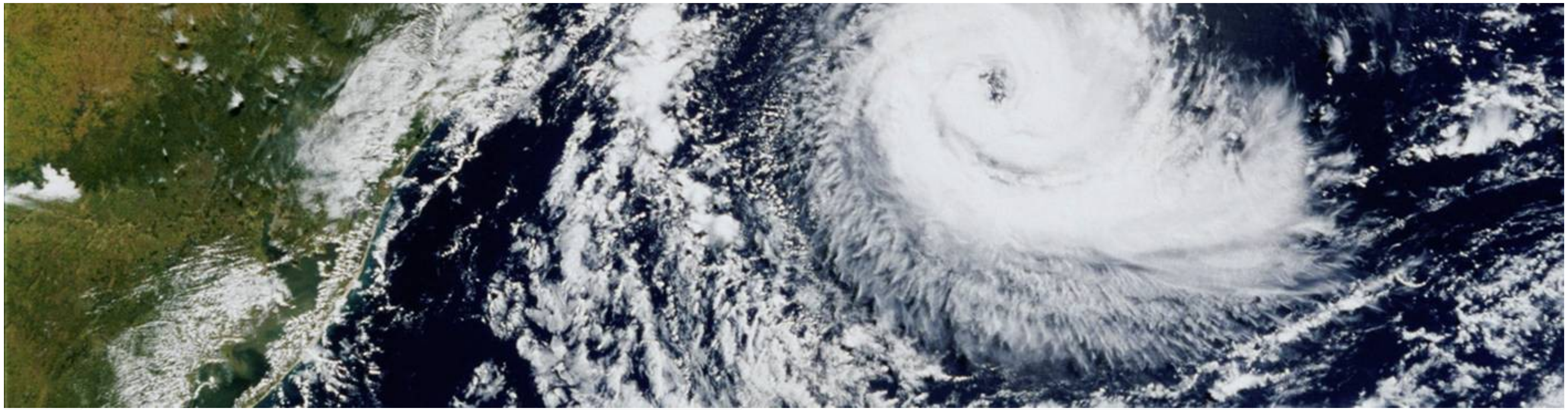
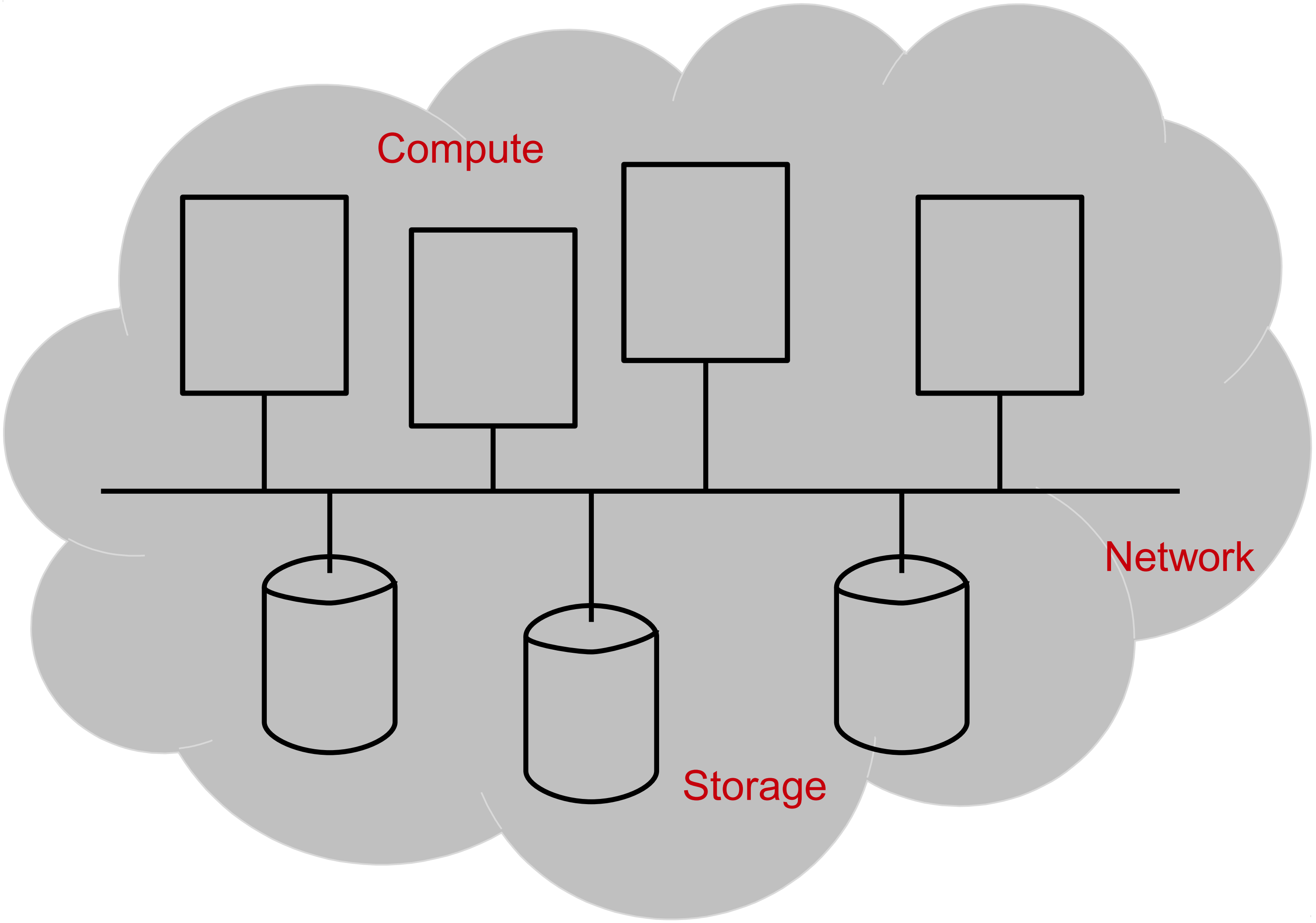
IBM Research – Zurich
Christian Cachin
November 2014

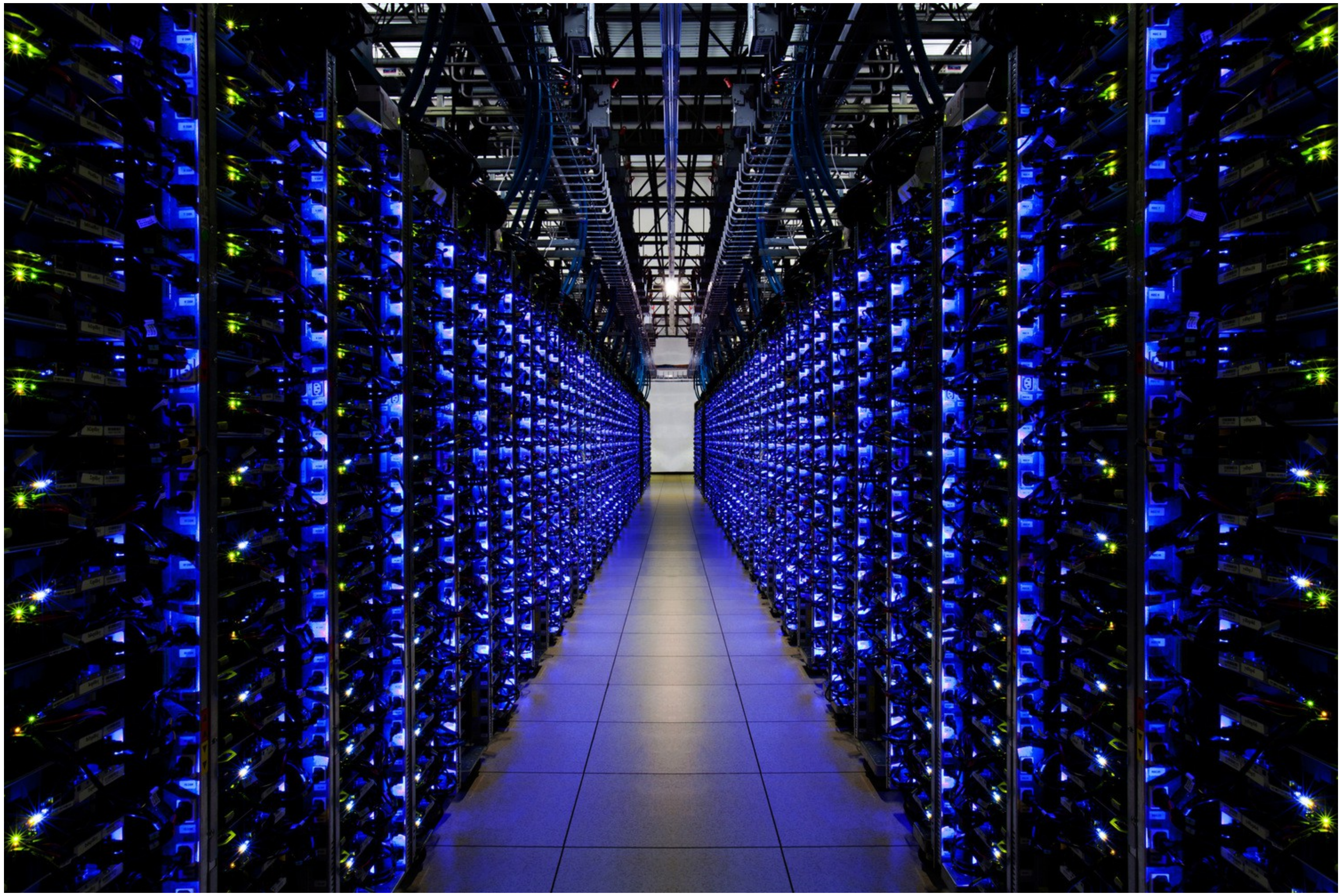
Integrity, Consistency, and Verification of Remote Computation



Cloud computing everywhere



Computing as a utility

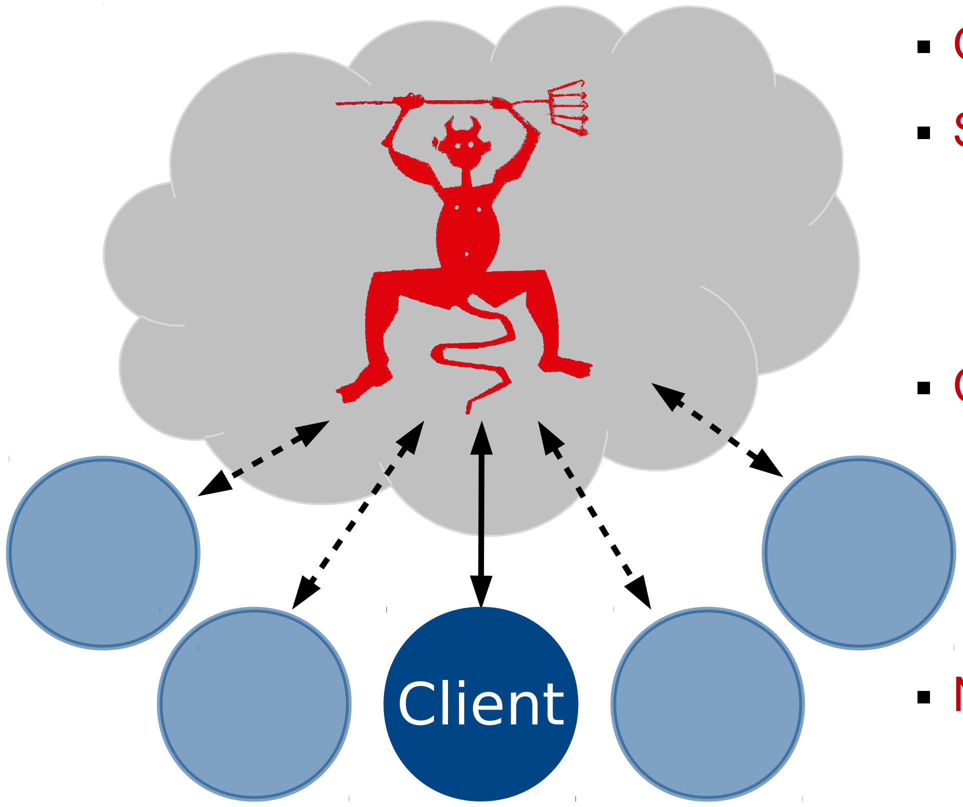

Google data center

Physical location irrelevant

Data center, Luleå (SE), near the Arctic circle

Verification, integrity, and consistency

Verification of responses by remote server

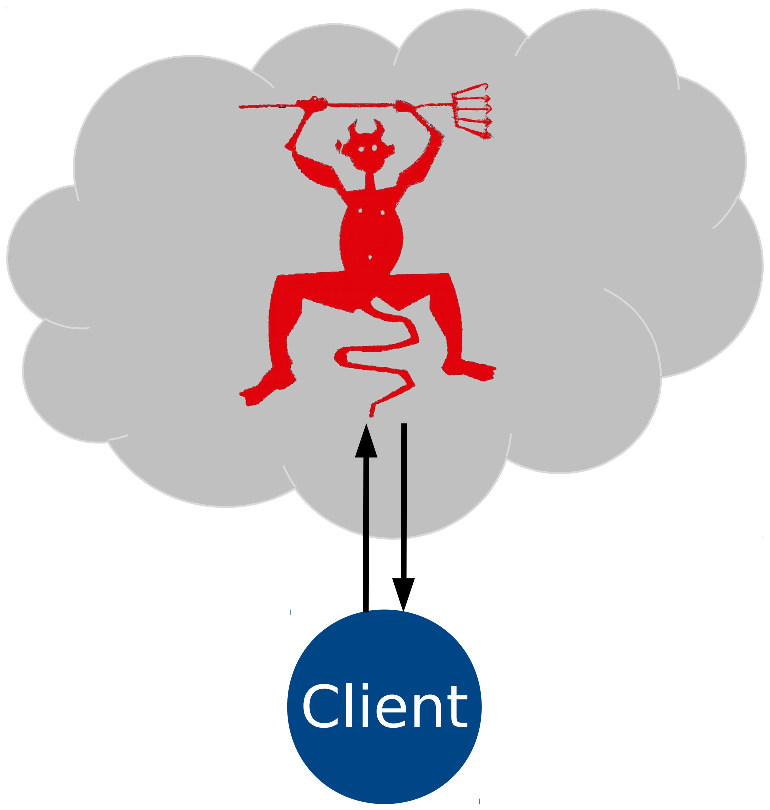


- One or more clients interact with server
- Server
  - Computes
  - Responds to requests
  - Maintains state
- Clients
  - Request operations and obtain responses
  - Verify correctness of responses
  - Maintain some (small) state
- No assumptions about server
  - Usually response is correct ... but sometimes not
  - No audits, no trusted hardware, no trusted 3rd party ...

Many dimensions of verification

- Server state
  - Server may maintain state across requests
- Server functionality F
  - Arbitrary (polynomial-time computable)
  - Specific data structures (maps, tables, ...)
  - Storage (only reads and writes)
- Single-client vs. multiple clients
  - With multiple clients, who verifies?
  - Clients need to agree on "correct" verification
- Prototypes and implementation

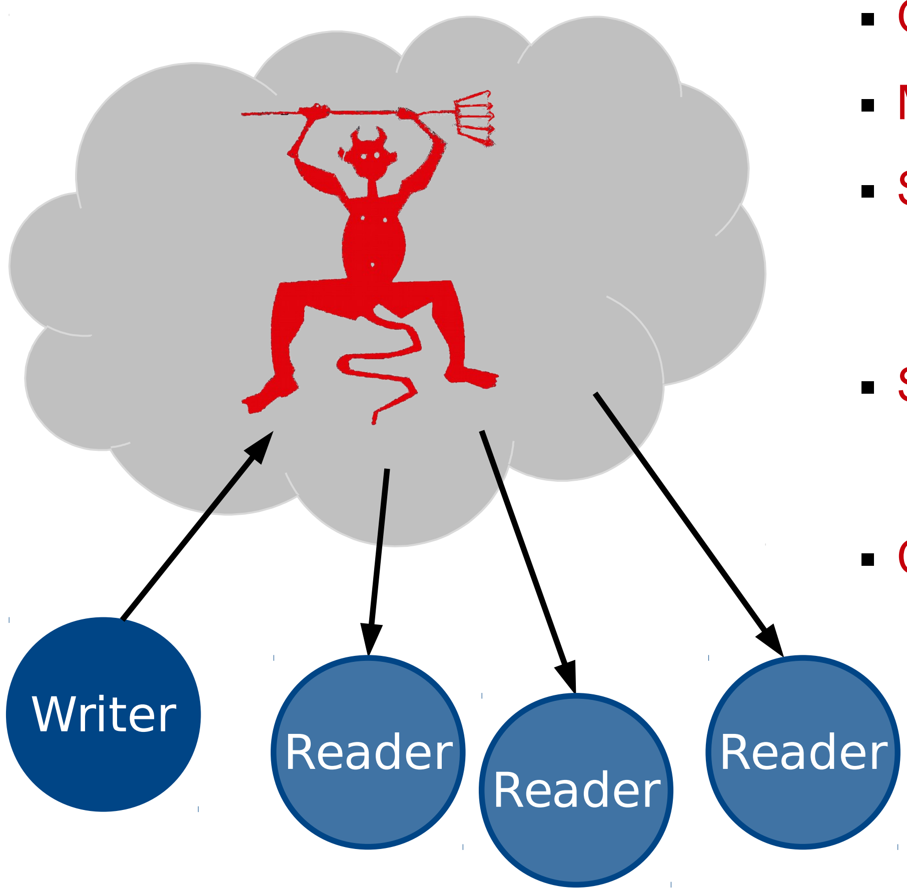
Part 1 — Verifiable computation



- One client
- Arbitrary functionality F
- No server state (mostly)

- Challenge
  - Verification should be faster than client recomputing the result by itself

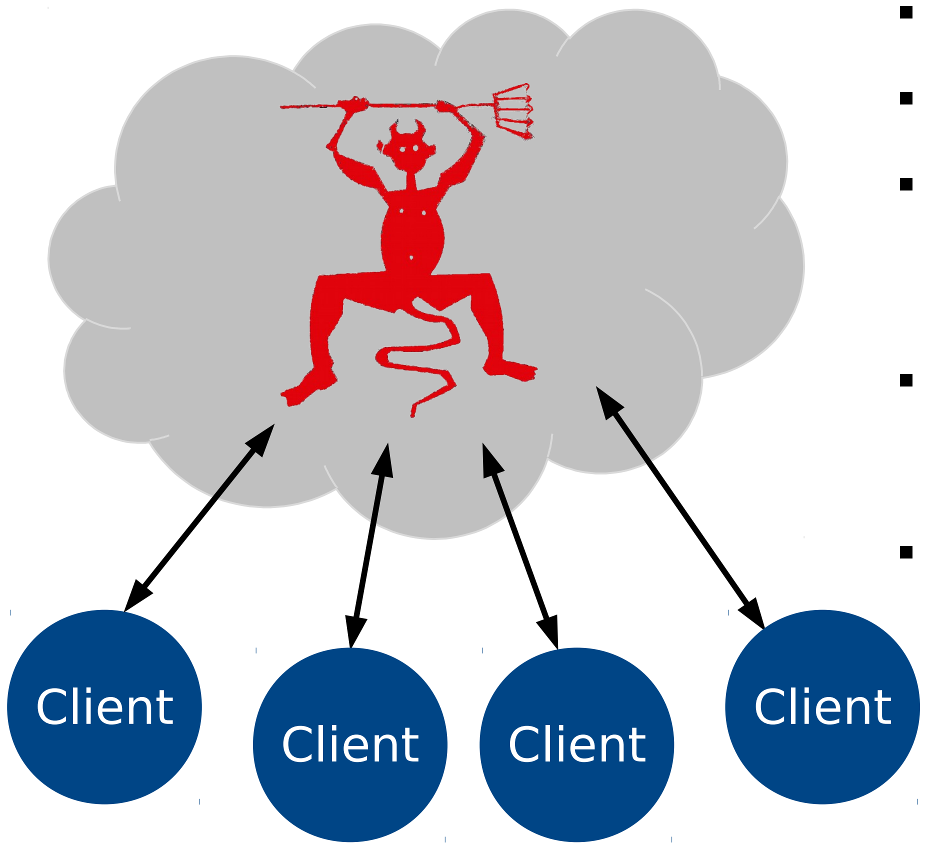
Part 2 — Authenticated data types



- One client writes
- Many clients read
- Specific class of functions F
  - Mostly related to queries over a database
- Server keeps state

- Challenge
  - Efficiency of verification operation
  - Generality of operations on data

Part 3 — Distributed consistency enforcement

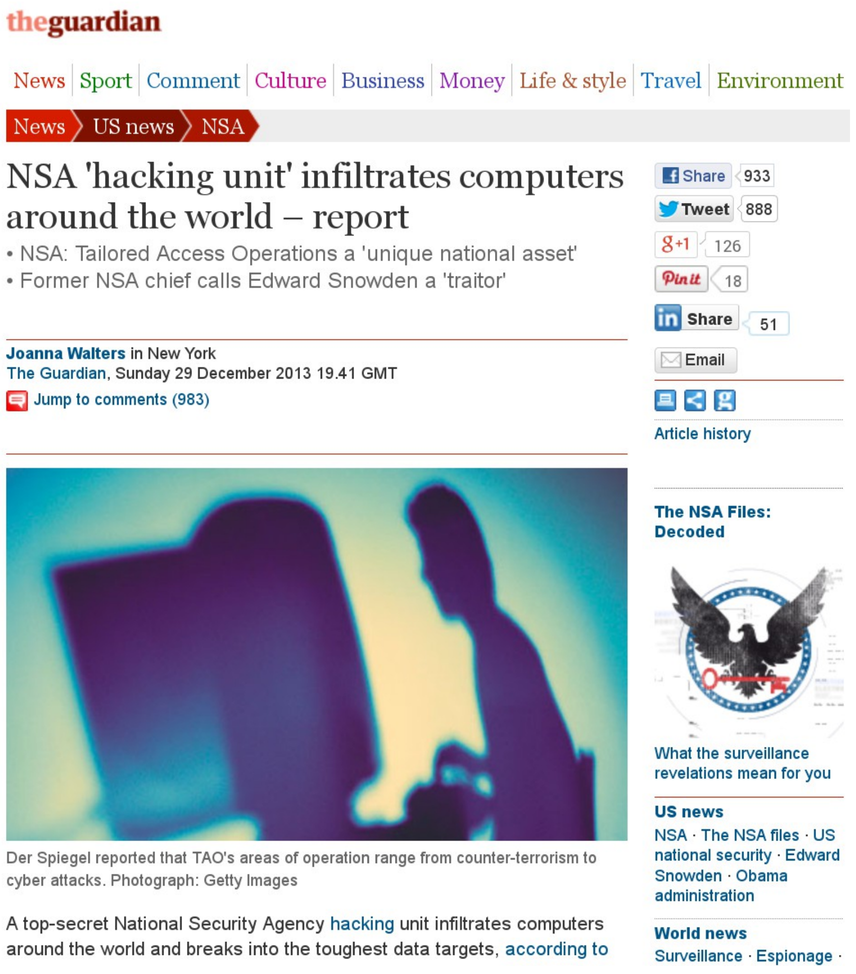


- All clients may write
- All clients may read
- Generic functions F
  - Data storage is an important special case
- Server keeps state

- Challenge
  - Some attacks cannot be prevented
  - How consistent are the views of the different clients?

Verifiable computation

Cloud computing — Do you get the correct answer?


theguardian

News | Sport | Comment | Culture | Business | Money | Life & style | Travel | Environment

News > US news > NSA

NSA 'hacking unit' infiltrates computers around the world – report

• NSA: Tailored Access Operations a 'unique national asset'
• Former NSA chief calls Edward Snowden a 'traitor'

Joanna Walters in New York
The Guardian, Sunday 29 December 2013 19.41 GMT
Jump to comments (983)

Der Spiegel reported that TAO's areas of operation range from counter-terrorism to cyber attacks. Photograph: Getty Images

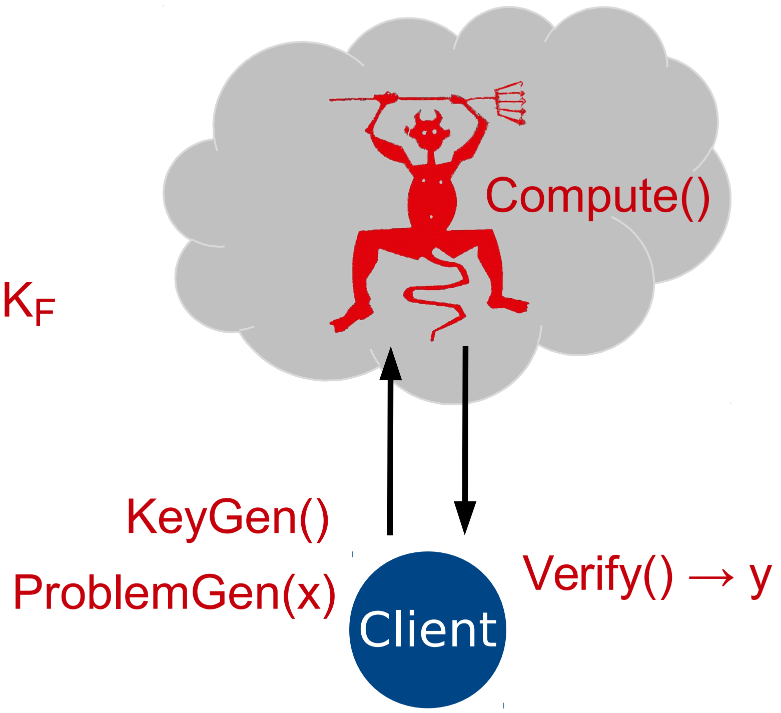
A top-secret National Security Agency hacking unit infiltrates computers around the world and breaks into the toughest data targets, according to

Share 933
Tweet 888
+1 126
Pin it 18
Share 51
Email
Article history
The NSA Files: Decoded
What the surveillance revelations mean for you
US news
NSA · The NSA files · US national security · Edward Snowden · Obama administration
World news
Surveillance · Espionage ·

- Clients depend on the results and outputs of remote computations
- Remote servers may not always give the correct response
- How can clients verify that a reply is correct?

Verifiable computation (VC) — Definition [GGP10]

- $\mathrm{KeyGen}(F, \kappa) \rightarrow (EK_F, VK_F, SK_F)$
  - Generates evaluation key EK_F, public verification key VK_F, secret key SK_F
- $\mathrm{ProblemGen}(SK_F, x) \rightarrow (\sigma x, \omega x)$
  - Client encodes problem instance x of F into σx, given to S, and secret ωx
- $\mathrm{Compute}(EK_F, \sigma x) \rightarrow (\sigma y)$
  - Server computes encoded result σy
- $\mathrm{Verify}(VK_F, \omega x, \sigma y) \rightarrow y$
  - Client obtains result y, where $y = \perp$ denotes that verification failed
- Designated verifier (as stated)
  - Only holder of secret keys may verify result
- Publicly verifiable (no secret keys, i.e., $SK_F = \perp$ and $\omega x = \perp$)
  - Everyone can verify result



Verifiable computation — Properties

- Correctness
  - For any polynomial-time functionality F and input x, run KeyGen(F, κ), ProblemGen(···x), Compute(···σx), Verify(···σy) → y
  - Then y = F(x)
- Security
  - For any F and any adversary **A**:
    - Run KeyGen(F, κ)
    - Repeatedly, **A** picks x' and obtains σx' from ProblemGen(···x')
    - **A** outputs x* and σy*, compute y* ← Verify(···σy*)
    - With overwhelming probability: y* ≠ ⊥ → y* = F(x*)
- Efficiency
  - Running ProblemGen() and Verify() takes less time together than evaluating F(x) directly

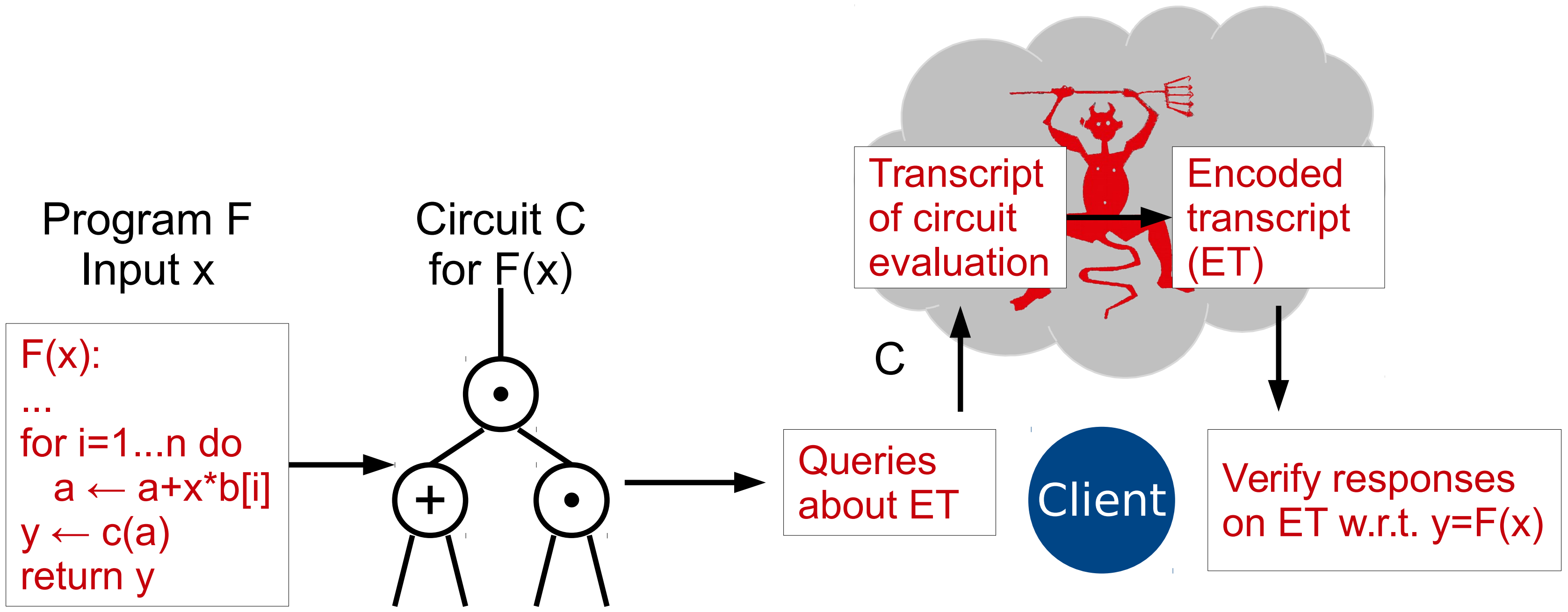
Background

- Zero-Knowledge Proofs (ZKPs)
  - Have similar goals as VC but generally do not achieve efficiency
- Probabilistically Checkable Proofs (PCPs)
  - Cost was prohibitive for a long time
  - Many improvements recently make this near-practical
- Technically VC is closely related to a SNARK
  - Succinct Non-interactive ARgument of Knowledge
  - Assume that prover is computationally bounded

→ VC from SNARK is immediate

→ Implementations of VC first construct a SNARK

Implementing verifiable computation



- Efficient program for F(x) is compiled into a circuit C for F(x)
  - Arithmetic or boolean gates
- Server evaluates C and proves that valid assignment exists (= transcript)
- Client verifies that transcript matches result y and y=F(x)
  - Must be faster than recomputing F or evaluating C
    (This condition rules out the existing interactive proof techniques)

How to verify the encoded transcript?

- Client and server run (randomized) interactive proof
  - May require interaction or large communication
  - As pioneered in "Interactive Proofs for Muggles" [GKR08]
  - Extended and implemented by "streaming" interactive proofs [Thaler et al., multiple works]
- Server commits to ET, client checks commitment probabilistically (PCP)
  - Based on ZK arguments and "short PCPs" [IKO07]
  - Implemented by Blumberg, Setty, Walfish et al. (multiple works)
- Client sends query in encrypted form
  - Server computes response and output without knowing checked locations (related to private query to databases - PIR)
  - Server uses FHE to blindly evaluate a garbled circuit for F [GGP10]
  - Realized by encoding F into a **Quadratic Program** [GGPR13]
    - Implemented by "Pinocchio" [PGHR13]
    - Coupled with a specific circuit compiler, in "SNARKs for C" [BCGTV13]

  → Focus here

Quadratic Programs (QPs) for circuits [GGPR13, PGHR13]

- Consider an arithmetic circuit C for F over a finite field $\mathbb{F}$ (also boolean circuit)
  - Quadratic Arithmetic Program (QAP)
- **Main result** [GGPR13] — **There is a QAP of size O(|C|) that computes F.**
- Define how a QAP computes F: $\mathbb{F} \to \mathbb{F}$ (one input and one output, wlog):

  $y = F(x)$ holds iff there exists $(c_3,\ldots,c_N) \in \mathbb{F}^N$, with $(c_0, c_1, c_2) = (1, x, y)$, s.t.

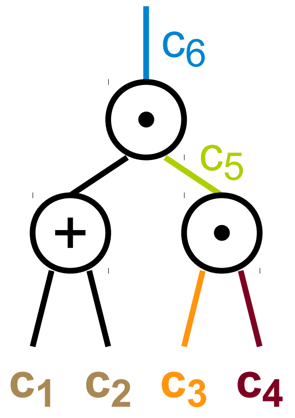
  a given target polynomial $t(z)$ divides $p(z)$,

  where $p(z) = (\Sigma_k\, c_k\, v_k(z)) \bullet (\Sigma_k\, c_k\, w_k(z)) – (\Sigma_k\, c_k\, y_k(z))$.

- QAP = [$\mathbf{V}=\{v_k(z)\}_k$, $\mathbf{W}=\{v_k(z)\}_k$, $\mathbf{Y}=\{v_k(z)\}_k$, $t(z)$]
- **Key step** — $t(z)$ divides $p(z)$ $\equiv$ $\exists\, h(z)$: $h(z) \bullet t(z) = p(z)$
  - Server (prover) picks $h(z)$ based on wires that satisfy C
  - Client (verifier) checks identity of polynomials efficiently, at random point

Converting a circuit to a QAP

Circuit for F(x)



- Pick a root r in $\mathbb{F}$ for each multiplication gate c
  - r_5 for c_5, r_6 for c_6 ...
- Set $t(z) = (z-r_5)(z-r_6)$
- Set $p(z)$ from $\mathbf{V}=\{v_k(z)\}_k$, $\mathbf{W}=\{v_k(z)\}_k$, $\mathbf{Y}=\{v_k(z)\}_k$, one polynomial for each gate g (input and mult. gate), all other points on the polynomials are 0

$c_5 = c_3 \bullet c_4$

$c_6 = c_5 \bullet (c_1 + c_2)$

…

	$z=r_5$	$z=r_6$
$v_1(z)$	0	1
$v_2(z)$	0	1
$v_3(z)$	1	0
$v_4(z)$	0	0
$v_5(z)$	0	0
$v_6(z)$	0	0

wire is left input

	$z=r_5$	$z=r_6$
$w_1(z)$	0	0
$w_2(z)$	0	0
$w_3(z)$	0	0
$w_4(z)$	1	0
$w_5(z)$	0	1
$w_6(z)$	0	0

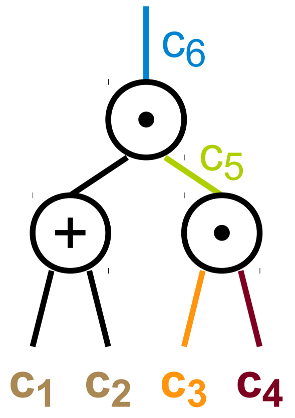
wire is right input

	$z=r_5$	$z=r_6$
$y_1(z)$	0	0
$y_2(z)$	0	0
$y_3(z)$	0	0
$y_4(z)$	0	0
$y_5(z)$	1	0
$y_6(z)$	0	1

wire is gate output

QAP computation of y = F(x)

Circuit for F(x)



$c_5 = c_3 \bullet c_4$

$c_6 = c_5 \bullet (c_1 + c_2)$

…

	$z=r_5$	$z=r_6$
$v_1(z)$	0	1
$v_2(z)$	0	1
$v_3(z)$	1	0
$v_4(z)$	0	0
$v_5(z)$	0	0
$v_6(z)$	0	0

wire is left input

	$z=r_5$	$z=r_6$
$w_1(z)$	0	0
$w_2(z)$	0	0
$w_3(z)$	0	0
$w_4(z)$	1	0
$w_5(z)$	0	1
$w_6(z)$	0	0

wire is right input

	$z=r_5$	$z=r_6$
$y_1(z)$	0	0
$y_2(z)$	0	0
$y_3(z)$	0	0
$y_4(z)$	0	0
$y_5(z)$	1	0
$y_6(z)$	0	1

wire is gate output

- Note $t(z) \mid p(z)$ iff $\forall$ gates g: $t(r_g) = 0 \Rightarrow p(r_g) = 0$
- The polynomials are "extremely sparse"
  - $t(r_5) = 0$ $\quad$ $p(r_5) = (c_3) \bullet (c_4) - (c_5)$
  - $t(r_6) = 0$ $\quad$ $p(r_6) = (c_1+c_2) \bullet (c_5) - (c_6)$
- Thus, p(z) encodes the satisfying assignment of C whenever C computes y = F(x)

Verifiable computation from a QAP (1)

- Idea — Check polynomial identity $h(z) \cdot t(z) = p(z)$ at $s \in_R \mathbb{F}$,
  - s is secret from server
  - Map $h(s)$, $t(s)$, $p(s)$ into exponents of a DL group $G=\langle g \rangle$ with bilinear map $e()$
  - Client uses bilinear map (pairing operation) to check equality
- Client computes KeyGen(F) and ProblemGen(⋯)
  - Create a QAP for F
  - Choose $s \in_R \mathbb{F}$ (randomly)
  - $EK_F \leftarrow [\ \{g^{v_k(s)}\}_k\ ,\ \{g^{w_k(s)}\}_k\ ,\ \{g^{y_k(s)}\}_k\ ,\ g^{(s)^{**i}}$ for $i=1...\mathrm{deg}, \ldots\ ]$
  - $VK_F \leftarrow [\ \{g^{y_k(s)}\}_k\ ,\ g^{t(s)}, \ \ldots\ \ ]$
- Note client's setup complexity must be amortized over many computations

Verifiable computation from a QAP (2)

- Server computes $y = F(x)$ and a wire assignment $\{c_k\}_k$ for C
  - Solves for $h(z)$ such that $h(z) \cdot t(z) = p(z)$
  - Computes $g^{v(s)} = \prod g^{v_k(s) \cdot c_k}$ ... only by linear operations "in the exponent"
  - Similarly, $g^{w(s)} = \prod g^{w_k(s) \cdot c_k}$ and $g^{y(s)} = \prod g^{y_k(s) \cdot c_k}$
  - Proof is short

$$[\ g^{v(s)}, g^{w(s)}, g^{y(s)}, g^{h(s)}\ ]$$

- Client verifies divisibility and other checks with a few pairing operations, e.g.,

$$e(\ g^{v(s)}, g^{w(s)}\ ) \,/\, e(\ g^{y(s)}, g\ ) = e(\ g^{h(s)}, g^{t(s)}\ )$$

- Remarks
  - Security holds under the d-PKE, q-PDH, and 2q-SDH assumptions (d-power knowledge of exponent; q-power Diffie-Hellman; 2q-strong DH)
  - Proof can be as short as 7 group elements (in G)
  - Server needs $O(|C|)$ cryptographic and $O(|C| \cdot \log^2 |C|)$ other operations
  - Actual protocol is more complex

Efficiency comparison

Circuit C of size T, input of size N	Rounds	Server/proof	Client/verif.
With (randomized) interactive proofs			
"Muggles" [GKR08]	d log(N)	poly(T)	O(N+log d)
CMT [Thaler et al.]	streaming	O(T log T)	$O(\log^2 N)$
Based on PCPs			
Blumberg, Setty, Walfish et al.	1	$O(T^{1.5})$	polylog
Encrypted queries based on QAP			
[GGPR13]	1	$O(T \log^2 T)$	const

Systems for verifiable computation

- Ginger [SVP+12], Zaatar [SBV+13], based on PCPs
  - For a C-like language (going back to FairPlay)
- CMT, implementing a streaming interactive protocol [Thaler et al.]
- Pinocchio [PGHR13], based on QAPs
  - Compiles a subset of C into a QAP
- SNARKs for C [BCG+13], based on QAPs
  - With a specific TinyRAM compiler that produces "good" circuits
- Allspice [VSBW13], a hybrid design, combining the best of PCPs and CMT
- Etc.

Typical performance

- Pinocchio [PGHR13]:
  - Evaluate a 5-var. multivariate polynomial with deg. 10 and 644k coefficients
  - Setup (KeyGen/ProblemGen): 42s
  - Compute: 246s
  - Verify: 12.5ms
  - |EK| = 56MB, |VK| = 640B, |Proof| = 288B

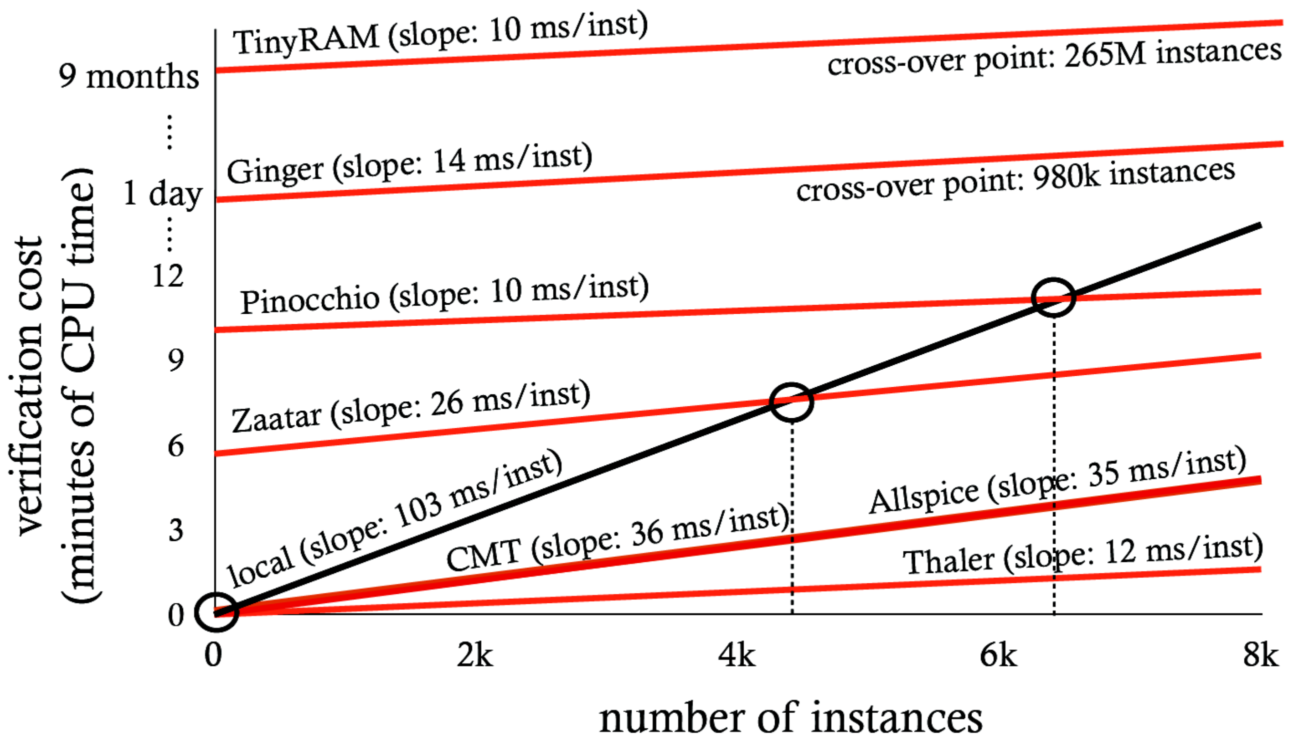
- Other systems are similar, Pinocchio is one of the most efficient

- Useful only for computations amortized over many instances
  - Setup for client exists always

- None is really practical today

- Server cost is much, much bigger than simply computing F

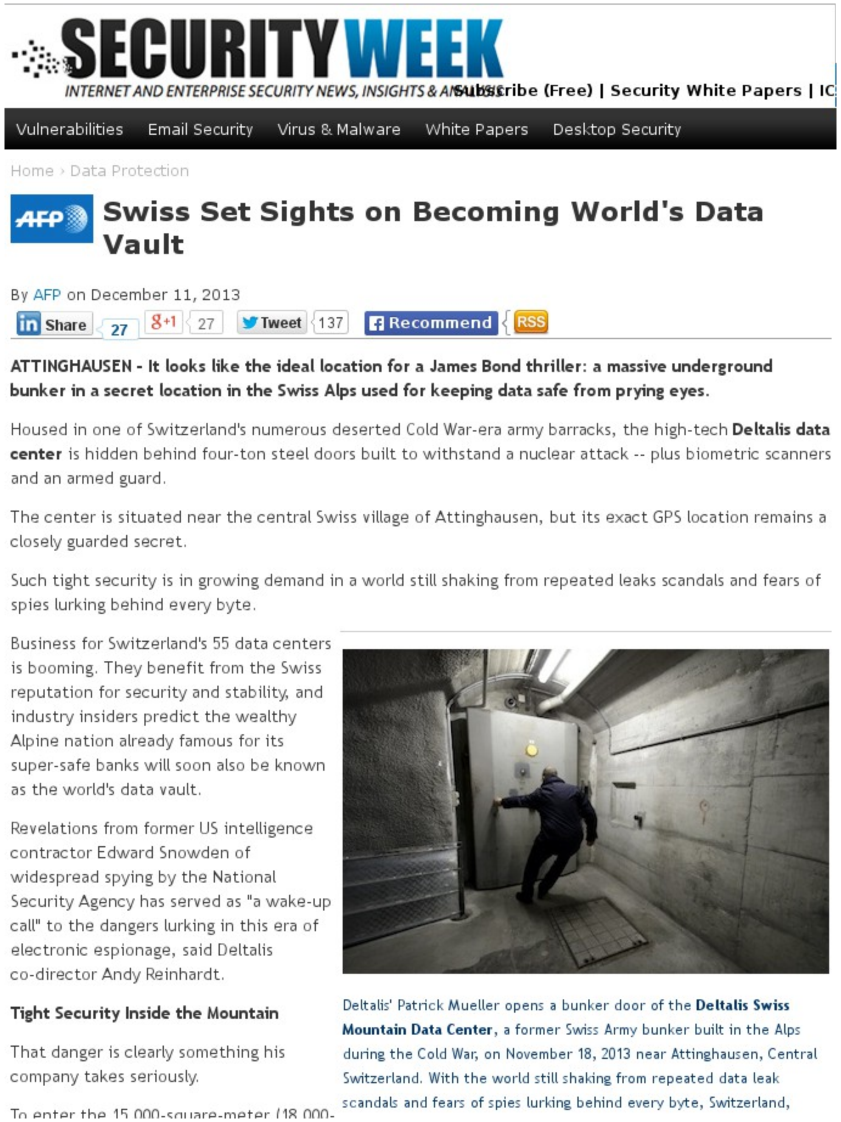
Cross-over points where amortization makes VC rational



- Figure source: [WB13],
- Multiplication of two 128x128 matrices, 64-bit floating point numbers
- Number of instances must be "large" before VC is cheaper than evaluating problem itself

Authenticated data types

Protecting outsourced data


SECURITYWEEK

INTERNET AND ENTERPRISE SECURITY NEWS, INSIGHTS & A Subscribe (Free) | Security White Papers | IC

Swiss Set Sights on Becoming World's Data Vault

By AFP on December 11, 2013

ATTINGHAUSEN - It looks like the ideal location for a James Bond thriller: a massive underground bunker in a secret location in the Swiss Alps used for keeping data safe from prying eyes.

Housed in one of Switzerland's numerous deserted Cold War-era army barracks, the high-tech **Deltalis data center** is hidden behind four-ton steel doors built to withstand a nuclear attack -- plus biometric scanners and an armed guard.

The center is situated near the central Swiss village of Attinghausen, but its exact GPS location remains a closely guarded secret.

Such tight security is in growing demand in a world still shaking from repeated leaks scandals and fears of spies lurking behind every byte.

Business for Switzerland's 55 data centers is booming. They benefit from the Swiss reputation for security and stability, and industry insiders predict the wealthy Alpine nation already famous for its super-safe banks will soon also be known as the world's data vault.

Revelations from former US intelligence contractor Edward Snowden of widespread spying by the National Security Agency has served as "a wake-up call" to the dangers lurking in this era of electronic espionage, said Deltalis co-director Andy Reinhardt.

Tight Security Inside the Mountain

That danger is clearly something his company takes seriously.

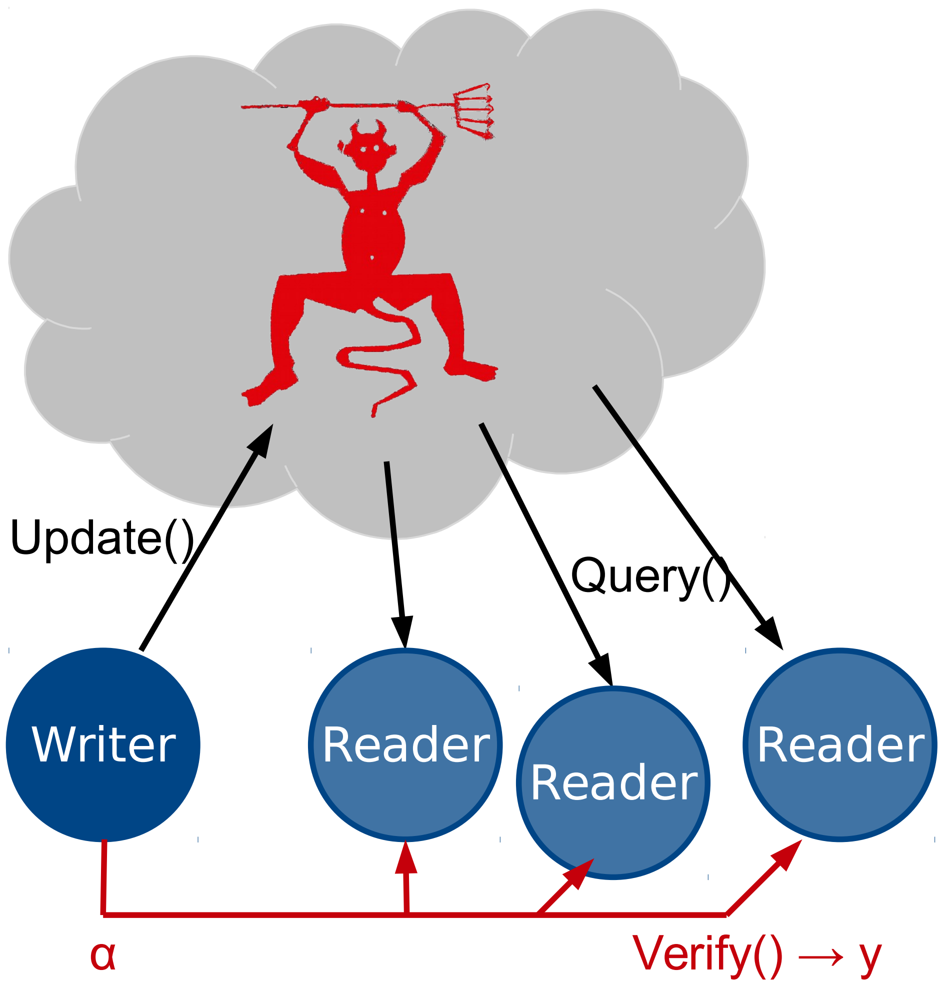
To enter the 15,000-square-meter (18,000-

Deltalis' Patrick Mueller opens a bunker door of the **Deltalis Swiss Mountain Data Center**, a former Swiss Army bunker built in the Alps during the Cold War, on November 18, 2013 near Attinghausen, Central Switzerland. With the world still shaking from repeated data leak scandals and fears of spies lurking behind every byte, Switzerland,

- Data stored remotely
  - Access from everywhere
  - Disaster recovery
  - Long-term archiving
- Data distributed over remote provider
  - Authorized publication
  - Protect critical information
- Database records, reports, files, audit logs ...
- How to prevent tampering by storage hosting provider or by the content-distribution service?

Authentication of remote data

- Writer publishes data
  - Issues incremental updates
  - Data is large
- Server (untrusted)
  - Stores data
  - Processes updates
  - Responds to queries
- Multiple readers
  - Selectively retrieve data with queries
  - Verify authenticity w.r.t. short authenticator α produced by writer



Database service model

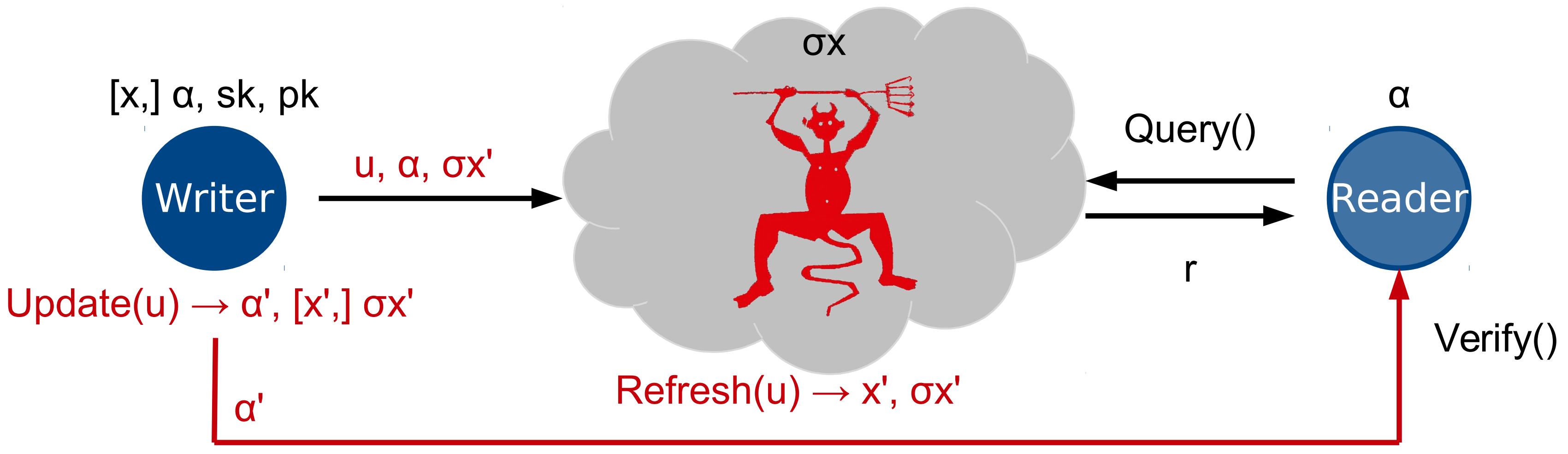
- Generic function F with state in $\mathbf{X}$ and operations $\mathbf{O}$

  $$F(x, o) \to (x', r)$$

  - State $x \in \mathbf{X}$
  - Operation $o \in \mathbf{O}$
  - New state $x' \in \mathbf{X}$
  - Response $r \in \mathbf{R}$

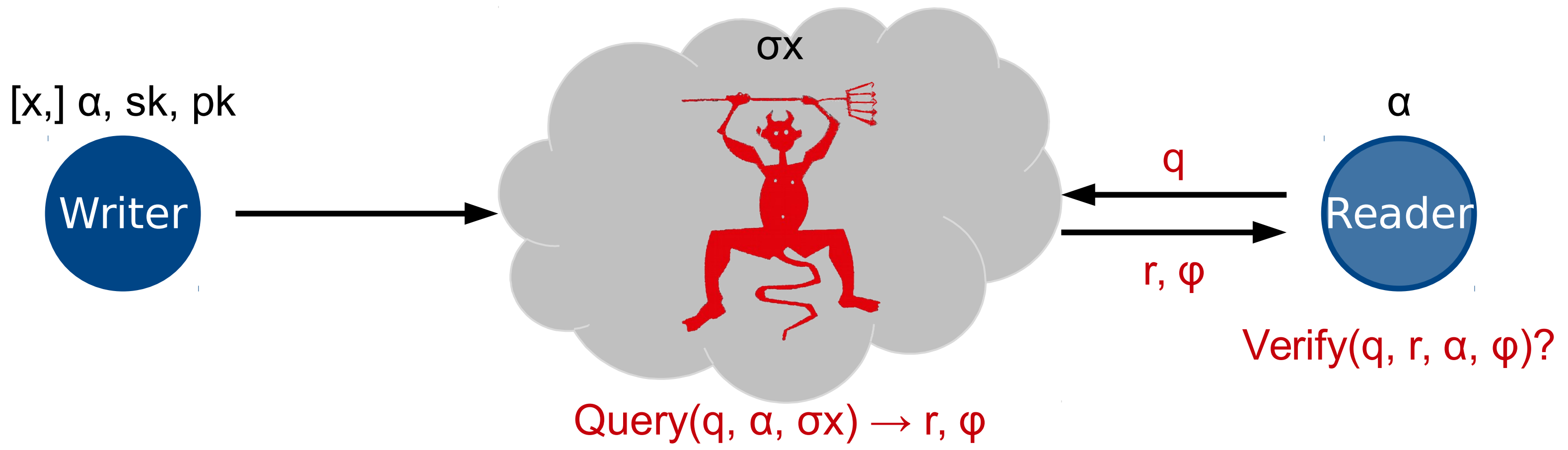
- Operations contain updates $\mathbf{U}$ and queries $\mathbf{Q}$ with $\mathbf{O} = \mathbf{U} \cup \mathbf{Q}$
  - Updates do not give a response: for $u \in \mathbf{U}$, it holds $F(x, u) = (x', \perp)$
  - Queries never change the state: for $q \in \mathbf{Q}$, it holds $F(x, q) = (x, r)$

Authenticated data types (ADT) — Model and setup



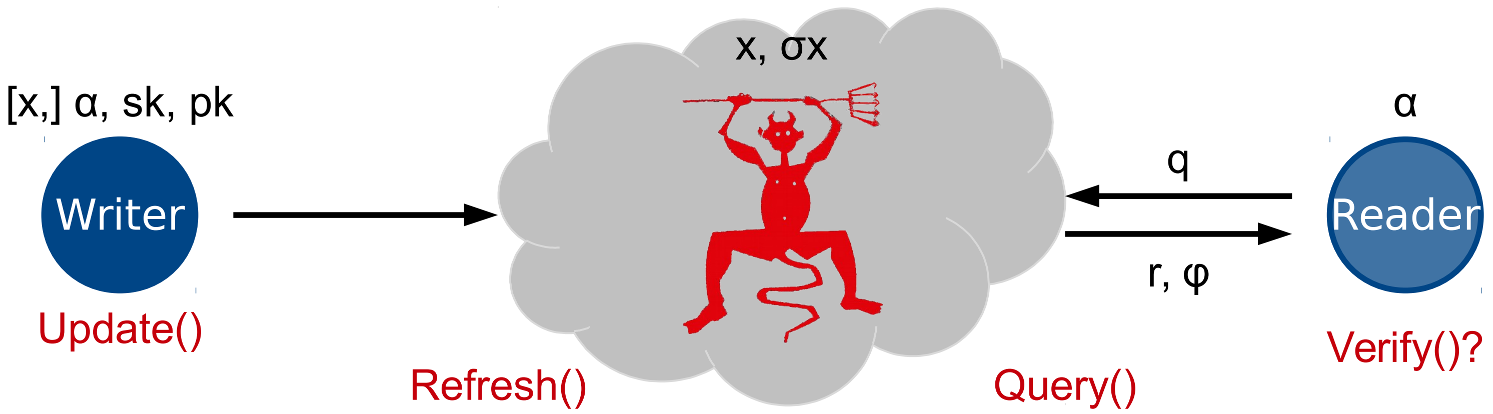
- $\mathrm{KeyGen}(\kappa) \rightarrow (pk, sk)$
  - Generates a public/secret key pair
- $\mathrm{Init}_F(pk, sk, x) \rightarrow (\alpha, \sigma x)$
  - Writer transforms state x of F into encoded σx, obtains authenticator α
- $\mathrm{Update}_F(u, \alpha, [x,]\ \sigma x, pk, sk) \rightarrow (\alpha', [x',]\ \sigma x')$
  - Writer performs update u, obtains new α' and σx' , possibly also x
- $\mathrm{Refresh}_F(u, \alpha, \sigma x, pk) \rightarrow (\alpha', \sigma x')$
  - Server performs update u on encoded state, σx → σx', uses no secret key

Authenticated data types (ADT) — Retrieval



- $\text{Query}_F(q, \alpha, \sigma x, pk) \rightarrow (r, \varphi)$
  - Server produces response r and proof φ for query q
- $\text{Verify}_F(q, r, \alpha, \varphi, pk) \rightarrow \{0, 1\}$
  - Client verifies response r and proof φ w.r.t. authenticator α
- Intuition — Server cannot forge a response and proof such that client accepts wrong value

Authenticated data types (ADT) — Properties



- Correctness
  - Set up and perform updates, resulting in state x and α
  - For all queries q, and responses r from Query(q, ···)

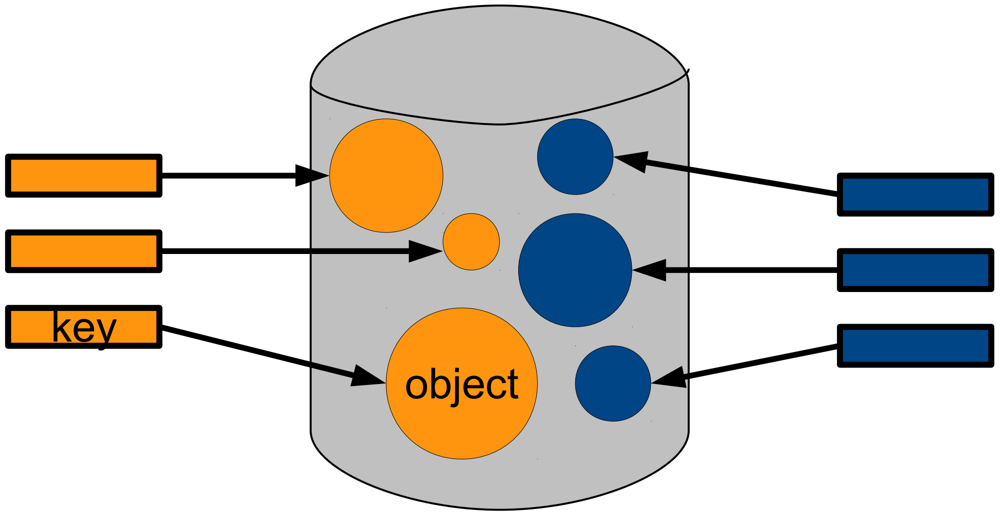
    Verify(q, r, ···) = 1 iff F(x,q) = (···, r)
- Security
  - Adversary **A** lets writer execute operations to produce state x and α
  - Then, **A** cannot forge q, r, φ, such that

    $\text{Verify}_F(q, r, \alpha, \varphi, pk) = 1$ but $F(x, q) \neq (\cdots, r)$
- Efficiency — Sizes of α and φ << size of x

Examples of ADT schemes

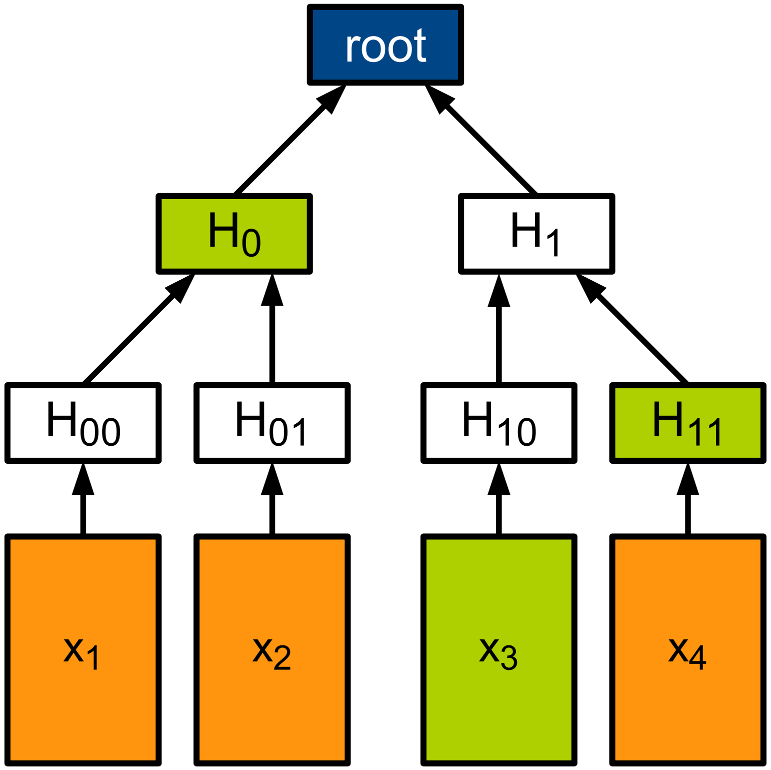
- Original motivation — certificate revocation lists (CRL) [NN00, MND+04]
- Merkle hash tree
  - Application to memory checking [BEGKN94]
- Authenticated dictionaries
- Many hierarchical indexing structures [GTT09]
  - Authenticated skip lists
  - B-trees etc.
  - Supporting 1-d and multi-dimensional range queries
- Sets with verifiable operations [CPPT14]
- Simple database functions
  - Updates and queries

Key-value store (KVS) — Authenticated dictionary



- Popular storage interface using unstructured objects ("blobs")
  - Every object ("value") identified by a unique name ("key")
  - Objects may be grouped into containers
- Practical use in many cloud storage systems (AWS S3, OpenStack Swift ...)
- Operations
  - put(key, val)
  - get(key) → val
  - list() → {keys ...}
  - remove(key)

(Merkle) Hash trees



- Parent node is hash of its children
- Top hash value (root) commits all data items $x_1, \ldots, x_n$
  - Root hash is authenticated, cryptographically protected or in trusted memory
  - Tree is on extra untrusted storage
- To verify x_i, recompute path from x_i to root with sibling nodes and compare to root
- To update x_i, recompute new root hash and nodes along path from x_i to root

Authenticated dictionary using a hash tree

- Dictionary stores key/value entries (k, v)
- Store (k, v) in hash tree leaves?
  No — does not allow to prove absence of a key
- For proving membership and non-membership, hash tree contains tuples
  (k_i, k_{i+1}, v_i)
- Uses only a cryptographic hash function, no keys
- Update, Refresh
  - Recompute all nodes along path from modified entry to root
  - Authenticator α is the root hash of tree
  - Hash tree is stored by untrusted server
- Query, Verify
  - Proof φ consists of sibling nodes along path from entry to root
  - Verification recomputes root hash and compares to α

Dynamic cryptographic accumulators [BP97, CL02]

- A dynamic accumulator is a cryptographic abstraction for collecting data values into a short digest and for checking their presence efficiently:
  - Init() → (a, pk, sk) — generates accumulator value a and a key pair pk/sk
  - Add(a, x, pk) → a' — adds x to accumulator
  - Delete(a, x, pk, sk) → a' — removes x from accumulator
  - Witness(a, x, pk) → w — produces a witness w for presence of x
  - Verify(a, x, w, pk) → {0, 1} — checks if witness w is valid for x and proves that x was added to accumulator
- History independence — The accumulator value a does not depend on the order of adding values, that is, adding values x and y is quasi-commutative:
  Add(Add(a, **x**, pk), **y**, pk) = Add(Add(a, **y**, pk), **x**, pk)
- Security — Given an accumulator value a (produced "under influence" of the adversary), it is infeasible to create x', w' without sk such that
  Verify(a, x', w', pk) = 1

Accumulator based on the strong RSA problem [CL02,GTH02]

- Take an RSA modulus $N = P \cdot Q$ (with P, Q safe primes), and $r \in \mathbf{Z}_N$
  – Secret key consists of factorization (P, Q)
- Strong RSA assumption — It is infeasible to find a, b s.t. $a^b = r \mod N$
- Accumulator α that "contains" $x_1, \ldots, x_n$ is $\alpha = r^{H(x1) \bullet \cdots \bullet H(xn)} \mod N$
  – Hash function H maps entries to distinct primes
  – Add an element x_i to α by computing $\alpha' \leftarrow \alpha^{H(xi)} \mod N$
  – Delete an element x_i from α by computing $\alpha' \leftarrow \alpha^{1/H(xi)} \mod N$ (secret key!)
  – Witness for x_i in α is $w_i \leftarrow \alpha^{1/H(xi)} \mod N$
    - Witness may also be computed without the secret key
      $w_i \leftarrow r^{H(x1) \bullet \cdots H(x[i-1]) \bullet H(x[i+1]) \bullet \cdots \bullet H(xn)} \mod N$
  – Verify that x_i is contained in α by checking $w_i^{H(xi)} = \alpha \mod N$?

Authenticated dictionary using the dynamic accumulator

- For key/value pairs (k, v) the dictionary contains tuples (k_i, k_{i+1}, v)
- The authenticator is the accumulator value α
  - Writer needs the secret key for updates
- Update
  - For put(), the writer adds the new tuple to α, perhaps removes the old first
  - For remove(), the writer removes the tuple from α, updates predecessor
- Refresh
  - Server recomputes all witnesses (expensive!)
    - Needs Θ(n) exponentiations (can be improved to O(1) and sublinear query-response cost, when amortized and server keeps state [PTT08])
- Query, Verify
  - Proof φ consists witness w for the key in get() and the accumulator α
  - Verification works according to accumulator's Verify()

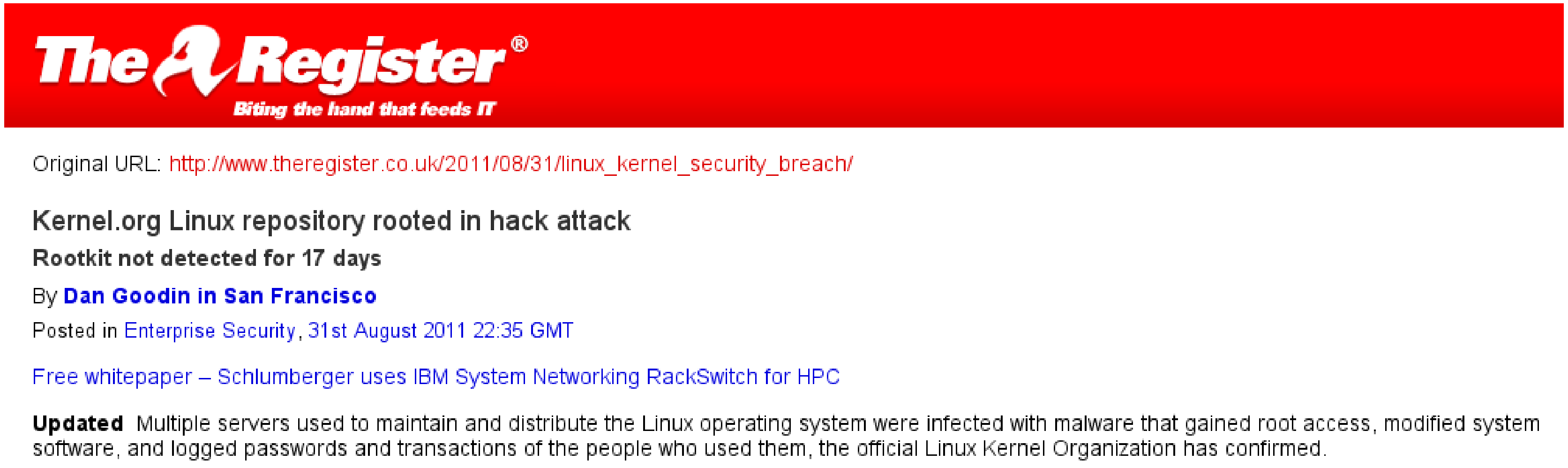
Efficiency comparison

Authenticated dictionary of size n	Hash tree	Accumulator	
Update time	O(log(n))	O(1)	
Refresh time	O(log(n))	O(n)	(!)
Verify time	O(log(n))	O(1)	
Proof size	O(log(n))	O(1)	

- In practice, public-key operations in accumulator scheme dominate cost [CW11]
  - Accumulator schemes are "unsuitable for production use"
- In a direct comparison, the small proofs of accumulators do not pay off [CW11]
  - One RSA modulus would correspond to a path in a hash tree of depth 17, holding 256'000 keys

Distributed consistency enforcement

Cloud storage integrity — consistent to multiple clients?


Original URL: http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/

Kernel.org Linux repository rooted in hack attack

Rootkit not detected for 17 days

By **Dan Goodin in San Francisco**

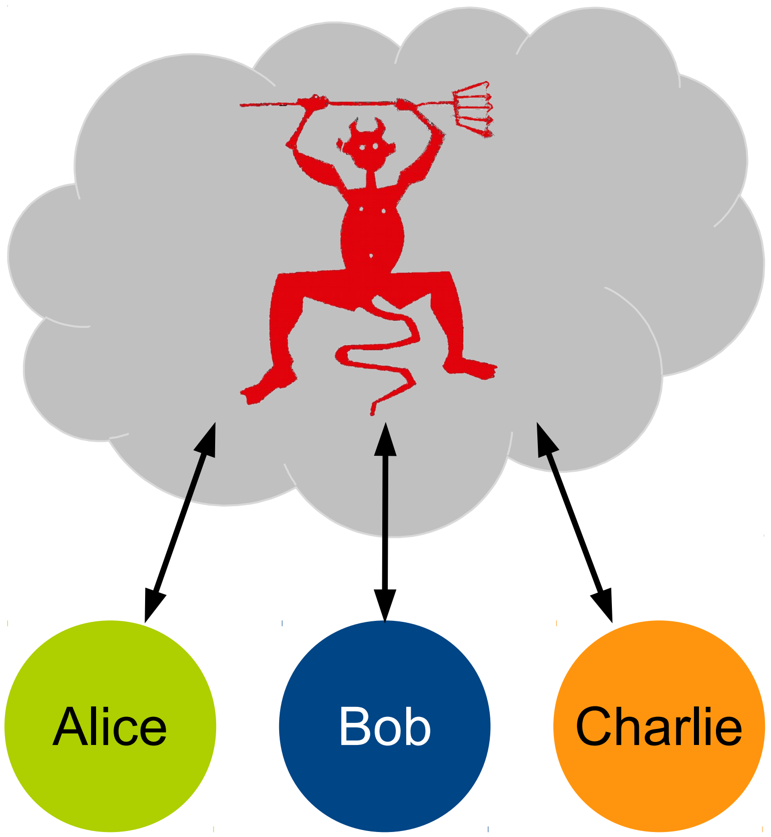
Posted in Enterprise Security, 31st August 2011 22:35 GMT

Free whitepaper – Schlumberger uses IBM System Networking RackSwitch for HPC

Updated Multiple servers used to maintain and distribute the Linux operating system were infected with malware that gained root access, modified system software, and logged passwords and transactions of the people who used them, the official Linux Kernel Organization has confirmed.

- Kernel.org Linux repository was compromised
  - Linux kernel sources exposed, but public open-source anyway
  - Thanks to cryptographic integrity protection in revision control system (git), kernel code modifications could be detected
  - Who determines the "true" kernel sources?
  - What if cloud service is subverted or client data are modified?

System model



- Server S
  - Normally correct
  - Sometimes faulty (untrusted, potentially malicious ... Byzantine)
- Clients: A, B, C ... (n in total)
  - Correct, may crash
  - Invoke operations on server
  - Disconnected
  - Small trusted memory
- Asynchronous
- No client-to-client communication

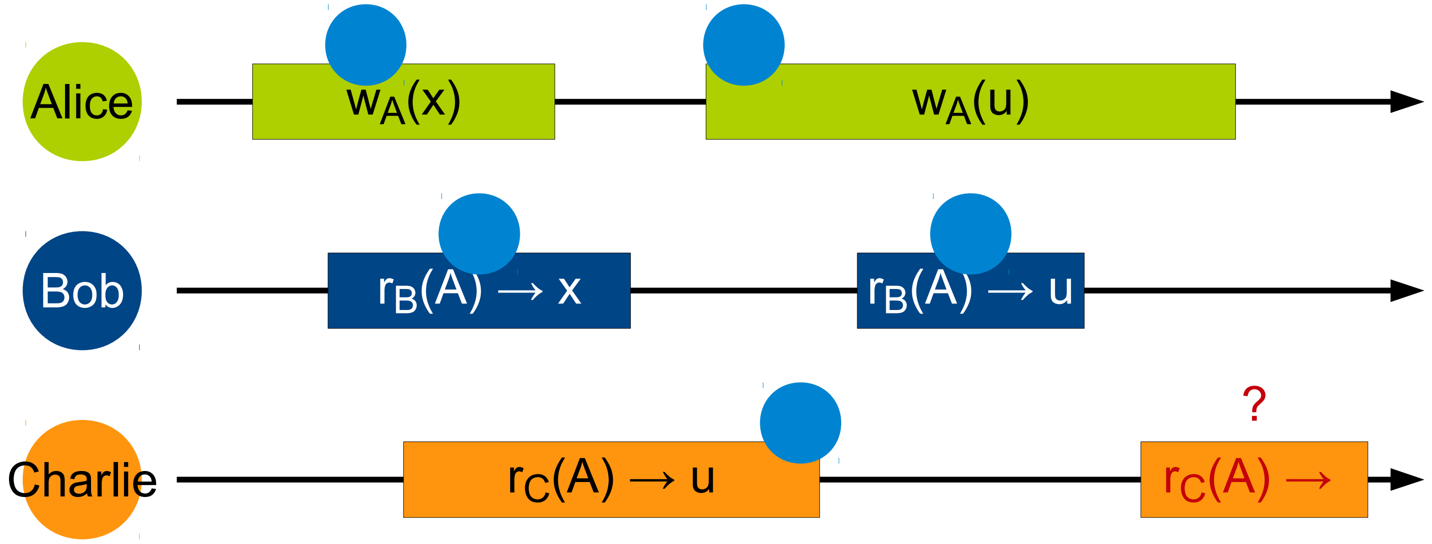
Consistency among verifiers?

- Verifiers need to be synchronized
  - In ADT, the writer publishes authenticator through a trusted channel
  - Applies also to stateful VC
- In principle, verifiers could agree among themselves on "the" authenticator
  - However, clients often cannot communicate
  - Contradicts the model of outsourced computation
- Thus, clients may observe different views → distributed computing

An abstract storage service

- Storage functionality MEM
  - Array of registers $x_1, ..., x_n$ // one register for every client
  - A register has two operations
    - Write(i, x) → ok // updates stored value x_i to x
    - Read(i) → x_i // returns value x_i stored at index i
- Popular model for shared storage
- Clients access MEM asynchronously
  - Every operation defined through an invocation and a response
- Consistency properties when operations execute concurrently
  - Sequential consistency
  - Regular semantics
  - Atomic semantics (linearizability)

Semantics of concurrent operations [L86, HW90]



- (Safe — Every *read* not concurrent with a *write* returns the most recently written value.)
- (Regular — Safe & a *read* concurrent with a *write* returns the most recently written value or the concurrently written value. $r_C(A) \to x$ or u.)
- Atomic (linearizable) — Regular & all *read* and *write* operations occur atomically; all clients observe the same sequence of operations. $r_C(A) \to u$!

Linearization points ●

Linearizability formally

- A history σ is linearizable (w.r.t. F)

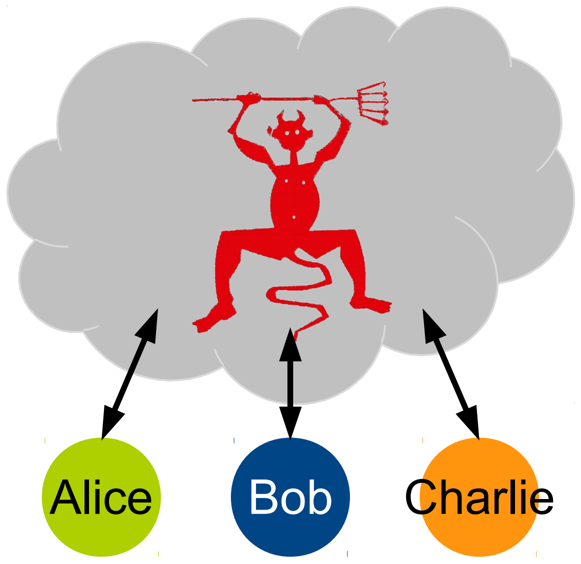
  ↔ ∃ a permutation π of σ such that

  - π is sequential and adheres to the sequential specification (of F)
  - ∀ clients c, the operations of c are in σ
  - π preserves the real-time order of σ.

Data on untrusted storage

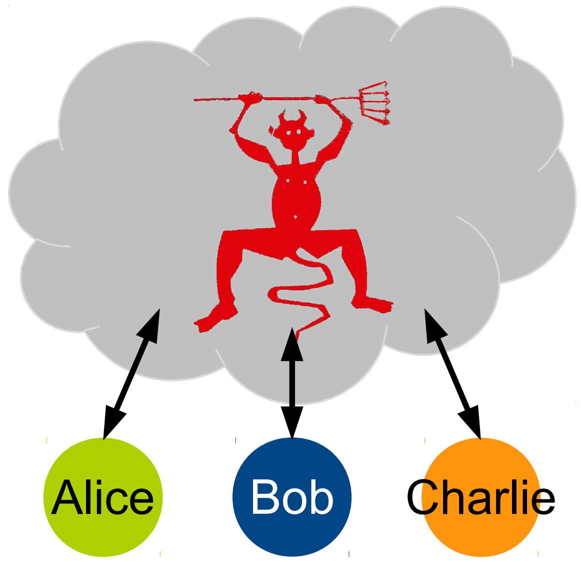
- Suppose clients can only write to and read from untrusted server
  - No outside communication or synchronization

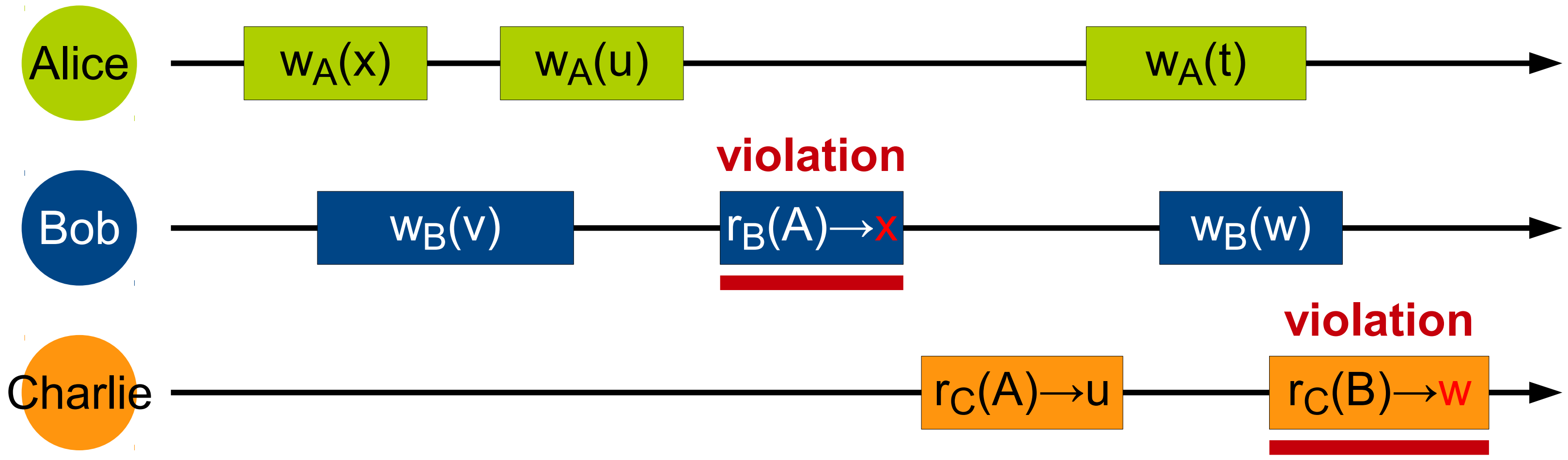
- Adding cryptographic authentication (digital signatures or MACs) protects clients' data
  - Server cannot forge values out of the blue
  - But, answer with an outdated value: "Replay attack" violates consistency
  - But, send different values (some outdated) to different clients: Violates consistency



Problem illustration

- Bob cannot detect the replay attack, $r_B(A) \rightarrow x$
- Charlie cannot detect the inconsistency between $r_C(A) \rightarrow u$ and $r_C(B) \rightarrow w$

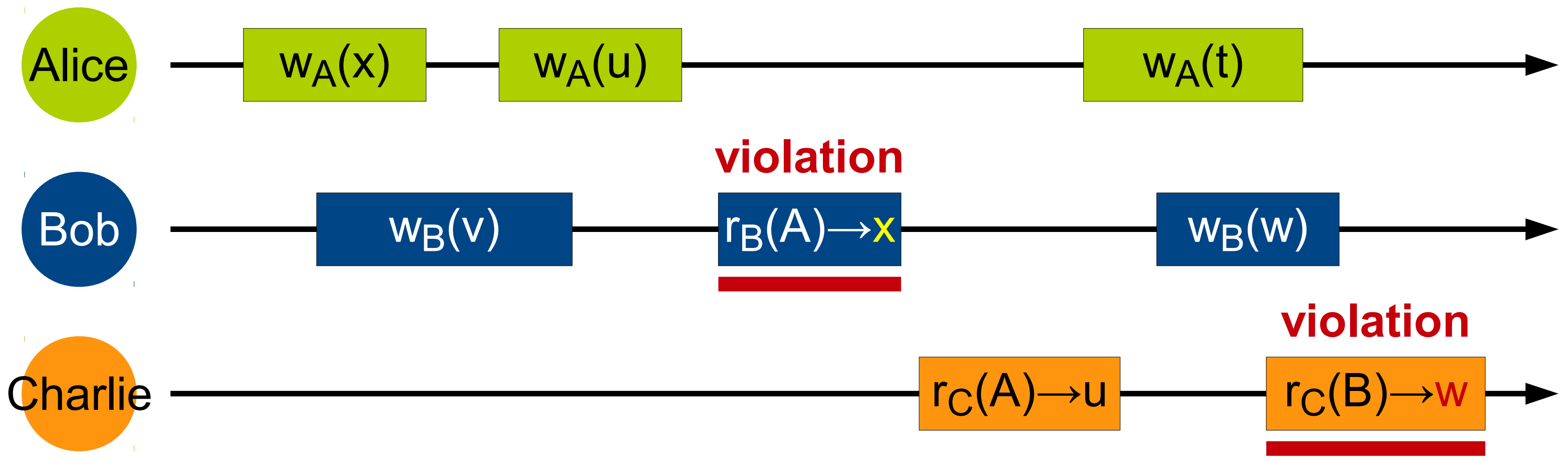




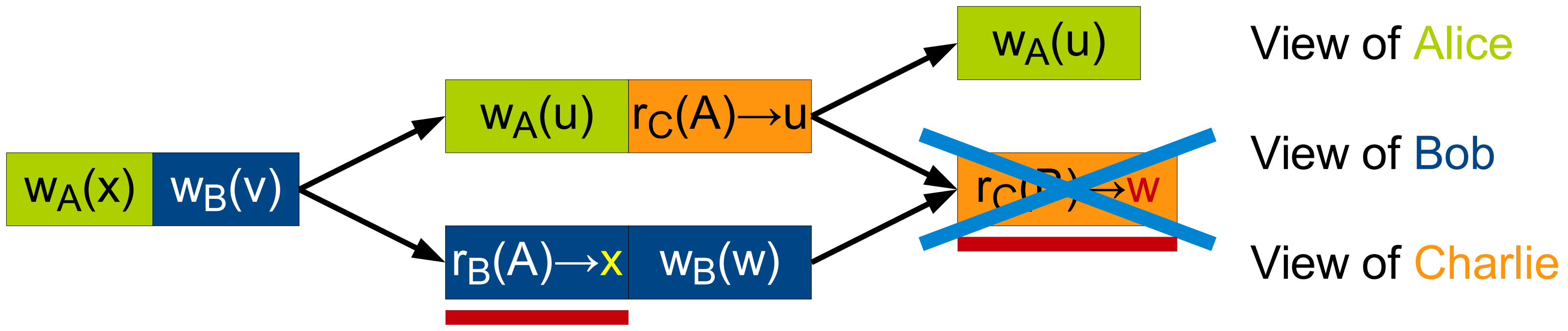
Fork-linearizability as a solution

- Server may replay old state and present different views to clients
  - Thereby "fork" their views of history
  - Clients cannot detect this unless they communicate
- A protocol that imposes fork-linearizability ensures
  - If the views of two clients are forked once, they are forked forever
    - They never again see each other's updates
  - Or they can detect the server's violation
- Every consistency violation results in a fork
  - Best achievable guarantee with an untrusted server
- Forks can be exposed on an external channel with low capacity and security
  - Synchronized clocks
  - Periodic gossip
- Introduced by Mazières and Shasha in SUNDR [MS02]

Fork-linearizability illustrated



- Cannot prevent forks: violation at $r_B(A) \to x$
- Prevents joins: violation at $r_C(B) \to w$



Fork-linearizability formally [MS02, CSS07]

- A history σ is fork-linearizable (w.r.t. F)

  $\leftrightarrow \forall$ clients c, $\exists$ a subset $\sigma(c) \subseteq \sigma$ and a permutation $\pi(c)$ of $\sigma(c)$ s.t.

  - The operations of c are in $\sigma(c)$;
  - $\pi(c)$ is sequential and adheres to the specification (of F);
  - $\pi(c)$ preserves the real-time order of $\sigma(c)$; and
  - If $o \in \pi(c) \cap \pi(c')$, then $\pi(c) = \pi(c')$ up to o.

- If two clients both observe the same operation o, then their views are the same up to to o.

Fork-linearizable Byzantine emulations [CSS07]

- Protocol P emulates functionality F on a Byzantine server S with fork-linearizability whenever
  - If S is correct, then the history of every (...) execution of P is linearizable w.r.t. F;
  - The history of every (...) execution of P is fork-linearizable w.r.t. F.

Trivial protocol to ensure fork-linearizability

- Suppose clients may cryptographically authenticate messags
  - Digital signature schemes
  - Alternatively, they share the key for a MAC

- Idea — Sign the complete history [MS02]
  - Server sends history with all signatures (one sig. on every prefix)
  - Client verifies all operations and all signatures
  - Client adds its operation, signs new history
  - Client sends back operation and signature

- Provides a fork-linearizable Byzantine emulation

- Not practical because messages and history grow with system age

Efficient fork-linearizable storage (1)

- Client C
  - Stores timestamp (counter) t_C
  - Stores version (vector of timestamps) T, where $T[C] = t_C$
  - At every operation, client increments t_C and updates T
  - Signs operations and versions

- Versions order operations
  - For every operation, client increments timestamp, signs version and data

$$V = \begin{bmatrix} v_A \\ v_B \\ v_C \end{bmatrix}$$

- Check consistency between V of current op. and version T of last completed op.
  - Version of subsequent operation must be $V \geq T$
  - Cryptographic authentication must be valid

Efficient fork-linearizable storage (2)

Alice

$T = \begin{bmatrix} t_A \\ t_B \\ t_C \end{bmatrix}$

$V = \begin{bmatrix} v_A \\ v_B \\ v_C \end{bmatrix}$

$\xrightarrow{\text{[SUBMIT, read, c']}}$

Values x, y, z ...
Signature φ of last op. by some cient d

$\xleftarrow{\text{[REPLY, V, ...x', d, φ]}}$

V >= T ?
verify $v_A = t_A$?
verify (φ, V|...x') ?
if not then abort()

$T \leftarrow V$
$t_A \leftarrow t_A + 1$
$\omega \leftarrow \text{sign}(T|...x_A)$

return(x')

$\xrightarrow{\text{[COMMIT, T, ω]}}$

$V \leftarrow T$
$\varphi \leftarrow \omega$
$d \leftarrow A$

Efficient fork-linearizable storage (3)

- If clients are forked, they will sign and store incomparable versions

$$\begin{bmatrix} v_A \\ v_B+1 \\ v_C \end{bmatrix} \quad ??? \quad \begin{bmatrix} v_A+1 \\ v_B \\ v_C \end{bmatrix}$$

- Authentication of versions and values prevents any other server manipulation
  – With n clients, protocol uses O(n) extra memory for emulating fork-linearizable shared memory on an untrusted server

- Clients must proceed in lock-step mode
  – Clients may be blocked
  – Wait-free protocols would be desirable instead

- How to increase concurrency? → See later

Benefits of fork-linearizable protocol

- Suppose client A writes many values: u, v, w, x, y, z ...
- Without distributed consistency enforcement
  - Server can show any of these values to another client B
- With fork-linearizable emulation
  - Client A writes z and tells B out-of-band
  - Client B reads r from location of z
    - If $r = z$, then all values that B read so far were correct
    - If $r \neq z$, then server is faulty and has violated consistency
  - Out-of-band communication might be only synchronized clocks

Fork-linearizable protocols are blocking

- All fork-linearizable emulations of storage have executions with a correct S, where some client A must wait for a client B [CSS07]
  - Due to conflicting operations
- Blocking needed also beyond fork-linearizability
  - Fork-sequentially consistent storage emulations are blocking
  - Fork-*-consistent [LM07] storage emulations are blocking
- Weak fork-linearizability, as implemented in FAUST, eliminates the need for blocking [CKS11]
  - Leaves out last operation of a client

Storage systems with forking consistency notions

- SUNDR [MS02, LKMS04]
  - Secure untrusted data repository
  - Overlay to an NFS file system, $O(n^2)$ space overhead
- CSVN [CG09]
  - Integrity-protecting Subversion revision-control system
  - $O(n)$ space overhead based on [CSS07]
- FAUST — Fail-aware untrusted storage [CKS11]
  - Never blocks, but weaker semantics
  - Uses sporadic client-to-client messages
- Venus [SCC+10]
  - Integrity protection for simple cloud object storage
  - Implements protocol very similar to FAUST
- Depot: Cloud storage with minimal trust [MSL+11]
  - Fork-causal consistency

Forking consistency for generic computations

- Consider generic function F that supports an authenticated data type
- Recall F with state in $\mathbf{X}$ and operations $\mathbf{O}$

$$F(x, o) \rightarrow (x', r)$$

  – State $x \in \mathbf{X}$, operation $o \in \mathbf{O}$, new state $x' \in \mathbf{X}$, response $r \in \mathbf{R}$

- Operations contain updates $\mathbf{U}$ and queries $\mathbf{Q}$ with $\mathbf{O} = \mathbf{U} \cup \mathbf{Q}$
  – Queries never change the state: for $q \in \mathbf{Q}$, it holds $F(x, q) = (x, \bullet)$

- Recall ADT operations: $\text{Init}_F()$, $\text{Update}_F()$, $\text{Refresh}_F()$, $\text{Query}_F()$, $\text{Verify}_F()$.

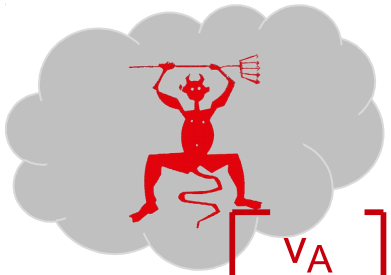
Approach [C11]

- Extend the fork-linearizable storage protocol
  - Server stores the state of F
  - Clients maintain only versions
- Fork-linearizable operation execution
  - Client announces operation o = u or o = q to S
  - For an update u:
    - S extracts necessary state for $\text{Update}_F()$ and sends to client
    - Client runs $\text{Update}_F()$, signs new authenticator, and sends to S
    - S runs $\text{Refresh}_F()$
  - For a query q:
    - S runs $\text{Query}_F()$ and sends response to client
    - Client runs $\text{Refresh}_F()$ and outputs response
    - Client signs authenticator again, and sends to S
- Protocol is lock-step
  - Wait-free progress is possible when operations commute [WSS09, CO13]

Fork-linearizable authenticated computation

Alice

$T = \begin{bmatrix} t_A \\ t_B \\ t_C \end{bmatrix}$



$V = \begin{bmatrix} v_A \\ v_B \\ v_C \end{bmatrix}$

$\xrightarrow{[\text{SUBMIT}, o]}$

State x
Authenticator α
Signature φ of last op. by d
If o = update, then extract state
If o = query, then $r \leftarrow \text{Query}_F(s)$

$\xleftarrow{[\text{REPLY}, V, \alpha, \text{state}/q, d, \varphi]}$

$V >= T$?
verify $v_A = t_A$?
verify $(\varphi, V|\alpha)$?
--verifyF(q, r, α) ?
if not then abort()

--$\alpha \leftarrow$ UpdateF(u, α)

$T \leftarrow V$
$t_A \leftarrow t_A + 1$
$\omega \leftarrow \text{sign}(T|\alpha)$

--$\text{Refresh}_F(u, \alpha, \ldots)$

$\xrightarrow{[\text{COMMIT}, T, \omega, \alpha]}$

$V \leftarrow T$
$\varphi \leftarrow \omega$
$d \leftarrow A$

return(x')

Protocol properties

- If server is correct, then all executions are linearizable
  - Correct server schedules operations in as they arrive
- With faulty server, executions are fork-linearizable w.r.t. F
  - Follows from versions and from authenticated data type for F
- Complexity
  - Three messages
  - Size O(n) with n cients, plus ADT authenticator
- Additionally, support for commuting concurrent operations [CO13]
  - If an operation commutes with other operations not yet committed (by other clients), then they can be executed concurrently

Hash chain instead of vector clock

- Replace vector clock by a hash chain
  - Compact representation of complete operation history

- Reduces communication overhead from O(n) to constant
  - Complicates the protocol when operations execute concurrently

- Used in multiple protocols and systems for fork-linearizable authenticated generic computation (Blind stone tablet, SPORC ...)

Collaboration systems with forking consistency

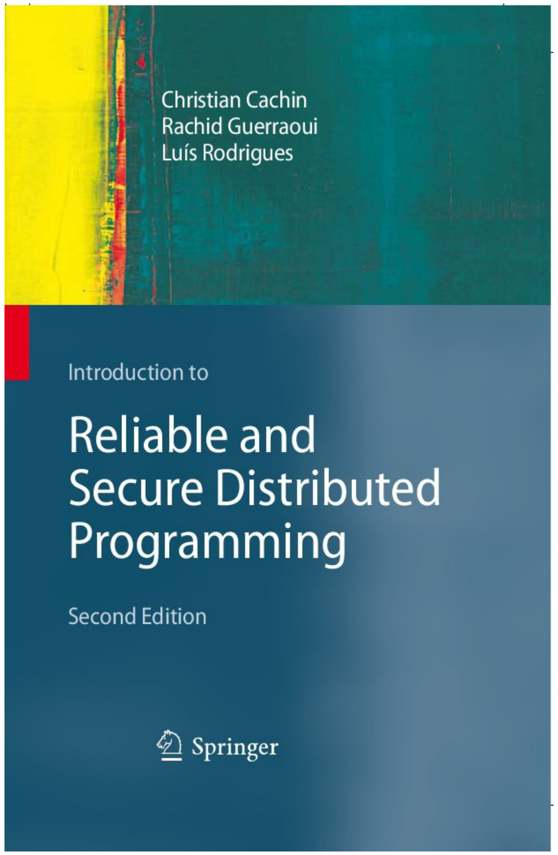
- Blind Stone Tablet [WSS09]
  - Runs a relational database
  - Considers that some operations commute, but no proofs
- SPORC: Group Collaboration using Untrusted Cloud Resources [FZFF10]
  - Editor for shared documents
  - Operational transforms
- Commutative-operation verification protocol (COP) [CO13]
  - Commuting operation sequences proceed without waiting
  - Authenticated data types for complex operations

Conclusion

Conclusion

- Integrity is often more important than confidentiality
- Diverse techniques address the problem
  - Single-client verifiable computation (for generic computations)
  - Single-writer, multiple-reader authenticated data types (specific schemes)
  - Multiple-client interactive computation (for storage and specific schemes)
- Active research area, combines cryptography, distributed systems, and security

More about distributed computing



Introduction to Reliable and Secure Distributed Programming

- C. Cachin, R. Guerraoui, L. Rodrigues
- 2nd ed. of Introduction to Reliable Distributed Programming
- Springer, 2011

www.distributedprogramming.net

Literature (Verifiable computation 1)

[BCG+13] E. Ben-Sasson, A. Chiesa, D. Genkin et al., "SNARKs for C: Verifying program executions succinctly and in zero knowledge," CRYPTO 2013.

[BFR+13] B. Braun, A. J. Feldman, Z. Ren et al., “Verifying computations with state,” SOSP 2013.

[CMT12] G. Cormode, M. Mitzenmacher, and J. Thaler, “Practical verified computation with streaming interactive proofs,” ITCS 2012.

[GGP10] R. Gennaro, C. Gentry, and B. Parno, “Non-interactive verifiable computing: Outsourcing computation to untrusted workers,” CRYPTO 2010.

[GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, “Quadratic span programs and succinct NIZKs without PCPs,” EUROCRYPT 2013.

[GKR08] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, “Delegating computation: Interactive proofs for muggles,” STOC 2008.

Literature (Verifiable computation 2)

[IKO07] Y. Ishai, E. Kushilevitz, and R. Ostrovsky, "Efficient arguments without short PCPs," Computational Complexity (CCC), 2007.

[PGHR13] B. Parno, C. Gentry, J. Howell, and M. Raykova, “Pinocchio: Nearly practical verifiable computation,” IEEE Security & Privacy 2013.

[SBB+13] S. Setty, B. Braun, A. Blumberg et al., "Resolving the conflict between generality and plausibility in verified computation," Eurosys 2013.

[SVP+12] S. Setty, V. Vu, N. Panpalia et al., “Taking proof-based verified computation a few steps closer to practicality,” USENIX Security, 2012.

[VSBW13] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish, “A hybrid architecture for interactive verifiable computation,” IEEE Security & Privacy 2013.

[WB13] M. Walfish and A. Blumberg, "Verifying computations without reexecuting them: from theoretical possibility to near practicality," ECCC, Report 165, 2013. To appear in CACM.

Literature (Authenticated data types 1)

[BEGKN94] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, “Checking the correctness of memories,” Algorithmica, vol. 12, pp. 225–244, 1994.

[BP97] N. Baric and B. Pfitzmann, “Collision-free accumulators and fail-stop signature schemes without trees,” in EUROCRYPT ’97, LNCS 1233, 1997.

[CL02] J. Camenisch and A. Lysyanskaya, “Dynamic accumulators and application to efficient revocation of anonymous credentials,” CRYPTO 2002.

[CPPT14] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos, “Verifiable set operations over outsourced databases,” in Proc. PKC 2014.

[CW11] S. A. Crosby and D. S. Wallach, “Authenticated dictionaries: Real-world costs and trade-offs," ACM TISSEC, vol. 14, no. 2, 2011.

[GTH02] M. T. Goodrich, R. Tamassia, and J. Hasic, “An efficient dynamic and distributed cryptographic accumulator,” in Proc. ISC, LNCS 2433, Springer, 2002.

Literature (Authenticated data types 2)

[GTT09] M. T. Goodrich, R. Tamassia, and N. Triandopoulos, “Efficient authenticated data structures for graph connectivity and geometric search problems,” Algorithmica, vol. 60, pp. 505–552, 2011.

[MND+04] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, “A general model for authenticated data structures,” Algorithmica, vol. 39, pp. 21–41, 2004.

[NN00] M. Naor and K. Nissim, “Certificate revocation and certificate update,” IEEE J. Selected Areas in Communications, vol. 18, pp. 561–570, Apr. 2000.

[PTT08] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables," ACM CCS 2008.

Literature (Distributed consistency 1)

[C11] C. Cachin, "Integrity and consistency for untrusted services," in Proc. Current Trends in Theory and Practice of Computer Science (SOFSEM 2011), LNCS 6543, 2011.

[CG09] C. Cachin and M. Geisler, "Integrity protection for revision control," in Proc. ACNS, LNCS 5536, 2009.

[CKS11] C. Cachin, I. Keidar, and A. Shraer, "Fail-aware untrusted storage," SIAM Journal on Computing, vol. 40, Apr. 2011.

[CO13] C. Cachin and O. Ohrimenko, "On verifying the consistency of remote untrusted services," IBM Research Report (2013) and OPODIS 2014 (to appear).

[CSS07] C. Cachin, A. Shelat, and A. Shraer, "Efficient fork-linearizable access to untrusted shared memory," in Proc. PODC, 2007.

Literature (Distributed consistency 2)

[FZFF10] A. Feldman, P. Zeller, M. Freedman, E. Felten, "SPORC: Group Collaboration using Untrusted Cloud Resources", OSDI 2010.

[LKMS04] J. Li, M. Krohn, D. Mazieres, and D. Shasha, "Secure untrusted data repository (SUNDR)," OSDI 2004.

[MSL+11] P. Mahajan, S. Setty, S. Lee et al., "Depot: Cloud Storage with Minimal Trust", ACM Transactions on Computing Systems, vol. 29, no. 4, 2011.

[MS02] D. Mazieres and D. Shasha, "Building secure file systems out of Byzantine storage," PODC 2002.

[WSS09] P. Williams, R. Sion, and D. Shasha, “The blind stone tablet: Outsourcing durability to untrusted parties,” in Proc. NDSS, 2009.

[SCC+10] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for untrusted cloud storage," CCSW 2010.