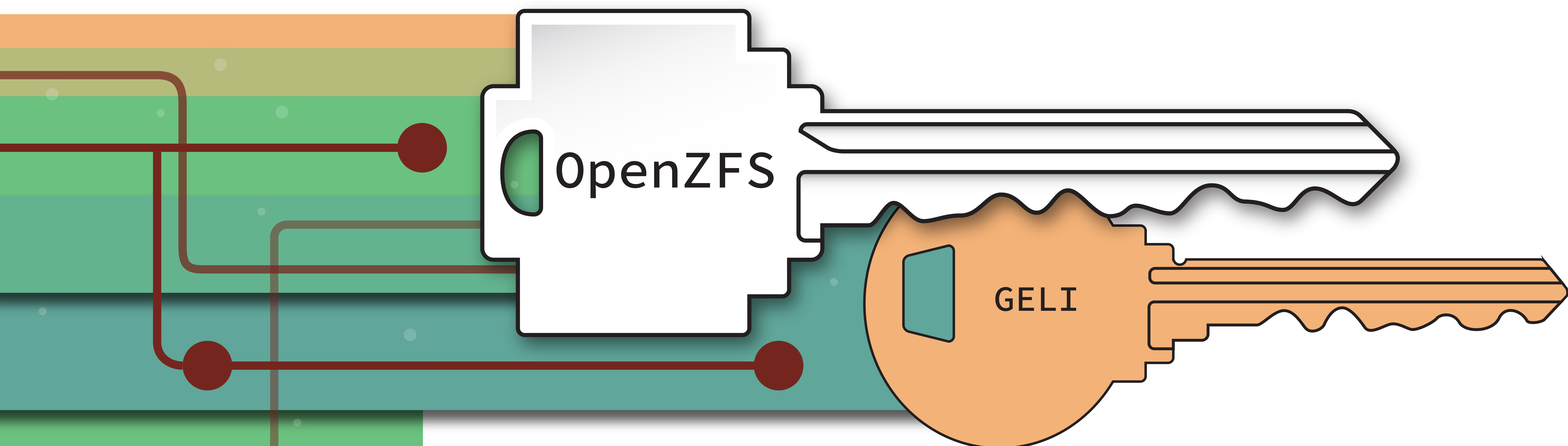


OpenZFS Encryption Arrives on FreeBSD

BY ALLAN JUDE AND KYLE KNEISL

Among the new features arriving in OpenZFS on FreeBSD is a clever implementation of industry standard encryption at the dataset level. Effectively, this means that the FreeBSD storage encryption ecosystem previously dominated by GELI now has a new contender: OpenZFS "native encryption." In this article, we discuss how it compares to GELI from both a technical and usability perspective, equipping you to evaluate which of these encryption solutions you will prefer (spoiler: the new one, in most cases).

GELI

GELI has been FreeBSD's main native full disk encryption system for some time. GELI uses AES (typically AES-256 in either the XTS or CBC modes, which are particularly suited for encryption of disk data), paired with SHA-256 hashing (the same hashing that Bitcoin is based upon) and message authentication. To do this, GELI leverages FreeBSD's `crypto(9)` subsystem to provide hardware offloading, thereby greatly accelerating performance on systems that support AES-NI or otherwise utilize cryptographic acceleration (basically, any up-to-date platform).

However, GELI *has absolutely nothing to do with ZFS*. GELI encrypts an entire device, or at finer granularity, a partition on it. Thus, before first use of any filesystem protected in this way, the relevant devices or partitions have to be "unlocked." Once unlocked, the underlying filesystem is recognized and mounted, typically for the duration of the uptime. Unlocking procedures are governed by a master key and up to two user keys that variously involve keyfiles and passphrases. This complex set of keys allows for the possibility of re-keying a GELI partition (e.g., in the case of compromise of one key) without having to re-encrypt data.

Integrated in the FreeBSD bootstrap code, there is thus only a small `freebsd-boot` partition that remains unencrypted, containing just enough code to prompt for the GELI key passphrase, decrypt the boot partition, and read the bootloader or kernel from the underlying ZFS (or UFS) filesystem. This means that the entire filesystem (along with every byte of data on the relevant drives) is encrypted, except for the small chunk of startup code.

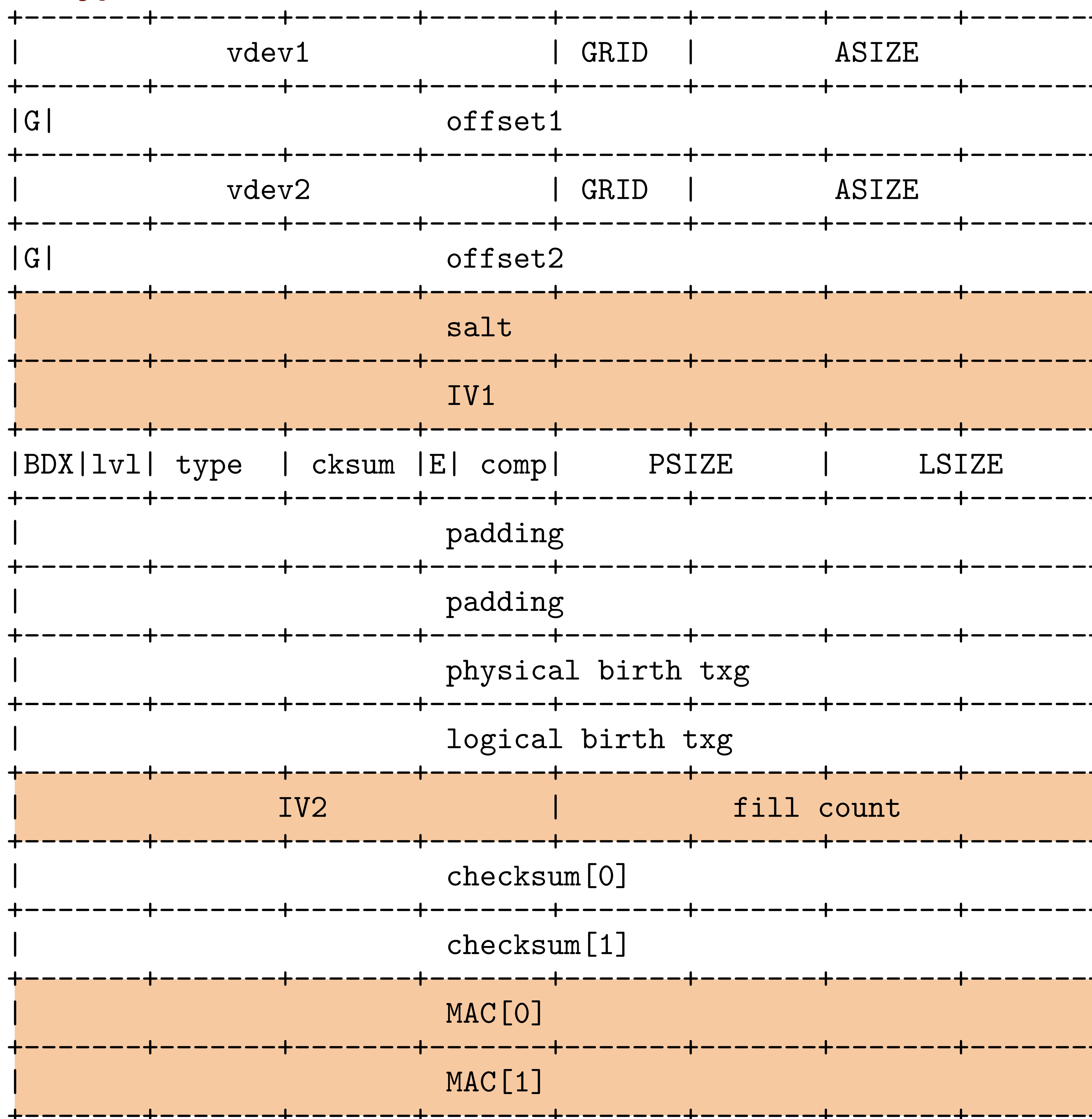
Properly implemented by the system administrator, GELI uses strong encryption, in strong modes, rotates data keys before they are overused, and has any other number of modern security practice elements in its implementation. Accordingly, GELI is, and will remain, a secure encryption implementation using best practices in all key areas.

Regular Block Pointer

```
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev1                | GRID  |         ASIZE         |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                         offset1                               |
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev2                | GRID  |         ASIZE         |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                         offset2                               |
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev3                | GRID  |         ASIZE         |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                         offset3                               |
+-------+-------+-------+-------+-------+-------+-------+-------+
|BDX|lvl| type  | cksum |E| comp|      PSIZE      |     LSIZE     |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          padding                                |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          padding                                |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          physical birth txg                     |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          logical birth txg                      |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          fill count                             |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          checksum[0]                            |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          checksum[1]                            |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          checksum[2]                            |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          checksum[3]                            |
+-------+-------+-------+-------+-------+-------+-------+-------+
```

```
/*
 * Legend:
 *
 * vdev          virtual device ID
 * offset        offset into virtual device
 * LSIZE         logical size
 * PSIZE         physical size (after compression)
 * ASIZE         allocated size (including RAID-Z parity and padding)
 * GRID          RAID-Z layout information (reserved for future use)
 * cksum         checksum function
 * comp          compression function
 * G             gang block indicator
 * B             byteorder (endianness)
 * D             dedup
 * X             encryption
 * E             blkptr_t contains embedded data
 * lvl           level of indirection
 * type          DMU object type
 * phys birth    txg when dva[0] was written; zero if same as logical birth txg
 *               note that typically all the dva's would be written in this
 *               txg, but they could be different if they were moved by
 *               device removal.
 * log. birth    transaction group in which the block was logically born
 * fill count    number of non-zero blocks under this bp
 * checksum[4]   256-bit checksum of the data this bp describes
 */
```

Encrypted Block Pointer



```
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev1              | GRID  |         ASIZE         |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                           offset1                           |
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev2              | GRID  |         ASIZE         |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                           offset2                           |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                             salt                              |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                             IV1                               |
+-------+-------+-------+-------+-------+-------+-------+-------+
|BDX|lvl| type  | cksum |E| comp|     PSIZE     |     LSIZE     |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                           padding                             |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                           padding                             |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                      physical birth txg                       |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                      logical birth txg                        |
+-------+-------+-------+-------+-------+-------+-------+-------+
|             IV2               |            fill count         |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          checksum[0]                          |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          checksum[1]                          |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          MAC[0]                               |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                          MAC[1]                               |
+-------+-------+-------+-------+-------+-------+-------+-------+
```

```
/*
 * Legend:
 *
 * salt          Salt for generating encryption keys
 * IV1           First 64 bits of 96-bit encryption IV
 * X             Block requires encryption handling (set to 1)
 * E             blkptr_t contains embedded data (set to 0, see below)
 * fill count    number of non-zero blocks under this bp (truncated to 32 bits)
 * IV2           Last 32 bits of encryption IV
 * checksum[2]   128-bit checksum of the data this bp describes
 * MAC[2]        128-bit message authentication code for this data
 */
```

OpenZFS Native Encryption

By mid-2019, ZFS-on-Linux (ZoL) 0.8 introduced native encryption support; that is, encryption of ZFS datasets themselves on supported Linux platforms. Thanks to the efforts of a number of people from ZoL and FreeBSD, this important capability will soon be realized for FreeBSD in version 13, expected around the summer of 2020.

At the level of a filesystem encryption solution, we do not encrypt entire devices as in GELI. This is already immensely compelling; it is common to only desire encryption on a subset of sensitive data, and that data is probably naturally segregated into its own dataset as it is! Acting entirely within a dataset, native encryption is realized as what feels like just another ZFS dataset property. Amazingly, the fundamental layout of ZFS's data structures need not be expanded to accommodate encryption! Native encryption is accomplished through a clever re-

purposing of slack space existing in the structure of ZFS block pointers. In particular, those familiar with some of the less frequently used ZFS dataset properties have surely encountered the `copies` parameter, which by default is set to 1, but may also optionally be set to 2 (rarely) or 3 (almost never). As it happens, the space in the data structure pointing to the third copy (relevant only in the almost-never-used case of `copies=3`) is of the correct size to implement an AES-based encryption strategy. So, for the price of losing the rarely needed ability to store data blocks in triplicate (plus parity!), we gain filesystem encryption. Like other dataset properties, this property is inherited by child datasets. In fact, the only way in which the encryption property is notably different from other familiar ZFS dataset properties is that the use of encryption must (understandably) be set at creation time and cannot be unset. Accordingly, the ZFS data and metadata within the dataset are encrypted while at rest until mounted, but the existence of the dataset, and the ZFS properties for the dataset (e.g., the `logicalused` and other properties that might reveal the size and scope of data contained within) are necessarily always visible in the operating system.

One of the most game-changing features of ZFS native encryption is that, surprisingly, the dataset may be scrubbed or resilvered *while it is unmounted* and hence still encrypted. This unexpected feat is accomplished by splitting what was once a 256-bit hash field into a still more-than-ample 128-bit hash juxtaposed with a 128-bit message authentication code (MAC). This means that a ZFS administrator, unlike in the case of GELI, need not have access to client decryption keys in order to perform ZFS maintenance tasks—quite a compelling feature indeed. Furthermore, when sensitive data need not be accessed for a time, the dataset can be routinely unmounted where it subsequently becomes encrypted-at-rest, inaccessible to those without the key; this is not easily possible with GELI where typically an entire filesystem is decrypted at boot time and remains decrypted during the entire system uptime. Additionally, with GELI, the entire filesystem is protected by one set of keys, giving little granularity for access to different users; a user can either access the entire system, or none of it. With Native ZFS dataset encryption, different datasets can easily be given distinct keys. Those keys may then be provided to a given user in whatever access combination is appropriate.

At the same time ZFS native encryption is being introduced, ZFS `send` will now have the ability to send "raw" data blocks; that is to say, blocks are not interpreted in any way, and are sent as-is to the receiving system. In the case of natively encrypted datasets, this means that the key does not need to be loaded and data blocks can remain encrypted while in transit over the network, even when that transit (by accident or design) is itself not encrypted. This allows sensitive data to be backed up to a remote ZFS system without the remote system or its administrator (and, as has already been said, the local administrator) ever having access to the plaintext. This is a significant security posture improvement over GELI-based solutions.

(n.b.: the FreeBSD boot code does not yet support booting from an encrypted dataset, but it is likely that even that will be possible in the future!)

Getting Your Hands on OpenZFS 2.0

At the time of writing, it will likely be a few months before OpenZFS 2.0 is released and then integrated into FreeBSD's source tree. For those unwilling to wait, you can install the pre-release version of OpenZFS on FreeBSD 12.1 (and later) using the `sysutils/openzfs` package. If not using a `-RELEASE`, it is best to compile `sysutils/openzfs` and `sysutils/openzfs-kmod` from ports (because the module needs to exactly match the kernel it will run against). Once installed, replace `zfs_load="YES"` with `openzfs_load="YES"` in `/boot/loader.conf`. The port or package will install to the standard ports prefix, `/usr/local`, so the PATH environment variable may need adjustment to ensure the shell finds `/usr/local/sbin/zfs` before the sys-

tem-provided `/sbin/zfs`. Once a pool contains any encrypted dataset, the pool itself (including its unencrypted datasets) can no longer be imported on a system that does not support the encryption feature flag. However, as is often the case with new ZFS features, once the last dataset using the encryption feature is destroyed, the encryption feature flag will revert from "active" to "enabled" and the pool will once again be able to be imported by older versions of ZFS. Reminder: pools created with OpenZFS 2.0 will enable new features that may not be easily backward-compatible, and thus this type of experimentation should be done cautiously. The `zpool` man page describes how to selectively disable individual feature flags at pool creation time.

Transitioning a Dataset to OpenZFS Native Encryption

To transition data from an existing unencrypted dataset (remember: GELI-encrypted devices contain unencrypted datasets at the filesystem layer, so this applies to GELI pools too!), start by creating a new dataset with OpenZFS native encryption enabled:

```
zfs create -o encryption=aes-256-gcm -o keyformat=passphrase poolname/newdataset
```

Then, create a snapshot of the original dataset and use ZFS replication to copy its contents to the new encrypted dataset. Once complete, you are free to destroy the original dataset, rename the encrypted dataset to the original name, and update its mountpoint. Don't forget to consider the security ramifications of the resulting unallocated space which may be expected to contain unencrypted versions of your data; you will probably want to overwrite this space. The `zpool initialize` command is one reasonable way to accomplish this.

If you are transitioning from GELI, recall that GELI resides in a layer preceding the filesystem and thus does not interact with the filesystem. Accordingly, the only way to remove GELI is for each GELI-encrypted device (or partition) to rematerialize as an unencrypted device. Fortunately, ZFS itself can be our partner in accomplishing this. For example, a sufficiently redundant pool can strategically leverage a sequence of `geli kill` and `zpool replace` commands, removing the GELI layer from each device in turn. For pools with good levels of ZFS redundancy, this is relatively easy and can be safely done in-place. For pools with insufficient or no redundancy, the procedure will require (or be more safely performed with) a temporary, appropriately sized helper disk.

Conclusion

GELI is great and has been with us a long time. As it encrypts entire devices (including the operating systems on them in most cases), it is certainly the way to go, for example, when your aim is to protect every byte on a stolen (or confiscated) laptop or an RMA'd disk. However, it provides little granularity as a security solution, and there is considerable risk in having "encrypted" data in an accessible state for the entire uptime of a system. For example, a hacker with endpoint access to devices mounted under the GELI regime will in principle have full access to all data.

OpenZFS native encryption, on the other hand, gives the administrator more control over the granularity of what is encrypted and what combinations of users might access it. It also allows individual encrypted datasets to be unmounted when not in use, where they are protected "at rest," while the remainder of the system remains accessible. This is more sensible than GELI for file servers that hold sensitive data whose uptimes can extend into months or potentially years. A hacker with unfettered endpoint access (either physical or network) would see what an unprivileged user would see of the dataset when it is unmounted. The only disadvantage to using

OpenZFS native encryption (as it currently stands) is that it necessarily leaks certain information about the nature of the data it encrypts: that the dataset exists in the first place, the name of the dataset, the names of any snapshots, how much data there is, when it may have been created, how compressible it was, and so on. Typically speaking, this type of information tends to be of little consequence—the fact that a server houses, for example, health records of 4,000 patients, or surveillance videos, is of little sensitivity, but the actual contents of those records and videos would be. It is also very hard to argue against any encryption solution that requires fewer system administrators (as opposed to content owners) be in a position to access data.

It seems difficult to envision a use case (beyond the aforementioned stolen laptop) where GELI makes as much sense and is as usable and scalable as OpenZFS native encryption. While GELI will certainly remain with us as a trusted and reliable encryption solution for devices, the elegance of a ZFS-based encryption solution granular to individual datasets is extremely compelling and likely to become a fixture of ZFS user expectations moving forward.

ALLAN JUDE is VP of Engineering at Klara Inc., a global FreeBSD Professional Services and Support company. He also hosts the premier weekly BSD podcast, BSDNow.tv, and was elected to the FreeBSD Core team in 2016 and 2018. He is the co-author of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* with Michael W Lucas.

KYLE “DRKK” KNEISL holds a PhD in mathematics and works as a scientist in the Washington, DC, area. He has been an active member and advocate of the FreeBSD and FreeNAS communities since 2013.