Certificates, Password Hashing, and Case Studies

CS 161 Spring 2022 - Lecture 11

YOU DON'T GET TO CHOSE YOUR THREAT MODEL
YOUR THREATS GET TO CHOSE YOU
imgflip.com

~Lea Kissner

Last Time: Public-Key Encryption and Digital Signatures

- Public-key cryptography: Two keys; one undoes the other
- Public-key encryption: One key encrypts, the other decrypts
  - Security properties similar to symmetric encryption
  - ElGamal: Based on Diffie-Hellman
    - The public key is g^b, and C_1 is g^r.
    - Not IND-CPA secure on its own
  - RSA: Produce a pair e and d such that $M^{ed} = M \bmod N$
    - Not IND-CPA secure on its own
- Hybrid encryption: Encrypt a symmetric key, and use the symmetric key to encrypt the message
- Digital signatures: Integrity and authenticity for asymmetric schemes
  - RSA: Same as RSA encryption, but encrypt the hash with the private key

Today

- Applied cryptography
- First half: In scope for the midterm
  - Certificates: How do we distribute public keys securely?
  - Password Hashing: How do we securely store passwords?
- Second half: Out of scope for the midterm
  - Stupid ~~Pet~~ Hash Tricks
  - How can we discover bad cryptography?
  - How do we securely send messages in practice?
  - How can we report abusive behavior?

Certificates

Textbook Chapter 13

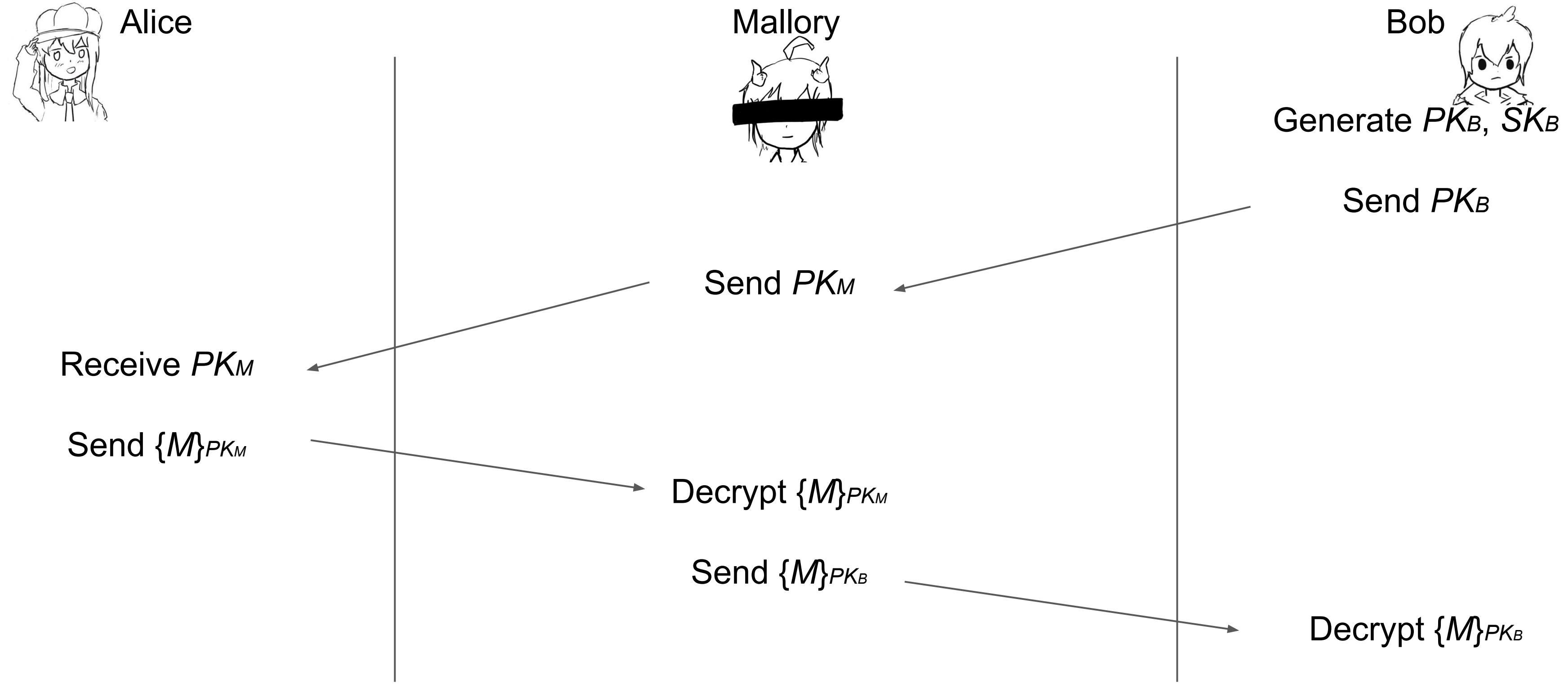
Review: Public-Key Cryptography

- Public-key cryptography is great! We can communicate securely without a shared secret
  - Public-key encryption: Everybody encrypts with the public key, but only the owner of the private key can decrypt
  - Digital signatures: Only the owner of the private key can sign, but everybody can verify with the public key
- What’s the catch?

Problem: Distributing Public Keys

- Public-key cryptography alone is not secure against man-in-the-middle attacks
- Scenario
  - Alice wants to send a message to Bob
  - Alice asks Bob for his public key
  - Bob sends his public key to Alice
  - Alice encrypts her message with Bob’s public key and sends it to Bob
- What can Mallory do?
  - Replace Bob's public key with Mallory’s public key
  - Now Alice has encrypted the message with Mallory’s public key, and Mallory can read it!

Problem: Distributing Public Keys



Problem: Distributing Public Keys

- Idea: Sign Bob’s public key to prevent tampering
- Problem
  - If Bob signs his public key, we need his public key to verify the signature
  - But Bob’s public key is what we were trying to verify in the first place!
  - Circular problem: Alice can never trust any public key she receives
- You cannot gain trust if you trust nothing. You need a root of trust!
  - **Trust anchor**: Someone that we implicitly trust
  - From our trust anchor, we can begin to trust others

Trust-on-First-Use

- **Trust-on-first-use**: The first time you communicate, trust the public key that is used and warn the user if it changes in the future
    - Used in SSH and a couple other protocols
    - Idea: Attacks aren’t frequent, so assume that you aren’t being attacked the first time communicate
    - Also known as “**Leap of Faith**”

Certificates

- **Certificate**: A signed endorsement of someone's public key
  - A certificate contains at least two things: The **identity** of the person, and the **key**
- Abbreviated notation
  - Encryption under a public key *PK*: $\{\text{"Message"}\}_{PK}$
  - Signing with a private key *SK*: $\{\text{"Message"}\}_{SK^{-1}}$
    - Recall: A signed message must contain the message along with the signature; you can't send the signature by itself!
- Scenario: Alice wants Bob's public key. Alice trusts EvanBot (PK_E, SK_E)
  - EvanBot is our trust anchor
  - If we trust PK_E, a certificate we would trust is $\{\text{"Bob's public key is } PK_B\text{"}\}_{SK_E^{-1}}$

Attempt #1: The Trusted Directory

- Idea: Make a central, trusted directory (TD) from where you can fetch anybody's public key
    - The TD has a public/private keypair PK_{TD}, SK_{TD}
    - The directory publishes PK_{TD} so that everyone knows it (baked into computers, phones, OS, etc.)
    - When you request Bob's public key, the directory sends a certificate for Bob's public key
        - {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
    - If you trust the directory, then now you trust every public key from the directory
- What do we have to trust?
    - We have received TD's key correctly
    - TD won't sign a key without verifying the identity of the owner

Attempt #1: The Trusted Directory

- Let's say that Michael Drake (MD, President of UC) runs the TD
  - We want Nick Weaver's public key: Ask MD
  - We want David Wagner's public key: Ask MD
  - We want Raluca Ada Popa's public key: Ask MD
  - MD has better things to do (like making sure his private key isn't stolen)!
- Problems: Scalability
  - One directory won't have enough compute power to serve the entire world
- Problem: Single point of failure
  - If the directory fails, *cryptography stops working*
  - If the directory is compromised, you can't trust anyone
  - If the directory is compromised, it is difficult to recover

Certificate Authorities

- Addressing scalability: Hierarchical trust
  - The roots of trust may **delegate** trust and signing power to other authorities
    - {"Carol Christ's public key is PK_{CC}, and I trust her to sign for UCB"}$_{SK_{MD}^{-1}}$
    - {"Dave Wagner's public key is PK_{DW}, and I trust him to sign for the CS department"}$_{SK_{CC}^{-1}}$
    - {"Nick Weaver's public key is PK_{NW} (but I don't trust him to sign for anyone else)"}$_{SK_{DW}^{-1}}$
  - MD is still the root of trust (**root certificate authority**, or **root CA**)
  - CC and DW receive delegated trust (**intermediate CAs**)
  - NW's identity can be trusted
- Addressing scalability: Multiple trust anchors
  - There are ~150 root CAs who are implicitly trusted by most devices
  - Public keys are hard-coded into operating systems and devices
  - Each delegation step can restrict the scope of a certificate's validity
  - Creating the certificates is an *offline* task: The certificate is created once in advance, and then served to users when requested

Revocation

- What happens if a certificate authority messes up and issues a bad certificate?
  - Example: {"Bob's public key is PK_M"}$_{SK_{CA}^{-1}}$
  - Example: Verisign (a certificate authority) accidentally issued a certificate saying that an average Internet user's public key belonged to Microsoft

Revocation: Expiration Dates

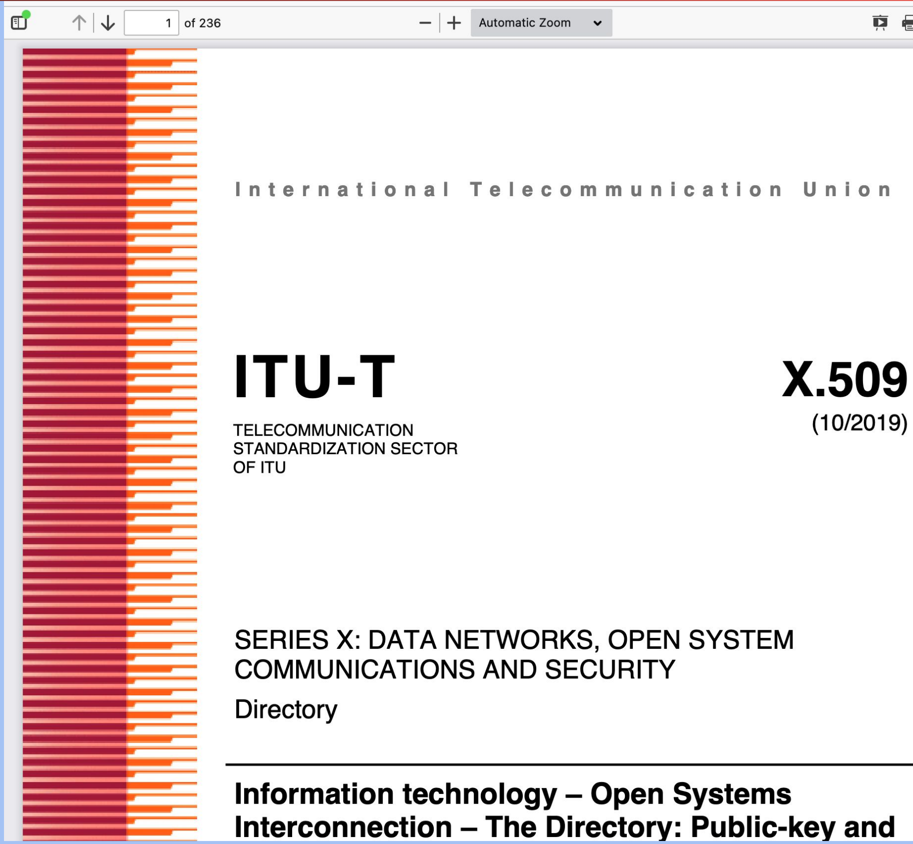
- Approach #1: Each certificate has an expiration date
  - When the certificate expires, request a new certificate from the certificate authority
  - The bad certificate will eventually become invalid once it expires
- Benefits
  - Mitigates damage: Eventually, the bad certificate will become harmless
- Drawbacks
  - Adds management burden: Everybody has to renew their certificates frequently
  - If someone forgets to renew a certificate, their website might stop working
- Tradeoff: How often should certificates be renewed?
  - Frequent renewal: More secure, less usable
  - Infrequent renewal: Less secure, more usable
- LetsEncrypt (a certificate authority) chose very frequent renewal
  - It turns out frequent renewal is more usable:
    It forces automated renewal instead of a once-every 3 year task that gets forgotten!

Revocation: Announcing Revoked Certificates

- Approach #2: Periodically release a list of invalidated certificates
  - Users must periodically download a Certification Revocation List (CRL)
- How do we authenticate the list?
  - The certificate authority signs the list!
    - {“The certificate with serial number 0xdeadbeef is now revoked”}$_{SK_{CA}^{-1}}$
- Drawbacks
  - Lists can get large
    - Mitigated by shorter expiration dates (don’t have to list them once they expire)
  - Until a user downloads a list, they won’t know which certificates are revoked
- What happens if the certificate authority is unavailable?
  - Fail-safe default: Assume all certificates are invalid? Now we can’t trust anybody!
    - Possible attack: Attacker forces the CA to be unavailable (denial of service attack)
  - Use old list: Potentially dangerous if the old list is missing newly revoked certificates

Certificates: Complexity

- Certificate protocols can get very complicated
  - Example: X.509 is incredibly complicated (a 236 page standard!) because it tried to do everything



International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.509
(10/2019)

SERIES X: DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

Directory

Information technology – Open Systems Interconnection – The Directory: Public-key and

Alternative: Web of Trust

- Modern public-key infrastructures are structured like trees
- Originally, public-key infrastructures looked like graphs instead
  - Everybody can issue certificates for anyone else
  - Example: Alice signs Bob's key. Bob signs Carol's key. If Dave trusts Alice, he trusts Bob and Carol.
  - Benefit: You know the trust anchor personally (e.g. because you met them in-person, or because you signed their key)
  - Problem: Graphs get far more complex than trees!
- OpenPGP (Pretty Good Privacy) originally used the web of trust model
  - Key-signing parties: meeting in-person to sign each other's public keys
  - It quickly proved to be a disaster
  - Instead, everyone just relies on MIT's central keyserver which is broken!
- **Takeaway**: Trust anchors make public-key infrastructures much simpler!

Summary: Certificates

- Certificates: A signed attestation of identity
- Trusted directory: One server holds all the keys, and everyone has the TD's public key
  - Not scalable: Doesn't work for billions of keys
  - Single point of failure: If the TD is hacked or is down, cryptography is broken
- Certificate authorities: Delegated trust from a pool of multiple root CAs
  - Root CAs can sign certificates for intermediate CAs
  - Revocation: Certificates contain an expiration date
  - Revocation: CAs sign a list of revoked certificates

Password Hashing

Textbook Chapter 14

Review: Cryptographic Hashes

- We love ourselves some cryptographic hashes
  - Some real-world examples: SHA-256, SHA-384, SHA3-256, SHA3-384
- Security properties
  - One way: Given an output y, it is infeasible to find any input x such that $H(x) = y$.
  - Second preimage resistant: Given an input x, it is infeasible to find another input $x' \neq x$ such that $H(x) = H(x')$.
  - Collision resistant: It is infeasible to find another any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.

Review: Cryptographic Hashes

- Hashes accept arbitrarily large inputs
- Hashes "look" random
  - Change a single bit on the input and each output bit has a 50% chance of flipping
  - And until you change the input, you can't predict which output bits are going to change
- The ones we talked about are *fast*
  - Can operate at many many MB/s: Faster at processing data than block ciphers

Storing Passwords

- Password: A secret string a user types in to prove their identity
  - When you create an account with a service: Create a password
  - When you later want to log in to the service: Type in the same password again
- How does the service check that your password is correct?
- Bad idea #1: Store a file listing every user's password
  - Problem: What if an attacker hacks into the service? Now the attacker knows everyone's passwords!
- Bad idea #2: Encrypt every user's password before storing it
  - Problem: The attacker could steal the passwords file *and* the key and decrypt everyone's passwords!
- We need a way to verify passwords *without* storing any information that would allow someone to recover the original password

Password Hashing

- For each user, store a *hash* of their password
- Verification process
    - Hash the password submitted by the user
    - Check if it matches the password hash in the file
- What properties do we need in the hash?
    - Deterministic: To verify a password, it has to hash to the same value every time
    - One-way: We don’t want the attacker to reverse hashes into original passwords

Password Hashing: Attacks

- What if two different users decide to use **`password123`** as their password?
    - Hashes are deterministic: They'll have the same password hash
    - An attacker can see which users are using the same password
- Brute-force attacks
    - Most people use insecure, common passwords
    - An attacker can pre-compute hashes for common passwords: *H*(**`"password123"`**), *H*(**`"password1234"`**), *H*(**`"1234567890"`**), etc.
    - **Dictionary attack**: Hash an entire dictionary of common passwords
- **Rainbow tables**: An algorithm for computing hashes that makes brute-force attacks easier

Salted Hashes

- Solution #1: Add a unique, random salt for each user
- **Salt**: A random, public value designed to make brute-force attacks harder
  - For each user, store: username, salt, *H*(password || salt)
  - To verify a user: look up their salt in the passwords file, compute *H*(password || salt), and check it matches the hash in the file
  - Salts should be long and random
  - Salts are not secret (think of them like nonces or IVs)
- Brute-force attacks are now harder
  - Assume there are M possible passwords and N users in the database
  - Unsalted database: Hash all possible passwords, then lookup all users' hashes ⇒ $O(M + N)$
  - Salted database: Hash all passwords for each user's salt ⇒ $O(MN)$

Slow Hashes

- Solution #2: Use slower hashes
- Cryptographic hashes are usually designed to be fast
  - SHA is designed to produce a checksum of your 1 GB document as fast as possible
- Password hashes are usually designed to be slow
  - Legitimate users only need to submit a few password tries. Users won’t notice if it takes 0.0001 seconds or 0.1 seconds for the server to check a password.
  - Attackers need to compute millions of hashes. Using a slow hash can slow the attacker by a factor of 1,000 or more!
  - Note: We are not changing the asymptotic difficulty of attacks. We’re adding a large constant factor, which can have a huge practical impact for the attacker

Slow Hashes: PBKDF2

- **Password-based key derivation function 2 (PBKDF2)**: A slow hash function
  - Setting: An underlying function that outputs random-looking bits (e.g. HMAC-SHA256)
  - Setting: The desired length of the output (*n*)
  - Setting: Iteration count (higher = hash is slower, lower = hash is faster)
  - Input: A password
  - Input: A salt
  - Output: A long, random-looking *n*-bit string derived from the password and salt
  - Implementation: Basically computing HMAC 10,000 times
- Benefits (assuming the user password is strong)
  - Derives an arbitrarily long string from the user's password
  - Output can be directly used as a symmetric key
  - Output can also be used to seed a PRNG or generate a public/private key pair
  - Algorithm is slow, but doesn't use a lot of memory (alternatives like Scrypt and Argon2 use more memory)

Offline and Online Attacks

- **Offline attack**: The attacker performs all the computation themselves
  - Example: Mallory steals the password file, and then computes hashes herself to check for matches.
  - The attacker can try a huge number of passwords (e.g. use many GPUs in parallel)
  - Defenses: Salt passwords, use slow hashes
  - If an attacker can do an offline attack, you need a really strong password (e.g. 7 or more random words)
- **Online attack**: The attacker interacts with the service
  - Example: Mallory tries to log in to a website by trying every different password. Mallory is forcing the server to compute the hashes.
  - The attacker can usually only try a few times per second, with no parallelism
  - Defenses: Add a timeout or rate limit the number of tries to prevent the attacker from trying too many times

Summary: Password Hashing

- Store hashes of passwords so that you can verify a user's identity without storing their password
- Attackers can use brute-force attacks to learn passwords (especially when users use weak passwords)
  - Defense: Add a different **salt** for each user: A random, public value designed to make brute-force attacks harder
- **Offline attack**: The attacker performs all the computation themselves
  - Defense: Use salted, slow hashes instead of unsalted, fast hashes
- **Online attack**: The attacker interacts with the service
  - Defense: Use timeouts

Case Study: iPhone Security

iPhone Security

- Apple's security philosophy:
  - In your hands, you can access everything on your phone
  - In anybody else's hands, the phone is an inert "brick" (nothing can be accessed)
- Apple uses a small co-processor in the phone to handle all cryptography
  - The "Secure Enclave" (recall: small TCB)
- The rest of the phone is untrusted
  - Memory is untrusted, so all data must be encrypted
  - The CPU must ask the Secure Enclave to decrypt data
  - Some data (e.g. credit card information for Apple Pay) is only readable by the Secure Enclave
- Effaceable Storage
  - Data is often stored in multiple places for redundancy, or not entirely wiped on deletion (for speed)
  - Effaceable storage: A section of memory where if memory is wiped, it is guaranteed to be gone
  - Requires some electrical engineering trickery to implement

iPhone Security

- A lot of keys encrypted by keys
  - There is a random master key, K_{phone}, that can be used to decrypt all the other keys
- K_{phone} is encrypted by the user's password and stored in flash memory
  - Run PBKDF on the password to get K_{user}, and use K_{user} to encrypt K_{phone}
- How do we prevent an offline brute-force attack?
  - Include a random 256-bit secret hardcoded into the Secure Enclave for encryption
  - The only way to get this secret is to take apart the chip!
  - The Secure Enclave can’t read the secret, but can only use it as an input for hardware cryptography
  - The user key K_{user} is actually function of the password and Enc(secret, password)
  - Offline attacks are not possible without the secret
- **Takeaway**: Mixing in on-chip secret requires online attacks!
  - If the attacker doesn't know the on-chip secret, they can’t perform an offline attack

iPhone Security

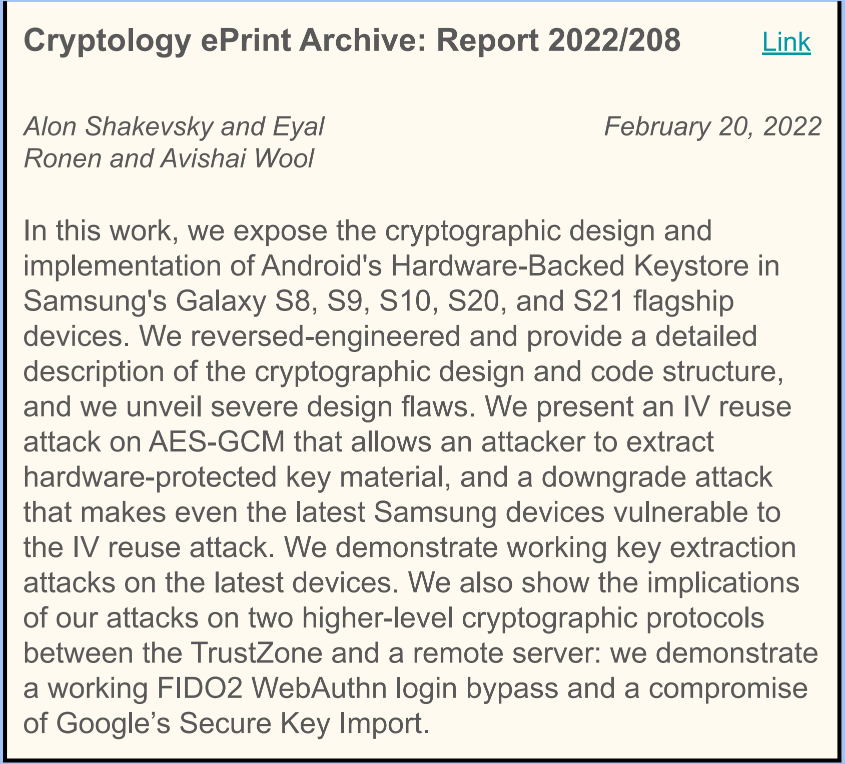
- How do we prevent online brute-force attacks?
  - Online attacks must go through the secure enclave
  - Timeouts: After 5 tries, add a delay to slow down each try
  - After 10 tries, optionally wipe $\boldsymbol{K}_{\textbf{phone}}$ forever
  - Remember: $\boldsymbol{K}_{\textbf{phone}}$ is the root key for decrypting everything else. Erasing $\boldsymbol{K}_{\textbf{phone}}$ effectively erases everything on the phone!
  - Even if an attacker compromises the secure enclave, guessing is still limited to 10 tries per second
- **Takeaway**: Timeouts and slow algorithms prevent online attacks. If possible, limit attackers to online attacks

iPhone Security: Backups

- A necessary weakness: Backups
  - Backup: Copying all data from the phone somewhere else
  - The data is copied unencrypted (decrypted with the secret on the chip)
  - Backups are necessary so you can recover your data on another phone
- Attack: Use backups to steal your data
  - The attacker finds a way to unlock your phone without your password (e.g. hold it to your face or use your fingerprint)
  - Remember: consider human factors!
  - The attacker syncs your phone with a new computer
- Change of policy as of iOS 11
  - To create a backup on a new computer, you need to enter your password to trust the computer
  - Previous attack is no longer possible: can't create a backup without knowing the password

Case Study: Samsung

- Samsung has a similar concept
  - But their implementation has, umm, issues…
- Guess with, GCM with IV reuse!
  - And although “fixed”, you could do a downgrade attack to cause the fixed version to still reuse IVs
- **Takeaway**: CTR mode is dangerous when used improperly…and even experts misuse it!



Cryptology ePrint Archive: Report 2022/208 Link

Alon Shakevsky and Eyal Ronen and Avishai Wool *February 20, 2022*

In this work, we expose the cryptographic design and implementation of Android's Hardware-Backed Keystore in Samsung's Galaxy S8, S9, S10, S20, and S21 flagship devices. We reversed-engineered and provide a detailed description of the cryptographic design and code structure, and we unveil severe design flaws. We present an IV reuse attack on AES-GCM that allows an attacker to extract hardware-protected key material, and a downgrade attack that makes even the latest Samsung devices vulnerable to the IV reuse attack. We demonstrate working key extraction attacks on the latest devices. We also show the implications of our attacks on two higher-level cryptographic protocols between the TrustZone and a remote server: we demonstrate a working FIDO2 WebAuthn login bypass and a compromise of Google’s Secure Key Import.

Case Study: Stupid Hash Tricks

Textbook Chapter 7.4

Application: Lowest-Hash Scheme

- Scenario
  - An attacker has stolen 150 million (150,000,000) records from a website
  - The attacker wants to prove to us that they didn’t steal fewer records and exaggerate the number
  - The attacker doesn’t want to send all 150M records to us
  - How can we be sure the attacker isn't lying?
- Idea: Use cryptographic hashes
  - Ask the attacker to hash all 150M records and send us the 10 records with the *lowest* hash value
  - Hashes are unpredictable, so the attacker is basically choosing 10 random records
- How do we know the attacker didn’t cheat?
  - Check the hashes of the 10 records returned, and verify the records
  - Hashes look random, so check if the hashes match what we expect the minimum of 150M random values to be

Sample a File

```
#!/usr/bin/env python
import hashlib, sys
hashes = {}

for line in sys.stdin:
    line = line.strip()
    for x in range(10):
        tmp = "%s-%i-nickrocks" % (line, x)
        hashval = hashlib.sha256(tmp)
        h = hashval.digest()
        if x not in hashes or hashes[x][0] > h:
            hashes[x] = (h, hashval, tmp)

for x in range(10):
    h, hashval, val = hashes[x]
    print "%s=\"%s\"" % (hashval.hexdigest(), val)
```

Why does this work?

- For each x in range 0-9…
  - Calculates H(line||x)
  - Stores the lowest hash matching so far
- Since the hash appears random…
  - Each iteration is an independent selection from the file
  - The expected value of minimum(H(line||x)) is a function of the size of the file: More lines, and the value is smaller
- To fake it…
  - Would need to generate fake lines, and see if the hash is suitably low
  - Yet would need to make sure these fake lines semantically match!
    - Thus you can't just go "John Q Fake", "John Q Fakke", "Fake, John Q", etc...The format for all lines should be the same

Limiting Attacker Resources

- Some protocols are vulnerable to attacks if the attacker has access to a huge amount of resources
- We can defend against these attacks by forcing the attacker to complete a task before accessing our system
- Example: CAPTCHAs
  - The "prove you are a human" web puzzles (we'll see these again later)
  - Attacker provides proof that they have wasted a few seconds of a human's time (or paid $.01 to waste a few seconds of another human's time)
- Example: Proof of *wait*
  - Alice has a secret key ***k***
  - Alice to Bob sends "Don't contact me until time ***T***, here is HMAC(***k***,***T***)"
  - When Bob gets back, he says "***T***, HMAC(k,***T***)"
  - Alice then verifies ***T*** is in the past and HMAC(k,***T***)

Hash Application: Proof-of-Work

- Alice wants Bob to waste a bunch of CPU resources
  - Alice wants to be able to check that Bob wasted a bunch of resources, without wasting a bunch of resources herself
- Alice issues a challenge to Bob
  - Alice provides a random message *M* and a difficulty factor *n*
  - Bob must compute *M*||*M*' such that H(*M*||*M*') starts with n consecutive zeros
  - Alice computes H(*M*||*M*') and verifies that it starts with n zero bits
- Bob's computation cost: O(2^n)
  - Hashes are unpredictable and look random, so to find a random bitstring that starts with *n* consecutive zeros, Bob has to compute approximately 2^n hashes
- Alice's verification cost: O(1)
  - Alice only needs to compute one hash to verify that Bob found a hash that ends in *n* consecutive zeros
- This scheme is called “proof of work” or “proof of waste”
  - Bob proves that he wasted a large number of hash computations

Hash Application: HKDF

- Recall: We want to avoid reusing the same key in different algorithms
  - Example: When using encrypt-then-MAC, use different keys for the encryption algorithm and the MAC algorithm
- Hash Based Key Derivation Function: An algorithm that uses HMAC to create several different keys, starting from a single key

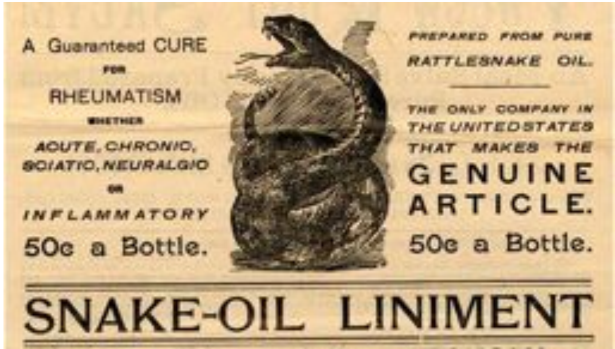
Hash Application: HKDF Construction

```
hkdf(keydata, info, L):
    T = Out = ""
    for (i = 1; i <= ceiling(L/hashlen); ++i){
        T = HMAC(keydata, T || info || i);
        Out = Out || T
    }
    return Out[0:L-1]
```

Case Study: Snake Oil Cryptography

Snake Oil

- Original meaning: Fraudulent “cure-all” medicines sold in the 1700s and 1800s
  - Sellers promised buyers that snake oil cures all diseases (even though it doesn’t)
  - Took advantage of uninformed buyers and lack of regulations
- Modern meaning: Scams
  - Using deceptive advertising to trick uninformed buyers into buying useless products
- **Snake oil security** (**snake oil cryptography**): Useless security products advertised to uninformed buyers


A Guaranteed CURE
FOR
RHEUMATISM
WHETHER
ACUTE, CHRONIC, SCIATIC, NEURALGIC
OR
INFLAMMATORY
50c a Bottle.
PREPARED FROM PURE RATTLESNAKE OIL.
THE ONLY COMPANY IN THE UNITED STATES THAT MAKES THE
GENUINE ARTICLE.
50c a Bottle.
SNAKE-OIL LINIMENT

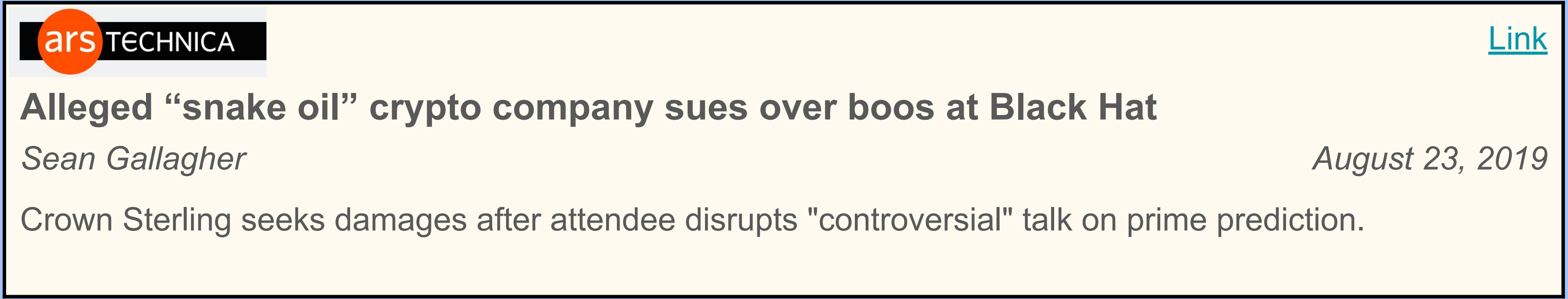
Signs of Snake Oil Cryptography

- Amazingly long key lengths
  - Once brute-forcing a key becomes astronomically hard, making it longer probably doesn't provide extra security
  - The NSA is super paranoid, and even they don’t use >256-bit symmetric keys or >4,096-bit public keys
- New algorithms and wild protocols
  - There is no reason to use a brand-new block cipher, hash algorithm, public-key algorithm, etc.
  - Existing protocols have been vetted by security experts for years: They’re widespread for a good reason!
  - New protocols probably means someone is trying to write their own crypto (and asking for trouble!)
- Fancy-sounding technical buzzwords
  - Claims of inventing “new math”

Signs of Snake Oil Cryptography

- “One time pads”
  - Recall: One-time pads are secure if you never reuse the key
  - Recall: Secure one-time pads are highly impractical
  - Almost all schemes advertised as “one-time pads” probably aren't true one-time pads
  - Wacky stream ciphers (often self-designed) are often advertised as “one-time pads”
- Rigged “cracking contests”
  - Advertising a secure scheme by challenging the public to break the scheme
  - The challenge is often “decrypt this message” with no context or structure
  - Example: Telegram offered a $300,000 prize in a contest to break their encryption

Snake Oil Example: Crown-Sterling


ars TECHNICA

Link

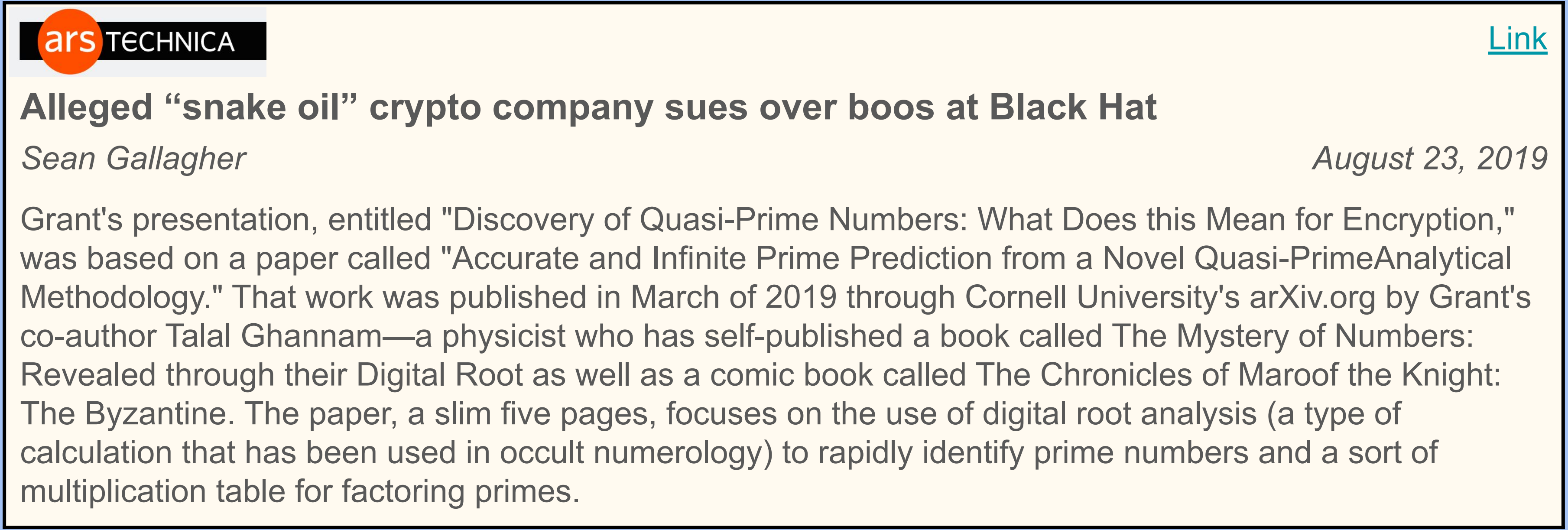
Alleged “snake oil” crypto company sues over boos at Black Hat

Sean Gallagher *August 23, 2019*

Crown Sterling seeks damages after attendee disrupts "controversial" talk on prime prediction.

Crown Sterling: A snake-oil cryptography company
Black Hat: A security research conference

Snake Oil Example: Crown-Sterling



ars TECHNICA

Link

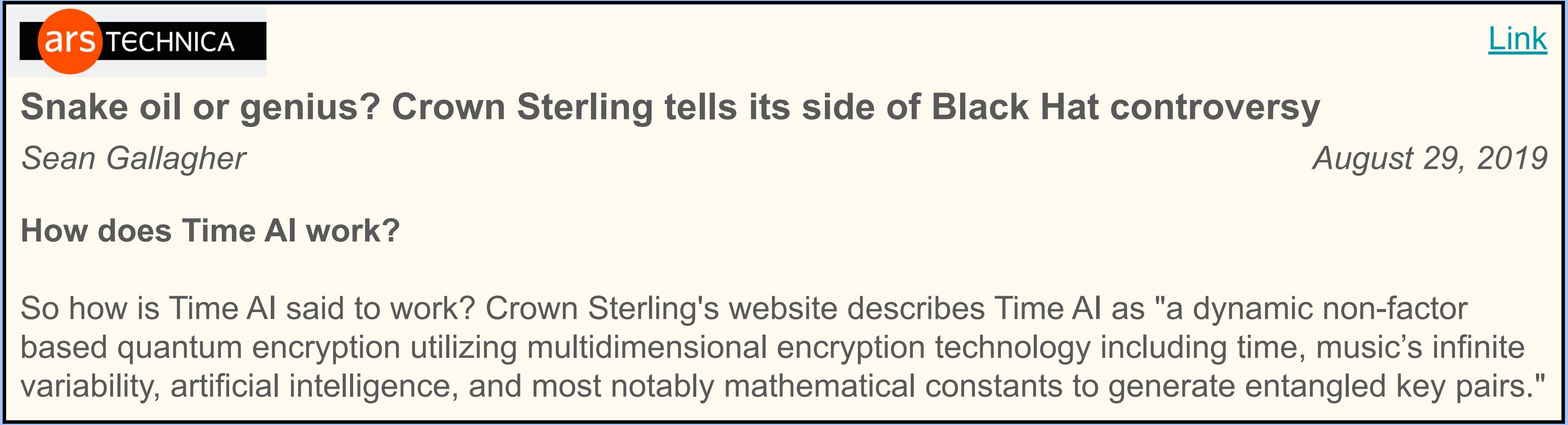
Alleged “snake oil” crypto company sues over boos at Black Hat

Sean Gallagher *August 23, 2019*

Grant's presentation, entitled "Discovery of Quasi-Prime Numbers: What Does this Mean for Encryption," was based on a paper called "Accurate and Infinite Prime Prediction from a Novel Quasi-PrimeAnalytical Methodology." That work was published in March of 2019 through Cornell University's arXiv.org by Grant's co-author Talal Ghannam—a physicist who has self-published a book called The Mystery of Numbers: Revealed through their Digital Root as well as a comic book called The Chronicles of Maroof the Knight: The Byzantine. The paper, a slim five pages, focuses on the use of digital root analysis (a type of calculation that has been used in occult numerology) to rapidly identify prime numbers and a sort of multiplication table for factoring primes.

Takeaway: Buzzwords are signs of snake oil cryptography

Snake Oil Example: Crown-Sterling



ars TECHNICA

Link

Snake oil or genius? Crown Sterling tells its side of Black Hat controversy

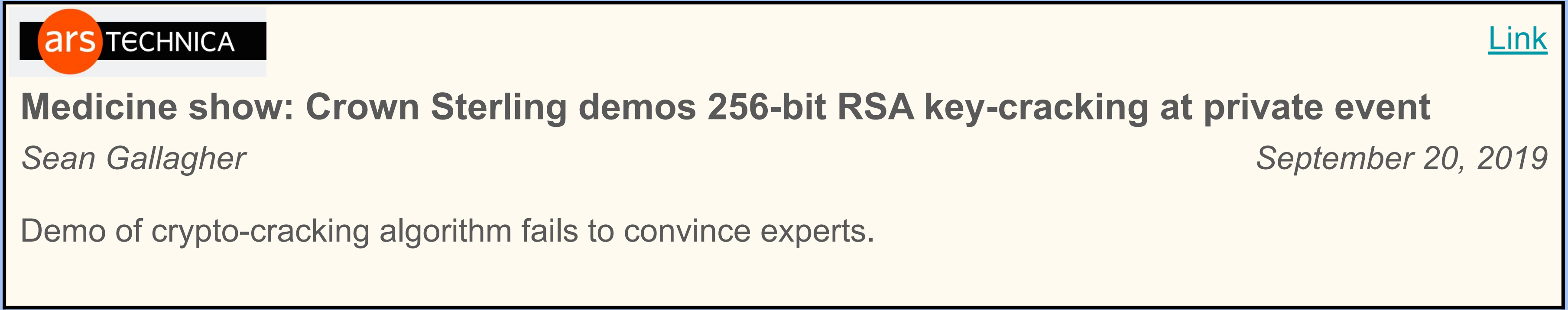
Sean Gallagher *August 29, 2019*

How does Time AI work?

So how is Time AI said to work? Crown Sterling's website describes Time AI as "a dynamic non-factor based quantum encryption utilizing multidimensional encryption technology including time, music's infinite variability, artificial intelligence, and most notably mathematical constants to generate entangled key pairs."

Takeaway: Brand-new math and buzzwords are signs of snake oil cryptography

Snake Oil Example: Crown-Sterling


ars TECHNICA

Link

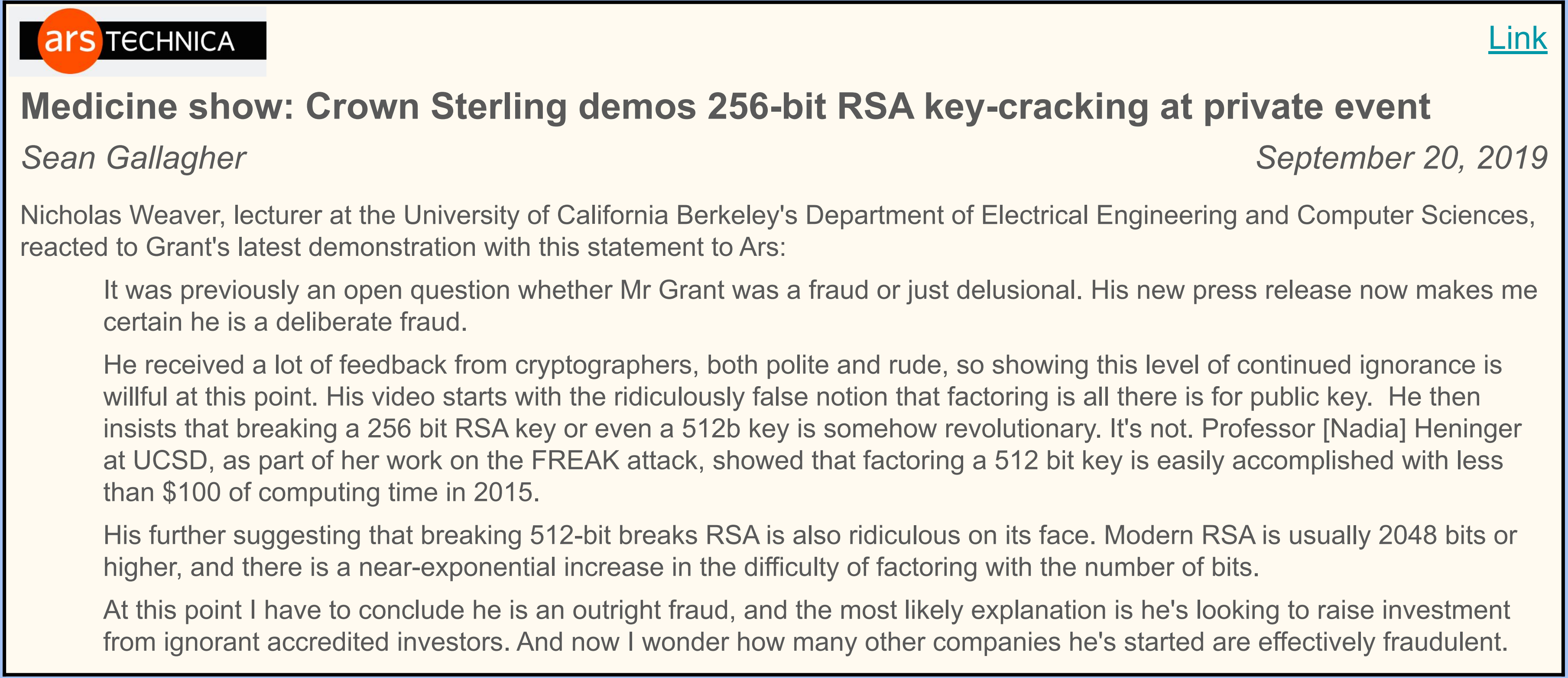
Medicine show: Crown Sterling demos 256-bit RSA key-cracking at private event

Sean Gallagher *September 20, 2019*

Demo of crypto-cracking algorithm fails to convince experts.

Takeaway: Flashy demonstrations are signs of snake oil cryptography

Snake Oil Example: Crown-Sterling



ars TECHNICA

Link

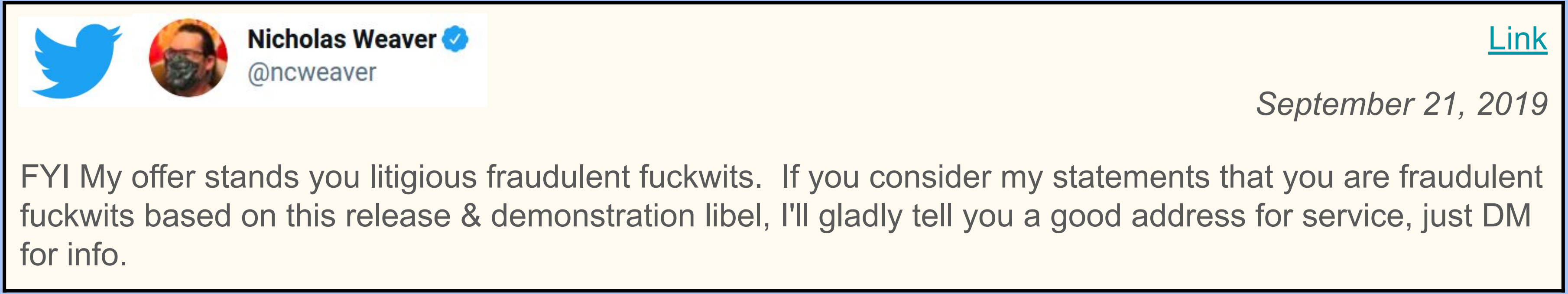
Medicine show: Crown Sterling demos 256-bit RSA key-cracking at private event

Sean Gallagher *September 20, 2019*

Nicholas Weaver, lecturer at the University of California Berkeley's Department of Electrical Engineering and Computer Sciences, reacted to Grant's latest demonstration with this statement to Ars:

> It was previously an open question whether Mr Grant was a fraud or just delusional. His new press release now makes me certain he is a deliberate fraud.
>
> He received a lot of feedback from cryptographers, both polite and rude, so showing this level of continued ignorance is willful at this point. His video starts with the ridiculously false notion that factoring is all there is for public key. He then insists that breaking a 256 bit RSA key or even a 512b key is somehow revolutionary. It's not. Professor [Nadia] Heninger at UCSD, as part of her work on the FREAK attack, showed that factoring a 512 bit key is easily accomplished with less than $100 of computing time in 2015.
>
> His further suggesting that breaking 512-bit breaks RSA is also ridiculous on its face. Modern RSA is usually 2048 bits or higher, and there is a near-exponential increase in the difficulty of factoring with the number of bits.
>
> At this point I have to conclude he is an outright fraud, and the most likely explanation is he's looking to raise investment from ignorant accredited investors. And now I wonder how many other companies he's started are effectively fraudulent.

Snake Oil Example: Crown-Sterling



Nicholas Weaver
@ncweaver

Link

September 21, 2019

FYI My offer stands you litigious fraudulent fuckwits. If you consider my statements that you are fraudulent fuckwits based on this release & demonstration libel, I'll gladly tell you a good address for service, just DM for info.

Example: Cryptocurrency Snake Oil

- IOTA: A cryptocurrency designed for the Internet of Things (IoT)
  - Uses a hash-based scheme instead of standard public key signatures, meaning you can never reuse a key
  - 10,000-bit signatures (compare with 450-bit RSA signatures, which are considered big)
  - Created their own hash function… that was quickly broken
  - Claims to be a distributed system, but relies entirely on a central authority (not distributed)
  - Uses trinary math? (Requiring entirely new processors?)
- **Takeaway**: Be able to recognize snake oil cryptography

Case Study: Signal (Encrypted Messaging)

End-to-End Messengers: Signal

- We love end to end cryptographic protocols…
  - Mallory, even if part of the infrastructure, gets nnothing
- We love forward secrecy…
  - If someone steals our private keys, they can't recover old messages
  - After all, we want things to stay secret even if our keys are compromised
- Forward secrecy is "easy" for online protocols
  - Just make sure to do a DHE/ECDHE key exchange, and throw away the session key when done
- Forward secrecy is much more annoying for an offline protocol
  - Alice wants to share data with Bob, but Bob is not online
    - Like in project 2…
    - Or any messenger system!

Signal Requirements For Key Agreement

- Three parties: Alice, Bob, and a messenger server
  - The messenger server is like the file store in project 2, an untrusted entity
  - A separate mechanism is used to provide key transparency
- Bob is **offline**:
  - He has prearranged data stored on the messenger server
- Alice and Bob want to create an ephemeral (DH) key…
  - To use for then encrypting messages
- They need **mutual authentication**
  - Assuming Alice and Bob have the correct public keys, only Alice and Bob could have agreed on a key
- They also need **deniability**
  - Alice or Bob can't create a record proving the other side sent a particular message:So no "Alice just signs her DH..." design

Case Study: Signal (Encrypted Messaging)
Case Study: Facebook Messenger Abuse Protocol

These sections are out of scope (all blue slides).
For a PDF copy of the slides, see:
https://cs161.org/assets/lectures/lec11-optional.pdf