PUREE: Password-based Uniform-Random-Equivalent Encryption

Jay Sullivan

June 4, 2020

Abstract

PUREE is a disk encryption header format which provides the ability to password-protect disk devices, using full-disk encryption, in such a way that the entire disk is indistinguishable from random data. Notably, this occurs without the need to store any associated data on a separate disk. The format is simple, secure, and extensible. This paper specifies the on-disk header format and algorithms behind the format.

1 Introduction

PUREE provides the ability to perform full-disk encryption in a way that ensures that it is impossible (in practice) for an eavesdropper to determine which of the following is true:

1. Your disk is encrypted, or
2. Your disk has been "wiped" with random bits.

For example, the most common disk encryption format used on Linux systems is LUKS (Linux Unified Key Setup) [1]. If you were to look at the first 512 bytes of a disk encrypted with LUKS, you would see something like this:

```
00000000 4c 55 4b 53 ba be 00 01 61 65 73 00 00 00 00 00 |LUKS....aes.....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000020 00 00 00 00 00 00 00 00 78 74 73 2d 70 6c 61 69 |........xts-plai|
00000030 6e 36 34 00 00 00 00 00 00 00 00 00 00 00 00 00 |n64.............|
00000040 00 00 00 00 00 00 00 00 73 68 61 32 35 36 00 00 |........sha256..|
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000060 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00 20 |............... |
00000070 6c e3 94 7b 01 3b cb b7 7d 0e 23 47 b3 46 10 1e |l..{.;..}.#G.F..|
00000080 1b a3 1e 63 14 66 1b 2b e2 a5 45 ad d8 d9 fc 65 |...c.f.(..E....e|
00000090 38 f9 dd 9f 10 77 30 e1 03 7d 2c 5f 47 d5 2e ef |8....w...}..G...|
000000a0 bc 88 4d 1b 00 02 bf 4b 33 63 62 39 64 39 33 30 |..M....K3cb9d930|
000000b0 2d 33 34 61 38 2d 34 36 35 64 2d 38 34 37 32 2d |-34a8-465d-8472-|
000000c0 30 35 32 65 34 63 39 66 31 32 32 38 00 00 00 00 |052e4c9f1228....|
000000d0 00 ac 71 f3 00 2b f4 be b5 12 a9 fc e7 1a 57 9d |..q..+........W.|
000000e0 2f f5 16 cc 38 ce 8a b7 70 86 b5 3a c4 0d 28 be |/...8...p..:..(.|
000000f0 68 b4 8c d9 30 54 be 6b 00 00 00 08 00 00 0f a0 |h...0T.k........|
00000100 00 00 de ad 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000120 00 00 00 00 00 00 00 00 00 00 01 08 00 00 0f a0 |................|
00000130 00 00 de ad 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000150 00 00 00 00 00 00 00 00 00 00 02 08 00 00 0f a0 |................|
00000160 00 00 de ad 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000180 00 00 00 00 00 00 00 00 00 00 03 08 00 00 0f a0 |................|
00000190 00 00 de ad 00 00 00 00 00 00 00 00 00 00 00 00 |................|
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
000001b0 00 00 00 00 00 00 00 00 00 00 04 08 00 00 0f a0 |................|
000001c0 00 00 de ad 00 00 00 00 00 00 00 00 00 00 00 00 |................|
000001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
000001e0 00 00 00 00 00 00 00 00 00 00 05 08 00 00 0f a0 |................|
000001f0 00 00 de ad 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

Clearly, it is trivial to detect that a disk is encrypted with LUKS. In fact, this isn't just a problem with LUKS; it is a problem with *most* of the popular full-disk encryption formats. [1][2][3][4]

To be fair, some tools do support support completely-random-looking disk layouts, but in most cases, they either:

1. Are key-based (e.g., require a 128-bit or 256-bit key) rather than password based, in which case, the key must stored elsewhere. *(Where do you store the key?)*

2. Ask the user to store a (non-random-looking) disk-encryption header elsewhere (i.e., "detached header mode"). *(Where do you store the header?)*

This is an unfortunate situation, because it rules out plausible deniability: if someone interrogates you about data stored on your disk, it will at least be obvious to them that there is in fact data on the disk—the only thing between them and the data is convincing you to give them your password.

There have been several other attempts at creating plausibly deniable disk encryption systems, but most of these tools are either obscure/poorly documented, closed-source [5], or use outdated cryptographic ciphers [6], or don't completely randomize the disk [7].

In contrast, PUREE solves the plausible deniability problem by simply guaranteeing that all data on the disk, including the header, looks exactly as though it has been wiped. Since it has long been common practice to wipe disks by overwriting with random data [8], PUREE introduces the plausibility that an encrypted disk may actually have been wiped and therefore contain no information. For example, a PUREE header might look like this:

```
00000000 54 28 89 8f f0 f3 8e 9d 44 37 e3 6c 3f be d0 bc |T(......D7.l?...|
00000010 8a 56 fb e0 9c 2f fd 50 fc 25 f4 20 b6 ed c4 3e |.V.../.P.%. ...>|
00000020 f6 7d 40 63 51 68 01 3b af b9 59 8f 5a 34 e6 c1 |.}@cQh.;..Y.Z4..|
00000030 2d d8 19 b4 51 fd 16 a3 6f db c0 5a e3 d4 ee 7c |-...Q...o..Z...||
00000040 a4 df 25 e9 69 03 f9 9c c1 03 4d a2 1a 25 19 b0 |..%.i.....M..%..|
00000050 14 54 59 c1 35 51 33 6a c3 15 60 57 b2 58 1d 3d |.TY.5Q3j...W.X.=|
00000060 ef 92 96 c4 f7 dd 9c 0b f6 cc 66 90 b0 09 64 20 |..........f...d |
00000070 ac fd d8 ec af b0 b3 09 07 e2 b4 2b 03 3a 80 1a |...........+.:..|
00000080 71 fa 31 99 0a c9 c1 4d 44 6f 54 2d 3c 83 dd 74 |q.1....MDoT-<..t|
00000090 7c 6d c1 b9 56 a7 8d 49 7e c9 91 11 a5 e5 8a d1 ||m..V..I~.......|
000000a0 54 3e 66 99 d4 ae 77 f6 67 5d 3f c3 a2 fd 87 1e |T>f...w.g]?.....|
000000b0 3d 04 91 23 1c 2d 18 71 42 c9 97 72 98 3a ba c4 |=..#.-.qB..r.:..|
000000c0 33 c8 3c 9f f5 b9 76 b8 57 e6 33 22 79 bc 56 53 |3.<...v.W.3"y.VS|
000000d0 47 62 b7 4a bb d7 a0 55 d6 ed 14 3a 3d 51 44 6f |Gb.J...U...:=QDo|
000000e0 16 db 10 47 18 78 01 65 04 4c 43 48 1f cb 2b a7 |...G.x.e.LCH..+.|
000000f0 f4 78 a0 4a d9 70 93 89 d7 85 48 47 f7 4f 65 0a |.x.J.p....HG.Oe.|
00000100 85 aa 99 e3 da 85 c7 7c 0f 89 db cc fc 1d 0f d0 |.......|........|
00000110 74 62 e3 31 9b 94 f0 bf 64 36 fa 67 f7 ef 82 5e |tb.1....d6.g...^|
00000120 3d b8 ae 78 b1 f7 5d 61 dd 46 6a dc c3 dd af da |=..x..]a.Fj.....|
00000130 da c5 54 df 44 43 c4 5e c9 34 fd a6 57 6e f8 c7 |..T.DC.^.4..Wn..|
00000140 17 cd 9d 42 6c f4 96 d9 09 b8 56 b4 27 78 ab 70 |...Bl.....V.'x.p|
00000150 6e fd 4d 4e 02 e0 b3 ff 4c 76 bf 29 4e 0c 20 09 |n.MN....Lv.)N. .|
00000160 d9 9b e3 5b 1d 75 b8 70 94 68 5d 7b 84 c7 12 15 |...[.u.p.h]{....|
00000170 b2 b2 ed d3 00 c0 f2 55 67 a6 71 fd 01 99 6c 31 |.......Ug.q...l1|
00000180 52 2b 67 a0 8c 3c e3 16 a2 01 13 34 e9 da f3 b7 |R+g..<.....4....|
00000190 65 6f d4 09 ba 10 34 61 01 4c f4 72 26 e8 7c 48 |eo....4a.L.r&.|H|
000001a0 71 8a ce 22 94 d0 81 fc bc 7d f1 17 73 58 b1 e9 |q..".....}..sX..|
000001b0 c0 84 8e 2f 04 c9 b8 48 f6 89 f1 0d ce b1 1b 73 |.../...H.......s|
000001c0 90 fb a8 df 16 31 48 d0 72 08 c2 e1 08 a9 c5 1a |.....1H.r.......|
000001d0 25 06 23 81 14 b5 c2 93 bf 8e bf 24 6f 6a b1 d2 |%.#........$oj..|
000001e0 d6 69 f7 f4 8b a0 95 90 e4 94 72 44 01 96 c1 ab |.i........rD....|
000001f0 de ab 09 8f a0 fc 74 c0 f4 3a 85 01 d1 6a 11 01 |......t..:...j..|
```

Despite being indistinguishable from random to an eavesdropper, there is in fact a structure to the disk layout—only observable if the password is known—which will be explained next.

2 PUREE Header Format

The complete PUREE header format is, essentially:

```
struct Header {
    byte[24]     salt
    byte[16]     box_1_auth
    byte[8]      box_1_subspec_id
    byte[0|1]    box_1_used_slots
    byte[0|1]    box_1_total_slots
    byte[0|2]    box_1_len_of_box_2
    byte[...]    slots       /* variable length */
    byte[16]     box_2_auth
    byte[...]    box_2_data  /* variable length */
    byte[...]    padding     /* pad with random bytes */
    byte[1MiB—32KiB]  anti_forensic_region
}
```

...where:

- "`byte[x]`" means "x bytes in size"
- "`a..b`" means "between a and b (inclusive)"
- "`a|b`" means "either a or b"
- "`...`" means "any number"

In other words, the PUREE header is composed of the following main elements:

1. `salt` - 24 random bytes
2. `box1` - An encrypted+authenticated chunk of data.
3. `slots` - A variable-sized list of password slots. (optional)
4. `box2` - A second encrypted+authenticated chunk of data. (optional)
5. `padding` - Random bytes filling the first 64 sectors.
6. `anti_forensic_region` - An anti-forensic region; ≈1MiB random bytes.

As might be inferred by the optionality of some elements, the PUREE disk format is “polymorphic,” in two senses:

1. PUREE has a total of three “*meta-parameters*” (discussed below) which affect the PUREE header’s layout on disk. When decrypting the disk, the meta-parameters are unknown (due to PUREE’s indistinguishable-from-random format), and thus all permutations of the meta-parameters must be attempted when decrypting the disk.
2. The format is extensible via the *sub-specification* system. This enables the support of a (possibly infinite) number of different disk-encryption formats. (Discussed later, in section “Sub-specifications”).

The three meta-parameters, all boolean, are, namely:

- `CipherHasParams` - The chosen cipher suite has additional tunable parameters (i.e., “`box2`”).
- `DontStretchPassword` - Don’t stretch the password hash (w/ `argon2id`)—just hash once.
- `UseAntiForensicRegion` - Include an anti-forensic region.

The significance of these is that, since there are 3 of them, and since there are two possible values of each (`true` or `false`), then, up to $2^3 = 8$ different decryption attempts must be made to before successfully decrypting the PUREE header.

The high-level pseudo-code for creating the header, assuming 0 password slots, for now, is:

```
byte[...] password = /* ... */
//------Meta-Parameters------
bool UseAntiForensicRegion = /* ... */
bool DontStretchPassword       = /* ... */
bool CipherHasParams           = /* ... */
//------Box1 Parameters------
byte[8] subspec_id = /* ... */
uint8 used_slots = 0  /* 8-bit unsigned integer */
uint8 total_slots = 0 /* 8-bit unsigned integer */
uint16 len_of_box2 = ... /* 16-bit big-endian unsigned integer */
//------Calculated Fields------
byte[24] salt = UseAntiForensicRegion ? blake2b(anti_forensic_region)
                                      : csprng()
byte[32] pwhash = DontStretchPassword ? blake2b(salt || password)
                                      : argon2id(password, salt)
byte[32] box_key = pwhash /* or, if we use slots, find in a password slot */
//------Disk Format------
if CipherHasParams or num_slots>0:   /* || means concatenate */
    byte[16+12] box_1 = chacha20poly1305(key=box_key, nonce=1, msg={
                            subspec_id ||  num_slots || len_of_box2
                        })
    byte[16+len_of_box2] box_2 = chacha20poly1305(key=box_key, nonce=2, msg={
                                     ...
                                 })
else:
    box_1[16+8] = chacha20poly1305(key=box_key, nonce=1, msg={
                      cipher_suite_guid
                  })
if UseAntiForensicRegion:
    byte[1MiB-32KiB] antiForensic = csprng()
```

The algorithm will now be explained in detail:

As a pre-requisite, we must decide on the meta-parameters, box parameters, and password. The `subspec_id` field must also be decided: this is a 64-bit unique identifier of the specific algorithm to use to encrypt the rest of the disk (described further in section "*Sub-specifications*").

The first step is to generate a `salt`. In the simplest case, this will be 24 random bytes. However, PUREE also supports an optional "anti-forensic" feature: when `UseAntiForensicRegion` is enabled, random data will be written to the disk at byte offsets $[2^{15}..2^{20} - 1]$, which can later be hashed to derive the salt (described further in section "*Anti-forensic region*").

Next, we calculate the password hash `pwhash`. Usually, this will be the result of running the password and salt through the `argon2id` [9] key derivation function. (Regarding how PUREE chooses `argon2id` hash parameters, see section "*The parameter character*".) If `DontStretchPassword` is enabled—which one might use, for example, in the case of a already-very-high-entropy password—the hash will simply be the `blake2b` [10] hash of the password. Assuming we're not using password slots, we'll define `box_key` (used next) as being equal to `pwhash`.

Next, we need to encrypt+authenticate the rest of the header information, which we divide into two chunks of data, resulting in `box1` and `box2`. The contents of both boxes are encrypted with `chacha20poly1305` (i.e., encryption with `chacha20`, authentication with `poly1305`) [11] [12], using the `box_key` as the key, and for the nonce, we use *1* and *2*, respectively, for `box1` and `box2`.

The reason the header information is divided into two boxes is as follows: when decrypting the disk, we need a fixed set of bytes which we can decrypt+verify, but, at the same time, we somehow want to allow variable-size arguments to some of our encryption algorithms. The solution is a trade-off between the two: `box1` is fixed-size, but contains information which tells us the format (`subspec_id`) and length in bytes of (`len_of_box2`) the second box.

The format of `box1` actually has two possibilities, dependening upon whether `CipherHasParams` is set and whether we're using password slots.

1. If `CipherHasParams` = `true` (which, again, means we won't have a `box2`) and we don't have any password slots, then obviously we don't need to keep track of the size of `box2` or the number of password slots. The only information we'll need to put into `box1` then is the `subspec_id`, which, depending on its value, specifies all remaining rules for how to format the disk.

2. If `CipherHasParams` = `false` is set, or if we're using password slots, then, we'll place 4 more bytes into `box1`: 2 bytes to specify how big `box2` is (`len_of_box2`) and 2 bytes to count how many slots we have.

The format of `box2` can also vary, and in fact its size is unrestricted. Its length will be specified in `len_of_box2`, and its format is dependent upon the PUREE sub-specification identified by `subspec_id` (which we will learn more about in section "*Sub-specifications*").

Finally, after all PUREE header elements are written to disk, the rest of the header is filled with random bytes—*padding*—up to the next multiple of 512 bytes. (And, if there is an anti-forensic region, all bytes between it and the header are also padded with random bytes.)

3 The parameter character

PUREE promises something which at first sounds impossible: password-based disk encryption which allows varying password-hashing parameters, without storing the password-hashing parameters onto the disk (or elsewhere). As with anything that sounds too good to be true, there's a catch: *PUREE stores the password-hashing parameters in the password itself.*

With PUREE, when you encrypt or decrypt a disk, you'll be asked for a password. The password will be hashed (along with the `salt`) using the `argon2id` [9] function. (Or, when ‘a’ is the parameter char, simply `blake2b`—see below.) But, this function requires three parameters:

1. Parallelism: the maximum number of parallel CPU threads
2. Memory: the amount of RAM required
3. Iterations: multiplier on amount of time required

Since you need to know these parameters as a prerequisite to decrypting a disk, and since a PUREE disk looks completely random prior to decryption, these parameters obviously can't come from the disk. So, PUREE mandates that when you choose a password, you must add an extra character at the beginning of the password—the *parameter character*—which fully describes which password-hashing parameters to use.[1]. Currently, valid parameter character values are:

<table>
<tr><th>Parameter Char</th><th>Parallelism</th><th>Memory</th><th>Iterations</th></tr>
<tr><td colspan="4">‘a’ ⇒ pwhash=blake2b(salt||password)</td></tr>
<tr><td>‘b’ ⇒</td><td>1</td><td>75MiB</td><td>1</td></tr>
<tr><td>‘c’ ⇒</td><td>1</td><td>250MiB</td><td>1</td></tr>
<tr><td>‘d’ ⇒</td><td>4</td><td>250MiB</td><td>4</td></tr>
<tr><td>‘e’ ⇒</td><td>1</td><td>1GiB</td><td>1</td></tr>
<tr><td>‘f’ ⇒</td><td>4</td><td>1GiB</td><td>4</td></tr>
<tr><td>‘g’ ⇒</td><td>1</td><td>4GiB</td><td>1</td></tr>
<tr><td>‘h’ ⇒</td><td>4</td><td>4GiB</td><td>4</td></tr>
<tr><td>‘i’ ⇒</td><td>1</td><td>16GiB</td><td>1</td></tr>
<tr><td>‘j’ ⇒</td><td>4</td><td>16GiB</td><td>4</td></tr>
</table>

As CPU, memory, and scaling requirements evolve in the world, new versions of PUREE will update this table with new characters (most likely of the form `[a-zA-Z0-9]`), based upon common server and desktop hardware requirements.

Some may argue that the requirement to remember one more character in the password corresponds to a loss of password entropy, because that character could have otherwise contributed to the entropy—specifically, a loss of about $\log_2(10+26+26) \approx 6$ bits. However, this isn't necessarily true. For example, if you choose one parameter character, you can likely use it for many years, for all of your passwords. The amount of entropy it really takes away is hard to measure, but is arguably negligible.

4 Sub-specifications

Previously, when we described the disk layout, we referred to “sub-specifications”—or “subspecs”—while avoiding how they really work. In this section, we'll explain why subspecs exist, and give a few examples.

We've already explained how to unlock `box1` and `box2`, but haven't elaborated on what's in `box2`, or happens next. The answer is that the rest of the behavior depends completely on the subspec: a subspec specifies both the structure of the `box2`, the structure of the rest of the disk, and the algorithm used to encrypt/decrypt the rest of the disk.

PUREE is extensible: anyone can define a subspec. We will now list a few examples:

[1] We've now discussed four different types of “parameters”: meta-parameters, box1 parameters, box2 parameters, and password-hashing parameters. The `parameter character` specifically refers to password-hashing parameters.

subspec-disk-dm-crypt	
subspec_id	`0x35225972d13cbd4b`

Parameters	`uint64    logical_start_sector // e.g., 2048`
	`uint64    num_sectors          // e.g., 2147481600`
	`byte[1]   cipher_name_bytes // e.g., 15`
	`byte[...] cipher_name      // e.g., 'aes-xts-plain64'`
	`uint16    disk_key_bytes  // e.g., 32`
	`byte[...] disk_key         // e.g., head -c 32 /dev/urandom`
	`uint64    iv_offset        // e.g., 0`
	`uint8     num_optional_params`
	`repeat {`
	`    byte[2]   optional_param_bytes`
	`    byte[...] optional_param`
	`}`

Specification	The disk will be encrypted with the given dm-crypt [13] parameters. All integers are stored in big-endian format: `uint8`, `uint16` and `uint64` represent 8-bit, 16-bit and 64-bit unsigned integers, respectively.

subspec-disk-aes128-cbc-essiv-sha256	
subspec_id	`0xf83789a7bf8f0e43`

Parameters	`byte[16] disk_key`
	`uint64 sector_start`
	`uint64 sector_count`

Specification	Each sector of the disk will be encrypted with AES-128 in CBC mode, using key `disk_key`, where the IV is generated with ESSIV-SHA-256. The first sector of the disk is considered sector 0.

subspec-disk-aes256-cbc-essiv-sha256	
subspec_id	`0x9abf8b191e4a84a4`

Parameters	`byte[256] disk_key`
	`uint64 sector_start`
	`uint64 sector_count`

Specification	Each sector of the disk will be encrypted with AES-256 in CBC mode, using key `disk_key`, where the IV is generated with ESSIV-SHA-256 [14].

subspec-disk-aes128-xts-plain64	
subspec_id	`0xa9d4d04dfaf36314`

Parameters	`byte[32] disk_key`
	`uint64 sector_start`
	`uint64 sector_count`

Specification	Each sector of the disk will be encrypted with AES-128 in XTS [15] mode, using key `disk_key`.

subspec-disk-aes256-xts-plain64	
subspec_id	`0xcf43556cf0b3ebb7`

Parameters	`byte[64] disk_key`
	`uint64 sector_start`
	`uint64 sector_count`

Specification	The disk will be encrypted with AES-256 in XTS [15] mode, using key `disk_key`.

subspec-disk-stream	
subspec_id	`0xc8d5d141300c1414`

Parameters	`uint64 cipher_id`
	`byte[32] disk_key`

Specification	The disk will be encrypted, without authentication, using the stream cipher identified by `cipher_id`. Valid `cipher_id`s are: 0 ⇒ salsa20 1 ⇒ salsa12 2 ⇒ salsa8 3 ⇒ chacha20 4 ⇒ chacha12 5 ⇒ chacha8 The cipher will use `disk_key` as the key, and will use sector index (where sector index 0 is the first sector of the disk) as the nonce.

subspec-file-chacha20poly1305	
subspec_id	`0x3439e4a3f151844a`
CipherHasParams	`false`
Specification	• This subspec is suited for encrypting files, rather than disks. • There are no cipher parameters, so there is no `box2`. The total header overhead is exactly 64 bytes (`salt`:24, `box_1_auth`:16, `box_1_subspec_id`:8, `box2_auth`:16). • The `salt` equals `blake2b(random[24]||plaintext)`. • The entire file is encrypted with `xchacha20poly1305`(*key* = `pwhash`, *nonce* = `salt`). • The authenticator is stored in `box2_auth`.

subspec-file-chacha12poly1305	
subspec_id	`0x7963be7d6aedf9de`
CipherHasParams	`false`
Specification	Similar to `subspec-file-chacha20poly1305`, except using the `chacha12` [11] cipher instead of `chacha20`.

5 Password slots

Password slots (which, so far, we've ignored) allow a disk to be unlocked by more than one password: if there are 255 password slots (the maximum that PUREE supports), then the disk can be unlocked with up to 255 different passwords.

Until now, we've assumed that, when decrypting a disk, the `box_key` (which is used to open `box1` and `box2`) is derived from the `pwhash`, where `pwhash = argon2id(password,salt)`. When we use password slots, however, we do things differently: instead, we treat the `pwhash` as a "slot key" and use it to encrypt the `box_key` (along with a `slot_checksum`, as explained below) using a random nonce. The structure of a slot is:

```
struct Slot {
    byte[24]   slot_nonce
    byte[16]   slot_auth
    byte[32]   box_key
    uint64     header_sector
    byte[16]   checksum
}
```

...where `box_key`, `header_sector`, and `checksum` are encrypted+authenticated with `xchachapoly1305` using `key=pwhash`, and `nonce=slot_nonce`.

If there are multiple slots, each slot contains the same exact `box_key`, but each slot uses a random (and unique) 24-byte `slot_nonce`, and, assuming the passwords differ, a different `pwhash`.

When the header is first formatted, the user may choose to allow anywhere from 0 to 255 *total* slots. At any point in time later, they may freely choose to use, or not use, any of those slots (i.e., add or remove passwords) but must reformat the header in order to change the *total* slot count.

Interestingly, because the number of slots is variable, and because the `box2` must be placed physically *after* the slots, it means `box2` is not always found at a fixed location. In order to calculate the byte offset of `box2`, one must take into account the `total_slots` field, found inside `box1`, and calculate accordingly.

Going further: before decrypting `box1`, the number of `total_slots` and `used_slots` fields are unknown. This means that *all 255 slot locations may need to be attempted* before unlocking a disk. In practice, however, most situations will require few, if any, slots and therefore implementations may choose to only consider the first few slot locations.

The `header_sector` field specifies which sector should be used to find `box_1` and `box_2`. For more information on this, see section "*Subvolumes*".

Finally, within each slot is a `checksum`. This is essentially a has of the entire header, thus providing the ability for each password owner to detect whether any other part of the header has changed, including modifying existing, or adding new slots. This checksum (which is encrypted and therefore hidden from other users) can be calculated by hashing as many header fields as possible:

```
checksum = blake2b(box_key || box1_subspecid
    || box_1_used_slots || box_1_total_slots || box1_len_of_box2
    || myslot.slot_nonce    || myslot.slot       // Pre-encryption
    || slot[0].slot_nonce   || slot[0].box_key   // Post-encryption
    || slot[1].slot_nonce   || slot[1].box_key   // Post-encryption
    || slot[...].slot_nonce || slot[...].box_key // Post-encryption
    || box_2_data)
```

6 Subvolumes

The typical way to use PUREE is to store a 1MiB header+anti-forensic-layer at the beginning and end of the device, with all sectors in between treated as the 'actual' volume, with the understanding than one or more passwords may unlock that header.

But, PUREE also allows the optional ability for each password to specify an *alternate* header located at *any arbitrary location* on the disk—i.e., a subvolume— while guaranteeing that an eavesdropper can't prove that the subvolume exists, or even that a password points to it.

This is done as follows: when a password slot is unlocked, one of the fields is the `header_sector` field. Usually, this equals 0, as the primary header (i.e., the PUREE header at the beginning of the disk) is normally stored in the first sector. However, it can point arbitrarily to any sector on the disk: if it points to, let's say, sector 1024000, and there is a valid PUREE header at that location, then that PUREE header will be used instead of the primary.

Alternate headers are just like the primary, except that instead of simply using `1` and `2` as the nonces for `box1` and `box2`, the values `header_sector+1` and `header_sector+2` will be used, respectively. (The `salt` and password slots of the secondary header will be ignored.)

7 Anti-forensic region

The anti-forensic region, if it is used, consists of 1MiB - 32KiB = 1015808 bytes (i.e., 1984 sectors) worth of random data, beginning at byte offset $2^{15} = 32768$ and ending at offset $2^{20} - 1$. If it is used, then the 24-byte `salt` of the first sector is ignored, and instead, the `blake2b` hash of the entire anti-forensic region is used as the salt.

The purpose of an anti-forensic region is essentially to "stretch" the `salt` across a large number of disk sectors. This solves two (very paranoid) concerns:

1. If the header sector is wiped, in some cases, it may be possible that the key is in fact still recoverable. [8]
2. If the header sector becomes inaccessible (e.g., due to hardware failure) its data cannot be wiped despite the information still remaining embedded within the disk hardware. [14]

Regardless of whether the anti-forensic region is used, the suggested behavior of PUREE is to treat the offset at 1MiB as first sector to place actual data. However, sub-specifications are free to overrule this suggestion.)

8 Backup and shadow headers

One concern with full-disk encryption is that, if the header somehow becomes corrupted or overwritten, then the key used to encrypt the disk will be lost, in which case all of the data on the disk would be unrecoverable.

A naive solution is to simply copy the header and place it onto some backup disk. However, this inherently destroys PUREE's indistinguishable-from-random feature: an attacker who inspects both disks can reasonably conclude that one disk is a backup of the other, that most likely this was done intentionally, and therefore that this most likely contains encrypted information.

The more cautious solution is to back up the header as follows: create a new header from scratch, using the same parameters and same password, but different salt. This way, the backup appears completely random to an eavesdropper, yet still, if the original copy is destroyed, the backup can be seamlessly copied back into its place. (It should be made clear that, at the time of creating such a backup, all passwords, including for all password slots, must be available to be provided, if they're still to be used.)

The suggested behavior of PUREE is to keep a backup at the end of (i.e, the final 1MiB of) the disk, which we refer to as the *shadow header* The shadow header is simply a backup header at a different location on the same disk, updated in tandem with the primary header. (However, sub-specifications are free to overrule this suggestion.)

Ideally, multiple shadow headers should be stored throughout the disk, but in practice, the ability to store them is at the mercy of the sub-specification algorithm's ability to ignore (i.e., not map) the 1-MiB byte arrays located at these regions of memory, and so is left up to sub-specifications to implement this behavior. If additional shadow headers are used, their suggested locations are at offsets (where possible) at the sectors starting at offsets 128MiB, 128GiB, and 128TiB.

9 Command-line tool suite

We present a command-line utility, `puree`, which is available at:

`https://github.com/notfed/puree`.

This tool provides the ability to format, and mount, disks with PUREE. The code is public domain.

Note that, while the PUREE disk format is system-independent and extensible, the `puree` tool currently (as of the time of this writing) only supports the Linux operating system, leveraging Linux's `dm-crypt` [13] disk encryption device-mapper target.

References

[1] Milan Broz. LUKS2 on-disk format specification version 1.0.0.

[2] Nitin Kumar and Vipin Kumar. Bitlocker and Windows Vista, 2008.

[3] Appelbaum Jacob and Weinmann Ralf-Philipp. Unlocking FileVault: an analysis of Apple's disk encryption system. In *23rd chaos communication congress, Berlin; December*, 2006.

[4] Milan Broz. dm-crypt: Linux kernel device-mapper crypto target, 2015.

[5] Mick Bauer. Paranoid penguin: Bestcrypt: cross-platform filesystem encryption. *Linux Journal*, 2002(98):9, 2002.

[6] J Assange, RP Weinmann, and S Dreyfus. Rubberhose filesystem. *Archive available at: http://web.archive.org/web/20120716034441/http://marutukku.org*, 2001.

[7] Andrew D McDonald and Markus G Kuhn. StegFS: A steganographic file system for Linux. In *International Workshop on Information Hiding*, pages 463–477. Springer, 1999.

[8] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, pages 77–89, 1996.

[9] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications. 2015.

[10] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[11] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5, 2008.

[12] Daniel J Bernstein. Cryptography in NaCl. *Networking and Cryptography library*, 3:385, 2009.

[13] Christophe Saout. dm-crypt: a device-mapper crypto target, 2007. *URL http://www. saout. de/misc/dm-crypt*, 2014.

[14] Clemens Fruhwirth. *New methods in hard disk encryption.* na, 2005.

[15] Luther Martin. XTS: A mode of AES for encrypting hard disks. *IEEE Security & Privacy*, 8(3):68–69, 2010.