BentoCrypt

A Stacked Encrypted File System Implemented in Rust

Prathik Gowda
CS Department
Grinnell College
Bloomington, Illinois, USA
gowdapra@grinnell.edu
https://github.com/prathikgowda/BentoCrypt

***Abstract*—Memory safety and data encryption are crucial to the security and efficacy of modern file systems. However, most contemporary solutions are either implemented in C, which sacrifices memory safety, or lack file-based data encryption. *BentoCrypt* is an attempt to bridge this gap by providing a stacked, encrypted file-system implemented in the memory-safe *Rust* language - one of the first of its kind. It does this by leveraging the *Bento* [1] project: a new file systems development framework which empowers the user to write their kernel file system in entirely safe Rust.**

I. Introduction

Two of the biggest threats to the security of file systems are memory safety vulnerabilites and unencrypted data. For evidence of the catastrophic consequences that memory vulnerabilities can have, look to the *OpenSSL* library's Heartbleed bug. The Heartbleed bug allowed attackers to read private data past the scope of a valid request. This was due to a lack of proper bounds checking - an issue that would have been avoided if *OpenSSL* had been implemented in a memory-safe language. *BentoCrypt* avoids these memory vulnerabilities by implementing the file system in safe *Rust* using the *Bento* file systems development framework. *Bento* provides a FUSE-like (Filesystem in Userspace) API while also allowing your file system to be run at the kernel level. This leaves the issue of data encryption. Most file systems leave your data unencrypted on the disk. This means that if a hacker gains access to your hard-drive, there is nothing stopping them from reading your data. *BentoCrypt* addresses this by providing file-based data encryption - meaning that your sensitive data lies encrypted on the disk. We expect to provide the aforementioned memory safety and file-based data encryption while also being faster than FUSE-based equivalents like *gocryptfs* [2] due to speed benefits intrinsic to the *Bento* framework.

II. Related Work

We can categorize the related work into two categories: traditional file systems and cryptographic file systems. The traditional file systems include *ext4*, *BtrFS*, and *OpenZFS*. All of them are implemented in C (and thus are prone to memory exploits). *ext4* and *OpenZFS* now provide encryption, but *BtrFS* still does not.

In the category of cryptographic file systems, we have implementations such as *gocryptfs* [2], *eCryptfs* [3], and *EncFS* [4]. However, *gocryptfs* is the only one which provides a memory-safe implementation (in the Go language) and file-based data encryption. What differentiates *gocryptfs* from *BentoCrypt* is the level at which it runs. *gocryptfs* is implemented as a FUSE module which limits it to running in user space (decreasing performance by up to 83 percent [5]). Our implementation leverages the *Bento* development framework to run at the kernel level with access to kernel primitives which significantly increase performance.

III. Approach

BentoCrypt's goal is to provide file-based data encryption using Bento's safe framework. To achieve this goal, we designed three major components: *pwd* - our password helper program, *encryption* - our AES-GCM encryption library, and *xv6fs_encrypted* - our encrypted file system itself.

We chose AES-GCM encryption as our cryptographic algorithm, because it is authenticated (meaning that it verifies the integrity of our encrypted data), with proven security according to the concrete security model [6]. Another plus is the popularity of AES-GCM, which meant that there was already an implementation available to us written by the Rust crypto team. Typically, it is also faster than comparable encryption algorithms due to its usage of specialized CPU instructions, but this feature had to be disabled in our implementation due to development issues we discuss later in the paper.

A. pwd: The Password Helper Program

To achieve file-based encryption, we need a master key to encrypt and decrypt file data. For AES256-GCM encryption (our authenticated encryption algorithm of choice), our key must be 32-bytes long - which is prohibitively long for a user to remember naturally. Thus we need a scheme to store our master key securely so that it can not be retrieved by others. Our solution is to encrypt our master key using AES256-GCM encryption with a key derived from the user's encrypted password (retrieved from /etc/shadow). Since our master key will be encrypted, it can then be safely stored anywhere.

Thank you to the National Science Foundation (NSF) for generously funding this research.

Since we need root privileges to access /etc/shadow, we split this functionality into a separate password helper program (see Fig. 1. above) to minimize the amount of code run with sudo; thereby avoiding the security vulnerability of running all our code with elevated priveleges.

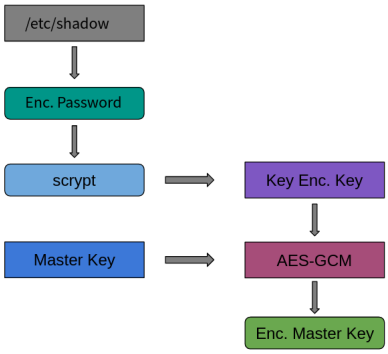

Fig. 1. Password Helper Diagram

Thus, we created *pwd*: an independent password helper program. *pwd* retrieves the current user's encrypted password from /etc/shadow and produces a key encryption key (a key to encrypt our master key) from it by using the key derivation algorithm *scrypt* [7]. Next, we create our master key using bytes from /dev/urandom and encrypt it using our key encryption key. Our solution prevents other users from accessing your data, since the key encryption key is seeded from the encrypted password which is unique to the appropriate user.

To retrieve the user's encrypted password we were forced to call unsafe C functions - as the /etc/shadow API has not been ported to Rust. Luckily, our design separtes pwd from our file system implementation - so the unsafe C code is confined to pwd itself.

You can access *pwd* here: https://github.com/prathikgowda/BentoCrypt/tree/main/pwd

B. *encryption: The AES-GCM Encryption Library*

Safely and correctly utilizing AES-GCM encryption is quite complex. Thus, instead of calling raw AES-GCM library functions directly from our encrypted file system, we created a standalone encryption library that provides a simpler wrapper around Rust's AES-GCM library. This also allows us to test our encryption functionality in isolation on arbitrary data - meaning that we don't have to compile and insert a kernel module every time we want to test encryption.

At the highest level, our API provides a function that takes an array of bytes and a masterkey, and returns the array of bytes encrypted with AES-GCM. Internally, our function generates a nonce using bytes from /dev/urandom and verifies the validity of our encrypted bytes before returning.

One issue that complicated development here is the resources available at the kernel level. While the userspace version of our file system is free to use the standard library, the kernel version does *not* have access to the standard library at all. Thus, this is where our project switched from userspace programming to kernel programming. We solved this issue by providing two versions of *encryption*: a kernel version and a userspace version. The kernel version has the feature [no_std] implemented, meaning that it will not compile with any standard library code. The userspace version doesn't have this feature, and thus is free to use comforts like Strings, Error types, and printing to stdout.

You can access *encryption* (kernel version) here: https://github.com/prathikgowda/BentoCrypt/tree/main/encryption_kernel

You can access *encryption* (userspace version) here: https://github.com/prathikgowda/BentoCrypt/tree/main/encryption_userspace

C. *xv6fs_encrypted: The Encrypted File System*

This brings us to the heart of the project: the file system itself. BentoCrypt is designed as a layer on-top of an existing file system (hence, a "stacked" file system). For our base, we chose the xv6 file system (xv6fs) included with the Bento framework. xv6fs is relatively simple (it has historically been used as a teaching tool for operating systems classes), but the Bento implementation also provides performance enhancements which make it faster than the original - making it a good candidate for our base.

Despite *Bento*'s ability to work at the kernel level, it still follows the FUSE API for the most part. Thus, like FUSE file systems, it provides data writing via the *write* function (here it's called *bento_write*) and data reading via the *read* function (here it's called *bento_read*). In order to keep our data encrypted on the disk, we want to call our encryption library to encrypt our data at the beginning of the *bento_write* function and decrypt our data at the beginning of the *bento_read* function.

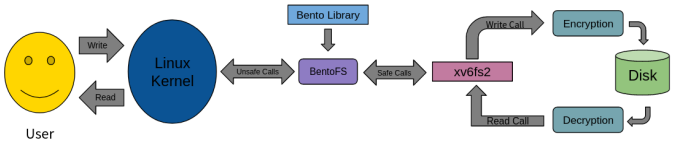

Fig. 2. BentoCrypt Diagram

You can access *xv6fs_encrypted* here: https://github.com/prathikgowda/BentoCrypt/tree/main/xv6fs_encrypted

IV. Development Pitfalls & The Current State of BentoCrypt

During the development of BentoCrypt we ran into several pitfalls which significantly hampered the speed of our progress. Ultimately, our project is unfinished. Here, I want to enumerate these difficulties, critique my mistakes, and devise strategies to overcome these obstacles in future projects. The goal is to learn and grow from my failures.

A. *Development Pitfalls*

1. My first pitfall was initially approaching this project as a userspace project. I knew that Bento's flexible framework

allowed for testing and debugging in userspace. However, in the development of the *encryption* library I relied too heavily on this feature and wrote the initial version using standard library resources that were unavailable when I switched to testing at the kernel level. This eventually forced me to rewrite the *encryption* library with no standard library code. But this could have been easily circumvented had I been testing at both the kernel and userspace level from the beginning.

2. Our second pitfall came during the development of the *encryption* library as well. To compile the *encryption* library for use at the kernel level, we needed to use the compilation target *x86_64-unknown-none-linuxkernel*. However, this target is not well supported by the Rust language. It's designated as a Tier 3 target - meaning that the Rust team does "not build or test automatically" and so it "may or may not work" [7]. While the *encryption* library compiled without a hitch with a native target, when we switched to the kernel target we quickly got an arcane LLVM error with no line numbers.

The source of the error was a bug in LLVM which arised when *sse* and *soft-float* target features were enabled at the same time. We were unable to fix the bug, which would require someone combing through the intermediate representation produced by LLVM, but we found a workaround: disabling *sse* and *avx2* target features. This unfortunately hurts the performance of our encryption library, but was our only option. If you're interested in this bug, then see the open issue we created on the Rust Github: https://github.com/rust-lang/rust/issues/87642

If I were doing another project like this again, I would spend much more time evaluating the maturity of the language's support for Linux kernel programming. Rust's relative novelty made achieving memory-safety effortless, but it also made kernel programming difficult and hacky. Despite its memory safety issues, it may be more natural and easier to write a kernel project in C rather than a new memory-safe language like Rust or Go. The Linux kernel is written entirely in C, and C isn't subject to issues like our LLVM bug. C is also fairly stable due to its age. During the development of *BentoCrypt*, Rust had an update that broke large portions of the upstream framework *Bento*. This meant that we had to freeze our Rust compiler version to a nightly version before the updates. This reduced the number of libraries that were available to us, as many encryption libraries would simply not compile on our specific nightly Rust version. In comparison, C has scarcely changed over the years: meaning that it avoids this sort of issue.

B. The Current State of BentoCrypt

As of now, the helper program *pwd* is completely finished. It runs at the user level so was not subject to the bugs related to our linuxkernel target and lack of standard library.

The *encryption* library is (nearly) finished. At the user level, it safely encrypts bytes of data and panics if the encrypted data does not pass verification. However, since we had to disable *sse* and *avx2* for our kernel version, we cannot guarantee that it works as expected at the kernel level at this point.

The file system itself, *xv6fs_encrypted*, is unfinished. Currently, we call our encryption library on the data parameter at the beginning of the bento_write function, which should theoretically ensure that the data written to disk is encrypted with AES-GCM. However, due to the difficulties enumerated above it's unclear whether the call is properly encrypting the data. We have also not yet implemented decryption on read.

V. Benchmarks

Using the current state of the xv6fs_encrypted (with the *encryption* library on write, but read unaltered), we ran benchmarks for read and write. To compare the performance to similar projects, we also ran the benchmarks on *ext4 gocryptfs*, and *encfs*. All benchmarks were run on a desktop with an *AMD Ryzen 3400G* CPU and 16GB of RAM. The OS is *Ubuntu 20.04 LTS*.

A. "Naive" Benchmarks

The first benchmark we ran was creating and writing the string "test" to 1000 files. Note that this benchmark relies much more on file creation than the IOZone benchmark we ran as well. Since our cryptographic file system encrypts on writes, and doesn't encrypt file names and data during file creation, the IOZone benchmark is more representative of the performance impacts incurred from encryption. This benchmark is more representative of day-to-day usage where users my create directories and files frequently.

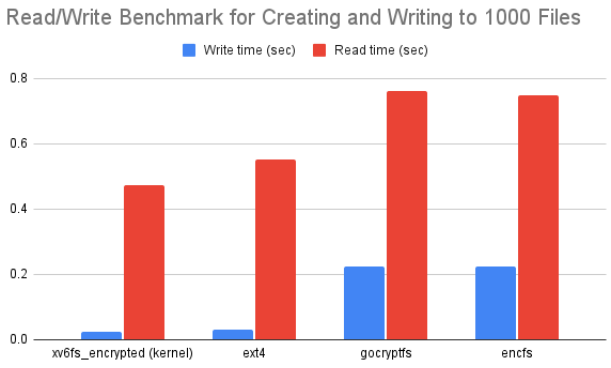

Fig. 3. Naive Benchmark Chart (Time Elapsed; Lower is Better)

The performance of *ext4* and *xv6fs_encrypted* are extremely similar. Note that *ext4* isn't providing encryption here, while *xv6fs_encrypted* is. However, *ext4* also provides more features than *xv6fs* which is generally used as an educational tool. Regardless, both are significantly faster (especially in write performance) than *gocryptfs* and *encfs*. *gocryptfs* and *encfs* provide encryption on write like our file system. What causes the significant performance lag in these two file systems is the FUSE framework - which limits the file systems to running in userspace. *ext4* and our file system *xv6fs_encrypted* run at the kernel level and thus have access to kernel primitives which increase performance (thanks to the *Bento* framework).

B. IOZone Benchmarks

For more complete benchmarks, we also ran the IOZone benchmark utility [9]. IOZone runs a file system through a more comprehensive set of read and write operations and generates a score based on its performance. Since IOZone is more reliant on read and write than file creation, it makes for a better indicator of the impact of encryption on performance.

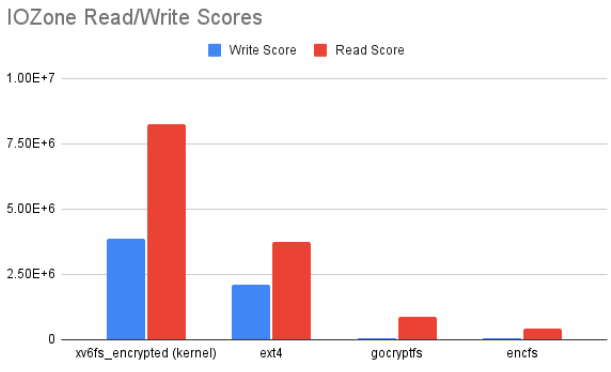

Fig. 4. Comprehensive Benchmark Chart (Score Value; Higher is Better)

encfs and *gocryptfs* lag behind here as expected. However, the performance of *xv6fs_encrypted* and *ext4* are surprising. *xv6fs_encrypted* is performs sizeably better than *ext4*. I believe this could be attributed to the additional features that *ext4* implements and the lack of data decryption from *xv6fs_encrypted*.

VI. Conclusion

Despite my inability to finish the project, I learned a lot from the development of *Bento*. Firstly, I learned that I need to be pragmatic in choosing development tools for a project. Although Rust was newer and provided features like memory safety, it may have been easier to develop a kernel file system using an older language/framework. Second, we learned the gravity of kernel primitives in enhancing the performance of a file system. Even though *BentoCrypt* was not able to use the target features *sse* and *avx2*, it still performed significantly better than comparable cryptographic FUSE file systems. Thus, I believe the future of performant encrypted file systems lies in kernel file systems using frameworks like *Bento*, not FUSE.

Acknowledgment

I would like to thank my mentor Professor Chandy for his incredible guidance and help through this REU. Chandy was incredibly supportive and patient throughout this entire project. Whenever I found myself stuck, Chandy helped me develop a plan to move forward.

I would also like to thank Samantha Miller for heading the Bento project and for helping me adapt the Bento framework to work with newer versions of the Linux kernel and Rust. Since Bento was released, the Linux kernel file system API has changed, meaning that modern Linux distros were uncompatible with the original Bento. Ms. Miller was extremely helpful in the process of patching Bento to work with the newer kernel, and in development with Bento more generally.

I would also like to thank Professor Khan for organizing this REU and introducing us to technical writing - this paper would not have been possible without him.

I also must thank the author of the Rust AES-GCM library Tony Arcieri, and Rust developer Hans Kratz. Both helped me in debugging the LLVM error.

Lastly, I must thank the National Science Foundation (NSF) for generously supporting this research.

References

[1] S. Miller, K. Zhang, M. Chen, R. Jennings, A. Chen, D. Zhuo, and T. Anderson, “High velocity kernel file systems with Bento,” in USENIX, 19th USENIX Conference on File and Storage Technologies, February 2021.

[2] rfjakob, ”Home Page - gocryptfs,” May, 2020. [Online]. Available: https://nuetzlich.net/gocryptfs/. [Accessed July 1, 2021].

[3] D. Kirkland, ”Home Page - eCryptfs,” 2020. [Online]. Available: https://www.ecryptfs.org/. [Accessed July 1, 2021].

[4] V. Gough, ”About Page - EncFS,” May, 2020. [Online]. Available: https://vgough.github.io/encfs/. [Accessed July 1, 2021].

[5] B. Vangoor, V. Tarasov, and E. Zadok, “To FUSE or not to FUSE: performance of user-space file systems,” in USENIX, 15th USENIX Conference on File and Storage Technologies, February 2017.

[6] M. Dworkin, “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC,” in NIST Special Publication 800-38D, November 2007.

[7] C. Percival, ”The Scrypt Key Derivation Function,” August, 2020. [Online]. Available: https://www.tarsnap.com/scrypt.html. [Accessed August 1, 2021].

[8] Rust Foundation, ”Platform Support,” September, 2020. [Online]. Available: https://doc.rust-lang.org/nightly/rustc/platform-support.html. [Accessed August 1, 2021].

[9] D. Capps, ”Home Page - IOzone,” Jan., 2016. [Online]. Available: http://iozone.org/. [Accessed July 1, 2021].