Booting from Encrypted Disks on FreeBSD

Allan Jude, *ScaleEngine Inc.* allanjude@freebsd.org

Abstract

FreeBSD has supported full disk encryption with GBDE and GELI since 2002 and 2005 respectively. However, booting the system required storing the loader and kernel unencrypted so that the requisite GEOM module could be loaded to handle decryption. This becomes a significantly larger challenge with the introduction of the ZFS file system, as having multiple separate partitions detracts from the advantages of ZFS, and also causes headaches when performing operating system upgrades. With the availability of ZFS Boot Environments, a solution that allowed the kernel and loader to remain part of the primary file system was desired. It was also desirable for it to support encryption. This paper provides an overview of the design of the GELI enabled boot code and loader, as well as the numerous challenges encountered during their development.

1. Introduction

The GEOM[1] framework provides an infrastructure in which "classes" can perform transformations on disk I/O requests as they transit from the application to the device drivers and back.

One such GEOM class is GEOM_ELI, or GELI[2], which supports multiple modes of AES[3], as well as a number of other ciphers. As blocks pass through the GELI class they are encrypted or decrypted. Additionally, GELI also supports optional authentication, where the content of the block is verified with an HMAC[4].

The FreeBSD boot process on the i386 and amd64 platforms is rather involved, and for historic reasons is performed in many individual steps[5]. This is further complicated by the fact that there are three supported partitioning schemes on these platforms: MBR, BSD, and GPT. This work has yet to be extended to cover UEFI.

1.1 Booting MBR

On disks formatted with the MBR[6] partitioning scheme, the BIOS[7] reads the first 512 bytes of the boot disk and runs the boot code that resides there. In FreeBSD this code is referred to as `boot0`. This code then finds the active slice, and reads the first 512 bytes of it, called `boot1`. That is then run, and in turn reads the first sixteen 512 byte sectors of the slice, which contain the `boot1` + `boot2` programs. `boot2` is the first of the programs that is not limited to a single 512 byte sector, but instead is limited to 7680 bytes (15 of the 16 reserved sectors at the start of the UFS slice, the first being `boot1`). `boot2` is a small program with just enough knowledge of the UFS file system to find `/boot/loader` and a kernel, and to interactively allow the user to select a different partition from which to read the kernel. `boot2` reads `/boot/loader` (also known as `boot3`) from the UFS partition and executes it. The loader is a fully interactive program that draws a menu and allows the user to control the startup of the system. The loader presents an interactive menu, and then proceeds to load the kernel and executes it, and the operating system is started.

For ZFS, the procedure is slightly different, `boot0` loads a different `boot1` file, which finds the 64KB `zfsboot` (analogous to `boot2`) at an offset of 1MB into the ZFS partition, in a predetermined blank space in the ZFS on-disk format. The `zfsboot` code understands the ZFS file system and loads `/boot/zfsloader`, which provides an interactive menu similar to that of `/boot/loader` for UFS, and can then load the kernel and boot the OS.

Support for booting encrypted MBR drives is not provided due to these size constraints.

1.2 Booting GPT

GPT[8] partitioned disks are booted in a more straightforward manner. A GPT partitioned disk still contains a dummy MBR, called a PMBR or

Protective MBR. The purpose of the PMBR is to keep other operating systems from misidentifying the disk as unpartitioned, and offering to reformat the disk for the user. The PMBR starts the boot process in a very similar fashion to an MBR, with a single 512 byte sector that is read and executed. With GPT, a dedicated partition is provided to contain the remaining bootcode, analogous to `boot1+boot2`. The PMBR reads up to 545KB from this boot partition and executes it. This limit is specific to the FreeBSD PMBR boot block implementation and could be changed if required. Users can install one of a number of bootcodes to this partition, the most common being `gptboot` and `gptzfsboot`. These boot blocks consist of two parts, `gptldr`, a small assembly program that relocates the code loaded from disk to the correct memory address and executes it, and `gptboot` or `zfsboot` respectively, analogous to `boot2`. `gptboot` loads and launches `/boot/loader` from a UFS formatted disk, while `gptzfsboot` loads and launches `/boot/zfsloader` from a ZFS pool.

1.3 Motivation

The main goal of this work was to allow ZFS Boot Environments to be used in combination with full disk encryption. This was not previously possible because booting from an encrypted disk required the bootloader and kernel reside on a separate unencrypted partition. Having the bootloader and kernel outside of the boot environment defeated the primary purpose of boot environments, to allow different versions of the OS to be installed concurrently. Having the kernel outside of the root file system also complicated upgrades.

2. Investigation Stage

For an initial implementation, a copy of `gptzfsboot` was made with the name `gptgeliboot`. The original idea was to create a single bootcode that could boot from an encrypted UFS or ZFS file system. Subtle differences in bits of the `boot2` code, and no clear way to define the behaviour of systems that use both UFS and ZFS file systems, caused this approach to be abandoned. It was decided to implement optional GELI support in each of the existing GPT bootcodes instead.

Initially, it was necessary to determine if the system boot partition was GELI encrypted. As with most all GEOM classes, GELI stores its metadata in the very last sector of the provider, which is usually a partition, to avoid conflicting with the backup copy of the GPT partition table that is stored in the last sector of the disk. The task seemed simple, read the partition table, identify the starting LBA of the partition and its size, and read the last sector of that partition. This turned out to not always be so simple. The function that reads from the disk, `drvread()`, takes a `struct dsk` as a parameter, which has a 'start' member that may be set to the beginning of the partition, which makes the offset parameter relative to the start of the partition, rather than to the start of the disk, but it isn't always set.

After reading through `sys/boot/i386/zfsboot/zfsboot.c`, it was discovered that ZFS might just have made this easier; rather than reading from the disk directly, `vdev_probe()` takes a pointer to a function that would read from the disk, and a pointer to an opaque structure that will be passed to that function to identify the disk. A duplicate version of the existing callback function, `vdev_read()`, was created that would pass the disk blocks through the GELI decrypt function before returning them. This approach would later be replaced with a check to see if the partition being read was GELI encrypted, to avoid the code duplication.

The hardest part about working in the bootcode is that there are no error reporting facilities. There isn't even a `panic()`. Pretty much all there is to work with is `printf()`, and when things go bad, the system just hangs, unless you manage to crash the BTX loader, which will give you a dump of the assembly instruction pointers and the like. This made development very iterative and almost brute force. Make a change, build, install it, reboot, fail, add printf, build, install, reboot, fail, repeat. Of course, you must moderate the quantity of printfs, because there is no pager; once data scrolls off the top of the screen, it is gone forever.

3. Initial Implementation

After the partition has been determined to contain encrypted data, the metadata needs to be read to determine which algorithm to use to decrypt it. Then the encrypted copy of the master key must be decrypted with the user provided password, and only then is it possible to read from the partition. To proceed, an implementation of AES simple enough to be used in the bootcode was required. GELI itself uses the kernel crypto framework, or openssl if run in userland. Neither of these are an option in the bootcode. Some initial google searching turned up a small AES-CBC 128 implementation with an acceptable license. This was dropped into the bootcode, along with some copy/pasted functions from GELI itself, to decode the metadata, decrypt the master key, verify it is valid, and take care of tasks such as determining the sector encryption key, and the unpredictable IV.

Now the rest of the dependencies needed to be solved. Elsewhere in the bootcode, when existing functionality is required, it is often just directly included into the source code using the preprocessor #include directive. GELI uses both SHA256, for generating the unpredictable IV for AES-CBC, and SHA512 for the various HMACs. The first method attempted was just directly including the `sha256c.c` and `sha512c.c` files from the kernel crypto implementation, but the SHA256 and SHA512 implementations declare different macros using the same name, but with different content. The next approach was to add them to libstand32, the stripped down version of libstand that is used in the bootcode. Eventually this approach was modified to create a separate libgeliboot, with most of the bits required for GELI in the bootcode, to avoid bloating libstand32 or the bootcodes that depend on it. But this approach, isolating GELI in the bootcode, brought its own set of obstacles that made it not always work.

The initial versions of this work statically defined the password as "test" to avoid the perceived complexity of creating an input prompt in the bootcode. This turned out to be a fairly trivial undertaking, although that didn't stop it from eventually being re-implemented, twice. The first rendition duplicated the `getstr()` implementation from `sys/boot/i386/common/cons.c` and changed it to echo an asterisk, rather than the character that was typed. Not only was this probably unnecessary code duplication, but the loader did not actually use that file and instead used `sys/boot/common/console.c` which didn't have a `getstr()` implementation. Worse, the `getstr()` from `cons.c` wouldn't work, because it didn't use `getchar()` but its own `xgetc()`. On top of all of this, in later review it was determined that the `getstr()` implementation in `cons.c` has a bug in it, where the echoing of the character is not inside the conditional statement that checks that the number of characters entered has not exceeded the requested limit, and that buggy version has been copied into a number of different files, including the i386 `boot2.c`. Cleaning that up is future work. In the end, a modified version of `ngets()` from libstand (originally from NetBSD) was used to create `pwgets()` which was placed in the newly created `libgeliboot.a`.

4. First Roadblock

So now it appeared we had everything required to do a first boot from an encrypted ZFS partition. The system would load `boot0`, which would read the new `gptzfsboot` from the freebsd-boot partition. The `boot1` part of that code would then relocate `boot2` to the correct location in memory, and run it. `boot2` would start, taste the partition, determine it was GELI encrypted, read the master key, prompt the user for the password, decrypt it with the passphrase, and stand ready to determine the sector key and decrypt each block as needed.

The bootcode was then installed in a virtual machine with a GELI encrypted disk. The first attempt to boot the VM in VirtualBox didn't go so well. What is a triple fault anyway? And why does it cause VirtualBox to crash? Attempts to boot on real hardware caused a continuous reboot loop.

It turns out that when `gptldr.S` was written, the author uttered the immortal words "$X ought to be enough for anyone". The entire point of ZFS was to do away with such arbitrary limits that will be "good enough for now". The `gptldr` is not part of ZFS, so we mustn't fault its author for this seemingly arbitrary limit. The assembly code used to boot the system is 16 bit, so copying more than 64KB of data at once is more complicated. Up until this point, no bootcode was really in danger of hitting this

limit, gptboot for UFS was not even 16KB, and gptzfsboot was only 42KB. However, now gptzfsboot has grown an AES implementation, both SHA256 and SHA512, and the important bits of GELI, leaving it on the heavy side of 90KB, and the work was not done yet. It turns out trying to boot with only the first two-thirds of the bootcode causes crazy things to happen. This would turn out to be a hard problem to solve, so it was decided to work around it in the meantime.

5. Booting from encrypted UFS

Work then started on implementing GELI in gptboot, for booting from encrypted UFS. The original size of gptboot (under 16KB vs gptzfsboot's 42KB bulk), meant there was more room for the new code to still remain under the 64KB limit. The code to read from UFS is quite a bit different from that of ZFS, but again the function responsible for reading from the disk was identified, and could be modified to GELI decrypt the blocks on their way from the disk to the UFS code. Of course, the GELI decrypt function that had just been written was designed to have the same prototype as the callback that ZFS used to read the disk. The easiest solution was to just adapt the UFS code to use the same prototype, and just not use the extra parameters.

Predictably, it was not that simple. In the very limited environment of the bootcode, there is no kernel yet, or libc, or even malloc(). So where does memory come from? Well, luckily zfsboot needs a bit of memory, so has an implementation of a very limited malloc() that consists of a cursor pointing to an address on the heap, and each malloc() would just increase that pointer by the amount requested, until the limit is reached, which can be as little as 3MB. The obvious downside to this simple approach is that there is no free(), so one needs to be very careful to avoid making a large number of allocations. This environment is also severely limited on which C library calls can be used; only a small subset of libstand is available. This also makes code reuse more difficult, as many of the standard .h files try to pull in string.h or other headers, which conflicts with stand.h.

At this point, after a bit of fiddling to get the decryption working properly, the first successful boot from an encrypted file system was achieved. Of course, successful is a bit of an overstatement at this point. gptboot had loaded /boot/loader which immediately errored out because it was unable to find a kernel, because the filesystem was full of encrypted gibberish, not a UFS filesystem. The loader still needed to be taught about GELI.

Finding the spot in the loader where data could be intercepted as it flows from the disk to the file system is a bit more involved. The first path that seems to present itself was to make additional "filesystems", geli_ufs and geli_zfs, adding those to the struct the loader uses to interpret the different filesystems. Eventually after mentally resolving much indirection, the author found the biosdisk part of libi386 that is the bottom of the indirection chain. When a disk is opened, we taste it for GELI, and if it is GELI, all reads are decrypted before they are passed back up the chain.

Now libi386 can transparently decrypt the sectors as they are read from the disk, and the loader is able to read the kernel. At this point the author had the first successful boot of a UFS encrypted file system, into multi-user mode. Of course, at this point it only worked if your GELI password was "test" and if you had used AES-CBC with only a 128bit key. Of course, the original goal was to be able to boot from an encrypted ZFS pool. The author was only working on the UFS bootcode and loader to prove the concept, in hopes of eventually solving the limitations of gptldr. The supposedly impossible task now seemed pretty much done, or so was thought.

6. Breaking The Limits

The author tried just about everything to shrink the new GELI enabled bootcode: marking almost every function as static, ripping out every unused byte, trying to optimize manually, and with the compiler (-Os rarely seems to actually save many bytes, -O2 turns out to be bigger than -O1). This approach obviously wasn't going to work, the gptldr assembly problem was going to need to be solved.

First, the naive approach was tried, modifying gptldr.S and changing the variable that controlled the number of 512 byte sectors to relocate in memory from 0x80 (128) to 0x100 (256).

However, when the compiler digested this, it laughed and set the variable back to 0x80. “This is 16 bit assembly, what do you think you are doing”. So then the author tried making the bootcode 32 bit, changing all of the %si to %esi etc, but that just crashed it even faster. So the author approached one of his mentors, Eitan Adler, for a bit of help with the assembly. At first Eitan seemed receptive to making the minor changes required to either make the assembly 32bit, or make it copy two blocks of 64KB instead of just one. It turned out he was too busy to spend much time on it, understandable. Next, the author asked John-Mark Gurney, who looked at the assembly, and again initially seemed receptive to doing the bit of work on it. However once he started reading the original code and understood the scope of the problem, he was quick to suggest seeking help from someone else. So the author approached John Baldwin, the original author of `gptldr`. Baldwin's suggestion was to find a different way to solve the problem, like keeping the loader unencrypted in a small partition or some other similar hack. Then at vBSDCon Peter Grehan heard of the authors troubles and offered to help, but again, after coming to understand the scope of the problem, was quick to demonstrate how to use gdb attached to qemu to debug why the previous attempts to modify `gptldr` were crashing. A friend of the author, Dylan Cochran, was also at vBSDCon and made a number of valiant attempts to rewrite the assembly, but didn't manage to finish it during the conference, and was quickly distracted by work once back at home. Then finally, at the EuroBSDCon 2015 Developers Summit in Stockholm, Sweden, Colin Percival heard about the apparently unsolvable assembly dilemma, and said something to the effect of "16 bit assembly, I know this". It turns out that 16 bit assembly was the last assembly Colin had even dabbled with, and he was intrigued by the problem. Colin came up with a few draft patches during the conference hacker lounges, but they too just crashed the BTX loader. Again the author’s hopes were dashed. A few days after returning from Sweden, Colin contacted the author via IRC and presented another version of the assembly patch. This one didn't copy any more bytes than the current one, but did the copy in Colin's new extensible way. It worked for copying the normal bootcode. A revision or two later, and now there was a `gptldr.S` that could copy a variable number of 32KB blocks. Work could finally progress.

7. Supporting Additional Ciphers

So now, we can boot ZFS as well, because our over-sized bootcode is now perfectly valid, and no longer gets truncated. The next step was to actually support encryption modes that people might actually want to use. Why not just reuse the AES-XTS implementation that is already in FreeBSD as part of the opencrypto framework? Including `sys/opencrypto/xform.c` pulls in EVERY supported algorithm, and their necessary dependencies. The bootcode could now be larger, but it was still preferable to keep it small, as it must fit in the freebsd-boot partitions existing users already have. The first approach was to copy/paste just the bits required to get AES-XTS to work into a new file. This created `geliboot_opencrypto.c`, with a copy of the AES-XTS implementation, and `geliboot_aes.c` with a copy of the required files from sys/crypto/rijndael/, and finally parts of GELI scattered in `geliboot.c` and `geliboot_hmac.c`. It was rather messy, but after much trial and error, and looking at endless streams of printf's of before/after decryption of each sector and comparing those to printf's added to the real GELI module, it worked. Now encrypted boot worked for both file systems, and it supported both AES-CBC and AES-XTS with either 128bit or 256 bit keys.

The next task was to clean it up and get it into a shape that might actually be accepted by the FreeBSD project. Again back to trying to reuse more code and stop with the copy pasta. The first and most obvious target was the AES algorithm itself. Can we just #include it instead? The first problem to solve is the conflicts from `string.h`, which turns out to be fairly easy to cheat our way around. Just define _STRING_H_ and when `rijndael-alg-fst.c` goes to include the standard header it thinks it has already done so, and skips the content of the `string.h` file. So now `geliboot.h` just includes `rijndael-alg-fst.c` and `rijndael-api.c` and we're on our way. Of course, this approach didn't work for AES-XTS, because `xform.c` provides every algorithm, including Blowfish, Camellia, CAST5, DES, 3DES, Skipjack, and a few others, plus a set of HMAC algorithms for every hashing algorithm under the sun, and deflate compression. In the meantime, our local copy/paste of AES-XTS had been modified to avoid the need for a `malloc()`,

and to pass the raw context struct around, rather than pointers to it. After discussion with the FreeBSD Security Team, the author was told they could break up `xform.c` if need be, so the work started in earnest. In an attempt to make the patch easier to review, svn copy was used to make a copy of `xform.c` for each family of algorithms. Then all of the other algorithms were removed from all but one of the copies. The resulting diff showed many removed lines, making the diff rather large and probably harder to review, but it also showed that not many lines of code had actually changed. It is not clear if this approach turned out to be that beneficial in the end. The refactoring took a few iterations to get right, especially deciding what code went in the .h files versus the .c files. The first, incorrect, approach, was to remove the 'static' qualifier from the definitions of the functions so that the `aes_xts_decrypt()` symbol would be exported so it could be used in the bootcode. Later it was explained that the way opencrypto works is that a struct for each algorithm is exported with pointers to the setkey, encrypt, decrypt, reinit, and zerokey functions, and that this should be used instead. The other issue was that opencrypto was designed for use in the kernel, so included some kernel headers. These were easily ifdef'd out of the way, but opencrypto also used the kernel's `malloc()` facility, which takes 3 parameters, rather than the usual 1. The kernel malloc expects a previously defined class for each memory consumer, as well as a series of flags. To work around this `xform_userland.h` was created to define macros for `malloc()` and `free()` to strip off these extra parameters when used in userland. In the end, opencrypto cleaned up rather nicely. After review by the FreeBSD Security Officer, Xin Li, and George Neville-Neil, the newly broken up version of opencrypto was committed to the FreeBSD base system in the last few days of 2015. Then it came time to try to reuse more of GELI, and remove the excessive copy and paste. A lot of the GELI code had survived unmodified, except for one parameter, the `g_eli_softc`, the struct used to track the internal running state of each instance of the GELI module. In the local version of the code, `geliboot` had passed around the much smaller GELI metadata struct. With a bit of refactoring, all of the callers of the GELI functions in the bootcode were updated to use the `g_eli_softc` struct.

GELI itself required some modifications to support its code being reused outside of the kernel. A number of ifdef's were added to mask chunks of code specific to the kernel. A number of macros and structs were moved from .c files to `g_eli.h` as they were depended upon in the bootcode. In the case of the HMAC functions, they were extracted to a separate file to separate them from the kernel specific crypto implementation. After the refactor, only a single function from GELI needed to be reimplemented locally in `geliboot`. g_eli_crypto_decrypt was made to use the kernel's AES implementation and opencrypto's XTS implementation, as the version in GELI used the kernel crypto framework or OpenSSL.

8. Password Caching

At this point, the system can be booted, but it prompts the user for the encryption passphrase an inordinate number of times. The test system was a ZFS mirror of two disks. `gptzfsboot` would prompt for the password for each of the two disks, then load the loader. The loader would then prompt the user for the passphrase for each disk, and load the kernel. At the mountroot prompt, the kernel would prompt for the passphrase for each disk, and finally the system would boot. With more than a few disks, this quickly becomes exceedingly cumbersome. Colin Percival, Devin Teske, and Kris Moore had already been suffering from similar problems and developed a solution. Colin implemented the `kern.geom.eli.boot_passcache` sysctl, which caches the password entered by the user at the mountroot prompt and attempts to reuse it on each new disk that is tasted during the boot process. This was extended with Colin's help by Kris Moore, to allow the passphrase entered in the GRUB2 boot loader to be passed via the kernel's environment to GELI, so that if the password was correct, it would avoid re-prompting for the password at the mountroot phase. This avoided the issue where the mountroot password prompt would become buried by late device attach notices. Devin Teske added an option to `loader.conf`, `geom_eli_passphrase_prompt`, that would cause the loader to prompt the user for the GELI passphrase ahead of time, and pass it to the kernel via the environment, the same way PC-BSD's GRUB2 was doing it, again with the goal of avoiding the mountroot prompt. Extra care was taken by the GELI

kernel module to zero the passphrase from the environment before single user mode starts.

The obvious approach was to make the loader pass the passphrase to the kernel using the existing mechanism. In addition, the password prompt that was implemented in the loader for its new GELI support was also given a similar caching mechanism whereby it automatically tries the previously entered passphrase, and only if that fails, gives the user 3 attempts to enter the correct passphrase. The number of password prompts to boot the system was reduced from 6 to 3.

The same "try the previously entered passphrase" mechanism was then implemented in `gptboot` and `gptzfsboot`, reducing the number of password prompts to 2, assuming the user used the same passphrase for each disk. Then came the question, how to pass the passphrase from the `boot2` stage, to the loader. The answer lay in `gptzfsboot`, where a flag is set, `KARGS_FLAGS_EXTARG`, which tells the loader to look for an additional argument after the end of the regular set of arguments. Here, it will find `struct zfs_boot_args`, containing information such as the pool that is being booted, and the root filesystem. The first member of this struct is 'size', which is set to `sizeof(struct zfs_boot_args)`. This allows the loader to safely access newer members of the struct, by first checking that the `offsetof()` the member is not greater than the `sizeof()` the loader's definition of the struct. This allows mismatched versions of the bootcode and loader to continue to work together. When a new member is added to the end of the struct, access to it is guarded by this mechanism. Using this design pattern, a new member was added to the `zfs_boot_args` struct to pass the GELI passphrase from `boot2` to the loader. For UFS support, the `KARGS_FLAGS_EXTARG` flag was previously unused. Following the same pattern again, the flag is set, and a new `struct ufs_boot_args` was created. At this time it only contains the GELI passphrase cache, but it too starts with the size member so it could be extended in the same fashion.

9. Conclusions

After assessing how complicated it is to boot the x86 architecture, one may question whether it is worth increasing the complexity to incorporate disk encryption. However, a large number of users have expressed a desire for GELI encrypted root-on-ZFS. This project was quite a harrowing learning experience, but the end result is a working implementation of GELI in the bootcode and loader.

The remaining goal is to build a GELI enabled EFI loader in time for FreeBSD 11.0. The recently completed work to add ZFS to the EFI loader included modular filesystem support, making it easier to add new filesystems in the future. It is unclear at this time whether a generic GELI module could be used with all filesystems, or if GELI support will need to be added to each existing filesystem module.

Due to the space constraints, support for booting from encrypted disks partitioned with MBR does not seem feasible. With the author's previous work on bsdinstall, building workarounds for firmware bugs that impeded the adoption of GPT, it is hoped that use of MBR will rapidly decline.

When the work started, it was uncertain if it would ever accomplish anything. Each roadblock seemed like it might spell the end of the effort. The constrained environment of the bootcode made debugging tedious, often leaving little recourse other than spamming `printf()` and repeatedly cycling the virtual machine. As it turns out, almost anything can be accomplished with a large dose of persistence and sage advice from the experts.

10. Future Work

There is much to be done still, but the biggest remaining item is to add support for GELI key files. In an original setup created by bsdinstall's "zfs auto" mode, these files would sit on the unencrypted boot partition, alongside the kernel. However, the goal of this work was to remove the necessity for that unencrypted partition, but the key files are now left homeless. Some more advanced users might keep the keys on USB devices, but not all users will want to do this. Some possible approaches are:

- Storing the keys on a USB Device

    - Pro: Separation of the keys from the encrypted disk
    - Con: Your laptop will have to have this USB device hanging out of it each time it is booted
    - Con: how reliable is access to ephemeral USB disks during the bootcode?
    - Con: New devices cannot be attached during the bootcode
- Making a small UFS partition to keep the keys in
    - Pro: easy key management
    - Con: can be difficult to find space in an already partitioned system
- Storing the keys in a new raw partition type, freebsd-gelikey
    - Pro: Low space overhead, partition only needs to be a few bytes larger than the key, to store a header indicating the length of the key
    - Con: can be difficult to find space in an already partitioned system
    - Con: Key management is harder, will require a new utility to manage the keys

It may be worth adding additional cryptographic algorithms, like blowfish and camellia to the bootcode, but at this time AES seems most popular because of the performance advantage of AES-NI[9]. AES-NI is not used during the boot process, but the number of reads from the encrypted disk is quite low, so there is little performance to be gained.

There are some possible optimizations to GELI itself, in the case of AES-CBC, the unpredictable IV is generated using SHA256, while SHA512 (or SHA512t/256, a version of SHA512 with a different IV truncated to 256 bits), is ~50% faster on 64 bit hardware. For those with performance concerns, AES-XTS seems to be the more popular option anyway.

Support for GELI's sector authentication. This feature is not recommended for use by the original author of GELI anymore, so implementing support for it in the bootcode does not seem especially attractive at this time.

Another remaining item is to support systems booting with UEFI. When this work was stalled waiting on a solution to the 64KB bootcode limit, investigation of UEFI, where the bootcode is a file on an arbitrarily large FAT partition seemed attractive. Initial investigation for this work has not yet begun.

There is also a lot of cleanup that could be done across the various bootcode implementations, to extract common bits and unify them, and fix bugs that exist across all of the bootcodes, like the one in `getstr()`. Work to create a single bootcode that can boot both UFS and ZFS may still be of value, if a system for deciding which partition to boot from can be determined.

Acknowledgments

The author would like to thank Kris Moore, for his continued requests to see this work finished, Colin Percival, for solving the biggest roadblock in only a few hours, Baptiste Daroussin for mentoring me for my src commit bit, as well Xin Li, Pawel Jakub Dawidek, and Steve Hartland for reviewing the code. The author also wishes to thank the entire FreeBSD Community for their support and answering his endless questions, but especially Devin Teske, Michael Dexter, Marie Helene Kvello-Aune, Michael W Lucas, Eitan Adler, and Craig Rodrigues for their mentorship, friendship, and encouragement.

References:

[1] GEOM: https://www.freebsd.org/cgi/man.cgi?query=geom(4)
[2] GELI: https://www.freebsd.org/cgi/man.cgi?query=geli(8)
[3] AES: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf
[4] HMAC: http://tools.ietf.org/html/rfc2104
[5] FreeBSD x86 Boot Process: https://www.freebsd.org/doc/handbook/boot-introduction.html
[7] MBR: https://en.wikipedia.org/wiki/Master_boot_record
[8] BIOS: https://en.wikipedia.org/wiki/BIOShttps://en.wikipedia.org/wiki/BIOS
[8] GPT: http://www.uefi.org/sites/default/files/resources/UEFI%20Spec%202_6.pdf (Chapter 5)
[9] AES-NI: https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/