hash-function-transition

Git hash function transition ==========

Objective --------- Migrate Git from SHA-1 to a stronger hash function.

Background ---------- At its core, the Git version control system is a content addressable filesystem. It uses the SHA-1 hash function to name content. For example, files, directories, and revisions are referred to by hash values unlike in other traditional version control systems where files or versions are referred to via sequential numbers. The use of a hash function to address its content delivers a few advantages:

- Integrity checking is easy. Bit flips, for example, are easily detected, as the hash of corrupted content does not match its name.
- Lookup of objects is fast.

Using a cryptographically secure hash function brings additional advantages:

- Object names can be signed and third parties can trust the hash to address the signed object and all objects it references.
- Communication using Git protocol and out of band communication methods have a short reliable string that can be used to reliably address stored content.

Over time some flaws in SHA-1 have been discovered by security researchers. On 23 February 2017 the SHAttered attack (https://web.archive.org/web/20260207211148/https://shattered.io/) demonstrated a practical SHA-1 hash collision.

Git v2.13.0 and later subsequently moved to a hardened SHA-1 implementation by default, which isn't vulnerable to the SHAttered attack, but SHA-1 is still weak.

Thus it's considered prudent to move past any variant of SHA-1 to a new hash. There's no guarantee that future attacks on SHA-1 won't be published in the future, and those attacks may not have viable mitigations.

If SHA-1 and its variants were to be truly broken, Git's hash function could not be considered cryptographically secure any more. This would impact the communication of hash values because we could not trust that a given hash value represented the known good version of content that the speaker intended.

SHA-1 still possesses the other properties such as fast object lookup and safe error checking, but other hash functions are equally suitable that are believed to be cryptographically secure.

Choice of Hash ---------- The hash to replace the hardened SHA-1 should be stronger than SHA-1 was: we would like it to be trustworthy and useful in practice for at least 10 years.

Some other relevant properties:

1. A 256-bit hash (long enough to match common security practice; not excessively long to hurt performance and disk usage).

2. High quality implementations should be widely available (e.g., in OpenSSL and Apple CommonCrypto).

3. The hash function's properties should match Git's needs (e.g. Git requires collision and 2nd preimage resistance and does not require length extension resistance).

4. As a tiebreaker, the hash should be fast to compute (fortunately many contenders are faster than SHA-1).

There were several contenders for a successor hash to SHA-1, including SHA-256, SHA-512/256, SHA-256x16, K12, and BLAKE2bp-256.

In late 2018 the project picked SHA-256 as its successor hash.

See 0ed8d8da374 (doc hash-function-transition: pick SHA-256 as NewHash, 2018-08-04) and numerous mailing list threads at the time, particularly the one starting at https://lore.kernel.org/git/20180609224913.GC38834@genre.crustytoothpaste.net/ for more information.

Goals ----- 1. The transition to SHA-256 can be done one local repository at a time. a. Requiring no action by any other party. b. A SHA-256 repository can communicate with SHA-1 Git servers (push/fetch). c. Users can use SHA-1 and SHA-256 identifiers for objects interchangeably (see "Object names on the command line", below). d. New signed objects make use of a stronger hash function than SHA-1 for their security guarantees. 2. Allow a complete transition away from SHA-1. a. Local metadata for SHA-1 compatibility can be removed from a repository if compatibility with SHA-1 is no longer needed. 3. Maintainability throughout the process. a. The object format is kept simple and consistent. b. Creation of a generalized repository conversion tool.

Non-Goals --------- 1. Add SHA-256 support to Git protocol. This is valuable and the logical next step but it is out of scope for this initial design. 2. Transparently improving the security of existing SHA-1 signed objects. 3. Intermixing objects using multiple hash functions in a single repository. 4. Taking the opportunity to fix other bugs in Git's formats and protocols. 5. Shallow clones and fetches into a SHA-256 repository. (This will change when we add SHA-256 support to Git protocol.) 6. Skip fetching some submodules of a project into a SHA-256 repository. (This also depends on SHA-256 support in Git protocol.)

Overview -------- We introduce a new repository format extension. Repositories with this extension enabled use SHA-256 instead of SHA-1 to name their objects. This affects both object names and object content — both the names of objects and all references to other objects within an object are

switched to the new hash function.

SHA-256 repositories cannot be read by older versions of Git.

Alongside the packfile, a SHA-256 repository stores a bidirectional mapping between SHA-256 and SHA-1 object names. The mapping is generated locally and can be verified using "git fsck". Object lookups use this mapping to allow naming objects using either their SHA-1 and SHA-256 names interchangeably.

"git cat-file" and "git hash-object" gain options to display an object in its SHA-1 form and write an object given its SHA-1 form. This requires all objects referenced by that object to be present in the object database so that they can be named using the appropriate name (using the bidirectional hash mapping).

Fetches from a SHA-1 based server convert the fetched objects into SHA-256 form and record the mapping in the bidirectional mapping table (see below for details). Pushes to a SHA-1 based server convert the objects being pushed into SHA-1 form so the server does not have to be aware of the hash function the client is using.

Detailed Design ---------- Repository format extension ~~~~~~~~~~~~~~~~~~~~~~~~~~~ A SHA-256 repository uses repository format version `1` (see linkgit:gitrepository-layout[5]) with `extensions.objectFormat` and `extensions.compatObjectFormat` (see linkgit:git-config[1]) set to:

```
[core]
    repositoryFormatVersion = 1
[extensions]
    objectFormat = sha256
    compatObjectFormat = sha1
```

The combination of setting `core.repositoryFormatVersion=1` and populating `extensions.*` ensures that all versions of Git later than `v0.99.9l` will die instead of trying to operate on the SHA-256 repository, instead producing an error message.

```
# Between v0.99.9l and v2.7.0
$ git status
fatal: Expected git repo version <= 0, found 1
# After v2.7.0
$ git status
fatal: unknown repository extensions found:
    objectformat
    compatobjectformat
```

See the "Transition plan" section below for more details on these repository extensions.

Object names ~~~~~~~~~~~~~ Objects can be named by their 40 hexadecimal digit SHA-1 name or 64 hexadecimal digit SHA-256 name, plus names derived from those (see gitrevisions(7)).

The SHA-1 name of an object is the SHA-1 of the concatenation of its type, length, a nul byte, and the object's SHA-1 content. This is the traditional <sha1> used in Git to name objects.

The SHA-256 name of an object is the SHA-256 of the concatenation of its type, length, a nul byte, and the object's SHA-256 content.

Object format ~~~~~~~~~~~~~~ The content as a byte sequence of a tag, commit, or tree object named by SHA-1 and SHA-256 differ because an object named by SHA-256 name refers to other objects by their SHA-256 names and an object named by SHA-1 name refers to other objects by their SHA-1 names.

The SHA-256 content of an object is the same as its SHA-1 content, except that objects referenced by the object are named using their SHA-256 names instead of SHA-1 names. Because a blob object does not refer to any other object, its SHA-1 content and SHA-256 content are the same.

The format allows round-trip conversion between SHA-256 content and SHA-1 content.

Object storage ~~~~~~~~~~~~~~~ Loose objects use zlib compression and packed objects use the packed format described in linkgit:gitformat-pack[5], just like today. The content that is compressed and stored uses SHA-256 content instead of SHA-1 content.

Pack index ~~~~~~~~~~ Pack index (.idx) files use a new v3 format that supports multiple hash functions. They have the following format (all integers are in network byte order):

- A header appears at the beginning and consists of the following:
  - The 4-byte pack index signature: '\377t0c'
  - 4-byte version number: 3
  - 4-byte length of the header section, including the signature and version number
  - 4-byte number of objects contained in the pack
  - 4-byte number of object formats in this pack index: 2
  - For each object format:
    - 4-byte format identifier (e.g., 'sha1' for SHA-1)
    - 4-byte length in bytes of shortened object names. This is the shortest possible length needed to make names in the shortened object name table unambiguous.
    - 8-byte integer, recording where tables relating to this format are stored in this index file, as an offset from the beginning.
  - 8-byte offset to the trailer from the beginning of this file.
  - Zero or more additional key/value pairs (4-byte key, 4-byte value). Only one key is supported: 'PSRC'. See the "Loose objects and unreachable objects" section for supported values and how this is used. All other keys are reserved. Readers must ignore unrecognized keys.
- Zero or more NUL bytes. This can optionally be used to improve the alignment of the full object name table below.

- Tables for the first object format:
  - A sorted table of shortened object names. These are prefixes of the names of all objects in this pack file, packed together without offset values to reduce the cache footprint of the binary search for a specific object name.

- A table of full object names in pack order. This allows resolving a reference to "the nth object in the pack file" (from a reachability bitmap or from the next table of another object format) to its object name.

- A table of 4-byte values mapping object name order to pack order. For an object in the table of sorted shortened object names, the value at the corresponding index in this table is the index in the previous table for that same object. This can be used to look up the object in reachability bitmaps or to look up its name in another object format.

- A table of 4-byte CRC32 values of the packed object data, in the order that the objects appear in the pack file. This is to allow compressed data to be copied directly from pack to pack during repacking without undetected data corruption.

- A table of 4-byte offset values. The index of this table in pack order indicates where that object can be found in the pack file. These are usually 31-bit pack file offsets, but large offsets are encoded as an index into the next table with the most significant bit set.

- A table of 8-byte offset entries (empty for pack files less than 2 GiB). Pack files are organized with heavily used objects toward the front, so most object references should not need to refer to this table.
  - Zero or more NUL bytes.
  - Tables for the second object format, with the same layout as above, up to and not including the table of CRC32 values.
  - Zero or more NUL bytes.
  - The trailer consists of the following:
- A copy of the full main hash checksum at the end of the corresponding packfile.

- Full main hash checksum of all of the above.

The "full main hash" is a full-length hash of the main (not compatibility) algorithm in the repository. Thus, if the main algorithm is SHA-256, this is a 32-byte SHA-256 hash and for SHA-1, it’s a 20-byte SHA-1 hash.

Loose object index ~~~~~~~~~~~~~~~~~~ A new file $GIT_OBJECT_DIR/loose-object-idx contains information about all loose objects. Its format is

```
# loose-object-idx
(sha256-name SP sha1-name LF)*
```

where the object names are in hexadecimal format. The file is not sorted.

The loose object index is protected against concurrent writes by a lock file $GIT_OBJECT_DIR/loose-object-idx.lock. To add a new loose object:

1. Write the loose object to a temporary file, like today.
2. Open loose-object-idx.lock with O_CREAT | O_EXCL to acquire the lock.
3. Rename the loose object into place.
4. Open loose-object-idx with O_APPEND and write the new object
5. Unlink loose-object-idx.lock to release the lock.

To remove entries (e.g. in "git pack-refs" or "git-prune"):

1. Open loose-object-idx.lock with O_CREAT | O_EXCL to acquire the lock.
2. Write the new content to loose-object-idx.lock.
3. Unlink any loose objects being removed.
4. Rename to replace loose-object-idx, releasing the lock.

Translation table ~~~~~~~~~~~~~~~~~~ The index files support a bidirectional mapping between SHA-1 names and SHA-256 names. The lookup proceeds similarly to ordinary object lookups. For example, to convert a SHA-1 name to a SHA-256 name:

```
1. Look for the object in idx files. If a match is present in the
   idx's sorted list of truncated SHA-1 names, then:
   a. Read the corresponding entry in the SHA-1 name order to pack
      name order mapping.
   b. Read the corresponding entry in the full SHA-1 name table to
      verify we found the right object. If it is, then
   c. Read the corresponding entry in the full SHA-256 name table.
      That is the object's SHA-256 name.
2. Check for a loose object. Read lines from loose-object-idx until
   we find a match.
```

Step (1) takes the same amount of time as an ordinary object lookup: O(number of packs * log(objects per pack)). Step (2) takes O(number of loose objects) time. To maintain good performance it will be necessary to keep the number of loose objects low. See the "Loose objects and unreachable objects" section below for more details.

Since all operations that make new objects (e.g., "git commit") add the new objects to the corresponding index, this mapping is possible for all objects in the object store.

Reading an object’s SHA-1 content ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ The SHA-1 content of an object can be read by converting all SHA-256 names of its SHA-256 content references to SHA-1 names using the translation table.

Fetch ~~~~~ Fetching from a SHA-1 based server requires translating between SHA-1 and SHA-256 based representations on the fly.

SHA-1s named in the ref advertisement that are present on the client can be translated to SHA-256 and looked up as local objects using the translation table.

Negotiation proceeds as today. Any "have"s generated locally are converted to SHA-1 before being sent to the server, and SHA-1s mentioned by the server are converted to SHA-256 when looking them up locally.

After negotiation, the server sends a packfile containing the requested objects. We convert the packfile to SHA-256 format using the following steps:

1. index-pack: inflate each object in the packfile and compute its SHA-1. Objects can contain deltas in OBJ_REF_DELTA format against objects the client has locally. These objects can be looked up using the translation table and their SHA-1 content read as described above to resolve the deltas.
2. topological sort: starting at the "want"s from the negotiation phase, walk through objects in the pack and emit a list of them, excluding blobs, in reverse topologically sorted order, with each object coming later in the list than all objects it references. (This list only contains objects reachable from the "wants". If the pack from the server contained additional extraneous objects, then they will be discarded.)
3. convert to SHA-256: open a new SHA-256 packfile. Read the topologically sorted list just generated. For each object, inflate its SHA-1 content, convert to SHA-256 content, and write it to the SHA-256 pack. Record the new SHA-1<-->SHA-256 mapping entry for use in the idx.
4. sort: reorder entries in the new pack to match the order of objects in the pack the server generated and include blobs. Write a SHA-256 idx file
5. clean up: remove the SHA-1 based pack file, index, and topologically sorted list obtained from the server in steps 1 and 2.

Step 3 requires every object referenced by the new object to be in the translation table. This is why the topological sort step is necessary.

As an optimization, step 1 could write a file describing what non-blob objects each object it has inflated from the packfile references. This makes the topological sort in step 2 possible without inflating the objects in the packfile for a second time. The objects need to be inflated again in step 3, for a total of two inflations.

Step 4 is probably necessary for good read-time performance. "git pack-objects" on the server optimizes the pack file for good data locality (see Documentation/technical/pack-heuristics.adoc).

Details of this process are likely to change. It will take some experimenting to get this to perform well.

Push ~~~~ Push is simpler than fetch because the objects referenced by the pushed objects are already in the translation table. The SHA-1 content of each object being pushed can be read as described in the "Reading an object’s SHA-1 content" section to generate the pack written by git send-pack.

Signed Commits ~~~~~~~~~~~~~~~ We add a new field "gpgsig-sha256" to the commit object format to allow signing commits without relying on SHA-1. It is similar to the existing "gpgsig" field. Its signed payload is the SHA-256 content of the commit object with any "gpgsig" and "gpgsig-sha256" fields removed.

This means commits can be signed

1. using SHA-1 only, as in existing signed commit objects
2. using both SHA-1 and SHA-256, by using both gpgsig-sha256 and gpgsig fields.
3. using only SHA-256, by only using the gpgsig-sha256 field.

Old versions of "git verify-commit" can verify the gpgsig signature in cases (1) and (2) without modifications and view case (3) as an ordinary unsigned commit.

Signed Tags ~~~~~~~~~~~~ We add new fields "gpgsig" and "gpgsig-sha256" to the tag object format to allow signing tags in both formats. The in-body signature is used for the signature in the current hash algorithm and the header is used for the signature in the other algorithm. Thus, a dual-signature tag will contain both an in-body signature and a gpgsig-sha256 header for the SHA-1 format of an object or both an in-body signature and a gpgsig header for the SHA-256 format of and object.

The signed payload of the tag is the content of the tag in the current algorithm with both its gpgsig and gpgsig-sha256 fields and "-----BEGIN PGP SIGNATURE-----" delimited in-body signature removed.

This means tags can be signed using one or both algorithms.

Mergetag embedding ~~~~~~~~~~~~~~~~~~ The mergetag field in the SHA-1 content of a commit contains the SHA-1 content of a tag that was merged by that commit.

The mergetag field in the SHA-256 content of the same commit contains the SHA-256 content of the same tag.

Submodules ~~~~~~~~~~ To convert recorded submodule pointers, you need to have the converted submodule repository in place. The translation table of the submodule can be used to look up the new hash.

Loose objects and unreachable objects
~~~~~~ Fast lookups in the loose-object-idx require that the number of loose objects not grow too high.

"git gc --auto" currently waits for there to be 6700 loose objects present before consolidating them into a packfile. We will need to measure to find a more appropriate threshold for it to use.

"git gc --auto" currently waits for there to be 50 packs present before combining packfiles. Packing loose objects more aggressively may cause the number of pack files to grow too quickly. This can be mitigated by using a strategy similar to Martin Fick’s exponential rolling garbage collection
~~~~~~

script: https://gerrit-review.googlesource.com/c/gerrit/+/35215

"git gc" currently expels any unreachable objects it encounters in pack files to loose objects in an attempt to prevent a race when pruning them (in case another process is simultaneously writing a new object that refers to the about-to-be-deleted object). This leads to an explosion in the number of loose objects present and disk space usage due to the objects in delta form being replaced with independent loose objects. Worse, the race is still present for loose objects.

Instead, "git gc" will need to move unreachable objects to a new packfile marked as UNREACHABLE_GARBAGE (using the PSRC field; see below). To avoid the race when writing new objects referring to an about-to-be-deleted object, code paths that write new objects will need to copy any objects from UNREACHABLE_GARBAGE packs that they refer to new, non-UNREACHABLE_GARBAGE packs (or loose objects). UNREACHABLE_GARBAGE are then safe to delete if their creation time (as indicated by the file's mtime) is long enough ago.

To avoid a proliferation of UNREACHABLE_GARBAGE packs, they can be combined under certain circumstances. If "gc.garbageTtl" is set to greater than one day, then packs created within a single calendar day, UTC, can be coalesced together. The resulting packfile would have an mtime before midnight on that day, so this makes the effective maximum ttl the garbageTtl + 1 day. If "gc.garbageTtl" is less than one day, then we divide the calendar day into intervals one-third of that ttl in duration. Packs created within the same interval can be coalesced together. The resulting packfile would have an mtime before the end of the interval, so this makes the effective maximum ttl equal to the garbageTtl * 4/3.

This rule comes from Thirumala Reddy Mutchukota's JGit change https://git.eclipse.org/r/90465.

The UNREACHABLE_GARBAGE setting goes in the PSRC field of the pack index. More generally, that field indicates where a pack came from:

- 1 (PACK_SOURCE_RECEIVE) for a pack received over the network
- 2 (PACK_SOURCE_AUTO) for a pack created by a lightweight "gc --auto" operation
- 3 (PACK_SOURCE_GC) for a pack created by a full gc
- 4 (PACK_SOURCE_UNREACHABLE_GARBAGE) for potential garbage discovered by gc
- 5 (PACK_SOURCE_INSERT) for locally created objects that were written directly to a pack file, e.g. from "git add ."

This information can be useful for debugging and for "gc --auto" to make appropriate choices about which packs to coalesce.

Caveats ------- Invalid objects ~~~~~~~~~~~~~~~~ The conversion from SHA-1 content to SHA-256 content retains any brokenness in the original object (e.g., tree entry modes encoded with leading 0, tree objects whose paths are not sorted correctly, and commit objects without an author or committer). This is a deliberate feature of the design to allow the conversion to round-trip.

More profoundly broken objects (e.g., a commit with a truncated "tree" header line) cannot be converted but were not usable by current Git anyway.

Shallow clone and submodules ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Because it requires all referenced objects to be available in the locally generated translation table, this design does not support shallow clone or unfetched submodules. Protocol improvements might allow lifting this restriction.

Alternates ~~~~~~~~~~ For the same reason, a SHA-256 repository cannot borrow objects from a SHA-1 repository using objects/info/alternates or $GIT_ALTERNATE_OBJECT_DIRECTORIES.

git notes ~~~~~~~~~ The "git notes" tool annotates objects using their SHA-1 name as key. This design does not describe a way to migrate notes trees to use SHA-256 names. That migration is expected to happen separately (for example using a file at the root of the notes tree to describe which hash it uses).

Server-side cost ~~~~~~~~~~~~~~~~ Until Git protocol gains SHA-256 support, using SHA-256 based storage on public-facing Git servers is strongly discouraged. Once Git protocol gains SHA-256 support, SHA-256 based servers are likely not to support SHA-1 compatibility, to avoid what may be a very expensive hash re-encode during clone and to encourage peers to modernize.

The design described here allows fetches by SHA-1 clients of a personal SHA-256 repository because it’s not much more difficult than allowing pushes from that repository. This support needs to be guarded by a configuration option — servers like git.kernel.org that serve a large number of clients would not be expected to bear that cost.

Meaning of signatures ~~~~~~~~~~~~~~~~~~~~~~ The signed payload for signed commits and tags does not explicitly name the hash used to identify objects. If some day Git adopts a new hash function with the same length as the current SHA-1 (40 hexadecimal digit) or SHA-256 (64 hexadecimal digit) objects then the intent behind the PGP signed payload in an object signature is unclear:

```
object e7e07d5a4fcc2a203d9873968ad3e6bd4d7419d7
type commit
tag v2.12.0
tagger Junio C Hamano <gitster@pobox.com> 1487962205 -0800

Git 2.12
```

Does this mean Git v2.12.0 is the commit with SHA-1 name e7e07d5a4fcc2a203d9873968ad3e6bd4d7419d7 or the commit with new-40-digit-hash-name e7e07d5a4fcc2a203d9873968ad3e6bd4d7419d7?

Fortunately SHA-256 and SHA-1 have different lengths. If Git starts using another hash with the same length to name objects, then it will need to change the format of signed payloads using that hash to address this issue.

Object names on the command line ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ To support the transition (see Transition plan below), this design supports four different modes of

operation:

```
1. ("dark launch") Treat object names input by the user as SHA-1 and
   convert any object names written to output to SHA-1, but store
   objects using SHA-256.  This allows users to test the code with no
   visible behavior change except for performance.  This allows
   running even tests that assume the SHA-1 hash function, to
   sanity-check the behavior of the new mode.

2. ("early transition") Allow both SHA-1 and SHA-256 object names in
   input. Any object names written to output use SHA-1. This allows
   users to continue to make use of SHA-1 to communicate with peers
   (e.g. by email) that have not migrated yet and prepares for mode 3.

3. ("late transition") Allow both SHA-1 and SHA-256 object names in
   input. Any object names written to output use SHA-256. In this
   mode, users are using a more secure object naming method by
   default.  The disruption is minimal as long as most of their peers
   are in mode 2 or mode 3.

4. ("post-transition") Treat object names input by the user as
   SHA-256 and write output using SHA-256. This is safer than mode 3
   because there is less risk that input is incorrectly interpreted
   using the wrong hash function.
```

The mode is specified in configuration.

The user can also explicitly specify which format to use for a particular revision specifier and for output, overriding the mode. For example:

```
git --output-format=sha1 log abac87a^{sha1}..f787cac^{sha256}
```

Transition plan ---------- Some initial steps can be implemented independently of one another:

- adding a hash function API (vtable)
- teaching fsck to tolerate the gpgsig-sha256 field
- excluding gpgsig-* from the fields copied by "git commit --amend"
- annotating tests that depend on SHA-1 values with a SHA1 test prerequisite
- using "struct object_id", GIT_MAX_RAWSZ, and GIT_MAX_HEXSZ consistently instead of "unsigned char *" and the hardcoded constants 20 and 40.
- introducing index v3
- adding support for the PSRC field and safer object pruning

The first user-visible change is the introduction of the objectFormat extension (without compatObjectFormat). This requires:

- teaching fsck about this mode of operation
- using the hash function API (vtable) when computing object names
- signing objects and verifying signatures
- rejecting attempts to fetch from or push to an incompatible repository

Next comes introduction of compatObjectFormat:

- implementing the loose-object-idx
- translating object names between object formats
- translating object content between object formats
- generating and verifying signatures in the compat format
- adding appropriate index entries when adding a new object to the object store
- --output-format option
- ^{sha1} and ^{sha256} revision notation
- configuration to specify default input and output format (see "Object names on the command line" above)

The next step is supporting fetches and pushes to SHA-1 repositories:

- allow pushes to a repository using the compat format
- generate a topologically sorted list of the SHA-1 names of fetched objects
- convert the fetched packfile to SHA-256 format and generate an idx file
- re-sort to match the order of objects in the fetched packfile

The infrastructure supporting fetch also allows converting an existing repository. In converted repositories and new clones, end users can gain support for the new hash function without any visible change in behavior (see "dark launch" in the "Object names on the command line" section). In particular this allows users to verify SHA-256 signatures on objects in the repository, and it should ensure the transition code is stable in production in preparation for using it more widely.

Over time projects would encourage their users to adopt the "early transition" and then "late transition" modes to take advantage of the new, more futureproof SHA-256 object names.

When objectFormat and compatObjectFormat are both set, commands generating signatures would generate both SHA-1 and SHA-256 signatures by default to support both new and old users.

In projects using SHA-256 heavily, users could be encouraged to adopt the "post-transition" mode to avoid accidentally making implicit use of SHA-1 object names.

Once a critical mass of users have upgraded to a version of Git that can verify SHA-256 signatures and have converted their existing repositories to support verifying them, we can add support for a setting to generate only SHA-256 signatures. This is expected to be at least a year later.

That is also a good moment to advertise the ability to convert repositories to use SHA-256 only, stripping out all SHA-1 related metadata. This improves performance by eliminating translation overhead and security by avoiding the possibility of accidentally relying on the safety of SHA-1.

Updating Git's protocols to allow a server to specify which hash functions it supports is also an important part of this transition. It is not discussed in detail in this document but this transition plan assumes it happens. :)

Alternatives considered ---------------------- Upgrading everyone working on a particular project on a flag day ~~~~~~ Projects like the Linux kernel are large and complex enough that flipping the switch for all projects based on the repository at once is infeasible.

Not only would all developers and server operators supporting developers have to switch on the same flag day, but supporting tooling (continuous integration, code review, bug trackers, etc) would have to be adapted as well. This also makes it difficult to get early feedback from some project participants testing before it is time for mass adoption.

Using hash functions in parallel ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ (e.g. https://lore.kernel.org/git/22708.8913.864049.452252@chiark.greenend.org.uk/) Objects newly created would be addressed by the new hash, but inside such an object (e.g. commit) it is still possible to address objects using the old hash function.

- You cannot trust its history (needed for bisectability) in the future without further work
- Maintenance burden as the number of supported hash functions grows (they will never go away, so they accumulate). In this proposal, by comparison, converted objects lose all references to SHA-1.

Signed objects with multiple hashes ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Instead of introducing the gpgsig-sha256 field in commit and tag objects for SHA-256 content based signatures, an earlier version of this design added "hash sha256 <SHA-256 name>" fields to strengthen the existing SHA-1 content based signatures.

In other words, a single signature was used to attest to the object content using both hash functions. This had some advantages:

- Using one signature instead of two speeds up the signing process.
- Having one signed payload with both hashes allows the signer to attest to the SHA-1 name and SHA-256 name referring to the same object.
- All users consume the same signature. Broken signatures are likely to be detected quickly using current versions of git.

However, it also came with disadvantages:

- Verifying a signed object requires access to the SHA-1 names of all objects it references, even after the transition is complete and translation table is no longer needed for

anything else. To support this, the design added fields such as "hash sha1 tree <SHA-1 name>" and "hash sha1 parent <SHA-1 name>" to the SHA-256 content of a signed commit, complicating the conversion process.
- Allowing signed objects without a SHA-1 (for after the transition is complete) complicated the design further, requiring a "nohash sha1" field to suppress including "hash sha1" fields in the SHA-256 content and signed payload.

Lazily populated translation table ~~~~~~ Some of the work of building the translation table could be deferred to push time, but that would significantly complicate and slow down pushes. Calculating the SHA-1 name at object creation time at the same time it is being streamed to disk and having its SHA-256 name calculated should be an acceptable cost.

Document History ----------

2017-03-03 bmwill@google.com, jonathantanmy@google.com, jrnieder@gmail.com, sbeller@google.com

- Initial version sent to https://lore.kernel.org/git/20170304011251.GA26789@aiede.mtv.corp.google.com

2017-03-03 jrnieder@gmail.com Incorporated suggestions from jonathantanmy and sbeller:

- Describe purpose of signed objects with each hash type
- Redefine signed object verification using object content under the first hash function

2017-03-06 jrnieder@gmail.com

- Use SHA3-256 instead of SHA2 (thanks, Linus and brian m. carlson).[1][2]
- Make SHA3-based signatures a separate field, avoiding the need for "hash" and "nohash" fields (thanks to peff[3]).
- Add a sorting phase to fetch (thanks to Junio for noticing the need for this).
- Omit blobs from the topological sort during fetch (thanks to peff).
- Discuss alternates, git notes, and git servers in the caveats section (thanks to Junio Hamano, brian m. carlson[4], and Shawn Pearce).
- Clarify language throughout (thanks to various commenters, especially Junio).

2017-09-27 jrnieder@gmail.com, sbeller@google.com

- Use placeholder NewHash instead of SHA3-256
- Describe criteria for picking a hash function.
- Include a transition plan (thanks especially to Brandon Williams for fleshing these ideas out)
- Define the translation table (thanks, Shawn Pearce[5], Jonathan Tan, and Masaya Suzuki)
- Avoid loose object overhead by packing more aggressively in "git gc --auto"

Later history:

- See the history of this file in git.git for the history of subsequent edits. This document history is no longer being maintained as it would now be superfluous to the commit log

References:

```
[1]
https://lore.kernel.org/git/CA+55aFzJtejiCjV0e43+9oR3QuJK2PiFiLQemytoLpyJWe6P9w@mail.gmail.com
/
[2] https://lore.kernel.org/git/CA+55aFz+gkAsDZ24zmePQuEs1XPS9BP_s8O7Q4wQ7LV7X5-
oDA@mail.gmail.com/
[3] https://lore.kernel.org/git/20170306084353.nrns455dvkdsfgo5@sigill.intra.peff.net/
[4] https://lore.kernel.org/git/20170304224936.rqqtkdvfjgyezsht@genre.crustytoothpaste.net
[5]
https://lore.kernel.org/git/CAJo=hJtoX9=AyLHHpUJS7fueV9ciZ_MNpnEPHUz8Whui6g9F0A@mail.gmail.com
/
```

□□□ #2
□ □□□ □□□ 2026-10-31 20:16:29 UTC
□ □□□ □□□ 2026-10-31 20:16:31 UTC