HOLEPUNCH: Fast, Secure File Deletion with Crash Consistency

Zachary Ratliff *Harvard University*
Wittmann Goh *Harvard University*
Abe Wieland *Harvard University*
James Mickens *Harvard University*
Ryan Williams *Northeastern University*

Abstract

A file system provides secure deletion if, after a file is deleted, an attacker with physical possession of the storage device cannot recover any data from the deleted file. Unfortunately, secure deletion is not provided by commodity file systems. Even file systems which explicitly desire to provide secure deletion are challenged by the subtleties of hardware controllers on modern storage devices; those controllers obscure the mappings between logical blocks and physical blocks, silently duplicate physical blocks, and generally make it hard for host-level software to make reliable assumptions about how file data is kept on the device. State-of-the-art frameworks for secure deletion also have no crash consistency, meaning that an ill-timed power outage or software fault will desynchonize keys and the associated encrypted file data, corrupting the file system.

In this paper, we present HOLEPUNCH, a new software-level approach for implementing secure deletion. HOLEPUNCH treats the storage device as a black box, providing secure deletion via cryptographic erasure. HOLEPUNCH uses per-file keys to transparently encrypt outgoing file writes and decrypt incoming file reads, ensuring that all physical data in the storage device is always encrypted. HOLEPUNCH uses puncturable pseudorandom functions (PPRFs) to quickly access file keys; upon the deletion of file f, HOLEPUNCH updates the PPRF so that, even if the PPRF is recovered, the PPRF cannot be used to generate f's key. By using PPRFs instead of the key trees leveraged by prior work, HOLEPUNCH reduces both the memory pressure caused by key management and the number of disk IOs needed to access files. HOLEPUNCH stores its master key in secure TPM storage, and uses a novel journaling scheme to provide crash consistency between TPM state and on-disk state.

1. Introduction

File systems allow users to read and write data kept on an underlying storage device. An important file system operation is deletion, i.e., the removal of a file from a storage device. A file system provides *secure deletion* if, post-deletion, a file's data is unrecoverable by an attacker who obtains unrestricted access to the storage device.

Secure deletion is a fundamental security property, ensuring the confidentiality of sensitive data (§2). However, commodity file systems often do not provide secure deletion. For example, on popular Linux file systems like EXT4, deleting a file will mark the file's on-device data blocks as unallocated, but will not synchronously overwrite the contents of those blocks; the original data is only replaced if the file system *eventually* chooses to allocate the blocks to a new file. Even when users take explicit action to overwrite old file data (e.g., by using the Linux `shred` command to zero-fill deleted files), the solid-state drives (SSDs) common in modern laptops and desktops implement wear-leveling schemes [11]. Wear leveling evenly distributes writes across physical storage pages, regardless of the logical storage pages that are targeted by file system IOs. Wear leveling avoids write hot spots (and thus premature failures) for any particular SSD page; however, wear leveling means that, even if a file system or `shred`-like tool tries to explicitly overwrite a file's data, the file's data may continue to persist in the underlying SSD pages.

Wear leveling is typically implemented via a hardware controller in the SSD itself. From the perspective of host software like an operating system, the controller is a black box. Early attempts to implement secure deletion via host-level software leveraged assumptions about how controllers worked [22], e.g., to allow the host to guess how logical storage regions are mapped to physical ones. However, for any given storage device, such assumptions might have never been true, or might have been true at some point but subsequently invalidated by an update to device firmware [20]. For example, prior attempts at secure deletion often assumed that each logical file block maps to a single physical block; this assumption of non-duplication is false on SSDs.

Subsequent research demonstrated that host-level software could indeed provide secure deletion for black-box storage [20], [23]. To do so, the host software must leverage a small, trusted storage device like a TPM chip [29] that can generate true random numbers (e.g., to use as keys), store small amounts of information (e.g., keys) in non-volatile memory, and resist physical tampering that would otherwise interfere with the first two tasks. Host-level software only exchanges encrypted data with the black-box storage device, keeping the relevant keys in trusted storage, and achieving secure file deletion by discarding a deleted file's key in the trusted storage.

Unfortunately, trusted devices like a TPM chip or a smart card have limited storage space, requiring host-level software to carefully manage that space to store key material for a large number of files [6], [20], [23]. For example, in ERASER [20], host software keeps an encrypted file-key tree in the untrusted storage device, with trusted hardware only storing the master key which decrypts the tree. This strategy requires ERASER to issue IOs that scale with the logarithm of the file system size; in practice, this overhead significantly degrades performance for IO-intensive workloads, particularly for non-SSD storage like hard drives that suffer from mechanical access penalties. To reduce IO amplification, ERASER stores the upper layers of the file-key tree (FKT) in memory, but the result is non-trivial memory requirements (roughly 65 MB per 1 TB of file system data).

A second problem with the traditional FKT approach involves crash consistency. During a file creation or file deletion, host-level software must update the FKT on the untrusted storage device. During rotation of the master key, software must update both the trusted hardware and the untrusted storage. If a machine crashes or suffers a power outage during these critical operations, keys may become inconsistent with file system state in the untrusted device, causing some files to become irrevocably lost.

In this paper, we introduce HOLEPUNCH, a block device driver[1] that provides memory-efficient, crash-consistent secure deletion. HOLEPUNCH treats the untrusted storage device as a black box. To access a file, HOLEPUNCH evaluates a *puncturable pseudorandom function* (PPRF) at a specific point, and then performs a *single* extra disk I/O to retrieve an encrypted per-file key that is decrypted using the output of the PPRF. To delete a file, HOLEPUNCH performs a puncture operation on the PPRF key, causing the PPRF to "forget" how to generate the associated per-file key.

The PPRF state grows in size after a puncture, and the new state must be written to untrusted storage to reflect the file deletion. A naive implementation of HOLEPUNCH would require per-puncture IO costs that grow linearly with each file deletion. To avoid this problem, HOLEPUNCH (1) periodically rotates the PPRF state to reset its size, and (2) leverages a tree-like structure to make the on-disk PPRF state efficiently updateable. HOLEPUNCH uses a TPM to store the encryption key for the root of the tree, with each leaf containing an encrypted chunk of the PPRF key. However, HOLEPUNCH's in-memory overheads are less than those of ERASER because HOLEPUNCH's PPRF allows it to efficiently generate many keys on-the-fly, unlike ERASER (which often must explictly store keys in memory to avoid the IO costs of fetching them from disk). The lower memory footprint is especially beneficial for machines with large disks, e.g., datacenter servers whose individual drives store tens of TBs (§6.2). Unlike ERASER, HOLEPUNCH also provides crash consistency by carefully updating TPM state and on-disk structures using a journaling scheme. HOLEPUNCH leverages *deletion batching* (§5.5) to reduce the IO cost of secure deletion without sacrificing the crash consistency provided by journaling.

In summary, HOLEPUNCH is a software-level approach for secure deletion that simultaneously:

- makes no assumptions about how storage devices internally manage physical regions,
- reduces both the memory space and storage IOs needed to manage file keys, and
- provides crash consistency, ensuring that the file system is usable after a crash or power outage occurs.

Given these properties, we believe that HOLEPUNCH is the first *practical* system for secure deletion of file system data.

2. Background

Secure deletion frameworks are increasingly relevant for several reasons. First, in recent years, governments have begun to issue laws that restrict how businesses may collect and retain user data. For example, the European Union's General Data Protection Regulation (GDPR) [8] and the California Consumer Privacy Act (CCPA) [26] both confer users with a "right to be forgotten." This right allows a user to demand that a business delete all information that the business stores about the user. The right to be forgotten is a boon for privacy advocates, but strong technical mechanisms for *implementing* this right are crucial for the practical enforcement of GDPR-style regulations. Secure deletion via cryptographic erasure would be a promising solution if challenges involving key management, performance, and crash consistency could be solved.

Secure deletion is attractive for non-regulatory reasons as well, because improper sanitization of storage media intrinsically poses a security risk to both individuals and organizations. For example, the Pennsylvania Department of Labor and Industry accidentally resold unsanitized computers that contained thousands of state employee records [30]. The United States Veterans Administration also gave away improperly-sanitized hard disks that contained sensitive patient information like credit card numbers [12]. If a computer does not enable secure deletion technology by default, many individuals and organizations will not bother to sanitize storage devices.

2.1. Prior Work on Encrypted Storage

Encrypted storage systems like BITLOCKER [16] and DM-CRYPT [19] aspire to guarantee that storage data at rest is encrypted. However, as we explained in Section 1, host-level software lacks insight into how storage controllers work, meaning that host-level software (whether residing in the OS or in user-space tools) cannot provide strong guarantees about secure deletion.

Some storage devices natively implement encryption. In these "self-encrypting drives" (SEDs), all cleartext writes from the host are automatically encrypted by the device before they become persistent on the device; all data retrieved from the device is automatically decrypted by the device before being returned to the host. SEDs implement the ATA security feature specification [27] or the newer TCG Opal specification [28]. However, SEDs are beset by a variety of problems arising from inherent weaknesses in the specs and poor implementation choices made by SED vendors [17]. For example, Meijer and van Gastel [17]

[1] A file system like EXT4 or NTFS sits atop a block driver that directly interacts with a storage device. The file system implements higher-level abstractions like files and directories atop the block interface.

found that the Crucial MX100 SSD prompted the user for a password to unlock the device, but did not use the password to derive encryption keys. Furthermore, the SSD's firmware was debugabble via JTAG [13] interfaces,[2] such that an attacker with physical possession of the SSD could use JTAG to force the firmware's password validation routine to always return "success" and allow the device to unlock. Meijer and van Gastel also demonstrated that, if an attacker can modify SED firmware, the attacker can often use the corrupted firmware to force a device to reveal cleartext versions of encryption keys. HOLEPUNCH provides secure deletion on arbitrary black-box storage devices, regardless of whether those devices do or do not correctly support native encryption, and regardless of whether those devices internally perform data duplication or otherwise obscure logical-to-physical storage mappings. From the perspective of HOLEPUNCH, the TPM is in the trusted computing base for secure deletion, *but the storage device itself is not.*

2.2. Security Goals and Threat Model

More concretely speaking, HOLEPUNCH has two security goals:

- **Confidentiality:** An adversary that obtains physical access to the storage device should be unable to view its contents unless the user's HOLEPUNCH password is known.
- **Secure deletion:** An adversary that obtains physical access to the storage device *and* knows the user's HOLEPUNCH password should be unable to recover previously deleted files.

We formalize these goals in Section 4.

Our threat model allows an attacker to physically examine the storage device in arbitrary ways and run attacker-controlled firmware on the device. However, we assume that the adversary is (1) computationally-bounded and (2) unable to recover *deleted* data from the TPM chip. The former constraint is a standard cryptographic one, whereas the latter is realistic given the tamper-resistant nature of hardware-implemented TPMs. Note that we do not restrict the adversary from reading data that is *currently resident* in the TPM's memory. The reason is that, upon gaining control of a user's machine, the adversary can use the local operating system to interact with the TPM via the TPM's standard interface. Through this interface, the adversary can retrieve the TPM's current memory contents and use the resident cryptographic information to derive the encryption keys for files that have not been securely deleted. However, securely-deleted files must be unrecoverable in the sense that, from the attacker's perspective, data belonging to a securely-deleted file should be indistinguishable from random bits.

3. Design

In this section, we provide an overview of HOLEPUNCH's design, explaining how its cryptographic mechanisms interact with file system abstractions and storage devices. We defer a detailed discussion of our HOLEPUNCH *implementation* to Section 5, since knowledge of those implementation details is not required to understand HOLEPUNCH's basic approach.

HOLEPUNCH is a block device driver that sits between the file system and the underlying storage device. The HOLEPUNCH driver makes no assumptions about the internal workings of a storage device. The driver encrypts each file f with a random per-file key k_f, and stores each k_f on disk; per-file keys are grouped into 4 KB blocks, with the keys in each storage block s all encrypted by a wrapping key k_{τ_s}. Each k_{τ_s} is generated using the in-memory PPRF, passing as input a random τ_s value that is stored alongside the associated keys in the block. The full collection of these blocks, which HOLEPUNCH calls the *file key table*, is stored in a deterministic location on disk. Given a file f, HOLEPUNCH finds the storage location of the relevant table entry by using f's inode number[3] to index into the table; HOLEPUNCH then retrieves the τ_s stored in the file table entry, evaluates the PPRF on τ_s to generate k_{τ_s}, and finally uses k_{τ_s} to decrypt the rest of the table entry and extract the relevant k_f.

HOLEPUNCH evaluates the PPRF using an in-memory copy of the PPRF state. However, when that state changes because of a file deletion or PPRF refresh, HOLEPUNCH also issues the change to an on-disk representation of the PPRF, so that the change persists across machine reboots and crashes. HOLEPUNCH stores the on-disk state of the PPRF using a tree-like structure. Each leaf node contains an encrypted chunk of PPRF state, and each encrypted interior node contains (1) pointers to children in the tree and (2) the key needed to decrypt those children. HOLEPUNCH's *master key* encrypts the second-layer nodes in the on-disk PPRF key. The master key is randomly generated by the TPM during HOLEPUNCH initialization. When a user's machine is powered off, HOLEPUNCH stores the master key (encrypted with the user's password fed through a KDF) in the TPM. During the boot process, HOLEPUNCH prompts the user for the password and uses it to decrypt the master key and thereby unlock access to the file system. Figure 1 gives an overview of the PPRF tree structure.

3.1. Puncturable PRFs

Ideally, an encrypted storage approach would use a different encryption key for each file, to limit the damage if a particular file's key were leaked. However, using a key per file introduces management challenges. A modern file system like EXT4 supports hundreds of millions of files,

[2] JTAG is a hardware debugging protocol that provides full control over a device. For example, JTAG allows a debugger to read and write arbitrary on-device registers and memory regions.

[3] An inode is an on-disk, per-file metadata structure maintained by the file system. Among other things, an inode stores pointers to the file's data blocks.

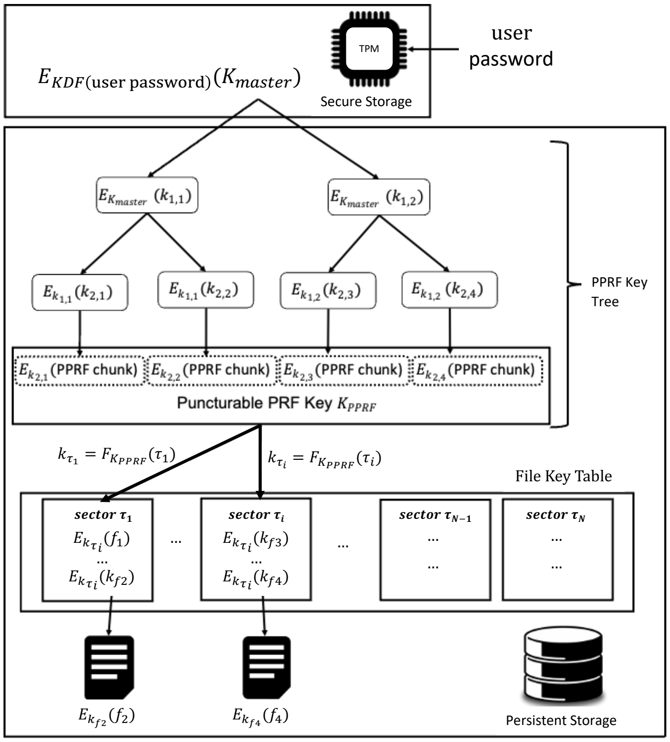

Figure 1. Overview of the HOLEPUNCH system design. Note that, in our HOLEPUNCH implementation, the second layer of the PPRF tree fits entirely within one block. This property is useful during journaling (§3.4.1).

and reading or writing any particular file requires storage software to first locate the file-specific key and pull it into memory. The storage system faces a tension between memory overhead and performance: aggressively caching keys in memory creates memory overhead (and may eventually lead to swapping behavior that hurts IO performance), but leaving most keys in encrypted form on disk will lead to IO amplification (since host software cannot read or write a file before fetching the appropriate on-disk key).

To alleviate this tension, HOLEPUNCH uses a puncturable pseudorandom function (PPRF) to efficiently generate some cryptographic keys on the fly, avoiding the need to explicitly store the keys in memory to enable fast access to them. Before explaining how PPRFs work, we first discuss how vanilla pseudorandom functions (PRFs) operate. A PRF simulates a random oracle by accepting a random key k and outputting strings that are computationally indistinguishable from a truly random function. Formally, we define a PRF as a family of functions:

$$\{F_k : \{0,1\}^{m(\lambda)} \rightarrow \{0,1\}^{\ell(\lambda)}\}$$

where $k \leftarrow_{\$} \{0,1\}^{\lambda}$, $\lambda \in \mathbb{N}$ is a security parameter, and m and ℓ polynomials. The PRF F should satisfy two properties:

- **Efficiency:** $F_k(x)$ is computable in time polynomial in λ given k and x.
- **Security:** No polynomial-time adversary can distinguish F_k from a truly random function except with probability negligible in λ.

Goldreich, Goldwasser, and Micali showed that PRFs exist under the assumption that pseudorandom generators exist [10]. While their PRF construction is primarily of theoretical utility, one can adapt it to construct a more powerful cryptographic primitive known as a puncturable PRF [3], [4], [15]. A puncturable PRF is a pseudorandom function F supporting a *puncture* operation that converts the PRF key k into a new key k^* that is punctured over a set $S \subseteq \{0,1\}^{m(\lambda)}$ with the following properties.

- **Functionality Preserving:** $F_k(x') = F_{k^*}(x')$ for all $x' \notin S$.
- **Pseudorandom at punctured points:** For all $x \in S$, and any polynomial time adversary with access to k^*, the value $F_{k^*}(x)$ is computationally indistinguishable from random.

In other words, a puncturable PRF can output a key k^* punctured at a set of points S such that the PRF will evaluate as before on every point not in S, but will reveal nothing about the PRF's prior output for points in S. As we explain below, HOLEPUNCH leverages this feature to forget the encryption keys associated with securely-deleted files.

3.2. Creating, Reading, and Writing Files

When a new file f is created, HOLEPUNCH asks the TPM to generate a random key k_f. HOLEPUNCH will use this key to encrypt the new file's data blocks.[4] HOLEPUNCH also maintains an on-disk *file key table* in which all k_f values are stored. The file key table is located at a fixed (logical) location on the storage device, and is indexed by the inode number i_f of a file. In the file key table, each 4 KB storage block s contains 127 32-byte k_f values (one for each of the 127 files with adjacent i_f numbers), a single 8-byte tag value τ_s, 16 magic bytes (to assist with PPRF key rotation (§3.4.3)), and 8 bytes of padding. HOLEPUNCH feeds τ_s to the PPRF to generate the wrapping key k_{τ_s} for the key table block; HOLEPUNCH uses k_{τ_s} to encrypt the 127 k_f values in the block. Note that the on-disk τ_s is stored in cleartext, which is safe because even if an attacker learns τ_s, the attacker cannot generate k_{τ_s} without knowing the PPRF key. Also note that we assume that writes to a 4 KB block are atomic; this assumption is true on modern storage devices and is important for HOLEPUNCH's crash consistency scheme (§3.4).

[4] In the file system literature, a "sector" typically refers to the minimum size of data that a magnetic disk can read or write, a "page" refers to the minimum IO size for an SSD, and a "block" refers to the granularity at which a file system allocates data structures on the storage device (such that a block must span at least one sector/page, but may span more than one). In this paper, we use the term "block" to refer to a chunk of a file key table or to a piece of data pointed to by an inode; however, we use the term in a way that is agnostic about whether the underlying storage device is a magnetic disk, an SSD, or something else. We use a 4 KB block size for the file key table because 4 KB is the smallest sector/page size on commodity storage devices.

When a file corresponding to inode i_f is opened, the file system fetches the unencrypted inode as usual. HOLEPUNCH also performs extra work to:

1) fetch the key table block s associated with inode i_f,
2) extract τ_s from the fetched block,
3) evaluate the PPRF on τ_s to generate k_{τ_s}, and
4) decrypt the ciphertext part of the block using k_{τ_s}, obtaining the plaintext k_f associated with inode i_f.

Later, when the file system wants to read or write a file block pointed to by i_f, HOLEPUNCH uses k_f to decrypt incoming reads of a file block, and encrypt outgoing writes to a file block.

In summary, relative to a traditional file system, HOLEPUNCH requires an extra disk IO during file creation (to update a key table block) and an extra disk IO during file opening (to read a key table block). However, our HOLEPUNCH implementation caches hot table blocks in memory to amortize the fetch costs. Also note that, even without caching, HOLEPUNCH's single extra disk IOs scale better than ERASER's IOs (which can scale logarithmically with the number of files unless large portions of the FKT state are kept in memory).

3.3. Deleting Files

Upon deleting file f, HOLEPUNCH must ensure that k_f is unrecoverable. To do so, HOLEPUNCH punctures the PPRF key at the point τ_s associated with the inode number i_f. A complication is that each τ_s covers 127 per-file keys, meaning that, absent remediating action, a puncture at τ_s would render an additional 126 files unrecoverable. To avoid this situation, HOLEPUNCH builds a new key table block for i_f, keeping the file keys belonging to non-deleted files intact, but assigning a new τ_s' to the block and re-encrypting the block via PPRF evaluation over τ_s'. Our HOLEPUNCH implementation generates a new τ_s' value simply by incrementing an in-memory atomic counter which is used to ensure that each tag value is unique. Ensuring uniqueness is important because, if a single tag were used by multiple key table blocks, then deleting a file covered by just one of those blocks would require re-encryption of all of those blocks.

The end-to-end process for file deletion involves the following steps:

1) read and decrypt the key table block for inode number i_f,
2) replace i_f's file key k_f in the block with a new random key
3) puncture the PPRF key at the block's tag τ_s,
4) set τ_s' for the block to be the current value of an atomic tag counter and then increment the counter,
5) encrypt the new key table block using the output of the PPRF evaluated on τ_s',
6) rotate the master key and encrypt the punctured PPRF key under the new master key (to provide forward secrecy), and finally
7) write the encrypted PPRF key, the atomic tag counter, and the encrypted key table block to disk.

HOLEPUNCH's PPRF key grows linearly with the number of deletions. As a result, Step 6 will gradually incur more IOs as HOLEPUNCH deletes more files. To mitigate this problem, HOLEPUNCH uses a tree structure to represent the PPRF on disk (Figure 1). HOLEPUNCH's PPRF tree is a three level n-ary tree. HOLEPUNCH's master key encrypts a block of keys at the second layer, which in turn encrypt blocks of third-layer keys. Each key in the third layer encrypts a PPRF key chunk. Using this scheme, each puncture operation only requires HOLEPUNCH to write a few blocks to update the on-disk PPRF key. That being said, HOLEPUNCH does periodically refresh the PPRF key to prevent it from growing unboundedly (§3.4.3).

3.4. Journaling

A key property of modern file systems is crash consistency [2]. Crash consistency ensures that, after an unexpected power failure or system crash, invariants involving on-disk file system metadata continue to hold. Loss of the most recent file system updates is acceptable, but the file system must remain usable post-crash.

The primary approach for guaranteeing crash consistency is to enforce atomic updating of on-disk file system metadata. For example, to create a new file `/foo/bar.txt`, an EXT-like file system [5] must update three on-disk structures: a bitmap (to indicate that a new inode for `bar.txt` has been allocated), the new inode itself (e.g., to record `bar.txt`'s creation time and to indicate that the file currently contains no data blocks), and an updated directory entry (so that the `/foo` directory has a pointer to the new inode). To ensure crash consistency, all three updates must be treated as an atomic unit—in other words, after a crash, the file system must perceive that either all three updates successfully hit the disk, or none of them did. Otherwise, problems would emerge if, e.g., the update to the directory entry succeeded but the update to the new inode did not, meaning that `bar.txt`'s data blocks would be whatever blocks the old, junk inode happened to point to.

In a cryptographic approach for secure deletion, key material must be kept in sync with the on-disk structures that are protected by those keys. If this invariant is broken, then encrypted files will become unrecoverable when the associated keys are lost or not kept up to date. ERASER, which is the current state-of-the-art file system for secure deletion, does not ensure consistency between encryption keys and the associated on-disk state. As a result, we have empirically observed that, post-crash, up to 64 files may be lost in ERASER. For example, such data loss will happen if a crash occurs *after* ERASER rotates its TPM master key, but *before* ERASER updates its on-disk FKT. The inconsistency arises from ERASER's non-atomic approach to updating the key material in the TPM and in on-disk FKT.

HOLEPUNCH provides crash consistency via a journaling approach [2]. The high-level idea is that, before HOLEPUNCH issues an update u to on-disk key material and/or TPM state, HOLEPUNCH first writes u to a special region of the disk called the journal. The write to the journal memorializes HOLEPUNCH's intent to update key material. Once the journal write completes, HOLEPUNCH actually issues u to the disk and/or the TPM. When u completes, HOLEPUNCH removes the journal entry. This approach provides crash consistency as follows:

- If the machine crashes before the journal is updated, the update is lost but key material remains synchronized between the disk and the TPM.
- If the machine crashes after the journal is updated but before the update is completely applied to the disk and/or TPM, then during the boot process, HOLEPUNCH detects the non-empty journal and issues u before resuming normal operation. u is idempotent, so reissuing it is always safe, even if u partially completed before the crash occurred.
- If the machine crashes after the disk and/or TPM are updated but before the journal is cleared, HOLEPUNCH reissues u. Unbeknownst to HOLEPUNCH the reissuing is unnecessary, but doing so is safe because u is idempotent.

In the remainder of the section, we provide more details about what HOLEPUNCH writes to the journal during various key operations.

3.4.1. Rotating the Master Key.

As shown in Figure 1, HOLEPUNCH's master key K_{master} is used as the root of the on-disk PPRF tree. When the machine is powered off, HOLEPUNCH stores K_{master} in the TPM. The version of K_{master} in the TPM must be the version of K_{master} used as the root of on-disk PPRF tree; otherwise, when the machine boots, HOLEPUNCH will be unable to decrypt the PPRF tree and generate per-file k_f keys (§3.2).

Suppose that K_{old} is the old master key and K_{new} is the new one. To atomically rotate the key, HOLEPUNCH does the following:

1) Write to the journal $E_{K_{new}}(\ell_2)$, where ℓ_2 is the block containing the second level of the PPRF tree. Since the write is a block in size (see Figure 1), it is atomic.
2) Write to the journal the values $E_{K_{old}}(K_{new})$ and $H(K_{old})$ where H is a cryptographic hash function. Since these values fit within a block, this write is also atomic. The journal update phase is now complete.
3) Atomically overwrite the second-level block of the on-disk PPRF tree; in particular, replace $E_{K_{old}}(\ell_2)$ with the version in the journal ($E_{K_{new}}(\ell_2)$).
4) In the TPM's NVRAM, atomically overwrite K_{old} with K_{new}.
5) Finally, clear the two journal blocks, setting all of the bytes to zero. Doing so requires two writes that are individually atomic, meaning that the machine may crash immediately after the issuing of the first write. However, during recovery, HOLEPUNCH will consider the journal empty if the first journal block is all-zeroes, regardless of whether the second block contains data. For this approach to work, Step 1 above must write to the *second* journal block, with Step 2 overwriting the zeroes in the *first* journal block.

During a reboot, HOLEPUNCH checks whether the journal is clear. If not, HOLEPUNCH enters recovery mode. HOLEPUNCH first uses the zeroes-checking approach described above to determine whether both blocks of the journal belong to the same logical update. If not, HOLEPUNCH terminates recovery because either (1) the on-TPM master key was not rotated and the PPRF tree root is encrypted with $E_{K_{old}}$, or (2) the on-TPM key was set to $E_{K_{new}}$ and the PPRF tree root is encrypted with $E_{K_{new}}$, but the machine crashed after one journal block was updated but before the other was updated. In either scenario, HOLEPUNCH just clears the journal and resumes normal operation.

If the journal blocks *do* belong to the same logical update, HOLEPUNCH reads the current key K_{TPM} from the TPM, and calculates $H(K_{TPM})$. If the calculated value matches the key hash in the journal, i.e., if $K_{TPM} == K_{old}$, then the machine crashed at some point after Step (2) and before Step (5). Thus, HOLEPUNCH redoes Steps (3) and (4), replacing $E_{K_{old}}(\ell_2)$ with the version in the journal ($E_{K_{new}}(\ell_2)$), and updating the TPM's version of the master key using the $E_{K_{old}}(K_{new})$ value from the journal (recall that, in this scenario, $E_{K_{old}}$ is equal to $E_{K_{TPM}}$, i.e., the key that is currently in the TPM). Finally, HOLEPUNCH clears the journal.

If the journal blocks belong to the same logical update, but the hash of K_{TPM} does not match the key hash in the journal, then HOLEPUNCH was able to finish Steps (1)–(4) before the crash happened. This means that, post-crash, HOLEPUNCH just needs to clear the journal.

3.4.2. File Deletion.

As described in Section 3.3, a file deletion affects the relevant block in the file key table, a few blocks in the on-disk PPRF key[5] (to represent the new puncture), and at most four blocks in the PPRF key tree (the single layer-2 block, and at most three layer-3 blocks in the tree (Figure 1)).

To make file deletion crash-consistent, HOLEPUNCH extends its approach for journaling a master key rotation, following the basic strategy from Section 3.4.1 but writing additional data blocks to the journal. In particular, HOLEPUNCH does the following:

1) Write to the journal $E_{K_{new}}(\ell_2)$, where ℓ_2 is the block containing the second level of the PPRF tree. Since the write is a block in size (see Figure 1), it is atomic.
2) Write to the journal the blocks $E_{K_{new}}(\ell_{3,1})$, $E_{K_{new}}(\ell_{3,2})$, and $E_{K_{new}}(\ell_{3,3})$ where $\ell_{3,i}$ are the modified blocks in the third level of the PPRF tree. Each individual block write is atomic.
3) Write to the journal the values $E_{K_{old}}(K_{new})$ and $H(K_{old})$. Since these values fit within a block, this write is also atomic. HOLEPUNCH has now memorialized its intention to rotate the master key and update

[5] Each puncture causes at most 2 × PPRF_DEPTH new nodes to be added. Our struct pprf_nodes (§5.1) are 33 bytes in size. Given this size, and the fact that our HOLEPUNCH prototype schedules a PPRF key rotation (and thus a tree shrinking) before the PPRF can grow too large, our prototype never sees more than 3 blocks affected by a puncture.

the entire path in the PPRF key tree that leads to the PPRF subkey to update.

4) Journal the data blocks associated with the updated PPRF subkey. Those blocks are encrypted with the relevant leaf node key in the PPRF key tree.
5) Journal the updated file key table block. This block replaces the deleted file's k_f with a new random key. The block also has a new τ_s', and is encrypted by the punctured PPRF evaluated on τ_s'.
6) Update the journal with the block address of each data block that was previously written to the journal in Steps (4) and (5).
7) Journal the new value of the atomic counter. The journal update phase is now complete.
8) Atomically overwrite each of the data blocks (i.e., the layer-2 and layer-3 PPRF tree blocks, the blocks belonging to the new PPRF subkey, and the file key table block).
9) In the TPM's NVRAM, atomically overwrite K_{old} with K_{new}.
10) Clear the journal.

During recovery, HOLEPUNCH checks whether a complete journal update was recorded. If not, HOLEPUNCH clears the journal and terminates recovery. If so, HOLEPUNCH reinitializes its atomic counter using the value in the log, and executes a procedure similar to the one in Section 3.4.1 to determine which data blocks must be updated.

3.4.3. PPRF Key Rotation. To journal PPRF key rotations (§3.3), HOLEPUNCH leverages the magic bytes stored in the header of each file key block (§3.2). In particular, HOLEPUNCH does the following:

1) Pick a new PPRF key $K_{PPRF_{new}}$, and write to the journal $E_K(K_{PPRF_{new}})$ (where K is the master key). A new, unpunctured PPRF key fits inside a single block, so this write is atomic.
2) Read all of the file key blocks into memory. For each one, use the current PPRF key to decrypt the block; reset the tag to be the index of the file key block within the table,, and then re-encrypt the block under the new tag using $K_{PPRF_{new}}$. Write the newly encrypted blocks to disk.
3) Use random data to overwrite the blocks belonging to the current on-disk PPRF tree. The practical effect is that the internal nodes of the key tree are reinitialized to contain new random keys.
4) Using the contents of $K_{PPRF_{new}}$, write the block for the updated on-disk PPRF tree; note that only one block will be written because a fresh, unpunctured PPRF key fits within one block.
5) Reset the tag counter to be equal to the total number of file key blocks in the table. Note that reusing tag values across different PPRF keys is safe; resetting the key decreases the (already small) likelihood of tag wraparound.
6) Clear the journal and schedule the master key rotation from K_{old} to K_{new}.

During recovery, HOLEPUNCH checks whether the journal is empty. If not, HOLEPUNCH reads $E_K(K_{PPRF_{new}})$ from the journal and decrypts it using the master key K stored in the TPM. HOLEPUNCH then reads each key table block and tries to decrypt each one with $K_{PPRF_{new}}$. If the magic number in the decrypted block is correct, the block was correctly re-encrypted with $K_{PPRF_{new}}$ in Step (2), and HOLEPUNCH requires no further action for the block. Otherwise, if the magic number is wrong, then a crash occurred at some point during Step (2); HOLEPUNCH consults the on-disk $K_{PPRF_{original}}$ (which must still exist because Step (3) never completed), using $K_{PPRF_{original}}$ to decrypt the key block, and then using $K_{PPRF_{new}}$ to re-encrypt the block and write it to disk. Note that if all key table blocks decrypt correctly with $K_{PPRF_{new}}$, then HOLEPUNCH completed at least Step (2) before crashing. Regardless, HOLEPUNCH performs Steps (3)–(6) (some of which may have been completed pre-crash but are idempotent), completing the recovery process.

Note that, for the journaling of all three operations mentioned above (master key rotation, file deletion, and PPRF key rotation), our concrete HOLEPUNCH prototype has at most one disk IO in flight at any given time. This simplifies the implementation's journaling code, but is not fundamental to the high-level approach.

4. Security Analysis

Cryptographic erasure reduces the problem of securely deleting a file to the problem of securely deleting a file's encryption key. We assume that a computer has two kinds of storage devices: *eraseable* ones and *non-eraseable* ones. Both kinds of storage allow a logical storage location ℓ to be read and written. However, an eraseable storage device is trusted to actually delete the data in ℓ when a new write to ℓ occurs. In contrast, a non-eraseable device may not actually discard the value stored at ℓ, e.g., because the device internally duplicates a single logical block in multiple physical blocks (§1 and §2.1). A machine's TPM is an example of eraseable storage that is also persistent (because the TPM's NVRAM retains its values across reboots or crashes). We also assume that volatile RAM is eraseable, with the RAM value stored at ℓ being deleted via explicit overwrites to ℓ or when a machine reboots. In contrast, we treat hard drives and SSDs as non-eraseable. Given those definitions, we say that a *file system for secure deletion* is a quadruple of algorithms $(Setup, Read, Write, Delete)$ that manipulate a computer's eraseable (and thus secure) storage S as well as the non-eraseable (and thus insecure) storage I. We model both S and I as indexable variable-length arrays, with I being append-only (to model the conservative assumption that no data is actually deleted) and S allowing in-place updates (to reflect the true eraseable nature of the storage). We also allow both S and I to be extended by allocation on one end. We define the four algorithms as follows:

- $\perp \leftarrow Setup_{S,I}(\lambda, N)$: Accepts a security parameter λ and a capacity N and initializes the internal state (S, I) to store up to N files of size $n \le poly(\lambda)$ with identifiers $t \in \{0,1\}^{\lceil \log(N) \rceil}$.
- $f, \perp \leftarrow Read_{S,I}(i)$: Accepts a file identifier $i \in \{0,1\}^t$ and retrieves from internal state a file f associated with i if it exists and $\perp$ otherwise.
- $i, \perp \leftarrow Write_{S,I}(i, f)$: Accepts a file identifier $i \in \{0,1\}^t$ and file content $f \in \{0,1\}^n$ and writes f to the file with the identifier i that is stored in the internal state. If no such file exists, $\perp$ is returned. Otherwise, return i on success.
- $i, \perp \leftarrow Delete_{S,I}(i)$: Accepts an identifier $i \in \{0,1\}^t$ and updates internal state (S, I) to remove the file f associated with i if it exists and returns $\perp$ otherwise.

We define security using the standard simulation model [9], i.e., we require that the real-world secure deletion functionality emulates a secure idealized functionality. In an ideal world, a file system that achieves secure deletion would use only the erasable medium S for storing information. Thus, we define an ideal secure deletion functionality $\mathcal{F}$ in Figure 2. The ideal functionality stores data only on S, overwriting deleted files with the constant value 0.

```
$Setup_{S,I}(\lambda, N)$
$S[0] = N$
return

$Read_{S,I}(i)$
if !stored($i$) :
    return $\perp$
$f = S[i+1]$
return $f$

$Write_{S,I}(i, f)$
if $i \ge N$ :
    return $\perp$
$S[i+1] = f$
$I$.append(“$wr$”$||i$)
return $i$

$Delete_{S,I}(i)$
if !stored($i$) :
    return $\perp$
$S[i+1] = 0$
$I$.append(“$del$”$||i$)
return $i$
```

Figure 2. The ideal secure deletion functionality $\mathcal{F}$. File data is never written to the insecure (i.e., non-erasable) storage I. The stored function checks whether a file id i is associated with a currently stored file, and the append function writes data to the end of I (see appendix for details).

Note that the ordered sequence of *Write* and *Delete* instructions, along with their associated file identifier values, are encoded into I by the ideal functionality. This is because a real-life instance of I like an SSD might explicitly track file activity, e.g., by writing to a special on-device log for each write or deletion; therefore, there is little hope for hiding this information in the first place. Furthermore, our scheme does not attempt to hide the file identifiers used by the system.

```
Real$^{\Sigma}_{\mathcal{E},Adv}(\lambda)$
$N, m \leftarrow \mathcal{E}(1^\lambda)$
run $Setup(\lambda, N)$ on $\Sigma$
for $i = 1..m$ do
    $\mathcal{E}$ issues $(Write(i, f), Read(i),$ or $Delete(i))$ to $\Sigma$
The internal state $(S, I)$ of $\Sigma$ is given to $Adv$
$v \leftarrow Adv(1^\lambda, S, I)$
$b \leftarrow \mathcal{E}(1^\lambda, v)$
return $b$

Ideal$^{\mathcal{F}}_{\mathcal{E},Sim}(\lambda)$
$N, m \leftarrow \mathcal{E}(1^\lambda)$
run $Setup(\lambda, N)$ on $\mathcal{F}$
for $i = 1..m$ do
    $\mathcal{E}$ issues $(Write(i, f), Read(i),$ or $Delete(i))$ to $\mathcal{F}$
The internal state $(S, I)$ of $\mathcal{F}$ is given to $Sim$
$v \leftarrow Sim(1^\lambda, S, I)$
$b \leftarrow \mathcal{E}(1^\lambda, v)$
return $b$
```

Figure 3. The real and ideal world games used in Definition 4.1 for environment $\mathcal{E}$, adversary Adv (respectively, simulator Sim), and a file system for secure deletion Σ (respectively, ideal functionality $\mathcal{F}$).

Definition 4.1 (Secure Deletion)**.** *We say that a file system* $\Sigma = (Setup, Read, Write, Delete)$ *for storing files of length* $n \le poly(\lambda)$ *achieves* secure deletion *if for all probabilistic polynomial time (PPT) adversaries Adv, there exists a PPT simulator Sim such that for all PPT environments* $\mathcal{E}$*:*

$$|\Pr[Real^{\Sigma}_{\mathcal{E},Adv}(\lambda) = 1] - \Pr[Ideal^{\mathcal{F}}_{\mathcal{E},Sim}(\lambda) = 1]| \le negl(\lambda)$$

where the games $Real^{\Sigma}_{\mathcal{E},Adv}$ *and* $Ideal^{\mathcal{F}}_{\mathcal{E},Sim}$ *are defined in Figure 3.*

In the appendix, we prove that HOLEPUNCH achieves secure deletion for file data.

5. Implementation

Given the HOLEPUNCH design from Section 3, we now describe the most interesting aspects of our concrete HOLEPUNCH implementation. We first explain the binary PPRF representation that HOLEPUNCH uses, and describe how HOLEPUNCH evaluates and modifies that PPRF (§5.1). We then explain how HOLEPUNCH initializes a block device (§5.2) for use by subsequent file system activity. We finally describe the implementation and optimization of HOLEPUNCH's file accesses (§5.3), file deletions (§5.4), and PPRF key refreshes (§5.5).

5.1. The HOLEPUNCH PPRF

Our PPRF implementation is inspired by the tree-based PRF construction of Goldreich, Goldwasser, and Micali [10]. In that construction, $G : \{0,1\}^{\lambda} \rightarrow \{0,1\}^{2\lambda}$ is a length-doubling pseudorandom generator. More specifically, $G(s) = G_0(s)||G_1(s)$ with $G_0(s)$ and $G_1(s)$ representing the first and second halves of the output of $G(s)$. Goldreich et al. then define a pseudorandom function family $F_k(\cdot)$ as:

$$F_k(x) = G_{x_{m(\lambda)}}(G_{x_{m(\lambda)-1}}(\ldots(G_{x_1}(k))))$$

where $x_1x_2\ldots x_{m(\lambda)}$ is the binary representation of the input x, m is a polynomial, and $k \leftarrow_R \{0,1\}^{\lambda}$ is a key chosen uniformly at random. This construction forms a tree for PRF inputs of length $m(\lambda)$, such that the tree's leaves are the outputs of the PRF. Although the tree has exponentially many outputs, we can efficiently evaluate any input to the PRF using only $m(\lambda)$ evaluations of the pseudorandom generator G. Such a PRF tree can be used to build a puncturable PRF [3], [4], [25]. Given a tree-based PRF F with key k and a point x that we wish to puncture, let P_x be the set of nodes along the path from the xth leaf to the root of the PRF tree. We can set the punctured key $k^* = \{N_x\}$, where N_x is the set of neighboring nodes for each of the nodes in P_x. As an example, Figure 4 shows a tree-based PRF F with key k and four leaves. After puncturing the value $x = 2$ (represented by the dotted leaf k_{10}), a punctured key $k^* = \{k_0, k_{11}\}$ is produced. The new key k^* corresponds to the neighboring nodes along the path from k_{10} to k. Using F_{k^*}, one can still evaluate all of the points in the original PRF tree except for the point $x = 2$.

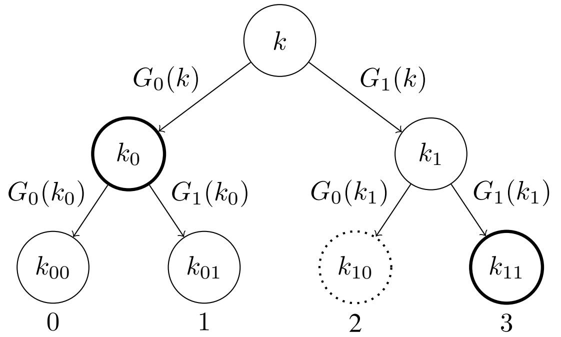

Figure 4. Example of a punctured PRF that uses the construction of Goldreich et al. [10]. The nodes $\{k_0, k_{1,1}\}$ represent the new punctured key k^* after puncturing the leaf k_{10} (the dotted node) associated with the point $x = 2$.

HOLEPUNCH represents the PPRF as an array of `struct pprf_nodes` (Figure 5). Each structure is a tagged union [14] that represents an inner node, an unpunctured leaf node, or a punctured leaf node. For an inner node, the `.next.il` and `.next.ir` fields are array indices representing the node's left and right child, respectively. An unpunctured leaf node contains a single 32-byte key while a punctured leaf node contains all zeroes.

```
struct pprf_node {
     union {
          struct {
               u32 il;
               u32 ir;
          } next;
          u8 key[32];
     } v;
     u8 type;
};
```

Figure 5. The `pprf_node` structure.

Evaluating the PPRF: Evaluating the PPRF at a point τ requires HOLEPUNCH to:

1) find the `pprf_node` leaf that evaluates τ, and then
2) use the leaf node's subkey value as the initial input to the length-doubling PRG of the GGM86 construction.

Starting at the `pprf_node` at index 0, HOLEPUNCH traces a path down the tree, following the left path at step i if the i-th bit in the binary representation of τ is 0, and following the right path otherwise. Let `search_depth` be the depth of the discovered leaf node that HOLEPUNCH eventually finds, and let `tree_height` be $\log(N)$; upon finding the (presumably unpunctured) leaf node, HOLEPUNCH invokes a length-doubling PRG `tree_height` − `search_depth` times, using the leaf node's subkey as the PRG seed and, during the i-th PRG iteration, taking the left or right half of the PRG output according to the i-th bit of τ's binary representation. The final PRG output represents the output of the PPRF on τ.

HOLEPUNCH uses the AES block cipher as a building block for the pseudorandom generator. Recall that, for any pseudorandom function $F : \{0,1\}^{\lambda} \times \{0,1\}^n \rightarrow \{0,1\}^m$, we can define a pseudorandom generator $G : \{0,1\}^{\lambda} \rightarrow \{0,1\}^{\ell m}$ as $G(s) = F_s(0)||F_s(1)||\ldots||F_s(\ell)$. Using this fact, HOLEPUNCH constructs a length-doubling PRG $G : \{0,1\}^{256} \rightarrow \{0,1\}^{512}$ as follows:

$$G(s) = \texttt{AES}(s,0)||\texttt{AES}(s,1)||\texttt{AES}(s,2)||\texttt{AES}(s,3)$$

with AES used in ECB mode, s being a 256 bit key, and the inputs to each AES block being 128 bits in length.

Puncturing the PPRF: Puncturing the PPRF at a point τ requires HOLEPUNCH to:

1) find the leaf `pprf_node` ℓ that evaluates τ;
2) allocate space for two children `pprf_node` structures at the end of the PPRF array;
3) zero out ℓ's `.key` field, set the `.type` to indicate that it is an internal node, and set `.next.il` and `.next.ir` to point to its children indexes that were added in Step 2;
4) proceed along the evaluation path from ℓ and repeat Step 3 for every internal node along the path;
5) for every neighboring node along the evaluation path from ℓ to τ, set the `.key` field to equal its correspond-

ing half of the PRG output in the PPRF tree, and set the `.type` to indicate that it is a leaf node;

6) set the `.type` of leaf node at the end of the evaluation path to indicate that it is punctured, zeroing out all of its fields.

Given the procedure above, the PPRF key grows by at most $33 \times 2 \times$ `PPRF_DEPTH` bytes with each puncture. Specifically, in Steps 2 and 3, we add two child `pprf_nodes` where one corresponds to an internal node along the evaluation path, and the other corresponds to a neighboring node along the evaluation path. We also modify the single `pprf_node` ℓ that evaluated τ by zeroing out its `.key` field and marking it as an internal node. Therefore, we are only expanding the key by adding new internal nodes and new neighbors, and modifying at most one node in the existing PPRF key state. The PPRF key tree thus enables efficient updating of the on-disk representation of the PPRF key.

Periodic refreshes of the PPRF key (§5.5) allow Holepunch to prevent the PPRF from growing unboundedly. Holepunch journals PPRF key rotations to make them crash-consistent (§3.4.3).

5.2. Initialization

When initializing a block device, Holepunch reserves the first portion of the device to store various cryptographic keys, namely,

1) the full PPRF key,
2) the PPRF key tree which Holepunch uses to encrypt and efficiently update the on-disk PPRF key,
3) the file key table, and
4) the Holepunch metadata header. The header contains the NVRAM index where the master key is located within the TPM. The header also contains the user password salt that is passed to the KDF which generates the key that encrypts the master key.

The amount of space required for the full PPRF key and the PPRF key tree is governed by a configurable refresh interval which specifies the maximum number of punctures that the Holepunch device can handle before Holepunch initiates a PPRF key refresh. For a 1TB hard disk, the refresh involves reading, re-encrypting, and writing a $\sim 2GB$ file key table; on a commodity SSD, the total refresh operation with its large, sequential IOs will need just a handful of seconds to complete. Holepunch derives the space requirements for the file key table by asking the file system how many `inodes` the disk will contain; for example, ext4 allocates an `inode` for every 16KB of available disk space.

During initialization, Holepunch creates $\lceil \frac{\text{total inodes}}{127} \rceil$. blocks in the file key table, initializing the encrypted keys to random bits.[6] Holepunch sets the tag of each key table block to be the block number; the atomic tag counter is set to the total number of file key table blocks. Holepunch then generates a random PPRF key and master key, and initializes the PPRF key tree with random bits.

5.3. File Access

Holepunch is a block device driver that sits between a file system and the underlying storage device. For every block I/O operation that is generated by the file system, Holepunch inspects the Linux `bio` object associated with the operation, and checks whether the associated `inode` corresponds to a file. If so, and the IO operation is a file read or write, Holepunch must do the following:

1) use the `bio` object to walk kernel data structures and extract the relevant `inode` number i;
2) read the associated key table block s, discovering the τ_s for the block and the encrypted per-file keys;
3) evaluate the PPRF on τ, generating k_{τ_s};
4) use k_{τ_s} to decrypt the key table block and obtain the file key associated with `inode` i;
5) use the key to decrypt an incoming file block (for a read) or encrypt an outgoing file block (for a write). We use AES-CBC with ESSIV for encryption and decryption.

Step 2 adds an extra disk IO, but Holepunch caches recently-accessed key table blocks to eliminate the penalty of actually reading such blocks from disk during accesses to hot `inodes`. Also note that our Holepunch implementation caches hot file keys; during the handling of a particular IO, hitting in the file key cache will avoid the extra disk IO and the need to evaluate the PPRF. Holepunch is able to cache many more file keys than Eraser due to the lower memory footprint of Holepunch's PPRF relative to Eraser's file key tree.

File reads and writes do not require Holepunch-level journaling. However, a higher-level file system like ext4 may journal written file data to provide file-system-level crash consistency for that data.

5.4. Securely Deleting a File

Like Eraser, Holepunch intercepts file deletion events using Linux's kprobe interface [1]. In particular, Holepunch interposes on the entry point of Linux's `vfs_unlink` function. When the kernel tries to enter that function, Holepunch inspects the system call arguments in the CPU registers and uses those arguments to retrieve a pointer to the `inode`-to-delete. Holepunch checks whether that `inode` corresponds to a file in the Holepunch partition. If so, Holepunch will:

1) read the key table block associated with the relevant `inode` number i, discovering the relevant τ,
2) evaluating the PPRF on τ to generate k_τ and using k_τ to decrypt the key table block in memory,
3) puncture the PPRF key at τ,
4) generate a fresh, random key to associate with the `inode`, and update the corresponding entry in the in-memory key table block,

[6] Recall that a block is 4096 bytes and Holepunch stores a 32-byte tag and 127 32-byte file keys in each key table block. Holepunch requires a key table block per 127 `inodes`, and thus needs $\lceil \frac{\text{total inodes}}{127} \rceil$ table entries.

5) update the in-memory key table block's τ to be the current value of the atomic tag counter (incrementing the atomic tag counter as a side effect),
6) re-encrypt the key table block using the output of the PPRF on the new τ, and finally
7) write the new (i.e., punctured) PPRF key, the atomic tag counter, and the encrypted key table block to disk.

Note that all of these steps occur before the normal `vfs_unlink` code executes on `inode` i. If the machine crashes after these steps occur, but before the file system itself commits the delete operation through file-system level mechanisms (e.g., file-system-level journaling), the file system will believe that the `inode` i is still valid.[7] However, attempts to read that a block from that file will result in HOLEPUNCH decrypting the storage block with the `inode`'s new key; thus, old file data will be unreadable, in accordance with the user's desire to securely delete the file. Attempts to overwrite old file data will succeed, and subsequent attempts to retrieve that newly-written data will result in correctly-decrypted reads. These semantics are aligned with those of traditional journaling file systems that always ensure metadata consistency, but may return junk *data* after a crash [21].

We now provide more detail about which happens when HOLEPUNCH must puncture a point x from the input domain of the PPRF (§3.4.2). Let S_x be the on-disk block containing the `pprf_node` that evaluates x, and let S_{new} be the blocks containing the newly added `pprf_nodes` that correspond to the nodes along the evaluation path of x (and their neighbors). HOLEPUNCH must update S_x such that the `pprf_node` that evaluates x is punctured (by zeroing out the key) and the old on-disk copy of S_x is unrecoverable. Additionally, HOLEPUNCH must write S_{new} to the end of the on-disk array that represents the PPRF key (§3.4.2). Note that S_{new} will contain at most two disk blocks (§3.4.2).

To accomplish these tasks, HOLEPUNCH refreshes the keys in the ancestor layers of the PPRF tree along the path from S_x to the root. Recall that the root of the tree is the master key K_{master} and so this key must also be refreshed. After rotating these keys, HOLEPUNCH then re-encrypts S_x (after zeroing out the punctured node's `.key` field and setting its `.type` to indicate that it is punctured) and re-encrypts its ancestors in the second and third layers of the PPRF tree. HOLEPUNCH then writes these three blocks to disk using three IOs. HOLEPUNCH must also encrypt and write S_{new} to disk using at most two additional writes in the case that S_{new} spans across a block boundary; S_{new} will never span two or more block boundaries for any realistic PPRF configurations. Thus, ignoring the journal overhead, every puncture operation can be persisted to disk with at most five disk IOs independent of the size of the on-disk PPRF key. HOLEPUNCH only requires four IOs if S_{new} fits inside a single disk block.

The size of the PPRF tree is small. For example, consider a 5TB storage device configured with a HOLEPUNCH refresh interval of $50,000$ punctures; such a configuration would incur roughly one PPRF key refresh per day for a typical file system workload on a desktop [24]. The PPRF key itself (i.e., the data that is encrypted by the keys in the leaves of the PPRF tree) would have a maximum size of $\sim$ 73MB.[8] Using an 128-ary PPRF tree, the PPRF tree would need to store $\sim$ 20K keys in order to encrypt the entire 73MB of the PPRF key.[9] Since each key consumes 32 bytes, the PRRF tree will only consume only $\sim$ 625KB of space.

5.5. Refreshing the PPRF Key

The size of the PPRF key is determined by (1) the size of the input domain, (2) the security parameter λ, and (3) the current number of punctured elements. The key size has a logarithmic dependency on the size of the input domain, and linear dependencies on λ and the number of punctured elements. The number of punctured elements grows whenever a file is deleted. To keep the PPRF key from growing unboundedly, HOLEPUNCH periodically refreshes the PPRF key, resetting its size to λ bits. A refresh consists of generating a new PPRF key, re-encrypting all of the on-disk key table blocks, and writing the master key to disk. Section 3.4.3 describes how HOLEPUNCH performs these updates in a crash-consistent way.

By default, our HOLEPUNCH implementation limits the maximum size of the PPRF key to 50MB, automatically refreshing the key if it would exceed this size. With a $\lambda = 256$ bit security parameter and an $N = 2^{21}$ input domain, the file system can generate approximately $\frac{50 \times 2^2 0}{33 \times 21 \times 2} \approx 37,000$ deletions before the PPRF key must be refreshed. However, in practice, our prototype implements an optimization called *deletion batching* that allows HOLEPUNCH to handle many more deletions before a key refresh is triggered. With deletion batching, a deletion is recorded in the HOLEPUNCH journal immediately, but can wait to issue the in-place updates to HOLEPUNCH structures made in Steps 8 and 9 of Section 3.4.2. N file deletions that target `inodes` covered by the same file key block only require a single PPRF puncture, dramatically reducing HOLEPUNCH's IO overheads and allowing the PPRF to absorb more punctures before needing a refresh. Applications like web browsers, compilers, and databases often repeatedly create, access, and then delete many files [7], [24], so deletion batching is helpful for common workloads. Note that the batching interval can be adjusted based on the setting. For example, the average user may prefer a longer batch interval to reduce

[7] Recall that HOLEPUNCH does not encrypt `inodes`, so HOLEPUNCH-level file deletion will not somehow re-encrypt an `inode` and make it unreadable from the perspective of the file system.

[8] To derive this number, first note that `PPRF_DEPTH` $= \lceil \log(2 \times \frac{5 \times 2^{26} \text{ inodes}}{127}) \rceil$. Then consider that the key will grow by at most (`PPRF_DEPTH` $\times$ sizeof(`pprf_node`) $\times 2$) bytes per deletion. Therefore, for $50K$ deletions, we need $50,000 \times 2 \times 23 \times 33$ bytes $\approx$ 73MB to store the PPRF key.

[9] The reason is that `num_leaves` $= \frac{73 \times 2^{20} \text{ bytes of PPRF key}}{4096 \text{ bytes per block}}$. So, the third layer of the tree will require `num_leaves`$/128$ keys, and the second layer will require `num_leaves`$/(128 \times 128)$ keys.

the frequency of PPRF key refreshes, while shorter batch intervals can be used in more security critical settings.

6. Evaluation

Prior sections demonstrated why HOLEPUNCH provides secure deletion (§4) and crash consistency (§3.4). In this section, we show that HOLEPUNCH enjoys these benefits while exhibiting similar IO performance as ERASER (§6.1) but more scalable memory consumption (§6.2).

We evaluated HOLEPUNCH on a Linux machine (kernel v4.7) with an 8-core AMD Ryzen 7 4700G CPU, 32GB of RAM, and a 256 GB SSD. The machine layered the EXT4 file system atop HOLEPUNCH's block driver.

6.1. Performance

We used the BONNIE++ benchmarking tool [18] for measuring IO performance. We compared HOLEPUNCH to ERASER (a secure deletion system with no crash consistency but none of HOLEPUNCH's journaling overhead) and DM-CRYPT (Linux's native encrypted block device that implements full-disk encryption but not secure deletion). The three systems showed neglible performance differences for workloads that read and wrote large files because these workloads did not stress the deletion code paths. So, in Table 1, we focus on tests involving many small files (each between 256 and 512 bytes in size). The first set of BONNIE++ experiments sequentially created, read the metadata of, and then deleted 1,048,576 small files, whereas the second set of BONNIE++ experiments *randomly* created, stat'ed, and deleted 1,048,576 small files. We also ran a simple script which sequentially `cat`'ed 10K small files (each containing the string "test") and then `rm`'ed each of these files. Finally, we examined the performance of a kernel compilation.

As shown in Table 1, HOLEPUNCH was at worst 3.87% slower than ERASER for all but the BONNIE++ random delete test. The reason was that random deletes rarely benefited from HOLEPUNCH's deletion batching optimization (§5.5), forcing HOLEPUNCH to issue IOs for a PPRF puncture during most deletes; as a result, HOLEPUNCH was 12.7% slower than ERASER for this experiment. However, the kernel compilation test showed that for more realistic workloads, HOLEPUNCH could indeed leverage deletion batching to offer very similar performance to ERASER.

We also note that the BONNIE++ experiments put more pressure on the kernel's in-memory buffer cache. The increased pressure led to more swapping between the disk and the buffer cache, increasing contention for the disk's finite IO bandwidth. The `cat` and `make` benchmarks were comparatively more CPU-intensive (and thus less sensitive to how the three approaches for encrypted storage used the disk's bandwidth).

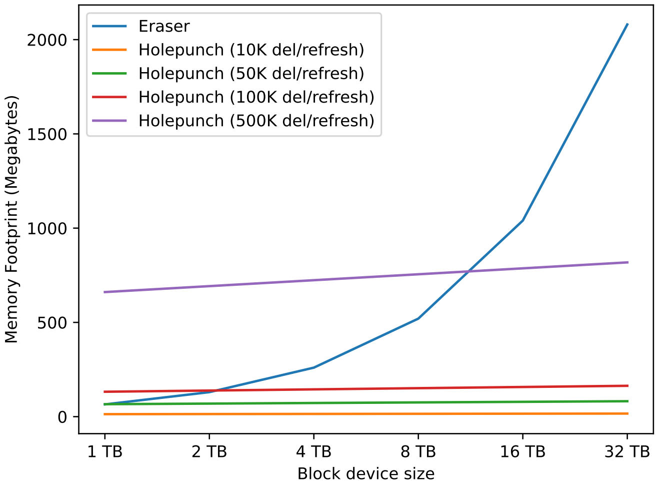

Figure 6. Baseline memory consumption of ERASER compared to various HOLEPUNCH configurations.

6.2. Memory Usage

In Figure 6, we compare the baseline memory footprint of HOLEPUNCH and ERASER. Recall that ERASER must store the upper layers of its FKT in memory or pay IO access costs that are logarithmic in the number of `inodes`. The FKT's upper layers scale linearly with the size of the number of `inodes`. Assuming one `inode` for every 16KB of storage space, a 1 terabyte storage device will have $2^{40}/2^{14} = 2^{26}$ `inodes`. Since each ERASER FKT entry is padded to 64 bytes and encrypts a block of 64 nodes, the total memory footprint of a 1 TB drive is approximately 65 MB.

Each HOLEPUNCH file key block contains 127 keys since we have a 32-byte tag and can fit at most 127 keys into the remaining space. Therefore, a minimum input domain size of 2^{20} is required to evaluate tags on $\lceil 2^{26}/127 \rceil$ file key blocks. HOLEPUNCH doubles this number to 2^{21} since it needs extra tags as new files are created and old ones are deleted. So, HOLEPUNCH's PPRF tree has a depth of 21 for a 1TB storage device. In the worst case, the PPRF tree grows by $2 \times \texttt{PPRF\_DEPTH} \times 33$ bytes, where 33 bytes is the size of each PPRF node as described in Section 5.1. Thus, for a 1TB drive, HOLEPUNCH generates a worst case of approximately 1.4 KB of in-memory key growth per deletion. As shown in Figure 6, HOLEPUNCH's memory consumption scales much better than that of ERASER as the size of the storage device grows. While the reduced memory pressure is more noticeable on larger file systems, we note that modern laptops already support 4TB drives, and we anticipate a continued increase in storage sizes.

Unsurprisingly, for a given disk size, HOLEPUNCH's worse-case memory overheads get larger for less aggressive schedules of PPRF rotation. With less aggressive scheduling, the in-memory PPRF representation is allowed to grow larger before it is pruned.

TABLE 1. Performance Benchmark of dm-crypt vs. Eraser vs. Holepunch

	dm-crypt		**Eraser**		**Holepunch**	
Bonnie++ Experiments ($n = 100$)	Avg. (files/sec)	Std. Dev.	Avg. (files/sec)	Std. Dev.	Avg. (files/sec)	Std. Dev.
Sequential Create	61605	48	57759	957	56785	697
Sequential Stat	95997	1147	87748	1238	84363	1987
Sequential Delete	62454	785	46370	888	45716	669
Random Create	61974	298	58104	982	56993	692
Random Stat	98583	1243	88575	2195	86965	2876
Random Delete	49755	502	40345	725	35210	810
Timed Experiments ($n = 5$)	Avg. (sec)	Std. Dev.	Avg. (sec)	Std. Dev.	Avg. (sec)	Std. Dev.
`cat` 10K files	12.76	0.07	12.64	0.09	12.15	0.07
`rm` 10K files	12.55	0.02	12.8	0.06	12.64	0.09
`make` Linux Kernel	3276.22	19.26	3246.71	3.27	3255.52	2.41

6.3. Crash-Consistency

We experimentally verified the fault tolerance of Holepunch. We used a debugger to hook into the kernel and set breakpoints at important code locations relevant to key synchronization (e.g., PPRF key refreshes). After pausing execution at such a breakpoint, we would forcibly power off the system and verify that the file system was consistent post-reboot.

7. Discussion

Metadata: Holepunch provides secure deletion for file data, but information can leak through metadata as well. For example, in an ext4 file system, a directory is essentially a file whose data blocks contain (`file_name`, `inode_number`) pairs. Securely deleting the data belonging to file f does not securely delete f's metadata in a parent directory's data block. Extending Holepunch to securely delete such metadata is left for future work.

Usability: In the file systems used by popular consumer OSes like Windows and macOS, deletion is typically reversible, at least to some extent. For example, on Windows, a deleted file goes to a "trash can" and can be recovered until the trash can is explicitly emptied by the user (or implicitly emptied by the system if disk space becomes low). Deletion undo is helpful if a user accidentally deletes a file, but it incurs an obvious risk of information leakage (§2). As a middle ground, a Holepunch-style system could incorporate a grace period in which a file deletion would not become a secure deletion until a grace period had passed.

Holepunch inherits Eraser's trigger for initiating a secure deletion: Holepunch securely deletes file f only after observing a system call that explicitly removes f's `inode`. With this approach, overwriting a file, e.g., using `mv fileA fileB` to overwrite the contents of `fileB`, will *not* securely delete `fileB`. Extending Holepunch to support such operations is left for future work.

8. Conclusion

Secure deletion provides users with confidence that a removed file is actually unrecoverable. However, implementing secure deletion in software is difficult because modern storage devices may arbitrarily manipulate the physical blocks that underlie the logical block interface. Cryptographic erasure avoids the problem by only sending encrypted blocks to disk and discarding the keys for deleted blocks. However, for a cryptographic erasure scheme to be *practical*, it must guarantee crash consistency. Holepunch provides such a scheme. Holepunch uses puncturable pseudorandom functions to efficiently manage key data (both in-memory and on-disk), and uses a journaling scheme to provide crash-consistent atomic updates of cryptographic material and the file data protected by those keys. Experiments show that Holepunch provides similar IO performance to the prior state-of-the-art system for cryptographic erasure, while having more scalable consumption of RAM as disk size increases. Thus, we believe that Holepunch is the first practical system for secure deletion at a file granularity.

Acknowledgements

We thank the anonymous reviewers for their feedback. We also thank Salil Vadhan for many insightful discussions.

This work was supported in part by Cooperative Agreement CB20ADR0160001 with the Census Bureau, and in part by Salil Vadhan's Simons Investigator Award.

References

[1] Kernel probes (kprobes) - the linux kernel documentation. https://docs.kernel.org/trace/kprobes.html.

[2] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*. https://pages.cs.wisc.edu/~remzi/OSTEP/, 2020, ch. Chapter 42: Crash Consistency–FSCK and Journaling.

[3] Boneh, D., and Waters, B. Constrained pseudorandom functions and their applications. In *International conference on the theory and application of cryptology and information security* (2013), Springer, pp. 280–300.

[4] Boyle, E., Goldwasser, S., and Ivan, I. Functional signatures and pseudorandom functions. In *International workshop on public key cryptography* (2014), Springer, pp. 501–519.

[5] CARD, R. Design and implementation of the second extended filesystem. In *Proc. First Dutch International Symposium on Linux, Dec. 1994* (1994).

[6] CRESCENZO, G. D., FERGUSON, N., IMPAGLIAZZO, R., AND JAKOBSSON, M. How to forget a secret. In *Annual Symposium on Theoretical Aspects of Computer Science* (1999), Springer, pp. 500–509.

[7] ELLARD, D., AND LEDLIE, J. Passive {NFS} tracing of email and research workloads. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)* (2003).

[8] EUROPEAN UNION. General Data Protection Regulation (GDPR). https://gdpr-info.eu/, 2016.

[9] GOLDREICH, O. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[10] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to construct random functions. *Journal of the ACM (JACM) 33*, 4 (1986), 792–807.

[11] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of ASPLOS* (2009), pp. 229–240.

[12] HASSON, J. Va toughens security after pc disposal blunders. *Federal Computer Week 26* (2002).

[13] IEEE 1149.1 WORKING GROUP. IEEE Std. 1149.1: Standard Test Access Port and Boundary-Scan Architecture. https://grouper.ieee.org/groups/1149/1/, 2023.

[14] IERUSALIMSCHY, R., DE FIGUEIREDO, L. H., AND CELES FILHO, W. The implementation of lua 5.0. *J. Univers. Comput. Sci. 11*, 7 (2005), 1159–1176.

[15] KIAYIAS, A., PAPADOPOULOS, S., TRIANDOPOULOS, N., AND ZACHARIAS, T. Delegatable pseudorandom functions and applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 669–684.

[16] KORNBLUM, J. D. Implementing BitLocker Drive Encryption for Forensic Analysis. *International Journal of Digital Forensics and Incident Response 5*, 3–4 (2009), 75–84.

[17] MEIJER, C., AND VAN GASTEL, B. Self-encrypting Deception: Weaknesses in the Encryption of Solid State Drives. In *Proceedings of the IEEE Symposium on Security and Privacy* (2019), pp. 72–87.

[18] MER-TOOLS. Mer-tools/bonnie: Benchmark suite.

[19] MILAN BROZ. dm-crypt: Linux kernel device-mapper crypto target. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt, 2023.

[20] ONARLIOGLU, K., ROBERTSON, W., AND KIRDA, E. Eraser: Your data won't be back. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (2018), IEEE, pp. 153–166.

[21] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track* (2005), vol. 194, pp. 196–215.

[22] REARDON, J., BASIN, D., AND CAPKUN, S. Sok: Secure data deletion. In *2013 IEEE symposium on security and privacy* (2013), IEEE, pp. 301–315.

[23] REARDON, J., RITZDORF, H., BASIN, D., AND CAPKUN, S. Secure data deletion from persistent media. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 271–284.

[24] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)* (2000).

[25] SAHAI, A., AND WATERS, B. How to use indistinguishability obfuscation: deniable encryption, and more. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing* (2014), pp. 475–484.

[26] STATE OF CALIFORNIA. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa, 2018.

[27] STEVENS, C. E. Technical Committee T13 AT Attachment: Working Drafts. https://www.t13.org/project-working-drafts?page=1, 2020.

[28] TRUSTED COMPUTING GROUP. TCG Storage Security Subsystem Class: Opal. https://trustedcomputinggroup.org/wp-content/uploads/TCG-Storage-Opal-SSC-v2p02-r1p0_pub24jan2022.pdf, January 24, 2022.

[29] TRUSTED COMPUTING GROUP. TPM 2.0 Library. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2023.

[30] VILLANO, M. Hard-drive magic: Making data disappear forever. *New York Times 2* (2002).

Appendix A. Security Proof

We provide the full API for the real functionality Σ used by `Holepunch` in Figures 7 and 8 and the API for ideal functionality $\mathcal{F}$ in Figure 2. Both functionalities include subroutines and constants that we describe here. Additionally, Σ makes use of an IND-CPA secure symmetric encryption scheme (E, D), a puncturable pseudorandom function F, and a cryptographic hash function H modeled as a random oracle.

Subroutines and constants.

- `RFSH`: a constant that specifies how many deletions can be applied before a PPRF refresh occurs.
- `K`: the constant 0 used to indicate the index in S storing the master key K_{master}.
- `PPRF`: the constant 1 used to indicate the index in S storing the PPRF key K_{PPRF}.
- `META`: the constant 2 used to indicate the index in S storing metadata about the file system capacity.
- `CTR`: the constant 3 used to indicate the index in S storing the atomic tag counter.
- `KTREE`: the constant 4 used to indicate the index in S storing the PPRF key tree data structure.
- `DELS`: the constant 5 used to indicate the index in S storing the current number of punctures on the current PPRF key.
- `newKeyTree`(K_{master}, K_{PPRF}): creates the PPRF key tree data structure that stores K_{PPRF} encrypted under the IND-CPA secure scheme (E, D) and using K_{master} as the root.
- `getFile`(i): accepts a file identifier and retrieves from I the encrypted file associated with i if it exists, and $\perp$ otherwise.
- `randbits`(λ): returns λ uniformly random bits
- `getKeyBlock`(i): accepts a file identifier i and returns the tuple (es, j, τ) corresponding to an encrypted file key block es, an index j into the file key block that points to the location of the file key associated with i, and a tag τ assigned to the block.
- `assignTag`(es, τ): accepts an encrypted file key block es and a tag τ and assigns the tag τ to the block (via writing to the header of es)

```
$Setup(\lambda, N)$
$S[\texttt{K}] = \texttt{randbits}(\lambda)$
$S[\texttt{PPRF}] = \texttt{randbits}(\lambda)$
$num\_blocks = \lceil N/127 \rceil$
$S[\texttt{META}] = (N, num\_blocks)$
$S[\texttt{CTR}] = 0$
$S[\texttt{KTREE}] = \texttt{newKeyTree}(S[\texttt{K}], S[\texttt{PPRF}])$
$I.\text{append}(\text{“}kt\text{”}||S[\texttt{KTREE}])$
$\textbf{for } s = 1 \ldots num\_blocks :$
  $ds = \texttt{randbits}(127 \cdot \lambda)$
  $es = F(S[\texttt{PPRF}], S[\texttt{CTR}])$
  $\texttt{assignTag}(es, S[\texttt{CTR}])$
  $I.\text{append}(\text{“}blk\text{”}||S[\texttt{CTR}]||es)$
  $S[\texttt{CTR}] = S[\texttt{CTR}] + 1$
$\textbf{return}$
```

```
$Write(i, f)$
$(N, \_) = S[\texttt{META}]$
$\textbf{if } i \geq N :$
  $\textbf{return } \perp$
$(es, j, \tau) = \texttt{getKeyBlock}(i)$
$k_\tau = F(S[\texttt{PPRF}], \tau)$
$ds = D(k_\tau, es)$
$k_i = ds[j]$
$c = E(k_i, f)$
$I.\text{append}(\text{“}file\text{”}||i||c)$
$\textbf{return}$
```

```
$Read(i)$
$c = \texttt{getFile}(i)$
$\textbf{if } c == \perp \textbf{ then}$
  $\textbf{return } \perp$
$(es, j, \tau) = \texttt{getKeyBlock}(i)$
$k_\tau = F(S[\texttt{PPRF}], \tau)$
$ds = D(k_\tau, es)$
$k_i = ds[j]$
$f = D(k_i, c)$
$\textbf{return } f$
```

```
$Delete(i)$
$c = \texttt{getFile}(i)$
$\textbf{if } c == \perp \textbf{ then}$
  $\textbf{return } \perp$
$(es, j, \tau) = \texttt{getKeyBlock}(i)$
$k_\tau = F(S[\texttt{PPRF}], \tau)$
$ds = D(k_\tau, es)$
$k'_f = \texttt{randbits}(\lambda)$
$ds[j] = k'_f$
$k_{\tau'} = F(S[\texttt{PPRF}], S[\texttt{CTR}])$
$es' = E(k_{\tau'}, ds)$
$\texttt{assignTag}(es', \tau')$
$S[\texttt{CTR}] = S[\texttt{CTR}] + 1$
$S[\texttt{PPRF}] = \texttt{puncture}(S[\texttt{PPRF}], \tau)$
$K_{new} = \texttt{randbits}(\lambda)$
$keyTree = \texttt{newKeyTree}(K_{new}, S[\texttt{PPRF}])$
$JournalDelete(keyTree, es', S[\texttt{K}])$
$I.\text{append}(\text{“}kt\text{”}||keyTree)$
$I.\text{append}(\text{“}ctr\text{”}||S[\texttt{CTR}])$
$I.\text{append}(\text{“}blk\text{”}||S[3]||es')$
$S[\texttt{K}] = K_{new}$
$S[\texttt{DELS}] = S[\texttt{DELS}] + 1$
$\textbf{if } S[\texttt{DELS}] > \texttt{RFSH} :$
  $Refresh()$
$\textbf{return } i$
```

Figure 7. The real-world secure deletion scheme Σ used internally by `Holepunch` to securely delete file data while ensuring crash consistency. We use the common memory locations $S[\texttt{K}]$ to represent the TPM slot that stores the master key, $S[\texttt{PPRF}]$ to represent the area in RAM that stores the PPRF, $S[\texttt{META}]$ to represent the area in RAM that stores metadata such as the file system capacity, $S[\texttt{CTR}]$ to represent the area in RAM that stores the tag counter, and $S[\texttt{KTREE}]$ to represent the area in RAM that stores the current state of the PPRF key tree. We show the journaling logic of the API along with the procedure for refreshing a PPRF key in Figure 8.

- $\texttt{puncture}(K_{PPRF}, \tau)$: accepts a PPRF key and a tag τ and applies the PPRF's *puncture* operation to obtain a new key K^*_{PPRF} punctured at the point τ.
- $\texttt{stored}(i)$ accepts a file identifier i and checks if i is associated with a currently stored file. This works by simply inspecting the contents of I which fully defines the set of identifiers associated with currently stored files.
- $I.\text{append}(data)$ writes *data* to the end of the insecure (append-only) storage I.

```
$JournalRotate(K_{new}, keyTree)$
$I.\text{append}(\text{"}jrnl\_kt\text{"}||keyTree)$
$I.\text{append}(\text{"}jrnl\_khash\text{"}||H(S[\texttt{K}]))$
$I.\text{append}(\text{"}jrnl\_key\text{"}||E(S[\texttt{K}], K_{new})$
$\textbf{return}$

$JournalDelete(keyTree, es', K_{new})$
$I.\text{append}(\text{"}jrnl\_kt\text{"}||keyTree)$
$I.\text{append}(\text{"}jrnl\_khash\text{"}||H(S[\texttt{K}]))$
$I.\text{append}(\text{"}jrnl\_key\text{"}||E(S[\texttt{K}], K_{new})$
$I.\text{append}(\text{"}jrnl\_ctr\text{"}||S[\texttt{CTR}])$
$I.\text{append}(\text{"}jrnl\_blk\text{"}||S[\texttt{CTR}]||es')$
$\textbf{return}$

$JournalRefresh(K'_{PPRF})$
$I.\text{append}(\text{"}jrnl\_rfsh\text{"}||E(S[\texttt{K}], K'_{PPRF}))$
$\textbf{return}$
```

```
$Refresh()$
$K_{new} = \texttt{randbits}(\lambda)$
$K'_{PPRF} = \texttt{randbits}(\lambda)$
$JournalRefresh(K'_{PPRF})$
$(N, num\_blocks) = S[\texttt{META}]$
$\textbf{for } t = 1 \ldots \texttt{num\_blocks}:$
    $(es, j, \tau) = \texttt{getKeyBlock}(t)$
    $k_\tau = F(S[\texttt{PPRF}], \tau)$
    $ds = D(k_\tau, es)$
    $k_t = F(K'_{PPRF}, t)$
    $es' = E(k_t, ds)$
    $\texttt{assignTag}(es', t)$
    $I.\text{append}(\text{"}blk\text{"}||t||es')$
$S[\texttt{CTR}] = num\_blocks + 1$
$keyTree = \texttt{newKeyTree}(K_{new}, K'_{PPRF})$
$JournalRotate(K_{new}, keyTree)$
$S[\texttt{K}] = K_{new}$
$S[\texttt{PPRF}] = K'_{PPRF}$
$\textbf{return}$
```

Figure 8. The internal journaling and refresh logic in the real-world secure deletion functionality Σ.

Simulator Construction. Given access to both S and I from $\mathcal{E}$'s interaction with the ideal functionality $\mathcal{F}$, the simulator runs an internal instance of Σ (which we call Σ_{Sim}) to generate a new (S', I') pair. By design, each non-empty memory location in S (excluding $S[0]$) contains either non-deleted messages or the constant 0. Additionally, I contains an ordered sequence of ("wr"$||i$) and ("del"$||i$) entries that encode the API requests issued by $\mathcal{E}$.

Sim starts at the beginning of I and one-by-one does the following for every $entry \in I$. If $entry$ = ("wr"$||i$), then Sim checks if the associated file f was later deleted by $\mathcal{E}$. To do this, Sim searches to the end of I looking for an entry of the form ("del"$||i$). If no such entry exists, then Sim learns f by looking at $S[i+1]$ and can issue a $Write(i, f)$ call to Σ_{Sim}. Otherwise, the file was deleted and Sim instead issues a $Write(i, 0^n)$ call to Σ_{Sim}. Lastly, if $entry$ = ("del"$||i$), then Sim issues a $Delete(i)$ to Σ_{Sim}. After iterating over all of I, the simulator takes the (S', I') pair from Σ_{Sim} and runs $Adv(1^\lambda, S', I')$ and outputs whatever Adv outputs.

Lemma A.1 (Efficient Simulator)**.** *Sim runs in time* $O(poly(\lambda))$.

Proof. Since $\mathcal{E}$ runs in time $O(poly(\lambda))$, I can be appended at most $O(poly(\lambda))$ times and thus $|I| \leq O(poly(\lambda))$. Observe that Sim loops over and processes each entry in I. For every entry, Sim issues either a *Write* or a *Delete* instruction to Σ_{Sim}, each of which runs in a constant number of steps. Furthermore, in the worst case, the entry is of the form ("wr"$||i$) and Sim must first search to the end of I to look for any associated *Delete* operations, which takes time at most $O(|I|) = O(poly(\lambda))$. Therefore, in the worst case, processing all of I takes time $O(poly(\lambda)) \cdot O(poly(\lambda)) = O(poly(\lambda))$. Finally, the simulator obtains (S', I') from Σ_{Sim} and runs $Adv(1^\lambda, S', I')$. Since Adv runs in time $O(poly(\lambda))$ the total run-time of *Sim* remains $O(poly(\lambda))$. □

Theorem A.2 (HOLEPUNCH achieves secure deletion of files)**.** *The* HOLEPUNCH *scheme* Σ *described in Figures 7 and 8 achieves secure deletion of files (Definition 4.1) when* F *is a puncturable PRF,* (E, D) *is an IND-CPA secure encryption scheme, and* H *is modeled as a random oracle.*

Proof. Observe that from the perspective of the environment $\mathcal{E}$, both the real and ideal games are identical up until step 6 in the games. This follows from the fact that $\mathcal{E}$ is only sending requests to read, write, and delete files, and any observable values that $\mathcal{E}$ sees up until this point were chosen by $\mathcal{E}$ itself. During step 6, the arrays (S, I) are given to either Adv or Sim which then return an output to $\mathcal{E}$. Since $\mathcal{E}$'s view is identical in both games up to step 6, it follows that if the output of Adv and Sim in step 6 are computationally indistinguishable then

$$|\Pr[\text{Real}^{\Sigma}_{\mathcal{E},Adv}(\lambda) = 1] - \Pr[\text{Ideal}^{\mathcal{F}}_{\mathcal{E},Sim}(\lambda) = 1]| \leq negl(\lambda)$$

By construction, Sim exactly replays $\mathcal{E}$'s interaction to a fresh instance of Σ (which we denote Σ_{Sim}), modulo the $Write(i, f)$ operations on files f that were later deleted by $\mathcal{E}$. For the deleted files, Sim instead issues a $Write(i, 0^n)$ instruction to Σ_{Sim}. Finally, Sim takes the arrays (S', I') from the interaction with Σ_{Sim} and outputs $Adv(1^\lambda, S', I')$. We therefore need to show that the (S', I') pair associated with Σ_{Sim} is computationally-indistinguishable from the

(S, I) pair associated with Σ in the real game. We proceed using a series of hybrids.

Hybrid 0**.** This is the $\text{Real}^{\Sigma}_{\mathcal{E},Adv}(\lambda)$ game defined in Figure 3.

Hybrid 1**.** This is the same game as $\text{Real}^{\Sigma}_{\mathcal{E},Adv}(\lambda)$ except for the following. Let f be the first file stored by $\mathcal{E}$ that is later deleted by a $Delete$ instruction, and let i be the file's associated tag. Then in Hybrid 1 we replace $\mathcal{E}$'s first $Write(i, f)$ call with a $Write(i, 0^n)$ call instead. We give a series of simple hybrids to show that Hybrid 0 and Hybrid 1 are indistinguishable.

Observe that the only difference between the (S, I) output by Σ in Hybrid 1 versus the (S, I) output by Σ in Hybrid 0 is that $(\text{``}\mathit{file}\text{''}||i||E(k_i, 0^n))$ is written to $I$ instead of $(\text{``}\mathit{file}\text{''}||i||E(k_i, f))$ as in Hybrid 0 (see lines 8 and 9 of the $Write$ pseudocode in Figure 7). But for any PPT adversary, as long as k_i is indistinguishable from random, then by the IND-CPA security of (E, D), the ciphertexts $E(k_i, f)$ and $E(k_i, 0^n)$ are computationally indistinguishable from random (and hence from each other). Observe that k_i is a uniformly random key (line 9 of *Setup*) and is only ever written to I as $(\text{``}\mathit{blk}\text{''}||\tau||E(k_\tau, ds))$ (lines 10-12 of *Setup*) where ds is the file key block containing k_i, and τ is the tag associated with the block. By the IND-CPA security of (E, D), the ciphertext $E(k_\tau, ds)$ is indistinguishable from random as long as k_τ is indistinguishable from random. We therefore need to show that the key k_τ is pseudorandom. Recall that $k_\tau = F(K_{PPRF}, \tau)$. Furthermore, for every PPT adversary Adv, even if Adv has access to K^*_{PPRF} punctured at the point τ, the value $F(K^*_{PPRF}, \tau)$ is indistinguishable from random by the *psuedorandom at punctured points* property (§3.1). Note that Σ only ever writes an unpunctured version of K_{PPRF} to I using the key tree data structure (lines 6 and 7 of *Setup*). The root of the key tree is a master key, that we will denote K_{old}. The key tree data structure uses K_{old} to encrypt the keys in layer 2, which in turn encrypts the keys in layer 3, which finally encrypt the leaves representing K_{PPRF}. After puncturing the PPRF key at a point τ, the PPRF subkey that evaluated τ is zeroed out and re-encrypted under a fresh key in layer 3 of the tree, which in turn is re-encrypted under a fresh key in layer 2 of the tree, which is finally re-encrypted under a new root (master) key K_{new}. Therefore, by the IND-CPA security of (E, D), and a series of simple hybrid arguments, the K_{PPRF} is indistinguishable from random as long as K_{old} is indistinguishable from random.

Finally, the master key K_{old} is only ever stored in $S[\texttt{K}]$. Additionally, the value $H(K_{old})$ is written to I during the journaling operation (line 2 of *JournalDelete* in Figure 7). However, after the $Delete(i)$ call, the master key is rotated to a new uniformly random key K_{new} such that $S[\texttt{K}] = K_{new}$. Thus, for any PPT adversary Adv that receives (S, I) from Σ, the probability that Adv queries H on K_{old} is negligible and therefore $H(K_{old})$ is indistinguishable from random. It follows that for any PPT adversary that receives (S, I) in Hybrid 1, the key K_{old} is indistinguishable from random, and thus Hybrid 0 and Hybrid 1 are indistinguishable.

Hybrid 2**.** This is the same as Hybrid 1 except now we also replace the second file that is stored and later deleted by $\mathcal{E}$ with the all zeros string, i.e., replace the corresponding $Write(i, f)$ call with a $Write(i, 0^n)$ call. The proof that Hybrids 1 and 2 are indistinguishable follows almost exactly the same as the proof that Hybrid 0 and Hybrid 1 are indistinguishable. The one difference is that we must account for the fact that the ciphertext $E(K_{old}, K'_{old})$ is written to I (line 3 of *JournalDelete*) after the deletion operation of the first file where K_{old} is original master key in Hybrid 1, and K'_{old} is the newly rotated master key from Hybrid 1, i.e., K'_{old} is K_{new} from Hybrid 1 that is stored in $S[\texttt{K}]$ after deleting the first file. Thus, to show that K'_{old} is pseudorandom, we must additionally show that $E(K_{old}, K'_{old})$ is indistinguishable from random. Because we showed that K_{old} is indistinguishable from random in Hybrid 1, by the IND-CPA security of (E, D) it follows that $E(K_{old}, K'_{old})$ is indistinguishable from random in Hybrid 2.

Hybrid 3 **through** d**.** Let d be the total number of files that were stored and later deleted in $\mathcal{E}$'s interaction. Then for $j = 3 \dots d$, Hybrid j is identical to Hybrid $j-1$ except that we additionally take the j^{th} deleted file and replace its entry in I with $(\text{``}\mathit{file}\text{''}||i||E(k_i, 0^n))$ where i is the file's identifier. Each hybrid uses the same argument as Hybrid 2 to show that Hybrid j and Hybrid $j-1$ are computationally indistinguishable.

Hybrid $(d+1)$. This is the $\text{Ideal}^{\mathcal{F}}_{\mathcal{E},Sim}(\lambda)$ game. In $\text{Ideal}^{\mathcal{F}}_{\mathcal{E},Sim}(\lambda)$, the internal state of the ideal functionality (S, I) is given to Sim. However, Sim then constructs (S', I') by interacting with a fresh instance of the real functionality Σ_{Sim} such that (S', I') exactly matches the (S, I) pair given to Adv in Hybrid d. Finally, Sim runs $Adv(1^\lambda, S', I')$ which outputs the same output as Adv in Hybrid d. It follows that Hybrid d and Hybrid $(d+1)$ are identically distributed.

By the indistinguishability of Hybrid 0 and Hybrid $(d+1)$ we have shown that the real-world functionality Σ emulates the ideal functionality $\mathcal{F}$ (Figure 2).

□