Modernizing Kernel Cryptography: From Complex APIs to Streamlined Libraries

Eric Biggers
Google
Upstream maintainer of lib/crypto/, lib/crc/, fs/crypto/, and fs/verity/

Linux Security Summit North America 2026

Linux kernel cryptography & CRCs

Cryptography and CRC (cyclic redundancy check) algorithms are everywhere!
There are many users of these algorithms in the kernel:

- Storage encryption, integrity, and authentication
  - dm-crypt, dm-verity, dm-integrity, fscrypt, fsverity, blk-crypto, blk-integrity, eCryptfs, filesystem checksums (ext4, f2fs, btrfs, xfs, ...), UBIFS auth
- Network encryption, integrity, and authentication
  - IPsec, WireGuard, mac80211, bluetooth, SMB, NVME-TCP, SCTP, TCP-AO, IPv6-SR, kTLS, Ceph
- Random number generation (e.g. /dev/random)
- Firmware and microcode integrity
- Denial-of-service protection: secure hash tables & cookies
- ... and much, much more!

Traditional kernel crypto API

- Linux "Crypto API" has existed for over 20 years
  - Includes both cryptographic and non-cryptographic algorithms
- Abstract API that is complex, hard-to-use, and slow
- Always been a problem, but hasn't aged well either
  - *Usability:* Doesn't match what most kernel developers actually want
  - *Hardware:* Not optimized for modern CPU-based acceleration
  - *Algorithms:* Outdated and rigid framework. New algorithms and implementation strategies often don't fit.

Case study: HMAC-SHA256

- "I just want to calculate a MAC."
  - E.g. HMAC-SHA256:
    - Input: `(key, data)`
    - Output: `mac`

Traditional crypto API: HMAC-SHA256 (part 1)

```
static int calc_hmac(const u8 *key, size_t keylen, const u8 *data, size_t datalen, u8 out[32])
{
        struct crypto_shash *tfm;
        int err;

        tfm = crypto_alloc_shash("hmac(sha256)", 0, 0);
        if (IS_ERR(tfm)) {
                pr_err("Failed to allocate hmac(sha256): %ld\n", PTR_ERR(tfm));
                return PTR_ERR(tfm);
        }
```

Traditional crypto API: HMAC-SHA256 (part 2)

```
        err = crypto_shash_setkey(tfm, key, keylen);
        if (!err) {
                SHASH_DESC_ON_STACK(desc, tfm);

                desc->tfm = tfm;
                err = crypto_shash_digest(desc, data, datalen, out);
                shash_desc_zero(desc);
        }
        if (err)
                pr_err("Failed to calculate HMAC-SHA256 value: %d\n", err);
        crypto_free_shash(tfm);
        return err;
}
```

Traditional crypto API: HMAC-SHA256 (analysis)

- The code seems to work, so it gets merged.
- But bug reports start coming in: On some systems it fails at runtime with:

```
Failed to allocate hmac(sha256): -2 [i.e. -ENOENT]
```

- The fix (for your Kconfig option):
  - `select CRYPTO_HMAC`
  - `select CRYPTO_SHA256`
- But what if this code needs to run during boot?
- The fix for the fix (for your .c file):
  - MODULE_SOFTDEP("hmac");
  - MODULE_SOFTDEP("sha256");
- Still out of luck if you want your code to run before late_initcall.

Traditional crypto API: HMAC-SHA256 (analysis)

- What about performance?
- Benchmark of `calc_hmac()` on v6.12.83 on x86_64 with key_len=32, data_len=128:
  - 2115 cycles/call, with only 38% spent in sha256_transform_ni().
  - (With CONFIG_CRYPTO_SHA256_SSSE3=y. This was easily forgotten.)
- The majority of the time (62%) is things other than the "actual" cryptography!
  - Allocating and freeing memory
  - Atomic operations, including manipulating reference counts and locking
  - Making indirect calls
  - Linearly searching list of all algorithms and doing string comparisons

Traditional crypto API: HMAC-SHA256 (analysis)

- Other issues
  - crypto_alloc_shash() can sleep. Can't do it in atomic context.
- Can avoid some issues by pre-allocating the crypto_shash
  - But this doesn't easily work if you need to set a new key on each call
    - In that case you have to roll your own cache of crypto_shashes
    - Or use transformation "cloning", but that still requires a kmalloc()

Library example: HMAC-SHA256

- A better way: Just add a straightforward function that directly does HMAC-SHA256 without going through the traditional crypto API.

```
hmac_sha256_usingrawkey(key, key_len, data, data_len, out);
```

- I added this function in v6.17, as part of a set of APIs for SHA-1 and SHA-2

Library example: HMAC-SHA256

- Easier to use
  - Single function call in common case. Other functions are available for incremental computations, but simple case is kept simple
  - Returns void and works in all contexts. No need to handle errors!
  - Standard link-time dependency. No runtime failures
- Faster
  - 2.5 times as fast in data_len=128 benchmark (2115 => 845 cycles)
  - Minimal overhead, and architecture-optimized code always used
- Testing: sha256_kunit. No need to learn an ad-hoc testing system!
- Compliance: FIPS 140 self-test included

Not really a new approach

- Long history of developers working around the traditional crypto API
- In 2019, library APIs were added for the algorithms used by WireGuard: ChaCha20, Poly1305, BLAKE2s, Curve25519. (Originally proposed as "Zinc"; a modified version of the code was merged and put in lib/crypto/.)
- SHA-1, SHA-256, SipHash, ARC4, AES, DES, Michael_MIC, CRCs, etc.
- Plus, library APIs have always been the norm outside crypto subsystem
- So, not particularly novel. We're just expanding and refining an approach that's already been used for many years and has been found to work well.

Case study: ChaCha20Poly1305

- What about encryption?
- "I just want to encrypt some data using an AEAD."
  - E.g. ChaCha20Poly1305
    - Input: key, nonce, plaintext, and associated data
    - Output: ciphertext

Traditional crypto API example: ChaCha20Poly1305 (1 of 4)

```
// Don't forget to add MODULE_SOFTDEPs on rfc7539, chacha20, and poly1305!
struct crypto_aead *tfm = crypto_alloc_aead("rfc7539(chacha20,poly1305)", 0, 0);
// TODO: handle IS_ERR(tfm)

int err = crypto_aead_setkey(tfm, key, key_len);
// TODO: handle err != 0
```

Traditional crypto API example: ChaCha20Poly1305 (2 of 4)

```
struct scatterlist src_sg[2], dst_sg[2];
sg_init_table(src_sg, ARRAY_SIZE(src_sg));
sg_init_table(dst_sg, ARRAY_SIZE(dst_sg));
// The below assumes that none of ad, src, and dst points to vmalloc memory
sg_set_buf(&src_sg[0], ad, ad_len);
sg_set_buf(&src_sg[1], src, data_len);
sg_set_buf(&dst_sg[0], ad, ad_len);
sg_set_buf(&dst_sg[1], dst, data_len + crypto_aead_authsize(tfm));
```

Traditional crypto API example: ChaCha20Poly1305 (3 of 4)

```
struct aead_request *req = aead_request_alloc(tfm, GFP_KERNEL);
// TODO: handle req == NULL
aead_request_set_ad(req, ad_len);
aead_request_set_crypt(req, src_sg, dst_sg, data_len, nonce);
DEFINE_CRYPTO_WAIT(wait);
aead_request_set_callback(req, CRYPTO_TFM_REQ_MAY_SLEEP |
                               CRYPTO_TFM_REQ_MAY_BACKLOG,
                          crypto_req_done, &wait);
```

Traditional crypto API example: ChaCha20Poly1305 (4 of 4)

```
err = crypto_aead_encrypt(req);
err = crypto_wait_req(err, &wait);
// TODO: handle err != 0

aead_request_free(req);
crypto_free_aead(tfm);
```

Library example: ChaCha20Poly1305

```
chacha20poly1305_encrypt(dst, src, src_len, ad, ad_len, nonce, key);
```

- Again, just a boring library function. Much simpler and faster.
- Not new; this was added in 2019 for WireGuard. Recently I've focused on hashing & MACs instead, where the traditional API isn't *quite* as bad.
- But this is still a great example that shows the extreme contrast between the two APIs. Traditional API has a huge amount of complexity to support asynchronous offload, despite this being unused/unhelpful in practice

Algorithms the library supports as of 7.1

- **CRCs**: CRC-T10DIF, CRC32, CRC32C, CRC64-NVME, CRC64-ECMA
- **Block ciphers**: AES, DES
- **Hash functions, MACs, and XOFs**: AES-CMAC, AES-XCBC-MAC, BLAKE2b, BLAKE2s, GHASH, MD5, NH, POLYVAL, Poly1305, SHA-1, SHA-2, SHA-3, SM3
- **Stream ciphers**: ChaCha20, ChaCha12
- **Authenticated encryption**: ChaCha20Poly1305, XChaCha20Poly1305
- **Key agreement**: Curve25519
- **Digital signature**: ML-DSA
- ... and several others. More are being added, too.
- About half of these I added recently. The rest already existed but have been improved. Library isn't new, but rather is just being more widely used now

Simple API for each algorithm

- The libraries just provide a separate set of functions for each algorithm.
  - This allows the API to be whatever works best for each algorithm. Just whatever is simplest, easiest to use, and most efficient.
- Libraries work best if you need a single algorithm. This is the common case
- You can still have "Crypto Agility". It's just handled by calling code instead.
  - For specific set of algorithms: Just dispatch to functions by algorithm
  - Traditional crypto API still remains available too and has been reimplemented using the libraries where possible.

Crypto & CRC library updates in v6.14..v7.1

- I started maintaining the kernel's crypto and CRC library code.
  - Crypto library co-maintainers: Ard Biesheuvel & Jason Donenfeld
- Established modern conventions: KUnit testing, documentation, direct integration of arch-optimized code, optimizations enabled by default
- Migrated many algorithms from traditional crypto subsystem into library, with traditional API reimplemented on top
- Converted lots of kernel code to use library APIs instead of traditional API
- Lots of negative diffs, performance improvements, and bugs fixed!

Kernel code now using libraries

- Lots of kernel code is now just using the libraries. (This is in addition to the internal use of the libraries by the traditional crypto API.) Some examples:
  - **SHA-256**: apparmor, btrfs, dm-ima, dm-verity, encrypted-keys, firmware_loader, fscrypt, fsverity, ipe, ipv6-sr, kexec, mptcp, net/ceph, purgatory, r8152, tsm-mr, sctp, smb/client, smb/server, tcp-ao
  - **SHA-512**: fscrypt, fsverity, smb/client, smb/server, tsm-mr
  - **AES-CMAC**: mac80211, bluetooth, smb/client, smb/server, tcp-ao
- The picture is similar for other algorithms

Crypto & CRC library updates in v6.14..v7.1

- New features (that didn't already exist in traditional crypto API)
  - v7.0: ML-DSA signature verification. "Post-quantum" cryptography
  - v6.19: SHAKE128 and SHAKE256 extendable-output functions (XOFs)
  - v6.18: SHA-256 interleaved hashing to accelerate dm-verity and fsverity
- Other improvements
  - CRC optimizations for x86_64, RISC-V, and arm64. Massive performance improvements in some cases (up to 12400%)
  - Some reductions in timing side channels. Borrowed modern GHASH & POLYVAL implementation strategy from BoringSSL / BearSSL.

Testing & code quality

- First priority is correctness! Optimizations welcome, but must be useful & tested
- Adoption of KUnit:
  - Crypto library uses KUnit. 18 test suites as of v7.1.
  - Much easier to run tests. Already enabled in KernelCI & some downstreams.
- Goal: 100% code coverage for lib/{crypto,crc}/
  - I'd guess we're at somewhere about 75%. Limited by platform-specific code
  - I'm requiring QEMU support for new code!
  - I've been testing ~50 combinations in QEMU myself. But I'm hoping this becomes less necessary over time due to the adoption of KUnit. If you care about Linux on a platform, please run **all** the kernel's KUnit tests!

When to use the library functions?

Generally, if there are library functions for the algorithm(s) you need, use them!

Exceptions:

- Library doesn't yet support everything the traditional crypto API does. I'm working on adding important missing/incomplete algorithms, e.g. AES-GCM
- Kernel features with many supported algorithms, especially ones that pass a userspace-provided string directly to `crypto_alloc_*()`, will generally need to continue using traditional API for backwards compatibility

Best practice: choose your algorithms!

- Best practice for new kernel features: Decide what algorithms you're using!
- Don't do e.g. `crypto_alloc_shash(user_provided_string)`
  - Enables weird mix of algorithms, many of which are insecure, obsolete, and/or make no sense for your feature. (MD5, MD4, CRC32, "null", ...)
  - Huge footgun for users & maintenance burden for kernel developers
  - Security vulnerabilities. Not holding up well against syzbot, AI tools, etc.
- Instead, just **thoughtfully chose which algorithm(s) to support!**
  - Preferably keep things simple by just choosing one good algorithm
  - You can always add more later if ever needed
  - Reach out to linux-crypto@vger.kernel.org if you need help

Call to action: Help harden and deprecate AF_ALG!

- Outside the scope of the crypto library, but this is important...
- AF_ALG (added in 2010) is a huge mistake. It exposes nearly all the traditional crypto API to unprivileged userspace, and it does so in a bug-prone way.
- The result: continuous vulnerabilities. https://copy.fail is just one example.
- AF_ALG is used only by a few userspace programs (e.g. iwd) that should use userspace crypto instead. Due to these, distros can't disable the kconfigs.
- You can help! Help migrate these userspace programs to userspace crypto. Disable AF_ALG on systems that don't need it. During the transition period, help harden AF_ALG, and (if applicable) opt into hardening as it's added.

Questions?

[Extra] What about hardware offload?

- There are three approaches to symmetric crypto acceleration:
  1. **[Modern] On-CPU acceleration.** Fast, simple, and widely used. The kernel's crypto library focuses on this approach, as do userspace crypto libraries
  2. **[Modern] Inline acceleration in storage or network controller.** Also works well and is widely used. Already part of block and net subsystems in kernel, not crypto
  3. **[Legacy] Off-CPU crypto engine that returns output to caller.** Not used much in Linux anymore. Hard to use efficiently, error-prone, and usually slower than CPU in practice — sometimes by a lot. The traditional crypto API focuses on this approach.
- Kernel features that really want to support approach (3) can still use traditional crypto API. It's not going away yet. Just keep in mind that in practice it tends to be harmful, not helpful.

[Extra] What about hardware-bound keys?

- The traditional crypto API has limited support for hardware-bound keys (e.g. "paes")
  - When used correctly, raw keys not stored in memory
  - Can provide a small security benefit in limited situations
  - Never actually used explicitly in the kernel. May be used implicitly, e.g. dm-crypt
  - Most in-kernel crypto users have copies of their keys in memory anyway
  - Only a few Linux drivers actually support this feature
  - *Hardware-wrapped inline encryption keys* already provide similar benefits
- Support for this could be added to the crypto library if ever needed, especially if/when it's supported by the CPU (e.g. x86 Key Locker & RISC-V High Assurance Crypto). So far it just hasn't made sense because it already wasn't being used

[Extra] Migrating from AF_ALG

AF_ALG use case	Replacement
Call CPU-based crypto code	Userspace crypto code. Already the standard solution in almost every program. It's more efficient, too (if done properly).
Protect data or keys from arbitrary reads in process	Isolated userspace process. Already the standard solution, e.g. ssh-agent. Also, PROT_NONE suffices to protect not-in-use data
Reduce memory footprint	Usually N/A, as userspace crypto lib is usually present anyway and AF_ALG just bloats the kernel size. When applicable, can be achieved in better way with improved / modular userspace lib
Access standalone hardware crypto engine	Usually N/A. But when still used in this way, migrate to CPU-based acceleration. In rare cases where this is still needed, optimized driver specific interfaces (several already exist) work much better

[Extra] What about FIPS 140-3?

- Library works fine with kernel's existing (partial) support for FIPS 140-3:
  - `lib/` has had crypto code since 2005. Not new, just more widely used now
  - I've added FIPS 140-3 self-tests
  - `lib/` doesn't block non-FIPS algorithms from being used, but callers can (and do) just check `fips_enabled` themselves. Much more flexible
- Out-of-tree loadable FIPS modules that override the library functions continue to be unsupported upstream, since no such module is upstream. Downstream patching can continue to be used if/where this is needed

[Extra] Traditional crypto API: CRC-32C

```
static inline u32 ext4_chksum(struct ext4_sb_info *sbi, u32 crc,
                              const void *address, unsigned int length)
{
        struct {
                struct shash_desc shash;
                char ctx[4];
        } desc;

        BUG_ON(crypto_shash_descsize(sbi->s_chksum_driver)!=sizeof(desc.ctx));

        desc.shash.tfm = sbi->s_chksum_driver;
        *(u32 *)desc.ctx = crc;

        BUG_ON(crypto_shash_update(&desc.shash, address, length));

        return *(u32 *)desc.ctx;
}
```

[Extra] Library example: CRC-32C

```
static inline u32 ext4_chksum(u32 crc, const void *address, unsigned int length)
{
        return crc32c(crc, address, length);
}
```

- `crc32c()` is technically not new, but it became practically usable in v6.14 when I made it reliably use the architecture-optimized CRC code
- Again, faster than traditional crypto API, especially if length is short. (ext4 often calls this with length = 2, 4, or 8.) Just calls CRC code directly.