University of Pittsburgh

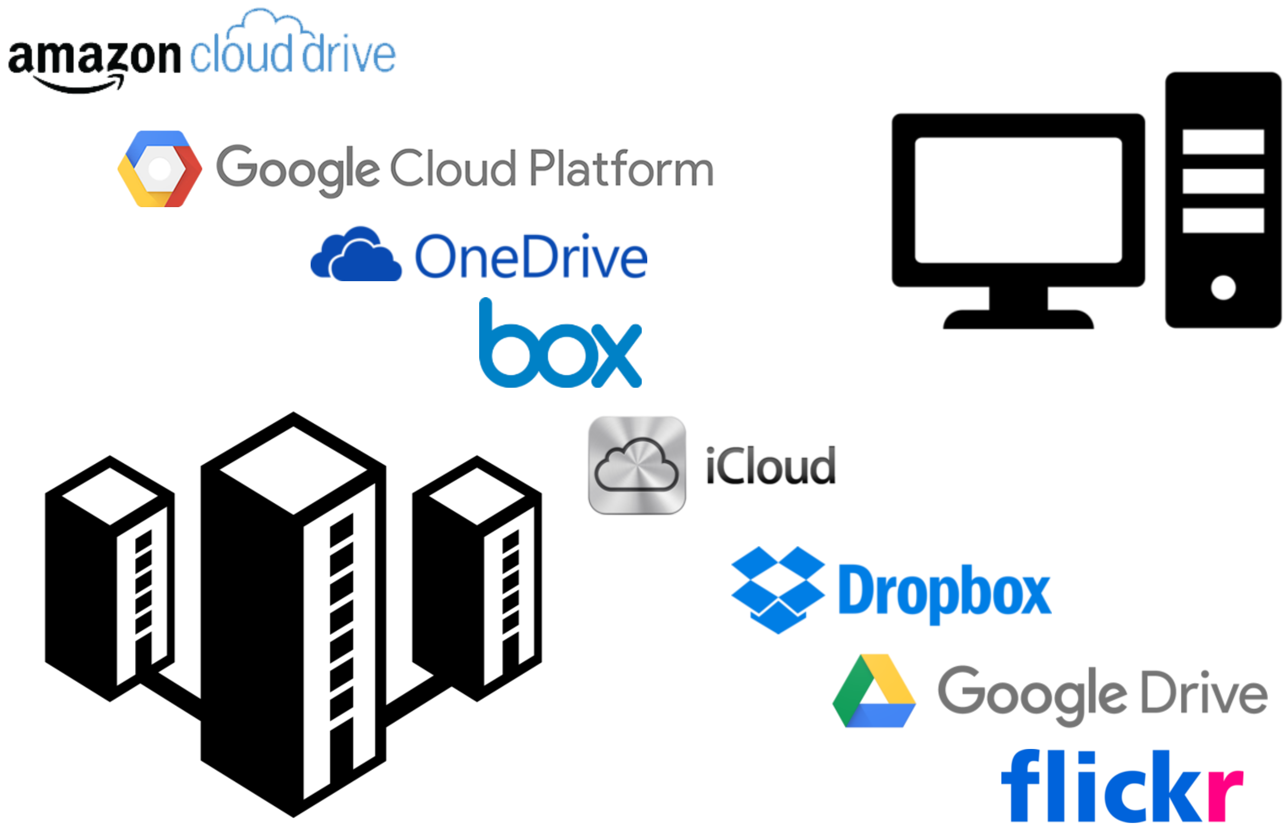
NeXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-side SGX

Adam J. Lee
Associate Dean for Academic Programs
Associate Professor of Computer Science
School of Computing and Information
University of Pittsburgh

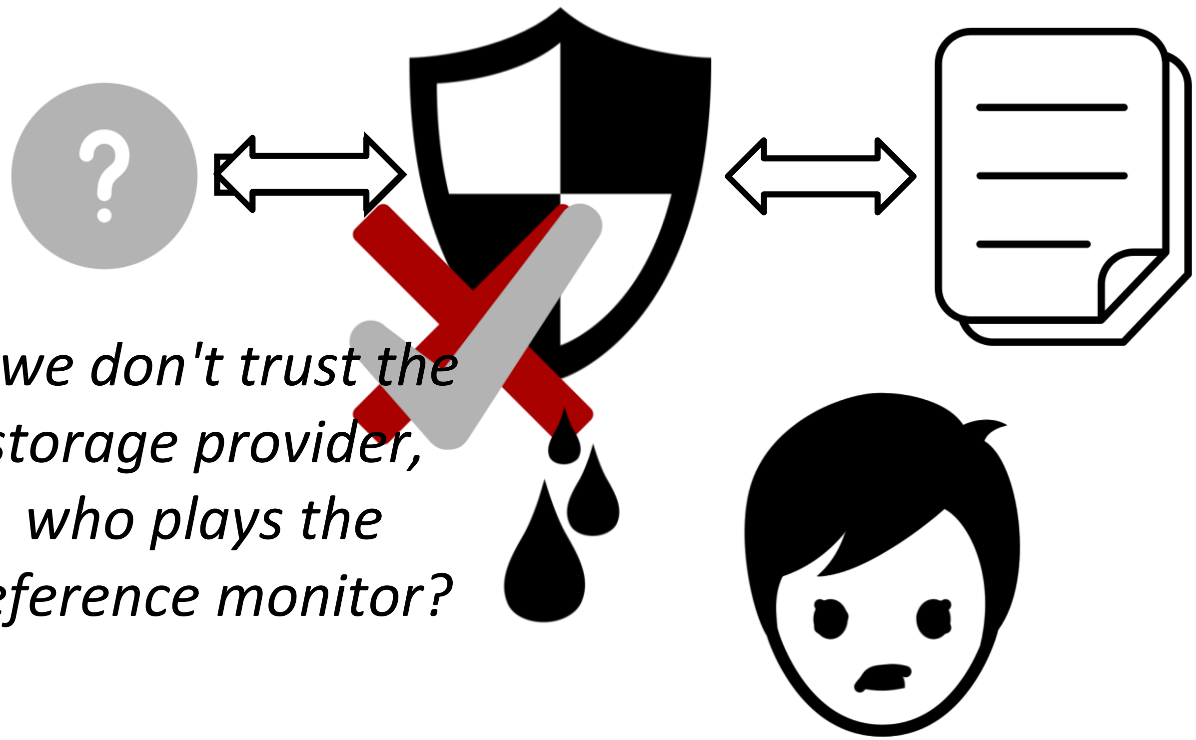
30 October 2018

NSF

Economic, management benefits have motivated a transition to cloud storage

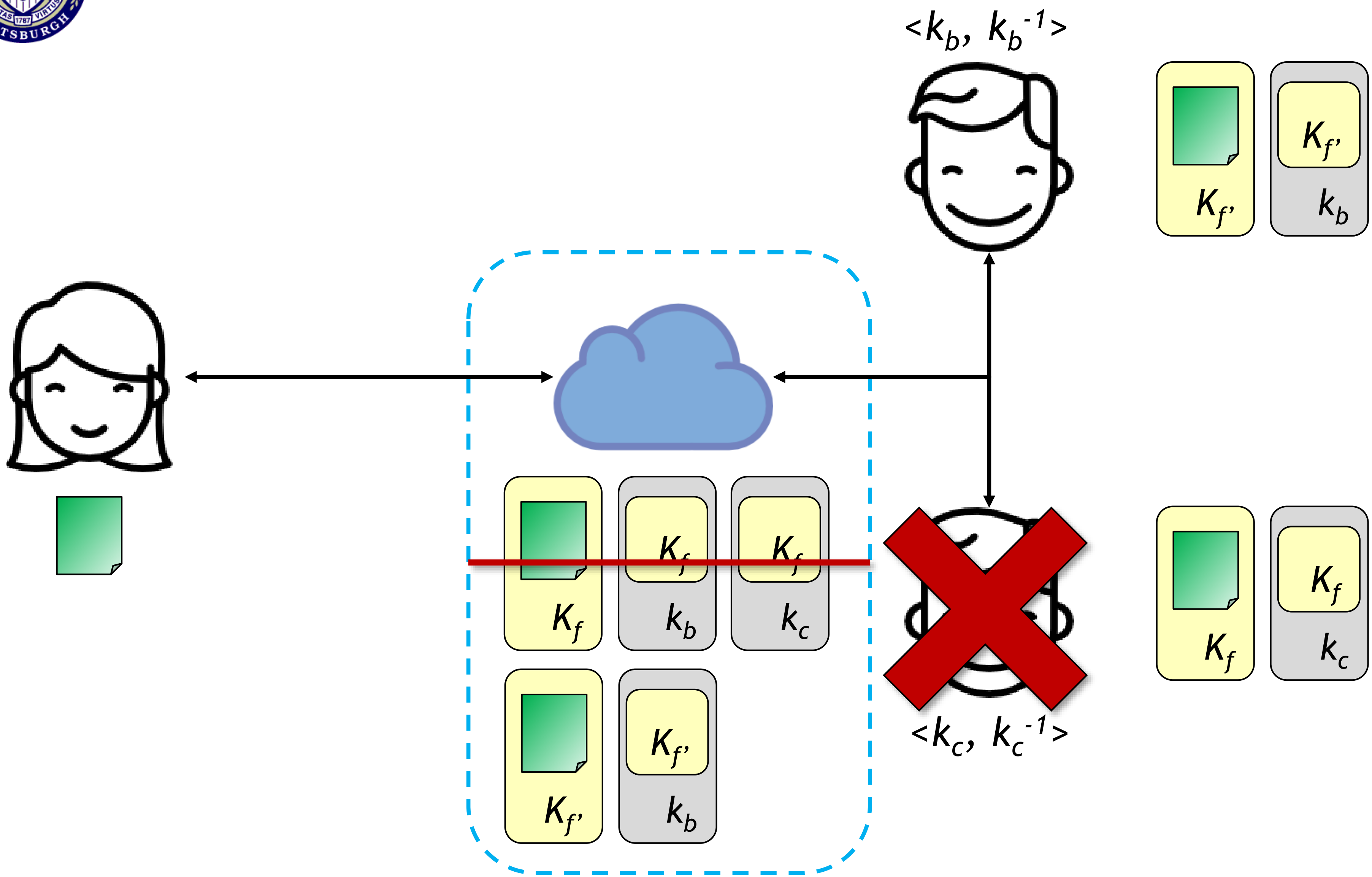


But what happens to access controls when we don't trust the provider?



If we don't trust the storage provider, who plays the reference monitor?

Cryptography can be used to enable secure file sharing



The complexities highlighted in the strawman construction are amplified in more realistic systems

On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud

William C. Garrison III, University of Pittsburgh
Adam Shull, Indiana University
Steven Myers, Indiana University
Adam J. Lee, University of Pittsburgh

***Abstract*—The ability to enforce robust and dynamic access controls on cloud-hosted data while simultaneously ensuring confidentiality with respect to the cloud itself is a clear goal for many users and organizations. To this end, there has been much cryptographic research proposing the use of (hierarchical) identity-based encryption, attribute-based encryption, predicate encryption, functional encryption, and related technologies to perform robust and private access control on untrusted cloud providers. However, the vast majority of this work studies static models in which the access control policies being enforced do not change over time. This is contrary to the needs of most practical applications, which leverage dynamic data and/or policies.**

In this paper, we show that the cryptographic enforcement of dynamic access controls on untrusted platforms incurs computational costs that are likely prohibitive in practice. Specifically, we develop lightweight constructions for enforcing role-based access controls (i.e., $RBAC_0$) over cloud-hosted files using identity-based and traditional public-key cryptography. This is done under a threat model as close as possible to the one assumed in the cryptographic literature. We prove the correctness of these constructions, and leverage real-world RBAC datasets and recent techniques developed by the access control community to experimentally analyze, via simulation, their associated computational costs. This analysis shows that supporting revocation, file updates, and other state change functionality is likely to incur prohibitive overheads in even minimally-dynamic, realistic scenarios. We identify a number of bottlenecks in such systems, and fruitful areas for future work that will lead to more natural and efficient constructions for the cryptographic enforcement of dynamic access controls. Our findings naturally extend to the use of more expressive cryptographic primitives (e.g., HIBE or ABE) and richer access control models (e.g., $RBAC_1$ or ABAC).

I. Introduction

In recent years, numerous cryptographic schemes have been developed to support access control on the (untrusted) cloud. One of the most expressive of these is attribute-based encryption (ABE) [31], which is a natural fit for enforcing attribute-based access control (ABAC) policies [40]. However, the practical implications of using these types of cryptographic schemes to tackle realistic access control problems are largely unexplored. In particular, much of the literature concerns static scenarios in which data and/or access control policies are rarely, if ever, modified (e.g., [5], [30], [31], [42], [49], [52], [59]). Such scenarios are not representative of real-world systems, and oversimplify issues associated with key management and revocation that can carry substantial practical overheads. In this paper, we explore *exactly* these types of issues in an attempt to understand the computational overheads of using advanced cryptographic techniques to enforce dynamic access controls over objects stored on untrusted platforms. Our primary result is negative: we demonstrate that prohibitive computational burdens are likely to be incurred when supporting practical, dynamic workloads.

The push to develop and use cryptography to support adaptive access control on the cloud is natural. Major cloud providers such as Google, Microsoft, Apple, and Amazon are providing both large-scale, industrial services and smaller-scale, consumer services. Similarly, there are a number of user-focused cloud-based file sharing services, such as Dropbox, Box, and Flickr. However, the near-constant media coverage of data breaches has raised both consumer and enterprise concerns regarding the privacy and integrity of cloud-stored data. Among the widely-publicized stories of external hacking and data disclosure are releases of private photos [56]. Some are even state-sponsored attacks against cloud organizations themselves, such as Operation Aurora, in which Chinese hackers infiltrated providers like Google, Yahoo, and Rackspace [20], [51]. Despite the economic benefits and ease-of-use provided by outsourcing data management to the cloud, this practice raises new questions regarding the maintenance and enforcement of the access controls that users have come to expect from file sharing systems.

Although advanced cryptographic primitives seem well-suited for protecting *point states* in many access control paradigms, supporting the *transitions* between protection states that are triggered by administrative actions in a dynamic system requires addressing very subtle issues involving key management, coordination, and key/policy consistency. While there has been some work seeking to provide a level of dynamism for these types of advanced cryptographic primitives, this work is not without issues. For instance, techniques have been developed to support key revocation [8] and delegated re-encryption [32], [58]. Unfortunately, these techniques are not compatible with hybrid encryption—which is necessary from an efficiency perspective—under reasonable threat models.

In this paper, we attempt to tease out these types of critical details by exploring the cryptographic enforcement of a widely-deployed access control model: role-based access control (specifically, $RBAC_0$ [61]). In particular, we develop two constructions for cryptographically enforcing dynamic $RBAC_0$ policies in untrusted cloud environments: one based on standard public-key cryptographic techniques, and another based on identity-based encryption/signature (IBE/IBS) techniques [11], [13], [59]. By studying $RBAC_0$ in the context of these relatively

IEEE S&P 2016

Using a hybrid IBE/IBS construction to enforce RBAC policies is non-trivial…

$addU(u)$
- Add u to USERS
- Generate IBE private key $k_u \leftarrow \mathbf{KeyGen}^{\mathbf{IBE}}(u)$ and IBS private key $s_u \leftarrow \mathbf{KeyGen}^{\mathbf{IBS}}(u)$ for the new user u
- Give k_u and s_u to u over private and authenticated channel

$delU(u)$
- For every role r that u is a member of:
  - $revokeU(u, r)$

$addP_u(fn, f)$
- Generate symmetric key $k \leftarrow \mathbf{Gen}^{\mathbf{Sym}}$
- Send $\langle \mathsf{F}, fn, 1, \mathbf{Enc}_k^{\mathbf{Sym}}(f), u, \mathbf{Sign}_u^{\mathbf{IBS}} \rangle$ and $\langle \mathsf{FK}, SU, \langle fn, \mathsf{RW} \rangle, 1, \mathbf{Enc}_{SU}^{\mathbf{IBE}}(k), u, \mathbf{Sign}_u^{\mathbf{IBS}} \rangle$ to R.M.
- The R.M. receives $\langle \mathsf{F}, fn, 1, c, u, sig \rangle$ and $\langle \mathsf{FK}, SU, \langle fn, \mathsf{RW} \rangle, 1, c', u, sig' \rangle$ and verifies that the tuples are well-formed and the signatures are valid, i.e., $\mathbf{Ver}_u^{\mathbf{IBS}}(\langle \mathsf{F}, fn, 1, c, u \rangle, sig) = 1$ and $\mathbf{Ver}_u^{\mathbf{IBS}}(\langle \mathsf{FK}, SU, \langle fn, \mathsf{RW} \rangle, 1, c', u \rangle, sig') = 1$.
- If verification is successful, the R.M. adds $(fn, 1)$ to FILES and stores $\langle \mathsf{F}, fn, 1, c, u, sig \rangle$ and $\langle \mathsf{FK}, SU, \langle fn, \mathsf{RW} \rangle, 1, c', u, sig' \rangle$

$delP(fn)$
- Remove (fn, v_{fn}) from FILES
- Delete $\langle \mathsf{F}, fn, -, -, -, - \rangle$ and all $\langle \mathsf{FK}, -, \langle fn, - \rangle, -, -, -, - \rangle$

$addR(r)$
- Add $(r, 1)$ to ROLES
- Generate IBE private key $k_{(r,1)} \leftarrow \mathbf{KeyGen}^{\mathbf{IBE}}((r,1))$ and IBS private key $s_{(r,1)} \leftarrow \mathbf{KeyGen}^{\mathbf{IBS}}((r,1))$ for role $(r,1)$
- Send $\langle \mathsf{RK}, SU, (r,1), \mathbf{Enc}_{SU}^{\mathbf{IBE}}\left(k_{(r,1)}, s_{(r,1)}\right), \mathbf{Sign}_{SU}^{\mathbf{IBS}} \rangle$ to R.M.

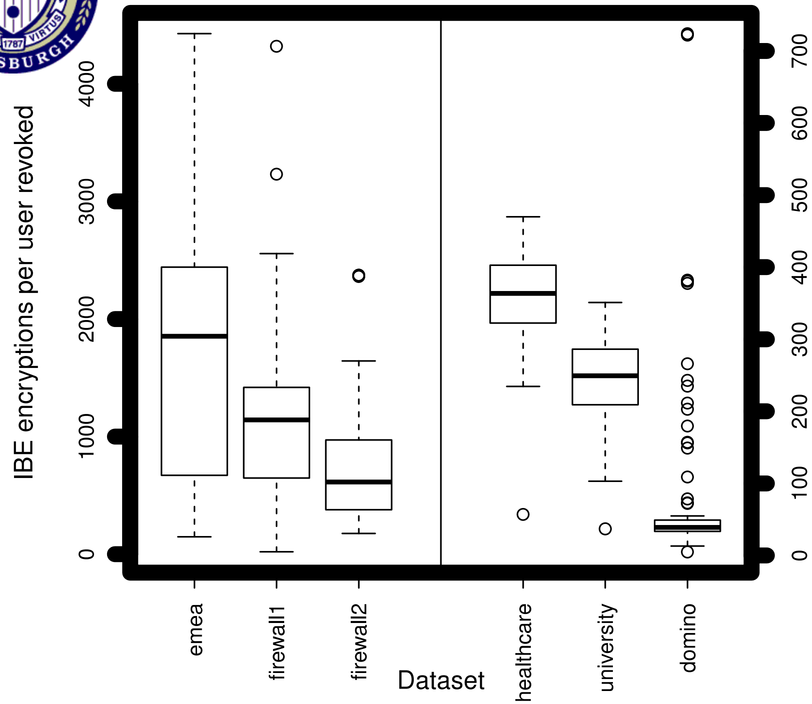
$delR(r)$
- Remove (r, v_r) from ROLES
- Delete all $\langle \mathsf{RK}, -, (r, v_r), -, - \rangle$
- For all permissions $p = \langle fn, op \rangle$ that r has access to:

$assignP(r, \langle fn, op \rangle)$
- For all $\langle \mathsf{FK}, SU, \langle fn, \mathsf{RW} \rangle, v, c, id, sig \rangle$ with $\mathbf{Ver}_{id}^{\mathbf{IBS}}(\langle \mathsf{FK}, SU, \langle fn, \mathsf{RW} \rangle, v, c, id \rangle, sig) = 1$:
  - If this adds Write permission to existing Read permission, i.e., $op = \mathsf{RW}$ and there exists $\langle \mathsf{FK}, (r, v_r), \langle fn, \mathsf{Read} \rangle, v, c', SU, sig \rangle$ with $\mathbf{Ver}_{SU}^{\mathbf{IBS}}(\langle \mathsf{FK}, (r, v_r), \langle fn, op' \rangle, v, c', SU \rangle, sig) = 1$:
    - Send $\langle \mathsf{FK}, (r, v_r), \langle fn, \mathsf{RW} \rangle, v, c', SU, \mathbf{Sign}_{SU}^{\mathbf{IBS}} \rangle$ to R.M.
    - Delete $\langle \mathsf{FK}, (r, v_r), \langle fn, \mathsf{Read} \rangle, v, c', SU, sig \rangle$
  - If the role has no existing permission for the file, i.e., there does not exist $\langle \mathsf{FK}, (r, v_r), \langle fn, op' \rangle, v, c', SU, sig \rangle$ with $\mathbf{Ver}_{SU}^{\mathbf{IBS}}(\langle \mathsf{FK}, (r, v_r), \langle fn, op' \rangle, v, c, SU \rangle, sig) = 1$:
    - Decrypt key $k = \mathbf{Dec}_{k_{SU}}^{\mathbf{IBE}}(c)$
    - Send $\langle \mathsf{FK}, (r, v_r), \langle fn, op \rangle, v, \mathbf{Enc}_{(r,v_r)}^{\mathbf{IBE}}(k), SU, \mathbf{Sign}_{SU}^{\mathbf{IBS}} \rangle$ to R.M.

$revokeP(r, \langle fn, op \rangle)$
- If $op = \mathsf{Write}$:
  - For all $\langle \mathsf{FK}, (r, v_r), \langle fn, \mathsf{RW} \rangle, v, c, SU, sig \rangle$ with $\mathbf{Ver}_{SU}^{\mathbf{IBS}}(\langle \mathsf{FK}, (r, v_r), \langle fn, \mathsf{RW} \rangle, v, c, SU \rangle, sig) = 1$:
    - Send $\langle \mathsf{FK}, (r, v_r), \langle fn, \mathsf{Read} \rangle, v, c, SU, \mathbf{Sign}_{SU}^{\mathbf{IBS}} \rangle$ to R.M.
    - Delete $\langle \mathsf{FK}, (r, v_r), \langle fn, \mathsf{RW} \rangle, v, c, SU, sig \rangle$
- If $op = \mathsf{RW}$:
  - Delete all $\langle \mathsf{FK}, (r, v_r), \langle fn, - \rangle, -, -, - \rangle$
  - Generate new symmetric key $k' \leftarrow \mathbf{Gen}^{\mathbf{Sym}}$
  - For all $\langle \mathsf{FK}, r', \langle fn, op' \rangle, v_{fn}, c, SU, sig \rangle$ with $\mathbf{Ver}_{SU}^{\mathbf{IBS}}(\langle \mathsf{FK}, r', \langle fn, op' \rangle, v, c, SU \rangle, sig) = 1$:
    - Send $\langle \mathsf{FK}, r', \langle fn, op' \rangle, v_{fn} + 1, \mathbf{Enc}_{id}^{\mathbf{IBE}}(k'), SU, \mathbf{Sign}_{SU}^{\mathbf{IBS}} \rangle$ to R.M.
  - Increment v_{fn} in FILES, i.e., set $v_{fn} := v_{fn} + 1$

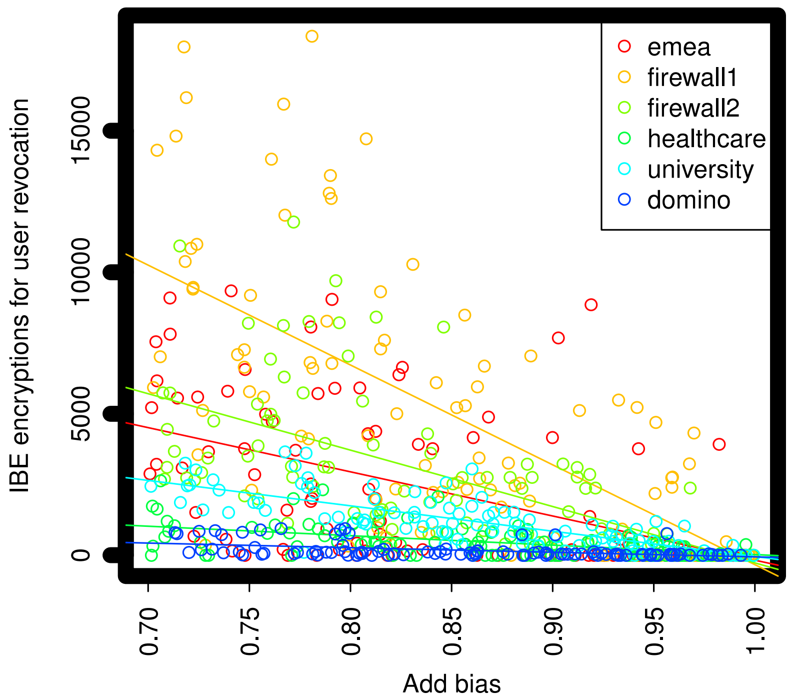
$read_u(fn)$
- Find $\langle \mathsf{F}, fn, v, c, id, sig \rangle$ with valid ciphertext c and valid signature sig, i.e., $\mathbf{Ver}_{id}^{\mathbf{IBS}}(\langle \mathsf{F}, fn, 1, c, id \rangle, sig) = 1$
- Find a role r such that the following hold:
  - u is in role r, i.e., there exists $\langle \mathsf{RK}, u, (r, v_r), c', sig \rangle$ with

Revocations incur enormous costs, even in settings that are only mildly dynamic



Tens to thousands of IBE encryptions to revoke a user from a role

Even when only 10% of admin operations are revocations, much system time is spent managing key distributions



What are we to do?

Sources of revocation overheads

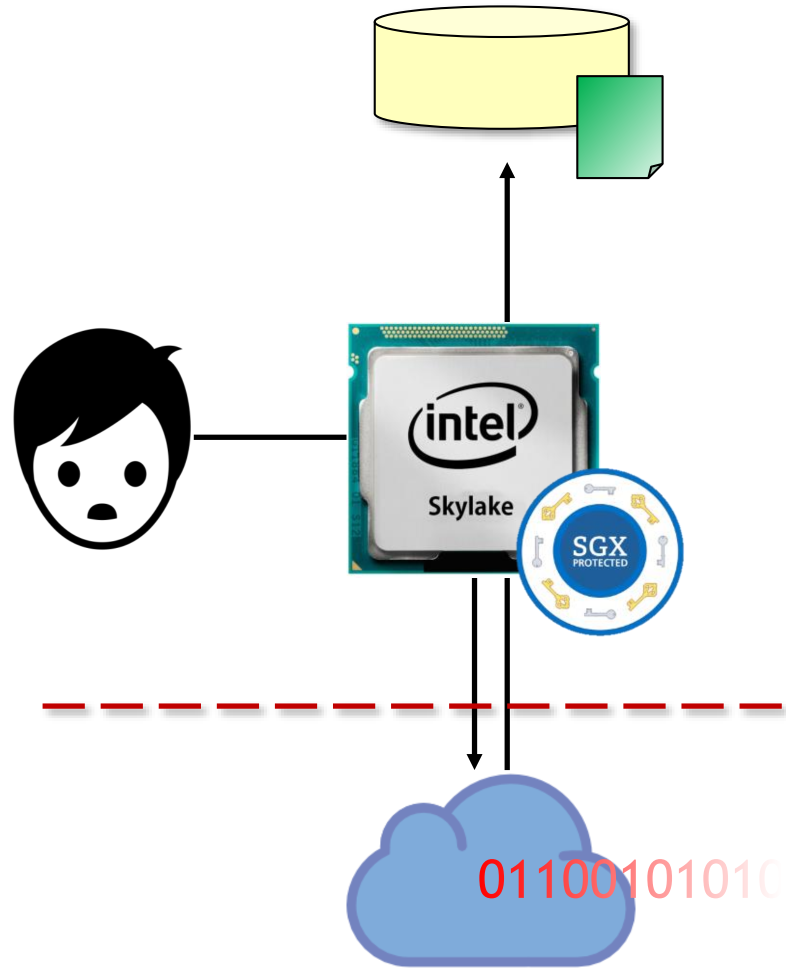
- Download, decrypt, re-encrypt, and upload of impacted file(s)
- Redistribution of new keys

Observation: All of this happens because *access* to the file implies *observation of the key* used to encrypt it

---

What if we could broker access to files *without* revealing keys?

Our recent work seeks to improve this state of affairs by combining cryptography and trusted hardware



SGX is a set of ISA extensions in recent Intel processors that enables secure execution environments

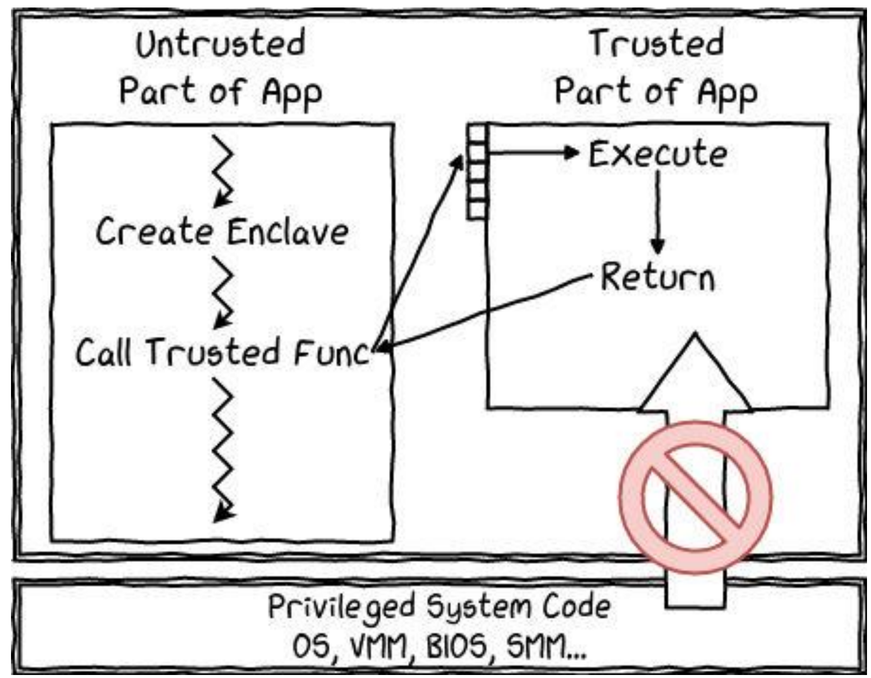
A key feature enabled by SGX is isolated execution

An enclave encodes the trusted portion of an untrusted application

- Hardware protected confidentiality and integrity for code and data
- Enclave are permitted to access application memory
- Applications cannot access enclave memory

Enclaves are even protected from a malicious OS/Hypervisor

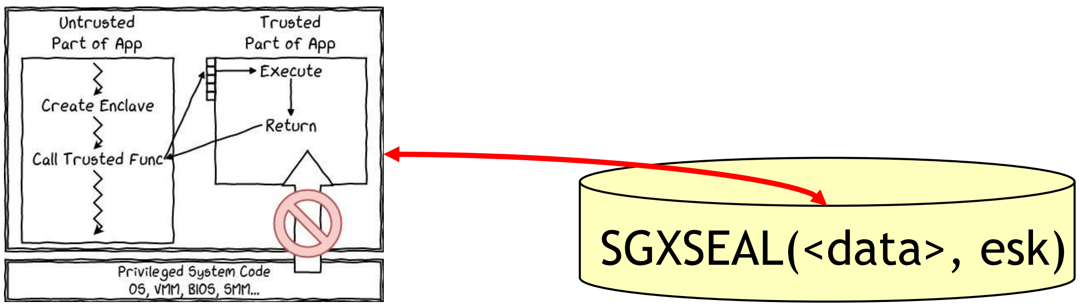
Caveat: Isolated execution alone is not terribly useful



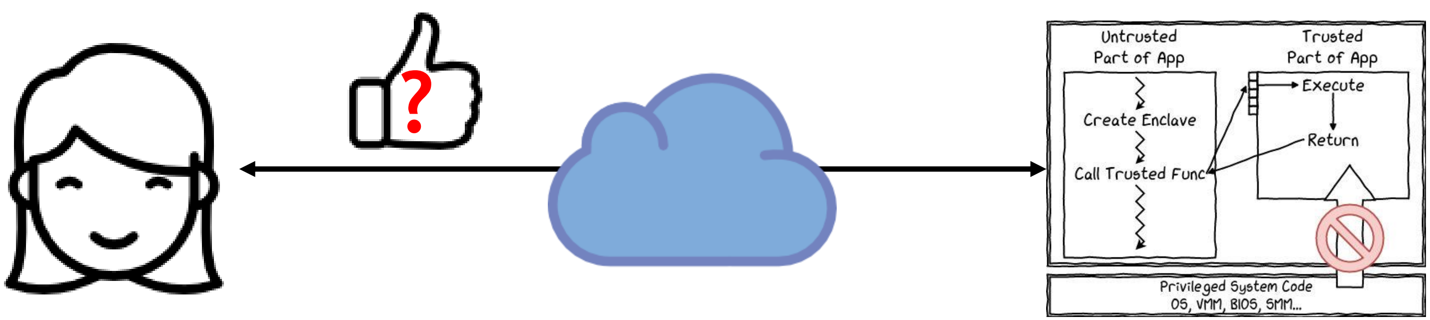
https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html

Two other features extend the utility of SGX protections to a wide class of applications

Sealed storage allows for the long-term storage of enclave-resident information



Local and remote attestation allow processes to ensure the authenticity of the enclaves that they rely on



NeXUS leverages SGX to enforce users' access controls on untrusted storage platforms

Cloud storage providers already allow rich access controls

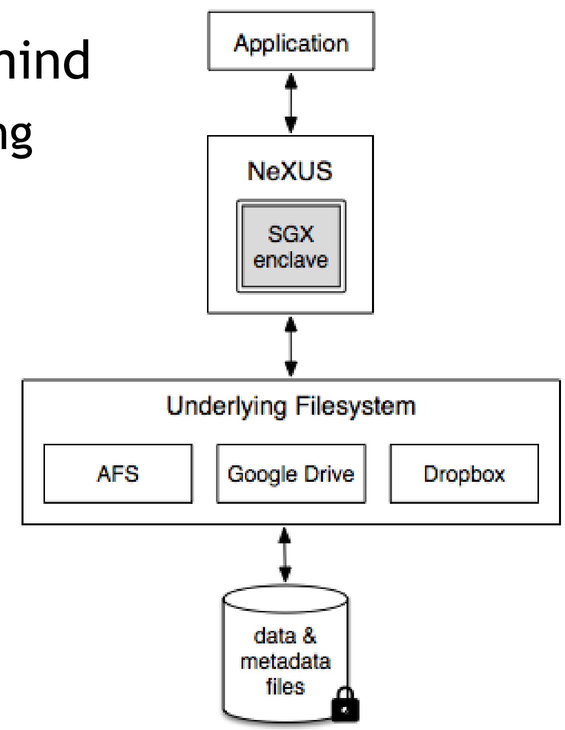
Our goal is to enforce these types of access controls, even when the storage platform is untrusted or compromised

NeXUS was built with two key design goals in mind

- Portability: Seamless integration with existing storage providers and services
- Practicality: The use of NeXUS should not negatively impact common user workflows

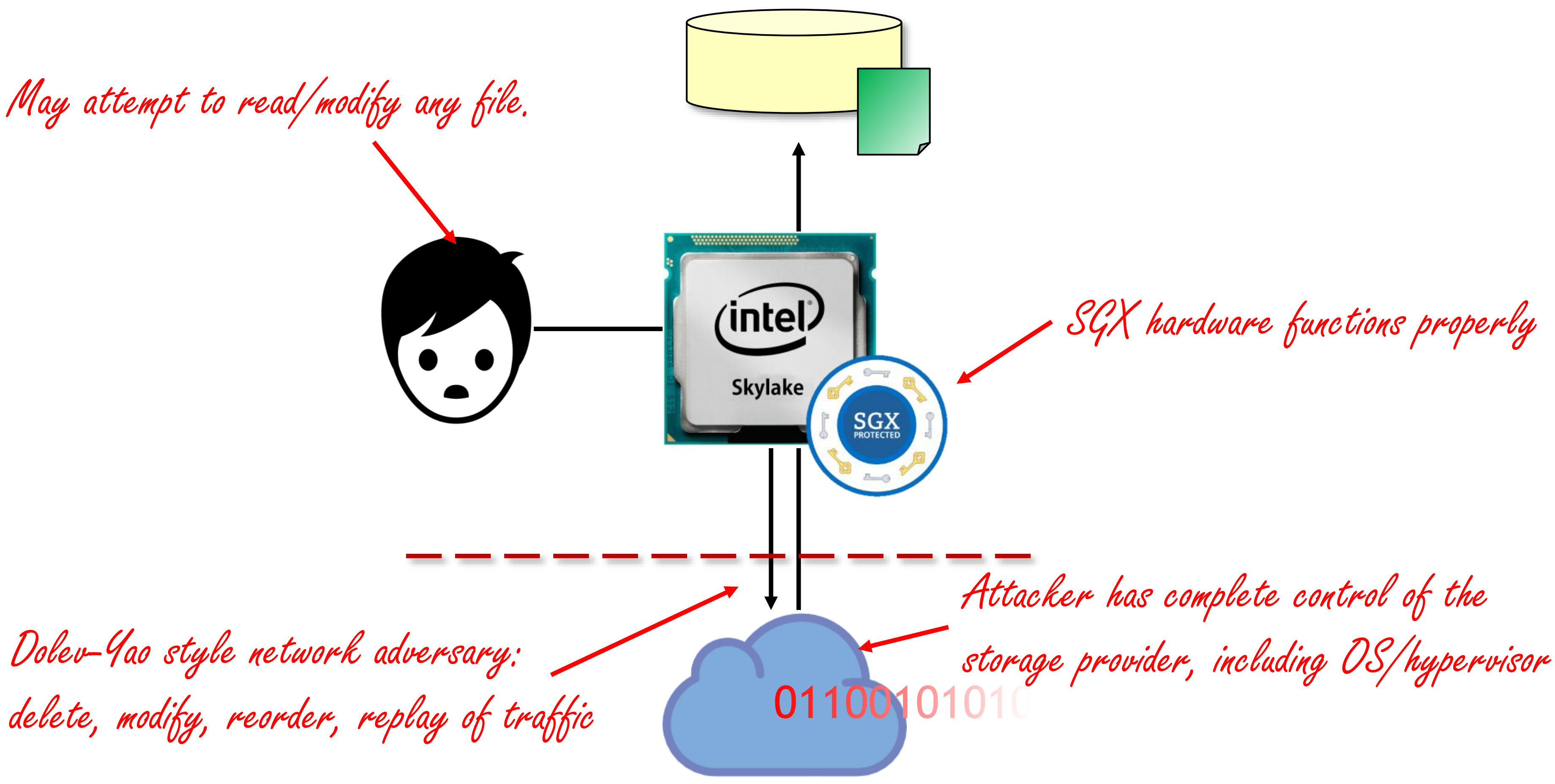
Deployment without server-side support

Minimal changes to UX

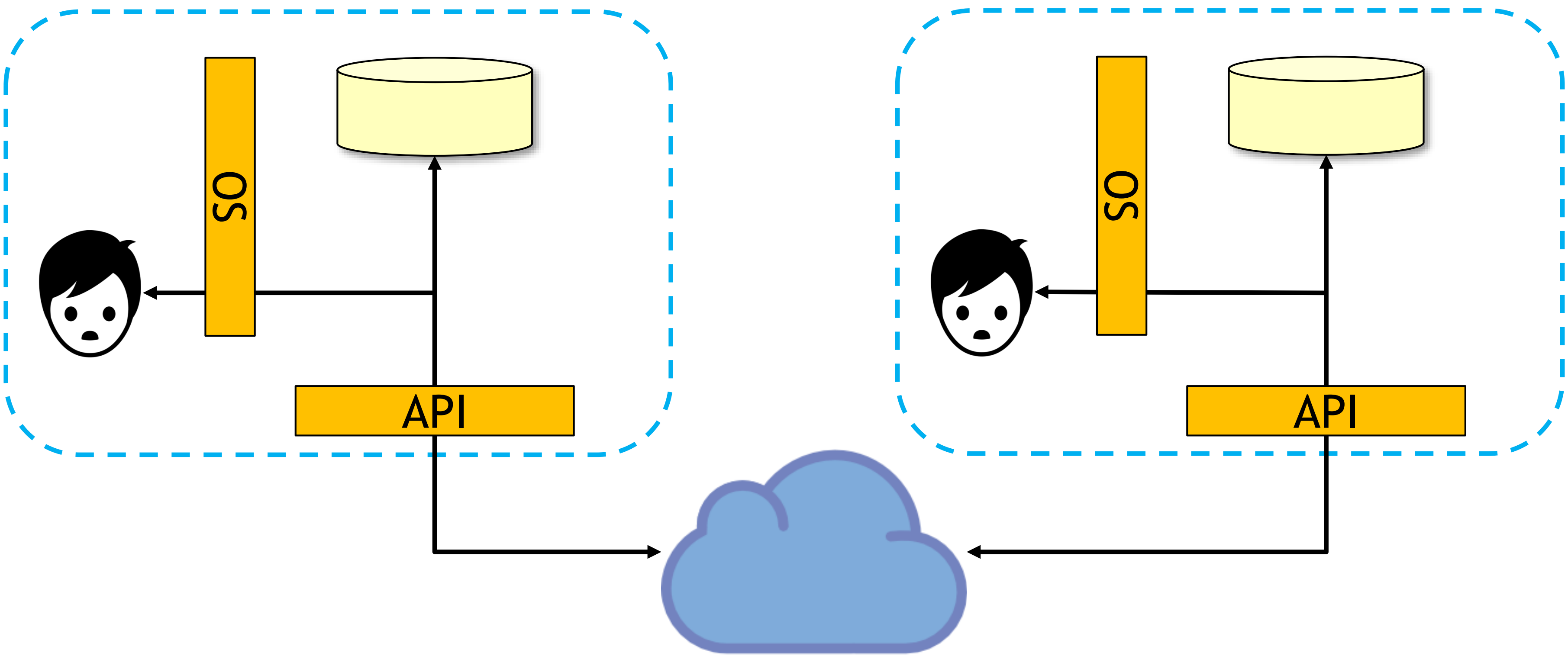


Threat Model

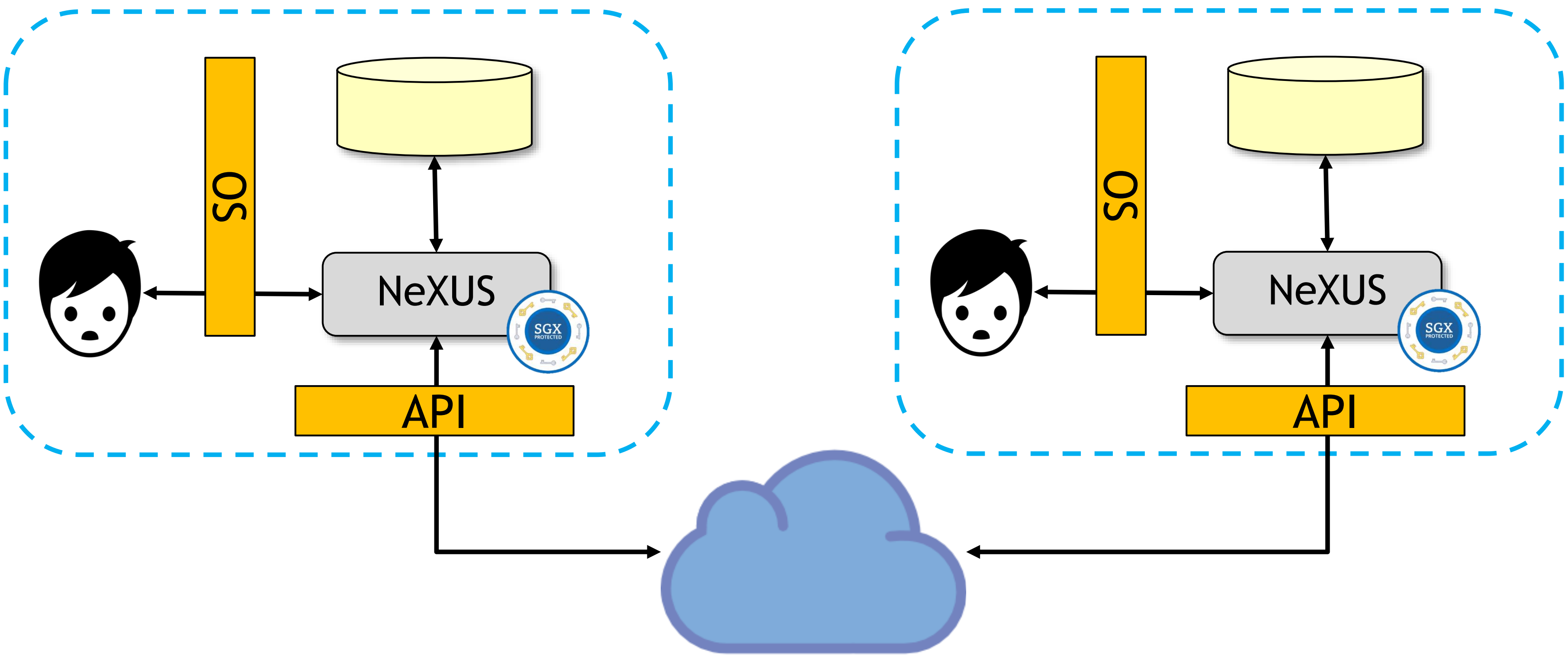
Security Objective: Unless granted explicit access by the owner, the contents of files and directories (i.e., file names) must remain *confidential* and *tamper-evident*.



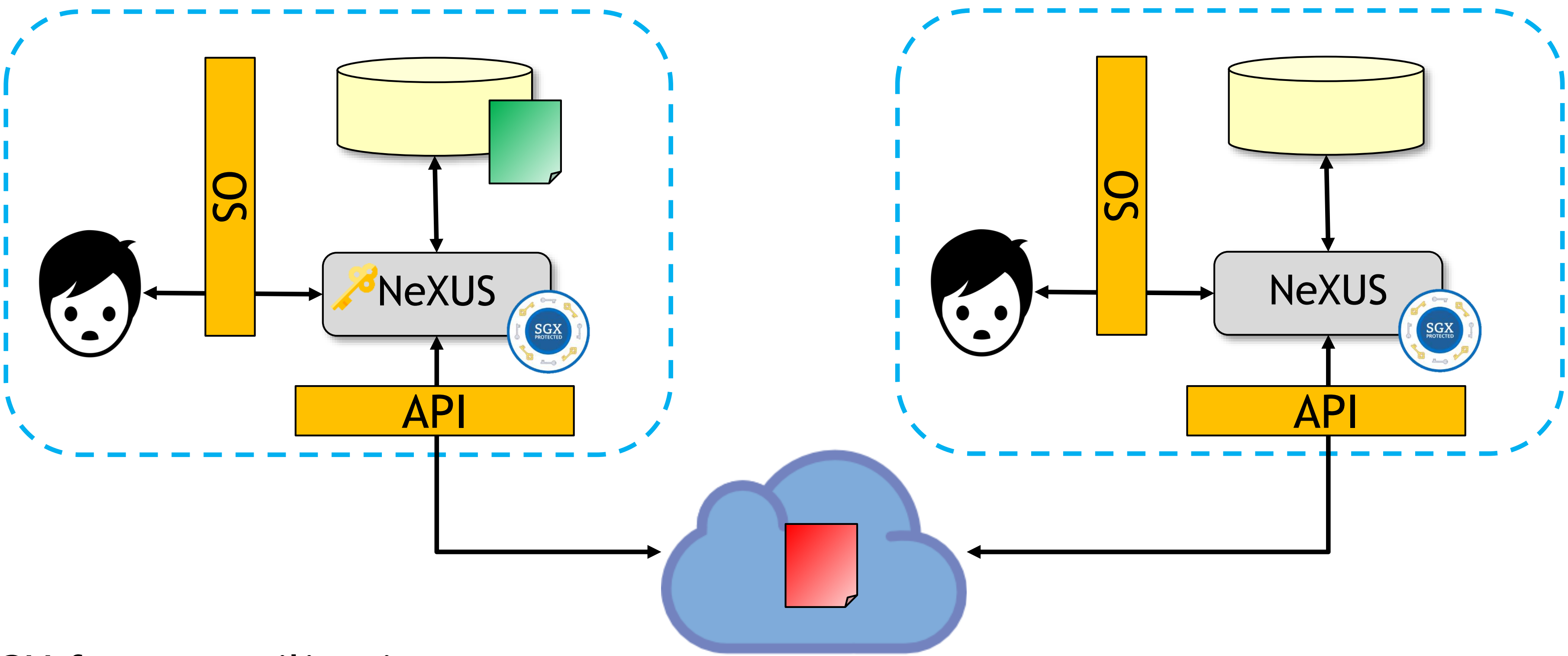
NeXUS combines the cryptographic techniques used in our straw-man solution with SGX security guarantees



NeXUS combines the cryptographic techniques used in our straw-man solution with SGX security guarantees



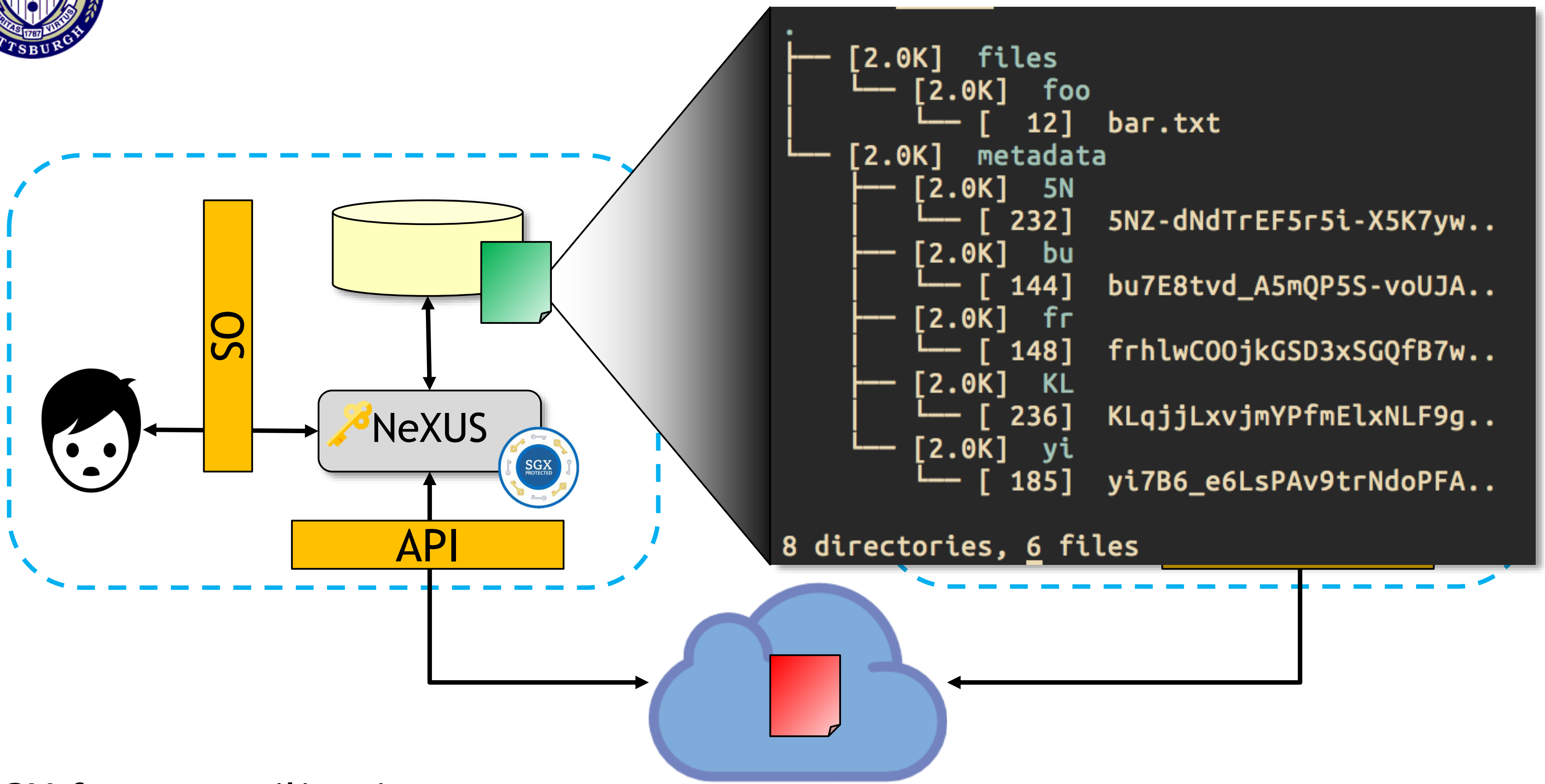
NeXUS combines the cryptographic techniques used in our straw-man solution with SGX security guarantees



SGX feature utilization

- Encryption takes place in enclave to protect keys (isolated execution)
- Enclave state protected on local disk via enclave-derived keys (sealed storage)

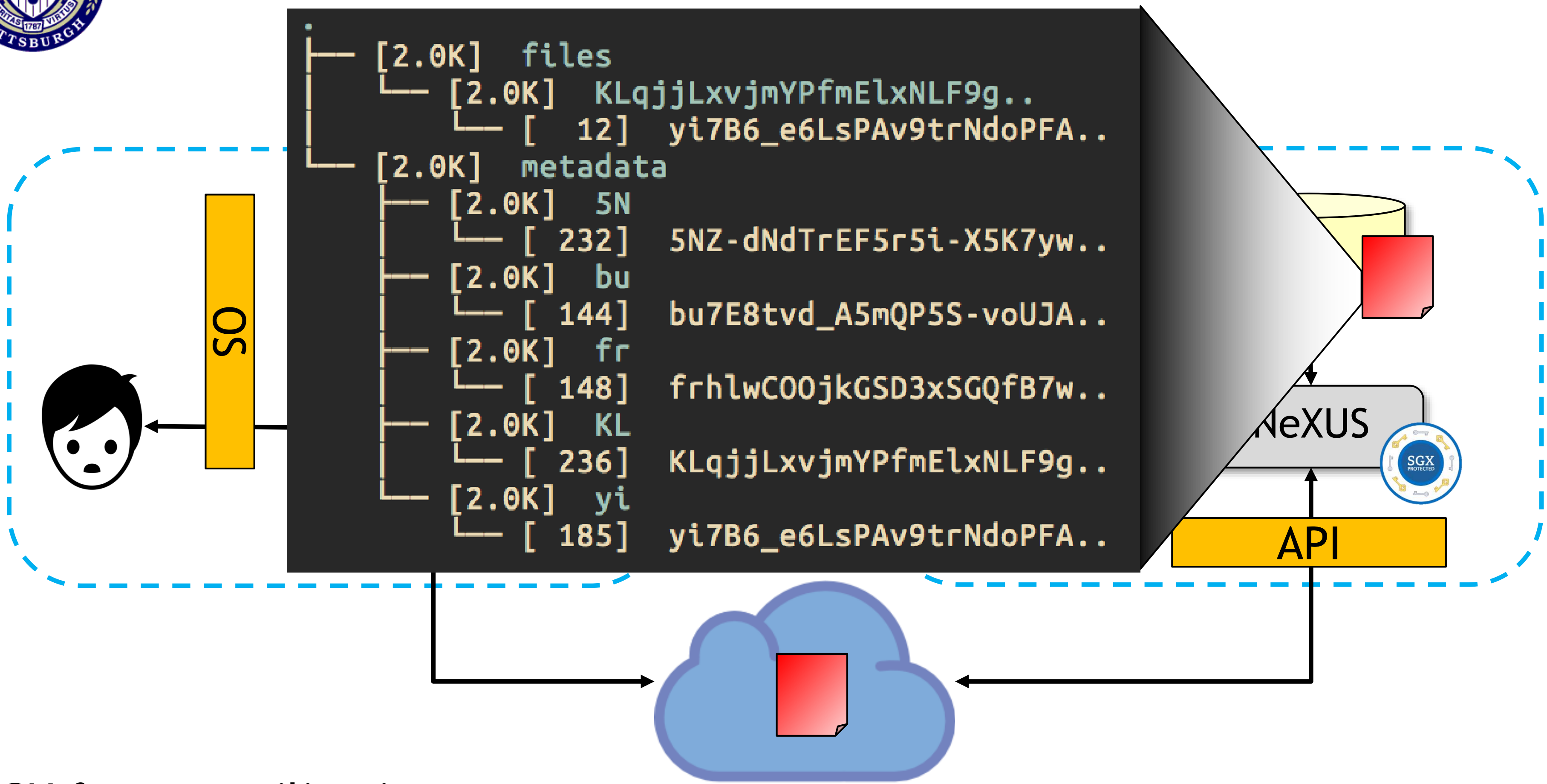
NeXUS combines the cryptographic techniques used in our straw-man solution with SGX security guarantees



SGX feature utilization

- Encryption takes place in enclave to protect keys (isolated execution)
- Enclave state protected on local disk via enclave-derived keys (sealed storage)

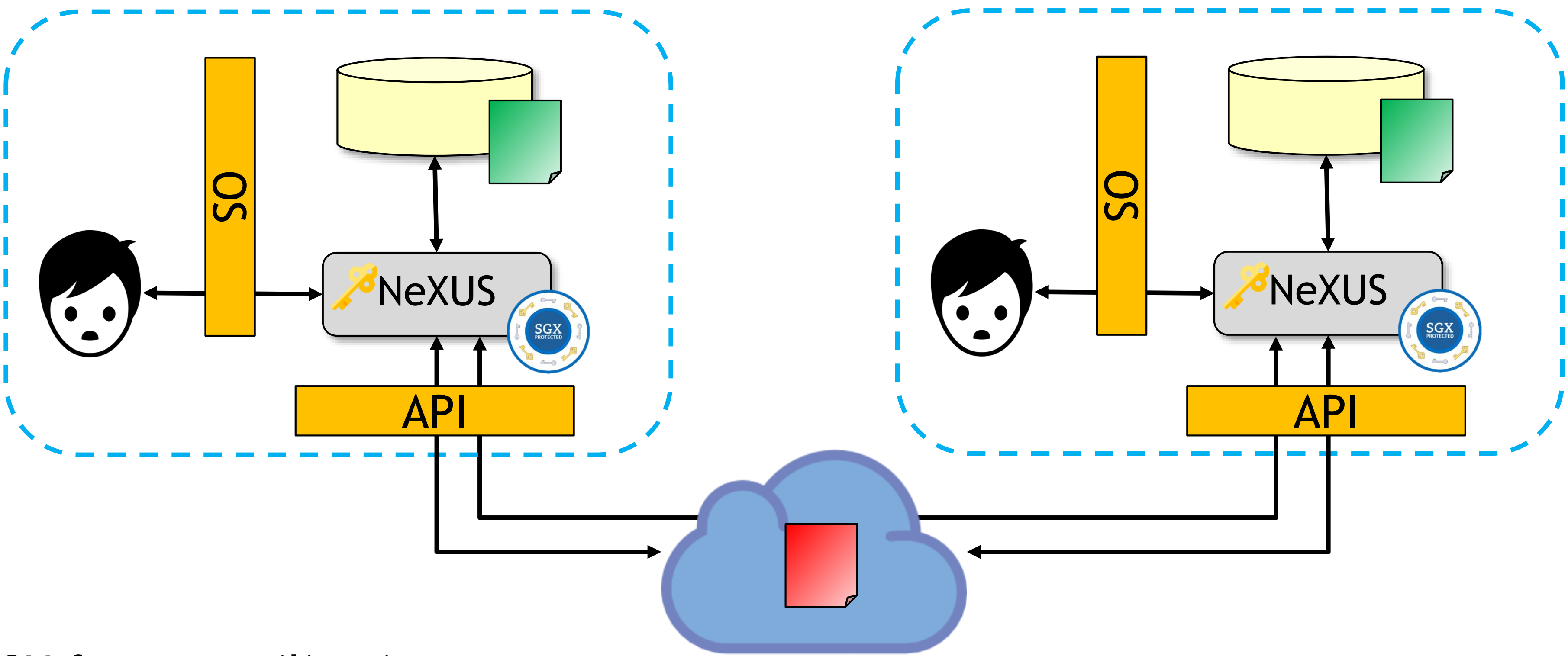
NeXUS combines the cryptographic techniques used in our straw-man solution with SGX security guarantees



SGX feature utilization

- Encryption takes place in enclave to protect keys (isolated execution)
- Enclave state protected on local disk via enclave-derived keys (sealed storage)

NeXUS combines the cryptographic techniques used in our straw-man solution with SGX security guarantees



SGX feature utilization

- Encryption takes place in enclave to protect keys (isolated execution)
- Enclave state protected on local disk via enclave-derived keys (sealed storage)
- Authorization and key exchange across machines (remote attestation)

Why this design?

This design facilitates easy deployment for user-centric workloads

- No server-side modifications necessary
- No global namespace needed for file sharing
- Minimal administrative changes to existing file management

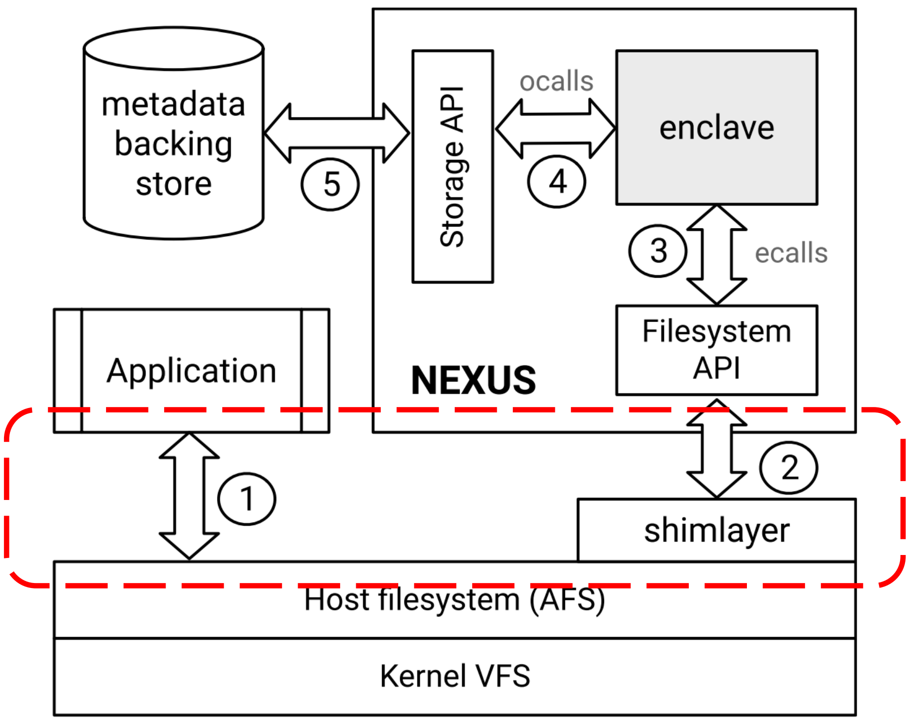
Getting this right involves a lot of moving parts

- Maintaining the metadata to support a filesystem *within* a filesystem
- Synchronization/consistency issues due to distributed enforcement
- Optimized communication between applications, kernel, and enclave
- Remote attestation with potentially offline partners
- …

I'll focus on the structure/management of a NeXUS volume and a brief performance evaluation of our prototype

NeXUS: A stackable virtual filesystem

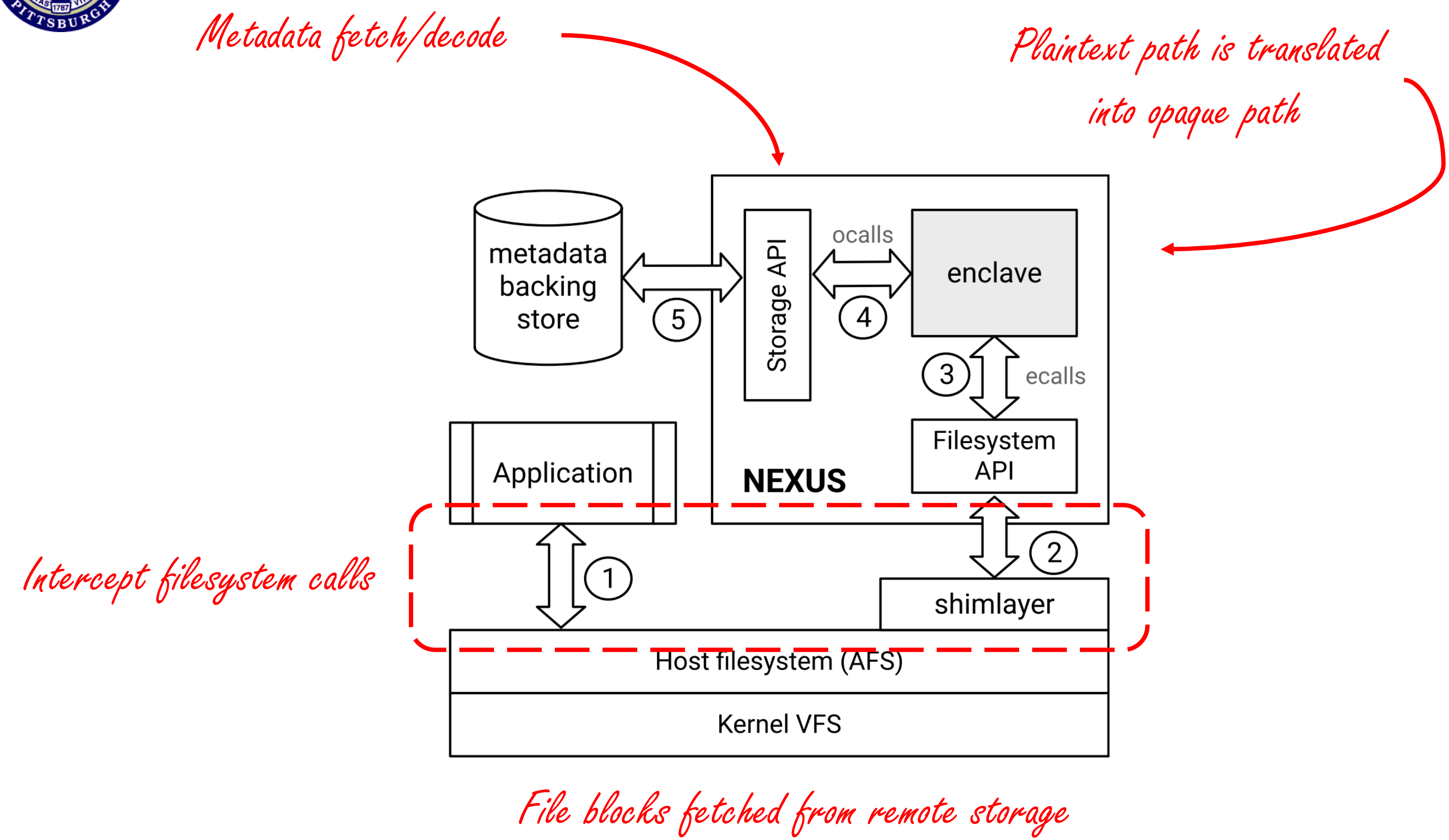
Intercept filesystem calls



Filesystem Call	Description
Directory Operations	
`nexus_fs_touch()`	Creates a new file/directory
`nexus_fs_remove()`	Deletes file/directory
`nexus_fs_lookup()`	Finds a file by name
`nexus_fs_filldir()`	Lists directory contents
`nexus_fs_symlink()`	Creates a symlink to a target path
`nexus_fs_hardlink()`	Hardlinks two files
`nexus_fs_rename()`	Moves a files between directories
File Operations	
`nexus_fs_encrypt()`	Encrypts a file contents
`nexus_fs_decrypt()`	Decrypts a file contents

Table 1: NEXUS Filesystem API.

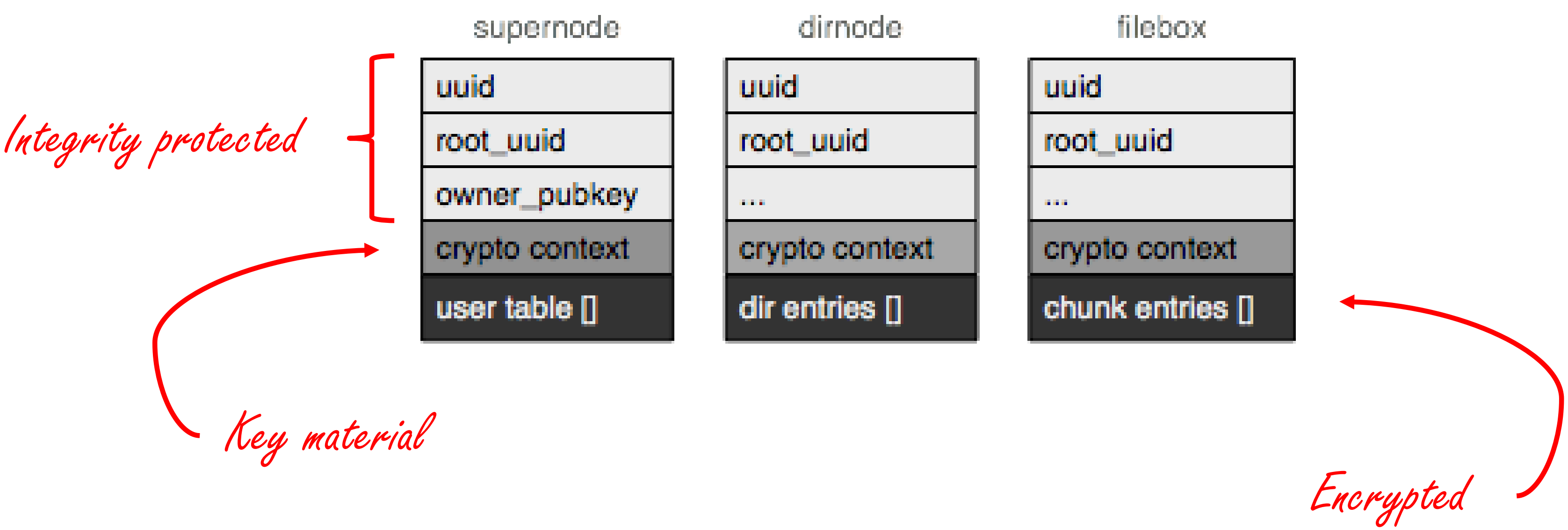
NeXUS: A stackable virtual filesystem



NeXUS stores sensitive filesystem data using metadata that reflects standard filesystem structures

Key data structures:

- Supernode: Stores filesystem info, including usertable
- Dirnode: Stores directory entries; maps filenames to UUIDs
- Filenode: Stores file chunk encryption keys



How is metadata recovered?

NeXUS Enclave i

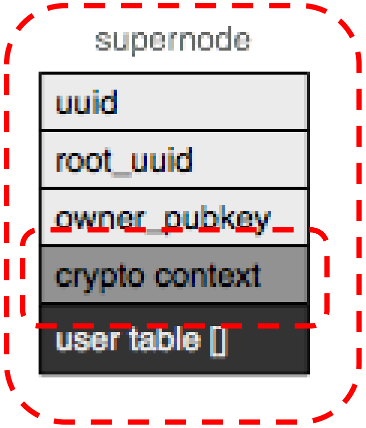
- Enclave sealing key: esk_i
- Volume rootkey: rk

Disk

- SGXSEAL(rk, esk_i)

Example: Mounting a NeXUS volume

- Load sealed rootkey (rk) from local disk
- Use the local enclave sealing key (esk_i) to decrypt
  - Note: Neither esk_i or rk ever leave the enclave!
- Use rk to decrypt the cryptographic context
  - Context = ENC(mek, rk)
  - mek = random metadata encryption key
- Use mek to decrypt and validate supernode



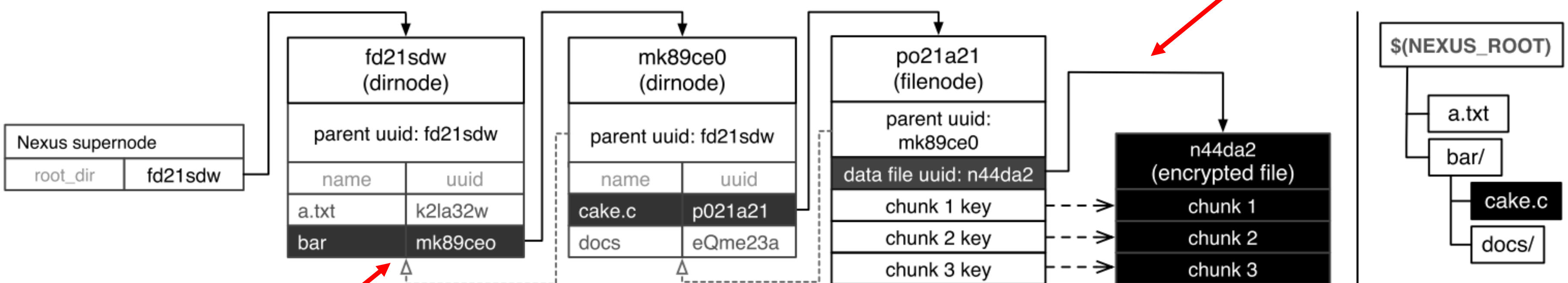
This process works similarly for all other metadata structures

File access example: `$/bar/cake.c`

Recover root dirnode

Locate filenode pointing to the contents of cake.c

Download encrypted contents of n44da2 (i.e., cake.c)



Find dirnode UUID corresponding to 'bar'

Separate keys for chunks within a file. WHY?

I’ve glossed over some important details…

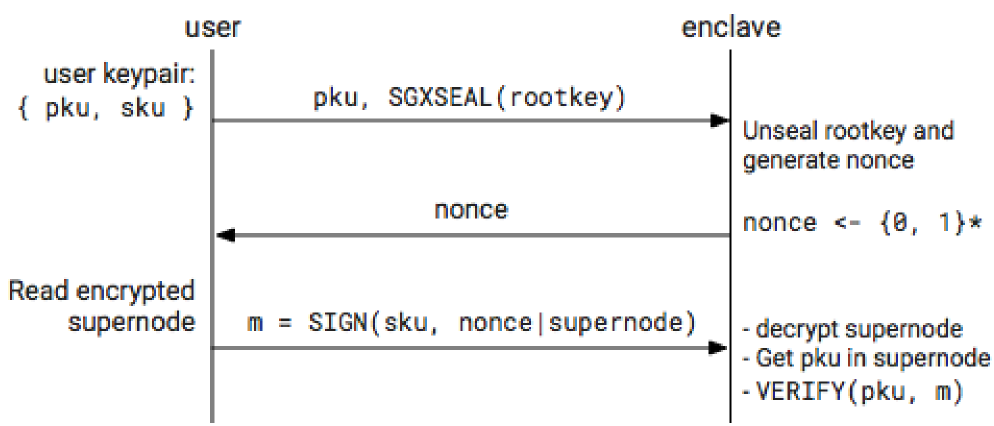
How do we figure out who is accessing a volume?

To mount a NeXUS volume, the user must provide

- The volume's (encrypted) supernode
- A sealed rootkey for the volume
- Their public key

Bound to their CPU

The NeXUS enclave carries out a challenge/response to authenticate the user via proof-of-possession of their private key

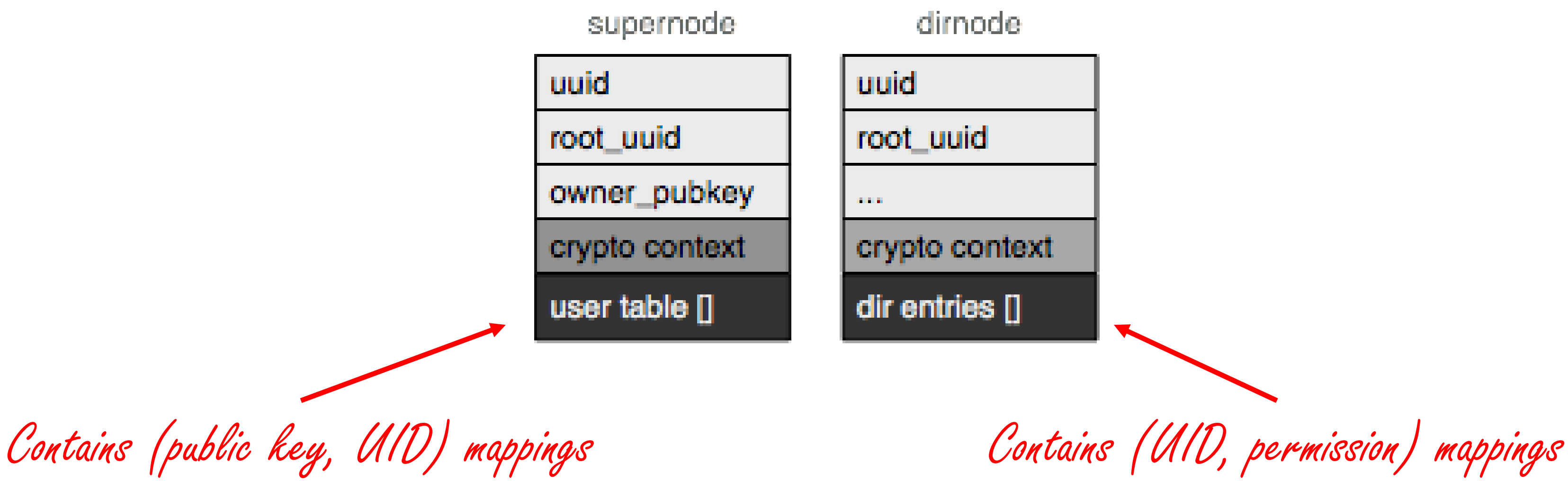


If the user successfully authenticates and is listed in the supernode, the NeXUS enclave mounts the volume

That works for the volume, but what about access control to individual files?

NeXUS has an ACL-based scheme for directory-level access controls

- Richer access control models are future work



The NeXUS enclave acts as a distributed reference monitor

- Every access must flow through the enclave (keys never leave!)
- Keys only used to decrypt files iff the authenticated user is authorized

We've integrated NeXUS with OpenAFS

Why OpenAFS? It's used at Pitt to offer networked storage to faculty, staff, and students!

Our implementation modifies the OpenAFS client and provides an administrative console for managing volumes and access controls

Implementation

- Total size: ~22k SLOC (excluding MbedTLS and keywrapping libraries)
- Shimlayer to interface with AFS: ~3200 SLOC
- Enclave size: ~9900 SLOC

Important: No modifications were made to the OpenAFS server!

We compared NeXUS over AFS to a stock AFS install

Microbenchmarks identify metadata I/O as a potential bottleneck

Prototype	File Size			
	1 MB	2 MB	16 MB	64 MB
OpenAFS	0.6154	1.5251	5.5504	22.2458
NeXUS	0.5143	1.4632	6.8117	28.5648
Metadata I/O	0.0957	0.1270	0.1438	0.8032
Enclave	0.0238	0.0973	0.5889	2.0774

(a) NeXUS runtime (seconds) on File I/O operations.

Enclave overhead is small in both I/O and directory benchmarks

Prototype	Number of files			
	1024	2048	4096	8192
OpenAFS	1.2713	2.6310	5.2658	11.9394
NeXUS	19.3864	38.6209	81.9818	172.2965
Metadata I/O	17.4407	34.6376	73.6640	154.3439
Enclave	0.3858	0.7909	1.6790	3.5514

(b) NeXUS runtime (seconds) on directory operations.

Larger directories incur significant overheads from Metadata I/O

We compared NeXUS over AFS to a stock AFS install

Database benchmarks show high performance for asynchronous operations, and expected delays for synchronous operations

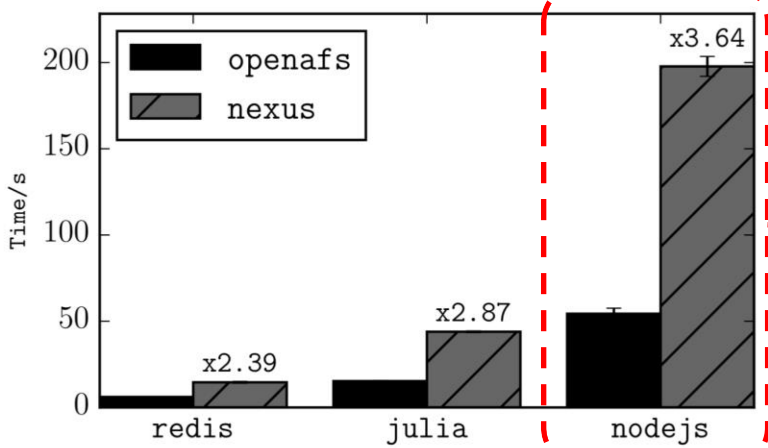
Operation	OpenAFS	NeXUS	Overhead
LevelDB			
Fillseq	10.5 MB/s	8.1 MB/s	1.29
fillsync	2.2 ms/op	4.5 ms/op	2.04
fillrandom	5.9 MB/s	3.7 MB/s	1.59
overwrite	4.0 MB/s	2.6 MB/s	1.53
readseq	664.6 MB/s	718.1 MB/s	0.94
readreverse	425.0 MB/s	425.7 MB/s	0.99
readrandom	2.27 μs/op	3.7 μs/op	1.62
fill100K	11.0 MB/s	7.2 MB/s	1.52
SQLITE			
fillseq	6.5 MB/s	6.4 MB/s	1.01
fillseqsync	14.4 ms/op	31.4 ms/op	2.18
fillseqbatch	70.2 MB/s	69.7 MB/s	1.00
fillrandom	4.2 MB/s	4.2 MB/s	1.00
fillrandsync	13.4 ms/op	31.2 ms/op	2.34
fillrandbatch	7.6 MB/s	7.7 MB/s	0.98
overwrite	3.4 MB/s	3.4 MB/s	1.00
overwritebatch	3.8 MB/s	4.4 MB/s	0.86

Table 2: Database benchmark results

Full propagation to disk involves waiting on sequential writes to metadata (i.e., fileboxes) and the data itself (i.e., file objects)

We compared NeXUS over AFS to a stock AFS install

In cloning git repositories, our overheads are impacted by metadata complexity



(a) Latency for cloning Git repositories.

	Redis	Julia	Nodejs
maxdepth	6	7	13
directories	59	116	1839
files	618	1096	19912
max dirsize	116	153	1458

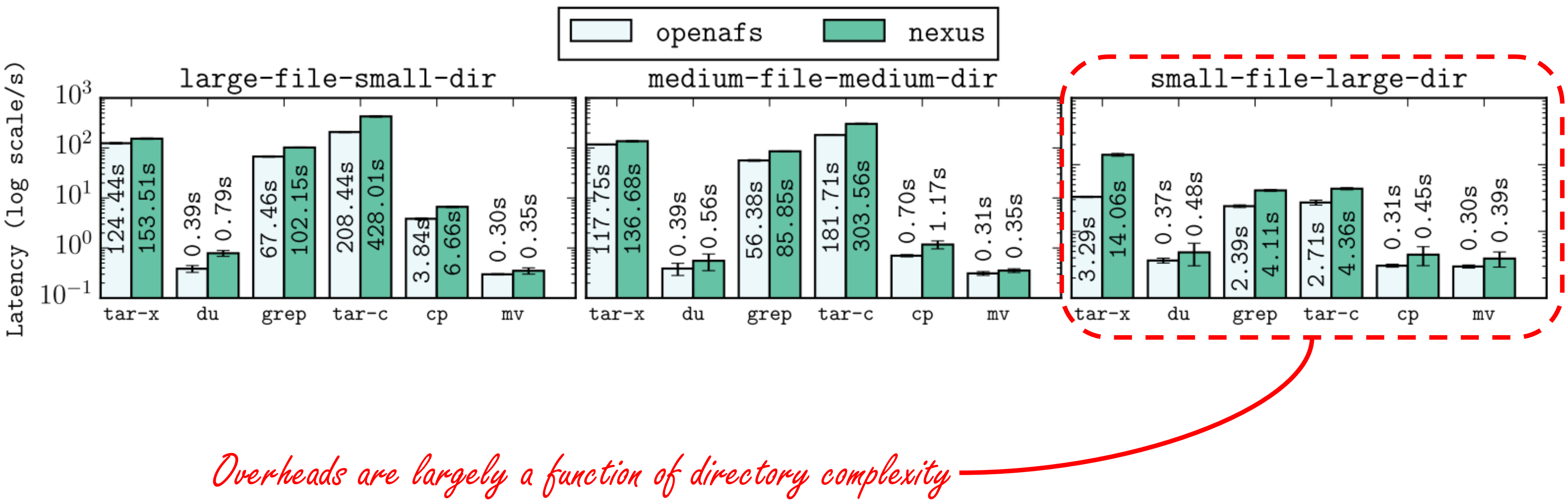
(b) Directory statistics for git repositories.

Deep directory trees and lots of files per directory means lots of dirnode and filebox operations

We compared NeXUS over AFS to a stock AFS install

Standard linux utilities run with acceptable overheads

Workload		#files	Total Size
LFSD	Large Files and Small Directory	32	3.2 GB
MFMD	Medium Files and Medium Directory	256	2.5 GB
SFLD	Small Files and Large Directory	1024	10 MB



What about the overheads of revocation?

Workload		#files	Total Size
LFSD	Large Files and Small Directory	32	3.2 GB
MFMD	Medium Files and Medium Directory	256	2.5 GB
SFLD	Small Files and Large Directory	1024	10 MB

Recall: Revocation in a purely cryptographic system is expensive!

- Download, decrypt, re-encrypt, upload, key distribution

Example: In LFSD, we're looking at 3.2 GB to shuffle around

Because keys in NeXUS never leave the enclave, life is simpler

- In LFSD, we're looking at modification of about ~3KB of metadata

Conclusions

Securing data stored in the cloud is of increasing importance

Revocation incurs high overheads in purely-cryptographic approaches

NeXUS combines client-side cryptography and trusted hardware

- Designed to balance portability and practicality
- Distributed access control via client-side SGX enclaves
- No server-side support necessary for deployment
- Key containment enables low-cost revocation

Reasonable overheads for a variety of workloads

Future work

- Increased throughput via server-side support
- Richer access controls