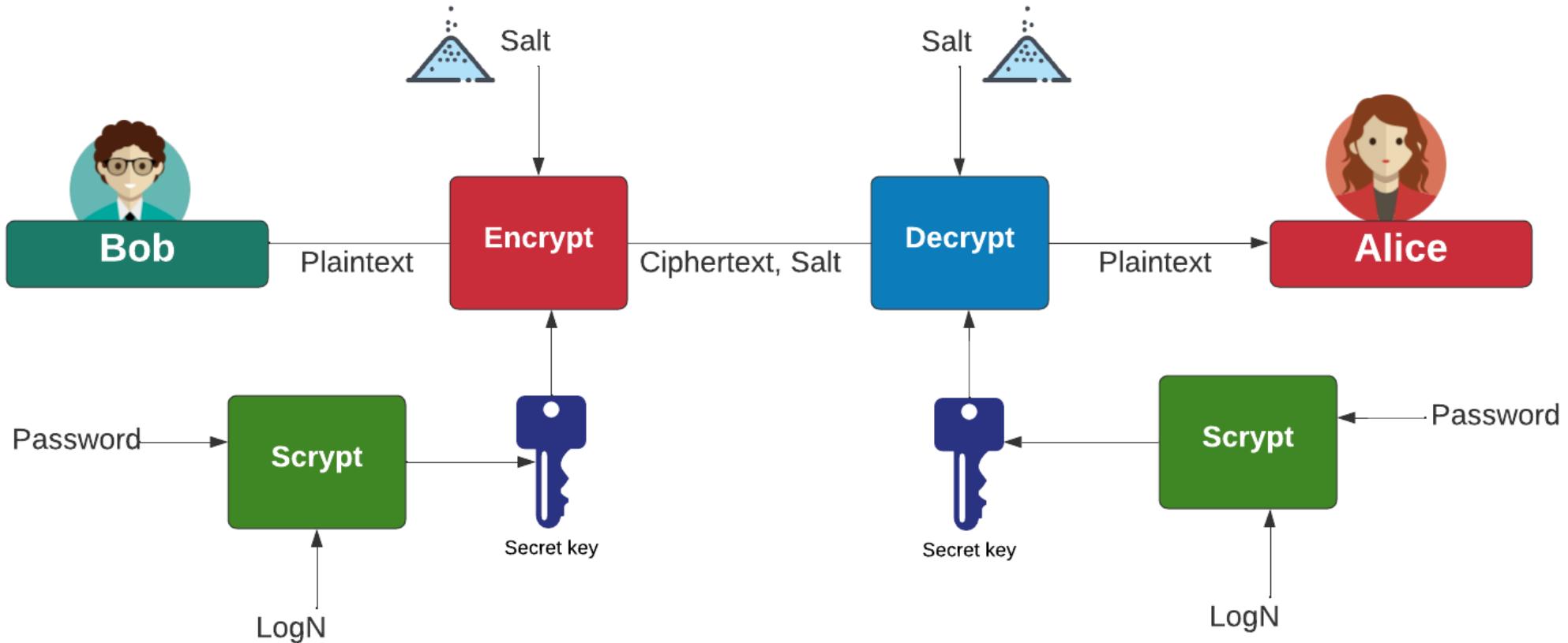


<https://asecuritysite.com/hash/htest>

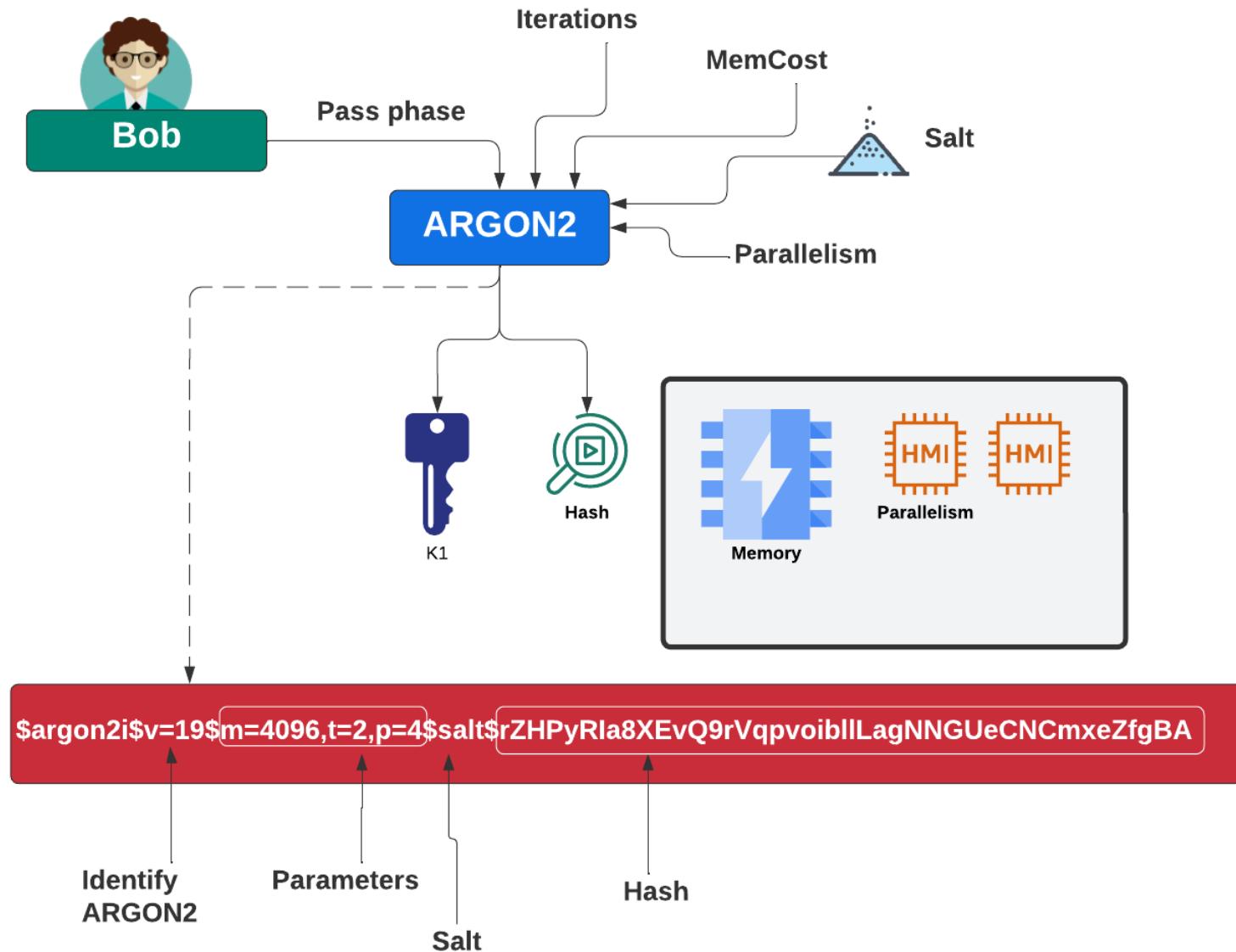
# PBKDF2



# scrypt



# Argon2



# Argon2

Argon2			Argon2			Argon2					
Cost		Time	Cost		Time	Cost		Time			
n	log_2(m)	p	sec.	n	log_2(m)	p	sec.	n	log_2(m)	p	sec.
1	8	1	0.003	1	8	2	0.002	1	8	4	0.002
2	8	1	0.003	2	8	2	0.003	2	8	4	0.002
4	8	1	0.005	4	8	2	0.004	4	8	4	0.002
8	8	1	0.008	8	8	2	0.006	8	8	4	0.003
16	8	1	0.015	16	8	2	0.011	16	8	4	0.005
32	8	1	0.029	32	8	2	0.016	32	8	4	0.008
64	8	1	0.051	64	8	2	0.027	64	8	4	0.015
128	8	1	0.105	128	8	2	0.044	128	8	4	0.029
256	8	1	0.151	256	8	2	0.079	256	8	4	0.057
512	8	1	0.362	512	8	2	0.143	512	8	4	0.117
1024	8	1	0.631	1024	8	2	0.327	1024	8	4	0.237
2048	8	1	1.397	2048	8	2	0.595	2048	8	4	0.474
4096	8	1	2.721	4096	8	2	1.291	4096	8	4	0.946
8192	8	1	5.787	8192	8	2	2.395	8192	8	4	1.918
16384	8	1	11.575	16384	8	2	4.859	16384	8	4	3.777
32768	8	1	24.095	32768	8	2	9.796	32768	8	4	7.591

The parameters include:

- Password (P): Defines the password bytes to be hashed
- Salt (S): Defines the bytes to be used for salting.
- Parallelism (p): Defines the number of thread that are required for the parallelism.
- TagLength (T): Define the number of bytes to return for the hash.
- MemorySizeKB (m): Amount of memory (in KB) to use.

A sample run is:

Message: abc Hash:

\$argon2i\$v=19\$m=8,t=1,p=1\$x/3yHil0MIE\$RDL1Jw

# OWASP

Use **Argon2id** with a minimum configuration of 19 MiB of memory, an iteration count of 2, and 1 degree of parallelism.

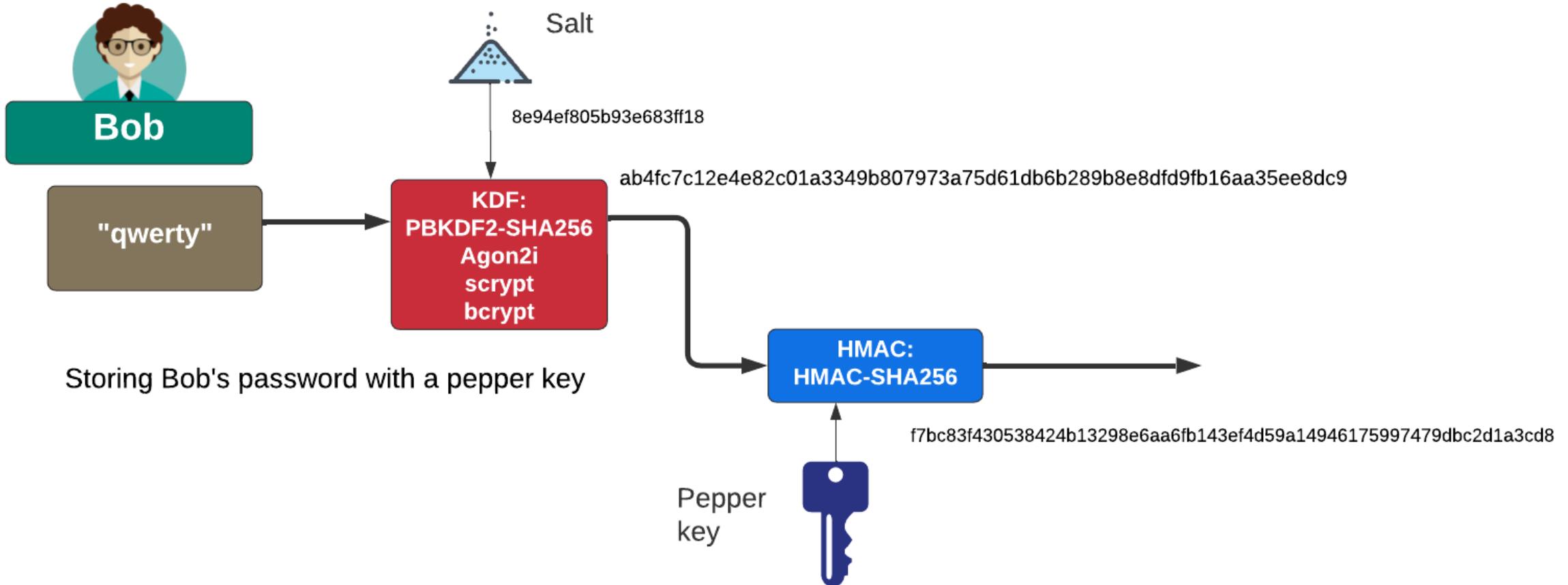
If **Argon2id** is not available, use **scrypt** with a minimum CPU/memory cost parameter of ( $2^{17}$ ), a minimum block size of 8 (1024 bytes), and a parallelization parameter of 1.

For legacy systems using **bcrypt**, use a work factor of 10 or more and with a password limit of 72 bytes.

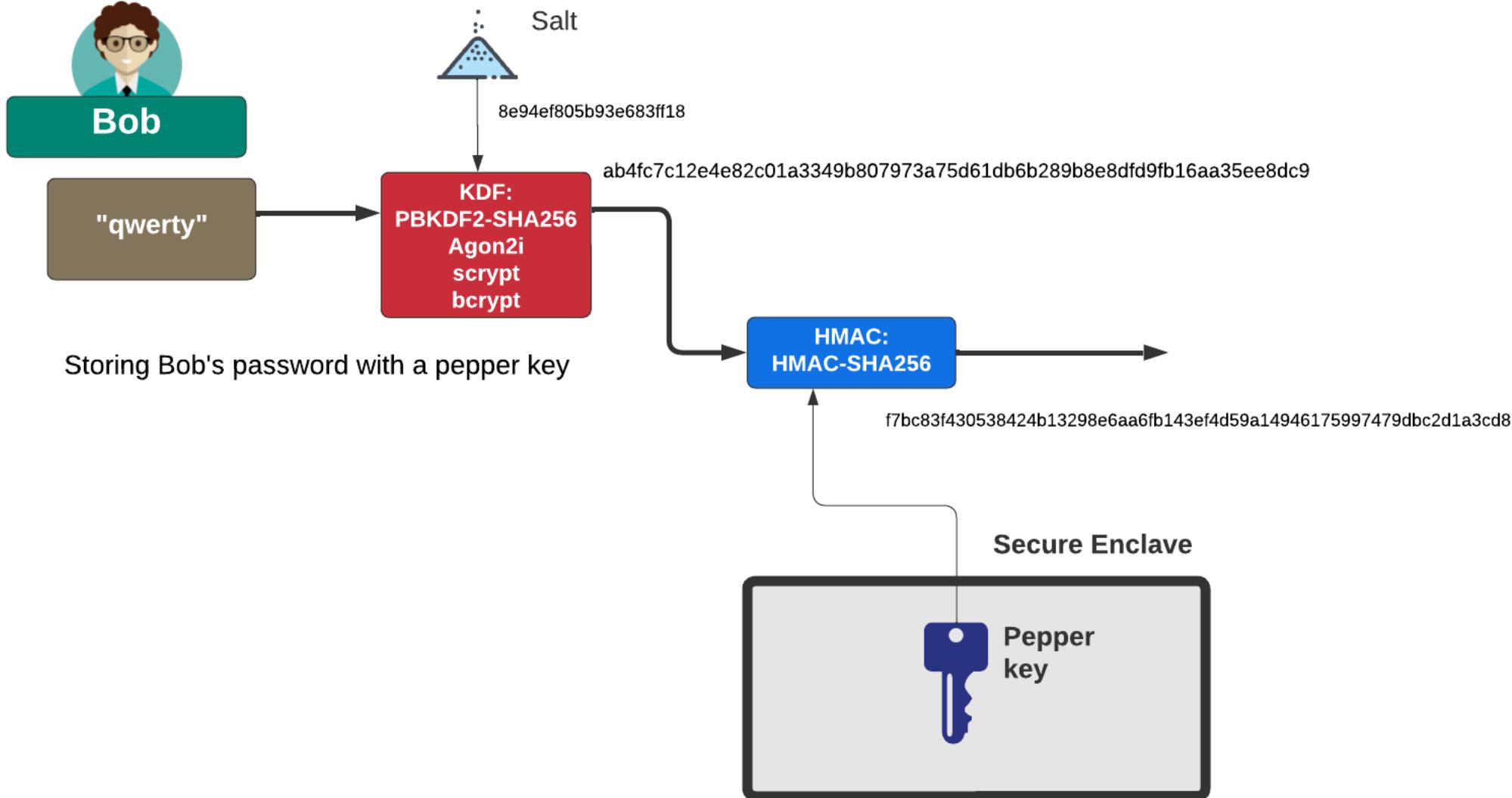
If FIPS-140 compliance is required, use **PBKDF2** with a work factor of 600,000 or more and set with an internal hash function of HMAC-SHA-256.

Consider using a **pepper** to provide additional defense in depth (though alone, it provides no additional secure characteristics).

# Using a Pepper Key



# Using a Pepper Key



Bob



Alice



# Cryptography: Public Key

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com/rsa>

<https://asecuritysite.com/ecc>

Fundamentals

Symmetric Key

Hashing

MAC

KDF

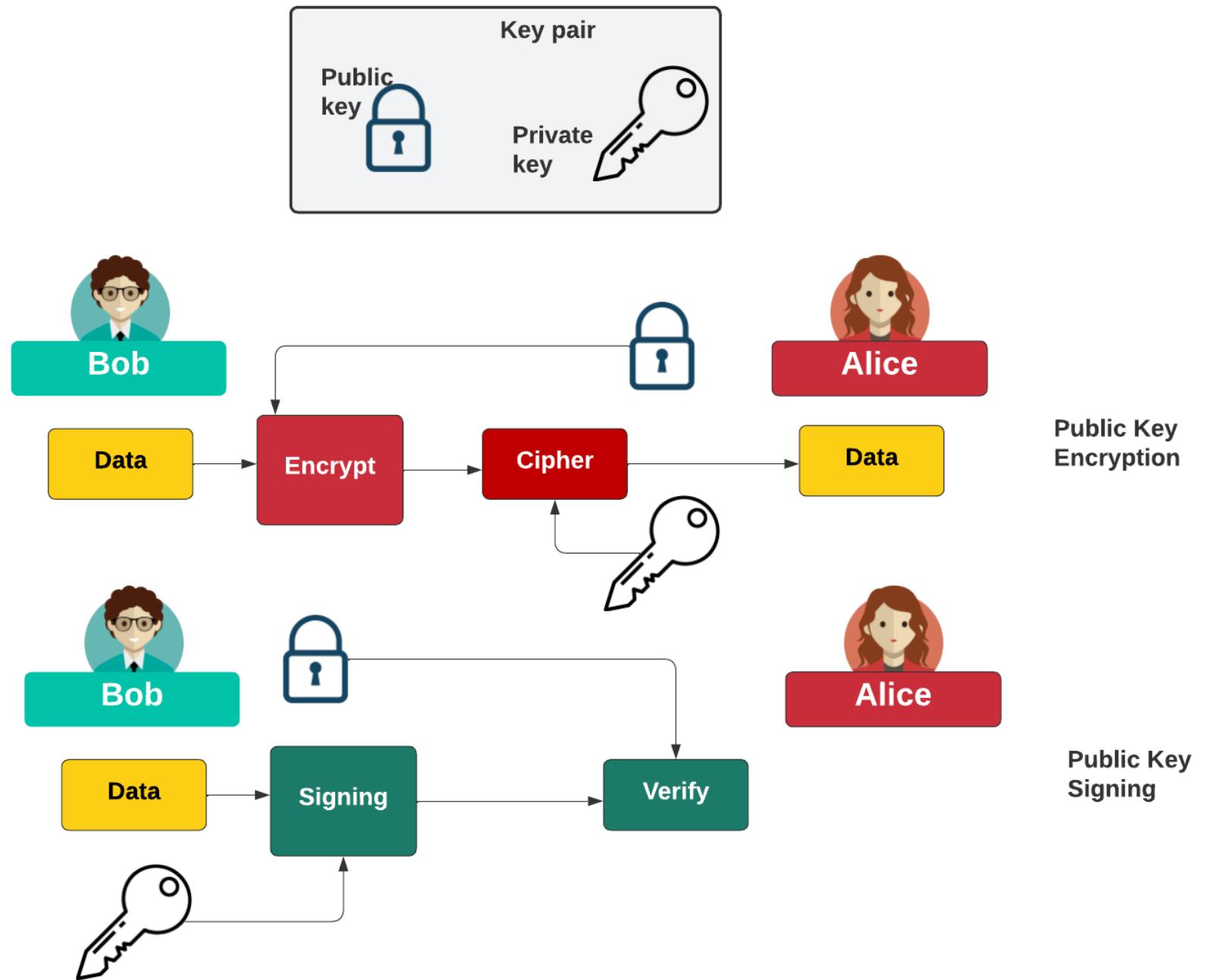
**Public Key**

Key Exchange

Signatures

Digital Certificates

# Public Key



# Public Key

## MATHEMATICAL GAMES

*A new kind of cipher that would take millions of years to break*

by Martin Gardner

Few persons can be made to believe that it is not quite an easy thing to invent a method of secret writing which shall baffle investigation. Yet it may be roundly asserted that human ingenuity cannot concoct a cipher which human ingenuity cannot resolve.

—EDGAR ALLAN POE

The upward creep of postal rates accompanied by the deterioration of postal service is a trend that may or may not continue, but as far as most private communication is concerned, in a few decades it probably will not matter. The reason is simple. The transfer of information will probably be much faster and much cheaper by "electronic mail" than by conventional postal systems. Before long it should be possible to go to any telephone, insert a message into an attachment and dial a number. The telephone at the other end will print out the message at once.

Government agencies and large businesses will presumably be the first to make extensive use of electronic mail, followed by small businesses and private individuals. When this starts to happen, it will become increasingly desirable to have fast, efficient ciphers to safeguard information from electronic eavesdroppers. A similar problem is involved in protecting private information stored in computer memory banks from snoops who have access to the memory through data-processing networks.

It is hardly surprising that in recent years a number of mathematicians have asked themselves: Is it possible to devise a cipher that can be rapidly encoded and decoded by computer, can be used repeatedly without changing the key and

is unbreakable by sophisticated cryptanalysts? The surprising answer is yes. The breakthrough is scarcely two years old, yet it bids fair to revolutionize the entire field of secret communication. In deed, it is so revolutionary that all previous ciphers, together with the techniques for cracking them, may soon fade into oblivion.

An unbreakable code can be unbreakable in theory or unbreakable only in practice. Edgar Allan Poe, who fancied himself a skilled cryptanalyst, was convinced that no cipher could be invented that could not be "unriddled." Poe was certainly wrong. Ciphers that are unbreakable even in theory have been in use for half a century. They are "one-time pads," ciphers that are used only once, for a single message. Here is a simple example based on a shift cipher sometimes called a Caesar cipher because Julius Caesar used it.

First write the alphabet, followed by the digits 0 through 9. (For coding purposes 0 represents a space between words, and the other digits are assigned to punctuation marks.) Below this write the same sequence cyclically shifted to the right by an arbitrary number of units, as is shown in color in the illustration on this page. Our cipher consists in taking each symbol in the plaintext (the message), finding it in the top row, and replacing it with the symbol directly below it. The result is a simple substitution cipher, easily broken by any amateur.

If the one-time pad provides absolute security, why is it not used for all secret communication? The answer is that it is too impractical. Each time it is employed a key must be sent in advance, and the key must be at least as long as the anticipated message. "The problem of producing, registering, distributing and canceling the keys," writes Kahn, "may seem slight to an individual who has not had experience with military communications, but in wartime the volumes of traffic stagger even the signal staffs. Hundreds of thousands of words may be enciphered in a day; simply to generate the millions of key characters required would be enormously expensive and time-consuming. Since each

encoded, the arrow is spun and the lower sequence is shifted accordingly. The result is a ciphertext starting with *i* and a cipher "key" starting with *k*. Note that the cipher key will be the same length as the plaintext.

To use this one-time cipher for sending a message to someone—call him *Z*—we must first send *Z* the key. This can be done by a trusted carrier. Later, *Z* sends to *Z*, perhaps by radio, the ciphertext. *Z* decodes it with the key and then destroys the key. The key must not be saved again because if two such ciphertexts were intercepted, a cryptanalyst might have sufficient structure for breaking them.

It is easy to see why the one-time cipher is unbreakable even in principle. Since each symbol can be represented by any other symbol, and each choice of representation is completely random, there is no internal pattern. To put it another way, any message whatever having the same length as the ciphertext is as legitimate a decoding as any other. Even if the plaintext of such a coded message is found, it is of no future help to the cryptanalyst because the next time the system is used the randomly chosen key will be entirely different.

One-time pads are in constant use today for special messages between high military commanders, and between governments and their high-ranking agents. The "pad" is no more than a long list of random numbers, perhaps printed on many pages. The sender and receiver must of course have duplicate copies. The sender uses page 1 for a cipher, then destroys the page. The receiver uses his page 1 for decoding, then destroys his page. When the Russian agent Rudolf Abel was captured in New York in 1957, he had a one-time pad in the form of a booklet about the size of a postage stamp. David Kahn, who tells the story in his marvelous history *The Codebreakers*, says that the one-time pad is the standard method of secret radio communication used by the U.S.S.R. The famous "hot line" between Washington and Moscow also makes use of a one-time pad, the keys being periodically delivered through the two embassies.

If the one-time pad provides absolute security, why is it not used for all secret communication? The answer is that it is too impractical. Each time it is employed a key must be sent in advance, and the key must be at least as long as the anticipated message. "The problem of producing, registering, distributing and canceling the keys," writes Kahn, "may seem slight to an individual who has not had experience with military communications, but in wartime the volumes of traffic stagger even the signal staffs. Hundreds of thousands of words may be enciphered in a day; simply to generate the millions of key characters required would be enormously expensive and time-consuming. Since each

message must have its unique key, application of the ideal system would require shipping out on tape at the very least the equivalent of the total communications volume of a war."

Let us qualify Poe's dictum by applying it only to ciphers that are used repeatedly without any change in the key. Until recently all cipher systems of this kind were known to be theoretically breakable provided the code breaker has enough time and enough ciphertext. Then in 1975 a new kind of cipher was proposed that radically altered the situation by supplying a new definition of "unbreakable," a definition that comes from the branch of computer science known as complexity theory. These new ciphers are not absolutely unbreakable in the sense of the one-time pad, but in practice they are unbreakable in a much stronger sense than any cipher previously designed for widespread use. In principle these new ciphers can be broken, but only by computer programs that run for millions of years!

The two men responsible for this remarkable breakthrough are Whitfield Diffie and Martin E. Hellman, both electrical engineers at Stanford University. Their work was partly supported by the National Science Foundation in 1973 and was reported in their paper "New Directions in Cryptography" (*IEEE Transactions on Information Theory*, November, 1976). In it Diffie and Hellman show how to create unbreakable ciphers that do not require advance sending of a key or even concealment of the method of encoding. The ciphers can be efficiently encoded and decoded, they can be used over and over again and there is a bonus: the user also provides an "electronic signature" that, unlike a written signature, cannot be forged. If *Z* receives a "signed" message from *A*, the signature proves to *Z* that *A* actually sent the message. Moreover, *A*'s signature cannot be forged by an eavesdropper or even by *Z* himself.

These seemingly impossible feats are made possible by what Diffie and Hellman call a trapdoor one-way function. Such a function has the following properties: (1) it will change any positive integer *x* to a unique positive integer *y*; (2) it has an inverse function that changes *y* back to *x*; (3) efficient algorithms exist for computing both the forward function and its inverse; (4) if only the function and its forward algorithm are known, it is computationally infeasible to discover the inverse algorithm.

The last property is the curious one that gives the function its name. It is like a trapdoor: easy to drop through but hard to get up through. Indeed, it is impossible to get up through the door unless one knows where the secret button is hidden. The button symbolizes the "trapdoor information." Without it one cannot open the door from below, but the button is so carefully concealed that



the probability of finding it is practically zero.

Before giving a specific example, let us see how such functions make the new cryptographic systems possible. Suppose there is a group of businessmen who want to communicate secrets to one another. Each devises his own trapdoor function with its forward and backward algorithms. A handbook is published in which each company's encoding (forward) algorithm is given in full. The decoding (inverse) algorithms are kept secret. The handbook is public. Anyone can consult it and use it for sending a secret message to any listed company.

Suppose you are not a member of the group but you want to send a secret message to member *Z*. First you change your plaintext to a long number, using a standard procedure given in the handbook. Next you look up *Z*'s forward algorithm and your computer uses it for rapid encoding of the plaintext. This new number is sent to *Z*. It does not matter at all if the ciphertext is overheard or intercepted because only *Z* knows his secret decoding procedure.

There is no way a curious cryptanalyst, studying *Z*'s public encoding algorithm, can discover *Z*'s decoding algorithm. In principle he might find it, but in practice that would require a supercomputer and a few million years of running time.

An outsider cannot "sign" a message to *Z* but any member of the group can.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	

A Caesar cipher with a 10-shift

## I. Introduction

L. Rivest<sup>\*</sup>  
MIT  
Public  
Adleman  
science

## ing Public- with the novel option key ing decryption not needed to miphoned ed by the r the message, decryption key, privately held signature using cryptograph key, or cannot later has obvious electronic funds ted by to a publicly public remainder ily specified umber p and q, secret, power d (q - 1). The difficulty of

- atures, public  
on, security,  
mail, message  
lography,  
1, 5-25
- In a "public-key cryptosystem" each user places in a public file an encryption procedure *E*. That is, the public file is a directory giving the encryption procedure of each user. The user keeps secret the details of his corresponding decryption procedure *D*. These procedures have the following four properties:

(a) Deciphering the encrypted form of a message *M* yields *M*. Formally,

$$D(E(M)) = M. \quad (1)$$

(b) Both *E* and *D* are easy to compute.

(c) By publicly revealing *E* the user does not reveal an easy way to compute *D*. This means that in practice only he can decrypt messages encrypted with *E*, or compute *D* efficiently.

(d) If a message *M* is first deciphered and then enciphered, *M* is the result. Formally,

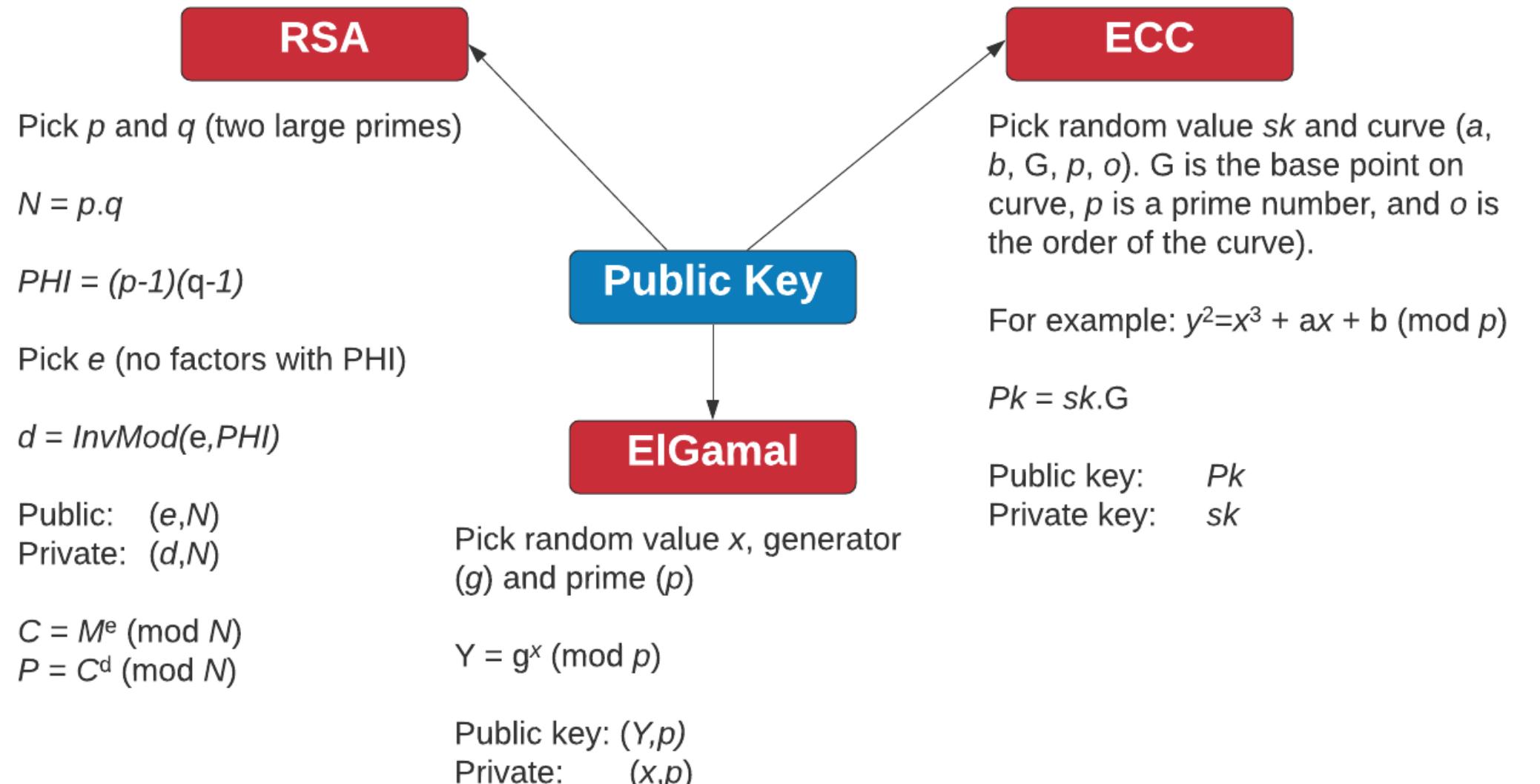
$$E(D(M)) = M. \quad (2)$$

An encryption (or decryption) procedure typically consists of a general method and an encryption key. The general method, under control of the key, enciphers a message *M* to obtain the encrypted form of the message, called the ciphertext *C*. Everyone can use the same general method; the security of a given procedure will rest on the security of the key. Decoding or encryption algorithms then means revealing the key.

When the user reveals *E* he reveals a very inefficient method of computing *D(C)*: testing all possible messages *M* until one such that *E(M) = C* is found. If property (c) is satisfied the number of such messages to test will be so large that this approach is impractical.

A function *E* satisfying (a)-(c) is a "trap-door one-way function"; if it also satisfies (d) it is a "trap-door one-way permutation." Diffie and Hellman [1] introduced the concept of trap-door one-way functions but

# Public Key



ECC was co-invented by Neal Koblitz and Victor S Miller  
ElGamal was invented by Tahir ElGamal

# RSA

Pick  $p$  and  $q$  (two large primes)

$$N = p \cdot q$$

$$\text{PHI} = (p-1)(q-1)$$

Pick  $e$  (no factors with PHI)

$$d = \text{InvMod}(e, \text{PHI})$$

Public:  $(e, N)$

Private:  $(d, N)$

$$C = M^e \pmod{N}$$

$$P = C^d \pmod{N}$$

Pick random  
( $g$ ) and

$$Y = g^x \pmod{N}$$

Public key:  
Private:

# ECC

Pick random value  $sk$  and curve ( $a, b, G, n$ )  
G is the base point

```
p, _ := rand.Prime(rand.Reader, psize)
q, _ := rand.Prime(rand.Reader, psize)

M, _ := new(big.Int).SetString(Mval, 10) // Base 10

N := new(big.Int).Mul(p, q)

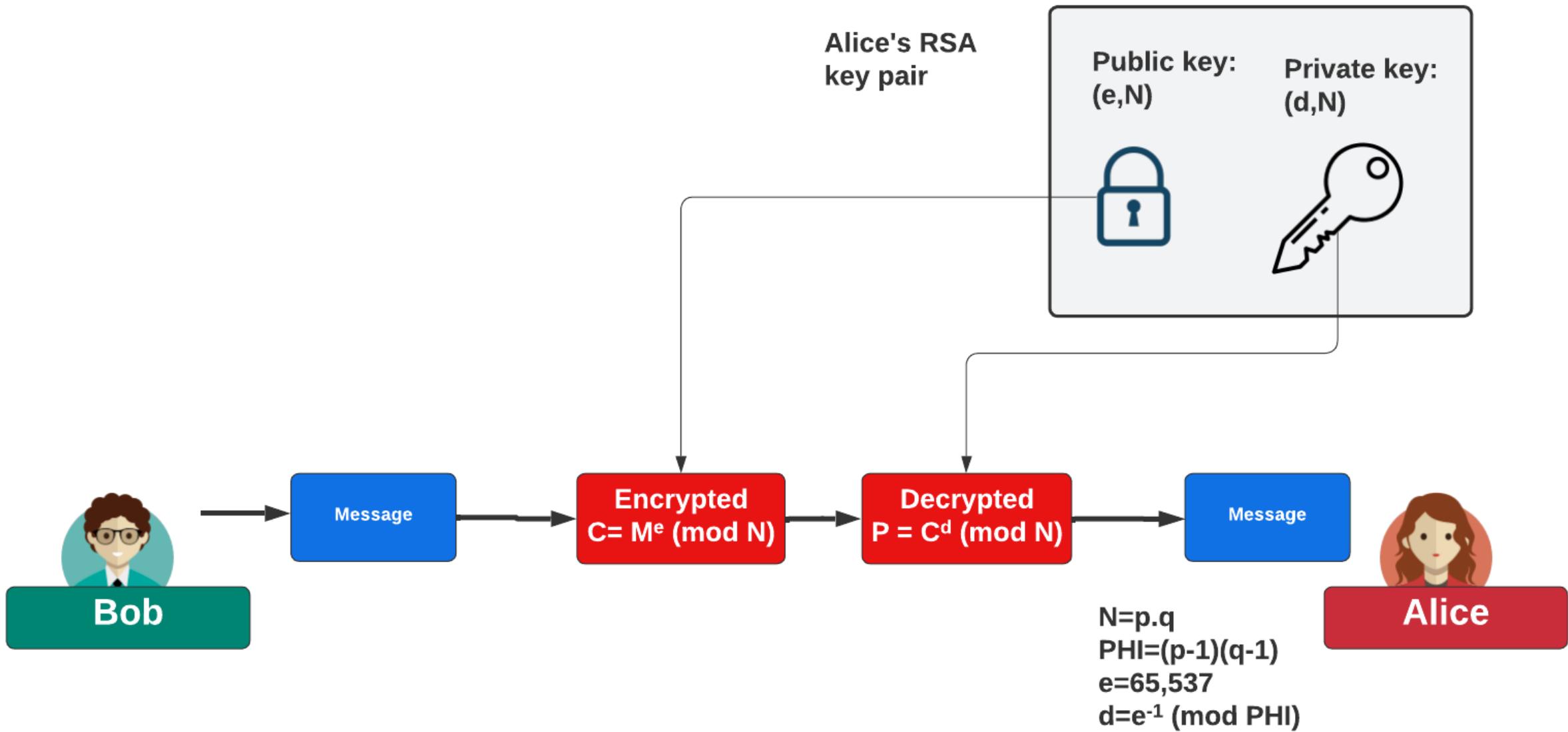
C := new(big.Int).Exp(M, e, N)

Pminus1 := new(big.Int).Sub(p, new(big.Int)..SetInt64(1))
Qminus1 := new(big.Int).Sub(q, new(big.Int)..SetInt64(1))
PHI := new(big.Int).Mul(Pminus1, Qminus1)

d := new(big.Int).ModInverse(e, PHI)

Plain := new(big.Int).Exp(C, d, N)
```

# RSA



# Optimal Asymmetric Encryption Padding (OAEP) padding

Parameters	
Message value (M):	Test
Modulus size (bits):	738
<b>Determine</b>	
<b>Input message:</b> [Test]	
<b>Key size:</b> [738]	
<b>Private key:</b> d=109d5c177608fd8afa68b20c05404cbe13c9bab993bdb48dde25f9e34c55b76e4c14fed045b676b5581e928302b54f4ddf31769cd8215b 68b8804ceeba2f5f6ef38de5644d57a28162d34580abc195d772161aaaf6b0178824b97ade01 N=2eef30f540f9dc342ba61897bd0bd9d8cfb60227b50bbe73516ec95f0db1a70fedacc9ae21441c0684fc96162066d016e0e7751dbaa59a 36716a24a1044dae2af00d0cbbc270eab3fc0769a02dec80ff9694233ffe718cf6213045ec7	
<b>Public key:</b> e=10001 N=2eef30f540f9dc342ba61897bd0bd9d8cfb60227b50bbe73516ec95f0db1a70fedacc9ae21441c0684fc96162066d016e0e7751dbaa59a 36716a24a1044dae2af00d0cbbc270eab3fc0769a02dec80ff9694233ffe718cf6213045ec7	
<b>Ciphertext:</b> 0121e712522185eb0306bc0c1127cf8114d460dab9a96c1021e8a08f009bf9b91ad33f7db155b1b7da8fa56dda89b150518028fd3d8f0026 4cd8f6b77b9a929b4028cae70f6609edfba94255e670f13c8e839c52c3b633fd209e803874	
<b>RSA decrypted to:</b> [Test]	

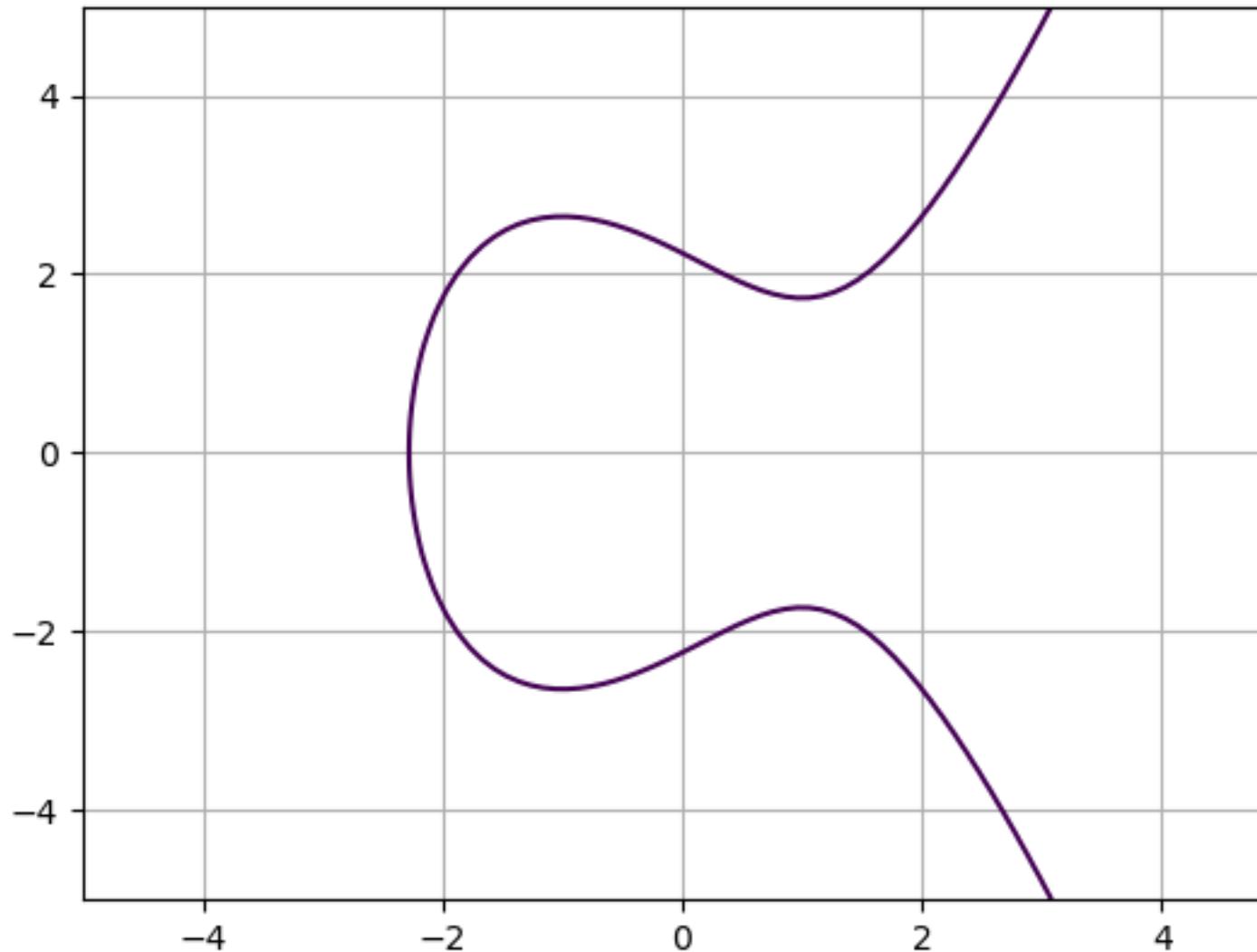
[https://asecuritysite.com/rsa/go\\_rsa2](https://asecuritysite.com/rsa/go_rsa2)

# RSA: PKCS1v15 padding

Parameters	
Message value (M):	Test
Modulus size (bits):	512
<button>Determine</button>	
<b>Input message:</b> [Test] <b>Key size:</b> [512]	
<b>Private key:</b> d=c662daac8a44ea6db2906153d66241f22fa2b30a3804bc402edba5dbb4bf1d850b46dee8392d4cb49ffe41e5c9d2012bd317a8471cb2df a6cae8aa22d0768ef9 N=c8b9e3173b2ed9545b6a8a6ecffbf5d9c9697827cf96fb4db24fbf245b0876982070b8217198884ec5bfa5411013c4a8d4e87937d3b78 4d5f2a9c4cfdddef717	
<b>Public key:</b> e=10001 N=c8b9e3173b2ed9545b6a8a6ecffbf5d9c9697827cf96fb4db24fbf245b0876982070b8217198884ec5bfa5411013c4a8d4e87937d3b78 4d5f2a9c4cfdddef717	
<b>Ciphertext:</b> b8d2148ff5072807b884db7aa2487811005fab1589a9f82818b593535e7b09ba69a1242e40e0d5c4780492b27ae6446781b8ef28e25a335e 9fd7b96a787d9675	
<b>RSA decrypted to:</b> [Test]	

[https://asecuritysite.com/rsa/go\\_rsa3](https://asecuritysite.com/rsa/go_rsa3)

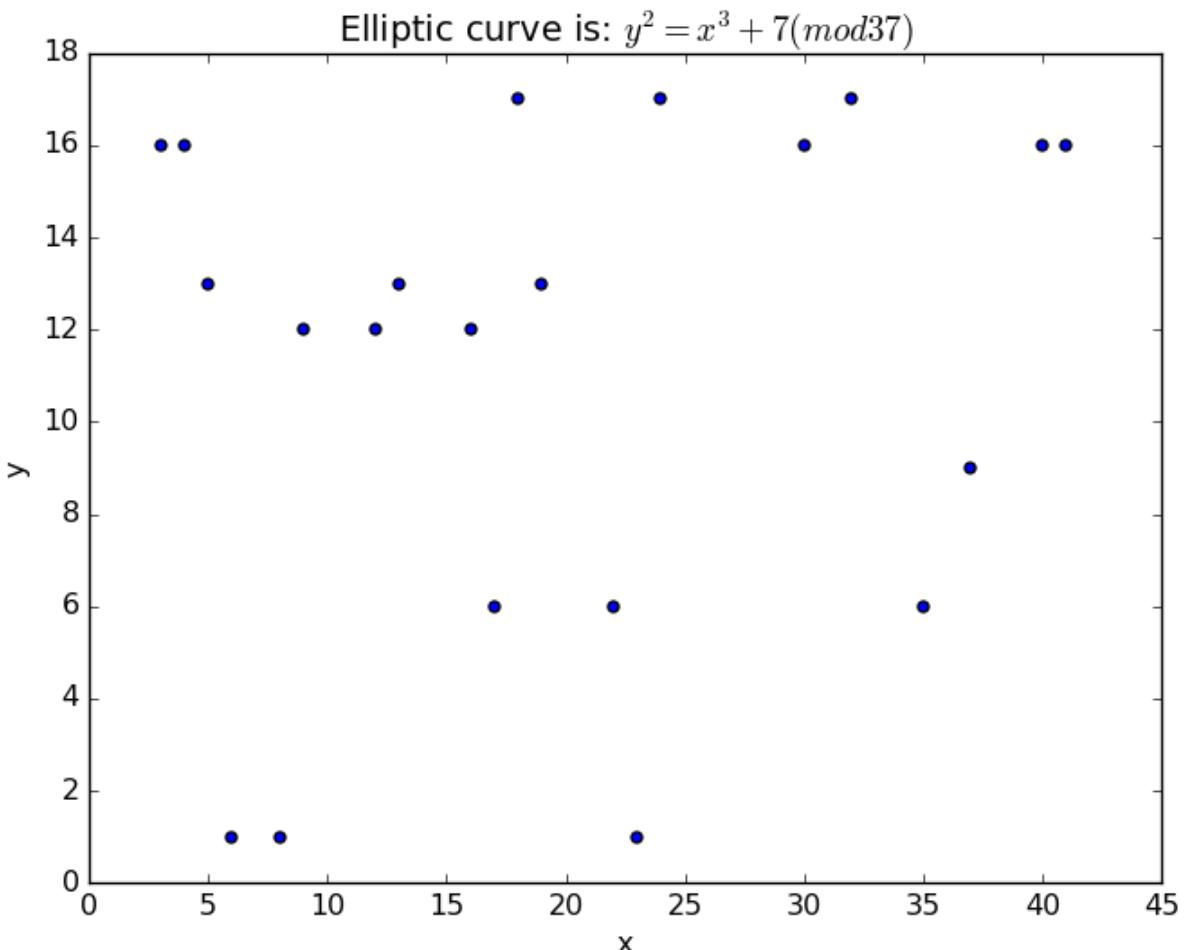
# Crypto – Elliptic Curve



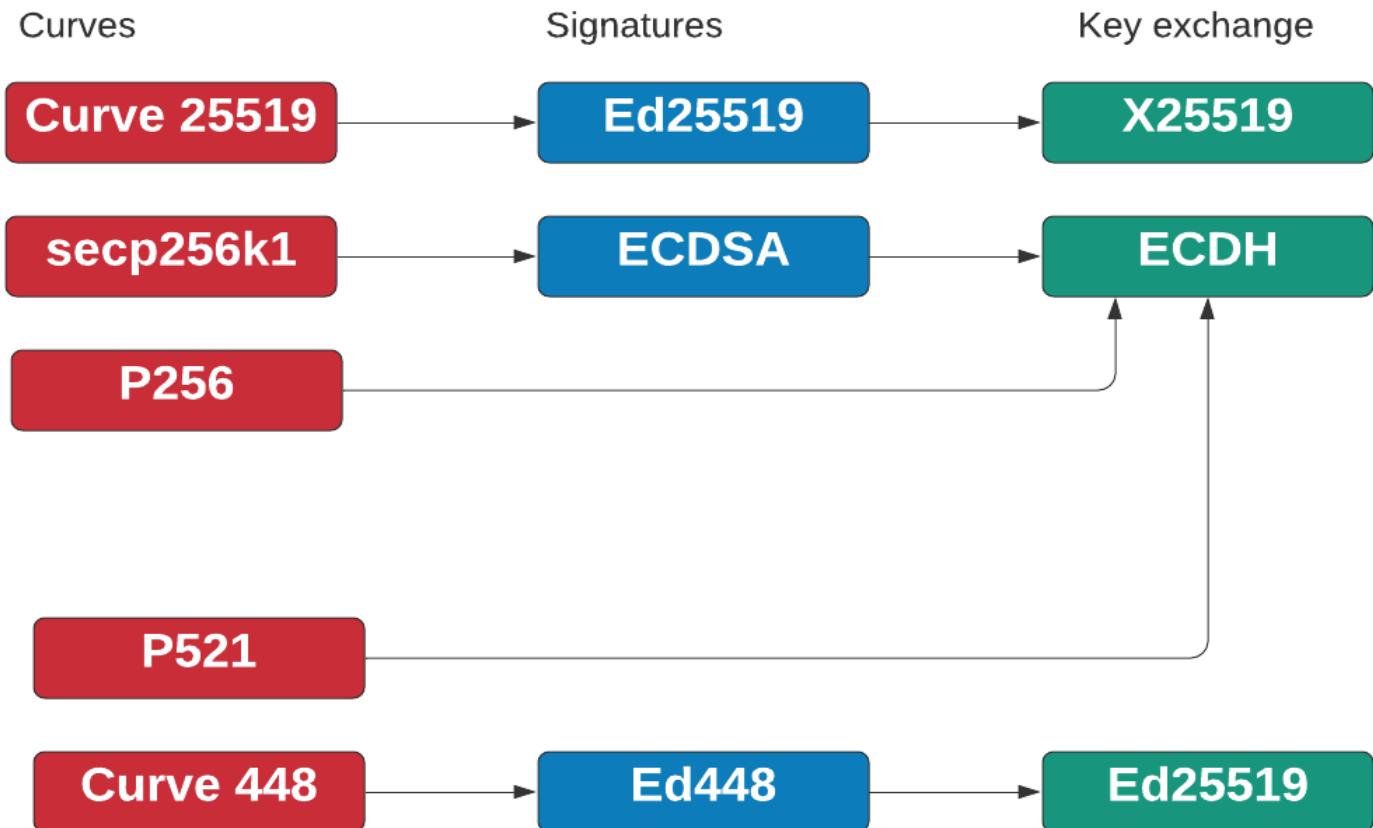
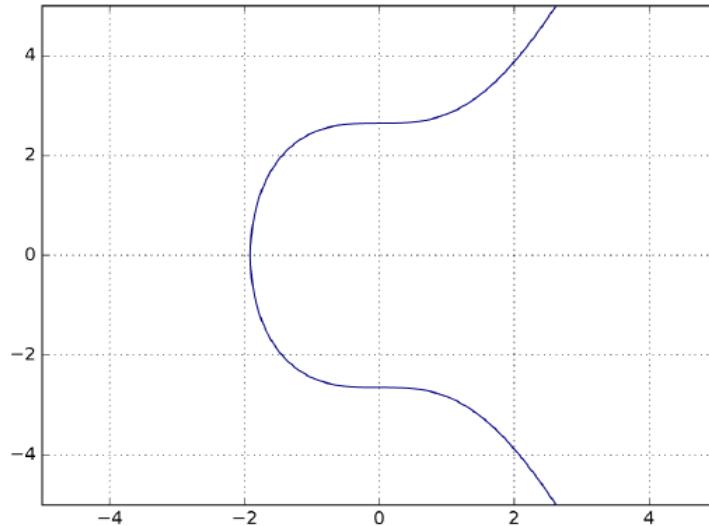
$$y^2 = x^3 - 3x + 5$$

# Elliptic Curve

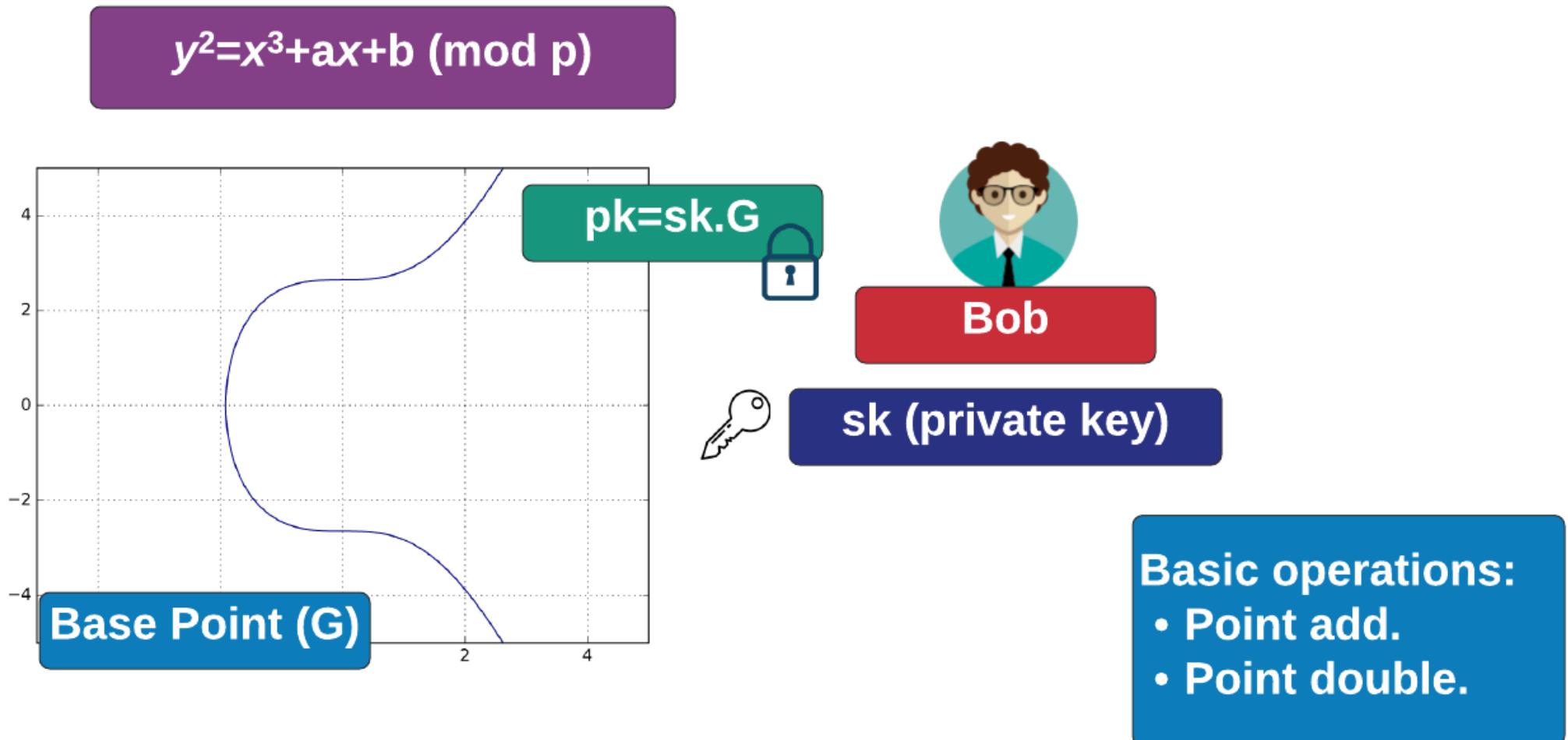
- A: 0
- B: 7
- Prime number: 37
- Elliptic curve is:  $y^2=x^3+7$
- Finding the first 20 points
- $\{ (3, 16), (4, 16), (5, 13), (6, 1), (8, 1), (9, 12), (12, 12), (13, 13), (16, 12), (17, 6), (18, 17), (19, 13), (22, 6), (23, 1), (24, 17), (30, 16), (32, 17), (35, 6), (37, 9), (40, 16), (41, 16) \}$  [Link]



# Elliptic Curve Types



# Crypto – Elliptic Curve Types



# ECC Keys

```
C \> openssl ec -in priv.pem -text -noout  
read EC key  
Private-Key (256 bit)  
priv  
        46 b9 e8 61 b6 3d 35 09 c8 8b 78 17 27 5a 30  
        d2 2d 62 c8 cd 8f a6 48 6d de e3 5e f0 d8 e0  
        49 5f  
pub  
        04 25 00 e7 f3 fb dd f2 84 29 03 f5 44 dd c8  
        74 94 ce 95 02 9a ce 4e 25 7d 54 ba 77 f2 bc  
        1f 3a 88 37 a9 46 1c 4f 1c 57 fe cc 49 97 53  
        38 1e 77 2a 12 8a 58 20 a9 24 a2 fa 05 16 2e  
        b6 62 98 7a 9f  
ASN1 OID    secp256k1
```

# ECC Parameters

```
C:> openssl ecparam -in priv.pem -text -param_enc explicit -noout
```

Field Type: prime-field

Prime:

```
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  
ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:  
ff:fc:2f
```

A: 0

B: 7 (0x7)

Generator (uncompressed):

```
04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:  
0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:  
f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:  
0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:  
8f:fb:10:d4:b8
```

Order:

```
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  
ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:  
36:41:41
```

Cofactor: 1 (0x1)

# Crypto – Compressed Point

We can reduce our public key point by compressing it. With a compressed point, we get a 32 byte compressed value (64 hex characters), and then add a 02 or a 03 to represent when y is even (02) or odd (03). This can be seen when we compress our public key here:

```
Private key: f91d8f3a49805fff9289769247e984b355939679f3080156fe295229e00f25af  
Public key: 52972572d465d016d4c501887b8df303eee3ed602c056b1eb09260dfa0da0ab288742f4dc97d9edb6fd946bab002fdb06f26caf117b9405ed79275763fdb1c  
Compressed public key 0252972572d465d016d4c501887b8df303eee3ed602c056b1eb09260dfa0da0ab2
```

In this case, we can see that we have a 02 at the start of the compressed public key point, and followed by the x-coordinate. This means that our y-coordinate value is even. Now let's look at another one:

```
Private key: ac609e0cc9681f8cb63e968be20e0f19721751561944f5b4e52d54d5f27ec57b  
Public key: 18ed2e1ec629e2d3dae7be1103d4f911c24e0c80e70038f5eb5548245c475f504c220d01e1ca419cb1ba4b3393b615e99dd20aa6bf071078f70fd949008e7411  
Compressed public key 0318ed2e1ec629e2d3dae7be1103d4f911c24e0c80e70038f5eb5548245c475f50
```

We can see that we now have a 03 at the start, followed by the x-coordinate value from the point. But how do we get the 02 or 03? Well in the first case the y-axis co-ordinate ends with a hex value of 'c' (1100), and which is even. In this case, we append a 02. In the second case, the y-coordinate ends with a hex value of '1' (0001), and is then odd. In this case, we add a 03. The compressed key is then:

```
02 (x-co-ordinate) // if y is even  
03 (x-co-ordinate) // if y is odd
```

When we need to uncompress, we can easily compute the y-coordinate value again from the x-coordinate value, and examine whether y is odd or even. This will recover the uncompressed point. Typically an uncompressed point will start with a 04.

In Ethereum, we also identify a user with their Ethereum address. This is a 160-bit (40 hex characters) address, and is derived from a Keccak 256 bits hash of the uncompressed x and y coordinate values. A sample run of this is:

```
Private key: f91d8f3a49805fff9289769247e984b355939679f3080156fe295229e00f25af  
Public key: 52972572d465d016d4c501887b8df303eee3ed602c056b1eb09260dfa0da0ab288742f4dc97d9edb6fd946bab002fdb06f26caf117b9405ed79275763fdb1c  
Compressed public key 0252972572d465d016d4c501887b8df303eee3ed602c056b1eb09260dfa0da0ab2  
Signer address 0x6eDBe1F6D48FbF1b053D6c9FA7997C710B84F55F
```

# Key Formats

```
Private key encoding: Encoding.DER
Private key format: PrivateFormat.PKCS8
Public key encoding: Encoding.DER
Public key format: PublicFormat.PKCS1
Curve: RSA
Key size: 512
```

Private key:

```
30820154020100300d06092a864886f70d01010105000482013e3082013a0201000241
668d99cd5079c7b58de99c3dd5ea2fab4a4bf19a41f1143aaef82452ea8609544986f01
5e83d398379a05aa5bee849bee9e70c2a7a6f75c7219a1bf763731c5ceb1230260ff50
1a3e55022100e523d1939db160871ff34adce4e3962c0748daf75b2333f3abf99ffba0
6b1becd83f2c2b7cf2b9b8de4d4736c13302200869863a2b21e5ef5006f88a9f849370
b01d68872d1b91b04b3efe8c52b05ee7b1989133b8c79deea295ce0a88e5022074a097
447812fee25cd7e6
```

Public key:

```
3048024100bfcf2c5b831cd6579189bab8418cb096372d101c668d99cd5079c7b58de9
44986f0117984afcdbcdbac154650203010001
```

== Key details ==

Private key p: 1036429284175408841708716171691326184059249873622204425

Private key q: 9692766106822671324849194294511378257445282574896448640 Public key:

Private key d: 3048024100c3c7efd9be6382903788fa956bf550692b8bf0c24efb276b3446fc5a87876f21a746e7bb01acc4943c4c3664a0db3058288e6f
3693412219618868338517214984699199561832564151843044761081684095783393
536781822416345057489540698633556754841173

Private key dmp1: 3804948132697611537225034879670004423221318027885047

Private key dmq1: 8680111125157126204363935077053767946800488797483445 == Key details ==

Private key iqmp: 52752034466750719807707789683704937408971067883174621237460030256227888715750

Public key N:

```
10045866637773885631932048908579848171148402441142877132476046091942572239751217003497317991502229329262409771
```

```
4420140016709442079080409919889220741190757
```

Public key e: 65537

Private key encoding:	Encoding.PEM
Private key format:	PrivateFormat.PKCS8
Public key encoding:	Encoding.DER
Public key format:	PublicFormat.PKCS1
Curve:	RSA
Key size:	512

Private key:

```
-----BEGIN PRIVATE KEY-----
MIIBVwIBADANBgkqhkiG9w0BAQEFAASCAUewggE9AgEAAkEAw8fv2b5jgpA3iPqV
a/VQaSuL8MJO+ydrNEb8WoeHbyGnRue7AazE1dxMNmSg2zBYKI5vwBNoBgfT5f7+
Zyh9nwIDAQABAkEAoHh/stTn0xuN2xuy/8MZ/qoP0o4joJ7tsvLAIx/VNZNI7C+w
I61nIYpjauIn1lsv9PsHzw7s6KQR5hLgYFXDCQIhAP7A/UeqgBVaRxYA63Qrg9+3
2eb0z1bx488u77070rTDAiEAxL0zk8FUC6JIHJFqY51hmX18pxCx/5cvB4frGAre
1fUCIQCAgZBhBeYtz1L8eM7xdrwHIt5NbQo2f7JG6TQgL1CKQIhAIvHmvuKuU8G
vtt1ScR3/osSSFVenEjvp1/IZvUo+XtdAiEARtkmgEr1hGtgOojs091SyyBMhNNC
-----END PRIVATE KEY-----
```

# Key Formats: Distinguished Encoding Rules (DER) encoding of ASN.1

## Details

DER:

```
3059301306072a8648ce3d02010  
6082a8648ce3d03010703420004  
2927b10512bae3eddcfe4678281  
28bad2903269919f7086069c8c4  
df6c732838c7787964eaac00e59  
21fb1498a60f4606766b3d96850
```

## Determine

The following are some examples:

- **Ex 1 (ECDSA P-256)** [Try.](#)
- **Ex 2 (ECDSA secp256k1)** [Try.](#)
- **Ex 3 (ECDSA secp256r1)** [Try.](#)
- **Ex 4 (ECDSA secp224r1)** [Try.](#)
- **Ex 5 (ECDSA Brainpool P224)** [Try.](#)
- **Ex 6 (Brainpool P224r1)** [Try.](#)
- **Ex 7 (secp521r1)** [Trv.](#)

DER string:

```
3059301306072a8648ce3d020106082a8648ce3d030107034200042927b10512bae3eddcfe467828128bad2903269919f7086069c8  
c4df6c732838c7787964eaac00e5921fb1498a60f4606766b3d9685001558d1a974e7341513e
```

==Sequence==

--->Sequence (30)

--->obj ID tag (06 - Object ID)

    ID algorithm: 1.2.840.10045.2.1 ECC (ecPublicKey)

--->obj ID tag (06 - Object ID)

    ID algorithm: 1.2.840.10045.3.1.7 secp256r1

--->obj ID tag (03)

    Bit value:

```
b'042927b10512bae3eddcfe467828128bad2903269919f7086069c8c4df6c732838c7787964eaac00e5921fb1498a60f4606766b3  
d9685001558d1a974e7341513e'
```

    Public key:

```
(18614955573315897657680976650685450080931919913269223958732452353593824192568, 902231163478598801665701987  
25387569567414254547569925327988539833150573990206
```

Now checking key if ECC - 256

```
EccKey(curve='NIST P-256',  
point_x=18614955573315897657680976650685450080931919913269223958732452353593824192568,  
point_y=90223116347859880166570198725387569567414254547569925327988539833150573990206)
```

(Abstract Syntax Notation One).

# Object identifier (OID)

```
Hashing:  
MD2      1.2.840.113549.2.2  
MD5      1.2.840.113549.2.5  
SHA-1    1.3.14.3.2.26  
SHA-224   2.16.840.1.101.3.4.2.4  
SHA-256   2.16.840.1.101.3.4.2.1  
SHA-394   2.16.840.1.101.3.4.2.2  
SHA-512   2.16.840.1.101.3.4.2.3  
Public key:  
RSA Encryption 1.2.840.113549.1.1.1  
DSA      1.2.840.10040.4.1  
Diffie-Hellman (dhPublicNumber) 1.2.840.10046.2.1  
ECC (ecPublicKey) 1.2.840.10045.2.1  
md2WithRsaEncryption 1.2.840.113549.1.1.2  
Signatures:  
md5WithRsaEncryption 1.2.840.113549.1.1.4  
sha1WithRsaEncryption 1.2.840.113549.1.1.5  
sha224WithRsaEncryption 1.2.840.113549.1.1.14  
sha256WithRsaEncryption 1.2.840.113549.1.1.11  
sha384WithRsaEncryption 1.2.840.113549.1.1.12  
sha512WithRsaEncryption 1.2.840.113549.1.1.13  
dsaWithSha1 1.2.840.10040.4.3  
dsaWithSha224 2.16.840.1.101.3.4.3.1  
dsaWithSha256 2.16.840.1.101.3.4.3.2  
ecdsaWithSha1 1.2.840.10045.4.1  
ecdsaWithSha224 1.2.840.10045.4.3.1  
ecdsaWithSha256 1.2.840.10045.4.3.2  
ecdsaWithSha384 1.2.840.10045.4.3.3  
ecdsaWithSha512 1.2.840.10045.4.3.4  
Password Base Encryption  
Algorithms:  
pbeWithMd2AndDesCbc 1.2.840.113549.1.5.1  
pbeWithMd5AndDesCbc 1.2.840.113549.1.5.3  
pbeWithSha1AndDesCbc 1.2.840.113549.1.5.10  
pbeWithMd2AndRc2Cbc 1.2.840.113549.1.5.4  
pbeWithMd5AndRc2Cbc 1.2.840.113549.1.5.6  
pbeWithSha1AndRc2Cbc 1.2.840.113549.1.5.11  
pbeWithSha1And40BitRc2Cbc 1.2.840.113549.1.12.1.6  
pbeWithSha1And128BitRc2Cbc 1.2.840.113549.1.12.1.5  
pbeWithSha1And40BitRc4 1.2.840.113549.1.12.1.2  
pbeWithSha1And128BitRc4 1.2.840.113549.1.12.1.1  
pbeWithSha1And3DesCbc 1.2.840.113549.1.12.1.3  
Symmetric Encryption  
Algorithms:  
DES CBC 1.3.14.3.2.7  
3DES CBC 1.2.840.113549.3.7
```

-- RSA Methods --

- 1.2.840.113549.1.1.1
- 1.2.840.113549.1.1.2
- 1.2.840.113549.1.1.3
- 1.2.840.113549.1.1.4
- ✓ 1.2.840.113549.1.1.5
- 1.2.840.113549.1.1.6
- 1.2.840.113549.1.1.7
- 1.2.840.113549.1.1.8
- 1.2.840.113549.1.1.9**
- T 1.2.840.113549.1.1.10
- e 1.2.840.113549.1.1.11
- a
- D

-- Hashing Methods --

- 1.2.840.113549.2.2
- 1.2.840.113549.2.4
- 1.2.840.113549.2.5

1.2.840.113549.1.1.5

**Determine**

te.com  
networksims

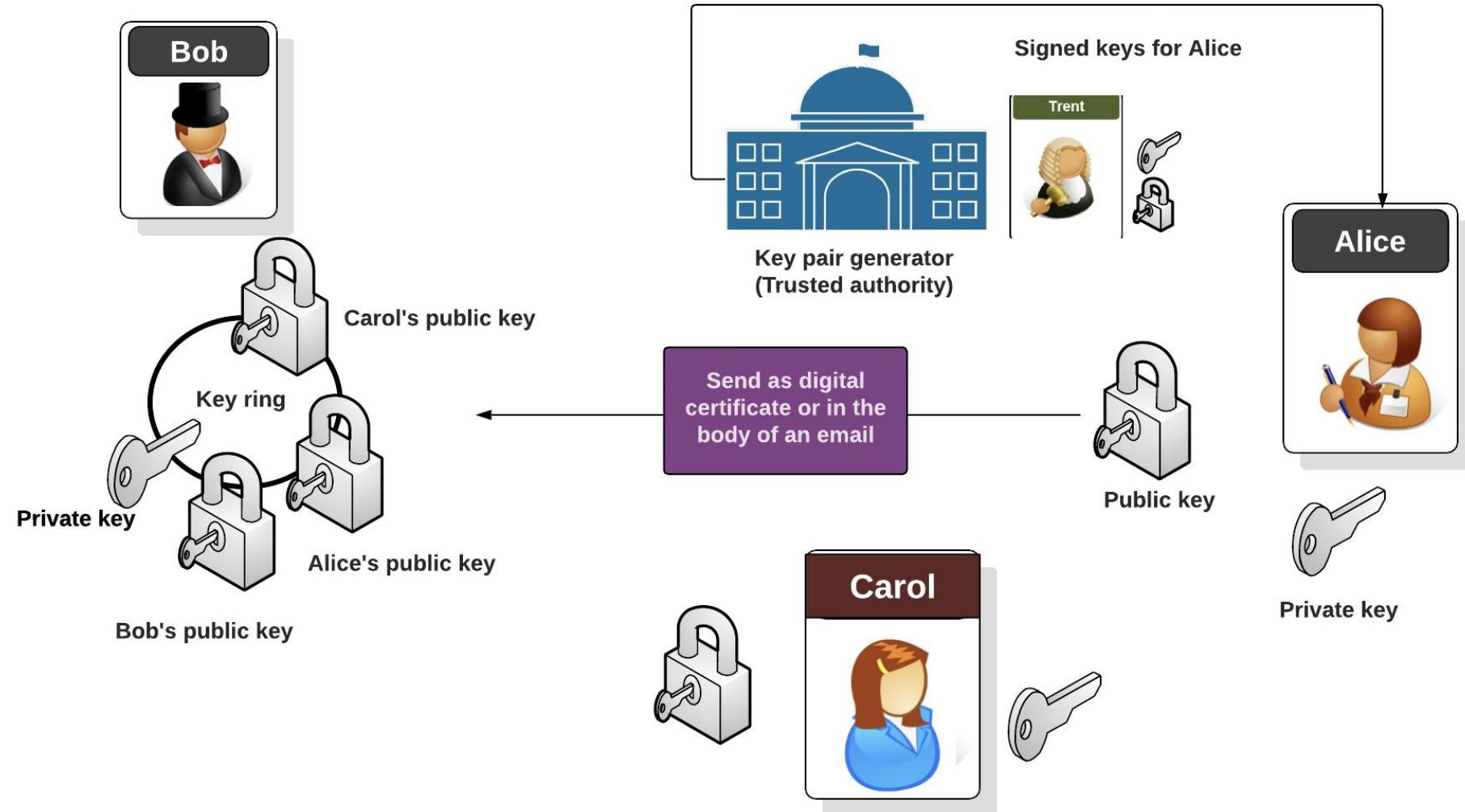
IDS MAGIC NET CISCO CYBER TEST

fine the cryptography methods used. An example is OSI, 2 is member body, 840 is US (ANSI). Algorithms are: "1.2.840.113549.1.1.1" (X509 RSA), "1.2.840.10046.2.1" (Diffie-Hellman — DH).

**OID mapping:**

1.2.840.113549.1.1.5 sha1RSA

# Key Ring



Bob



Alice



# Cryptography: Key Exchange

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com/keyexchange>

Fundamentals

Symmetric Key

Hashing

MAC

KDF

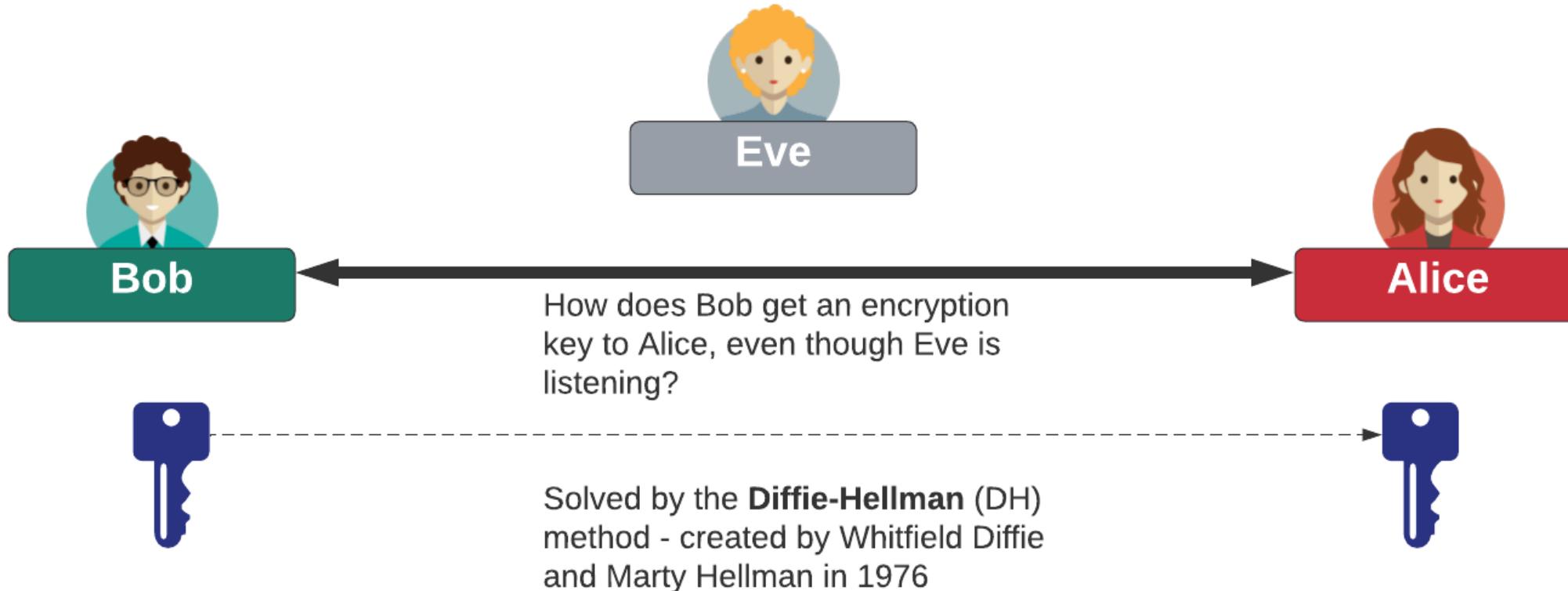
Public Key

**Key Exchange**

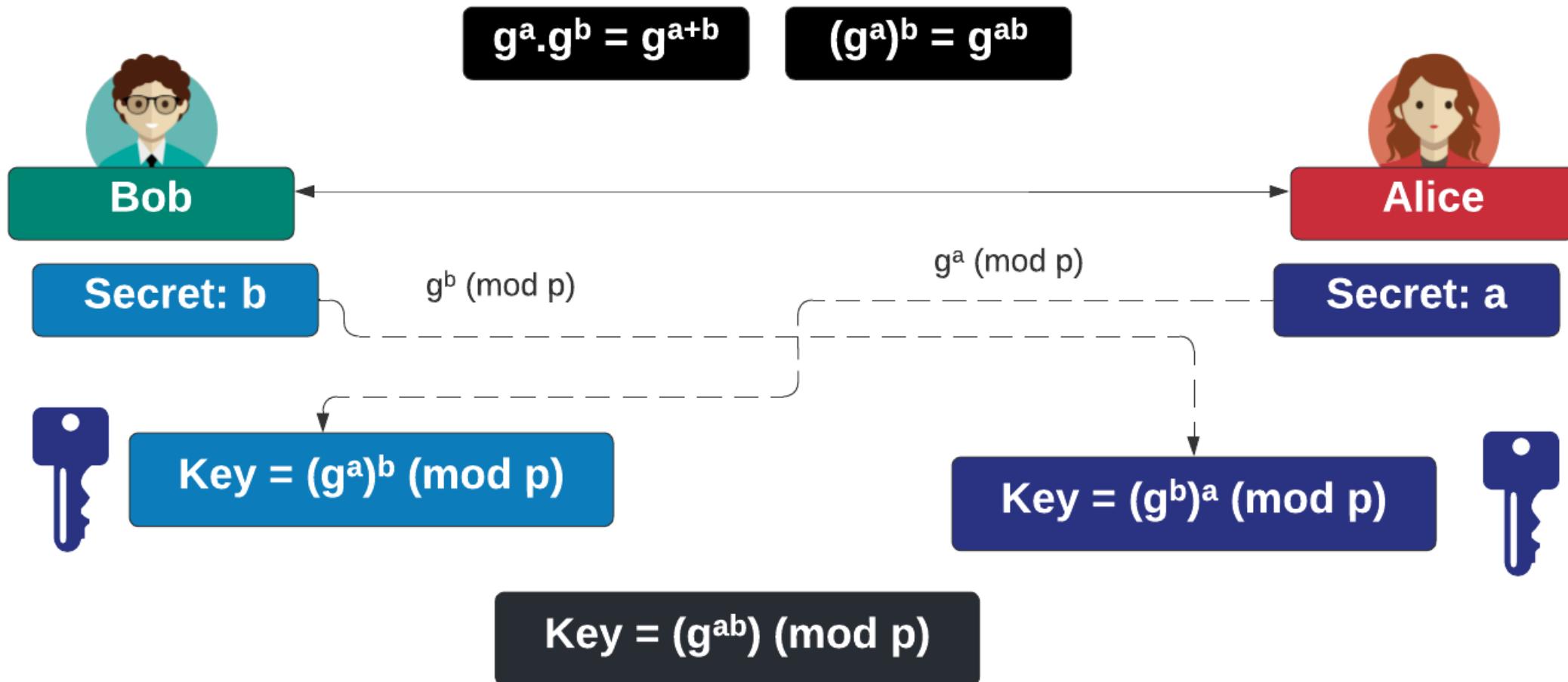
Signatures

Digital Certificates

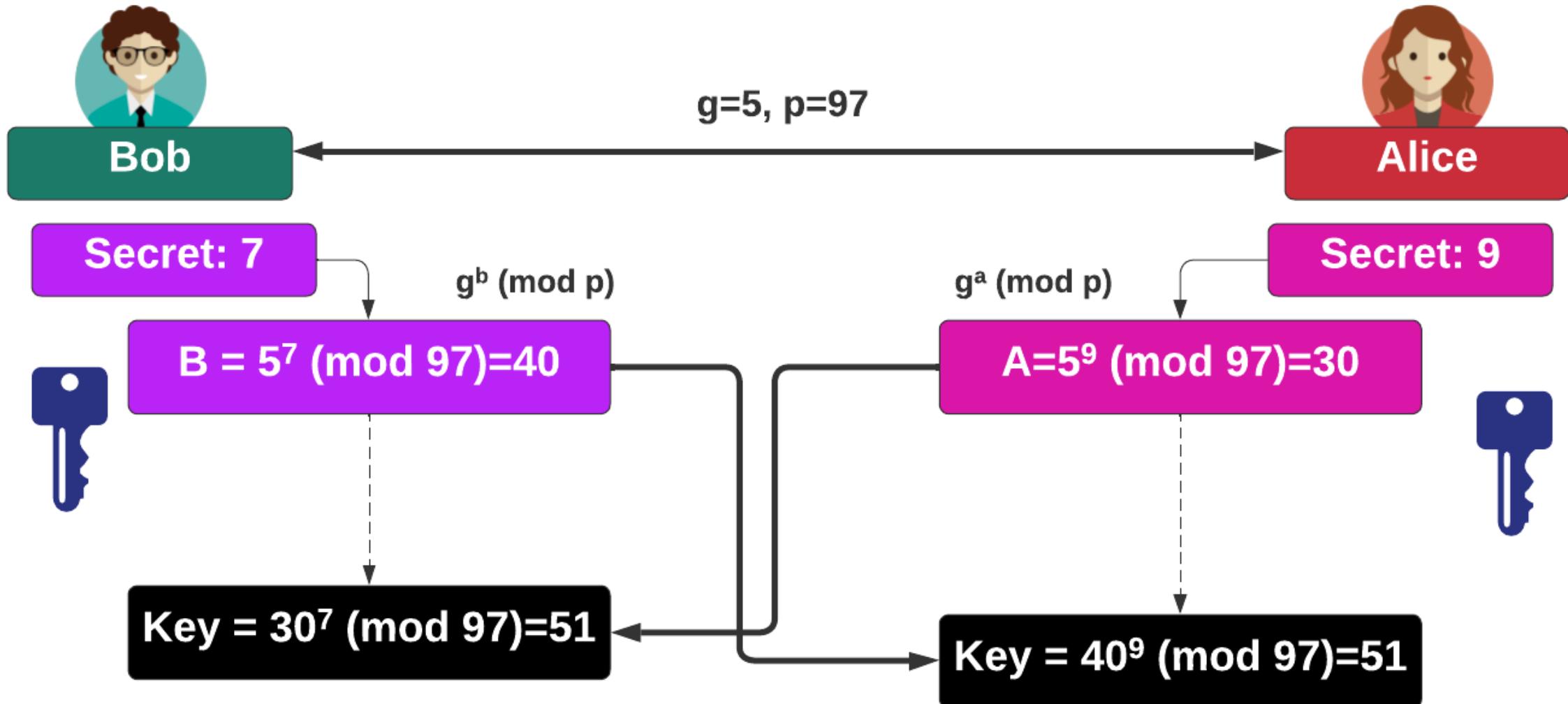
# Key Exchange Problem



# The Diffie-Hellman Method



# The Diffie-Hellman Method



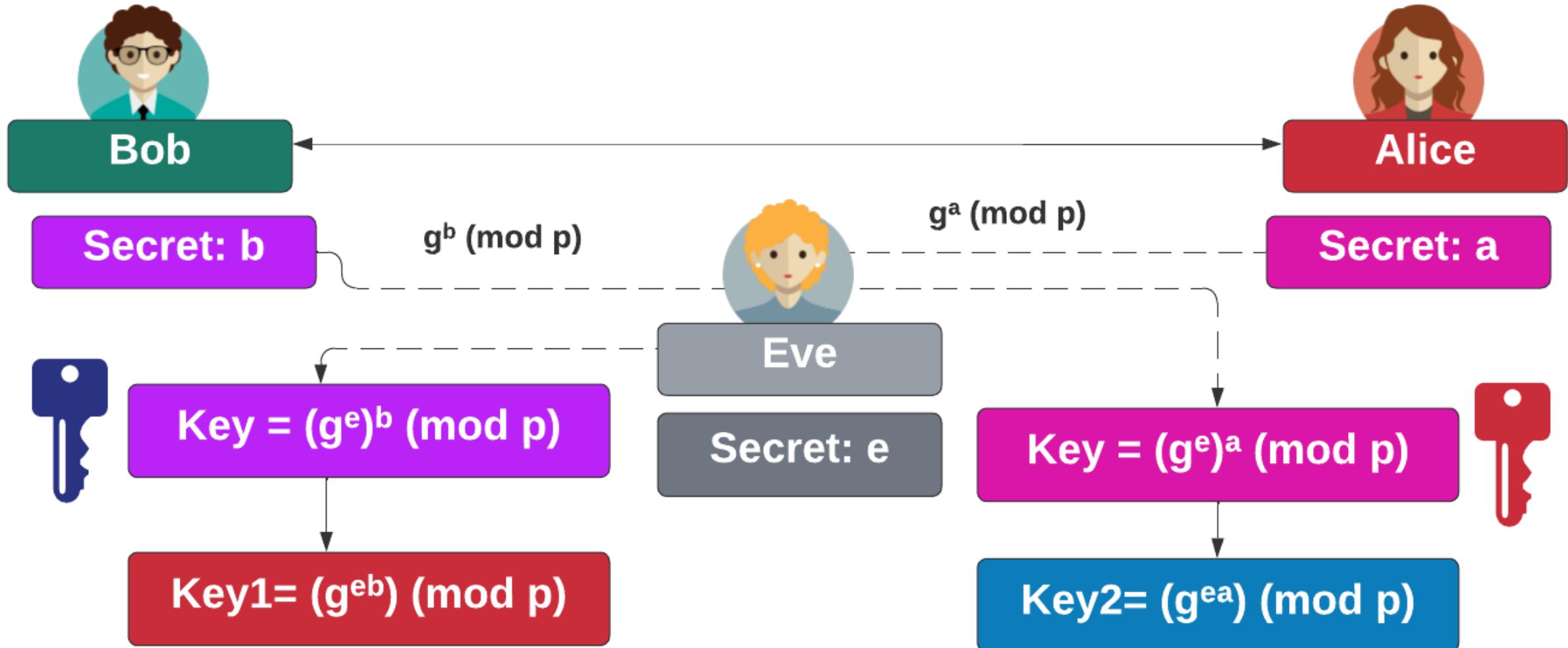
# Picking g and p

$$Y = g^x \pmod{p}$$

Let's try  $p=11$

# Picking G

# Eve-in-the-Middle



## Diffie-Hellman Generation

```
C:\> openssl dhparam -out dhparams.pem 768 –text  
C:\> type dhparams.pem  
Diffie-Hellman-Parameters: (768 bit)  
prime:  
00:d0:37:c2:95:64:02:ea:12:2b:51:50:a2:84:6c:  
71:6a:3e:2c:a9:80:e2:65:b2:a5:ee:77:26:22:31:  
66:9e:fc:c8:09:94:e8:9d:f4:cd:bf:d2:37:b2:fb:  
b8:38:2c:87:28:38:dc:95:24:73:06:d3:d9:1f:af:  
78:01:10:6a:7e:56:4e:7b:ee:b4:8d:6b:4d:b5:9b:  
93:c6:f1:74:60:01:0d:96:7e:85:ca:b8:1f:f7:bc:  
43:b7:40:4d:4e:87:e3  
generator: 2 (0x2)  
----BEGIN DH PARAMETERS----  
MGYCYQDQN8KVZALqEitRUKKEbHFqPiypgOJlsqXudyYiMWae/MgJlOid9  
M2/Ojey  
+7g4LlcoONyVJHMG09kfr3gBEGp+Vk577rSNa021m5PG8XRgAQ2WfoXKu  
B/3vEO3  
QE1Oh+MCAQI=  
----END DH PARAMETERS----
```

- **DH Group 5:**  
1,536 bit prime.
- **DH Group 2:**  
1,024 bit prime.
- **DH Group 1:**  
768-bit prime.

# Elliptic Curve Diffie Hellman

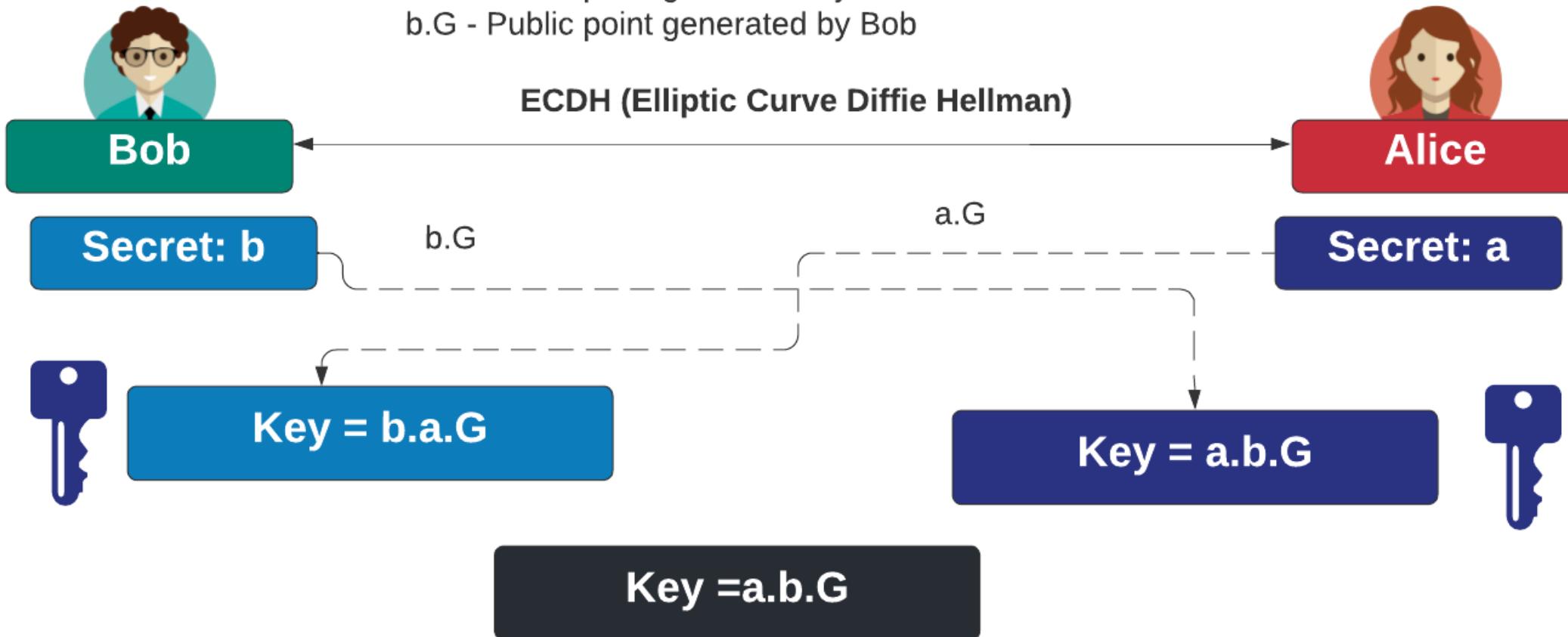
G - The base point on the curve (eg secp256k1 or P256)

a - Random scalar selected by Alice

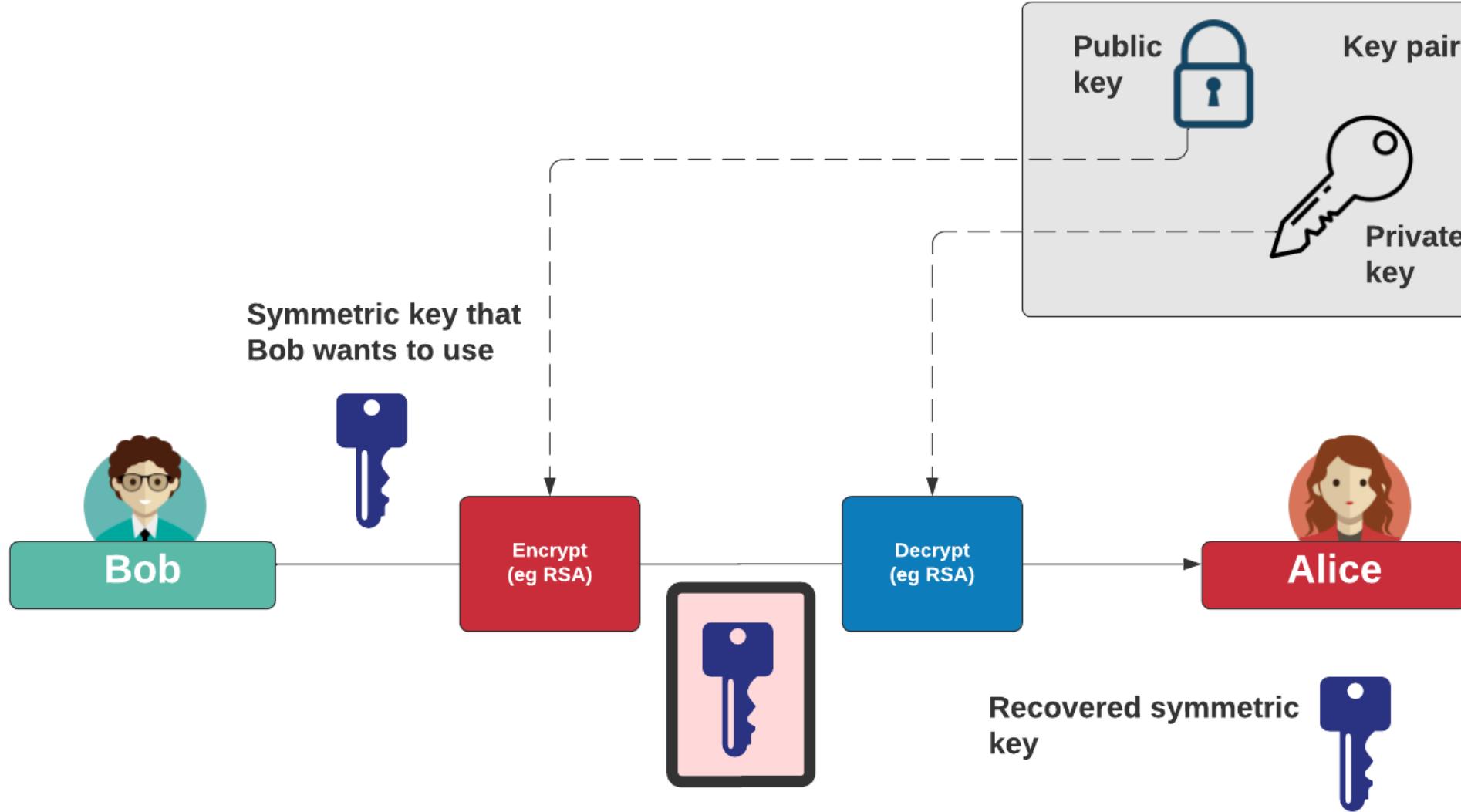
b - Random scalar selected by Bob

a.G - Public point generated by Alice

b.G - Public point generated by Bob



# Key Exchange with Public Key

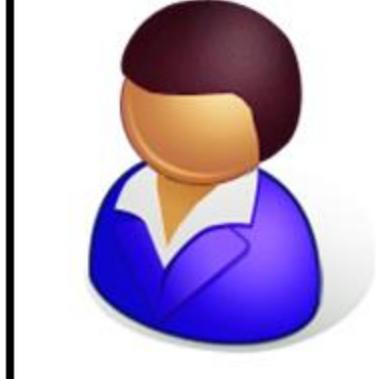


## Key Exchange

- **Forward secrecy (FS)**, which means that a compromise of the long-term keys will not compromise any previous session keys. A leakage of the public key of the server would cause all the sessions which used this specific public key to be compromised. FS thus aims to overcome this by making sure that all the sessions keys could not be compromised, even though the long-term key was compromised.
- **Ephemeral**. With some key exchange methods, the same key will be generated if the same parameters are used on either side. This can cause problems as an intruder could guess the key, or even where the key was static and never changed. With ephemeral methods, a different key is used for each connection, and, again, the leakage of any long-term would not cause all the associated session keys to be breached.



Bob



Alice



# Cryptography: Digital Signatures

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com/signatures>

Fundamentals

Symmetric Key

Hashing

MAC

KDF

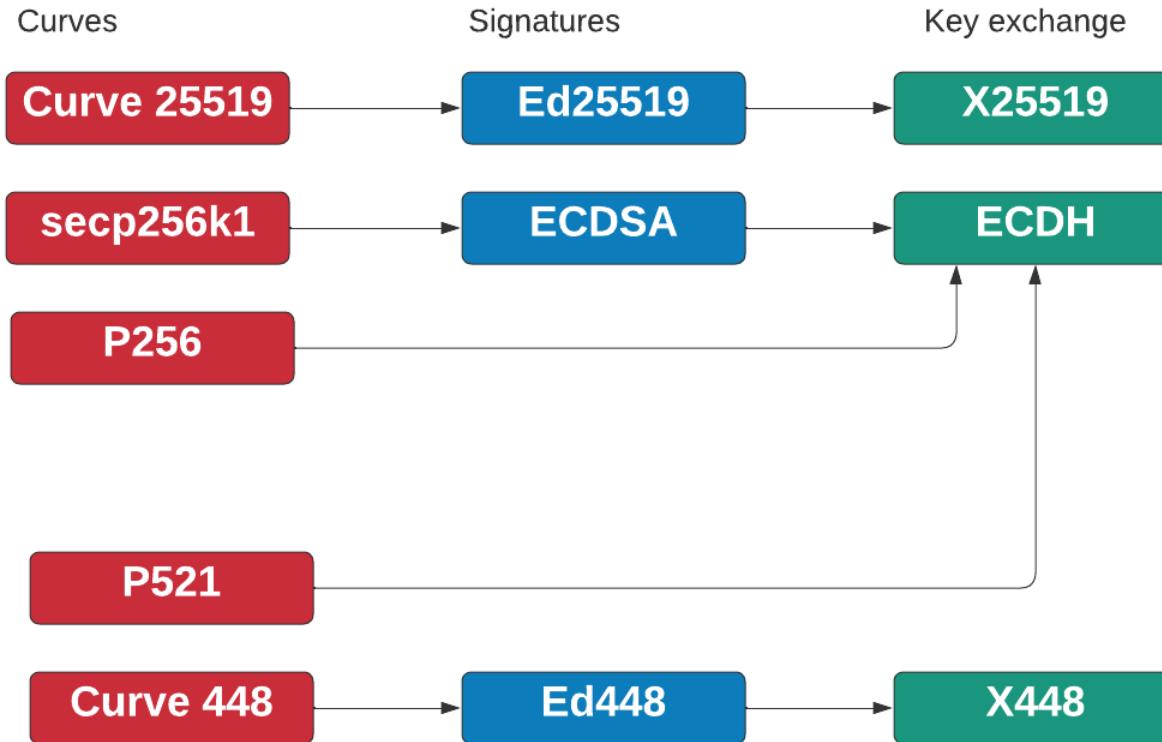
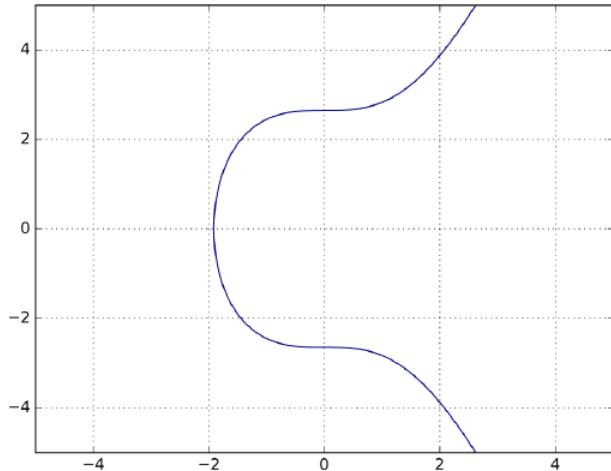
Public Key

Key Exchange

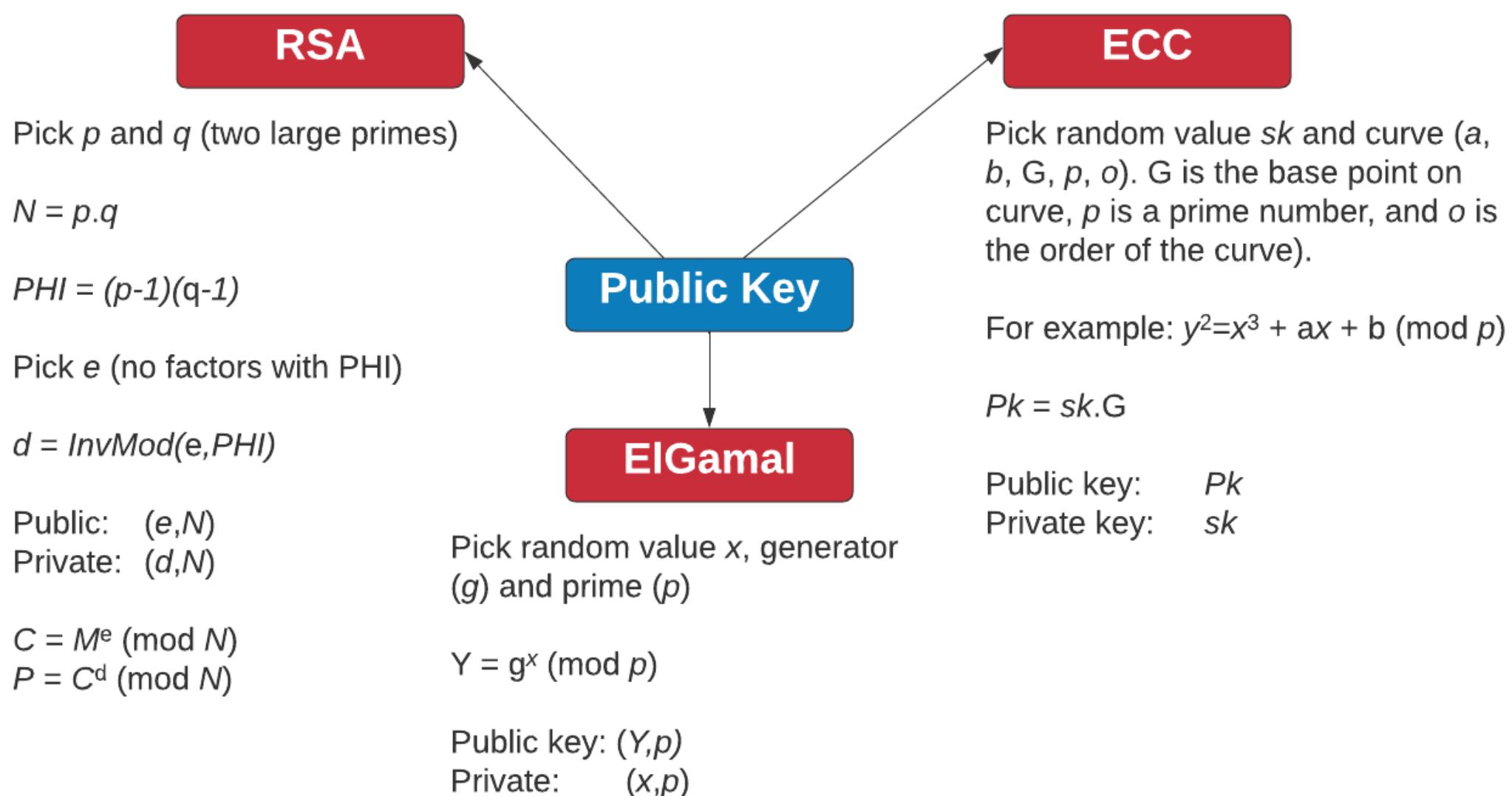
**Signatures**

Digital Certificates

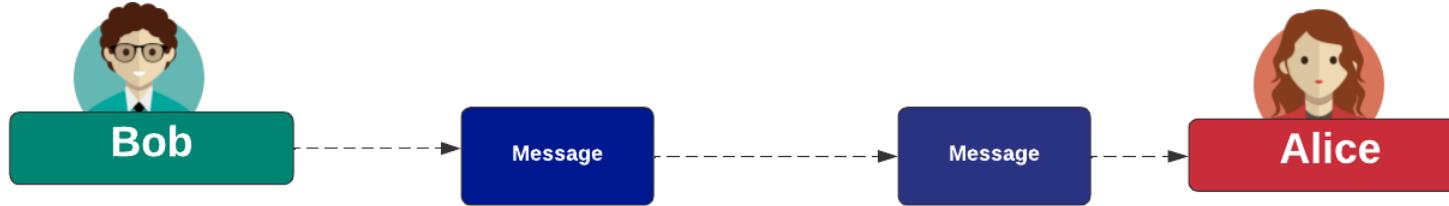
# Digital Signatures



# Digital Signatures



# Digital Signatures



## Hashing:

- SHA-256, SHA-3 or BLAKE. All 256-bit hashes (or greater).



## Symmetric key (Privacy and integrity):

- AES (128-bit or 256-bit). Normally with GCM mode for integrity.
- ChaCha20 (256-bit). Normally with Poly1305 for integrity.



## Key exchange (Privacy and Integrity):

- ECDH (eg P256: 256-bit private key, 512-bit public key)
- X25519 (eg Curve 25519: 256-bit private key and 256-bit public key)
- RSA key exchange (no recommended, and not supported in TLS 1.3)
- Not used any more: Diffie-Hellman



## Public Key Encryption (Privacy and Integrity)

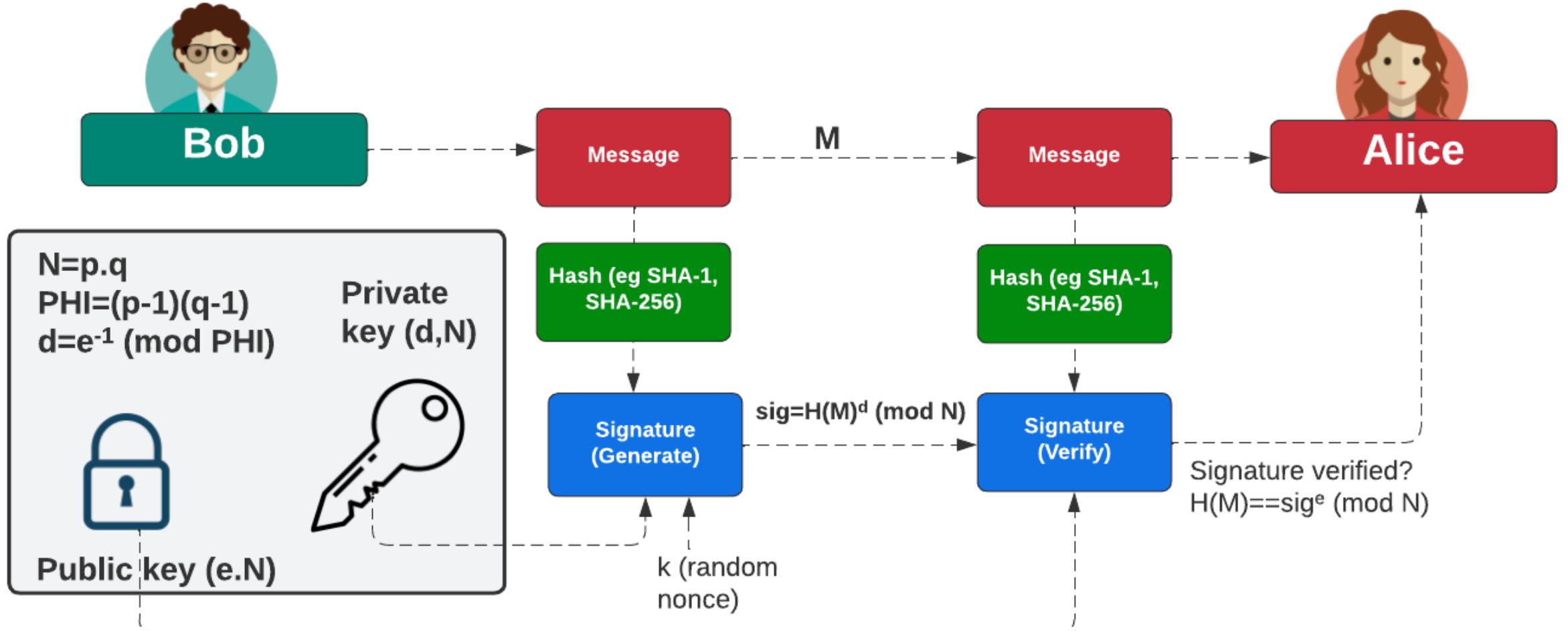
- RSA: 2,048-bit (2K) or 4,096-bit (4K) public key (Never less!)
- Not used any more: ElGamal (Discrete logs).



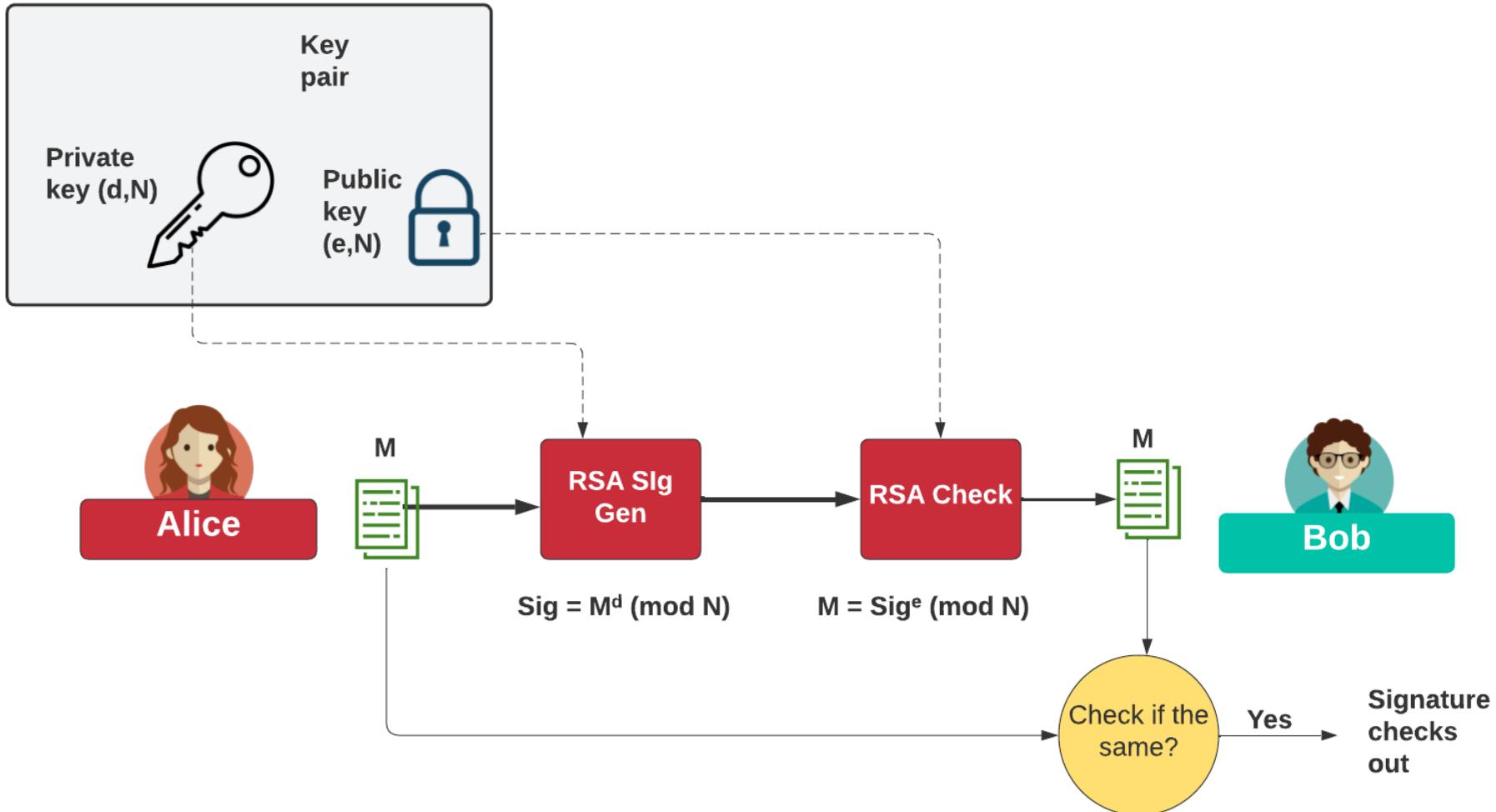
## Digital signatures (Trust and Integrity):

- RSA: 2,048-bit (2K) or 4,096-bit (4K) public key (Never less!)
- ECDSA (eg P256: 256-bit private key, 512-bit public key)
- EdDSA (eg Ed25519: 256-bit private key, 256-bit public key).
- Not used any more: DSA (Digital Signature Algorithm).

# Digital Signatures



# Digital Signatures - RSA



# RSA Digital Signatures – PSS and PKCS1v15



## RSA Signatures in Golang (PKCS1v15 padding)

[RSA Home][Home]

With RSA, we have the magic of public key encryption, and where Bob can generate a key pair: a public key and a private key, and then send Alice his public key. If she wants to encrypt something for him, she encrypts it with his public key, and then the only key which can decrypt it is his private key. A truly wonderful concept, and it was Rivest, Shamir and Adleman who made it come alive with the RSA method. In RSA, we start by generating two prime numbers ( $p, q$ ) and then calculate the modulus ( $N$ ) of  $N = pq$ . It is this modulus ( $N$ ) that provides the core of the security of RSA, and it must be difficult to determine the prime numbers for a given modulus value. Our prime numbers must thus be of a size that makes it difficult to factorize, and they must be randomly generated. In this case, we will create a signature with PKCS1v15 padding.

### Parameters

Message value (M):

Test

Modulus size (bits):

512

Determine

Input message: [Test]

Message hash: [532eaabd9574880dbf76b9b8cc00832c20a6ec113d682299550d7a6e0f345e25]

Key size: [512]

Private key:

```
d=5715372a1e8c7edf9e6aeb634bbae300c751db8c1165237a1f397ab6e41434815a3377f8f59a8  
191f4967b1cb2cbbb1  
N=bc4aeb2a997855a4dc6e044688b59d0266c2966c7490d721b8fb7048f6b7f5c88571c8b207b99  
875b4caf8d99aa329f
```

Public key:

```
e=10001  
N=bc4aeb2a997855a4dc6e044688b59d0266c2966c7490d721b8fb7048f6b7f5c88571c8b207b99  
875b4caf8d99aa329f
```

Signature:

```
11ae9681ac1cdff6c7ce65114920f7f2fe1d5e75ee9a7eae11effdc26169f5cb8b68807151ed0a8  
45b9a07925032f6e
```

RSA signature verified



RSA

C=M<sup>e</sup> (mod N) N=p·q  
@asecuritysite.com

## RSA PSS (Probabilistic Signature Scheme) Signatures in Golang

[RSA Home][Home]

With RSA, we have the magic of public key encryption, and where Bob can generate a key pair: a public key and a private key, and then send Alice his public key. If she wants to encrypt something for him, she encrypts it with his public key, and then the only key which can decrypt it is his private key. A truly wonderful concept, and it was Rivest, Shamir and Adleman who made it come alive with the RSA method. In RSA, we start by generating two prime numbers ( $p, q$ ) and then calculate the modulus ( $N$ ) of  $N = pq$ . It is this modulus ( $N$ ) that provides the core of the security of RSA, and it must be difficult to determine the prime numbers for a given modulus value. Our prime numbers must thus be of a size that makes it difficult to factorize, and they must be randomly generated. In this case, we will create a signature with PSS.

### Parameters

Message value (M):

Test

Modulus size (bits):

512

Determine

Input message: [Test]

Message hash (SHA-256): [532eaabd9574880dbf76b9b8cc00832c20a6ec113d682299550d7a6e0f345e25]

Key size: [512]

Private key:

```
d=7a35e2f676df00f2c607a72c1edf0660ba7124faa9186423aac904c6e0eb0cc9769626a2e66e5540dd27807525690f5e7f9616b493716e  
fea6af235d43c64401  
N=acf5152f9e77f05a9cd663ed317bd6b120cdb008522cff8a710e9f3f538385ff3cf164119366435c10aeab3b638e15abf8aaff69e0ede  
682b5b0e4e1d9a0519
```

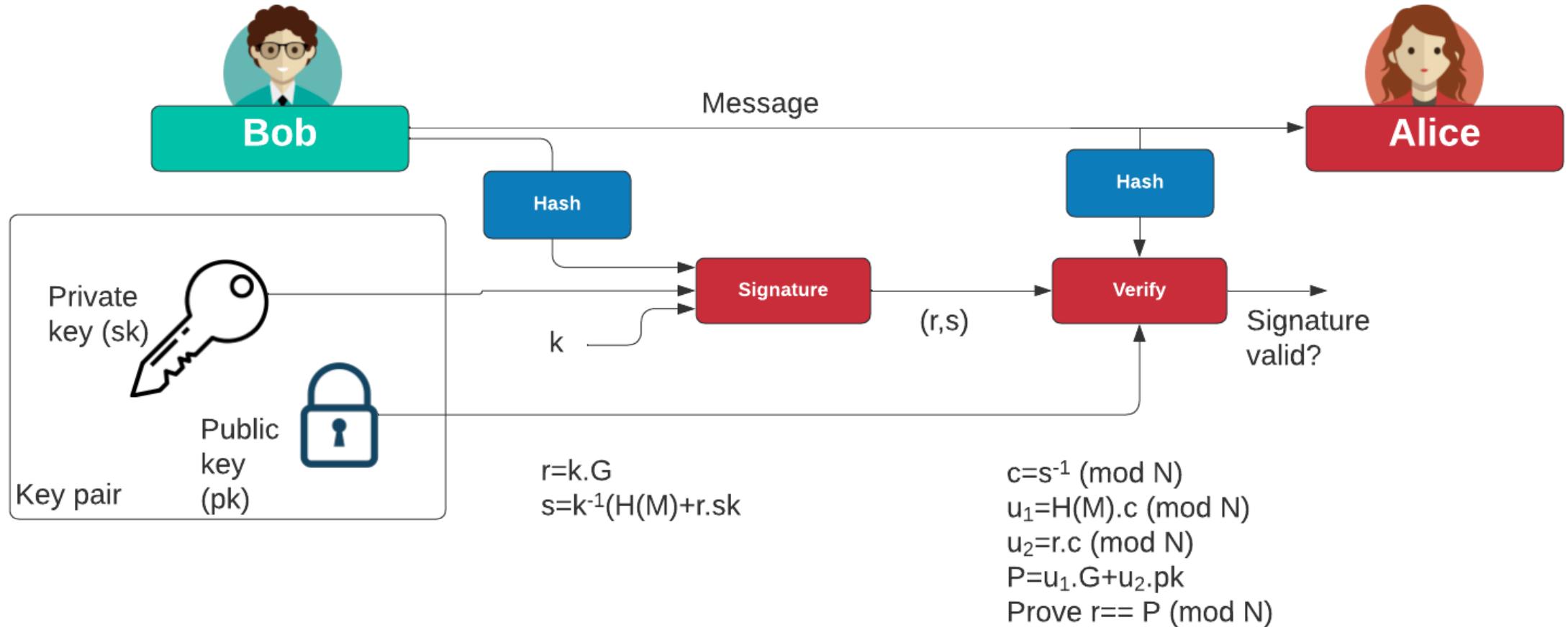
Public key:

```
e=10001  
N=acf5152f9e77f05a9cd663ed317bd6b120cdb008522cff8a710e9f3f538385ff3cf164119366435c10aeab3b638e15abf8aaff69e0ede  
682b5b0e4e1d9a0519
```

[https://asecuritysite.com/rsa/go\\_rsa4](https://asecuritysite.com/rsa/go_rsa4)

[https://asecuritysite.com/rsa/go\\_rsa5](https://asecuritysite.com/rsa/go_rsa5)

# ECDSA

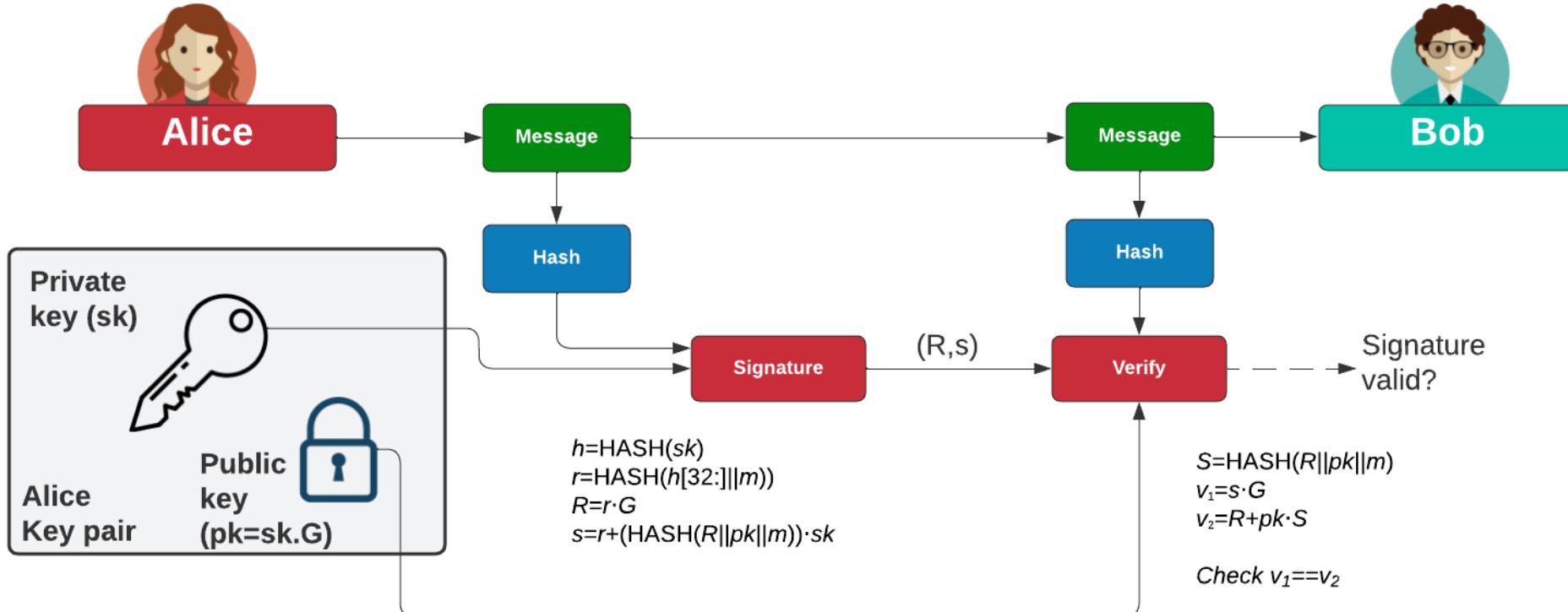


# ECDSA: Leak of nonce

Parameters	
Word: hello	<b>Curve detail</b> CurveFp( $p=115792089210356248762697446949407573530086143415290314195533631308867097853951$ , $a=-3$ , $b=41058363725152142129326129780047268409114441015993725554835256314039467401291$ , $h=1$ ) Order: $115792089210356248762697446949407573529996955224135760342422259061068512044369$ $G_x = 48439561293906451759052585252797914202762949526041747995844080717082404635286$ $G_y = 36134250956749795798585127919587881956611106672985015071877198253568414405109$  Message 1: hello Sig 1 r,s: $111556199447188233464592626218928344198308613139349238222544272467155808532621$ $36071171636130124290147336598589939732306126558321434039919651474179812189850$  Found Key: $108471298133852082312951871271653626071185712600280089079900425690596745399845$  Key: $108471298133852082312951871271653626071185712600280089079900425690596745399845$  The private key has been found $108471298133852082312951871271653626071185712600280089079900425690596745399845$

<https://asecuritysite.com/cracking/ecd2>

# EdDSA



[https://asecuritysite.com/golang/ecdsa\\_rs2](https://asecuritysite.com/golang/ecdsa_rs2)

[https://asecuritysite.com/golang/ecdsa\\_rs](https://asecuritysite.com/golang/ecdsa_rs)

# Cloud Keys and Signatures

## Configure key

**Key type** [Help me choose](#)

Symmetric  
A single key used for encrypting and decrypting data or generating and verifying HMAC codes

Asymmetric  
A public and private key pair used for encrypting and decrypting data or signing and verifying messages

**Key usage** [Help me choose](#)

Encrypt and decrypt  
Use the key only to encrypt and decrypt data.

Sign and verify  
Key pairs for digital signing  
Uses the private key for signing and the public key for verification.

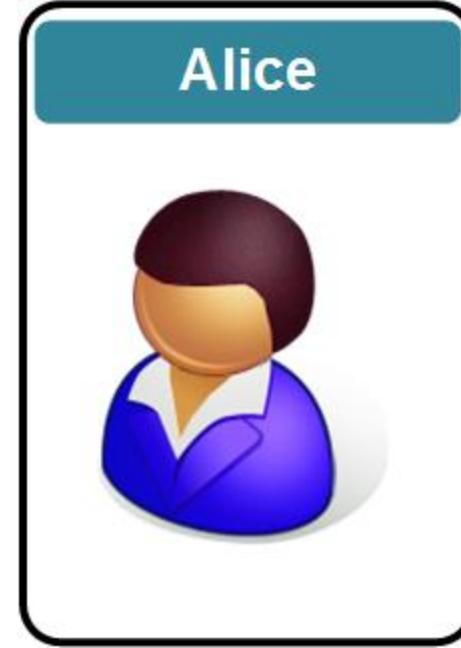
**Key spec** [Help me choose](#)

RSA\_2048  
 RSA\_3072  
 RSA\_4096  
 ECC\_NIST\_P256  
 ECC\_NIST\_P384  
 ECC\_NIST\_P521  
 ECC\_SECG\_P256K1

<https://asecuritysite.com/aws/lab09>



Bob



Alice

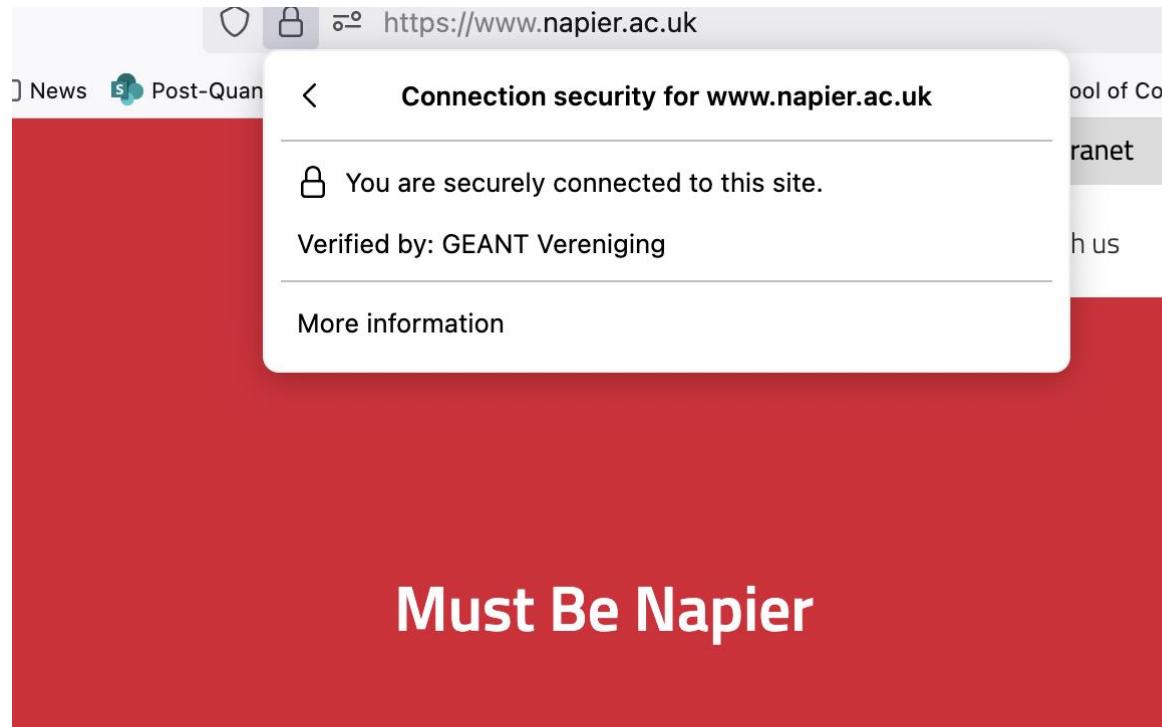
# Cryptography: Digital Certificates

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com/digitalcert>

Fundamentals  
Symmetric Key  
Hashing  
MAC  
KDF  
Public Key  
Key Exchange  
Signatures  
Digital Certificates

# Digital Certificates



## Certificate

[www.napier.ac.uk](https://www.napier.ac.uk)

GEANT OV RSA CA 4

USERTrust RSA Certification Authority

### Subject Name

Country: GB  
State/Province/County: Midlothian  
Organisation: Edinburgh Napier University  
Common Name: www.napier.ac.uk

### Issuer Name

Country: NL  
Organisation: GEANT Vereniging  
Common Name: [GEANT OV RSA CA 4](#)

### Validity

Not Before: Fri, 20 Sep 2024 00:00:00 GMT  
Not After: Sat, 20 Sep 2025 23:59:59 GMT

### Subject Alt Names

DNS Name: www.napier.ac.uk  
DNS Name: napier.ac.uk  
DNS Name: researchrepository.napier.ac.uk

### Public Key Info

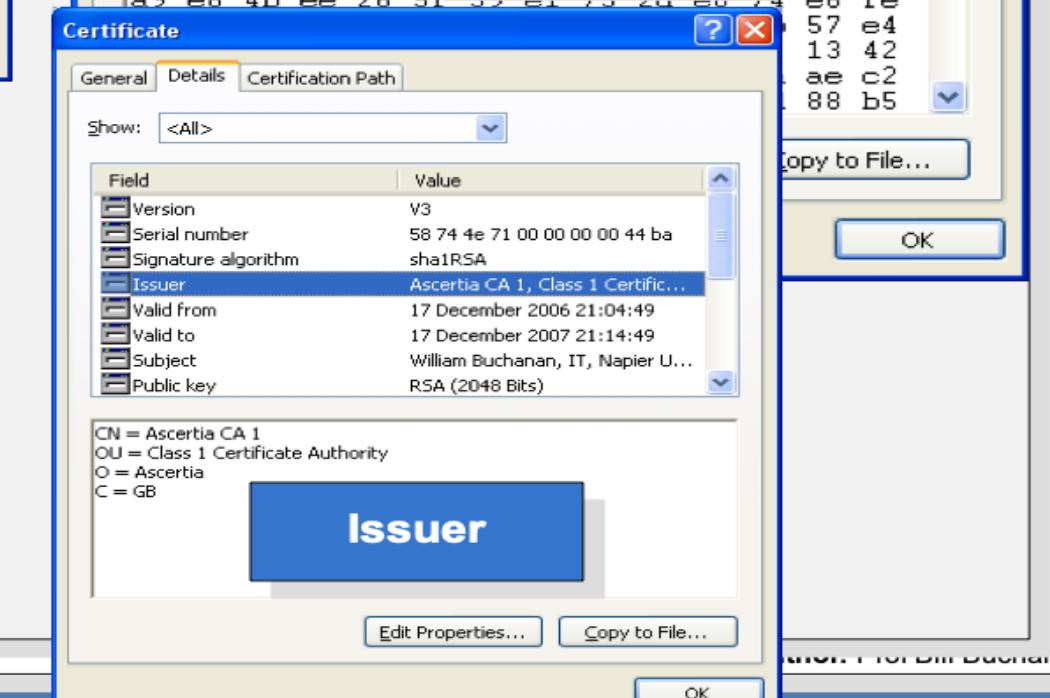
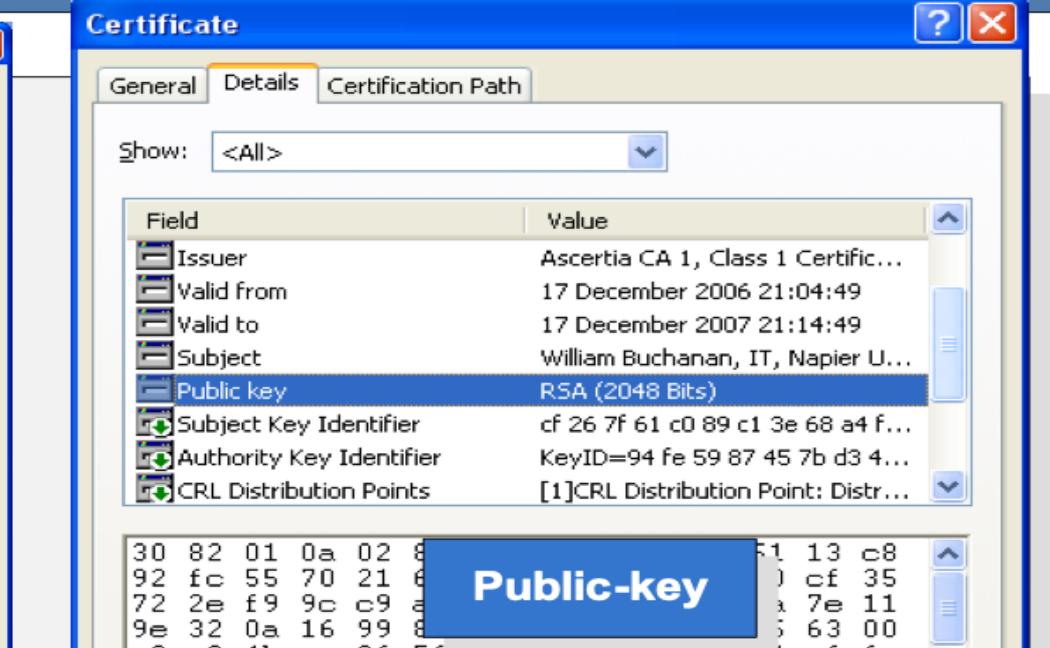
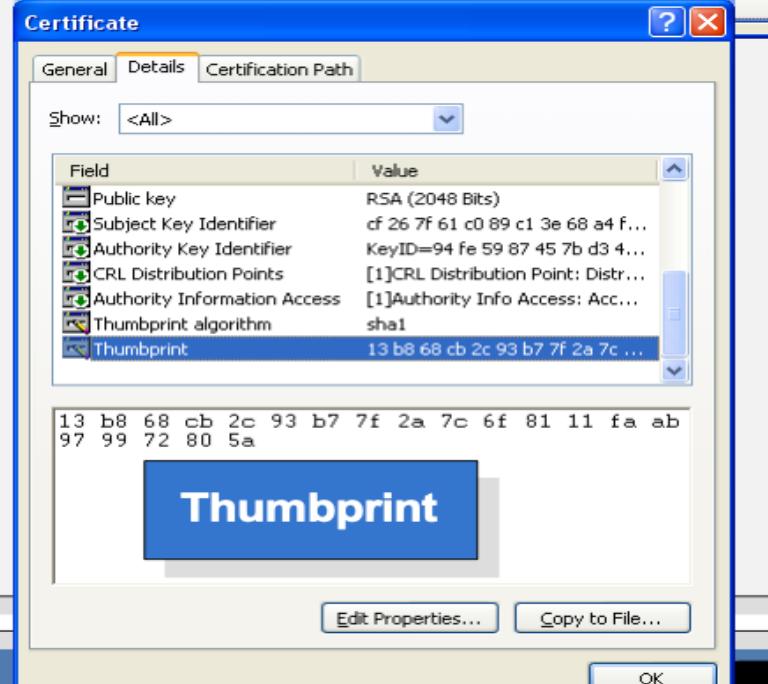
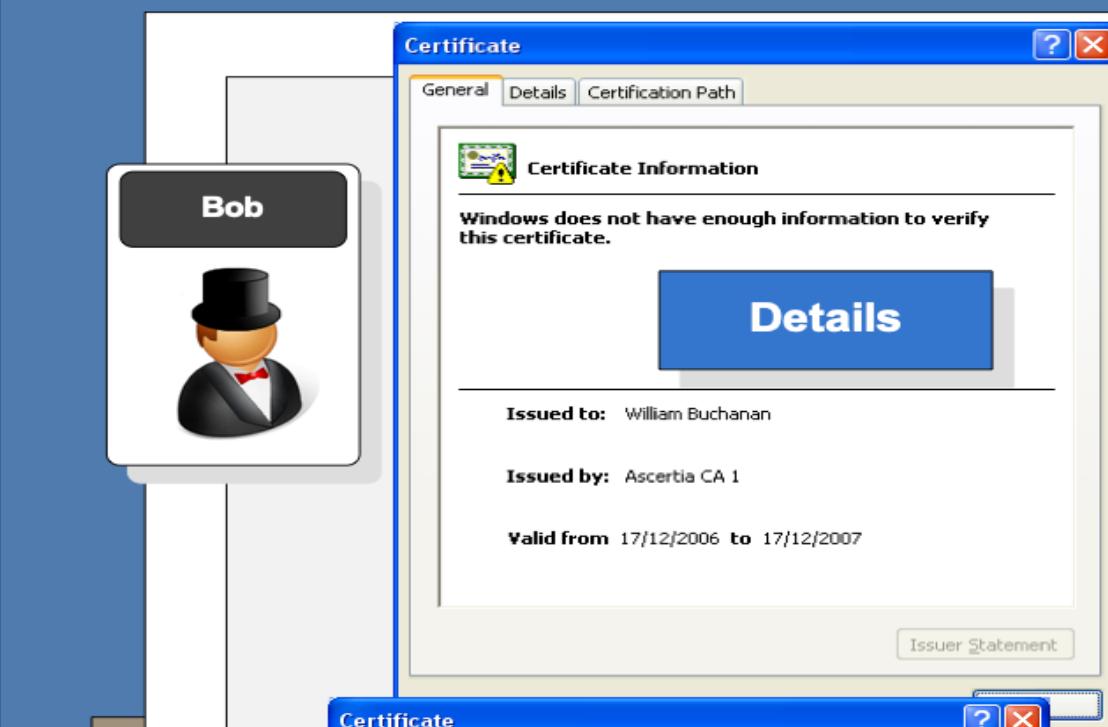
Algorithm: RSA  
Key Size: 2048  
Exponent: 65537  
Modulus: 88:5F:4B:B0:80:5F:37:65:E5:4B:CC:18:22:13:5C:24:0F:6A:FA:65:C0:87:C...

### Miscellaneous

Serial Number: 00:95:C1:B6:79:0A:BC:5E:B0:C8:9E:21:0B:6B:B0:A5:3E  
Signature Algorithm: SHA-384 with RSA Encryption  
Version: 3  
Download: [PEM \(cert\)](#) [PEM \(chain\)](#)

## Authentication

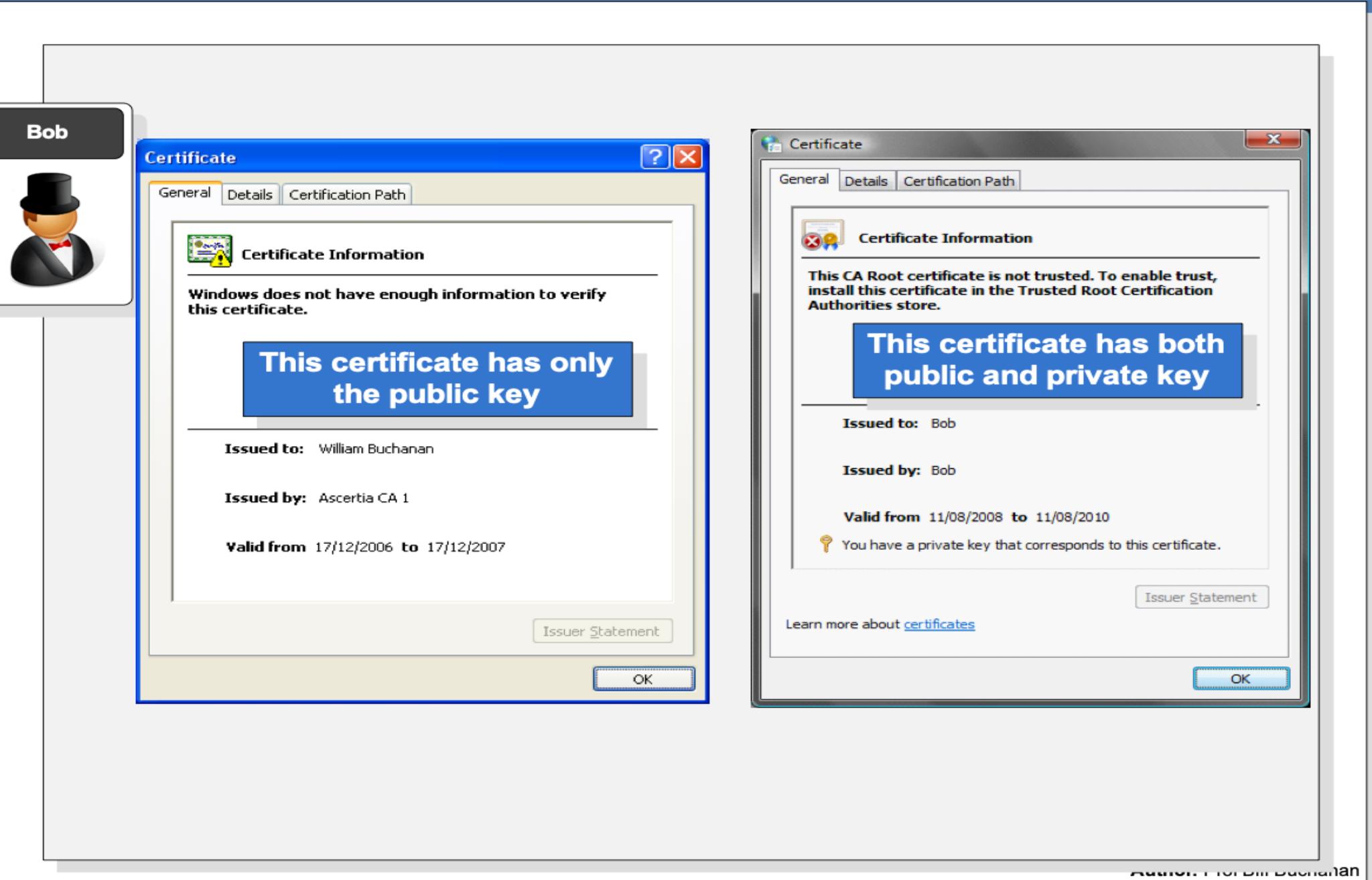
### Digital Cert.



Digital certificate contains a thumbprint to verify it

## Authentication

### Digital Cert.



Digital certificates should only be distributed with the public key

## Authentication

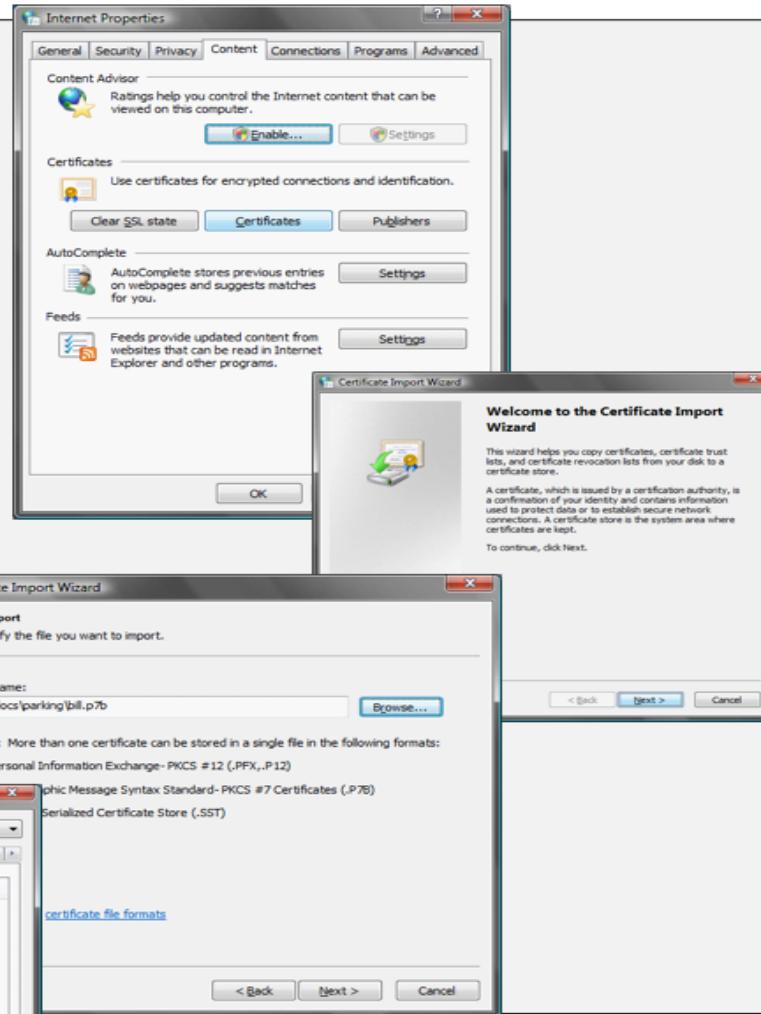
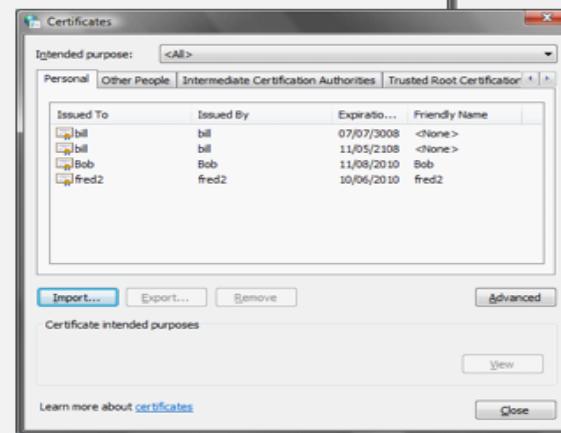
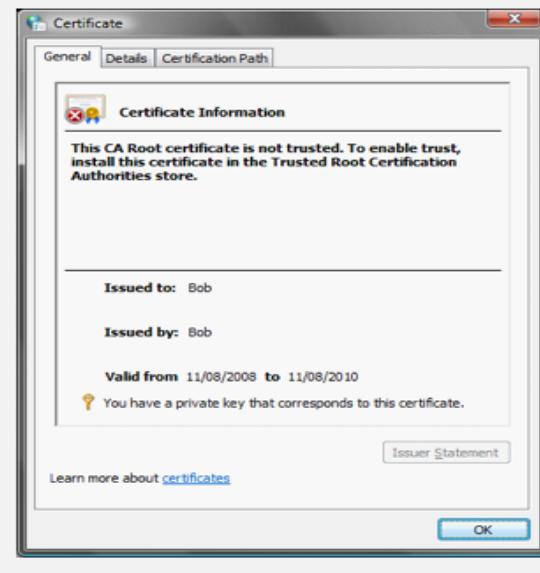
## Digital Cert.

### P7b format

Bob

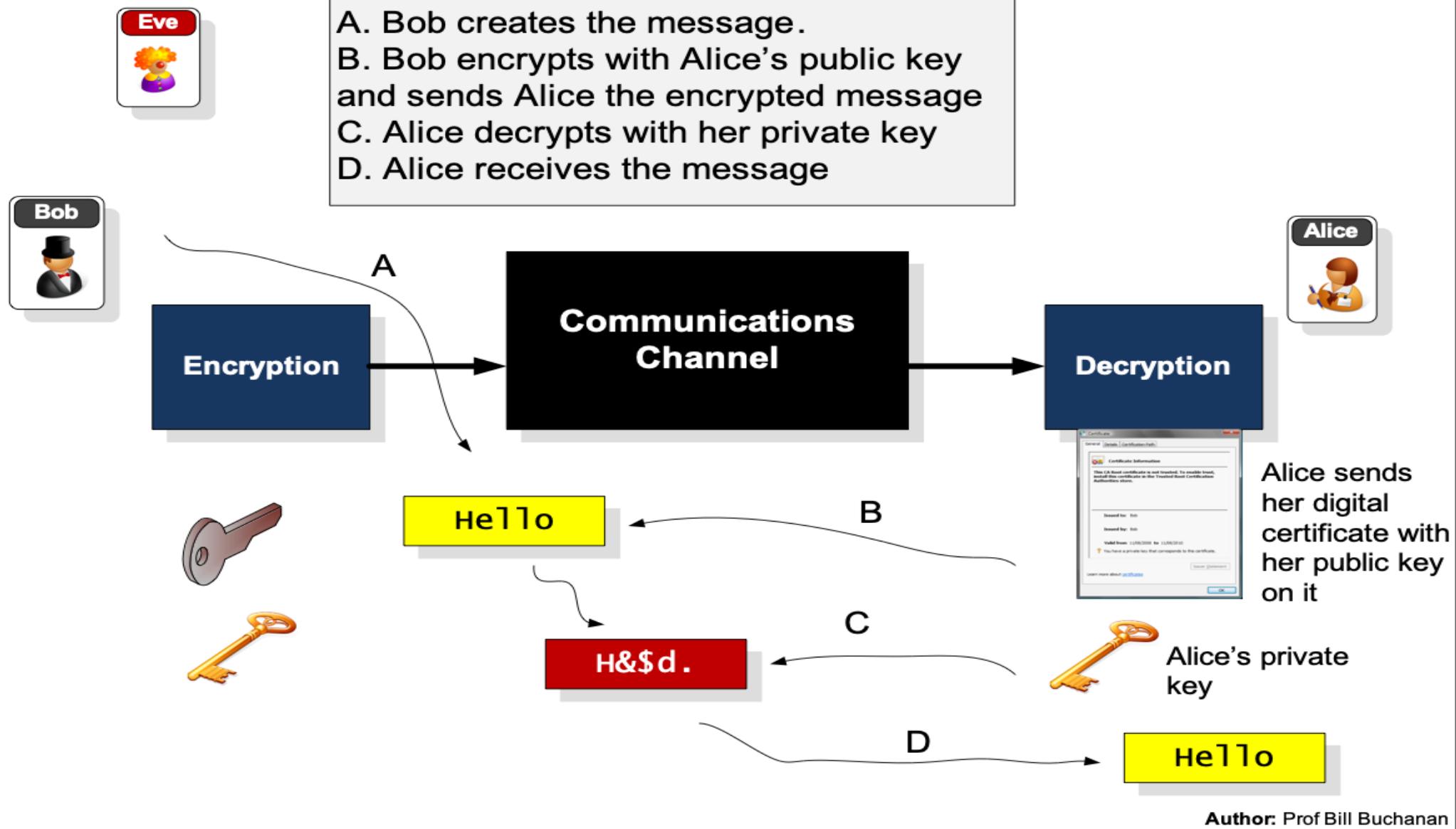


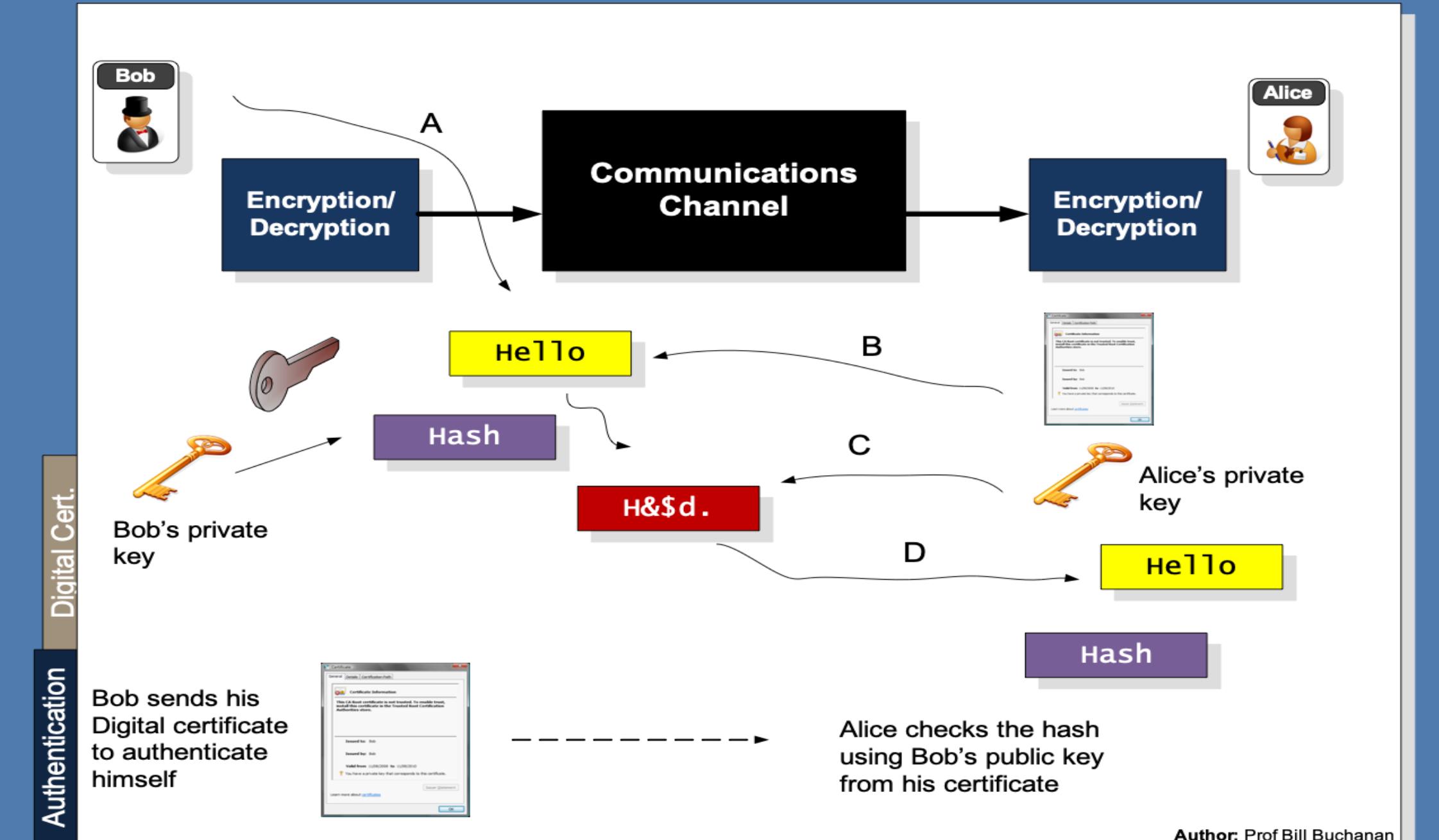
```
-----BEGIN CERTIFICATE-----
MIID2zCCA4WgAwIBAgIKWHR0cQAAAABEujANBgkqhkiG 9w0BAQUFADBgMQswCQYD
VQQGEwJHoJERMA 8GA1UEChMIQXnjZXJ 0awExJjAkBgNVBAsTHUNsYXNzIDEgQ 2Vy
dGlmawNhdGugQXV 0ag9yaXR5MRYWFAYDVQQDEw 1Bc2N1cnRpYSBDQSAXMB 4XDTA2
MTIxNzIxMDQ0OVoXTDA3MTIxNzIxMTQ 00VowgZ 8xjAkBgkqhkiG 9w0BCQEWf3cu
YnVjaGFuYW 5AbmFwaVjLmFjLnVrMQswCQYDVQQGEwJVSEQMA 4GA1UECBMHTG90
aG1hbjESMBAGA 1UEBxMjRWRpbmJ 1cmdoMRowGAYDVQQKExFOYXBpZXIgVW 5pdMVy
c210eTELMAkGA 1UECxMCsvQxGTAXBgNVBAMTEFdpbGxpYw 0gQnVjaGFuYW 4wggEi
MA0GCSqGSIb3DQEBAQAA 4IBDwAwggEKAoIBAQCVCFETYJL 8VXAhBEMRzQ10gM81
ci75nmsoamjZcb 6fhGeMgwMycoscmkrVjAkNoS +4mxznhcY3mdb+sZbwOvax
M5Foxhsrv+q86hsk8cDc+1lsqyJ8TQtufuDNs 0NFNY6tR6q7Cggq8/VjSxNqzK39
iLUF1ahhyCet /ab60/qwzL4ivsz2nmL4dyAuyi1hLPlvbppHGdE 6sdQxwYd0Cpfv
Zn7pauD5fqBESf06bukcIE147AzRMQj3kHuDt7Mexw7aoX+nXLP4wn7IamaxasF
qvhd0kyCZhYs 82JQDGatXRCqkk1ztzmw 5i6GkPsE7Xvux265wjQ5afhp2hY1AgMB
AAGjggEXMIEzAdBgNVHQ 4EFgQUzyZ/YccJwT5opPHPL1cQkko1kJwwYwYDVrOj
BFwwWoAUtP5zh0v700k6CorvRMWB9ifvkbmhP6Q9MDsxCzAJBgnVBAYTAkdCMREw
DwYDQQKEwhBc 2N1cnRpYTEZMBCGA 1UEAxMQQXnjZXJ 0awEgUm9vdCBQYIBDTBN
BgNVHR8ERjBEMEkQKA+hjxodHRwOi 8vd3d3LmFzY2VydG1hLmNvbs 9PbmxbmVD
QS9jcmxzL0FzY2VydG1hQ0ExL2NsYXNMS 5jcmwwPgYIKwYBBQUHAQEEMjAwMC 4G
CCsGAQUFBzAChiJodHRwOi 8vb2Nzcc5nbG9iYwx0cnVzdGZpbmRlci 5jb20vMA0G
CSqGSib 3DQEBBQUAA 0EATOCwGJ 1tS0ktlupmpjkml 81dxMmD5wuhszb1Gsmhpxi
H+vxhL9ya0w+Prpzy7ajS4/3xxU8vRAnhyU 9yU4qDA==
-----END CERTIFICATE-----
```

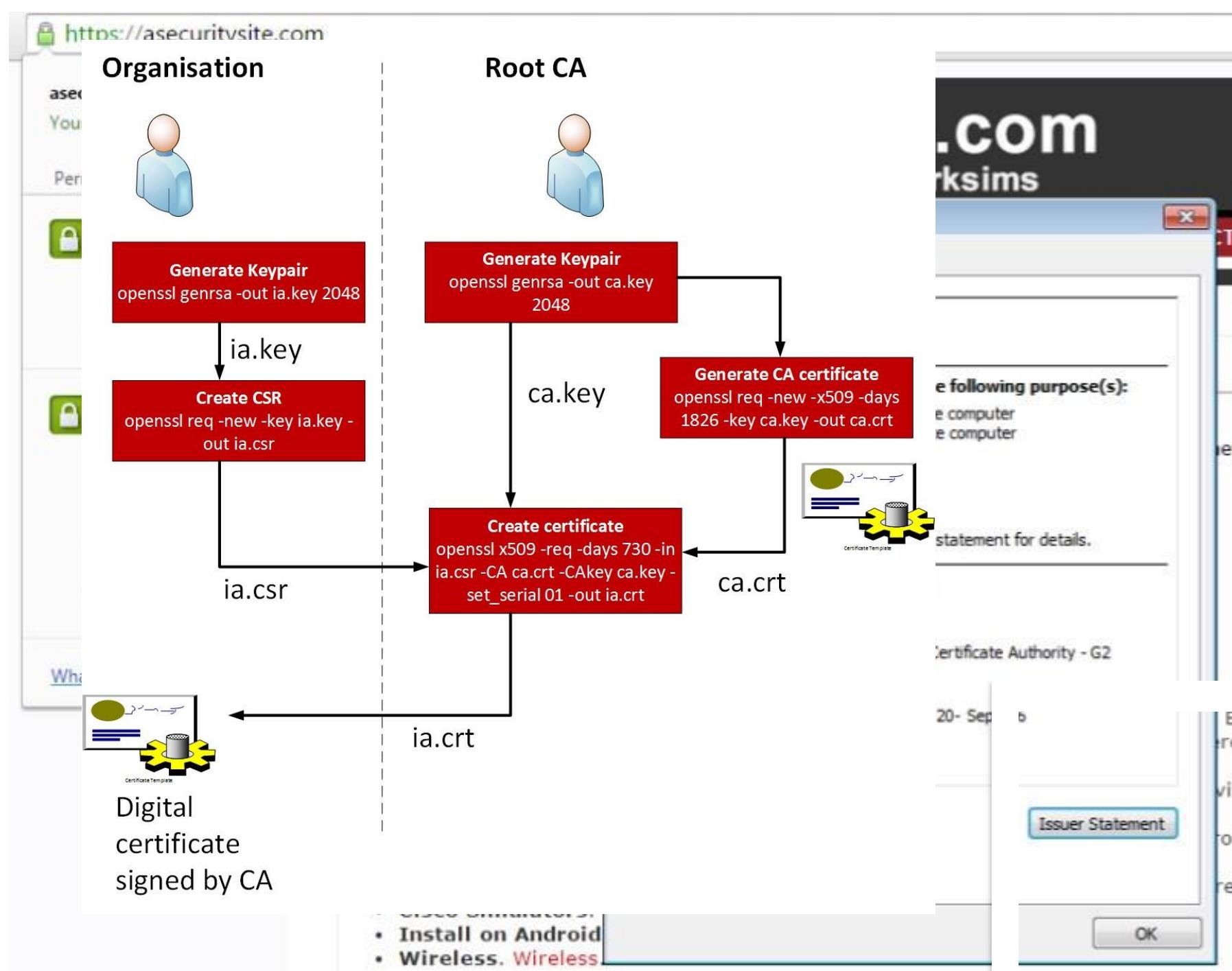


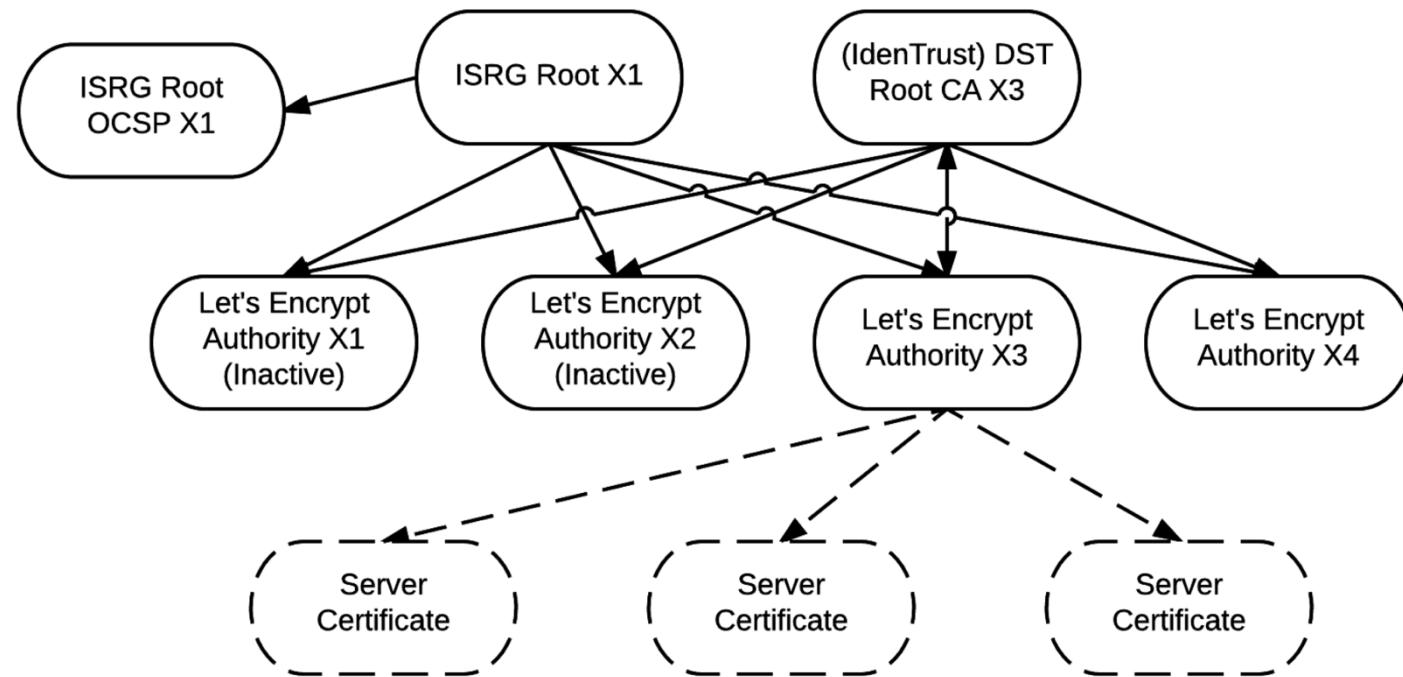
- The main certificate formats include:**
- P7b. Text format
  - PFX/P12. Binary.
  - SST. Binary.

**Digital certificates should only be distributed with the public key**









-----BEGIN CERTIFICATE-----

MIIEwTCCA6mgAwIBAgIRAO93GGFLfHwOC... hvcNAQELBQAw  
 QjELMAkGA1UEBhMCVVMxHjAcBgNVBAoTFUdvb2dsZSBUcnVzdCBTZXJ2aWNlczET  
 MBEGA1UEAxMKR1RTIENBIDFPMTAeFw0yMDAyMTIxMTQ3NDFaFw0yMDA1MDYxMTQ3  
 NDFaMGgxCzAJBgNVBAYTAIVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQH  
 Ew1Nb3VudGFpbIBWaWV3MRMwEQYDVQQKEwpHb29nbGUgTExDMRcwFQYDVQQDEw53  
 d3cuZ29vZ2xILmNvbTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABCzzLJYFJb8W  
 TpQxWLOTySDvfEAVjKMc7181CFsp1tCtn13YiLn1TM22sUwPycyogKjBSaQZ9Axi

PEM

- X.509 Certificate (DER)
- X.509 Certificate (PKCS#7)

PKCS#1 v2 - Padding for Public Key

PKCS #7 v1.5 - Cryptography Message Syntax

PKCS 10 v1.7 - Certificate Request Standard.

tIHtqVqlTZPCAI/ANvbHdZiMEePB8eA+oTiw2ucChqLIsop5Mio8ckg7aXG4/QfC  
 AIDbvkokF44rs4UEpsaqGe9QqMjjTuO7ZCzrFd1m3geq2ARKPHgPoPM6UbDdqgO  
 TNQo9C0F5ZlOkOgV/qshvpTO4YzE7TB2U571eYEqNXH48syVx8Xsk3P7fjM7FIZ2  
 IzbJSEipZjm8DsP10fFXpToIn+zK

-----END CERTIFICATE-----

-----BEGIN PKCS7-----

MII8gYJKoZlhvcNAQcCollE4zCCBN8CAQ... FMIIE  
 wTCCA6mgAwIBAgIRAO93GGFLfHwOCAAAAAAucZgwDQYJKoZlhvcNAQELBQAwQjEL  
 MAkGA1UEBhMCVVMxHjAcBgNVBAoTFUdvb2dsZSBUcnVzdCBTZXJ2aWNlczETMBEG  
 A1UEAxMKR1RTIENBIDFPMTAeFw0yMDAyMTIxMTQ3NDFaFw0yMDA1MDYxMTQ3NDFa  
 MGgxCzAJBgNVBAYTAIVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1N  
 b3VudGFpbIBWaWV3MRMwEQYDVQQKEwpHb29nbGUgTExDMRcwFQYDVQQDEw53d3cu  
 Z29vZ2xILmNvbTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABCzzLJYFJb8WTpQx  
 WL0TySDvfEWCKMe7181CFspT60kn13YLNdTM22sUwPycyogKjBSaQZ9AxihCeG  
 EAiJkbejggJVMICUTAOBgNVHQ8BAf8EBAMCB4AwEwYDVR0IBAwvCgYIKwYBBQUH  
 AwEwDAYDVROTAQH/BAIwADAdBgNVHQ4EFgQU9Ty3t9o07UW9+Hc6kv/j9511Bbw  
 HwYDVR0jBBgwFoAUmNH4bhDrz5vsYJ8YkBug630J/SswZAYIKwYBBQUHAQEEWDBW  
 MCcGCCsGAQUFBzABhhtodHRwOi8vb2NzcC5wa2kuZ29vZy9ndHMxbzEwKwYIKwYB  
 BQUHMAKGh2h0dHA6Ly9wa2kuZ29vZy9n3lyL0duUzFPMS5jcnQwGQYDVR0RBBiW  
 EIId3d3Lmdvb2dsZ5jb20wIqYDVROgBBowGDAIbgZngQwBAlgwDAYKwYBBAHW  
 eQIFAzAvBgnVHR8EKDAmMCsgIqAghh5odHRwOi8vY3jsLnBraS5nb29nL0dUUZFP  
 MS5jcmwwggEFBgorBgEEAdZ5AgQCBIH2B1HzAPEAdgCyHgXMi6LNiiBOh2b5K7mK  
 JSBna9r6cOeySVMt74uQXgAAAXA5cNqwAAAEAwBHMEUCIQCojKtzOe8i1JYKHmKn  
 bt1puqT4AE3peMVAVk/WewGp1QlgBvRMbNkwF6i8+jVv3CfTHKvFd8e00+GYPbxJ  
 Zb1drKEAdwBep3P531bA57U2SH3QSeAyepGaDISHhKEGHWWgXFFWAAXAXA5cNrL  
 AAAAEAwBIMEYCIQCQSONppELPODh65oFT5ZAGsSQz4FsjNNZedgS7WqzLnQhAMAr  
 u0QUuoE3RWngfhI6W7n1kxxEmd0vSZTe6+Ql+JuVMA0GCSqGSIb3DQEBCwUAA4IB  
 AQCDU8CVusGstVklAmrtg3DyYhNV1UnKyLk3RrHkumwYz5mC25bWGoLVKvvpxb  
 N3zA329/9JZq0mn0vmPMxjTDvh9sVQ7g4znwha/KJfzL8AZKUldD9O+hDOtIHt  
 qVqlTZPCAI/ANvbHdZiMEePB8eA+oTiw2ucChqLIsop5Mio8ckg7aXG4/QfCAIdb  
 vkoffK44rs4UEpsaqGe9QqMjjTuO7ZCzrFd1m3geq2ARKPHgPoPM6UbDdqgOTNQo  
 9C0F5ZlOkOgV/qshvpTO4YzE7TB2U571eYEqNXH48syVx8Xsk3P7fjM7FIZ2zbJ  
 SEipZjm8DsP10fFXpToIn+zKoQAxAA==  
 -----END PKCS7-----

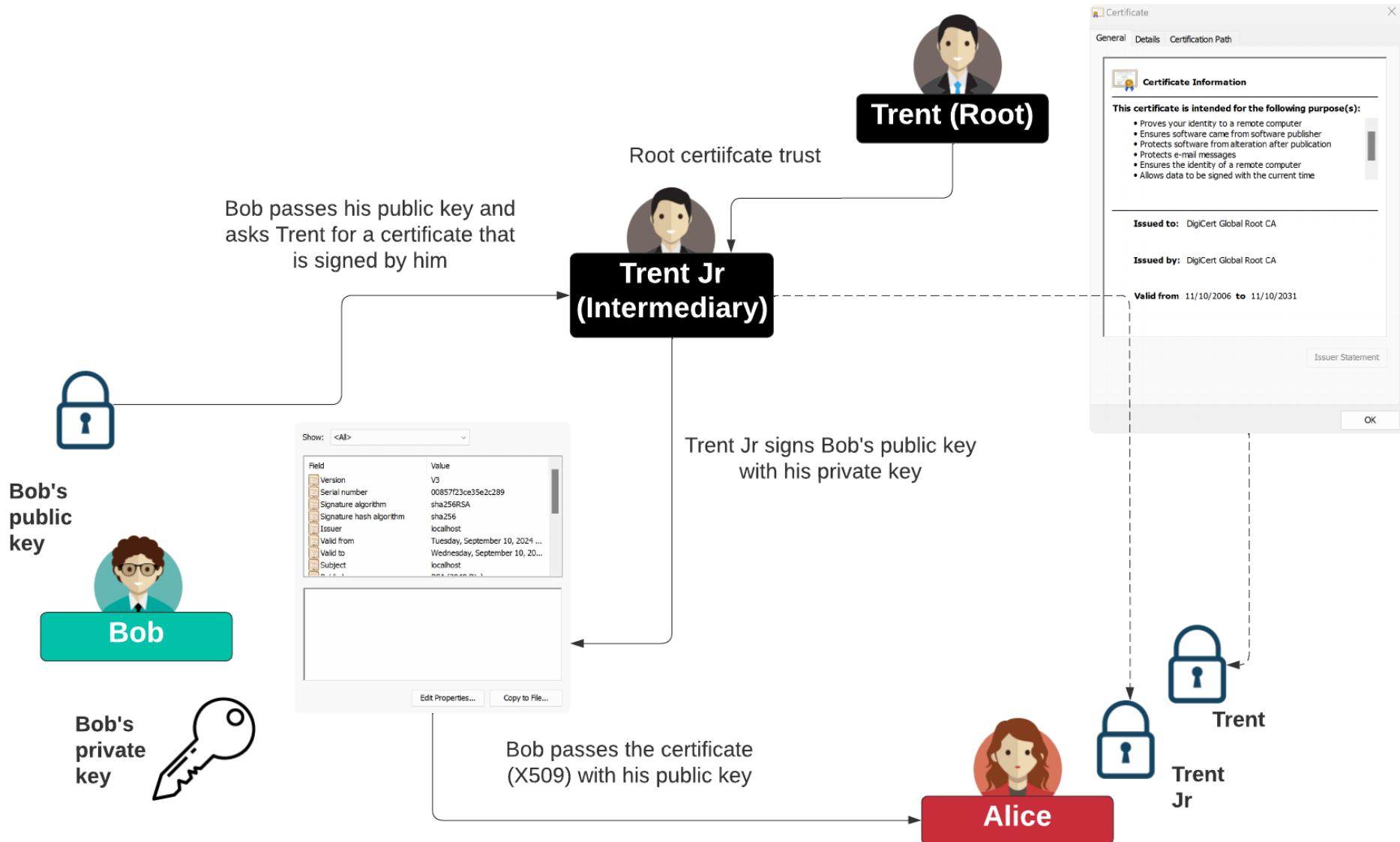
Base64

```
openssl x509 -outform der -in www-google-com.pem -out google.crt
openssl pkcs12 -export -in server.pem -out keystore.pkcs12
```

Binary

DER  
CER

# Digital Certificates



## Certificates

Intended purpose: <All>

Intermediate Certification Authorities Trusted Root Certification Authorities Trusted Publ

Issued To	Issued By	Expiration...	Friendly
Microsoft Authenticode(tm)...	Microsoft Authenticode(tm)...	31/12/1999	Microsoft
Microsoft Root Authority	Microsoft Root Authority	31/12/2020	Microsoft
Microsoft Root Certificate ...	Microsoft Root Certificate ...	09/05/2021	Microsoft
NetLock Expressz (Class C...	NetLock Expressz (Class C...	20/02/2019	NetLock I...
NetLock Kozieggyzoi (Class ...	NetLock Kozieggyzoi (Class ...	19/02/2019	NetLock I...
NetLock Uzleti (Class B) Ta...	NetLock Uzleti (Class B) Ta...	20/02/2019	NetLock I...
NO LIABILITY ACCEPTED, (...	NO LIABILITY ACCEPTED, (...	07/01/2004	VeriSign
PTT Post Root CA	PTT Post Root CA	26/06/2019	KeyMail F...

Import...

Export...

Remove

Advanced...

Certificate intended purposes

<All>

**Trusted Root CA**  
- always trusted

## Trusted Root CA



### Certificate purposes:

- Secure email.
- Server authentication.
- Code signing.
- Driver authentication.
- Time stamping.
- Client authentication.
- IP tunnelling.
- EFS (Encrypted File System).

## Certificate

General Details Certification Path

### Certificate Information

This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.

**Self signed**  
- Can never be trusted

Issued to: William Buchanan

Issued by: William Buchanan

Valid from 22/02/2007 to 29/01/2107

You have a private key that corresponds to this certificate.

Issuer Statement

OK



## Trust2



**Intermediate CA**  
- Can be trusted for some things

## Certificates

Intended purpose: <All>

Intermediate Certification Authorities Trusted Root Certification Authorities Trusted Publ

Issued To	Issued By	Expiration...	Friendly
GTE CyberTrust Root	Root SGC Authority	23/02/2006	<N
Microsoft Internet Authority	GTE CyberTrust Global Root	23/02/2007	<N
Microsoft Internet Authority	GTE CyberTrust Global Root	19/04/2009	<N
Microsoft Secure Server Authority	Microsoft Internet Authority	23/02/2007	<N
Microsoft Secure Server Authority	Microsoft Internet Authority	19/04/2009	<N
Microsoft Windows Hardware C...	Microsoft Root Authority	31/12/2002	<N
Microsoft Windows Hardware C...	Microsoft Root Authority	31/12/2002	<N
MS SGC Authority	Root SGC Authority	01/01/2010	<N

Import... Export... Remove Advanced...

Certificate intended purposes

Signing, Windows Hardware Driver Verification

View

Close

chanan

**Levels of trust**

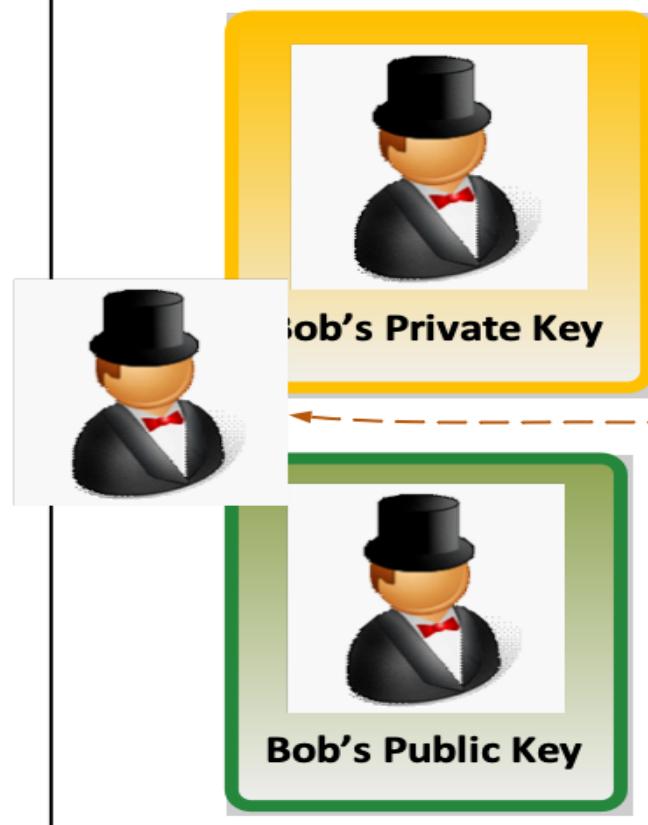


## Public key encryption ... secret ... identity ... trust



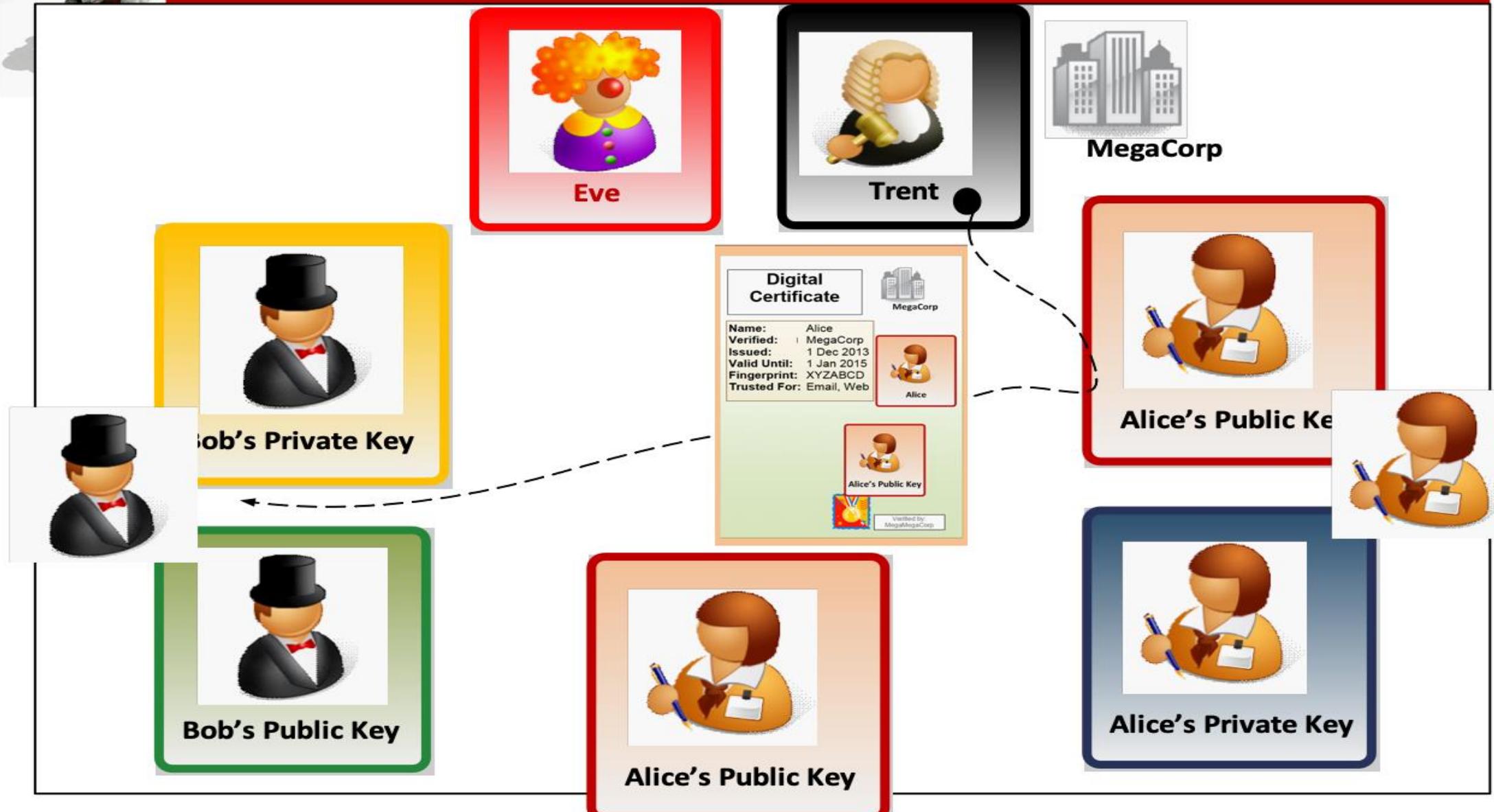


## Public key encryption ... secret ... identity ... trust





# Public key encryption ... secret ... identity ... trust

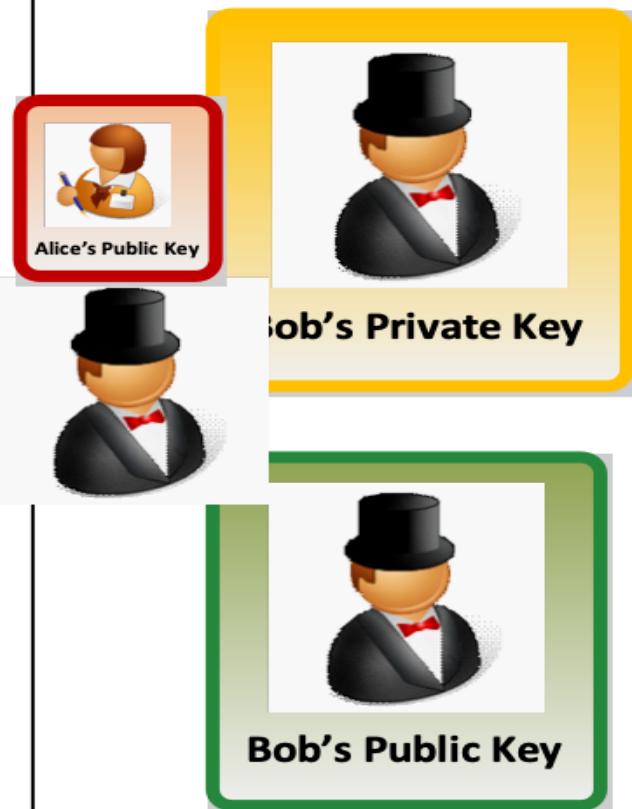




## Public key encryption ... secret ... identity ... trust



MegaCorp



Hello Alice,  
Wish you were  
here!  
- Bob

Bob.



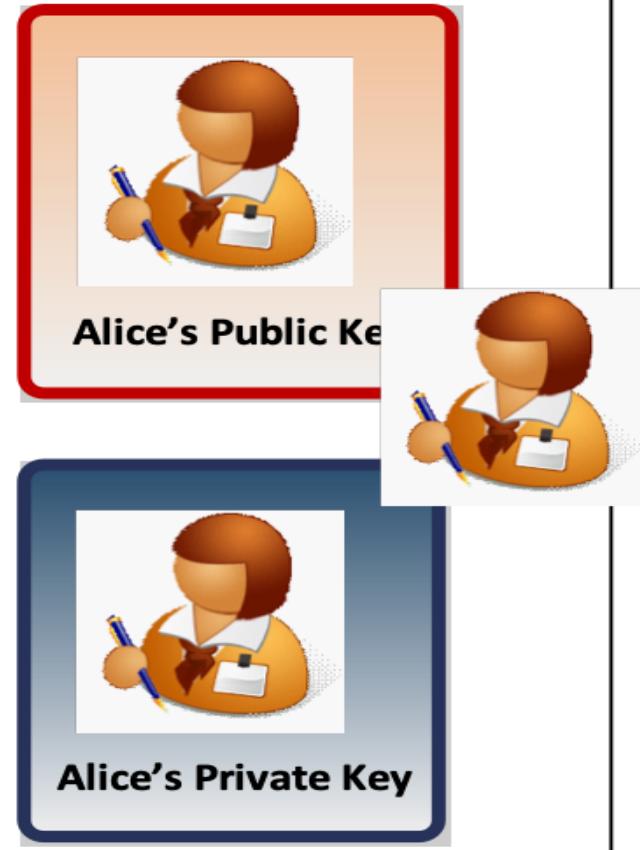
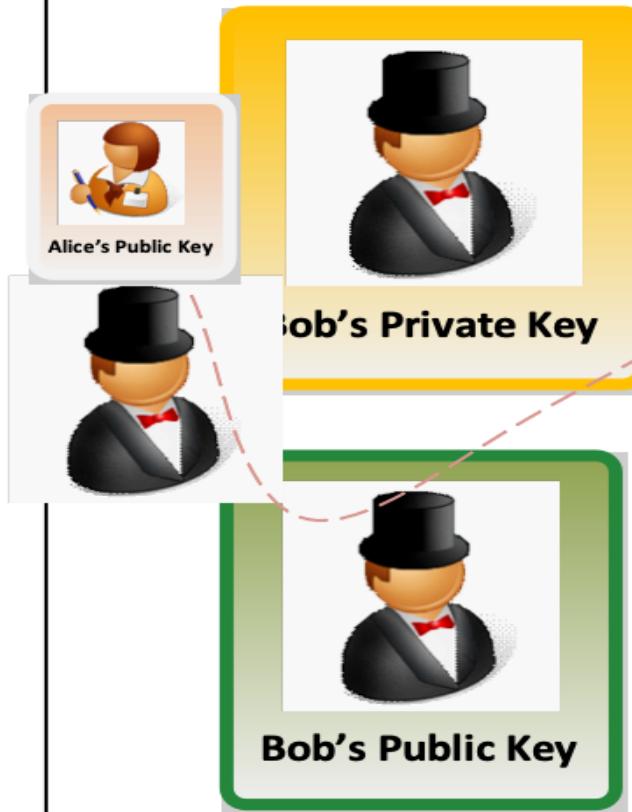


# Public key encryption ... secret ... identity ... trust





# Public key encryption ... secret ... identity ... trust





## Public key encryption ... secret ... identity ... trust



Eve



Trent



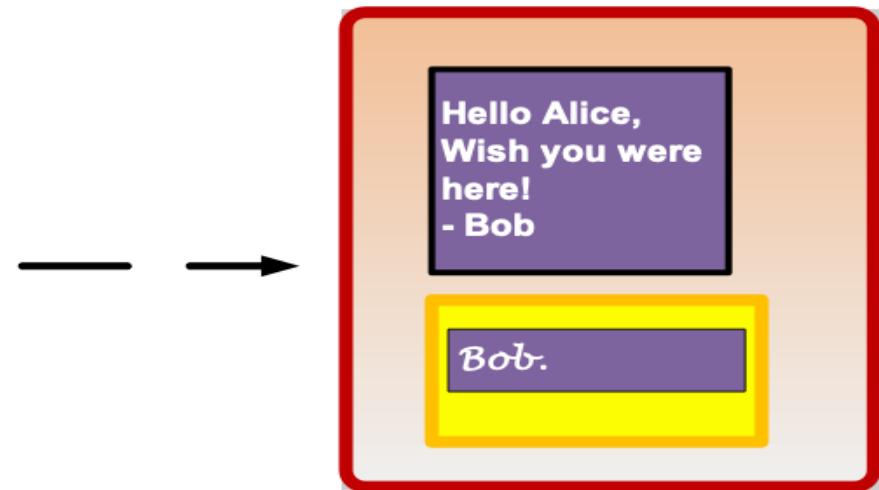
MegaCorp



Bob's Private Key



Bob's Public Key



Alice's Public Key



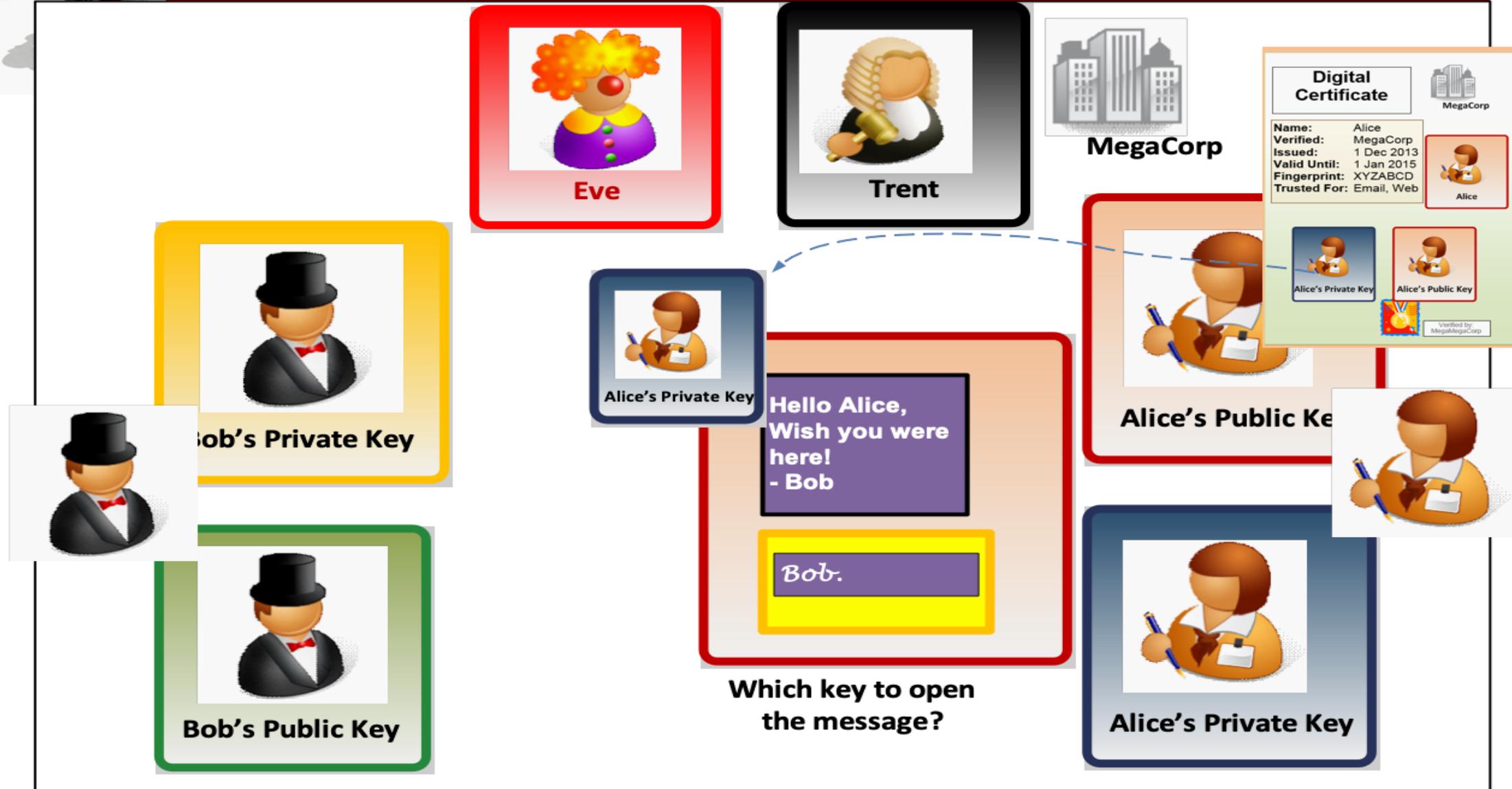
Alice's Private Key



Which key to open  
the message?

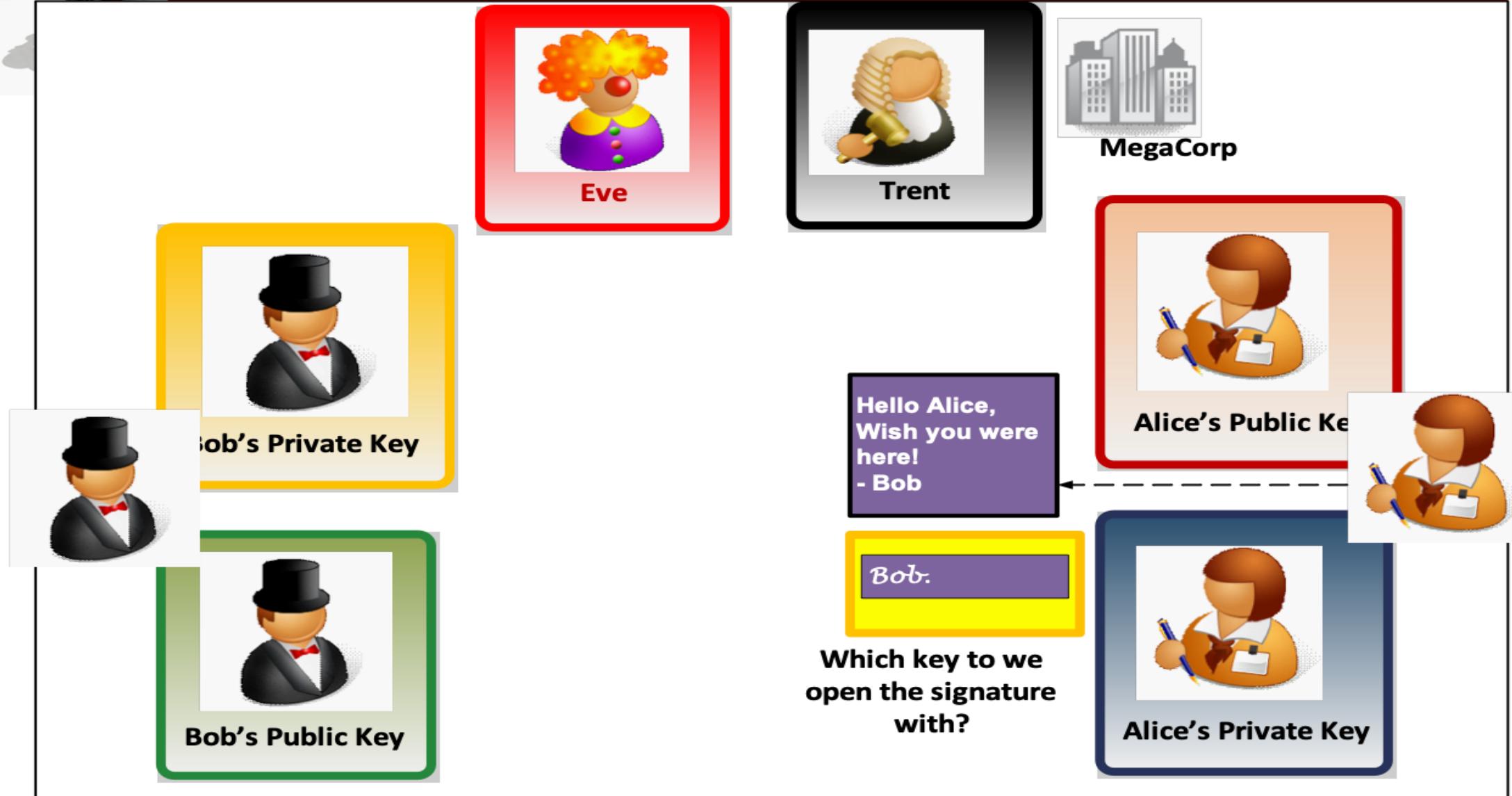


# Public key encryption ... secret ... identity ... trust



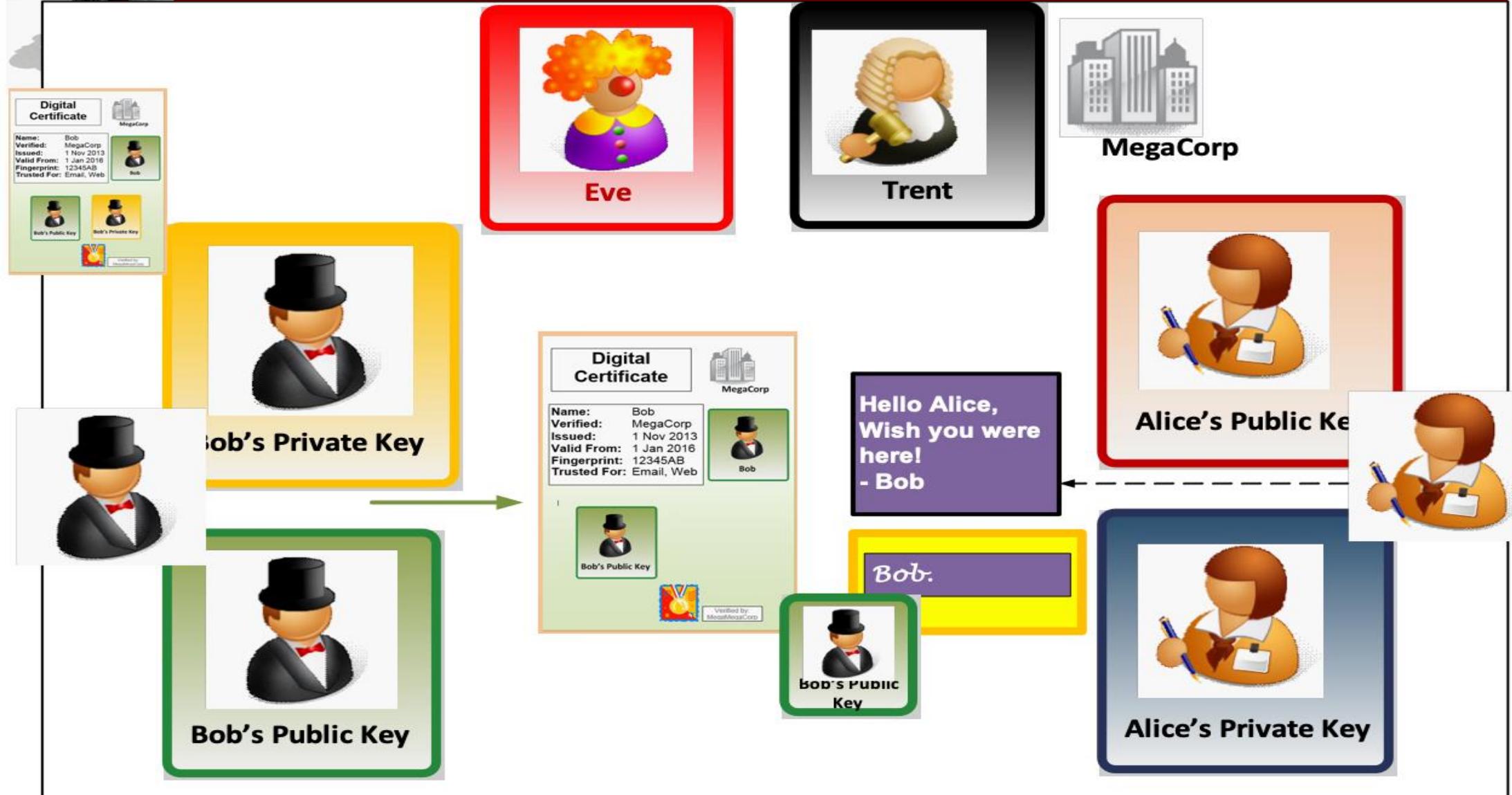


# Public key encryption ... secret ... identity ... trust



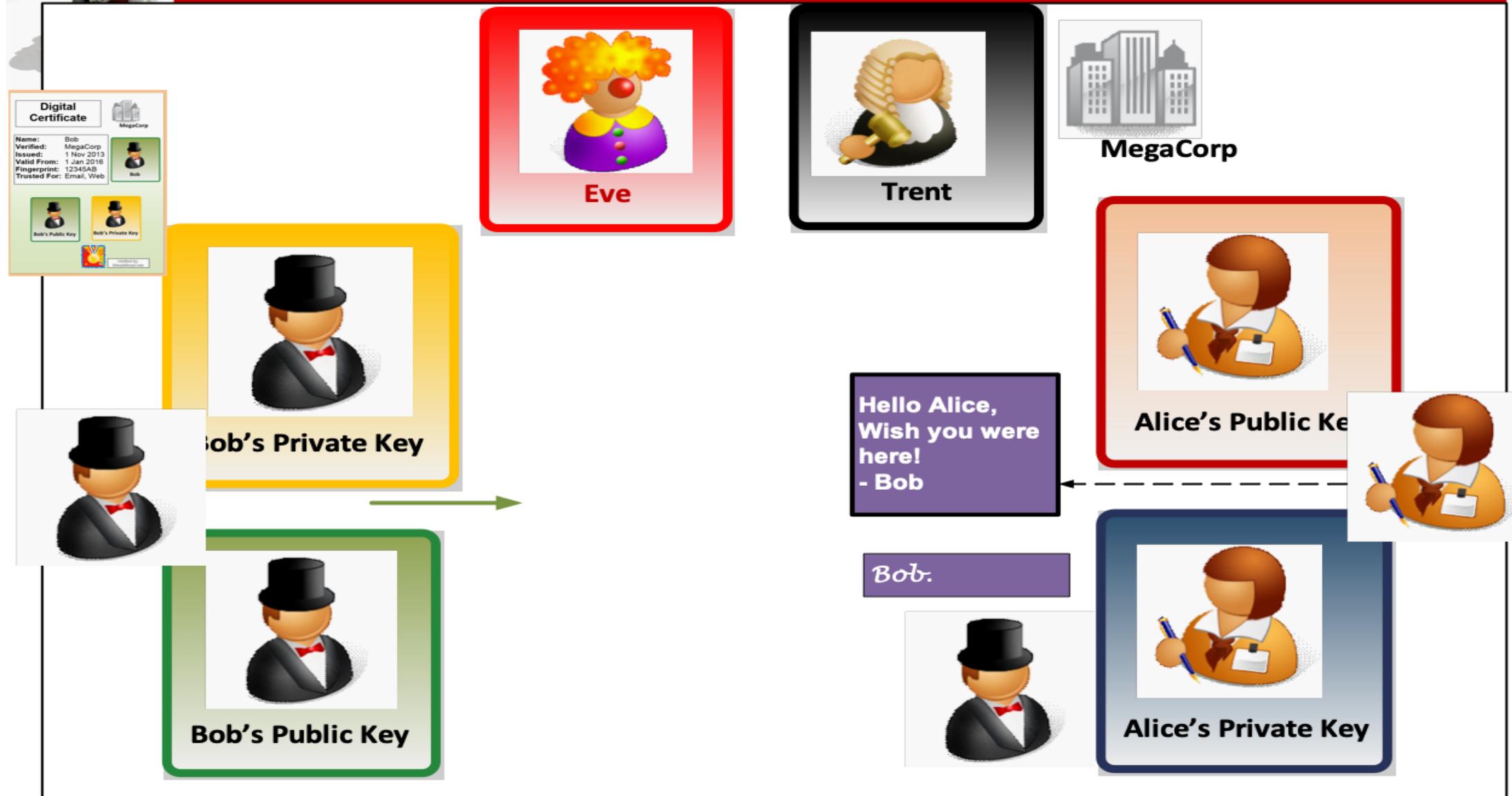


# Public key encryption ... secret ... identity ... trust

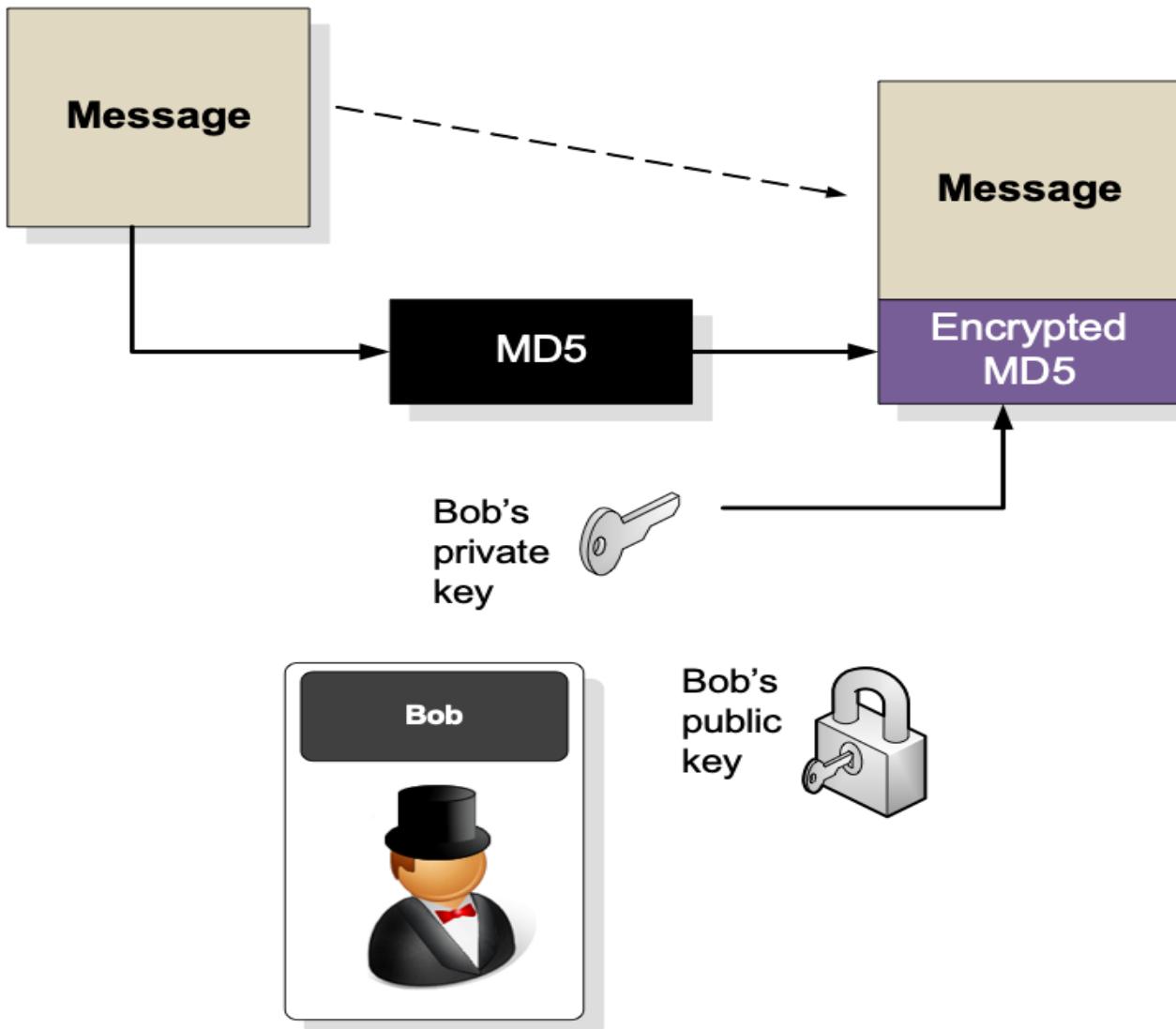




# Public key encryption ... secret ... identity ... trust



Authentication      The magic private key

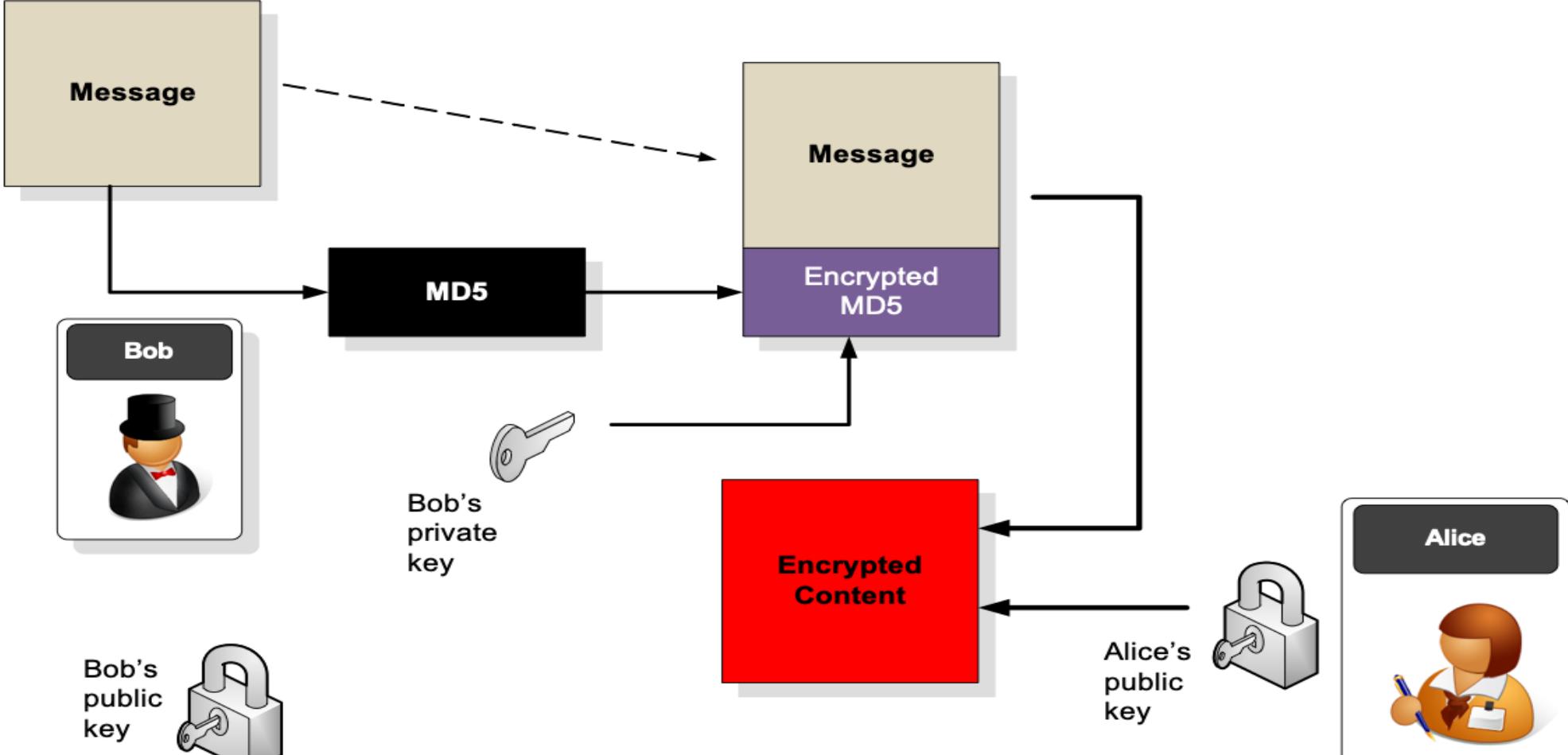


Author: Prof Bill Buchanan

Using Bob's private key to authenticate himself

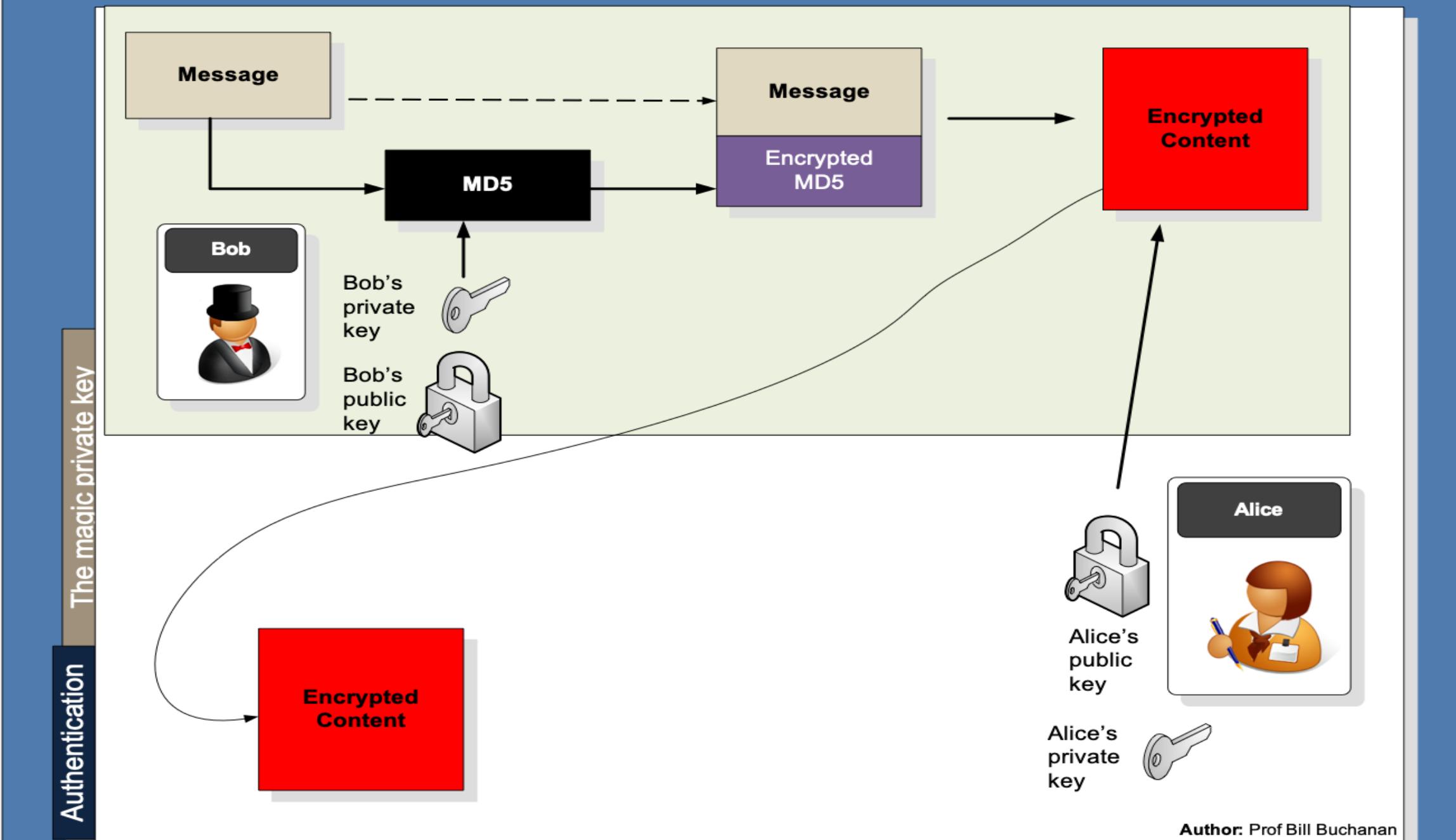
## Authentication

### The magic private key

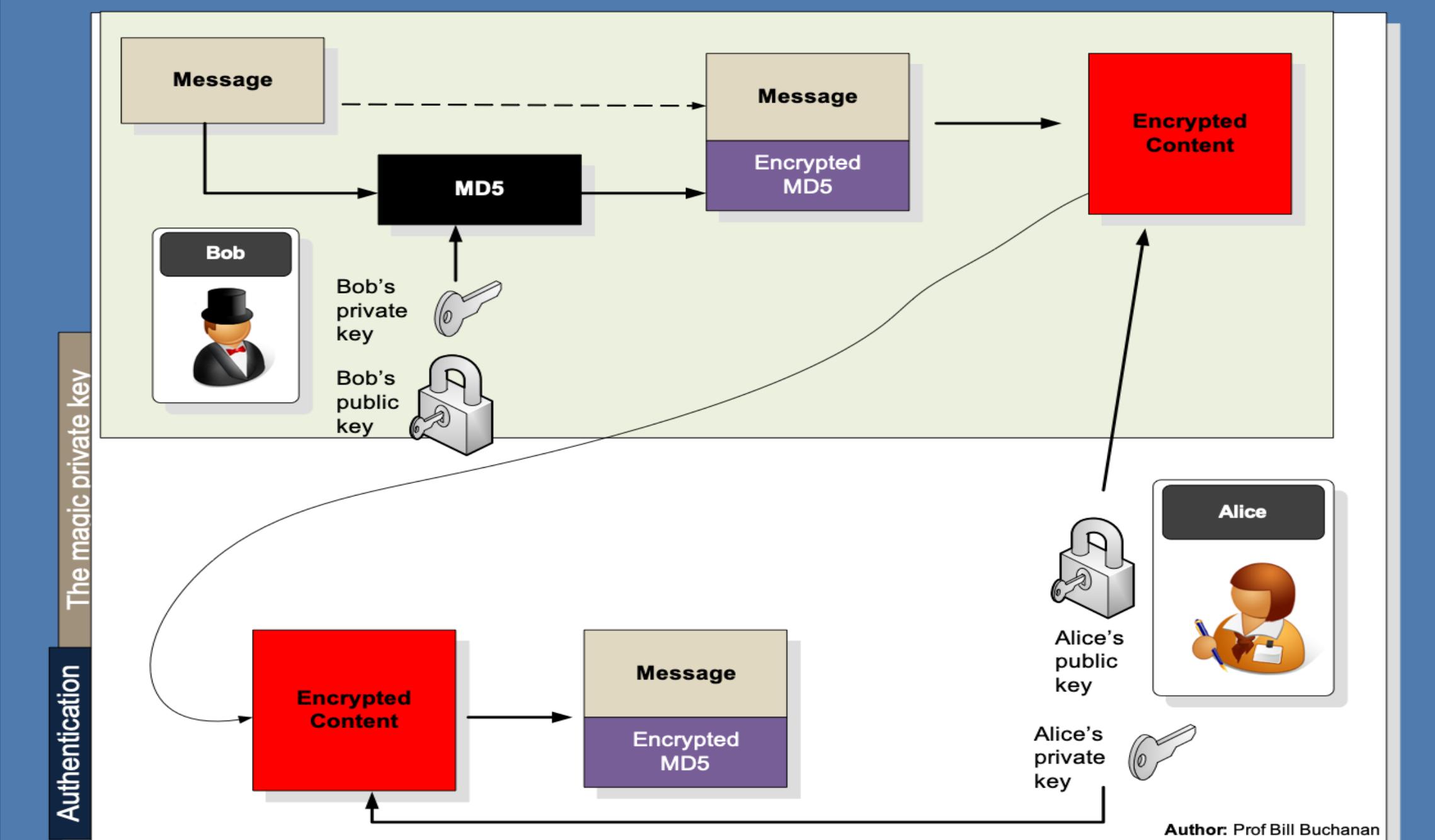


Author: Prof Bill Buchanan

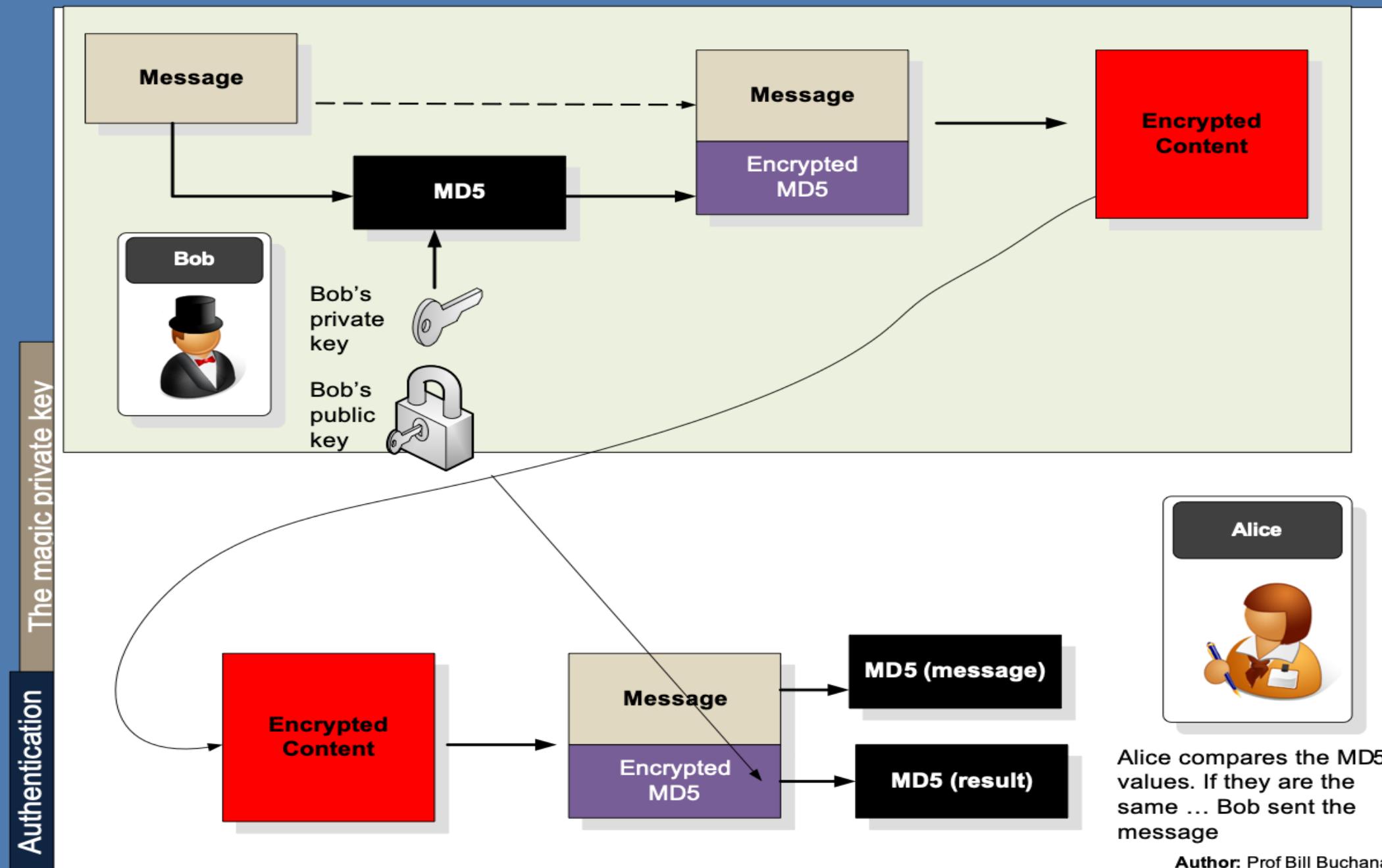
**Bob encrypts the message/hash with Alice's public key**



**Bob encrypts the message/hash with Alice's public key**

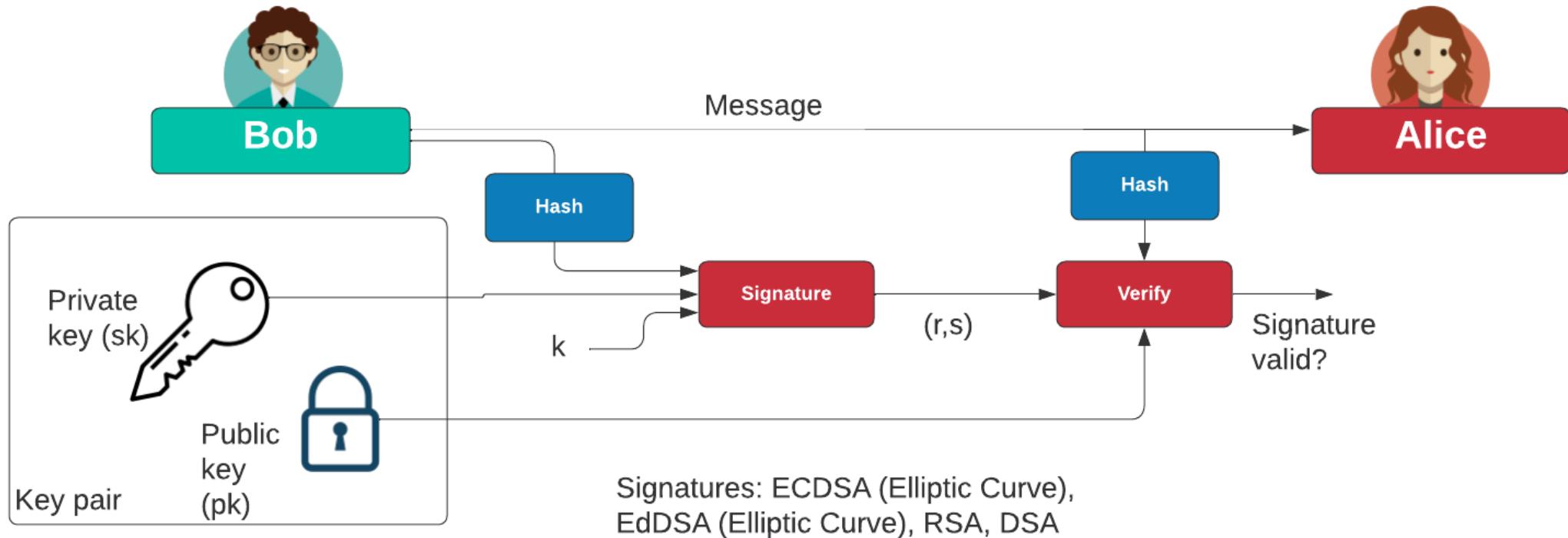


Alice decrypts the message

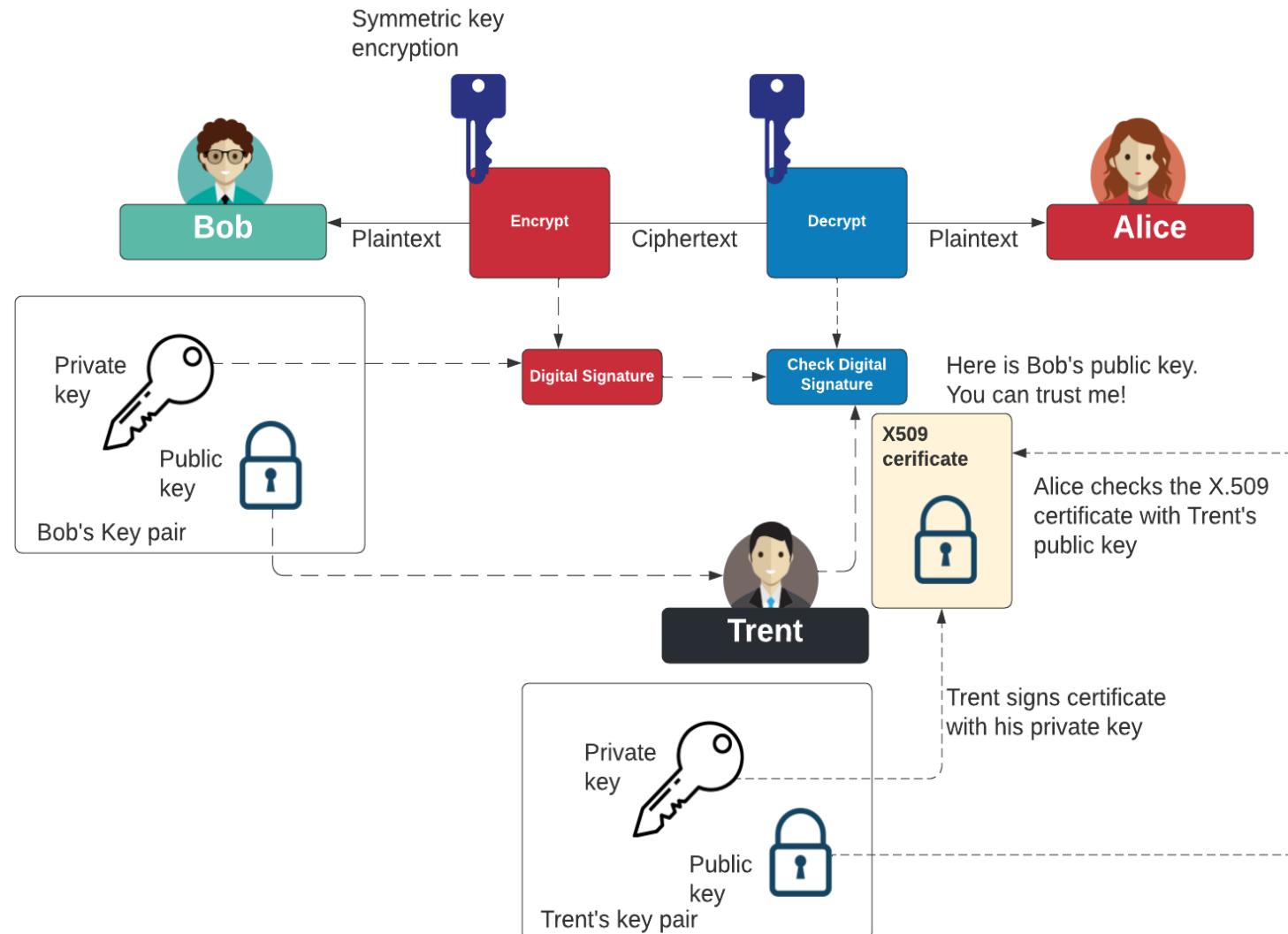


**Alice decrypts the message**

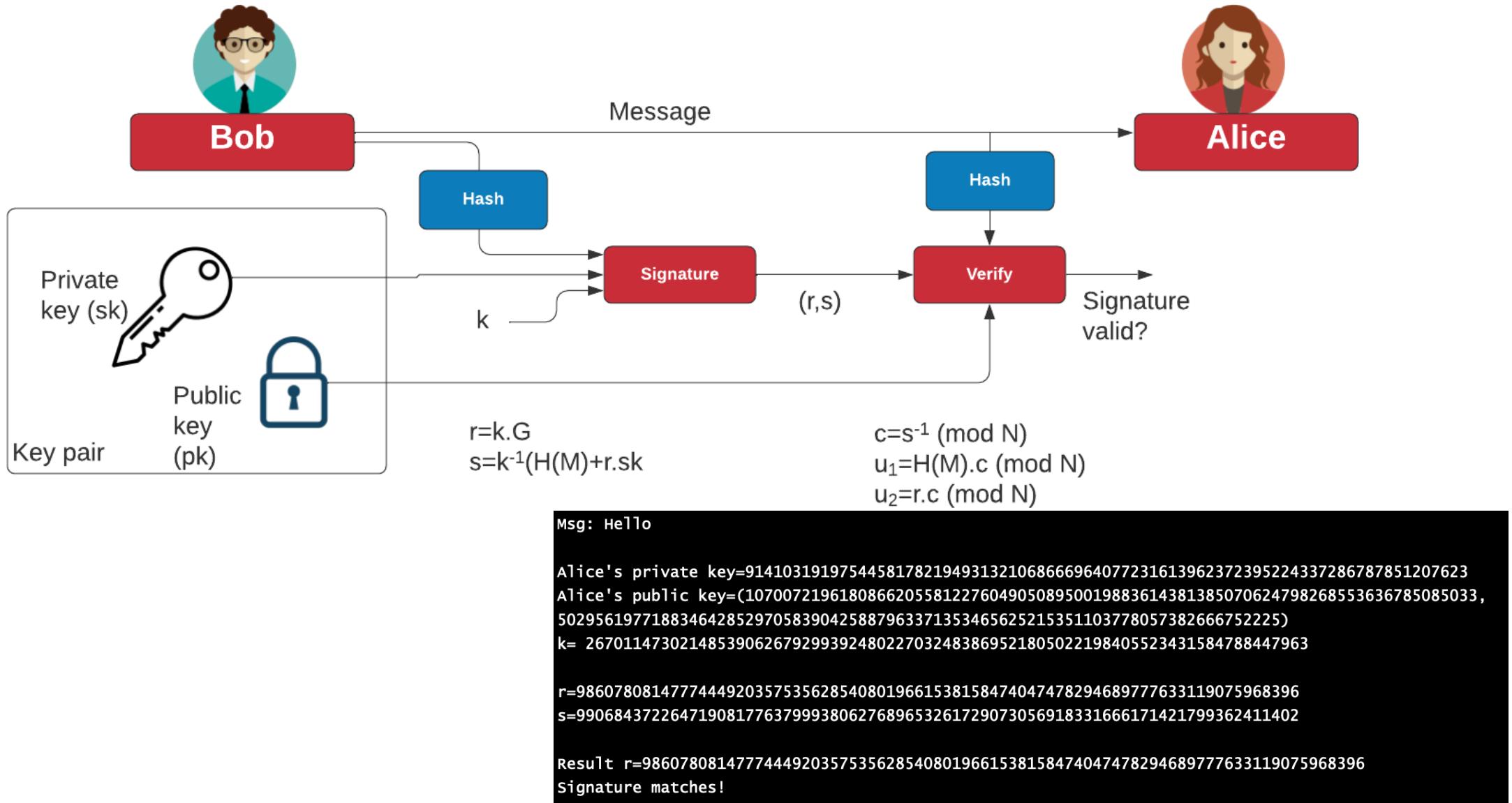
# Overview



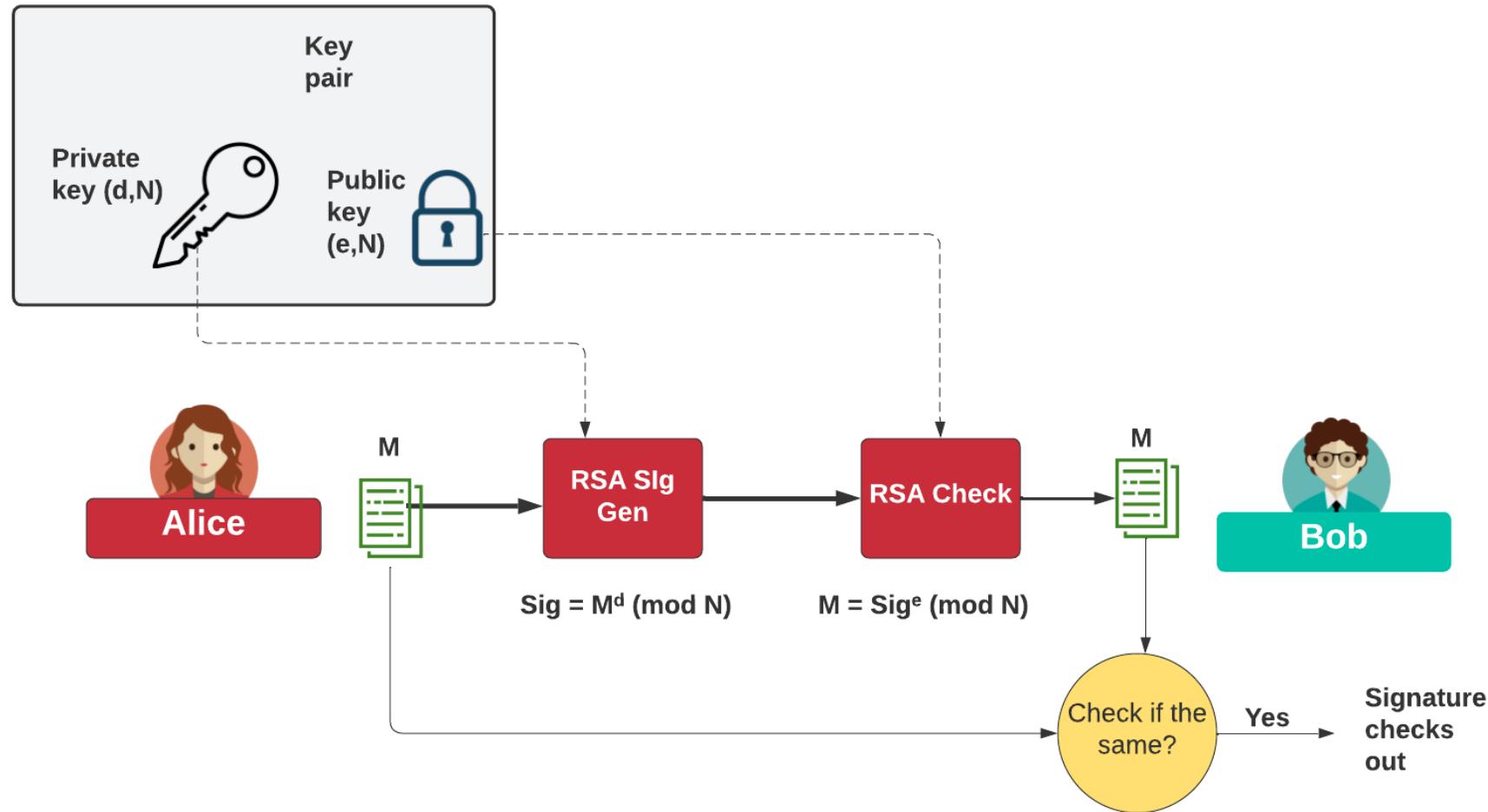
# Trent



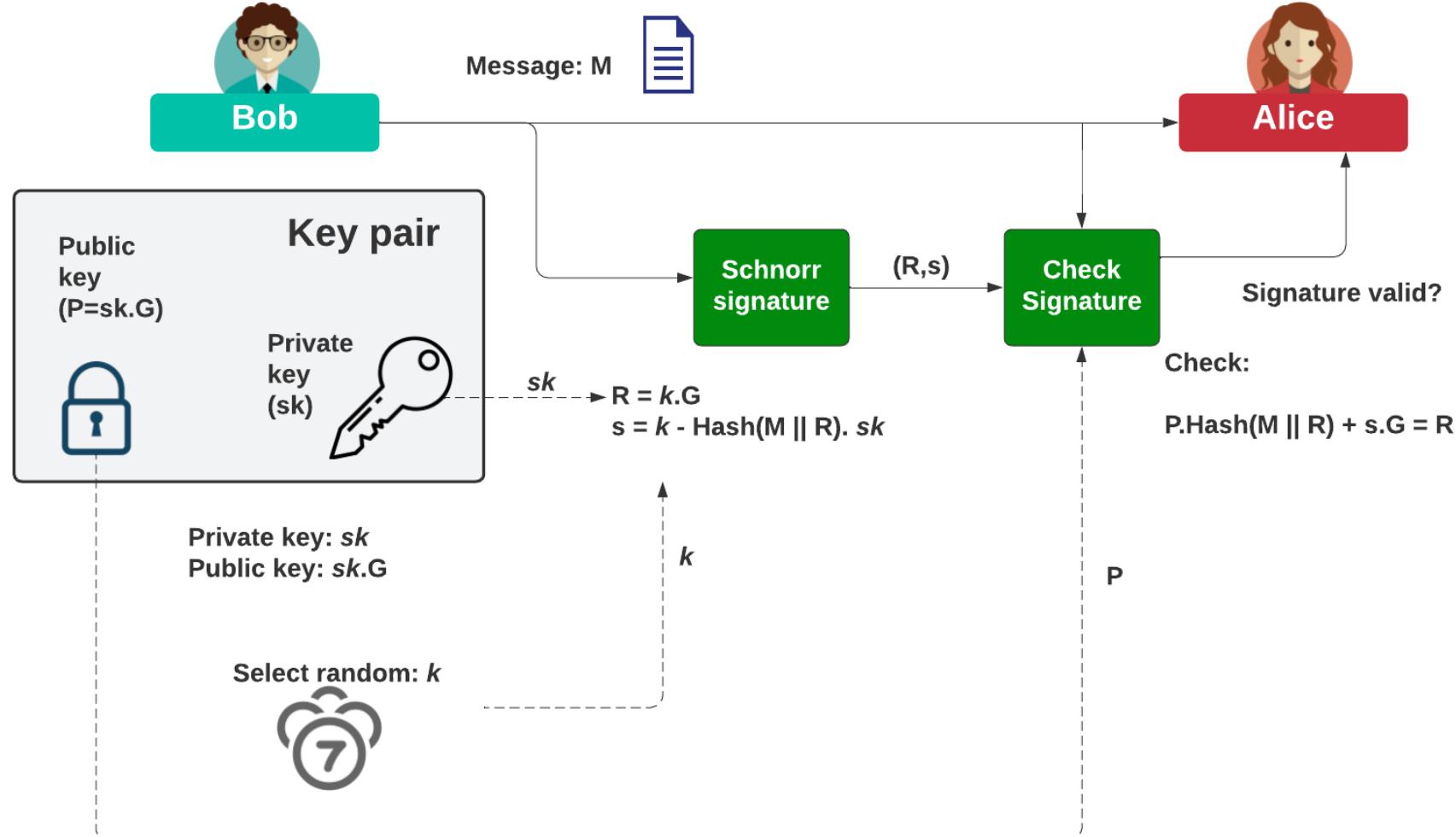
# ECDSA



# RSA Signatures



# Schnorr Signatures – used with EdDSA



# The Four Signatures

- ECDSA. [Example](#).
- EdDSA (Ed25519). [Example](#).
- RSA PKCS#1 v1.5. [Example](#).
- RSA PSS (Probabilistic signature scheme). [Example](#).

Bob



Alice



# Cryptography

Prof Bill Buchanan OBE FRSE

<https://asecuritysite.com>

Fundamentals  
Symmetric Key  
Hashing  
MAC  
KDF  
Public Key  
Key Exchange  
Signatures  
Digital Certificates

Eve

