Obliviate: portable, efficient, and crash-consistent secure deletion enforced using the Rust compiler

Eugene Chou, Leo Conrad-Shah
Ethan Miller, Darrell Long, Andrew Quinn

UC SANTA CRUZ
BaskinEngineering

Center for Research
in Systems and Storage

Meta

NSF

Systems need to provide secure deletion

- **Secure deletion renders data irrecoverable either physically or computationally**
  - Adversaries cannot recovery securely deleted (erased data)
  - Even with direct access to the storage media
- **Motivated by data autonomy…**
  - Users should have control over their own data (how it’s shared, stored, removed etc.)
- **And by modern-day data privacy regulations**
  - GDPR, CCPA, GDPA, etc.

Obliviate

- **A system for fine-grained secure deletion on arbitrary storage media**
  - All data deletion (including truncates and overwrites) is securely deleted without undue delay
- **Sole requirement: erasable storage for a small, bounded amount of encryption keys**
- **Designed to be a portable* interposition layer**
  - Equip any application with transparent secure deletion
- **Achieves efficient crash consistency using novel principles around encryption key usage**
- **The first formally-verified secure delete system****

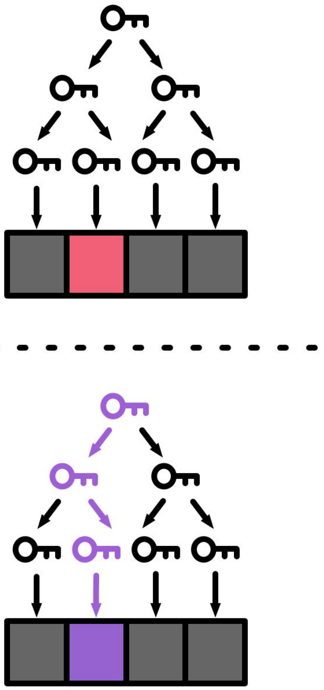
* across POSIX-compliant systems
** when completed

The rest of the talk

- **What we’ve done**
  - Background on secure delete systems
  - Obliviate’s original design principles
- **What we’re working on**
  - Addressing Obliviate’s performance with new design principles
- **What’s coming next**
  - Lightweight methods for formally verifying Obliviate

State-art-of-the-art: Large erasable memory[1]

- **Hierarchical application of cryptographic erasure**
  - Deletes cause **O(log n)** change to the key hierarchy
  - Changes to the hierarchy are commonly batched into **epochs**[2,3]
- **Secure deletion only requires the ability to erase the root key**
  - Only the root key needs to be stored in truly erasable storage
- **A *key management scheme (KMS)* implements large erasable memory**



[1] Di Crescenzo et. al., "How to Forget a Secret." (STACS '99)
[2] Reardon et. al., "Secure Data Deletion From Persistent Media." (CCS '13)
[3] Ratliff et. al., "Holepunch: Fast, Secure File Deletion with Crash Consistency (IEEE S&P '24)

Overwrite requires atomic data and KMS update

- **Encrypted overwrite of data d with key derived from KMS K**
  - End result should be data d' with key derived from KMS K'
  - Possible on-disk crash states:
    1. KMS, Enc(KMS, d)
    2. KMS', Enc(KMS, d) (data corruption!)
    3. KMS, Enc(KMS', d') (data corruption!)
    4. KMS', Enc(KMS', d)
- **Existing state-of-the-art secure delete systems resort to journaling for atomicity[1]**
  - Or don't support secure delete for overwrites[2]

[1] Reardon et. al., "Secure Data Deletion From Persistent Media." (CCS '13)
[2] Ratliff et. al., "Holepunch: Fast, Secure File Deletion with Crash Consistency (IEEE S&P '24)

Stability prevents data corruption

- **Stable key management scheme principle**
  - A KMS' key space doesn't change during an epoch
- **Just requires a unique, public IV to be atomically written for each write**
  - This prevents *key-reuse attacks*

Crash states without stability

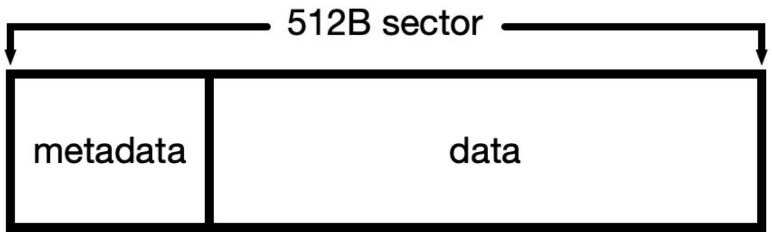
1. KMS, Enc(KMS, d)
2. KMS', Enc(KMS, d) (data corruption!)
3. KMS, Enc(KMS', d') (data corruption!)
4. KMS', Enc(KMS', d)

Crash states with stability

1. KMS, Enc(KMS, d)
2. KMS, Enc(KMS, d)
3. KMS, Enc(KMS, d')
4. KMS', Enc(KMS', d')

Atomic sector packing

- Atomic sector writes are portable across systems [1]
  - Not guaranteed by specifications, but observed to be true
- Idea: logically structure sectors to pack data and metadata together

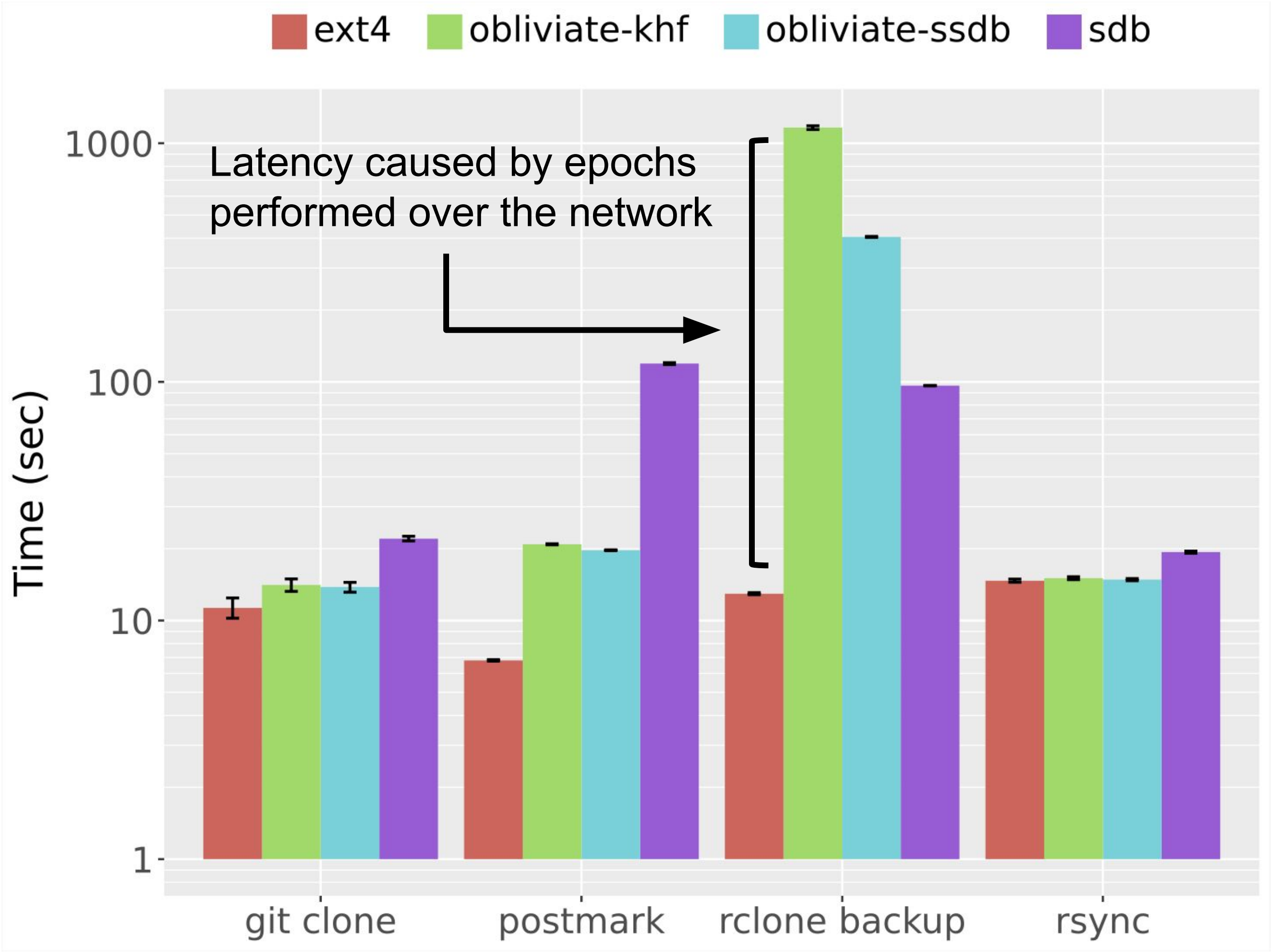


- Obliviate packs 16B of IV for every 496B of encrypted data
  - Packing isn't very amenable for use in the Linux block IO layer

[1] Pillai et. al., "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications." (OSDI '14)

Stability comes at a cost due to overwrites

- **Overwrites during an epoch require re-encryption to uphold secure delete guarantees**
  - Example:
    1. Block *b* is written with key *k* and IV *s*
    2. Block *b* is overwritten with key *k* and IV *s'*
  - Overwritten contents of *b* are still accessible using *k* (IVs are public)
    - Must re-encrypt *b* with a new key *k'*
- **With stability, epochs incur up to 2x write amplification**


ext4
obliviate-khf
obliviate-ssdb
sdb
Latency caused by epochs
performed over the network
Time (sec)
1000
100
10
1
git clone
postmark
rclone backup
rsync

The rest of the talk

- What we've done
  - Background on secure delete systems
  - Obliviate's original design principles
- What we're working on
  - Addressing Obliviate's performance with new design principles
- What's coming next
  - Lightweight methods for formally verifying Obliviate

Combining stability with single-use keys

- **Insight: re-encryption during epoch isn't needed if keys are used exactly once**
  - **Single-use key principle**
- **Obliviate realizes the single-key use principle using a *userspace buffer cache***
  - The buffer cache merges writes to sectors
  - This prevents epochs from occurring on each sector overwrite
- **Why a userspace buffer cache?**
  - Obliviate is implemented as a userspace interposition layer
  - Storage layers below the VFS don't have enough information for secure deletion
    - **To some extent, only *applications* have enough information**

The rest of the talk

- What we've done
  - Background on secure delete systems
  - Obliviate's original design principles
- What we're working on
  - Addressing Obliviate's performance with new design principles
- **What's coming next**
  - Lightweight methods for formally verifying Obliviate

How do we know Obliviate is correct?

CRSS

- **Problem: computationally intractable to determine if data has been securely deleted**
  - Black-box testing can't be done
  - We don't know if the implementation matches a correct specification
- **Idea: provide correctness by construction**
  - Step 1: proof-of-concept leveraging strong typing for assurances
  - Step 2: more powerful formal methods

Enforcing correct key usage with types

CRSS

- **Rust's type system can be used to encode the run-time state of an object in its type**
  - This is the ***typestate pattern***
  - Incurs no run-time overhead due to Rust's promise of zero-cost abstractions
- **Goal: use typestate as a lightweight method to verify key components of secure deletion**
  - The Rust compiler can guarantee ***compile-time*** correctness of things like:
    - Only encrypting data using a key that hasn't been used
    - Only writing encrypted data
    - Disallowing copying of keys that haven't been used

```rust
pub struct UsedNever;

#[derive(Clone, Copy)]
pub struct UsedOnce;

pub struct AffineKey<S, const KEY_SIZE: usize> {
    bytes: [u8; KEY_SIZE],
    _state: PhantomData<S>,
}

impl<const KEY_SIZE: usize> AffineKey<UsedNever, KEY_SIZE> {
    pub const fn from_rng(mut rng: impl CryptoRngExt) -> Self {
        Self {
            bytes: rng.gen_bytes(),
            _state: PhantomData,
        }
    }

    pub const fn consume(self) -> AffineKey<UsedOnce, KEY_SIZE> {
        AffineKey {
            bytes: self.bytes,
            _state: PhantomData,
        }
    }
}
```

zero-sized state types for an AffineKey

```rust
pub struct UsedNever;

#[derive(Clone, Copy)]
pub struct UsedOnce;

pub struct AffineKey<S, const KEY_SIZE: usize> {
    bytes: [u8; KEY_SIZE],
    _state: PhantomData<S>,
}

impl<const KEY_SIZE: usize> AffineKey<UsedNever, KEY_SIZE> {
    pub const fn from_rng(mut rng: impl CryptoRngExt) -> Self {
        Self {
            bytes: rng.gen_bytes(),
            _state: PhantomData,
        }
    }

    pub const fn consume(self) -> AffineKey<UsedOnce, KEY_SIZE> {
        AffineKey {
            bytes: self.bytes,
            _state: PhantomData,
        }
    }
}
```

```rust
pub struct UsedNever;

#[derive(Clone, Copy)]
pub struct UsedOnce;

pub struct AffineKey<S, const KEY_SIZE: usize> {
    bytes: [u8; KEY_SIZE],
    _state: PhantomData<S>,
}

impl<const KEY_SIZE: usize> AffineKey<UsedNever, KEY_SIZE> {
    pub const fn from_rng(mut rng: impl CryptoRngExt) -> Self {
        Self {
            bytes: rng.gen_bytes(),
            _state: PhantomData,
        }
    }

    pub const fn consume(self) -> AffineKey<UsedOnce, KEY_SIZE> {
        AffineKey {
            bytes: self.bytes,
            _state: PhantomData,
        }
    }
}
```

zero-sized state types for an AffineKey

only keys that have been used can be cloned/copied

```rust
pub struct UsedNever;

#[derive(Clone, Copy)]
pub struct UsedOnce;
```

zero-sized state types for an AffineKey

only keys that have been used can be cloned/copied

```rust
pub struct AffineKey<S, const KEY_SIZE: usize> {
    bytes: [u8; KEY_SIZE],
    _state: PhantomData<S>,
}
```

generic over S and KEY_SIZE

PhantomData is zero-sized and allows for logical association of a type

```rust
impl<const KEY_SIZE: usize> AffineKey<UsedNever, KEY_SIZE> {
    pub const fn from_rng(mut rng: impl CryptoRngExt) -> Self {
        Self {
            bytes: rng.gen_bytes(),
            _state: PhantomData,
        }
    }

    pub const fn consume(self) -> AffineKey<UsedOnce, KEY_SIZE> {
        AffineKey {
            bytes: self.bytes,
            _state: PhantomData,
        }
    }
}
```

```rust
pub struct UsedNever;

#[derive(Clone, Copy)]
pub struct UsedOnce;
```

zero-sized state types for an AffineKey

only keys that have been used can be cloned/copied

```rust
pub struct AffineKey<S, const KEY_SIZE: usize> {
    bytes: [u8; KEY_SIZE],
    _state: PhantomData<S>,
}
```

generic over S and KEY_SIZE
PhantomData is zero-sized and allows for logical association of a type

```rust
impl<const KEY_SIZE: usize> AffineKey<UsedNever, KEY_SIZE> {
    pub const fn from_rng(mut rng: impl CryptoRngExt) -> Self {
        Self {
            bytes: rng.gen_bytes(),
            _state: PhantomData,
        }
    }

    pub const fn consume(self) -> AffineKey<UsedOnce, KEY_SIZE> {
        AffineKey {
            bytes: self.bytes,
            _state: PhantomData,
        }
    }
}
```

can only construct keys using cryptographically-secure PRNGs

```rust
pub struct UsedNever;
```

zero-sized state types for an AffineKey

```rust
#[derive(Clone, Copy)]
pub struct UsedOnce;
```

only keys that have been used can be cloned/copied

```rust
pub struct AffineKey<S, const KEY_SIZE: usize> {
    bytes: [u8; KEY_SIZE],
    _state: PhantomData<S>,
}
```

generic over S and KEY_SIZE
PhantomData is zero-sized and allows for logical association of a type

```rust
impl<const KEY_SIZE: usize> AffineKey<UsedNever, KEY_SIZE> {
    pub const fn from_rng(mut rng: impl CryptoRngExt) -> Self {
        Self {
            bytes: rng.gen_bytes(),
            _state: PhantomData,
        }
    }
```

can only construct keys using cryptographically-secure PRNGs

```rust
    pub const fn consume(self) -> AffineKey<UsedOnce, KEY_SIZE> {
        AffineKey {
            bytes: self.bytes,
            _state: PhantomData,
        }
    }
}
```

takes ownership of the key and returns it as UsedOnce
(this is optimized out by the compiler)

```rust
pub trait AffineCrypter<const KEY_SIZE: usize> {
    type Error;

    fn encrypt(
        key: AffineKey<UsedNever, KEY_SIZE>,
        data: &mut [u8],
    ) -> Result<AffineKey<UsedOnce, KEY_SIZE>, Self::Error>;

    fn decrypt(key: AffineKey<UsedOnce, KEY_SIZE>, data: &mut [u8]) -> Result<(), Self::Error>;
}
```

```rust
pub trait AffineCrypter<const KEY_SIZE: usize> {
    type Error;

    fn encrypt(
        key: AffineKey<UsedNever, KEY_SIZE>,
        data: &mut [u8],
    ) -> Result<AffineKey<UsedOnce, KEY_SIZE>, Self::Error>;

    fn decrypt(key: AffineKey<UsedOnce, KEY_SIZE>, data: &mut [u8]) -> Result<(), Self::Error>;
}
```

generic over KEY_SIZE
must have an associated error type

```rust
pub trait AffineCrypter<const KEY_SIZE: usize> {
    type Error;

    fn encrypt(
        key: AffineKey<UsedNever, KEY_SIZE>,
        data: &mut [u8],
    ) -> Result<AffineKey<UsedOnce, KEY_SIZE>, Self::Error>;

    fn decrypt(key: AffineKey<UsedOnce, KEY_SIZE>, data: &mut [u8]) -> Result<(), Self::Error>;
}
```

generic over KEY_SIZE
must have an associated error type

takes ownership of a key that hasn't been used
encrypts data, returns the "consumed" key

```rust
pub trait AffineCrypter<const KEY_SIZE: usize> {
    type Error;

    fn encrypt(
        key: AffineKey<UsedNever, KEY_SIZE>,
        data: &mut [u8],
    ) -> Result<AffineKey<UsedOnce, KEY_SIZE>, Self::Error>;

    fn decrypt(key: AffineKey<UsedOnce, KEY_SIZE>, data: &mut [u8]) -> Result<(), Self::Error>;
}
```

generic over KEY_SIZE
must have an associated error type

takes ownership of a key that hasn't been used
encrypts data, returns the "consumed" key

decrypts data using a "consumed" key

Empirical results of using typestate (as of now)

- **Caught a logic error when placing data into the buffer cache**
  - Forgot to decrypt sector before buffering it
  - Manifested as a compiler error reporting mismatched types
    - E.g., expected `Sector<Plaintext>`, found `Sector<Ciphertext>`
- **Type-driven design of key management scheme update**
  - Obliviate KMS: copy-on-write B+-tree
  - Batch update was designed to enforce that updated nodes are only paged to disk once
    - A natural consequence of having single-use keys

Covering the “proof gap”

- **Typestate cannot enforce correctness of all aspects of Obliviate**
  - But it does provide a lot of coverage
- **Kani (https://github.com/model-checking/kani)**
  - Model-checking to see if functions meet their intended specification
- **Verus (https://github.com/verus-lang/verus)**
  - For more complex theorem proving
- **Goal: minimize the proof gap needed to be covered by Kani/Verus**

Goals for 2024 - 2025

CRSS

- **Submissions to:**
  - ATC ‘25
  - ???
- **Future work:**
  - Applying model checking and proof checking to Obliviate
  - Potential application of Obliviate to single-level stores

Conclusion

- **Obliviate is a system for portable fine-grained secure deletion**
  - All data deletion (including truncates and overwrites) is securely deleted
  - Works on any application, and on any storage media
- **Sole requirement: erasable storage for a small, bounded amount of encryption keys**
- **Achieves efficient crash consistency using novel principles around key usage**
- **The (hopefully soon-to-be) first formally-verified secure delete system**

Thanks for listening!

CRSS

Questions?

email: euchou@ucsc.edu

And thanks to all the sponsors!