Bundesamt
für Sicherheit in der
Informationstechnik

Security Evaluation of VeraCrypt

Authors and Acknowledgments

This study was executed by the Fraunhofer Institute for Secure Information Technology (SIT) on behalf of the Federal Office for Information Security (BSI). This publication was authored by (in alphabetic order):

Hülya Evkan
Norman Lahr
Ruben Niederhagen
Richard Petri
Andreas Poller (project lead)
Philipp Roskosch
Michael Tröger

The authors would like to thank Johannes Mittmann, Steven Arzt, Peter Weiland and Mounir Idrassi for their valuable support and advice.

Project contact at Fraunhofer SIT

Fraunhofer Institute for Secure Information Technology
Andreas Poller
Rheinstrasse 75, D-64295 Darmstadt, Germany
E-Mail: andreas.poller@sit.fraunhofer.de
Phone: +49 6151 869-170
Internet: https://www.sit.fraunhofer.de

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-0
E-Mail: bsi@bsi.bund.de
Internet: https://www.bsi.bund.de
© Federal Office for Information Security 2020

Executive Summary

VeraCrypt is a popular open-source tool for disk encryption available for Windows, Linux and macOS. VeraCrypt is a successor of TrueCrypt, an encryption software whose development stopped in 2014 and which is no longer maintained by its developers. VeraCrypt adopted most of TrueCrypt's source code and to this day shows considerable similarities to TrueCrypt.

This report summarizes the results of a year-long project of Fraunhofer Institute for Secure Information Technology, Darmstadt, Germany on behalf of the Federal Office for Information Security (BSI), Bonn, Germany. After starting off with an extensive research into the project evolution and related work, we executed a security analysis of VeraCrypt with a focus on its cryptographic mechanisms and the security of the application as a whole. During the research process we followed a security model that includes pertinent usage scenarios including the use of VeraCrypt for secure online sharing of data and the use on public systems and servers. Our research efforts included both automated and manual testing techniques, manual code and documentation review, as well as the creation and use of dedicated test tools.

We found that although VeraCrypt is a well-acknowledged software project, it appears that the project is still mostly driven by a single developer rather than a development team. The data we collected for VeraCrypt's development history indicate that the project did not follow an elaborated software-development cycle with acknowledged best practices for software engineering, for instance, quality gates, peer reviews, and documentation of code changes. The code base still mainly consists of code from the TrueCrypt project that has been repeatedly criticized for its poor coding style as the case of differing implementations of the random number generator for different operating systems still tellingly shows. The inherited code base has not been cleaned up, moreover, the development still follows questionable coding practices.

The basic functionality such as the parsing of container files and the interface to the kernel driver did not show security issues in any of our tests. We also did not find vulnerabilities in the cryptographic algorithms of VeraCrypt. However, VeraCrypt still uses the outdated and deprecated RIPEMD-160 hash algorithm and we found peculiarities with respect to the implementation of the random number generators and the GOST block encryption cipher. The recently integrated memory encryption has a weak rationale from a security perspective and increases the cost of related attacks by a relevant margin in very limited scenarios only.

We recommend the VeraCrypt project to switch to well-acknowledged and reliable open-source libraries for the implementation of cryptographic functions instead of proceeding to provide and use own outdated cryptographic code in the VeraCrypt code base. We also recommend to switch to a state-of-the-art key derivation function. As a final remark, we want to stress that VeraCrypt can only protect data effectively in case of theft or loss of encrypted devices but not against any form of online attacks on a running system. Also, VeraCrypt cannot provide any protection in scenarios where an attacker can visit a target system multiple times.

In conclusion, we did not find substantial security issues in VeraCrypt. VeraCrypt in its current version does seem to protect the confidentiality of data in an encrypted volume as long as the volume is not mounted. Authenticity and integrity, however, are not protected. A mounted VeraCrypt volume is exposed to a multitude of attack vectors including vulnerabilities of the host system. Hence, any volume-access scenario exceeds the protection envelope of VeraCrypt. The development practices and the resulting code quality of VeraCrypt are a cause for concern. Therefore, we cannot recommend VeraCrypt for sensitive data and persons or applications with high security requirements. We recommend to execute similar security assessments also for future versions of the software.

Contents

1 Introduction

1.1 Overview of VeraCrypt

Since the discontinuation of TrueCrypt was announced in May 2014, multiple projects emerged to carry on the development of open-source hard drive encryption software on the basis of the TrueCrypt source code. VeraCrypt is one popular fork of TrueCrypt incepted in the aftermath of TrueCrypt's end of life. While maintaining most of TrueCrypt's design, architecture and code base, VeraCrypt aims on incrementally improving and extending the original software. For this reason, VeraCrypt is often considered a natural replacement in scenarios where TrueCrypt was in use before.

Encryption scopes

The possible encryption scopes remain unchanged in VeraCrypt compared to its predecessor TrueCrypt. In general VeraCrypt allows the full encryption of partitions on a hard drive or to create an encryption container file that can be mounted as a virtual disk. In both cases, VeraCrypt uses the term *volume* to point to an encryption drive the user mounts either from a file or a partition.

For Windows as operating system, it is also possible to encrypt the full system partition. VeraCrypt provides its own bootloader to start the system decryption after machine startup.

A third major feature are *hidden volumes*. Hidden volumes are volumes nested inside another outer volume. The outer volume serves as a decoy allowing to deny the existence of the hidden volume, a property called *plausible deniability*. A person in possession of the encrypted outer volume, perhaps even in possession of the keys to decrypt it, cannot distinguish the space of the hidden volume in the outer volume from any arbitrary random data. Consequently, the legitimate user of the hidden volume can plausibly deny its existence if there does not exist further evidence that indicates otherwise.

Encryption modes and key management

VeraCrypt offers a range of well-known encryption algorithms to encrypt volumes. The user can even choose to combine several encryption algorithms.

For starting volume decryption and encryption, the user needs to provide a volume password that VeraCrypt uses to decrypt master keys stored in a volume header. In addition, one or multiple keyfiles can be combined with the password. This way it is possible to some extend to implement an access scheme relying on knowledge (password) and possession (keyfiles, e.g., on a USB flash drive). There also exists a limited support for security tokens and smartcards. VeraCrypt itself, however, does not provide a functionality where two or more passwords can be used independently or two or more passwords need to be combined to decrypt a volume.

Once a volume is mounted as a drive, the de- and encryption is working transparently for the user, because the file system handling is done by the operating system.

User interfaces

VeraCrypt provides three user interfaces to interact with the software. All three were already present for TrueCrypt and underwent only limited changes:

- The bootloader user interface allows to enter a password during system boot as well as a parameter for the key derivation function. It is only available if the user activated the full system encryption.
- The main graphical user interface provides access to all major functions such as creating and mounting VeraCrypt volumes, managing keyfiles and configuring software options.
- The command line interface has similar capabilities as the graphical user interface but can be used in a command line console, e.g., on headless computer systems.

1.2 Project Scope and Methodology

We executed this analysis on behalf on the Federal Office for Information Security (BSI) in the course of a joint security evaluation project. The project goal was to extend the 2015 TrueCrypt security evaluation to VeraCrypt as its successor. Consequently, both the 2015 TrueCrypt study and this VeraCrypt study share a similar approach. For both studies, the Federal Office for Information Security commissioned Fraunhofer SIT to investigate the security properties and features of the software, evaluate the software for security weaknesses and examine whether it is aligned to state-of-the-art cryptographic requirements.

In detail, we performed the following steps:

- **Tracking the evolution of VeraCrypt:** We investigated the difference between TrueCrypt and VeraCrypt both on the level of the software itself but also with a focus on how development actions take place. To this end, we assessed in depth the development documentation of the software and changes on source code level. Our goal was to provide insights into how the project moved forward since the discontinuation of TrueCrypt with respect to how the software changed and how development work is organized.
- **Security modeling:** We systematically collected and assessed potential usage scenarios and security goals for VeraCrypt. We evaluated attack scenarios that can impede the achievement of the described security goals. This security model is a starting point to guide the following steps in the security assessment.
- **Security analysis of cryptographic algorithms**: Cryptographic algorithms are the very basis of every encryption software. We started off with meticulously documenting the cryptographic algorithms in VeraCrypt and checking them against current state-of-the-art standards and recommendations. This includes an assessment of the random number generators used in VeraCrypt as a fundamental function related to cryptographic mechanisms. We compared the implementation of cryptographic algorithms in VeraCrypt to reference implementations from public sources while looking for deviations that might affect the security of the software. We evaluated the cryptographic algorithms manually and with known-answer tests. We also checked VeraCrypt's handling of cryptographic secrets in a system's main memory.
- **Application security review:** Our security model showed that not only weak cryptography could expose VeraCrypt users to security risks but also the overall application security can be a concern in specific security scenarios. Hence, we evaluated VeraCrypt with automated code analysis tools and through manual code inspection. We focused on the security of the kernel driver as a potential entry point for attempts to undermine security measures of the operating system and container files as a further entry point for external attackers. We explored both issues by using fuzzing techniques.
- **Review of code quality and documentation:** As a further indicator of the trustworthiness of VeraCrypt, we evaluated the quality of the source and the software documentation.
- **Review of related work:** During the project we reviewed related work about the security of VeraCrypt and TrueCrypt. This includes research publications from academia, non-governmental bodies and industry, but also reports retrieved from VeraCrypt's development management platform. The related work steered our evaluation and allowed us to adopt, extend, and complement the evaluation methods that were successfully applied by other researchers.

As agreed with the BSI, we focused primarily on the Windows and Linux version of VeraCrypt. We also left out a comprehensive evaluation and discussion of the plausible deniability function (*hidden volume* feature, see 1.1) since we did not consider it a relevant feature for a majority of VeraCrypt users.

We investigated VeraCrypt in version 1.23 as published by the VeraCrypt project in September 2018. In October 2019, during the course of our study, a new version 1.24 of VeraCrypt was released. Together with the BSI we decided to investigate some security-related changes in this new version. The resulting findings of these extended investigations are marked in this report as pertinent to version 1.24. However, if not stated otherwise the remainder of this report refers to version 1.23 only.

1.3 Report Structure

This report summarizes the results of our project. It is organized along the multiple steps in our research project (cp. 1.2): Chapter 2 explains the development of VeraCrypt after the discontinuation of TrueCrypt. The development history is examined in detail, differences are analyzed on source code level, on the level of single source code commits, and with a focus on cryptographic primitives.

In Chapter 3, we describe the security model that we considered to select the security tests for execution in the course of this project. It includes an analysis of potential usage scenarios, pertinent security goals and attack scenarios to consider.

In Chapter 4, we document the results of our assessment of cryptographic algorithms including ciphers, hash functions, key derivation functions and the random number generators. We also document our results for assessing a recently introduced memory encryption function and the secure deletion of sensitive cryptographic data from the program memory.

We discuss the various aspects of VeraCrypt's application security starting with results from automated code scanning tests in Chapter 5. We describe our inspection of the kernel driver interface and our fuzzing tests for the container file processing routines. We also briefly discuss the security of third-party libraries included into VeraCrypt.

Results from our evaluation of the code quality and the software documentation are described in Chapter 6. Finally we discuss results of previous security evaluations in Chapter 7. The Appendix in Chapter 9 provides further details about our testing procedures and results.

1.4 Project Results

Overall, we found that VeraCrypt is mostly the effort of a single developer (see 2.2). The code base increased considerably over time since the VeraCrypt project started in 2012. Multiple changes in the code base addressed security issues reported by other projects such as the Open Crypto Audit Project or Project Zero (see 7.2). New block ciphers and hash functions were added to the project, while others were removed over time (see 2.5).

The VeraCrypt project continues the coding practices of TrueCrypt including its failure to follow well-acknowledged best practices (see 6.1). Moreover, no visible efforts took place to improve the quality of the legacy code. The documentation was adopted from TrueCrypt and extended, but the descriptions differ from the actual implementation in some regards (see 6.2).

While the cryptographic mechanisms show issues, none of these issues leads to a considerable security weakness in most usage scenarios: The key derivation function is outdated (see 4.3), the random number generators of Linux and Windows differ without any obvious reason (see 4.4), the impact of the memory encryption introduced with version 1.24 is limited (see 4.7) and the block ciphers are prone to side-channel attacks (see 4.5). VeraCrypt includes RIPEMD-160 as outdated hash function (see 4.2). We suggest the developers to switch to a trustworthy cryptographic library instead of further maintaining the cryptographic legacy code (see 4.8).

Our application security review did not reveal any security issue. The findings of the automated analysis tools were checked manually and turned out to be negligible (see 5.1). The interface to the Windows kernel driver passed our various security tests without any security-relevant findings (see 5.3). VeraCrypt's routines to parse the header of container files resisted our attack attempts with common fuzzing techniques (see 5.4). Included third-party libraries are up to date without any currently known vulnerabilities (see 5.5).

2 Evolution from TrueCrypt to VeraCrypt

Before assessing the security of VeraCrypt in detail, we analyzed the development history of the software with specific focus on the software's evolution since the fork from TrueCrypt. We also analyzed the general development history of VeraCrypt on the basis of data from GitHub and created and analyzed a full comparison between TrueCrypt 7.1a and VeraCrypt 1.23 ("diff" or "full diff"). Furthermore we reviewed each source code commit with respect to potential impact on software security. This section presents the results of our efforts.

2.1 Identifying the Source Code Bases to Compare

To compare TrueCrypt and VeraCrypt and to track and evaluate the changes the VeraCrypt developers made since branching of VeraCrypt it is necessary to have a reference code base for both software products. We used the last stable source code version 7.1a as reference for the TrueCrypt source code. Likewise for VeraCrypt the source code version 1.23 was considered, which was the latest stable version by the time this analysis started.

VeraCrypt source code version 1.23 was retrieved from the official project site at GitHub[1]. Git commit `82050910`[2] corresponds to the last stable version 1.23, dated September 12, 2018, which marks the reference VeraCrypt source code used for the analysis.

The TrueCrypt source code version 7.1a can also be retrieved from the history of VeraCrypt's GitHub repository, which makes the analysis more convenient. However, attention needs to be paid to the fact that the original TrueCrypt source code was only added gradually in two steps: The first, initial commit to the repository dating back to June 22, 2013 adds the source code from the TrueCrypt "Windows Source ZIP" archive. After this initial commit, the VeraCrypt GitHub repository actually mirrored the original full source code of the TrueCrypt 7.1a Windows version. We cross-checked that the initial commit is identical to the source code used for the previous security analysis of TrueCrypt version 7.1a[3].

However, after the initial commit, the GitHub repository did not mirror the *full* cross-platform source code base of TrueCrypt 7.1a, because source code specific for macOS and Linux from the original "tar.gz" source code archive of TrueCrypt 7.1a was still missing. It was only added later in May 31, 2014 by commit `7ffce028`[4]. Nevertheless, since the VeraCrypt developers changed the source in their repository between the initial commit and the macOS and Linux source code commit, the repository did not reflect the original full source code of TrueCrypt 7.1a anymore by May 31, 2014.

Hence, all further analysis described in this report was executed two-staged, considering the initial commit from June 22, 2013 as reference point for the full Windows source code of TrueCrypt 7.1a, and the macOS and Linux source code commit from May 31, 2014 as reference point for the macOS and Linux source code parts of TrueCrypt 7.1a. When comparing versions in the repository, the later macOS and Linux source code commit was factored out, i.e., not considered as a change performed by the VeraCrypt development team, but as input originating from the original TrueCrypt sources.

[1] https://github.com/veracrypt/VeraCrypt

[2] commit `82050910f8c742631c733c475e6e6a8a1876d904`

[3] Compare https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Truecrypt/Truecrypt.pdf

[4] commit `7ffce028d04a6b13ef762e2b89c34b688e8ca59d`

2.2 General Overview of the Development History

After identifying the source code versions that mark the range of the further analysis, we retrieved some basic figures reflecting the general activities in the VeraCrypt project.

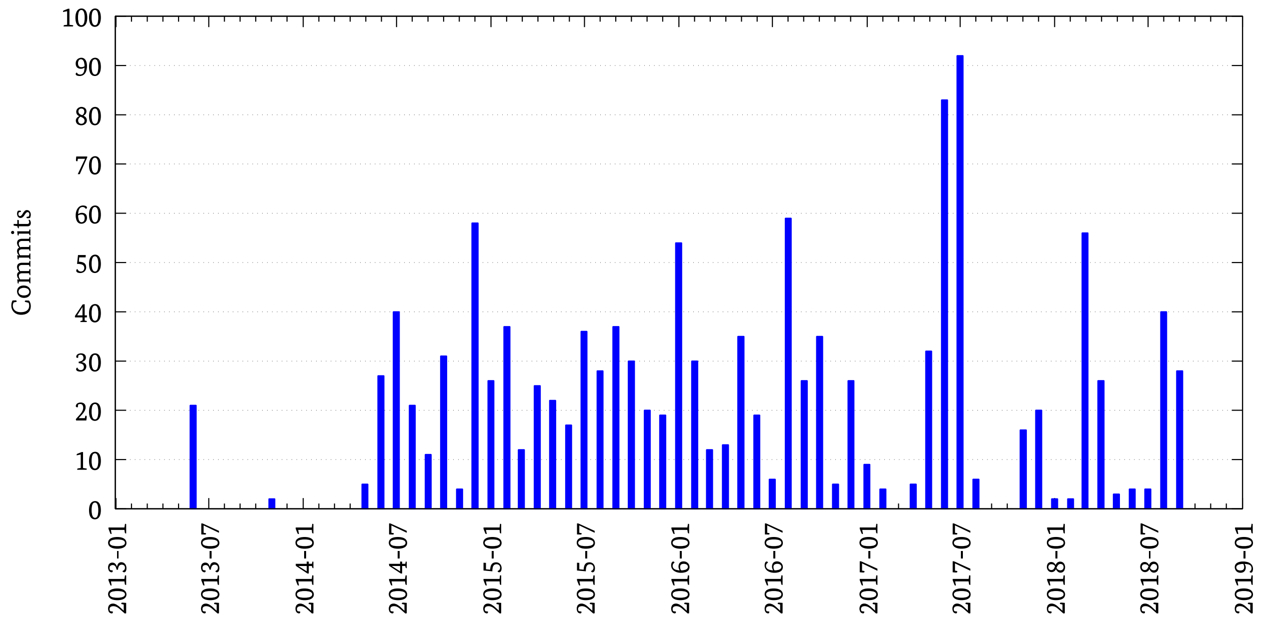


Figure 1: Number of code commits in the VeraCrypt project by year and month.

Development activity over time

As show in Figure 1, the number of commits (broken down to calendar year and month) varied since the project's inception[5]. Until mid-2014, only little development activity took place. After the announcement that TrueCrypt is discontinued in May 2014, development pace increased considerably with a much higher commit frequency. The commit of macOS- and Linux-specific source files from the original TrueCrypt source – only few days after the discontinuation was announced – marks a turning point in this respect.

Specific time periods can be identified where the number of commits increases noticeably:

December 2014: User interface and program performance was improved and minor bugs were fixed.

January 2016: Handling of volumes on all platforms was improved and multiple bugs were fixed.

August 2016: New encryption algorithms were introduced. Significant implementation work on UEFI boot was done. Vulnerabilities were fixed.

June and July 2017: Program libraries were updated. Documentation was extended. The user interface was improved. Security fixes were introduced, as well as bug fixes in the bootloader component.

March 2018: The user interface was improved as well as the handling of Windows Secure Desktop[6]. Further bugs were fixed.

It appeared that the developers did not continuously follow a coherent long-term development agenda. There are productive bursts of work in the project in limited time periods. Prioritization of work, for instance, which feature to work on next, is not visible in the data available on GitHub.

[5] Figure generated with the tool *gitstats* available under http://gitstats.sourceforge.net/.

[6] Technique for creating an isolated desktop environment in Windows to prevent other processes accessing security-relevant elements of the graphical user interface. See also https://docs.microsoft.com/de-de/windows/win32/api/winuser/nf-winuser-switchdesktop

Generally speaking only few comprehensive descriptions of self-contained features chosen for implementation are available in the documentation. One example that we found is the Personal Iterations Multiplier (PIM), which improves the header key derivation by increasing the number of its iterations, thereby fixing a weakness of the original TrueCrypt source. This feature is broadly described on a dedicated web page[7]. By contrast, other features such as EFI Boot or SecureBoot are described in the release notes but lack similar extensive documentation.

Regarding the development documentation, the commit messages are in general detailed and work is broken down in small commits, which supports the analysis of the development history. Isolated, single commits often consist of small fixes or workarounds for issues.

Growth in lines of code over time

An indicator for how much the VeraCrypt developers newly contributed to the original TrueCrypt source code is the development of the number of lines of code of the project, as shown in Figure 2[8].

There is a continuous increase in the number of line of code, with noticeable steps:

May 2014: Addition of Linux- and macOS-specific code parts of TrueCrypt 7.1a. This is the first commit.

July 2014: Removal of deprecated cryptographic algorithms led to a decrease of lines of code.

January 2015: Small-step increase in the lines of code due to the addition of *VeraCryptExpander*, a utility for Windows that is able to expand a VeraCrypt encrypted volume.

August 2016: Windows *XZip* library files were added.

October 2016: Replacement of *XZip* library with *zlib* and *libzip* caused a further noticeable increase in lines of code.

June 2017: Optimized assembly version of SHA-512 and SHA-256 is added to VeraCrypt.

Letting aside the first mentioned event of the addition of macOS- and Linux-specific source files from the original TrueCrypt source, the code base overall increased by about 50% compared to the original TrueCrypt source code.

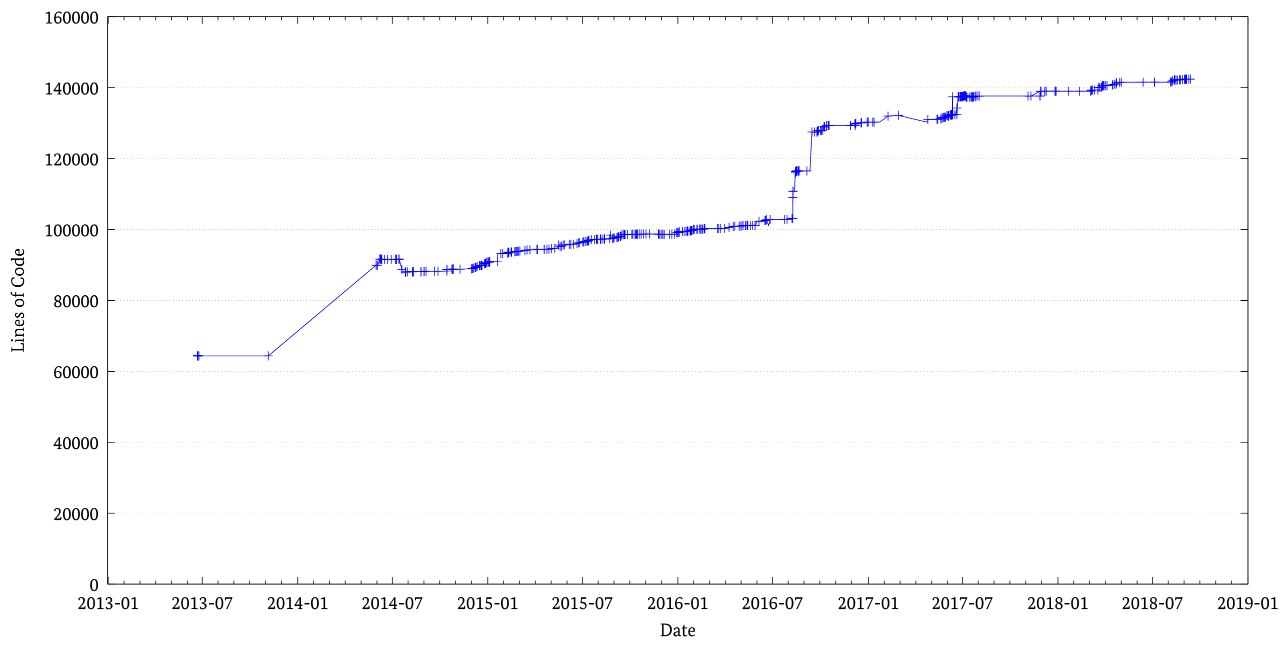


Figure 2: Changes in the number of code lines over time.

[7] See https://www.veracrypt.fr/en/Personal%20Iterations%20Multiplier%20(PIM).html

[8] Figure generated with the tool *cloc* in version 1.74 incorporating every commit, aggregating all C, C++ and Assembly code lines.

Number of commits by author

Analyzing how the number of code commits is distributed among developers indicates how the project is internally organized. Table 1 shows the number of commits for the top five developers who did most commits to the VeraCrypt project.

Table 1: Top 5 Authors of VeraCrypt in respect to code commits and active days. A vast majority of code contributions came from a single developer.

Author	*Commits*	*Active Days*
Mounir Idrassi	1209 (94.38%)	467
kavsrf	10 (0.78%)	9
David Foerster	10 (0.78%)	4
Ettore Atalan	7 (0.55%)	5
TigerxWood	5 (0.39%)	5

It is remarkable that by far the largest amount of commits can be attributed to one single developer, Mounir Idrassi, who appears to be the main developer in the project. A review process including other developers seems to be non-existent. All other 30 remaining developers identified in the development history contributed well below 1% of the commits, with only few active days in the project. These numbers suggest that the VeraCrypt project is not an effort of a team, but rather of a single developer alone.

Time periods when commits take place

Analyzing the points in time when commits were executed can give further information about how the project is organized and when work is achieved. As Figure 3 outlines, the most code commits to the project generally take place during the months June, July, August and December.

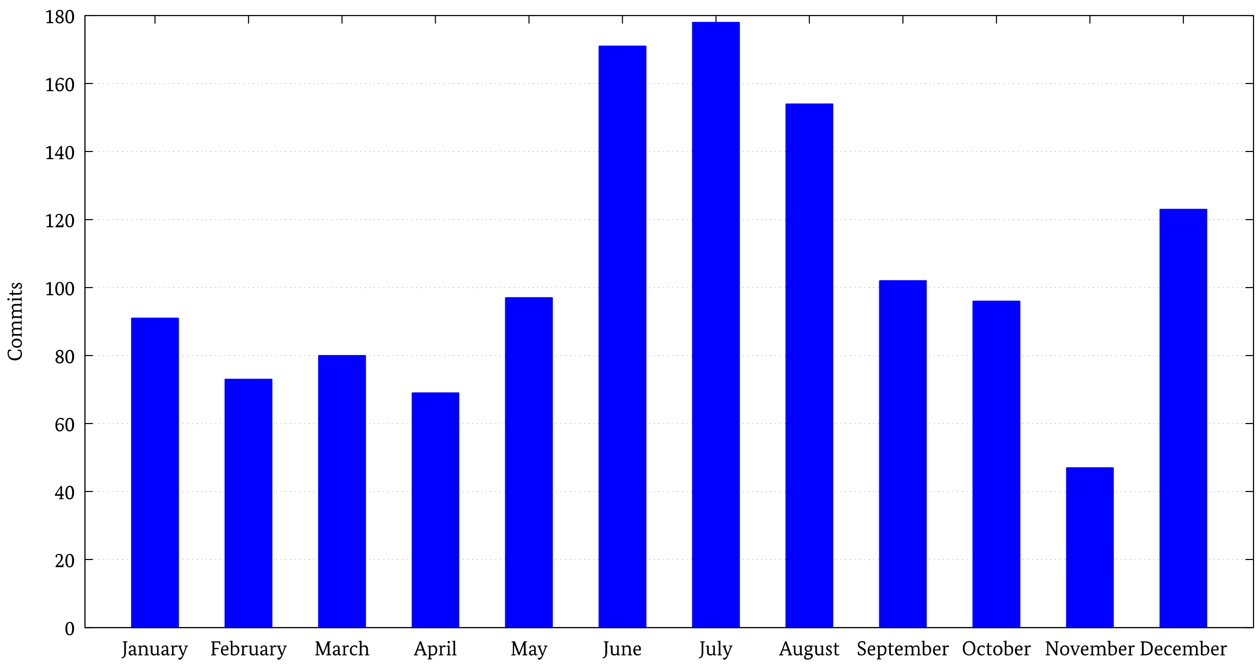


Figure 3: Number of code commits by month.

Also, as Figure 4 shows, most commits were done on Sundays or Mondays, while the number of commits was considerably lower on the other week days.

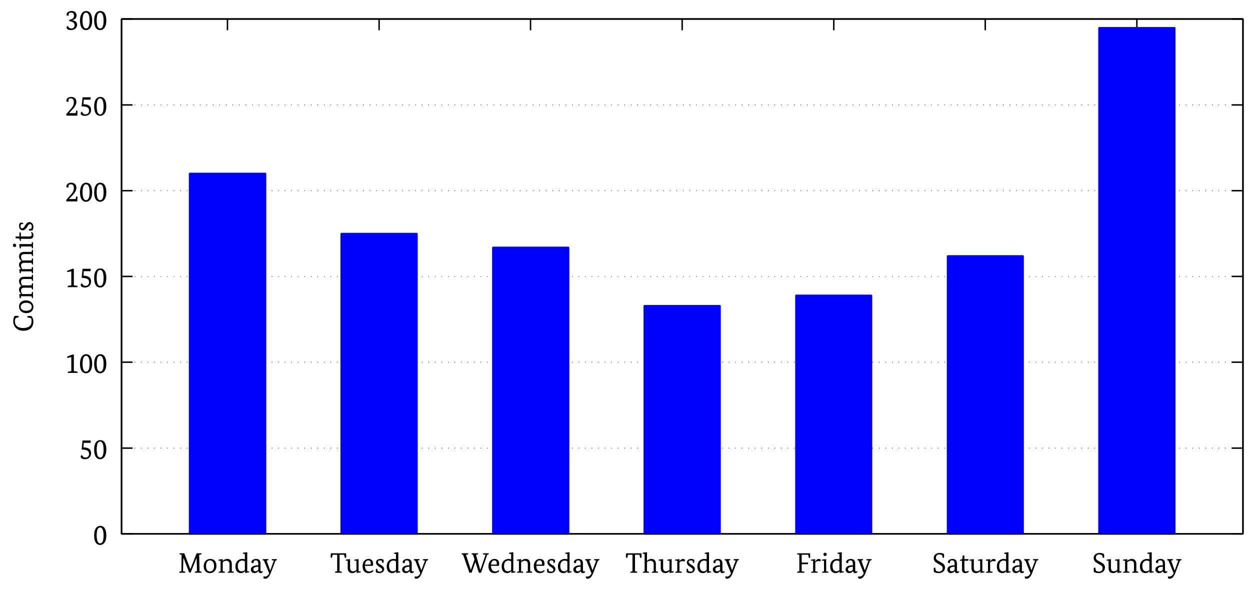


Figure 4: Number of code commits by week day.

When analyzing the hours of the day when commits take place, there is a notable peak of commits one hour before and after midnight, as shown in Figure 5.

This data would support the assumption that the main developer works on VeraCrypt mostly in his spare time, during vacations (e.g., summer and Christmas vacation periods) and the later hours of the day. However, there might be other reasons for these patterns and the meaningfulness of this analysis may be rather limited.

2.3 Full Source Code Comparison ("Full Diff")

Since the fork of TrueCrypt into VeraCrypt, a considerable number of commits and changes was applied to the code. Quantifying these changes with respect to their locations in the source code allowed us to focus the security assessment on specific differences between both software products. To this end, we performed a full source code comparison between TrueCrypt 7.1a and VeraCrypt 1.23, which was the latest stable version by the time this analysis was done.

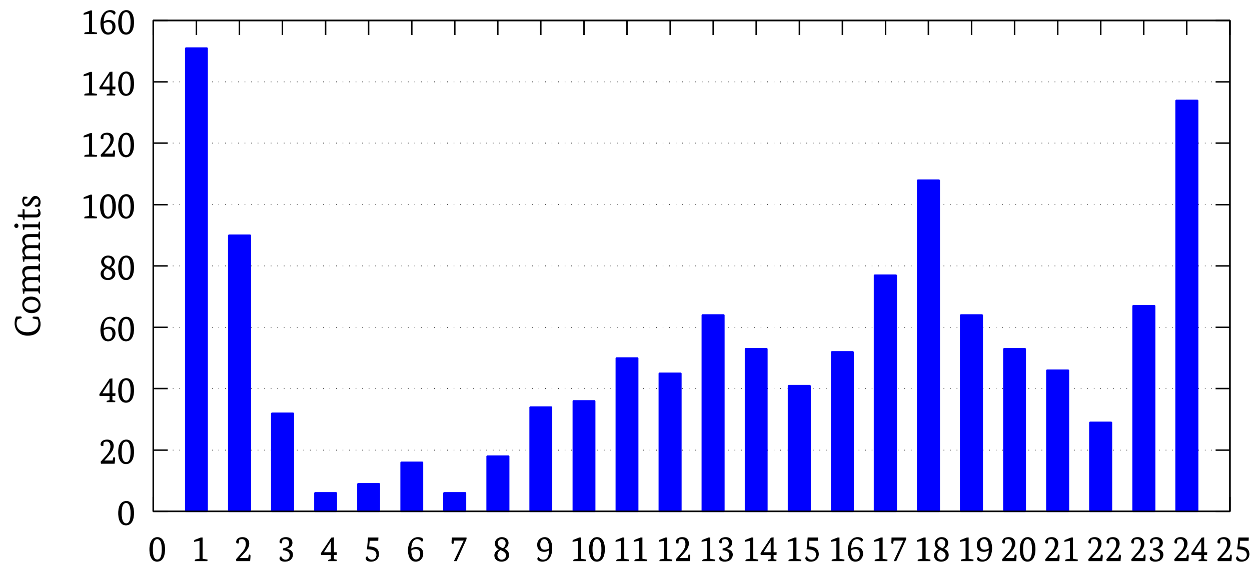


Figure 5: Number of commits by the hour of the day.

As already described in Section 2.1, the GitHub repository of VeraCrypt was used to create the comparison, but during the process attention had to be paid on the issue that the full windows source of TrueCrypt 7.1a was committed initially, while further additional cross-platform source was only committed at a later point in time. Therefore, a simple diff between the initial repository commit and the commit tagged as version 1.23 would give meaningless results, because it would falsely factor in the Linux and macOS code as code contributions of the VeraCrypt team. With the two-step approach described in the following Section 2.3.1, this problem was avoided together with further issues that could distort the results.

2.3.1 Approach

The approach for performing the full source code comparison ("full diff") comprises two steps: First, preprocessing is performed in order to remove parts of the software project that do not represent program logic and thus are not relevant for any further security analysis. Then the comparison (or "diff") is executed in two stages to cover all changes between the full source code base of TrueCrypt and VeraCrypt.

Preprocessing

Preprocessing is done for each commit that is considered for the complete diff.

- Extract the `src/` directory ignoring all other directories. This way, files not related to actual implementations of VeraCrypt's functionality are removed. Examples for removed files are files for internationalization, documentation and test cases as they are not relevant for the security analysis.
- Remove all files that do not contain source code and only keep `c`, `cpp`, `h`, `asm` and `S` files. For instance pre-build binaries, files containing cryptographic material, make files, and `xml` files were removed as changes in those files do not represent changes in VeraCrypt's implementation.
- Remove all comments from each source code file. Comments may account for a considerable amount of committed source-code file content, e.g., when license agreements were changed. Hence they may distort and mislead any analysis attempting to quantify changes in the program logic.
- Fix differing encodings of line endings between Windows and Linux/macOS files. If the encoding of line endings was not aligned, the comparison tool would falsely indicate changes when switches between encodings occur.
- Remove all white spaces except new lines. Such as with coding line endings, this measure was also performed to avoid false positives during source code comparison.

Executing the actual comparison ("diff")

The complete comparison ("diff") is performed in two stages: First all code changes were extracted that happened after the initial source code commit to the VeraCrypt repository, but before the Linux and macOS cross-platform source code of TrueCrypt 7.1a had been committed (cp. 2.1). In the next step, all code changes were extracted that happened after the cross-platform source code had been submitted until the commit representing VeraCrypt 1.23. This procedure yields two diffs, which were summed up to create the complete diff of the two source code bases. From the complete diff, information was extracted about the parts of the code changed most frequently.

2.3.2 Detecting Linux and macOS Source Files

VeraCrypt has been forked initially from the Windows sources of TrueCrypt. Linux and macOS source files have been added later on (cp. 2.1). As the structure of the VeraCrypt source code is useful for further analysis, platform-specific components are an important entity. The commit `7ffce028`[9], which added the

[9] commit `7ffce028d04a6b13ef762e2b89c34b688e8ca59d`

Linux and macOS sources and contains only platform-specific files, was further analyzed to reveal those files and directories that actually contain macOS- or Linux-specific software code.

Unix: The directory `src/Core/Unix` contains files for Unix-based operating systems. There are subdirectories for Linux, macOS, FreeBSD and Solaris.

Fuse: The directory `src/Driver/Fuse` contains files for using *fuse*[10]. This is a Unix-specific software that is not relevant for Windows.

Core: The directory `src/Core` contains files that are not exclusively for Unix-based operating systems. Multiple `#ifdef TC_WINDOWS` statements are contained in the source code, indicating that the code was not designed as "Unix part" of the software.

Main: The directory `src/Main` contains source files implementing the graphical user interface (GUI).

Platform: The directory `src/Platform/Unix` contains sources for system-specific implementations, e.g., interactions with the operating system or file system.

Volume: The directory `src/Volume` contains several cryptography-related implementations like encryption modes and hash functions. Similar implementations can also be found in the `src/Common` directory.

2.3.3 Changed Parts

Figure 6 was generated by counting lines of code changed in each file between the initial repository commit and the commit tagged with version 1.23, and aggregating the numbers per source code subdirectory. Only C, C++ or assembler source code files were considered and all comments were removed. The subdirectory names seem to be expressive with respect to the functionality implemented in the contained source-code files.

A new feature of VeraCrypt is the encryption of volumes that run in UEFI boot[11]. This may explain the changes in the `Mount`, `Driver` and `Volume` folders.

[10] https://github.com/libfuse/libfuse

[11] https://www.heise.de/security/meldung/Verschluesselungs-Software-VeraCrypt-kann-jetzt-UEFI-3301464.html

Cryptographic algorithms were also changed, added or removed, as seen in the `Crypto` subdirectory. The setup routine was modified, and also the Graphical User Interface was changed as can be seen in the `Main/Forms` directory. The *zlib* and the *libzip* libraries were added.

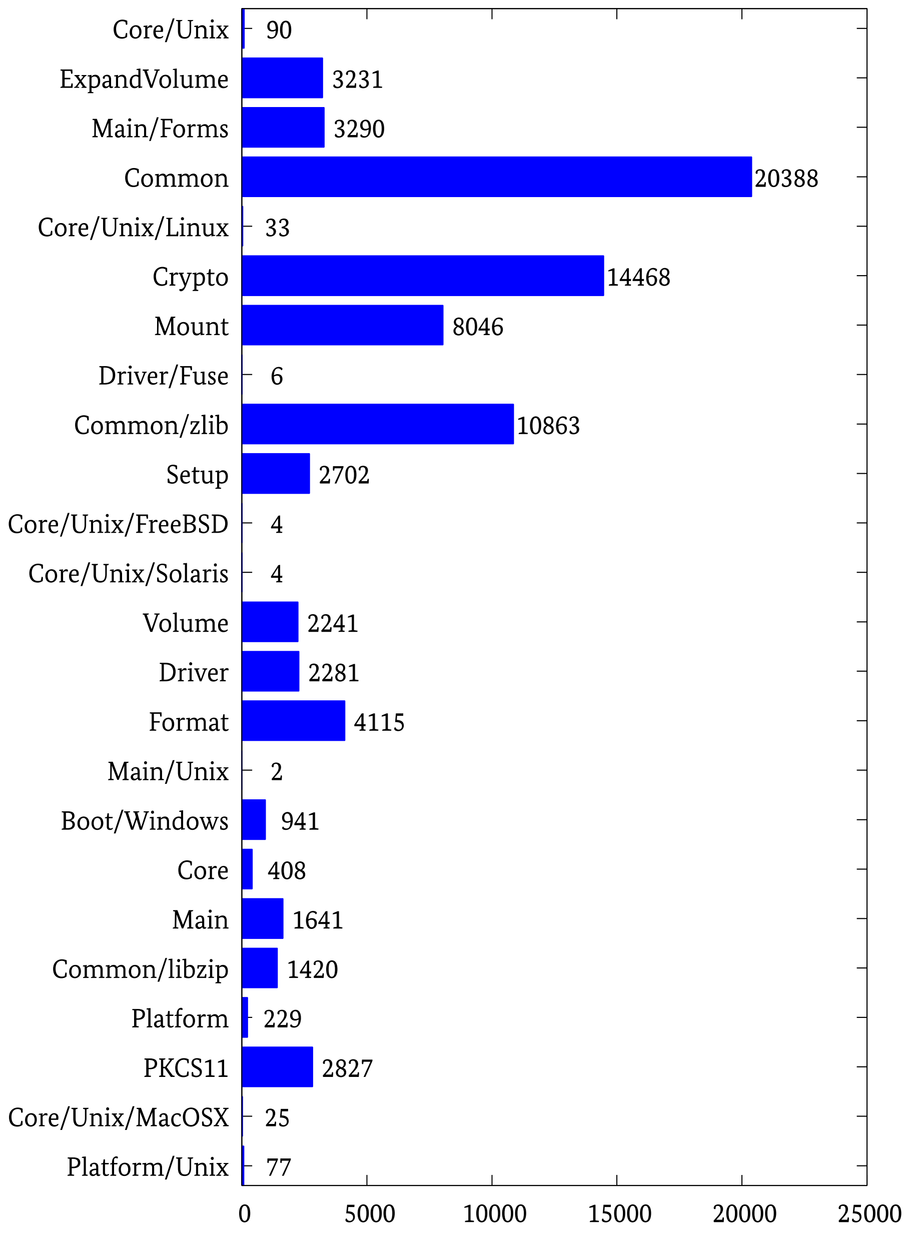


Figure 6: Number of lines changed per source code directory.

The files in the `Common` subdirectory were altered the most. Figure 7 shows the number of line changes per file in the `Common` directory. In this figure, only changes in source-code files are shown, header files were omitted.

The file `Tests.c` was modified as new test cases were added. The *PKCS5* implementation also changed heavily. The file `Crypto.c` also received substantial changes, removing or adding interfaces to different ciphers.

The most changes within the `Common` subdirectory can be found in the file `Dlgcode.c`, which mostly handles the creation and processing of user dialogs. The second most changes can be found in the file

`BootEncryption.cpp`, where handling, processing and verification of encrypted disks is implemented. This file was continuously refactored.

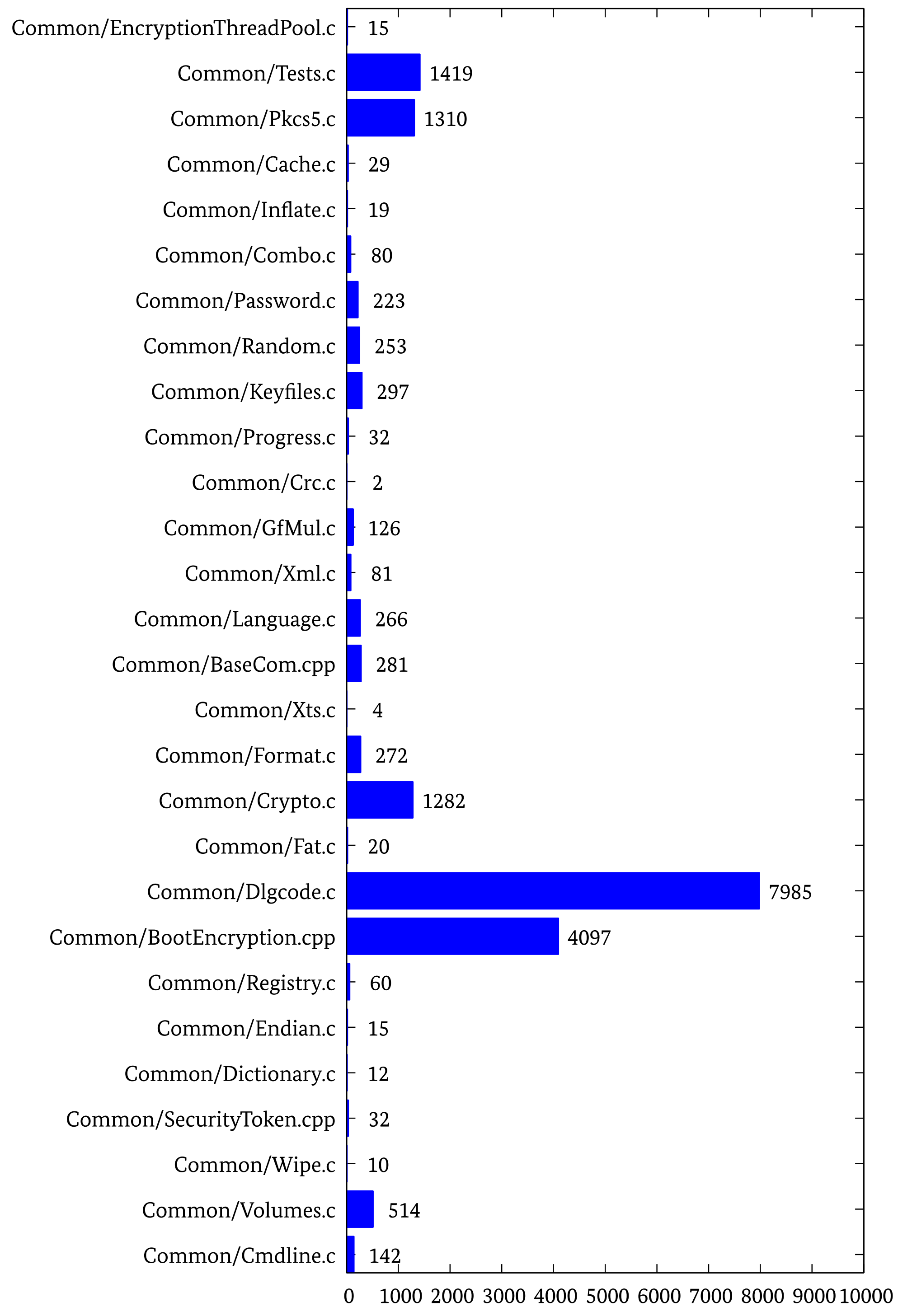


Figure 7: Number of lines changed per file in the `Common` *directory.*

2.4 Categorization of Source Code Commits

Since the beginning of the VeraCrypt project up to its version 1.23, 1276 commits have been committed to VeraCrypt's GitHub repository. In order to filter commits that are important for this project, we manually investigated and tagged each individual commit. This was done independently by two researchers.

Each researcher first investigated each commit message for whether it contains enough information to tag the commit accordingly. If the researcher found the commit contains too little information to tag the commit, the files changed by the commit were examined too. Each commit got at least one tag. A code book containing the tags was created by the two research successively while they analyzed the commits. Both researchers discussed and agreed on the tags. After the tagging was done, both cross-checked the tags they applied to the commits and aligned their individual tagging when they diverged. Tagging was done progressively; in case of doubt a tag was assigned rather than omitted. The results are presented in the following.

2.4.1 Tags

The following tags were used for tagging commit messages:

Relevant for the security audit

crypto: Commits adding, deleting or modifying cryptography-related parts of VeraCrypt. This includes hash functions, encryption or decryption routines and also parts of the software using those implementations, e.g., a GUI element for selecting an algorithm.

security: Commits fixing security related issues or potential issues.

boot: Commits regarding parts for the boot process when using VeraCrypt for system encryption.**static:** Commits applying changes to account for results from static analysis tools. These changes lead to better code quality and reduced the risk of security issues in most of the cases.

driver: Commits regarding the driver for the abstraction between VeraCrypt and the operating system.

Operating systems

windows: Commits targeting explicitly the Windows platform.

linux: Commits targeting explicitly the Linux platform.

macos: Commits targeting explicitly the macOS platform.

freebsd: Commits targeting explicitly the FreeBSD platform.

Miscellaneous

code: Commits aimed on improving code quality.

volumes: Commits regarding VeraCrypt volumes.

partition: Commits regarding VeraCrypt partitions.

library: Commits changing external libraries or address version changes of external libraries.

version: Commits changing version numbers.

gui: Commits changing and extending the graphical user interface (GUI). This also includes language files for internationalization.

docs: Commits changing or extending documentation and `Readme` files.

merge: Commits merging other branches into the master branch.

compile: Commits affecting the build system and changes related to compiler use and parameterization.

2.4.2 Results and Use for the Further Analysis

For the security evaluation of VeraCrypt, the following tags are of specific interest:

- crypto,
- security,
- boot,

- driver and
- static.

By singling out all commits with these tags, it is possible to analyze which files contain the most changes relevant for the security assessment of VeraCrypt. The full statistics for these tags are provided in the Appendix in Section 9.1.1. The detailed results in the Appendix give an overview of how often individual files were changed by commits with the aforementioned tags. We used this data in the further analysis of changes and for our security evaluation.

2.5 Changes in Cryptographic Functions

In this section, the changes between TrueCrypt and VeraCrypt are discussed in regard to the cryptographic primitives, the generation of high-entropy randomness, the derivation of cryptographic material from user inputs, and hidden volumes. We follow a top-down approach by investigating the publicly available documentation of TrueCrypt and VeraCrypt[12] as well as the VeraCrypt Release Notes[13] and then cross-checking the findings with the source code for plausibility.

2.5.1 Cryptographic Primitives

TrueCrypt as well as VeraCrypt provide several symmetric block ciphers for encryption, a mode of operation for data stream encryption, as well as cryptographic hash functions.

Cryptographic primitives of TrueCrypt

TrueCrypt supports the following three symmetric encryption schemes:

- AES (1),
- Serpent (2), and
- Twofish (3).

Since Version 7.1a of TrueCrypt, XTS (4) is used as mode of operation for the block ciphers. The encryption process uses two ciphers in cascade (5) in the following five different combinations:

- AES-Twofish,
- AES-Twofish-Serpent,
- Serpent-AES,
- Serpent-Twofish-AES, and
- Twofish-Serpent.

The mode of operation used before Version 7.1a of TrueCrypt was the LRW mode (6) in versions 4.1 to 4.3a and the CBC mode (7) in versions 4.0 and earlier. Even though in the most recent version containers are created using the XTS mode, TrueCrypt is backward-compatible with older versions and is able to handle containers using LRW and CBC modes.

There are three cryptographic hash functions available in TrueCrypt:

- RIPEMD-160 (8),
- SHA-512 (9), and
- Whirlpool (10).

[12] https://www.veracrypt.fr/en/Documentation.html
[13] https://www.veracrypt.fr/en/Release Notes.html

These hash functions are used, e.g., for key derivation and random number generation.

Cryptographic primitives of VeraCrypt

VeraCrypt provides the following encryption schemes (schemes provided in addition to those in TrueCrypt are written in bold letters):

- AES,
- Serpent,
- Twofish,
- **Camellia** (11), and
- **Kuznyechik** (12).

VeraCrypt uses the same mode of operation for the symmetric block ciphers as TrueCrypt, the XTS mode of operation. There now are ten different combinations of the symmetric block-cipher algorithms used in cascade:

- AES-Twofish,
- AES-Twofish-Serpent,
- **Camellia-Kuznyechik**,
- **Camellia-Serpent**,
- **Kuznyechik-AES**,
- **Kuznyechik-Serpent-Camellia**,
- **Kuznyechik-Twofish**,
- Serpent-AES,
- Serpent-Twofish-AES, and
- Twofish-Serpent.

The cryptographic hash functions available for use in VeraCrypt are:

- RIPEMD-160,
- **SHA-256**,
- SHA-512,
- **Streebog** (13), and
- Whirlpool.

Summary

The two block ciphers Camellia and Kuznyechik and the two cryptographic hash functions SHA-256 and Streebog have been added to VeraCrypt compared to TrueCrypt. The Magma (GOST) (14) cipher also had been added in commit `0b2c8b09`[14] in VeraCrypt version 1.18a, but was deprecated with commit `d18ecc1a`[15] in version 1.19 in response to a security audit (15); however, the source code is still present in the code repository for compatibility reasons. See Sections 4.1 and 4.2 for a detailed description and Section 4.5 for an analysis of the cryptographic functions in VeraCrypt.

[14] commit `0b2c8b09c6eb3ddce22fa88c34a640881f8f2177`
[15] commit `d18ecc1a37b5f83d70b204f0bcb097fb8525314f`

The source code files for the encryption schemes and the hash functions are located in directory `src/Crypto/`; the implementation of the XTS mode of operation is located mainly in file `src/Common/Xts.c`, the cascade cipher in file `src/Common/Crypto.c`.

2.5.2 Random Number Generation

TrueCrypt and VeraCrypt are using a Random Number Generator (RNG) for the generation of cryptographic keys, keyfiles, and salts. They maintain their own pool of entropy fed from various sources of entropy depending on the operation system.

Random number generation in TrueCrypt

The size of the entropy pool is 320 B. On all supported operating systems, the sources of entropy include mouse movements and keystrokes. On Linux and macOS, further entropy is derived from the OS sources `/dev/random` and `/dev/urandom`. On Windows, also entropy from the MS Windows CryptoAPI, network interface statistics, and further sources is used.

TrueCrypt provides functions to add entropy byte-wise to the pool and a function to “mix” or “diffuse” the entropy in the pool, i.e., to spread added entropy over the entire pool, using a cryptographic hash function. Before entropy is obtained from the pool, always new entropy from the respective sources is added and mixed to the pool. The construction is based on work by Gutmann (16) and Ellison[16] from the 1990s.

Random number generation in VeraCrypt

The RNG has received some modifications in VeraCrypt: SHA-256 has been added as choice for the hash function for mixing the entropy pool. The Windows source code has been adapted for a 64-bit execution environment and the CPU instructions *RDRAND* or *RDSEED* (17) as well as an entropy source based on CPU timing jitter have been added as additional entropy sources.

Summary

In particular the Windows version of the random number generator has been changed in VeraCrypt by adding additional sources of entropy. See Section 4.4 for a detailed description and analysis of random number generation in VeraCrypt and Section 4.5.4 for randomness tests.

The Windows random number generator is mainly implemented in file `src/Common/Random.c`, the Unix (Linux/macOS) version in file `src/Core/RandomNumberGenerator.cpp`.

2.5.3 Key Derivation

TrueCrypt and VeraCrypt are using a Key Derivation Function (KDF) to derive an encryption key from a password chosen by the user. This so called *header key* is then used to encrypt the master key and other metadata of a volume.

Key derivation in TrueCrypt

TrueCrypt is using PBKDF2 as defined in PKCS #5 V2 (RFC 2898) (18) as key derivation function with 512 bit salt values. Its construction is based on the iterative evaluation of an HMAC (19) function. TrueCrypt allows to choose any of the supported hash functions as primitive in the HMAC function. The iteration count is fixed and generally set to 1000 (with the exception that 2000 iterations are used for RIPEMD-160 when not used for boot volumes).

Key derivation in VeraCrypt

VeraCrypt is still using PBKDF2 for key derivation, but the implementation has significantly been modified. Support for the newly introduced hash functions SHA-256 and Streebog has been added and legacy code for SHA-1 has been removed. The iteration count has been increased (while providing a TrueCrypt

[16] http://world.std.com/~cme/P1363/ranno.html

compatibility mode) and optionally now the user can chose the number of iterations using a Personal Iterations Multiplier (PIM, see Section 4.3.1). Several optimizations for speed and memory usage have been introduced, e.g., pre-computation of hash-functions states for repetitive HMAC computations.

Summary

The code base for the key derivation function has significantly been altered for VeraCrypt. While RFC 2898 has been obsoleted by RFC 8018 (20), the basic construction of PBKDF2 has been taken over by RFC 8018. However, nowadays key derivation functions that impose not only cost in time but also in memory are preferred, e.g., bcrypt, scrypt and Argon2; Argon2 is recommended, e.g., in BSI TR-02102-1 (21). See Section 4.3 for a detailed analysis of the key derivation in VeraCrypt.

PBKDF2 is mainly implemented in file `src/Common/Pkcs5.c`.

2.5.4 Hidden Volumes

An additional encrypted volume can be hidden inside an encrypted volume in order to allow the user to hide sensitive data even if he is forced to reveal his password.

Hidden volumes in TrueCrypt

For each encrypted container, TrueCrypt provides two slots for encrypted volume headers. If no hidden volume is used, the second slot is filled with an encrypted zero block using a random key. Otherwise, two volume headers are stored and the user can enter the password for either volume in the mount dialog and the volume associated with the entered password is mounted. If only the password for the main volume is provided, the setup behaves like a single volume without hidden volume. This means that data in the hidden volume might get damaged when data is written to the mounted volume. In order to prevent data loss in the hidden volume when data is written to the main volume, both passwords need to be provided at mount time.

Hidden volumes in VeraCrypt

VeraCrypt uses the same approach for hidden volumes as TrueCrypt. However, a critical bug that allowed an attacker to detect the presence of a hidden volume was fixed in VeraCrypt version 1.18a. Now, the second slot is not only simply filled with an encrypted block of zeros when no hidden volume is used but a proper volume header is created and encrypted with a random key in order to prevent distinguishing attacks.

Summary

VeraCrypt received a bug fix to prevent detection of the presence of hidden volumes.

2.6 Changes Related to Application Security

VeraCrypt introduced multiple changes affecting the security of the application that are not directly related to cryptographic or security functions. We identified the changes using a detailed evaluation of commits tagged with “security” as explained in Section 2.4.

Operating-system specific changes

Some commits marked with the tag “security” are specific to a certain operating system. Table 2 gives an overview over the operating-system specific changes.

Table 2: Operating-system specific source-code commits tagged with “security”.

Operating Sytem	*Amount*	*Relative Amount*
Windows	76	66%
Linux/macOS	9	8%
Non specific	31	26%
Total	117	100%

The largest portion of these changes with 76 of the commits are related to Windows. These commits are often vulnerability fixes or aim on preventing crashes due to race conditions or uninitialized memory. Also, Windows-specific protection mechanisms are activated or mount problems are fixed.

The high number of Windows-related commits suggests that most users of VeraCrypt are using Windows, so it appears that most bugs are reported for this operating system. However, another reasonable explanation might be that as Windows is more widely used, security researchers focus on finding vulnerabilities specifically in the Windows version of VeraCrypt.

Added exploit protections

With VeraCrypt, multiple protections against exploits were introduced. In June 2017, Address Space Layout Randomization (ASLR) was integrated, which is a technique to impede the execution of exploit code by randomizing the address of data in the process memory with every program start. This measure makes it difficult for an attacker to predict the process memory layout which is hindering the development of exploits (22).

Another important mitigation against memory-corruption vulnerabilities is a non-executable stack. Many exploit techniques rely on a stack that is both writable and executable; marking the stack non-executable can make it difficult to exploit vulnerabilities. The VeraCrypt developers fixed a related problem under Linux in commit `ba1fbb68`[17].

For the Windows driver, VeraCrypt changes the way non-paged memory is allocated. Windows 8 introduces the functionality to allocate non-paged memory from a no-execute non-paged pool[18]. VeraCrypt uses memory from this pool to further mitigate exploits.

In Windows, it is possible to use the so-called Secure Desktop mode[19]. This mode is used to protect against password sniffing attacks or GUI manipulations by other processes and is used in VeraCrypt.

Change to safe functions

Some functions, especially functions handling character strings, are deemed unsafe. For example, some unsafe string functions do not consider the length of the buffer they are writing to, resulting in buffer overruns, which can in turn lead to vulnerabilities. Over time, the developers of VeraCrypt changed calls to these unsafe functions to safer variants.

For example, in commit `016edc15`[20], the changes shown in Listing 1 were issued. The unsafe functions `strcpy` and `strncat` were replaced with safer variants, in this case `StringCbCopyA` and `StringCbCatA` from the Windows API strsafe.h[21]. These functions ensure that no data is written to the memory after the end of the allocated buffer. These calls to unsafe functions were found with static analysis tools (cp. commit with tag "static" as explained in Section 2.4).

```
- strcpy (outputFile, szDestDir);
- strncat (outputFile, OutputPackageFile, sizeof (outputFile) - strlen
(outputFile) - 1);
+ StringCbCopyA (outputFile, sizeof(outputFile), szDestDir);
+ StringCbCatA (outputFile, sizeof(outputFile), OutputPackageFile);
```

Listing 1: Example for a switch from unsafe to safe functions.

[17] commit `ba1fbb688edf7db0edf3ea9d24a95d5a5ef2260c`
[18] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/no-execute-nonpaged-pool
[19] https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-switchdesktop
[20] commit `016edc150b034d7401a1652bd3482d613ff4b9d4`
[21] https://docs.microsoft.com/en-us/windows/desktop/api/strsafe/nf-strsafe-stringcbcopya

Improving robustness

Some commit messages refer to fixes to source code causing program crashes. To lower the possibility of crashes or vulnerabilities, multiple measures were introduced by the developers:

- Checks for null pointers were introduced to prevent null pointer de-references.
- On multiple occasions, variables are now correctly initialized.
- Multiple memory buffers are initialized with zeros.
- Correct error codes are returned when handling volumes.
- SSE[22] instructions are only used if supported by the CPU.
- More potential runtime exceptions are caught.

Multiple commits refer to very specific fixes of crashes like a problem crashing Kaspersky Internet Security 2016 in commit `723fcfa6`[23] or a rare failure of a call to the function `FormatMessage` in commit `59611b8b`[24].

Adding safe overwrite

On multiple occasions, changes were applied to overwrite obsolete data on disk or on memory thereby making it harder to access it by an attacker.

- The password is wiped from memory to protect against attacks in which the clear text password is extracted from the VeraCrypt process.
- The volume header is overwritten during the encryption of a partition. The number of times this happens has been lowered by the VeraCrypt developer.
- Cached PIM values are wiped when passwords are wiped.
- The path to keyfiles is wiped from memory.

Fixed vulnerabilities

Various vulnerabilities were fixed over time by the VeraCrypt developers. There does not seem to be a clear structure when which vulnerability is found and fixed. It appeared that fixes take place in an ad hoc manner after a vulnerability is reported.

- Sensitive data is erased from memory using the method `burn` instead of the method `memset`. Both methods can in general be used to overwrite data in memory, but `memset` may be removed by compilers to increase performance, while `burn` cannot be removed and overwrites the given memory as intended.
- Kernel pointer disclosures were fixed.
- A vulnerability in `CryptAcquireContext` was fixed. A buffer overrun in the XML parser was fixed. Error handling for the initialization of the random number generator was added.
- An evil-maid detection mechanism was introduced.
- A local elevation of privilege vulnerability because of an incorrect impersonation token was fixed.
- A DLL hijacking vulnerability affecting the installer was fixed, which was reported as CVE-2016-1281. Among other mitigation strategies, the new Windows API is used to prevent such hijackings.
- A vulnerability was fixed enabling an attacker to detect if a hidden volume is present.
- Multiple memory leaks were fixed.

[22] https://docs.oracle.com/cd/E26502_01/html/E28388/eojde.html
[23] commit `723fcfa64dc2e4e4c6efc8a0c8d5cd05c0eaf944`
[24] commit `59611b8b378238e5a589a87061d06fe4f337d1a0`

The Open Crypto Audit Project also found several vulnerabilities which were fixed in VeraCrypt after they were reported:

- There was a possible “blue screen of death” attack.
- An integer overflow issue was fixed.
- The device name is now checked correctly.
- The bootloader decompressor was made more robust.

Summary

There have been multiple improvements to the application security of VeraCrypt. The focus seems to be rather on Windows than on other platforms. To find vulnerabilities or problems in the code, the developers claim to use static analysis tools, as well as manual methods.

3 Security Model

This chapter describes the security model used to steer the following security analysis. Section 3.1 explains the considered application use cases in which VeraCrypt is usually operated and where VeraCrypt security properties are expected to uphold against attacks on an IT system. The security goals collected in Section 3.2 systematically outline the protection to expect from using VeraCrypt. These goals are challenged by attack scenarios or threats as described in Section 3.3 that can take place in the various use cases and can impose a risk to achieving the described security goals. VeraCrypt is tested for weaknesses and vulnerabilities that may allow implementing these attack scenarios, see Chapters 4 and 5.

Since the aim of this project is not a formal security verification of VeraCrypt, functional security requirements are not derived. Instead, an attacker-centric analysis starting from described attack scenarios is performed which leads directly to the security assessment presented in the remainder of this report.

3.1 Application Use Cases

In the following, multiple use cases are discussed regarding their specific implications on the use of VeraCrypt. At a glance, these are:

- **Personal computers.** Encryption of mobile or stationary personal computers with one or multiple users. Encryption can be restricted to single volumes or cover the whole system disk, see 3.1.1.
- **External data storage devices.** Encrypted external storage devices, such as external hard disks or USB flash drives are used to transport or backup data, see 3.1.2.
- **Sharing of encrypted data**. Data are shared with encrypted volumes stored in online services such as file sharing services, or with portable storage devices, see 3.1.3.
- **Server systems and virtual machines.** Encryption of volumes on a server system in a data center or on a virtual machine in a virtualization environment, see 3.1.4.
- **Public systems.** Publicly accessible systems such as kiosk systems with encrypted volumes, see 3.1.5.

Finally, **maintenance** and **decommissioning** are discussed as further aspects of IT system use that may apply to any of the aforementioned use case scenarios, see 3.1.6.

Considering a use case for the security model does not imply that from a security perspective it is appropriate to use VeraCrypt in that scenario. It only implies that VeraCrypt can be operated without obvious downsides making its use infeasible from a practical standpoint.

3.1.1 Personal Computers

We differentiate the uses cases for VeraCrypt on personal computers in two dimensions:

1. the overall system usage scenario in terms of portability, and the number of users, and
2. the scope of the encryption.

All possible configurations of the encryption scope of VeraCrypt can be combined with the different system usage scenarios, and each combination achieves a different level of protection.

System usage

For the system usage, two crucial aspects are relevant: the degree of the system's mobility defining the degree of physical access by a potential attacker, and the number of users, i.e., single- and multi-user systems.

- **System mobility and physical access.** An IT system protected by VeraCrypt can be a stationary system, typically a work station computer at an office work place. The system is not routinely removed from this

designated environment, and removal is easy to discover. Physical access to the system is often limited to selected persons by access control policies enforced through technical or organizational measures. This limits the circle of people who have the ability to physically interact with the system, perhaps in a malicious way. However, stationary systems often stay unattended for a longer time period, for instance, outside office hours or in case the regular user is on vacation. They are also difficult to lock away during times of temporary disuse.
By comparison, portable systems such as laptops are used rather differently: Their physical location often changes, and it is not unusual to carry them outside a controlled physical environment such as a secured office building. Portable systems might even be used in public places like public transport. In general, policies to limit physical access to portable systems are more difficult to enforce in most usage scenarios. Nevertheless, portable systems can be kept in closer proximity to their legitimate users, because they can be taken along when the user changes location. Because of their smaller dimensions compared to work stations, they can also easily be locked away in secure places during times of temporary disuse, for example in a locked cabinet.

- **Single- or multi-user systems.** Personal computers might be used by a single user alone or by multiple users. The latter is often the case for stationary systems, for instance in shared workspace environments. But also if administrators have access to a system from time to time, they might do so by means of dedicated user accounts allowing them to log into the system.
  In case a system is used by different users, these users might have different access rights to the system's resources or services, for example, not all users might have the rights necessary to administrate the system. In case multiple users require access to the same resource protected by VeraCrypt, they need to share access credentials such as the VeraCrypt password and perhaps necessary keyfiles. A common example is a system with a full system disk encryption, where users need to share the volume password to boot into the operating system.
  The circle of users for an IT system might also change over time. New users might be granted access to the system, while access rights of former users might be revoked due to changes in the organization.

Encryption Scope

In all these mentioned usage scenarios, the scope of the encryption can be set rather differently. A full system encryption that encrypts all data on the system, including the operating system, has the broadest scope. However, in multi-user scenarios this comes with the disadvantage that VeraCrypt credentials need to be shared among all legitimate users.

It is also possible to encrypt only specific data volumes on the system, such as a data partition on the system's hard drive, or to create a dedicated encryption container where sensitive data is stored. For both options, there is a considerable risk that confidential data is accidentally stored in unencrypted areas of the system's drives. However, an encrypted container can be easily moved from one system to another, which might have advantages in certain usage scenarios.

Summary of variants

- System usage
  1. System mobility
     - Stationary system
     - Portable system
  2. System sharing
     - Single user using a dedicated personal system
     - Multiple users sharing a single system
- Encryption scope
  1. Full system encryption

2. Encrypted data volumes
3. Encrypted VeraCrypt containers

3.1.2 External Data Storage Devices for Personal Use

External data storage devices such as external hard disks, USB flash drives or memory cards are typically used to transport data from one IT system to another, or to backup data from one system. Apart from executing a firmware necessary to interface to the connected host IT system and transferring data from and to the device, external storage devices do not process the stored data or compute any program code provided from outside the device.

VeraCrypt can be used either to create an encrypted partition on an external storage device, or to create an encrypted data container in an existing file system on the device. The device might be carried outside a controlled (corporate) environment, e.g., to public places. Devices are prone to theft but also to getting lost, in particular when devices have small physical dimensions.

An external storage device might also be shared with other users. This scenario is discussed in the following Section 3.1.3, because it imposes specific challenges.

Summary of variants

1. Encrypted container stored on external storage device
2. Encrypted volume on external storage device

3.1.3 Sharing of Encrypted Data

Users could also distribute or share VeraCrypt containers, e.g., via network services, be it in a corporate network or the internet. Users could upload encrypted containers to file sharing services such as Dropbox, Google Drive or Nextcloud, and might grant other users access to these network shares. In intranets, file servers running network protocols such as SMB can be used to store encrypted containers in remote directories accessed by multiple users. Another possibility is to send and receive encrypted containers by communication means such as e-mail or instant messengers.

For these scenarios of sharing data online, container encryption is the only appropriate operation mode of VeraCrypt. While it might be possible to share snapshots of virtual drives also containing VeraCrypt encrypted partitions, this approach has no practical benefit compared to using container encryption.

It is reasonable to assume that providers of online services used to share encrypted containers have full access to these container files, i.e., to the cipher text. Moreover, they have the ability to change the overall content of shared directories, i.e., they can add, alter or remove arbitrary files, including container files. In addition, during file transfer over a network, weak or no network encryption might be applied thereby allowing others than the legitimate users to get access to encrypted containers.

Instead of sharing encrypted data with online services, sharing can also take place with encrypted storage devices (cp. 3.1.2). For instance, an encrypted USB flash drive can be used to handover data from one person to another.

For all of these scenarios, sharing of data requires to share the credentials for decryption, too. This means that more than one user is in possession of the VeraCrypt password and perhaps keyfiles required to decrypt a volume or container.

Summary of variants

1. Sharing of encrypted container files with an online file sharing service such as Dropbox, Google Drive or Nextcloud

2. Sharing of encrypted containers with fileservers in an intranet using conventional file sharing services such as Windows file sharing or network folders
3. Distribution of encrypted containers via e-mail or messaging services
4. Sharing of an encrypted external storage device

3.1.4 Encrypted Server Systems and Virtual Machines

VeraCrypt can also be used to encrypt server systems. For a Windows server system, this could be a full system disk encryption. For both Windows and Linux servers encrypted data partitions and containers can be used as well.

Server systems are usually operated in a protected physical environment, like a data center or single server room, with access policies restricted to administrative and maintenance personnel. However, server systems are also a subject to theft and tampering in these environments, when measures and policies to enforce physical security are insufficient. Attacks might not necessarily focus on the IT system as such, but on server hardware as an asset.

Using VeraCrypt on server systems imposes operational challenges, e.g., entering passwords and perhaps providing keyfiles to unlock encrypted partitions or containers at system boot time or when required for operation. For instance, when using a full system disk encryption, a password needs to be entered during system startup through a console connected to the server that can be accessed remotely or on site.

Encrypting disks with VeraCrypt on a virtual machine is a similar scenario to disk encryption on a server system. Challenges to unlock devices—especially if required during boot-up time—remain. However, on a virtual machine, the memory can be dumped easily by persons who administer the virtualization environment, e.g., by creating a snapshot of the running machine.

Summary of variants

1. Full system encryption of a Windows server system
2. Encrypted data partition or container on a server system
3. Virtual machine instead of a physical server system.

3.1.5 Publicly Accessible Systems

Publicly accessible systems such as kiosk systems can also be an application environment for VeraCrypt. These systems are placed in a public setting so that users having access to this space can interact with their interfaces (touchscreens, keyboards, screens, printers, etc.) to use their services. The computing hardware itself is usually surrounded by physical protections such as locked encasements.

However, these systems can be subject to theft, tampering and other kinds of misuse. While usually only providing a frontend to remote services connected via network, these systems might nonetheless store sensitive information such as configuration data, usage protocol records or even credentials to access services or data.

Similar to server systems, publicly accessible systems need to be provided with passwords and keyfiles to unlock the encrypted volume or container if required for the systems' operation. For instance, if a full system encryption is used, the credentials need to be provided during startup. This can be done, e.g., by instructed personnel in the public area that has access to respective system interfaces such as a keyboard.

Summary of variants

1. Full system encryption of the publicly accessible system
2. Encrypted data partition or container on the publicly accessible system

3.1.6 Further Usage Aspects

Beside these system use cases, specific steps in an IT system's lifecycle can be of importance from a security perspective: During maintenance it can be necessary to provide specialized staff access to VeraCrypt volumes, for instance, in case the system planned for maintenance has a full disk encryption with VeraCrypt.

A further stage to consider is the decommissioning of IT systems or hard drives that contained VeraCrypt volumes. Here, the question is as to whether it is required to erase the whole hard disk, perhaps with special measures and equipment.

3.2 Security Goals

Using VeraCrypt aims on achieving multiple security goals that are discussed in the following; an overview is given in Table 3. Complying with a security goal requires VeraCrypt to implement security measures while potential attack scenarios or threats (discussed in Section 3.3) impose a risk to violate the security goal.

Table 3: Overview of the security goals relevant to the use of VeraCrypt

Priority	*Security Goal*	*Potential Attack Scenarios (Threats)*
Primary	Data confidentiality	• System loss or theft (see 3.3.1) • Multi-access attacks (see 3.3.2) • Side-channel attacks (see 3.3.7) • Online attacks (see 3.3.8)
Secondary	Data integrity	• Targeted data alteration (see 3.3.3)
	Data availability	• Blocking data access (see 3.3.6)
	IT system integrity	• Privilege escalation (see 3.3.4) • Preparing targeted attacks (see 3.3.5)

3.2.1 Primary Security Goal

The primary goal of using VeraCrypt is to protect the **confidentiality** of data stored on a VeraCrypt volume against unauthorized persons. Only authorized persons in legitimate possession of the passwords and keyfiles to decrypt VeraCrypt volumes must be able to access data protected by VeraCrypt. This protection promise stretches over multiple phases of an IT system's lifecycle in particular operation, maintenance, hand-over and decommissioning.

3.2.2 Secondary Security Goals

VeraCrypt achieves further security goals besides protecting the confidentiality of data. To some degree VeraCrypt protects the **integrity** of the encrypted data. It does not prevent or even detect malicious alterations of cipher text stored on disk. However, because VeraCrypt uses XTS as cipher mode alterations of the cipher text cause unpredictable alterations of the plain text thereby largely reducing the likelihood that targeted manipulation attempts succeed. For instance, attacks to modify program code stored in the plain text data are likely to fail. Because the impact of alterations on the plain text is not only unpredictable but also broadly distributed, alterations more likely cause observable system failures, which are easier to detect.

Also to a very limited extent VeraCrypt ensures the **availability** of the encrypted data, because VeraCrypt provides mechanisms to backup the volume header. In case of data loss affecting the header, the volume can still be decrypted. It is also possible to backup volume headers with special passwords dedicated for backup purposes.

Furthermore, the use of VeraCrypt on an IT system shall not lead to additional risks to the **integrity of the IT system** as such. VeraCrypt must not extend the attack surface of the IT system by creating new entry points for adversaries to interfere with the system's operation.

3.3 Attack Scenarios

This section describes the attack scenarios and threats that target the security goals of VeraCrypt described in Section 3.2.

3.3.1 System Loss or Theft

Loss or theft of systems or storage devices protected by VeraCrypt is the most relevant attack scenario. An attacker might gain possession of such a system either by accident or by a targeted action. Especially mobile systems (cp. 3.1.1), external data storages (cp. 3.1.2) and kiosk systems (cp. 3.1.5) are prone to these attacks, but also VeraCrypt volumes stored in online services (cp. 3.1.3) can easily be accessed by unauthorized third parties. If IT systems or single data storage devices are decommissioned, they might also get easily into the hands of a potential adversary.

These attack scenarios can be classified into two major categories, each with very different implications:

1. Scenarios where the attacker has only access to a VeraCrypt-encrypted disk or volume while a potentially attached computing system is disabled or does not run VeraCrypt while the attack takes place.
2. The attacker gets into possession of a system with a running instance of VeraCrypt actively encrypting and decrypting a volume.

The first scenario is commonly associated with opportunistic attacks such as pick pocketing in public places. Often, the attacker does not know that the targeted system is encrypted with VeraCrypt as the thief is more interested in the value of the computing hardware rather than the system running on it. However, attacks can also be strategic, and in case the attacker targets the data on the system, theft of a running system provides more opportunities for successful attacks on the stored data.

VeraCrypt not actively running on obtained system / single disk or volume obtained

In case the attacker gets in possession of an IT system with encrypted VeraCrypt volumes but the system does not run a VeraCrypt instance during the time of the attack, a **brute-force attack** on the password and/or keyfiles can be applied. The attacker generates potential candidates for the password or keyfile and tests whether they are the correct credentials. To this end, the attacker can use password dictionaries. The attacker might also try to guess the master key used for the data encryption instead of attacking a password or keyfile.

Brute-force attacks are supported by weak passwords or weak random number generators used to create keys, but also weaknesses in cryptographic algorithms, both allowing to limit the search space for a key, password or keyfile.

As an alternative to brute-force attacks, the attacker can also examine the obtained system for plain text data from the encrypted VeraCrypt volume or credentials in plain text (**forensic attack**). This data might be unintentionally stored on the system in plain text if the system has unencrypted volumes, and user programs or the operating system accidentally copied data to these volumes without erasing it on shutdown.

VeraCrypt is running on obtained system

In case the attacker comes into possession of a running system, further attack strategies can be applied. The attacker can try to read out the content of the main memory for instance by means of a cold-boot attack (cf. (23)) or attacks using system interfaces with entry points such as DMA (see (24) for an overview). It depends on the system's hardware and software configuration whether these attacks succeed, but, in any case, they require a high level of technical skills.

If the attacker manages to read out the memory content, she can access the encryption master keys for those VeraCrypt volumes that were mounted during the time of the attack. If no volumes were mounted, but the legitimate user accessed them before the attack took place, there is a chance that keys and passwords are still stored in the main memory if they were not properly deleted after un-mounting.

A further option is to attack the operating system to unlock the screen and input devices so that the attacker can fully interact with the system and thereby obtains access to potentially mounted volumes. VeraCrypt cannot protect against these kinds of attacks.

Summary

- **Description:** Adversary gains possession of an IT system or storage device with VeraCrypt-encrypted volumes and attempts to decrypt the data.
- **Affected security goal:** Data confidentiality
- **Most relevant scenarios:**
  - Personal computers (see 3.1.1)
  - External storage devices (see 3.1.2)
  - Sharing of encrypted data (see 3.1.3)
  - Public systems (see 3.1.5)
  - Decommissioning (see 3.1.6)

3.3.2 Multi-Access Attacks

Multi-access attacks assume that an attacker gets access to a VeraCrypt-protected IT system at least more than one time. During the first visit, the attacker alters the system so that passwords or keyfiles entered to the system are stored in a way the attacker can freely gather them during a second visit.

The second visit does not need to be physical. The attacker might also use remote data connections, for instance via wireless networks, to receive stolen credentials. In case the attacker copied the encrypted data already during the first visit, the attacker can then immediately start to decrypt it. If the attacker gets physical access during the second visit, she can steal the IT system together with the collected credential data.

Conventional methods to implement multi-access attacks are, for instance, attaching a **key logger** to the system, **altering the operating system** to execute a key logging software, or installing a **forged password input screen** into the bootloader software if a full-system encryption is present.

Systems that are not stored in secured areas with limited access, or are unattended for a longer time period are particularly prone to these kinds of attacks. A case in point are notebooks stored in hotel rooms where manipulations could be performed by hotel staff (an attack pattern commonly called "evil maid")[25]. Multi-access attacks can also be particularly relevant if IT systems are handed over to less trustworthy third parties for maintenance reasons.

If the attacker was formerly a legitimate user of the IT system, she might also **swap the header** of an encrypted VeraCrypt volume to a volume header bound to the credentials the attacker once used to decrypt the data. If only the encryption password has changed during transfer of ownership but the volume was not properly re-encrypted, the volume master key still remains unchanged and the attack succeeds.

[25] This attack was already showcased for TrueCrypt, see http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html

Summary

- **Description:** An adversary gets temporary, physical access to an IT system with VeraCrypt-encrypted volumes and manipulates it in a way that it stores entered passwords and keyfiles in plaintext. During a second physical or remote visit, the attacker retrieves these credentials and decrypts the VeraCrypt-protected data.
- **Affected security goal:** Data confidentiality
- **Most relevant scenarios:**
  - Personal computers (see 3.1.1)
  - Server systems and virtual machines (see 3.1.4)
  - Public systems (see 3.1.5)

3.3.3 Targeted Alteration of Data

In general, any attacker in possession of a VeraCrypt-encrypted volume can alter the encrypted data and thus the plaintext content without having the required encryption password or keyfiles. The question is whether an attacker might be able to alter data in an advantageous way. For instance, such a benefit could be to damage program code stored on the volume with the consequence that its execution fails.

If a weak cipher mode was used such as CBC an attacker can specifically switch single bits in the ciphertext[26]. Hypothetically, an attacker could also **move a specific part of the ciphertext** to another position on the volume for causing a targeted change in the plaintext though VeraCrypt's cryptographic algorithms shall prevent such attacks if implemented correctly. Another possibility is to **replay a part of the cipher text**, for instance, to reach a former state of the IT system.

In addition, there is a small but not unrealistic chance that an attacker succeeds in **predicting the position of certain files** on the hard drive, especially system files of a standard operating system installation. In this case, the attacker can try to damage these files on purpose. If systems are cloned from unified disk images, the likelihood of a successful attack further increases.

The described attacks usually serve the purpose to prepare other attacks. For instance, damaging parts of the operating system or setting it back to a former state might support attacks on the operating system itself, such as overriding one of its access control mechanisms. However, these attacks require a high level of technical skills, probable assumptions about the state and structure of the encrypted volume, and hence have a rather small chance of success.

Summary

- **Description:** An adversary gets access to a VeraCrypt-encrypted volume and alters its ciphertext to provoke a specific behavior of the IT system once it processes the data.
- **Affected security goal:** Data integrity
- **Most relevant scenarios:**
  - Personal computers (see 3.1.1)
  - Server systems and virtual machines (see 3.1.4)
  - Public systems (see 3.1.5)

[26] An example for this kind of attacks is given in https://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/

3.3.4 Privilege Escalation on Host System

An IT system running VeraCrypt might have multiple users with different privileges on this system, for instance administrator rights might not be granted to every system user. In such a scenario, a user with low privileges might try to **attack the VeraCrypt driver** running with administrator privileges in the kernel space to escalate her own privileges.

This can be achieved by injecting code through an unsafe driver interface to be executed in the higher privileged kernel execution space. Another possibility is to trigger functions of the driver that were accidentally exposed at the driver interface to perform actions not regularly allowed for the user.

For all these attack strategies, VeraCrypt serves as an entry point to perform other attacks. They are not directly aiming on VeraCrypt-encrypted data. However, in cases escalating local privileges also enables access to restricted mounts of VeraCrypt volumes, these attacks can also compromise the confidentiality of VeraCrypt-protected data.

Summary

- **Description:** By exploiting erroneous or missing input checks at the interface of VeraCrypt's kernel driver an attacker escalates her access privileges on the IT system running VeraCrypt.
- **Affected security goal:** IT system integrity
- **Most relevant scenario:** Personal computer as multi-user system (see 3.1.1)

3.3.5 Preparing Targeted Attacks

VeraCrypt can also provide **system-external entry points for attacks** to compromise IT systems running VeraCrypt. An attacker who has access to a VeraCrypt volume could try to manipulate this volume with the result that the input routines processing the volume on a target system, for instance the volume header parser, fall into an undefined state. Having this kind of weakness in an input routine may allow an attacker to plant and execute malicious code on the vulnerable target system.

Persons who have legitimate access to a VeraCrypt volume come into question for these attacks. Such a shared access might be the case in scenarios where volumes are intentionally shared (see 3.1.3).

This kind of attacks could also be used in **spear phishing attacks** when an attacker sends a victim a self-made malicious VeraCrypt volume containing the attack code, and the password or keyfiles necessary to decrypt it. If the attacker can trick her victim into using the volume, the attack could work out.

In general an attacker who has access to the encrypted volume but not decryption credentials could also perform this attack. It depends on the kind of weaknesses available to the attacker whether it can be performed without being able to decrypt the volume. This can be the case, e.g., if it is possible to jump to malicious code embedded in the ciphertext area of a container file.

Summary

- **Description:** An attacker tricks a VeraCrypt user into using a manipulated VeraCrypt volume and exploits a weakness in VeraCrypt enabling the attacker to control or alter the victim's IT system.
- **Affected security goal:** IT system integrity
- **Most relevant scenario:** Sharing of encrypted data (see 3.1.3)

3.3.6 Blocking Access to Data

By **damaging the volume header**, an attacker might try to block access to a VeraCrypt volume for its legitimate users. While there are other ways like physically damaging a device, or overwriting the whole volume space with random data, such an attack is overly effective since it damages the complete volume by a quick and easy to perform action. If no volume header backup exists, it is impossible for the victims to restore even a portion of the encrypted data.

Such attacks can be useful for people who have legitimate access to an encrypted volume but are asked to handover the encrypted volume against their own intentions, for instance, disgruntled employees who are about to quit. Another way for them to block access to legitimate others is to **change the encryption password** and to not inform responsible others about it, perhaps in conflict with a policy of their institution. If the institution does not backup volume keys, without cooperation of the attacker, access to the volume cannot be restored.

Summary

- **Description:** An attacker blocks access to VeraCrypt-protected data of legitimate users by damaging a VeraCrypt volume header or changing the encryption password in an unauthorized manner.
- **Affected security goal:** Data availability
- **Most relevant scenario:** Personal computers (see 3.1.1)
- **Note:** For decommissioning of storage devices, the described attack technique of erasing the volume header can be applied by authorized personal on purpose.

3.3.7 Side-Channel Attacks

Side-channel attacks can be performed against a system if an attacker is able to record physical or computational parameters while the system is operating on secret data, e.g., during key derivation or while encrypting or decrypting VeraCrypt volumes. If VeraCrypt's implementation is prone to side-channel attacks, by statistical analysis of these recordings the attacker can gather information about the keys used.

Examples for these parameters are power consumption and execution time. The attacker either needs to have physical access to the system or needs to be able to execute code—not necessarily privileged code—on the target system. For the latter, running a program in the web browser could be sufficient from a theoretical point of view. Also a virtual machine controlled by the attacker and running on a system that uses VeraCrypt (in the host system itself or in another virtual machine on the same physical host) can be an entry point for side-channel attacks. Nonetheless, these attacks require a high level of technical skills on part of the attacker, and potentially complex technical equipment.

Also recent attacks that are enabled by the CPU microarchitecture like Meltdown and Spectre[27] can be mounted against VeraCrypt. These attacks exploit features of a microarchitecture together with side channels like timing in order to extract data and potentially secret information by running malicious code on the target platform. These are generic attacks that do not exploit weaknesses in VeraCrypt itself but that can be used to attack VeraCrypt just as any other software running on a vulnerable system.

Summary

- **Description:** Adversary records parameters of an IT system running VeraCrypt to predict or reveal the VeraCrypt keys processed by the system.
- **Affected security goal:** Data confidentiality
- **Most relevant scenarios:**
  - Personal computers (see 3.1.1)
  - Server systems and virtual machines (see 3.1.4)

[27] https://meltdownattack.com/

3.3.8 Online Attacks

An attacker may also get access to encrypted data or data encryption keys by planting malware or backdoors into systems used to decrypt VeraCrypt volumes. Protection against these kinds of attacks is outside the protection scope of VeraCrypt. They are mentioned here for the sake of completeness.

Summary

- **Description:** Adversary infects an IT system running VeraCrypt with malware or gets otherwise local access to the system and can thereby read out data decrypted on the system or encryption keys.
- **Affected security goal:** Data confidentiality
- **Most relevant scenarios:**
  - Personal computers (see 3.1.1)
  - Server systems and virtual machines (see 3.1.4)

4 Security Analysis of Cryptographic Mechanisms

Several of the attack scenarios from Section 3.3 (3.3.1, 3.3.3, and 3.3.7) including the most relevant scenario "System Loss or Theft" are related to the use of cryptography. This section analyzes the cryptographic mechanisms used by VeraCrypt based on the available documentation and to some extend the source code[28]. The cryptographic mechanisms, e.g., encryption, hash-functions, random-number generators, nonces, salts, initialization vectors, and key derivation functions are investigated and assessed based on current recommendations.

4.1 Encryption Schemes

VeraCrypt is using symmetric block ciphers for bulk encryption of data using a mode of operation for operating on data streams. Optionally, several ciphers can be cascaded.

4.1.1 Block Ciphers

VeraCrypt offers five symmetric block ciphers, three of the finalists of the Advanced Encryption Standard contest by NIST, namely AES, Serpent, and Twofish, as well as Camellia and Kuznyechik. The block ciphers with their key and block sizes are listed in Table 4. AES is standardized, e.g., by NIST (1) and ISO (25) and recommended by BSI in TR-02102-1 (26). Camellia is standardized, e.g., by ISO (25) and the IETF (11). Kuznyechik is standardized, e.g., by GOST (27) and the IETF (12). There has been extensive cryptanalysis on these ciphers and currently they are all considered secure. There is some criticism on Kuznyechik due to the use of a secret algorithm for constructing S-boxes (28) (29); however, no resulting attack is known.

All five block ciphers have a block size of 128 bit and are used by VeraCrypt with a key length of 256 bit. This key size is well within the generally recommended key sizes for symmetric ciphers of 128 bit, 192 bit, and 256 bit (see e.g., BSI (26) and NIST (30)).

VeraCrypt tests the functionality of the ciphers with a simple test-vector test. On the Windows platform the tests are part of the `src/Common/Tests.c` file and called during program startup and during performance tests. On the Linux platform, they are part of the `src/Volume/EncryptionTest.cpp` and called during volume creation and performance tests.

Table 4: Encryption and hash algorithms with key, block, and output sizes.

Block Cipher	*Key Size (bit)*	*Block Size (bit)*
AES	256	128
Camellia	256	128
Kuznyechik	256	128
Serpent	256	128
Twofish	256	128

Hash Algorithm	*Output Size (bit)*
RIPEMD-160	160
SHA-256	256
SHA-512	512
Whirlpool	512
Streebog	512

4.1.2 Mode of Operation

VeraCrypt is using the XTS mode of operation. This mode is standardized as IEEE Std 1619-2018 (31) and is specifically recommended for the encryption of storage devices, e.g., by NIST (4). It is also mentioned by BSI in (26), Sect. 1.5. Each block (also called cluster) of the storage device is encrypted individually. XTS uses two independent keys, one for encrypting the block initialization vector (IV), and one for data encryption. The IV is an integer consecutively assigned to each device block. In case the block size of the storage device is not an integer multiple of the block size of the cipher, ciphertext stealing is applied in the XTS standard for padding the last cipher input block. VeraCrypt offers the user to choose the device block size when the

[28] The instances where source code was examined are explicitly mentioned.

device is set up. Typical block sizes range from 512 B to 16 kB on FAT file systems, 4 kB to 64 kB on NTFS for Windows, and 1 kB to 64 kB on the extended file systems (ext) for Linux. The default for many modern file systems is 4 kB.

VeraCrypt does not make use of ciphertext stealing; the device block size is always a multiple of the cipher block size. Both the use of keys and the choice of the IV in VeraCrypt is compliant with the XTS standard.

VeraCrypt tests the functionality of the XTS mode implementation with test vectors for an instance using the AES cipher. These self-tests are part of the tests for the ciphers as mentioned above.

4.1.3 Cascading Ciphers

VeraCrypt offers several combinations for cascading ciphers (see Section 2.5.1). The cascading is implemented by simply encrypting the same data under different keys and ciphers using XTS mode several times such that the output of one encryption is the input of the next. Cascading ciphers may have an impact on some security properties of the individual ciphers (see (32) for details and (33) for an overview); however, these issues mostly do not apply in the context of VeraCrypt (no authenticated encryption, XTS as mode of operation) and the resulting security is not lower than that of using a single encryption.

4.2 Cryptographic Hash Functions

VeraCrypt is using cryptographic hash functions within the random number generator and for key derivation. VeraCrypt supports the five algorithms RIPEMD-160, SHA-256, SHA-512, Streebog, and Whirlpool. The hash functions with their output size are listed in Table 4. RIPEMD-160 has the smallest output length with 160 bit. SHA-256 has an output length of 256 bit and SHA-512 of 512 bit. VeraCrypt is using the 512-bit versions of Streebog and Whirlpool.

RIPEMD-160

RIPEMD-160 was published in 1996 (8) as strengthened version of its predecessor RIPEMD from 1992. RIPEMD-160 does not fulfill current general recommendations for cryptographic hash functions. For example, BSI is recommending to use hash functions with an output length of at least 240 bit to achieve at least 120-bit security (26). Furthermore, RIPEMD-160 is on the “Monitored Ciphers List”[29] of the Japanese Cryptography Research and Evaluation Committees (CRYPTREC), meaning that it is considered unsafe and only permitted for maintaining compatibility with legacy systems. We recommend to remove RIPEMD-160 from the set of supported hash functions.

SHA-256 and SHA-512

SHA-256 and SHA-512 of the SHA-2 family have been standardized, e.g., by NIST as FIPS PUB 180-4 (9), and are recommended, e.g., by the BSI (26). Both functions provide sufficient security and output length as recommended by the BSI. SHA-2 has received extensive analysis and scrutiny. Some partial attacks (using a lower number of rounds) have been reported; currently, no feasible attack on SHA-2 is known.

Streebog

Streebog is standardized as GOST R 34.11-2012 (34). The output length of 512 bit chosen by VeraCrypt well exceeds the general recommendations. However, similar to the block cipher Kuznyechik, there is criticism that the involved S-boxes were generated by a secret algorithm (28) (29).

Whirlpool

Whirlpool is standardized as ISO/IEC 10118-3 (35). The output length of 512 bit well exceeds the general recommendations. It is based on the AES block cipher. It is recommended by the NESSIE project (New European Schemes for Signatures, Integrity and Encryption) (36).

[29] https://www.cryptrec.go.jp/en/method.html

4.3 Key Derivation Function

The documentation of VeraCrypt recommends to use strong passwords that are not composed from words that can be found in a dictionary. Passwords should not contain any names, dates of birth, or other information that can be obtained by social engineering. The documentation and the user interface to set up a new volume show recommendations to use at least 20 characters in a random combination of upper and lower case letters, numbers, and special characters as password. VeraCrypt does not give or enforce a specific set of characters to be used for a password. If a password shorter than 20 characters is used, a warning is issued to the user, which can be ignored. There is a variation to this when a Personal Iterations Multiplier (PIM) is used; see Section 4.3.1 for details.

VeraCrypt uses the key derivation function PBKDF2 in order to derive a symmetric key for encrypting parts of the VeraCrypt volume header. PBKDF2 is specified in PKCS #5 v2.0 and RFC 2898, and recommended by NIST Special Publication 800-132 (37). PBKDF2 is based on iterative chain-hashing where the output of one hash function iteration is the input to the next iteration. The input to the first iteration is the user password with an additional salt. The security of PBKDF2 is based on increasing the computational effort required to derive a single key thereby making brute-force attacks inefficient. The computation effort can be controlled by the number of iterations of PBKDF2 which can be chosen as security parameter.

VeraCrypt is using a salt derived from its RNG (see Section 4.4). It uses the hash functions described above in an HMAC construction. For most hash functions, by default VeraCrypt is using 500,000 iterations with the exception of the weak RIPEMD-160 for which it uses 655,331 iterations. Alternatively, the user can specify a PIM value to set up a desired number of iterations (see Section 4.3.1). Furthermore, keyfiles can be used in addition to the password (see Section 4.3.2). VeraCrypt checks the functionality of the HMAC constructions for all hash-functions with test vectors as part of the block cipher self-tests mentioned in Section 4.1.1. NIST Special Publication 800-132 (37) recommends:

- to use a minimum iteration count of 1000 and for “especially critical keys, or for very powerful systems or systems where user-perceived performance is not critical, an iteration count of 10,000,000 may be appropriate” (37 p. 6) (the default iteration count of VeraCrypt is on the mid-low end of this range),
- to use a hash function in the HMAC construction for PBKDF2 (which is followed in VeraCrypt),
- to use a salt of at least 128 bit derived from a true random number generator (the salt length of 512 bit in VeraCrypt more than suffices for this requirement), and
- to use both a hash function and a true random number generator that have been approved by NIST (which is given for some of the hash functions in VeraCrypt; see Section 4.4 for a discussion of the VeraCrypt RNG).

PBKDF2 is listed as “Agreed Key Derivation Function” by the SOG-IS Crypto Working Group (38), Sect. 3.7. If an HMAC construction is used in PBKDF2 (as applied by VeraCrypt), SOG-IS explicitly points out that if the HMAC key length (i.e., the password) exceeds the hash-function block size, the effective entropy of the resulting output key may be reduced compared to the entropy of the password input.

The security of PBKDF2 is based solely on computational cost, which means that it can be attacked by a powerful attacker with massively parallel special purpose hardware. There are key derivation functions that are designed such that in addition to computing power, they also require a large amount of memory. This makes an attack also with dedicated hardware more expensive. Examples are bcrypt (39) and scrypt (40) (published by IETF as RFC 7914) as well as Argon2 (41), the winner of the *Password Hashing Competition*[30]. We recommend to transition from PBKDF2 to a state-of-the-art key derivation function like Argon2 as recommended by BSI in TR-02102-1 (21).

[30] https://password-hashing.net/

4.3.1 Personal Iterations Multiplier

The PIM functionality was introduced in VeraCrypt version 1.12. In the previous versions, the security relied solely on the password strength since the number of iterations was fixed. PIM provides a two-dimensional security space that provides more flexibility to get a desired security level and to control the performance of the mount (and boot) operation. It is not mandatory for the user to specify the number of iterations. If no PIM value is specified, VeraCrypt will use the default value as discussed above.

If the user wants to choose a value, there are two ways to specify the number of iterations: The user can specify a PIM value in the command line or in the password dialog. When the PIM value is chosen, the calculation of the number of iterations depends on what kind of encryption is used by system. For the system encryption, which does not offer SHA-512 or Whirlpool, the number of iterations is 2048 times the PIM value. For volume encryption it is 1000 times the PIM value plus 15,000.

The PIM value is an additional secret value that must be entered each time together with the password; it is not stored along with the volume header. Hence, an incorrect PIM value will let the mount/boot operation fail since the correct decryption key for the volume header cannot be computed.

Choosing a small or a high PIM value affects security. High PIM values give better security but at the same time require higher number of iterations which results in a slower overall mounting (or booting) time. Small PIM values make mounting (or booting) faster but do not give as much security, which is a particular risk if a weak password was chosen. If the password is too short (less than 20 characters), VeraCrypt forces the PIM value to be greater than 98 for system encryption and greater than 485 otherwise. This setting is enforced by the volume creation and password change dialogs. More specifically, the dialogs first check whether the password length is less than 20 characters, and then issue an error if the PIM is too small. If the password length is sufficient, but the PIM too small, only a warning is issued. Using a PIM value of zero disables the PIM functionality and returns to the default values as described before.

4.3.2 Keyfiles

VeraCrypt offers the option to use one or more “keyfiles” in combination with a password. There are no restriction on the type of file that can be used, however only the first 1 MB of data within the file is used. The documentation of VeraCrypt recommends the use of files with compressed contents. Alternatively, VeraCrypt also includes a generator for keyfiles which utilizes the random number generator (RNG) to generate a file from random content (see Section 4.4). The use of a keyfile is optional; it can be used to supplement the password to increase protection against brute force attacks. Furthermore, the program allows for the storage of keyfiles on security tokens adhering to the PKCS #11 (2.0 or later) standard (42), such as smart cards.

The keyfiles are processed sequentially with a compression function as shown in Algorithm 1. A "keyfile pool" is defined, which is an array of bytes with 64 elements (the maximum length of a password). This pool is initialized with the password, padded by zero bytes. Then for each keyfile, the `KeyfileApply` function is called. This function calculates the CRC32 digest of the file byte by byte. After each byte, the intermediate value of the CRC32 digest is added byte by byte to the pool with integer addition modulo 256. This is done akin to a ring-buffer, i.e., a cursor pointing to the next byte of the pool is managed, which wraps to the beginning of the pool when the end is reached. The whole process is commutative when several keyfiles are used; the order of the keyfiles does not influence the result. After all keyfiles have been processed, the resulting 64 byte pool is treated as the encryption password. This process is described in the documentation[31], and matches the behavior of the source code for both the Windows and Unix platforms.

```
input  : keyfile[n] byte array with size of file $n$ in bytes, keypool[l] byte array with size of pool $l$
output : The updated keypool[l]

crc ← crc32_init();
$p \leftarrow 0$;
foreach $0 \leq i < \min(n, 1048576)$ do
    crc ← crc32_update(crc, keyfile[i]);
    keypool[p] ← keypool[p] + (crc ≫ 24) (mod $2^8$);
    $p \leftarrow p + 1 \pmod{l}$;
    keypool[p] ← keypool[p] + (crc ≫ 16) (mod $2^8$);
    $p \leftarrow p + 1 \pmod{l}$;
    keypool[p] ← keypool[p] + (crc ≫ 8) (mod $2^8$);
    $p \leftarrow p + 1 \pmod{l}$;
    keypool[p] ← keypool[p] + crc (mod $2^8$);
    $p \leftarrow p + 1 \pmod{l}$;
```

Algorithm 1: The `KeyfileApply` algorithm.

4.4 Random Number Generation

The documentation[32] of VeraCrypt specifies the functionality of the random number generator (RNG). The authors also specify two sources (16) (43) as the origin for the design. The VeraCrypt source code contains two implemented variants of the RNG: One is exclusively used on the Windows platform, while the other is used for Unix-based platforms. The implementation sources for the Windows platform are contained in the `src/Common/Random.c` files, while the Unix sources are contained in `src/Core/RandomNumberGenerator.cpp` (as well as the respective C header files). In the following, we analyze the source code of these two variants separately and match them to the documentation.

4.4.1 Documentation

At the time of writing, the documentation contained a textual description of the RNG. The RNG is described as the source for the master encryption key, secondary key for the XTS mode, salt, and keyfiles. The construction is specified as a pool of 320 bytes, which is filled with data from the following sources:

- User inputs, i.e., mouse movements and keyboard strokes,
- values from the RNG of the system, i.e., MS CryptoAPI on the Windows platform and the `/dev/random` and `/dev/urandom` files on Unix platforms,
- network interface statistics on the Windows platform, and
- "various Win32 handles" (i.e., API call results), time variables and counters.

[31] https://www.veracrypt.fr/en/Keyfiles.html
[32] https://www.veracrypt.fr/en/Random%20Number%20Generator.html

Each source is divided into individual bytes and added to the pool with integer addition modulo 256 at the position of the current “pool cursor”. This cursor is advanced for each added byte and wraps to the beginning of the pool when the end is reached.

For every 16 bytes added to the pool, a “mixing” function is to be applied to the contents of the pool. This mixing function is specified as follows:

1 Let R be the randomness pool with z as the size in bytes of R.

2 Let H be the hash function selected by the user, with l as the size of the output of the hash function H in bytes.

3 Let $q = \frac{z}{l} - 1$.

4 Divide R into l -byte blocks $(B_0, \dots, B_q)$.For $0 \leq i \leq q$ (i.e., for each block B) the following steps are performed:

$M = (H(B_0 || B_1 || \dots || B_q)$ (i.e., the randomness pool is hashed using the hash function H, which produces a digest M) and

$B_i = B_i \oplus M$.

5 $R = B_0 || B_1 || \dots || B_q$.

The output function of the RNG never outputs values from the pool directly, but prepares the output as follows:

1 Data obtained from the specified sources is added to the pool as described above.

2 The requested number of bytes is copied from the pool to the output buffer, starting from the position of the pool cursor. The cursor wraps to the beginning of the pool accordingly, when the end of the pool is reached.

3 Each bit in the pool is inverted.

4 Data obtained from the specified sources is added to the pool as described above.

5 The mixing function is applied to the pool.

6 The mixed pool is added to the output buffer with the “exclusive or” function, starting from the position of the pool cursor. The cursor wraps to the beginning of the pool accordingly, when the end of the pool is reached.

The output function is specified to reject requests when the number of requested bytes is larger than the size of the pool.

4.4.2 Windows Platform

As described in Section 2.5.2 and Section 4.4.1, the functionality of the RNG is based upon a pool in which entropy is gathered. The pool itself is an array of 320 B to which entropy is added. The pool is then diffused by a “mixing” function `RandMix`, which is triggered every time 16 B of entropy are gathered, as well as during the output function.

The entropy sources are grouped into three categories: *user inputs*, *fast sources*, and *slow sources*. These three groups are gathered in varying intervals. The RNG is initialized and set into an active state whenever the user performs an action which requires the use of the RNG, for example, during the creation of a volume or while changing the password. All entropy sources are interpreted as a string of bytes, which are added in a ring-buffer-like fashion on top of the pool. The state of the RNG retains an index to the “top” byte in the pool, to which the next byte of entropy is to be added and increments it whenever a byte is mixed. The index rolls over to the beginning of the pool when the end is reached. The operation used when adding new entropy byte-wise to the pool is the *integer addition* operation, which is also explicitly specified in the documentation. The current “top” byte of the pool and the new entropy byte are interpreted as 8-bit integers, added modulo 256, and the result is stored as a byte in the pool at the current index replacing the “top” byte. The entropy sources and the addition to the pool match the description of the documentation (apart from an unmentioned additional “slow” source), albeit that the documentation does not group the sources.

User inputs

Mouse and keyboard input events are gathered constantly while the RNG is in an active state. For both event classes, the CRC-32 function is used to compress the gathered data. In the case of the mouse events, a CRC-32 checksum of the event data structure provided by the Windows API is computed. If the checksum of the mouse input event matches the checksum of either of the last two events, it is discarded. Otherwise, the checksum of the current timestamp, as well as the difference between the last and current timestamp is added to the checksum of the mouse event (unsigned 32-bit integer addition), and the resulting bytes are added to the RNG pool as described above. Furthermore, the state of the RNG contains a counter of the number of mouse input events, which is used as an estimator for the amount of available entropy. Keyboard input events are handled in a similar fashion. In both cases, four bytes are added to the pool for each input event. These four bytes are compressed by the CRC-32 function, but are in both cases based on the input data (i.e., key stroke or mouse pointer position), as well as timing information about the event.

Fast entropy sources

These sources are polled periodically by a timed function every 500 ms while the RNG is in an active state. This `FastPoll` function queries various Windows APIs for gathering system information. However, in most cases this function does not add the actual gathered information into the pool, but only the “handles” (i.e., data pointers to the information returned by the API calls). Any properties of these handles, such as the value range and distribution, cannot be judged properly due to the closed source of the operating system. The `FastPoll` function adds handles of the following API objects:

- active window,
- window capturing the mouse,
- window owning the clipboard,
- first clipboard viewer,
- current process,
- current thread,
- desktop window,
- window with keyboard focus,
- window which has the clipboard open,
- handle for the process heap, and
- current window station for the process.

Apart from these handles, the `FastPoll` function adds the actual information of queries for:

- process ID,
- thread ID,
- system uptime in milliseconds,
- current position of the input caret,
- current position of the mouse cursor,
- memory status of the system, e.g., the amount of available and used memory,

- creation and exit timestamp of the current thread and process[33],
- time spent in user and kernel mode for the current process and thread,
- information about the working set of the process, i.e., information about the memory pages of the process, and
- value of the performance counter, i.e., the value of a fine grained timer.

The process and thread IDs do not change between calls, however, the `FastPoll` function is called from different threads. Any timers or timestamps are monotonic counters and likely include some variation due to jitter introduced by the underlying clocks and process scheduling of the operating system.

Finally, the `FastPoll` function also queries the system RNG via the `CryptoGenRandom` function. The backing implementation of this system function varies depending on the Windows version: Starting with Windows XP until and including Windows Vista it is based on a PRNG matching the FIPS 186-2 (44) standard; starting with Windows Vista SP1, all further version of Windows use an AES counter-mode based PRNG based on the NIST SP 800-90 (45). The sources of entropy for the seed of the system RNG are not known but specified to be based on user input timings and other jitter from hardware components[34]. The `FastPoll` function queries 320 B (i.e., the size of the pool) from the system RNG and adds it to the pool. After all sources are queried, the `FastPoll` function additionally triggers the `RandMix` function to diffuse the pool.

Slow entropy sources

These sources are only polled on demand when output is requested from the RNG. The `SlowPoll` function queries various network statistics as well as I/O statistics about the system hard drives. The I/O statistics are *not* mentioned as a source of entropy in the documentation. When the function fails to poll these statistics, no information is added to the pool and no error feedback is given. Finally, the `SlowPoll` function also queries the system RNG in the same manner as the `FastPoll` function. However, in this case an error is captured and handled. If the system RNG is successfully polled, the pool is diffused with the `RandMix` function.

Pool mixing and RNG output functions

The pool mixing function `PoolMix` is shown in Algorithm 2. It utilizes the hash function chosen by the user. The output size m of the hash function must divide the pool size n. The pool is mixed in $\frac{n}{m}$ rounds, where each of the m sized blocks of the pool is manipulated. In each round, the entire pool is hashed and the resulting digest is added to the current block of the hash. In this case, bitwise addition is used, i.e., bitwise "exclusive or" (XOR). The implementation of this function matches the specification described in the documentation.

The output function of the RNG is shown in Algorithm 3. It first triggers an optional polling of the slow sources (this option is always used when the RNG is queried for the first time), after which the fast sources are polled. Note, that each polling also triggers the `PoolMix` function. Afterwards, contents of the pool are copied to the output buffer using a rolling index akin to a ring-buffer. To avoid leaking the contents of the pool, the entire contents of the pool are then bitwise inverted and the fast sources are polled (again also triggering the `PoolMix` function). Afterwards, the modified pool contents are added on top of the output buffer using the bitwise XOR operation. The output function largely matches the specification as described

[33] According to the Windows API, the exit timestamp is undefined for running processes and threads.
[34] See the remarks of the API documentation: https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptgenrandom

in Section 4.4.1, except that larger requests are not automatically rejected, but addressed by automatically repeating the output function.

```
input  : Entropy pool pool[$n$] with $n$ bytes of data
input  : Functions hash_{init,update,final} for hash with $m$ bytes output and $m|n$
output : The diffused pool entropy data array

for $i \leftarrow 0, m, 2m, \ldots, n-m$ do
    ctx $\leftarrow$ hash_init();
    hash_update(ctx, pool);
    digest $\leftarrow$ hash_final(ctx);
    for $u \leftarrow 0, 1, \ldots, m-1$ do
        pool[$i$+$u$] $\leftarrow$ pool[$i$+$u$] $\oplus$ digest[$u$];
```

Algorithm 2: The `RandMix` *function for diffusing the entropy pool of the VeraCrypt RNG (Windows version).*

```
input  : Desired length len, entropy pool pool[$n$], option ForceSlowPoll, current pool index poolIdx
output : Buffer with random bytes output[len]

if ForceSlowPoll? then
    SlowPoll();
FastPoll();
$i \leftarrow 0$;
while len $> 0$ do
    if len $> n$ then
        looplen $\leftarrow n$;
    else
        looplen $\leftarrow$ len;
    len $\leftarrow$ len $-$ looplen;
    /* Copy directly from pool                                   */
    for $u \leftarrow i, i+1, \ldots, i+$ looplen $- 1$ do
        output[$u$] $\leftarrow$ pool[poolIdx];
        poolIdx $\leftarrow$ poolIdx $+ 1 \pmod n$;
    /* Bitwise invert entire pool                                */
    for $u \leftarrow 0, 1, \ldots, n-1$ do
        pool[$u$] $\leftarrow \neg$pool[$u$];
    FastPoll();
    for $u \leftarrow i, i+1, \ldots, i+$ looplen $- 1$ do
        output[$u$] $\leftarrow$ output[$u$] $\oplus$ pool[poolIdx];
        poolIdx $\leftarrow$ poolIdx $+ 1 \pmod n$;
    $i \leftarrow i +$ looplen;
```

Algorithm 3: The output function of the VeraCrypt RNG (Windows version).

4.4.3 Unix Platform

The construction of the RNG for the Unix platform is very similar to the construction for the Windows platform described in Section 4.4.2. An entropy pool of the same size is used to which data is added, subsequently diffused with a “mixing” function, and finally processed and returned to the caller in a very similar manner by an output function. However, there are differences in the entropy sources as well as slight differences in the mixing function.

Due to the substantial difference and availability of the system APIs, the Unix based RNG of VeraCrypt does not poll any sources for random entropy beyond the system RNGs. Hence, there is no distinction between "slow" and "fast" entropy sources. The polling function for the system RNG opens the `/dev/urandom` device file and reads bytes in the amount of the size of the pool. The random bytes are then added in the same manner as in the Windows version, i.e., with the byte-wise addition function in a ring-buffer like manner, triggering the mixing function after every 16 B. After polling the `urandom` file, the RNG optionally also polls the `/dev/random` file for the same amount of data. Depending on the underlying operating system, this file offers the same functionality as the `urandom` file. However, in case of Linux this file is backed by a different RNG, which only outputs a very limited amount of bytes and blocks reads when not enough entropy is available. The VeraCrypt RNG polls this file in a "non-blocking" manner, so if the system does not offer any entropy *the read fails silently*[35]. In contrast to the Windows version, other sources of entropy are not polled by the Unix RNG code. However, the RNG functions offer an API to add data to the pool, which is used in other places of the code base. For example, mouse movements are captured and fed to the RNG pool in a manner very similar to the Windows RNG by the code of the graphical interface for the volume creation. Similar to the Windows variant, the RNG is initialized and in an active state, whenever the user performs an action which requires the RNG. For example, the RNG is active and mouse movements are captured while the volume creation dialog is displayed.

The slight differences in the mixing function are shown in Algorithm 4. Firstly, the initialization function of the hash function is not called. The finalization function of the hash implementations usually just pads the input and produces the final digest. Thus the omission of the initialization function (which is only called once during startup), essentially means that the mixing function does not repeatedly hash the buffer. Instead this will lead to a *continuous* hashing of the modified buffer. The state of the hash function is then part of the state of the RNG code. This is likely to be harmless in this context, but nonetheless an oversight. The second minor difference in the mixing function is the use of integer addition instead of bitwise XOR addition.

```
input  : Entropy pool pool[n] with $n$ bytes of data
input  : Functions hash_{init,update,final} for hash with $m$ bytes output and $m|n$ and
         context for hash function ctx
output : The diffused pool entropy data array

for $i \leftarrow 0, m, 2m, \ldots, n-m$ do
    // The hash_init() is not called
    hash_update(ctx, pool);
    digest $\leftarrow$ hash_final(ctx);
    for $u \leftarrow 0, 1, \ldots, m-1$ do
        // Integer addition instead of XOR
        pool[i+u] $\leftarrow$ pool[i+u] $+$ digest[u];
```

Algorithm 4: The `RandMix` function for diffusing the entropy pool of the VeraCrypt RNG (Unix version). The differences to the Windows version are highlighted.

[35] When reading from `/dev/random`, VeraCrypt ignores the `EAGAIN` error code that is returned when no entropy is available.

The output function largely matches the Windows version. Instead of calling the function, the system RNG is called. The option causes the `/dev/random` file to be polled in the beginning, in addition to the `/dev/urandom` file. A notable difference is that the output buffer is not overwritten; instead the pool values are *added* to the pool with the integer addition function. An overview is given in Algorithm 5.

```
input  : Desired length len, entropy pool pool[$n$], option ForceSlowPoll, current pool index poolIdx
output : Buffer with random bytes output[len]

SystemRng(ForceSlowPoll);
RandMix();
$i \leftarrow 0$;
while len $> 0$ do
    if len $> n$ then
        looplen $\leftarrow n$;
    else
        looplen $\leftarrow$ len;
    len $\leftarrow$ len $-$ looplen;
    /* Copy directly from pool                                  */
    for $u \leftarrow i, i+1, \ldots, i +$ looplen $- 1$ do
        output[$u$] $\leftarrow$ output[$u$] $+$ pool[poolIdx];
        poolIdx $\leftarrow$ poolIdx $+ 1 \pmod n$;
    /* Bitwise invert entire pool                               */
    for $u \leftarrow 0, 1, \ldots, n-1$ do
        pool[$u$] $\leftarrow \neg$pool[$u$];
    SystemRng(False);
    RandMix();
    for $u \leftarrow i, i+1, \ldots, i +$ looplen $- 1$ do
        output[$u$] $\leftarrow$ output[$u$] $\oplus$ pool[poolIdx];
        poolIdx $\leftarrow$ poolIdx $+ 1 \pmod n$;
    $i \leftarrow i +$ looplen;
```

Algorithm 5: The output function of the VeraCrypt RNG (Unix version).The differences to the Windows version are highlighted.

4.4.4 Considerations for Entropy

In general, it is reasonable not to rely solely on the RNG provided by the operating system for security-sensitive requirements like the generation of secret keys. Both Windows and Unix variants of VeraCrypt query the RNG of the operation system for entropy. In the case of Linux, the system RNG has seen considerable scrutiny over the years (e.g., in (46)), and can be relied upon delivering random bytes for cryptographic applications. However, failing to read from `/dev/random` should not silently be ignored. Whether the queries to the various system APIs (i.e., the "slow" sources) by the Windows version of the RNG add any entropy to the pool cannot be evaluated due to the closed source of the operating system. It should be considered likely that the values are allocated by a deterministic algorithm. It is not specified in the Windows API, whether or not they contain any jitter, for example, introduced by random race conditions due to scheduling.

The only other source of entropy in common on both platforms are mouse pointer movements. Comments in the source code for both platforms express the authors estimate that each mouse event handled by the software is considered to be "worth" 1 bit of entropy[36], which is a debatable stance. The information content of each mouse movement can be expressed in terms of direction and speed, which is unlikely to change

[36] The source code comments cite the following as a reference: https://security.stackexchange.com/a/32848

drastically in a random manner during a single stroke. However, on both platforms a "progress bar" is shown in the user interface for key generation, which is in a full state when 2,560 mouse events have been captured. The code continues capturing mouse events even after the progress bar is filled.

4.4.5 Changes in VeraCrypt 1.24

In version 1.24 VeraCrypt added a further source of entropy by including code from the project *CPU Jitter Random Number Generator*[37] in both the Windows and Unix variants of the software. This "jitterentropy" source gathers entropy by measuring the execution time of the repeated invocation of a linear feedback register as well as the execution time of a set of memory-access operations. The rationale is that the high complexity of modern CPU architectures makes it hard to predict an exact execution time in the context of a multitasking operating system. The execution time of memory operations heavily depends on, e.g., the many caching layers of modern CPUs. Since software running in a multitasking environment shares this cache with many other tasks, the processing time of memory operations strongly varies with the memory usage of other tasks and hence is hard to predict. The execution time of non-memory-related code is similarly affected in a multitasking environment, e.g., due to the complexity of modern CPU pipelines.

The VeraCrypt authors added version 2.2 of the "jitterentropy" implementation to the source code, with very slight modifications that do not touch the functionality of the source. These modifications include changes to achieve compatibility with C++ source code, additional compilation options to ensure that the implementation is not compiled with enabled compiler optimizations (as required according to the "jitterentropy" documentation), as well as the addition of some platform-specific functionality for time measurement (as intended by the original implementation). One notable change, however, is the deactivation of the "FIPS mode" of the software, which disables a simple continuous self-test of the implementation[38].

The quality of "jitterentropy" as an entropy source is contested, see for example (47). The output produced by this process depends on the operating environment, i.e., the operating system, other running processes, and the execution hardware at hand. As such, the output is deterministic, albeit extremely hard to predict due to the immense complexity. In a controlled environment, e.g., during operating system boot time or in a real-time application, this entropy source should be considered unreliable. As VeraCrypt is not usually executed in such an environment, the addition of this entropy source to the RNG should not be problematic. The added benefit is nonetheless debatable, as this entropy source measures execution-time noise in a very similar manner to common operating system RNGs, e.g., the Linux RNG (46). Furthermore, the VeraCrypt authors did not document the reasons as to why they chose to deactivate the self-test functionality of the "jitterentropy" implementation.

4.5 Cryptographic Primitives

Being the foundation of information security, implementations of the cryptographic primitives requires particular scrutiny. The original public sources of the cryptographic primitives and changes by VeraCrypt to these sources are identified in Section 4.5.1. In Section 4.5.2, the implementations in VeraCrypt are further inspected for mistakes, oddities, and general issues. Section 4.5.3 describes known-answer tests to assess the correctness of the implementations in comparison to a third-party reference implementation. Finally, in Section 4.5.4, standard randomness tests are applied to output of the VeraCrypt random number generator.

[37] https://www.chronox.de/jent.html

[38] We reported this to the VeraCrypt developers, who already included a fix in their code repository: https://github.com/veracrypt/VeraCrypt/commit/425e4e7d365795b820fa145403b2be372894c48b#diff-a25aec91ade1d42e9b0aa1db03784011

4.5.1 Comparison with Public Sources

The cryptographic primitives in VeraCrypt have not been implemented by the TrueCrypt or VeraCrypt contributors themselves but originate from other public sources. In this section, the original public sources of the cryptographic primitives in the VeraCrypt repository are identified if possible and the sources in VeraCrypt are compared to these original sources.

AES

VeraCrypt includes several AES implementations, which cover 32- and 64-bit x86 CPUs with vector instructions, a pure C implementation, as well as a variant utilizing the AESNI extensions of Intel CPUs. All except the AESNI variant of the AES ciphers stem from an outdated version of the AES implementations by Brian Gladman, an updated version of which can be found on GitHub[39]. The source files remained largely untouched since the original import of the TrueCrypt sources. We were unable to find an original version of the sources with the same date of 2007. The equivalent files to the updated upstream source are, however, largely unchanged. The only changes concern the name of some symbols and types, as well as the addition of some compiler information that enables Linux stack protection mechanisms. None of these changes introduce a change in functionality. The variant utilizing the AESNI instructions comes from TrueCrypt and was not modified since then apart from non-functional changes similar to those already mentioned.

Serpent

The sources include two variants of the Serpent cipher. The original variant was already included in TrueCrypt and was taken from the *Crypto++* library[40]. The implementation included in VeraCrypt is mostly equivalent to an early revision of the original, except that the source code was rewritten to the C programming language. A version titled “fast”, was later introduced to VeraCrypt, originating from the *Botan* library[41]. The variant is heavily adapted to VeraCrypt, especially with a change to the C programming language. An SIMD variant was further taken and adapted from the *Botan* library.

Twofish

We identified three different libraries as the sources for the Twofish implementations included in VeraCrypt. The first is the *cppcrypto* library[42], not to be confused with the Crypto++ library mentioned above. The *cppcrypto* sources were adapted to utilize some approaches taken by the implementation of the *Botan* library. Finally, separate assembly implementations for 32- and 64-bit processors were integrated, taken from a third source[43].

Camellia

VeraCrypt includes two implementations of the Camellia cipher. The original source of the first variant stems from the authors of Camellia, the Nippon Telegraph and Telephone Corporation. An additional optimized assembly version was adapted from one of the Twofish cipher sources[44]. VeraCrypt then introduced changes that concern checks for present CPU features. These checks are used to select an implementation suitable for the current CPU, for example between a variant capable of processing 16 blocks at once, or a variant utilizing the AESNI processor extensions. A second smaller variant of Camellia is only used by the VeraCrypt bootloader. This implementation was taken from the *MbedTLS* library[45], and was slightly adapted to work outside of the framework of the original library.

[39] https://github.com/BrianGladman/aes, commit `d05d6f02c15ece6e`.
[40] https://github.com/weidai11/cryptopp
[41] https://github.com/randombit/botan
[42] http://cppcrypto.sourceforge.net/
[43] https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/twofish128ctr
[44] https://github.com/jkivilin/supercop-blockciphers/tree/beyond_master/crypto_stream/camellia128ctr
[45] https://github.com/ARMmbed/mbed-crypto

Magma (GOST)

We were unable to identify an original source of the Magma cipher. A comment in the source mentions Alex Kolotnikov as an author for the implementation, while the C header file mentions the TrueCrypt Developers Association, although the file was added after VeraCrypt was forked.

Kuznyechik

Two implementations of the Kuznyechik cipher are included in VeraCrypt. The first comes from the *cppcrypto* library and remains largely unchanged. The only changes introduced by VeraCrypt are again an adaption to the C programming language and some checks for CPU features. Depending on the CPU features present during runtime, an optimized implementation taken from the *lg15* library[46] is used, which contains optimizations for SIMD instructions.

SHA-2

The implementations for the SHA-2 family hash functions come from multiple sources. The first source is again the *cppcrypto* library, and is similarly adapted as the Kuznyechik and Twofish ciphers to check for present CPU features and chose an appropriate implementation of the hash permutation function. Depending on the present features, optimized variants utilizing various instruction sets are called. Variants utilizing various SIMD instruction sets come from Intel and are apparently sourced from the Linux kernel[47]. The sources are, however, adapted to use a different assembler syntax, but seem to be functionally equivalent. A further non-SIMD variant of the hash permutation function is sourced from "Project Nayuki"[48].

RIPEMD-160

The *cppcrypto* library is the original source of the RIPEMD-160 hash function implementation. The sources are adapted to the C programming language, and a data streaming interface is added to the code.

Whirlpool

The implementation of the Whirlpool hash function is a similar mixture of implementations as of the Serpent cipher. The original was taken from the *Crypto++* library, which in turn was adapted from an implementation of the original Whirlpool authors. However, with commit `00eb49` VeraCrypt integrated an implementation originating from the Botan library.

Streebog

We were unable to identify an original source of the Streebog hash function. A comment in the source code points to Alexey Degtyarev as the author, but no original source is linked. A public source code repository by presumably the same author exists[49] that includes an implementation of the hash function. This repository, however, does not include the implementation used by VeraCrypt. The Linux kernel, starting with version 5.0, includes an implementation of Streebog that is very similar to that used by VeraCrypt and also lists the same author.

4.5.2 Manual Source Code Examination

During the comparison of the sources with their public origins (if available), we also examined the source code manually for potential problems. Most of the implementations employ extensive code unrolling techniques, making a full line by line audit of the implementations for correctness infeasible. Thus, we

[46] https://github.com/aprelev/lg15
[47] We identified the following as a possible source:
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/crypto/sha256-avx-asm.S
[48] https://www.nayuki.io/page/fast-sha2-hashes-in-x86-assembly
[49] https://github.com/adegtyarev/streebog

mostly rely on the tests described in Section 4.5.3 to ascertain the correctness of the ciphers and hash functions. The manual examination, however, yielded several general findings.

The general quality of the code is problematic. This stems mostly from the fact that many cipher and hash-function implementations are a mix of several adapted open source libraries. The most serious consequence of this approach is that there is no uniform programming interface to any of the cryptographic primitives. This does not just extend to irregular naming of functions, but also to the order of parameters. For example, some schemes take the expanded key as the first argument and then the in- and output buffers, some take it as the last argument. This may, in the future, lead to possibly undetected programming errors, especially if changes to the code are not carefully tested with all cipher configurations. Furthermore, mixing several implementations may lead to programming errors, for example, due to oversights concerning the differing assumptions of the internal data types, as evident by an alignment error described in Section 4.5.3. In the described error, an assembly implementation assumes that the address of a buffer is aligned by 16 bytes, which is not documented in the API. The VeraCrypt authors alleviate the problem of the mixed APIs by providing an abstraction layer, present in the `Common/Crypto.c` (Windows) or `Volume/EncryptionAlgorithm.cpp` (UNIX) files, which is used by most of the rest of the code. Nonetheless, in the example of the mentioned alignment error, the VeraCrypt code does *not* ensure the alignment. We assume that this error is not getting triggered only because of coincidental alignment. We believe that more hidden and so far untriggered programming errors exist due to the large number of different and merged implementations.

Most of the ciphers and hash functions utilize implementations that are highly optimized for execution speed. These kinds of implementations are usually not protected against side channel attacks. In the context of the security model described in Section 3.3.7, most side channels such as power and electromagnetic emanation are not problematic, as attacks using these channels usually require very sensitive measurements, not available to the attacker in most scenarios. However, timing variance is a highly problematic side channel for the relevant applications. The speed of a read or write access to an encrypted volume is directly affected by the timing variance of the underlying cipher. These timings are usually easily measurable by an attacker, especially in the context of a server or cloud application, albeit with a lot of noise. Such a timing variance is usually caused by key- or data-dependent branches, or differing access times to lookup tables used by ciphers, caused by the sophisticated caching mechanisms of modern processors. Most of the cipher implementations utilize large lookup tables and must therefore be assumed to be susceptible to timing attacks in some application scenarios. The Serpent cipher implementation is the only exception to this, as its small substitution boxes are usually implemented directly.

For the most prominent example of the affected ciphers, AES, a timing attack is well known and the implementation should be directly affected by this issue. We were able to reproduce the timing attack published in (48). The attack considers a scenario in which an attacker is able to measure the timing of the encryption of a known plaintext block. The timing attack is a profiling attack and requires a training phase using known secret key before attacking the target key. In a table-based implementation the access pattern to a table T_i corresponds to $T_i[p \oplus k]$, where p is a plaintext byte and k is a key byte, for the first accesses. The hypothesis of the attack is that for a table-based implementation the access time to a table is similar for the same index.

Due to the caching mechanism of the CPU the access times can vary. The attack exploits statistical properties, i.e., the maximum or the minimum, of the access timing. By choosing the zero key for the training phase the attacker gets times for $T_i[p \oplus 0] = T_i[p]$. In the attack phase, she gathers timings for the original pattern $T_i[p \oplus k]$ and accumulates the timings for the known plaintext byte p. After gathering the mean value from a several million measurements per byte value, the timings of both phases are correlated by accumulating the measured times per plaintext byte value for a specific key byte value: $t_t[p_j] \cdot t_a[p_j \oplus k]$ (t_t stands for the training phase and t_a for the attack phase; for details see (48). Afterwards, the maximum value for a byte value represents the most likely value for a key byte. The result is that the attack usually does not recover the exact value of a key byte, however, a successful analysis results in a clustering with just a few key byte candidates (cf. Figure 8, Figure 9, and Figure 10) such that a guided brute-force attack can be applied to recover the full key.

We analyzed the AES-128 variants of the implementations provided by VeraCrypt running on x86_64 machines. We extracted the required source files from VeraCrypt:

1. `src/Crypto/Aescrypt.c` (pure C),
2. `src/Crypto/Aes_x64.asm` (64-bit assembler),
3. `src/Crypto/AesSmall.c` (pure C), and
4. `src/Crypto/Aes_hw_cpu.asm` (64-bit assembler using Intel AESNI).

We integrated them into the test setup and acquired measurements on an Intel(R) Xeon(R) Gold 6254 machine while running the attack setup on fixed CPU cores to reduce the noise in the measurements. We gathered $5 \cdot 10^6$ timings per byte value to raise the probability of a successful recovery. The implementations 1 and 2 lead to a clear clustering of groups of 8 key byte candidates (cf. Figure 8 and Figure 9). Variant 3 shows a rough clustering and yields a much larger number of key candidates (cf. Figure 10) and is not as susceptible as the other variants. This implementation uses a 32-bit integer pattern to access larger tables. Nevertheless, all three variants are vulnerable to timing attacks.

The exception to this is the implementation 4 utilizing the AESNI CPU extensions, which should, depending on the processor, execute with a timing independent of the key or plaintext. As expected, the analysis result in Figure 11 shows a random distribution of timings and is not exploitable with the applied attack approach.

In practice, VeraCrypt utilizes the XTS mode of operation the encryption of volumes. As described in Section 2.5.1, XTS cascades two encryption instances using separate secret keys (one to recover the IV, one to recover the data). Consequently, the application of a timing attack has to consider two stages to recover two keys which considerably raises the effort for a successful key recovery. Furthermore, the attacker does not know the output of the first stage which is required to verify a successful key recovery when having a rough timing classification. Nevertheless, using the software AES implementations of VeraCrypt is not recommended when precise timing measurements are feasible due to the general vulnerability presented before. As such, we discourage the use of any affected cipher in a scenario where timing attacks are relevant to the attack model, for example, in the context of server applications. In those scenarios, only the Serpent cipher should be used, or the AES cipher if the system in question supports the AESNI extension.

Concerning the cryptographic hash algorithms, the Whirlpool implementation is based on the AES block cipher. Consequently, similar tables are used to implement the function in VeraCrypt. This should, in principle, lead to a leak of cache timing information about the original data, which can be a password or a seed for a random number generator. The Streebog implementation includes large lookup tables. The attack scenario on a hash function in the context of VeraCrypt is, however, much more limited, as the hash functions are only in use during key derivation when opening a volume, or otherwise when the RNG is active. It is unlikely that an attacker would be able to observe the millions of necessary key derivations to gather enough timing information, in any reasonable usage scenario.

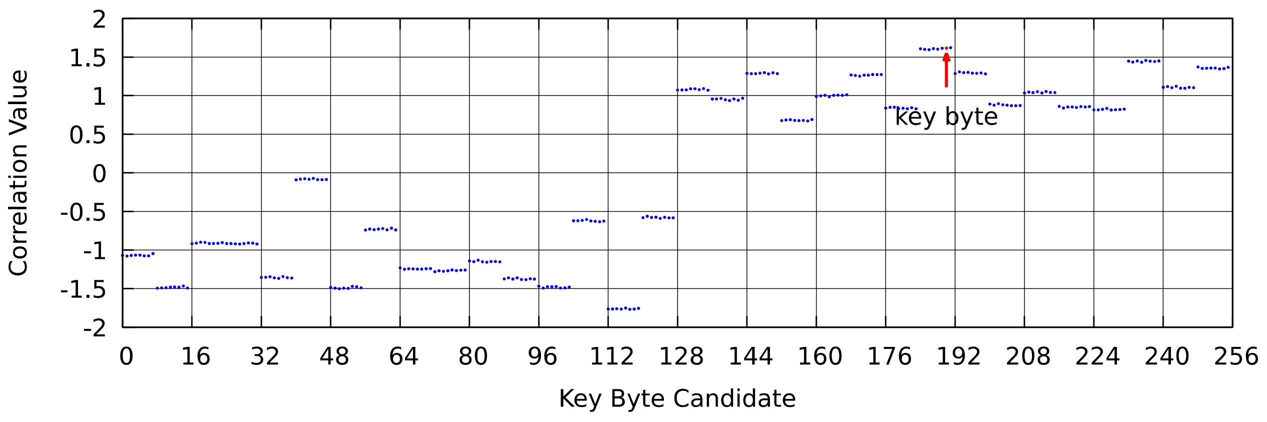


Figure 8: Exemplary timing clustering on Intel(R) Xeon(R) Gold 6254 at key byte position 0 for the pure C AES-128 implementation found in `src/Crypto/Aescrypt.c`.

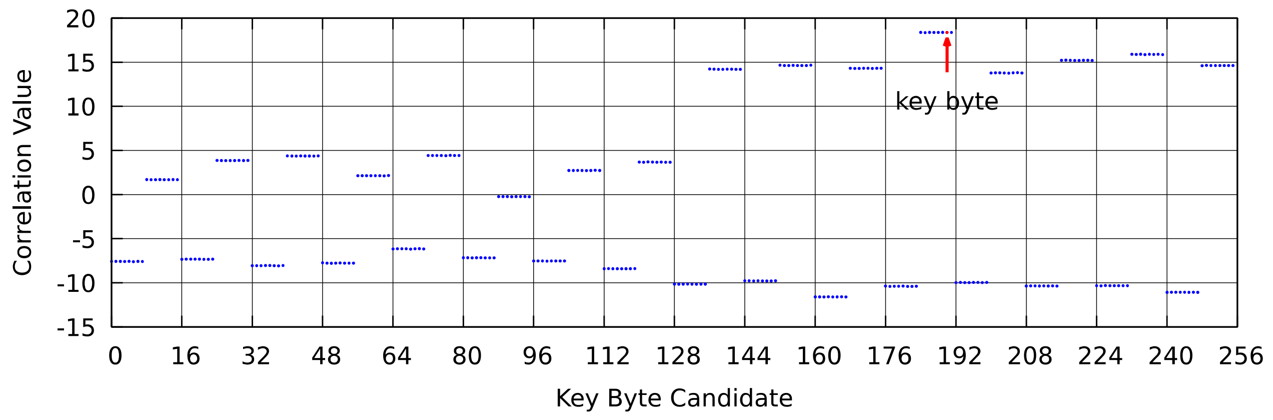


Figure 9: Exemplary timing clustering on Intel(R) Xeon(R) Gold 6254 at key byte position 0 for the 64-bit assembler AES-128 implementation found in `src/Crypto/Aes_x64.asm`.

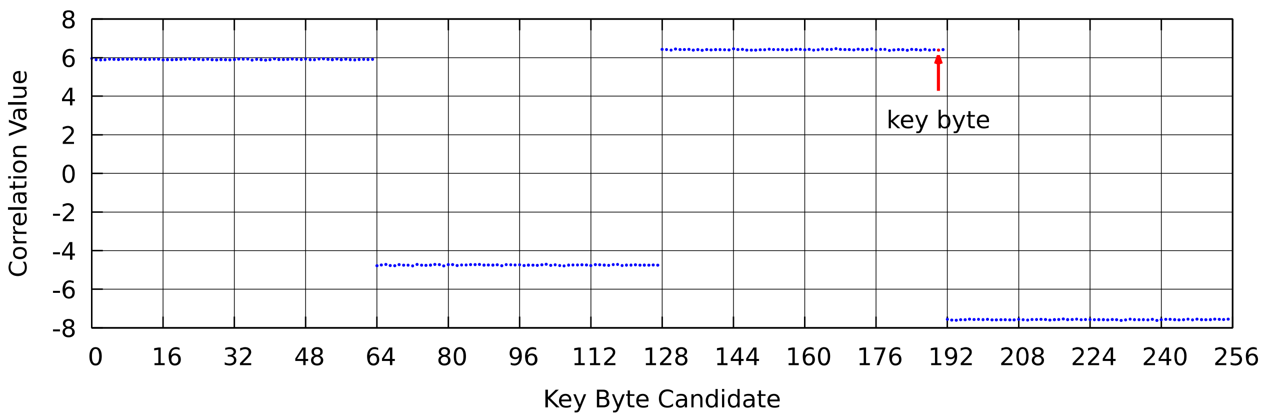


Figure 10: Exemplary timing clustering on Intel(R) Xeon(R) Gold 6254 at key byte position 0 for the pure C AES-128 implementation found in `src/Crypto/AesSmall.c`.

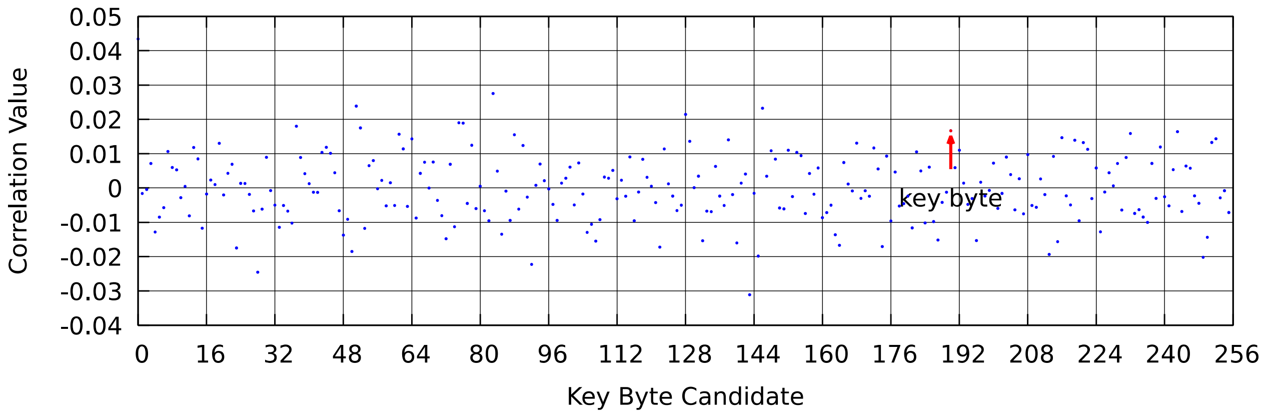


Figure 11: Exemplary timing clustering on Intel(R) Xeon(R) Gold 6254 at key byte position 0 for the Intel AESNI-based AES-128 implementation found in `src/Crypto/Aes_hw_cpu.asm`*.*

Apart from the encryption timing issues, we found a peculiar construction present in the Magma cipher implementation. The Magma cipher (now deprecated in VeraCrypt) is the only cipher with a 64-bit block size, as opposed to the 128-bit blocks of all other ciphers. The implementation of the Magma cipher, however, was extended to support 128-bit sized blocks. This extension uses cipher block chaining to create a 128-bit block cipher. Let $ENC_k\colon \{0,1\}^{64} \mapsto \{0,1\}^{64}$ be the Magma block cipher encryption function. The Magma implementation essentially defines an extended block cipher ENC_k^* as follows: Let $x \leftarrow x_1||x_2$ with $x_i \in \{0,1\}^{64}$ be the 128-bit input block. Then ENC_k^* is defined as:

$$low \leftarrow ENC_k(x_1)$$

$$high \leftarrow ENC_k(low \oplus x_2)$$

$$ENC_k^* \leftarrow low||high$$

The origin or motivation of this construction is not documented in the source code or commits in the source-code repository. It stands to reason that the authors attempted to match the block size of the Magma cipher to the block size of the other ciphers to avoid extensive changes to the implementation of the XTS encryption mode, which is not able to handle block sizes other than 128 bit.

4.5.3 Known-Answer Tests

To ascertain the correctness of the ciphers and hash functions, we implemented an automated known-answer test (KAT). The test platform consists of a *GNU Make* script that performs the following steps: It unpacks the source package of VeraCrypt and for each cipher, compiles and runs a program that generates the test vectors for the KAT, and finally compiles and runs a small isolated test program that uses the generated vectors to check the cipher. The generator randomly produces 100 KATs, each of which are iterated up to 100 times, i.e., each KAT consists of a plain- and ciphertext pair, with the ciphertext being the result of an repeated application of the block cipher.

For all but the Kuznyechik cipher, we are using the *libgcrypt* library[50] for the KAT-vector generator programs. This library is available on most UNIX platforms. The Kuznyechik cipher is not supported by the *libgcrypt* library; instead, we used a reference implementation by Markku-Juhani O. Saarinen[51] as it does not share a history with the VeraCrypt implementation. The test programs use the cipher and hash-function sources of VeraCrypt. As VeraCrypt includes most ciphers in multiple variants, e.g., in variants optimized to certain CPU features, we perform these test on all variants used by VeraCrypt. First and foremost, we compiled and ran the code for both 32- and 64-bit processors. Furthermore, the test forces, to the extent possible without modifying the code, the execution of all variants, e.g., variants with and without SIMD

[50] https://www.gnupg.org/software/libgcrypt/index.html
[51] https://github.com/mjosaarinen/kuznechik

instructions. If a cipher supports the encryption and decryption of multiple blocks in parallel, the KAT is multiplied and performed in parallel.

The compilation of the ciphers and hash functions and their tests are performed with all warnings and diagnostic messages of the compiler activated to check for potential programming errors. No cipher or hash function implementations raises any relevant warning. The AES C implementation produces a warning concerning the "fall-through" of a switch-case statement that is usually a sign of a programming error, which in this case is, however, an intentional and correct use. Apart from this, the Streebog hash-function implementation defines a number of constant arrays, which are never used.

We also added the possibility to use the "sanitization" frameworks available in modern compilers. These frameworks, of which we use the address- and undefined behavior sanitizer, will automatically instrument the compiled code with runtime checks. The instrumented code will detect various errors during the execution of the test. Specifically relevant for these tests are the detection of stack- or buffer overflows, as well as the use of C language constructs with undefined behavior. Only one cipher exhibits an implementation problem: The 32-bit implementation of the Streebog hash function crashes due to an unaligned memory access. The `STREEBOG_add` function, which feeds the input into the state, assumes that the address of the input buffer has an alignment of 16 bytes. If this is not the case, the implementation will crash as the compiled code will use load instructions which require an aligned address. This requirement is not documented, and the rest of the VeraCrypt code using this hash function does *not* ensure that the input is aligned. It is possible, that triggering this bug is compiler dependent, and that some compilers may automatically issue a variant of a load instruction that allows unaligned addresses[52].

The Magma (GOST) cipher uses a substitution box, for which multiple variants have been defined over time since the first standardization in GOST 28147-89. For the Magma variant included in VeraCrypt, we identified the S-box to be the variant defined in the most recent version, i.e., the GOST R 34.12-2015 specification document (27). We were able to use the *libgcrypt* library to implement a test vector generator for both the standard version with 64-bit block size as well as for the construction used in VeraCrypt to handle 128-bit blocks. However, we had to specify the OID of the substitution box to be used. The OID is defined in RFC 7836 (49) as "1.2.643.7.1.2.5.1.1", or "id-tc26-gost-28147-param-Z".

4.5.4 Randomness-Tests

To assess the quality of the construction of the VeraCrypt random number generators, we used the statistical test suite described in SP 800-22 Rev. 1a (50). The statistical tests are designed to reject the null hypothesis that a sequence of bits is sourced from a perfect random number generator. In general, these tests are only suitable to detect significant implementation problems in the random number generator construction. All of the tests determine a different statistic about a sequence of bits and examine whether the determined statistic deviates from the value expected of a random sequence. Note that a rejection is an *indication* that a sequence is not random, as even a random sequence can be expected to fail statistical tests, especially since the tests work on finite sequences of bits. To counteract this, all tests are performed multiple times to assess whether a tests fails an expected number of times. All in all, 15 tests are performed, some of which in different variants. The test procedures are specified in (50) and not re-iterated here. For the actual implementation of the test, we used the software supplied in tandem with (50), however, to fix programming errors we applied a few patches that are shown in the Appendix, Section 9.4.

The tests take as input a stream from an RNG, which we generated and stored in individual files for examination. To generate these files from the RNG in VeraCrypt, we used the dialogs for generating keyfiles in the Windows and Linux versions of VeraCrypt. We confirmed that the keyfile generation uses the same RNG as is used during volume creation. The dialogs also gather mouse movements as source for entropy. We only started generation when the dialog indicated the collection of enough mouse movements. As the

[52] We reported this to the VeraCrypt developers, who already included a fix in their code repository: https://github.com/veracrypt/VeraCrypt/commit/7d1724e93b17c5095bb01c9f053600b3a85c3cdc

tests have to be repeated many times, we used the keyfile generation dialogs to simply generate the required number of keyfiles, each of which has the required length for the individual tests. As reference we also generated the same amount of random data from the Linux RNG `/dev/urandom`, the quality of which was assessed in (46).

To pick the parameters of the tests, we consulted the test specification and set them as follows: Each examined sequence has a length of 2^{20} bit, the block sizes for the block frequency and linear complexity tests are 512 bit, the length of the templates for the template tests are 9 bit, the block size for the approximate entropy tests is 13 bit and that of the serial test 17 bit. All tests are performed on 2000 sequences, i.e., 2000 keyfiles were generated and examined.

The results of the assessment are shown in Figure 12, Figure 13, and Figure 14 for the VeraCrypt Linux, Windows and Linux `/dev/urandom` RNGs respectively. The graphs show the p-value distribution of the various repeated tests. All of the p -values exhibit a near uniform distribution, which indicates an RNG of sufficient quality for cryptographic applications. The only outlier is the “Approximate Entropy” test, which shows a distribution biased towards a failure. The statistical testing software rates this single test as a failure. However, including the Linux RNG all tested variants fail this test, which indicates that this is likely the result of the choice of test parameters, rather than a failed RNG.

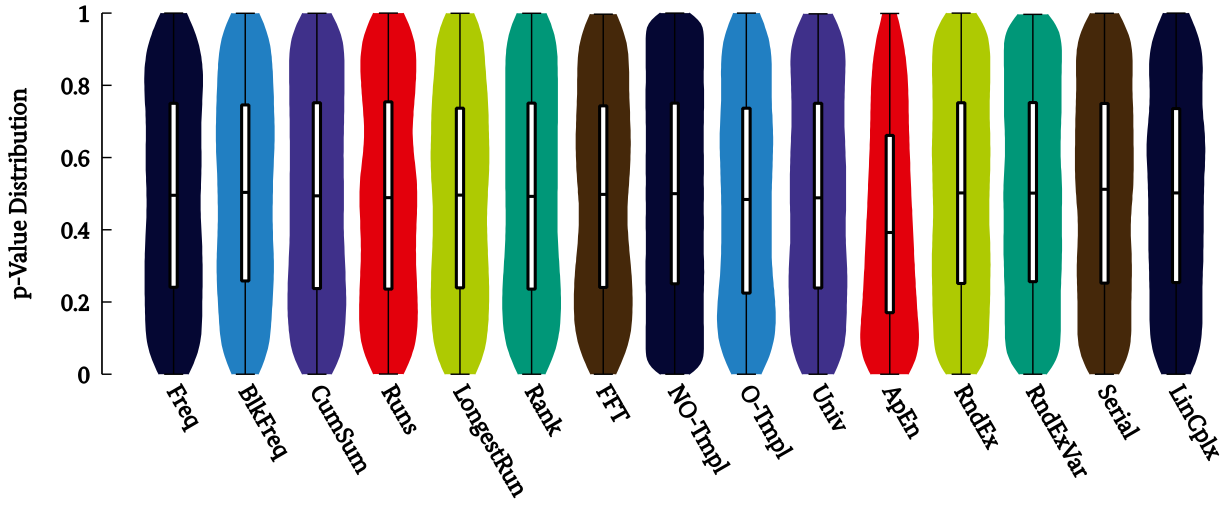


Figure 12: Statistical assessment results of the VeraCrypt Linux RNG.

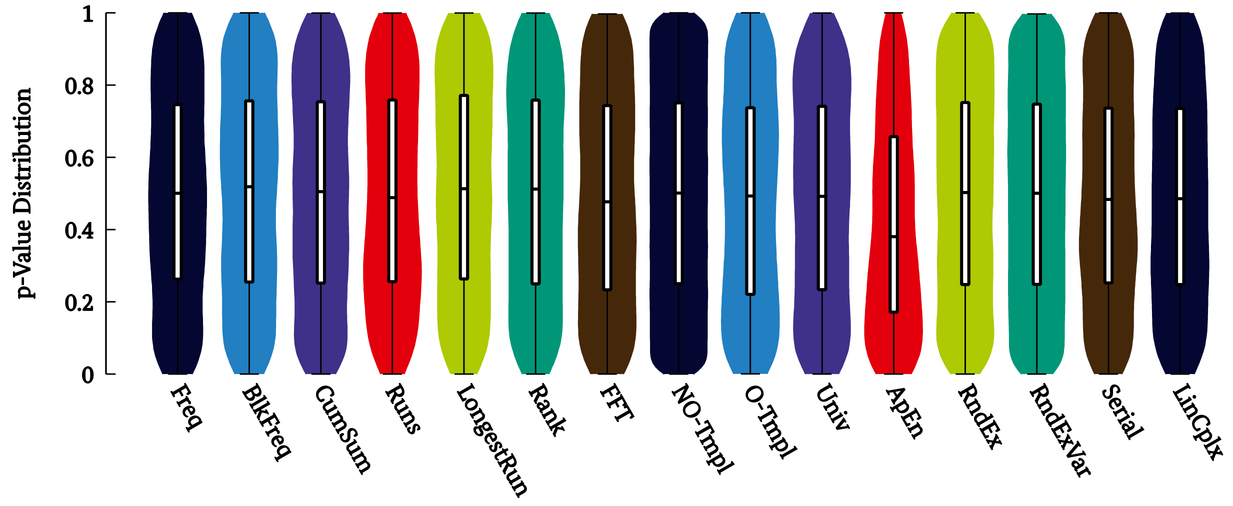


Figure 13: Statistical assessment results of the VeraCrypt Windows RNG.

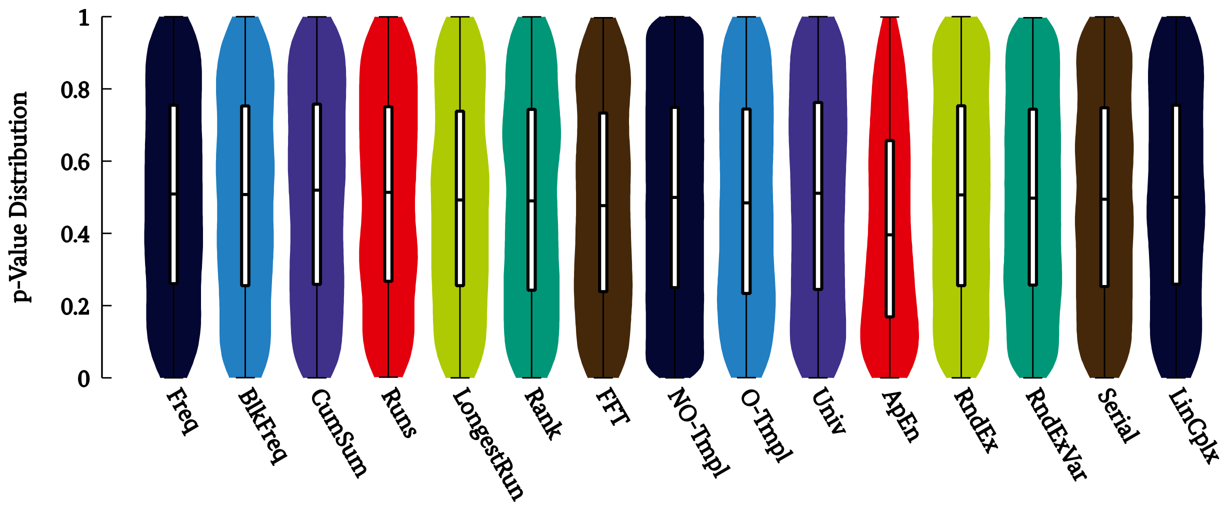


Figure 14: Statistical assessment results of the Linux /dev/urandom RNG.

4.6 Management of Secret Data

Throughout all functional blocks of the VeraCrypt source code, buffers holding secret data such as volume headers, passwords, or encryption keys, are routinely cleared after use. As mentioned in 2.6, TrueCrypt also makes use of this procedure to impede an attacker with access to memory dumps searching for secret data.

TrueCrypt, however, only uses the function `memset` for this purpose, i.e., to overwrite the memory with zeros. As the memory is regularly deallocated just after it was zeroed by calling `memset`, an optimizing compiler will usually remove this seemingly redundant function with the result that the secret data still remains in memory.

To avoid this behavior, the VeraCrypt authors replaced these uses of `memset` with a call to a short macro function `burn` to zero out a memory portion. In difference to the approach in TrueCrypt, `burn` uses a typecast of the memory buffer pointer to a volatile character pointer before the memory is zeroed. This pointer is used to iterate over each buffer character and to zero it. The assumption is that an operation including a pointer instance of a volatile pointer type must not be removed by optimization routines of the compiler.

In addition, for the Windows version the operating system function `RtlSecureZeroMemory` is used to clear the memory. In the Linux source code, `burn` is further encapsulated by a class `SecureBuffer` for the use in object-oriented code.

By checking a sample of source code files we confirmed that the macro is used throughout the code, among other places during volume formatting, password handling, key handling, handling of sensitive data derived from keys such as round keys, or by the random number generator, to clear variables or memory buffers containing secret data.

4.7 Memory Encryption

The latest VeraCrypt version 1.24 at the time of writing claims to implement a memory encryption mechanism, although VeraCrypt's security model explicitly states that memory is not encrypted. We inspected the quality of the memory encryption with a code review. In line with the security model, this newly introduced memory encryption is another layer of complicating potential attempts to extract key material. However, it cannot serve as a final solution but is merely an obfuscation measure, because the review showed that the key material for encrypting volume keys is also stored in memory. An attacker with access to the main memory can therefore extract all necessary data to decrypt encrypted volume keys. Scenarios in which this protection provides an actual benefit for security are rare if any exist.

In detail, the mechanism uses a large (1 MB or two memory pages), unprotected Key Derivation Area of pseudo random data to generate memory encryption keys. A ChaCha20-based pseudo-random number generator (i.e., using the ChaCha stream cipher[53] with 20 rounds) seeded with the VeraCrypt RNG is used for generating this Key Derivation Area.

For the actual memory encryption, the Key Derivation Area is hashed with the encryption ID of the memory area in question as seed to compute a memory encryption key using the non-cryptographically secure, non-standardized hash-function "Fast Positive Hash", short t1ha2[54]. The requested memory area is then encrypted under this key using ChaCha12 (i.e., the ChaCha stream cipher with 12 rounds) with a 128-bit key (doubled by concatenation to obtain the required 256-bit key). Furthermore, 64-bit masks are used to mask both the seed for the t1ha2 hash-operation and the ChaCha12 initialization vector.

This design of the memory encryption does not offer additional protection in case an attacker has access to the entire memory of the target system (e.g., with root privileges). Rather, the goal seems to be to offer some additional protection against cold-boot attacks, where an attacker may only gain access to partially corrupted memory; if the key derivation area is partially corrupted, it becomes harder for the attacker to reconstruct the key – the larger the key derivation area, the more likely a memory corruption. However, the scope of this protection measure is limited given the low error rates of modern cold-boot attacks (23). For performance reasons, the implementation is not using a cryptographically secure hash function and only a reduced number of ChaCha rounds for the encryption itself (12 as opposed to 20 as recommended in RFC 7539), which further reduces the strength of this countermeasure.

4.8 Recommendations

We recommend to remove the support for RIPEMD-160 as hash function and to transition to a modern key derivation function like Argon2. For the RNG, we recommend to harmonize the difference of the RNG construction between the two platforms and to initialize the hash-function state in the Unix `RandMix` function. The return value when reading from `/dev/random` in Unix should be checked and errors should be handled properly. Entropy from Windows system API calls could be increased by not only including handles to but also the content of the obtained data structures. A further option would be to replace the RNG mechanism by a construction that was subject to more scrutiny, such as the RNG of the Linux Kernel (46), the "Hash_DRGB" construction described in NIST Special Publication 800-90A Rev. 1 (45), or the "Fortuna" RNG construction (51).

Cryptographic functions must be implemented by experts and the implementations must be up to date, secure, and bug-free. Therefore, VeraCrypt should remove the implementations of cryptographic functions from its code base and instead use an open, vetted, and trustworthy cryptographic library. Using a single library will significantly reduce the effort required for the maintenance of the software and help to avoid programming errors. For each program release, this cryptographic library must be updated to the most recent version. Examples for cryptographic libraries are *libgcypt*[55], the *Botan* library[56], and *OpenSSL*[57]; all of these platforms are available for, e.g., Windows and Linux.

In scenarios where side-channel security is an issue (see Section 3.3.7), special precautions need to be taken. Many cryptographic libraries have versions of cryptographic primitives that are protected against, e.g., timing attacks; however, these versions are usually not used by default and may need to be specifically

[53] https://tools.ietf.org/html/rfc7539
[54] https://github.com/erthink/t1ha
[55] https://gnupg.org/software/libgcrypt/index.html
[56] https://botan.randombit.net/
[57] https://www.openssl.org/

activated. For example, the *Botan* library provides information about the side channel security of the different implementations that are provided[58].

[58] https://botan.randombit.net/handbook/side_channels.html

5 Application Security Review

In this section, we report results from our assessment of the general application security of VeraCrypt. As pointed out in our security model (see Chapter 3), the use of VeraCrypt might also affect the overall security of a system running VeraCrypt in case the application has further security vulnerabilities or weaknesses not directly related to its cryptographic mechanisms (cp. 3.2.2, 3.3.4 and 3.3.5).

We focused on such issues with the analyses detailed in the following. We started with various automated code analysis techniques as described in 5.1 and 5.2, continued with specific tests for the Windows driver kernel (see 5.3) and container files (see 5.4), and finally evaluated the security of integrated third-party libraries as described in 5.5.

5.1 Static Code Analysis

We performed static code analyses with the tools *CppCheck*, *TScanCode*, *Clang Static Analyzer*, *Clang-Tidy*, *VisualCodeGrepper*, and *cpplint*. The results of these tools are described in the following sections.

5.1.1 CppCheck

CppCheck[59] is a tool for static analysis of C/C++ code. Its focus lies on detecting undefined behavior like dead pointers or integer overflows and security analysis. It can also be used to check code quality. The analysis of VeraCrypt was performed using version 1.82 of *CppCheck*.

5.1.1.1 Usage

The tool *CppCheck* was called as shown in Listing 2. The first flags, `–std=c++11` and `–language=c++`, tell the tool which language and language standard is used. In this case, it is C++ in version 11. The flag `–enable-all` enables all checks available. Some include files were not automatically found by the tool, they had to be specified using the `-I` flag. The flag `-f` enables CppCheck to consider every preprocessor macro.

```
$ cppcheck --std=c++11 --language=c++ -I src/ -I src/Common/ --enable=all -f
src/ > cppcheckoutput.txt 2> cppcheckfindings.txt
```

Listing 2: The CppCheck command with the parameters used for the analysis.

5.1.1.2 Results

CppCheck assigns a category to each issue found. The different categories are:

- Style,
- Information,
- Warning,
- Error, and
- Performance.

The categories *Information* and *Performance* were not considered in this report, as issues assigned to these categories do not indicate security-related problems in the analyzed source code.

Figure 15 shows the number of results in the different categories. In total, 617 issues were found.

[59] https://packages.ubuntu.com/bionic/cppcheck

Style

The largest category with 583 found issues is the category *Style*. Findings in this category are related to code quality and not security issues. They are discussed in Section 6.

Warnings

The second largest category of findings is *Warning* with 21 issues. The issues were manually checked as to whether they are a security problem. In one case *CppCheck* warns, that there is a copy constructor in a `struct`, but no assignment operator (`/src/Common/Dlgcode.h:548`). In another case, there is an implementation of the assignment operator but no copy constructor (`/src/Common/Dlgcode.h:548`). This can lead to erroneous code, but in general not to security issues.

Another four issues are reported in `src/Platform/File.h:57` in which four member variables of a class are not initialized in the constructor. This may lead to problems using this class rather than a security issue.

In 14 cases, *CppCheck* marks the usage of the wrong format placeholders in the file `src/Common/zlib/trees.c` of the *zlib* library. This usually does not directly impact security.

The last warning occurred in `src/Crypto/SerpentFast.c:248`. The scanner cautions against a redundant assignment of a variable to itself. This case is a false positive, caused by *CppCheck* not correctly parsing the preprocessor directive used in this statement.

Performance

This category deals with code constructs that have an impact on performance. For example, *CppCheck* suggests to pass some parameters by reference or to initialize variables not in the constructor body but rather in the initialization list in order to reduce execution time. As this does not impact security this category is omitted.

Errors

CppCheck reported six errors in VeraCrypt. The first two were reported in `src/Common/Inflate.c` in lines 1030 and 1036. The error message states that the variable `e` is uninitialized. A manual analysis found that the variable is indeed not initialized. Nevertheless, the variable is passed by reference to another function, in which the variable is written before read. Although this is not best practice, it is not a security issue.

Another error is reported in *libzip*, in files `src/Common/libzip/zip_source_win32a.c:102` and `src/Common/libzip/zip_source_win32w.c:114`. Both are structurally very similar and the code snippets in which the error occurs are the same. In both cases, *CppCheck* reports a memory leak of a variable `temp`. A manual analysis concluded that this is a false positive, as the marked line is a statement inside an if block, which checks if the malloc call assigning a memory address to the variable `temp` succeeded. Thus, if the call to malloc fails, no memory is allocated and the variable does not have to be freed. In the same manner, another identified memory leak at position `src/Common/zlib/gzlib.c:294` is also a false positive.

The last reported error reads `Syntax Error: AST broken, ternary operator lacks ':'`. In this case, *CppCheck* is unable to parse the code line, as a ternary operator is used correctly.

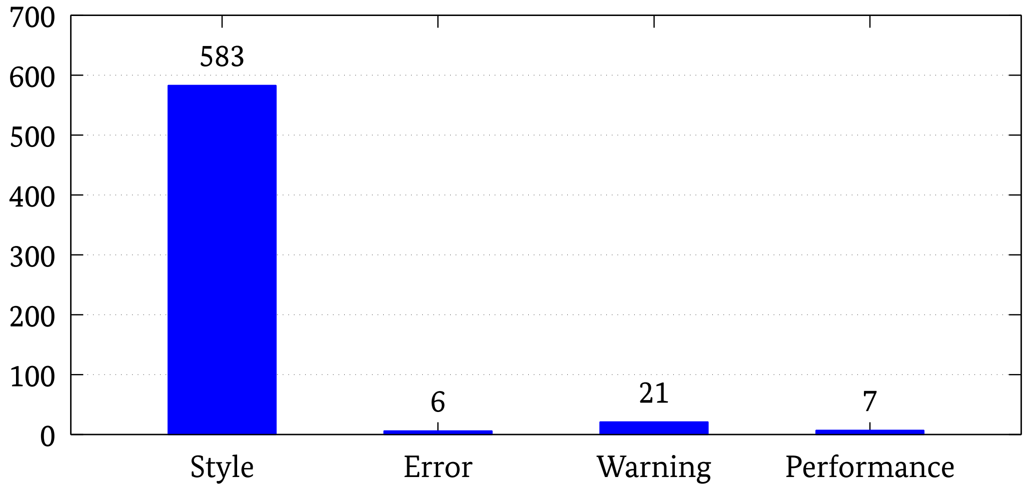


Figure 15: Number of issues found by the CppCheck analysis per category.

5.1.2 TScanCode

TScanCode[60] is a static analysis tool for C/C++, C# and Lua code. It is maintained by the Chinese company Tencent as open-source software. It focuses on a fast and accurate analysis, ease of use and extensibility. Version 2.14.2397 was used for the analysis of VeraCrypt.

5.1.2.1 Usage

The command in Listing 3 was used to run TScanCode on the VeraCrypt source code. The `–enable-all` flag enables all available analyses and the flag `-I` enables TScanCode to find header files. The stdout and stderr output were piped to different files.

```
$ ./tscancode --enable=all -I ~/veracryptscanner/VeraCrypt/src/ -I
~/veracryptscanner/VeraCrypt/src/Common/ ~/veracryptscanner/VeraCrypt/src >
~/veracryptscanner/tscanstdout.txt 2> ~/veracryptscanner/tscanstderr.txt
```

Listing 3: The TScanCode command with parameters as used for the analysis.

[60] https://github.com/Tencent/TscanCode

5.1.2.2 Results

TScanCode groups findings in different categories based on severity. Figure 16 shows an overview of these categories and the number of findings. Issues not directly related to security are tagged with the category *Information*, e.g., for cases in which code constructs can be simplified. This category is not considered in this report as it did not include security issues. The *Warning* category is used for common programming issues that might lead to problems while developing code, e.g., member variables of a class that are not initialized in the constructor. The categories *Serious* and *Critical* include code constructs that might lead to undefined behavior or crashes and thus might cause security issues.

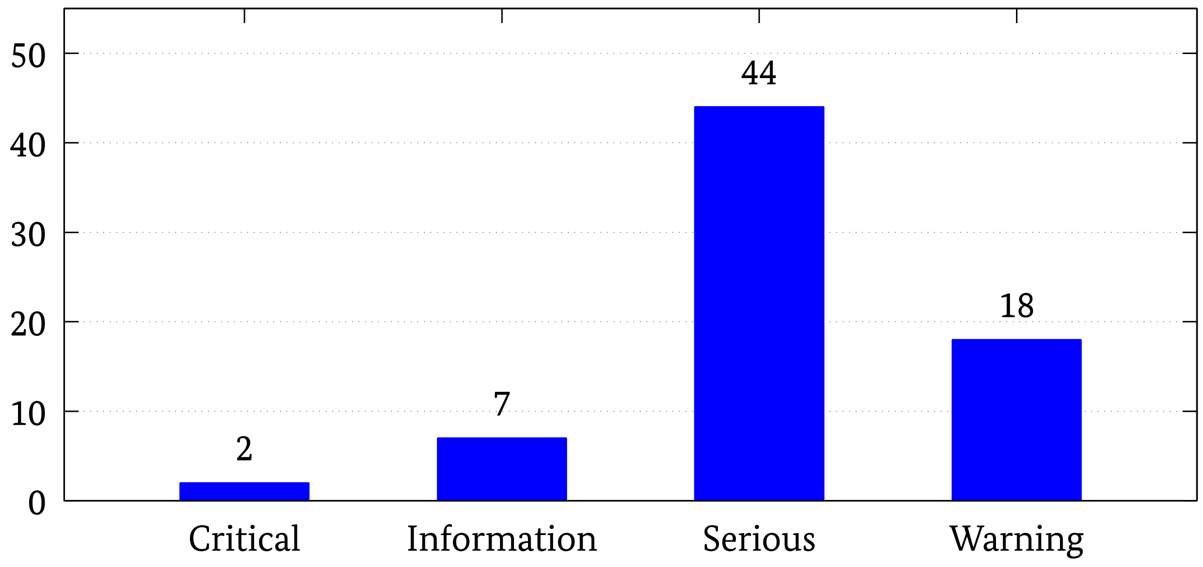


Figure 16: Number of findings by the TScanCode analysis per category.

Warning

This category includes 18 warnings. In 15 cases, TScanCode reports that a member variable of a class is not initialized in the constructor. This usually does not directly indicate a vulnerability but can lead to undefined behavior.

Another warning was issued for `src/Main/GraphicUserInterface.cpp:1989` indicating that an address of a stack variable is returned. The returned variable is of type `shared_ptr` and thus not subject to premature deletion. We found that this is not a security issue.

In another case located in src/Core/CoreBase.cpp:172, TScanCode reports a potential integer-overflow expression in which two unsigned 32-bit integers are multiplied and then stored in an unsigned 64-bit integer. We concluded after a manual inspection that this is not a security issue. One of the two unsigned 32-bit integers (sectorSize) is checked to be between 0 and 4096 thereby preventing an overflow.

The last case to discuss is a memory leak in *libzip* in `src/Common/libzip/zip_source_filep.c:332`. The variable `temp` is indeed allocated on the heap, and if the condition in line 330 evaluates to true, then the variable `temp` is not freed anymore, resulting in a memory leak. However, this does not directly lead to a security vulnerability.[61]

Serious

The *Serious* category is the largest with 44 issues.

24 of those issues indicate an uninitalized variable. While not a good coding practice, an uninitalized variable does not directly lead to a security vulnerability.

[61] According to the VeraCrypt development team, after merging *libzip* version 1.6.1 to VeraCrypt's source code, this issue was resolved starting with VeraCrypt version 1.24-Update5.

Another four cases are reported because a null check on a variable was made and handled, but after the check the variable is de-referenced. Three of those reported cases are located in *libzip* and are not a false positive. In one case in `src/Driver/DumpFilter.c:201`, it is a false positive as the null check is effective because the routine returns. Another eight issues report a de-reference of a variable before a presumed null-checking condition. This falls in the same subcategory as the previous four cases. Nevertheless, while de-referencing a null pointer will crash the program, it does not directly lead to a security issue.

It is reported twice that there is no space left for the terminating null character in the FAT filesystem formatter code, when assigning a string to the volume name. In this case the hardcoded string "`NO NAME`" with trailing spaces is assigned as volume name. After examining the code, it appears that the terminating null character is not needed in this case and that no security issue is present.

In two cases, a division by zero is reported in the *zlib* library. A manual analysis cannot confirm the finding. A division by zero might lead to a crash but not to a security vulnerability.

The last four issues are buffer accesses out of bounds in the `Driver` subfolder. Each issue is reported at `memcmp` function calls comparing two buffers. In these cases a unicode string is compared to a hardcoded string. A manual analysis concluded that this is no security issue.

Critical

Only two issues are reported in this category. Both of them state that a null pointer is de-referenced. The first case reported in `src/Core/CoreBase.cpp:40` is a false positive as the de-referenced variable gets reset in the previous line. The second case is located in `src/Common/zlib/deflate.c:928` in the *zlib* library. The control flow suggests that a null pointer de-reference is possible. An impact on security is unlikely.

5.1.3 Clang Static Analyzer

The Clang Static Analyzer[62] is an analysis tool for C, C++ and Objective-C. It is part of the Clang compiler project, which can be used to compile VeraCrypt. The tool needs the source code of the target program. While compiling a project, it can detect a variety of program errors.

5.1.3.1 Usage

Listing 4 shows the used command to start the scanner. Various checking procedures are activated with the option `-enable-all`. The used version is Clang version 6.0.0-1ubuntu2.

```
$ scan-build -enable-checker core -enable-checker alpha.core -enable-checker
security -enable-checker alpha.security make
```

Listing 4: The Clang Static Analyzer command with parameters as used for the analysis.

5.1.3.2 Results

The scanner outputs 64 potential defects. Table 5 shows an overview of the results of the Clang Static Analyzer. The table shows the type and number of findings and whether the findings are present in the VeraCrypt source itself or in a third-party library. Some bugs are only found in third-party libraries of VeraCrypt, namely the *wxwidgets* library. Issues in third-party libraries are not considered as the focus lies on the VeraCrypt code. Hence, the last three entries in Table 5 are not further discussed.

[62] https://clang-analyzer.llvm.org/scan-build.html

Table 5: The Clang Static Analyzer results with quantity and location (core code or third party).

Bug Type	***Quantity***	***In Third-Party Library***
Dead assignments	1	No
Dead initialization	2	No
Branch condition evaluates to garbage value	1	No
Cast from non-struct type to struct type	37	Partly
Dangerous pointer arithmetic	2	No
Out-of-bound array access	3	No
Result of operation is garbage or undefined	1	No
Uninitialized argument value	1	No
Use fixed address	1	No
Memory leak	2	No
Use-after-free	1	No
Potential insecure memory buffer bounds restriction in `strcat`	5	Yes
Potential insecure memory buffer bounds restriction in `strcpy`	5	Yes
`malloc` size overflow	2	Yes

Dead assignments

A dead assignment is reported in `src/Crypto/Whirlpool.c:936`. The variable `num` is written, but this value is never read again. This does not indicate a security vulnerability.

Dead initialization

In two locations in file `src/Volume/EncryptionModeXTS.cpp`, in lines 45 and 219, dead initalizations of variables are reported. Manual analysis concluded that this is the case, but not security relevant.

Branch condition evaluates to a garbage value

In `src/Main/UserPreferences.cpp:98` a branch condition contains a variable, which might evaluate to a *garbage value* (undetermined value). The code decides if a volume should be mounted read-only or not, indicating a security relevance. But, the code which might lead to an evaluation to a *garbage value* uses information entered by the user when using the VeraCrypt GUI. A user has to willfully manipulate the GUI elements to reach a state in which the branch condition evaluates to a garbage value. No realistic attack scenario can be postulated, in which an attacker could derive an actual benefit from such actions (cp. 3.3).

Cast from non-struct type to struct type

In 37 cases, a non-struct type is cast to a struct type. The analysis states that this might lead to memory access errors or data corruption. While true in general and not necessarily a good coding style, a manual review on a sample basis of the cases concluded, that the findings are not security relevant.

Dangerous Pointer Arithmetic

In two cases, in `src/Crypto/Streebog.c:4312` and `src/ Crypto/kuznyechik_simd.c:4312`, *dangerous pointer arithmetic* is reported. The findings are technically correct but the reported positions are special cases. They seem to be implementations of certain CPU instructions, in case the CPU does not have this instruction. These functions are called intrinsics. As these code constructs perform low level manipulations of data, pointer arithmetic is needed. A manual analysis found that these findings do not appear to be a security issue.

Out-of-bound array access

These issues are reported in `src/Crypto/Rmd160.c` and `/src/Common/GfMul.c`. The code locations are part of hashing or encryption code. The issues in file `/src/Common/GfMul.c` are not relevant, as the code in question is not called in the project. In `src/Crypto/Rmd160.c`, the issue is reported in code calculating a round of the hash function. A detailed assessment of the reported issues requires considerable effort and resources, and we decided to not follow up on it.

Result of operation is garbage or undefined

The reported issue in `src/Crypto/cpu.c:336` claims that the result of an operation is undefined. This code tests which x86-CPU features are available on the used processor and thus contains a considerable amount of assembler code. Clang Static Analyzer does not seem to be able to parse this code section and, hence, produces this false positive.

Uninitialized argument value

This issue is also reported in `src/Crypto/cpu.c` and is, again, a result of the inability of the Clang Static Analyzer to correctly interpret the assembler code. This is most likely a false positive.

Use fixed address

The reported issue is located in `src/Driver/Fuse/FuseService.cpp:61`. The analysis tool notes that a fixed address is used and that this code might not be portable. But the code is used in the part of VeraCrypt that manages the userspace file system. The fixed address is a flag set to the Unix-style signal handler. A manual analysis concluded that this is a false positive.

Memory Leak

Two memory leaks are reported. The first location is `src/Main/Forms/WaitDialog.cpp:26`. A new command event of the *wxwidgets* library, which is used for the GUI, is initiated and passed to a *wx* library function. This is most likely a false positive and the library cleans up the object. In either case, this is not security relevant.

The second location is `src/Core/Unix/Linux/CoreLinux.cpp:131`. The code is in a custom implementation of a smart pointer in VeraCrypt. Smart pointers are used as a wrapper around basic pointers and automatically free the memory they are pointing to when going out of scope. This simplifies memory management and there exist implementations of this concept in more recent C++ standard libraries. A manual analysis concluded that this is not a security issue.

Use-after-free

In `src/Main/TextUserInterface.cpp:78` a use-after-free memory error is reported. This issue is also related to the custom implementation of a smart pointer from the previous paragraph. The memory the smart pointer is pointing to is freed, and then the smart pointer is used again. This is a potential use-after-free vulnerability. But as this is part of the textual user interface, it might be hard for an attacker to construct an attack in this scenario. Consequently, our security model does not include such an attack, see 3.3.

5.1.4 Clang-Tidy

Clang-Tidy[63] (version 6.0.0) is a C++ linter based on Clang. It is an extensible framework for detecting and fixing errors in C++ programs. A static analysis is performed in order to detect issues like style violations, bugs or interface misuse. While providing an interface for implementing own checks, it brings a set of predefined checks.

5.1.4.1 Usage

For using Clang-Tidy, a compile command database is needed. The database is a JSON file with all commands used for compiling a project. A database for the VeraCrypt project can be created with the `bear` tool, see Listing 5.

```
cd Veracrypt/src
make clean
bear make
```

Listing 5: Creating a JSON file for Clang-Tidy using the bear command.

In Listing 6, the command for analyzing the file `Main.cpp` is shown. First, the file is specified, followed by the checks to be executed. After the double dash, parameters required for the actual compile process are passed to Clang-Tidy.

```
clang-tidy Unix/Main.cpp -checks='-*,clang-analyzer-security.*,clang-analyzer-
unix*' -- -I/tmp/bla/VeraCrypt/src/Main -I/usr/lib/x86_64-linux-
gnu/wx/include/gtk2-unicode-3.0 -I/usr/include/wx-3.0 -I/tmp/bla/VeraCrypt/src -
I/tmp/bla/VeraCrypt/src/Crypto -I/tmp/bla/VeraCrypt/src/PKCS11 -DWXUSINGDLL -
D__WXGTK__  -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE -D_LARGE_FILES
```

Listing 6: Command with parameters to run Clang-Tidy on Main.cpp.

To analyze the complete source code, the program `run-clang-tidy` can be used in combination with the compile command database as shown in Listing 7.

```
run-clang-tidy-6.0.py -checks='-*,clang-analyzer-security.*,clang-analyzer-
unix*' > clang-tidy.txt
```

Listing 7: Command with parameters to run Clang-Tidy on the whole VeraCrypt project.

5.1.4.2 Results

Clang-Tidy brings a set of predefined checks. In this evaluation, we configured the analyzer to focus on security issues. Except of some syntax errors with assembler files, Clang-Tidy did not find any issues.

Clang-Tidy tidy provides a functionality to automatically fix detected issues. Perhaps, developers used that feature in the past so that detectable issues are no longer present in the current version of VeraCrypt.

5.1.5 VisualCodeGrepper

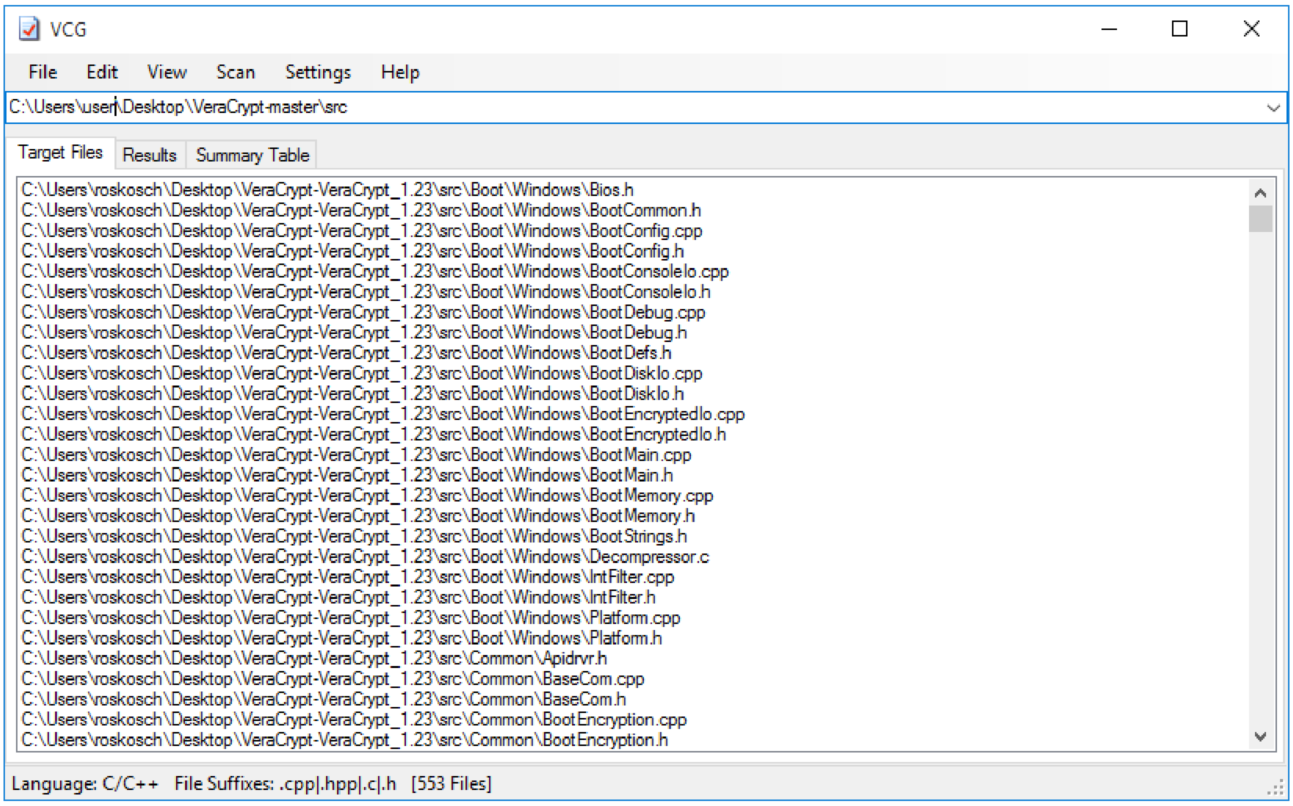
VisualCodeGrepper[64] (version 2.1.0) is a program that locates problematic and insecure code locations in order to speed up the code review process. It also scans comments for suspicious phrases that indicate

[63] https://clang.llvm.org/extra/clang-tidy/
[64] https://sourceforge.net/projects/visualcodegrepp/

broken code. The documentation of VisualCodeGrepper claims to detect buffer overflows, signed/unsigned comparisons in C and violations of OWASP[65] recommendations.

5.1.5.1 Usage

VisualCodeGrepper is a Windows program with a GUI, see Figure 17. Source code files or directories can be selected for analysis.



```
C:\Users\user\Desktop\VeraCrypt-master\src
```

```
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\Bios.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootCommon.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootConfig.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootConfig.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootConsoleIo.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootConsoleIo.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootDebug.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootDebug.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootDefs.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootDiskIo.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootDiskIo.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootEncryptedIo.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootEncryptedIo.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootMain.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootMain.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootMemory.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootMemory.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\BootStrings.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\Decompressor.c
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\IntFilter.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\IntFilter.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\Platform.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Boot\Windows\Platform.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Common\Apidrvr.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Common\BaseCom.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Common\BaseCom.h
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Common\BootEncryption.cpp
C:\Users\roskosch\Desktop\VeraCrypt-VeraCrypt_1.23\src\Common\BootEncryption.h
```

```
Language: C/C++   File Suffixes: .cpp|.hpp|.c|.h   [553 Files]
```

Figure 17: GUI dialog of VisualCodeGrepper with target source code files from VeraCrypt.

5.1.5.2 Results

Overall, 1381 findings were detected, mostly the usage of banned functions and `goto` statements. Each finding is assigned to one of the following categories: *STANDARD* (306), *MEDIUM* (1064), *HIGH* (9), *SUSPICIOUS COMMENT* (89).

HIGH

The nine findings rated *HIGH* are calls of the method `LoadLibrary`. In a source code review, we could not determine a security issue at these locations in the source code.

MEDIUM

The most interesting finding from VisualCodeGrepper was a potentially hard-coded password in the file `/src/Core/Unix/CoreService.cpp`. The found password is a dummy value: It serves as a mandatory but later omitted input to execute an elevation of privileges. Therefore, this finding is rated as false positive. All other findings are caused by potentially unsafe method calls, the number of which was infeasible to examine manually. We manually checked a sample set of those findings and could not find any true positives.

[65] Open Web Application Security Project, see https://owasp.org/

STANDARD

Many usages of potentially insecure method calls are detected and rated as *STANDARD* finding. The most interesting finding is a potential TOCTOU[66] vulnerability in the file `/src/Common/Keyfiles.c`. In a manual analysis, we figured out that this is a false positive.

SUSPICIOUS COMMENT

Suspicious comments are comments indicating locations in code that might be unfinished. As a result of a manual analysis, we rejected these findings as irrelevant for the purpose of our analysis.

5.1.6 cpplint

The tool cpplint[67] (version 1.4.4) is an open source program developed by Google. It performs static analysis to check whether code conforms Google's coding style guidelines. With a set of rules and heuristics, it tries to identify non-compliant code.

5.1.6.1 Usage

The tool cpplint was executed as shown in Listing 8. All `c`, `cpp` and `h` files have been passed to the cpplint command. The filtering of relevant input files was done with a nested find command. Results were redirected into a text file.

```
cpplint $(find . -type f -name "*.cpp" -o -name "*.h" -o -name "*.c") 2>
cpplint_results.txt
```

Listing 8: Command and parameters used for the cpplint analysis.

5.1.6.2 Results

Overall, cpplint returned 173,102 findings (see Figure 18). Each finding belongs to one of the following categories: *build*, *legal*, *readability*, *runtime*, *whitespace*. As far as we could judge the findings, they have limited security relevance. With this vast amount of findings, manual checking the impact was not feasible. We manually verified a sample set of all issues and could not find any security issues.

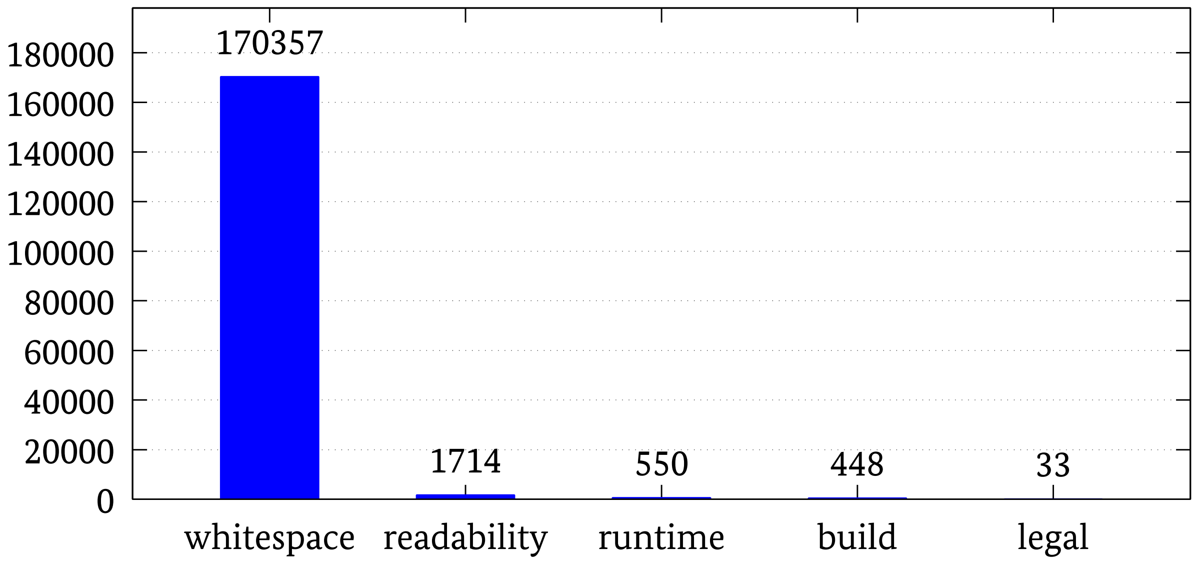


Figure 18: Number of results reported by cpplint by category.

[66] Time-of-Check-to-Time-of-Use, a specific type of a race condition.
[67] https://github.com/cpplint/cpplint

The different categories are described in the following paragraphs.

Whitespace

This is the category with most findings of 170,357. Findings in this category have no security impact. Exemplary findings: Tabulator instead of spaces, lines longer than 80 characters, missing spaces, or too many spaces before parentheses.

Readability

The tool cpplint found 1714 issues in this category. Findings in this category mention code constructs that are hard to read and therefore difficult to understand. This might introduce unnecessary bugs into software, which can be avoided by following best practices.

Runtime

The tool cpplint found 550 issues in this category. The issues mentioned in this category can cause problems when compiling software or during runtime.

Build

The tool cpplint found 448 issues in this category. It mainly tackles the style of including header files.

Legal

The tool cpplint found 33 issues in this category. The only aspect are missing copyright messages.

5.2 Dynamic Code Analysis

The previously described analysis tools perform static analysis on source code. This requires only source code without the compiled binary. The program is not executed and runtime behavior is not analyzed.

To cover dynamic analysis as well, we used Dr. Memory (Windows) and Valgrind (Linux). These tools check memory behavior of a running executable. The Linux version of VeraCrypt consists of one binary which was the subject of this analysis. The Windows version consists of multiple binaries (`VeraCrypt Format.exe`, `VeraCrypt Setup.exe`, `VeraCryptExpander.exe`, the driver `veracrypt.sys` and the GUI `VeraCrypt.exe`). For the dynamic Windows analysis, the GUI application was used.

We started VeraCrypt with both analysis tools and performed common use cases manually. This way, we triggered the major program control flows. As best effort approach this procedure has limits insofar as it is incapable to gain a complete code coverage nor path coverage.

5.2.1 Dr. Memory

Dr. Memory[68] (version 2.2.0) is a dynamic analysis tool. It is built on top of DynamoRIO[69]. With binary instrumentation, the memory of the program under test is monitored in order to find memory-related programming errors. For this evaluation, we downloaded the latest VeraCrypt binary from VeraCrypt's website.

5.2.1.1 Usage

Using Dr. Memory does not require much configuration effort. The program under test is passed to Dr. Memory, see Listing 9. Dr. Memory executes the targeted binary in its analysis environment. After starting VeraCrypt within Dr. Memory, it can be used as regularly, but program execution and, in particular, reaction to user input is considerably slowed down.

[68] https://drmemory.org/
[69] https://dynamorio.org/

```
C:\Users\user>"C:\Program Files (x86)\Dr. Memory\bin64\drmemory.exe" "C:\Program
Files\VeraCrypt\VeraCrypt.exe"
```

Listing 9: Command and parameters for the Dr. Memory analysis.

5.2.1.2 Results

Listing 10 shows an overview of the results of our Dr.Memory analysis run.

```
ERRORS FOUND:
9 unique,  1005 total unaddressable access(es)
29 unique,   557 total uninitialized access(es)
0 unique,     0 total invalid heap argument(s)
6 unique,     6 total GDI usage error(s)
0 unique,     0 total handle leak(s)
0 unique,     0 total warning(s)
1 unique,     1 total,     98 byte(s) of leak(s)
0 unique,     0 total,      0 byte(s) of possible leak(s)

ERRORS IGNORED:
1136 potential error(s) (suspected false positives)
(details: C:\Users\user\AppData\Roaming\Dr. Memory\DrMemory-
VeraCrypt.exe.5600.000\potential_errors.txt)
63 potential leak(s) (suspected false positives)
(details: C:\Users\user\AppData\Roaming\Dr. Memory\DrMemory-
VeraCrypt.exe.5600.000\potential_errors.txt)
1605 unique,  3681 total, 2320148 byte(s) of still-reachable allocation(s)
```

Listing 10: Output of a Dr. Memory analysis run.

Each finding has a description, an example is shown in Listing 11. A brief description of the error is given with a stack trace, which leads to the error. As the analysis is performed on the binary, it is hard to correlate the output of Dr. Memory to actual source code snippets. Nor is it possible to make statements whether the findings have any impact on the security of VeraCrypt or not.

```
Error #20: UNINITIALIZED READ: reading 0x0000000000146080-0x0000000000146084 4
byte(s) within 0x0000000000145cc0-0x000000000014628c
# 0 system call NtUserMessageCall COPYDATASTRUCT.lpData
# 1 USER32.dll!SendMessageTimeoutW
+0x125    (0x00007ffbf6d6cff6 <USER32.dll+0x1cff6>)
# 2 SHELL32.dll!Shell_NotifyIconW
+0x4ce    (0x00007ffbf73c169f <SHELL32.dll+0xd169f>)
# 3 zip_unchange_archive
+0x1b8db  (0x000000014002823c <VeraCrypt.exe+0x2823c>)
# 4 zip_unchange_archive
+0x32858  (0x000000014003f1b9 <VeraCrypt.exe+0x3f1b9>)
[...]
#15 USER32.dll!DialogBoxParamW
+0x84     (0x00007ffbf6d78fe5 <USER32.dll+0x28fe5>)
#16 zip_unchange_archive
+0x364e2  (0x0000000140042e43 <VeraCrypt.exe+0x42e43>)
#17 zip_unchange_archive
+0xab74f  (0x00000001400b80b0 <VeraCrypt.exe+0xb80b0>)
#18 KERNEL32.dll!BaseThreadInitThunk
```

```
+0x21     (0x00007ffbf6b21212 <KERNEL32.dll+0x11212>)
Note: @0:00:07.559 in thread 5576
```

Listing 11: A (shortened) Dr. Memory result in detail.

5.2.2 Valgrind

Valgrind is a dynamic instrumentation tool for performing memory analysis on binaries. It is capable of detecting issues in memory management and threading. Valgrind can also be used to build tools to perform custom analysis.

The source code was obtained from VeraCrypt's GitHub repository in version 1.2.3.

5.2.2.1 Usage

Valgrind[70] (version 3.13.0) does not require much configuration effort as seen in Listing 12. The program under test is passed to Valgrind. Valgrind executes the passed binary in its analysis environment. After starting VeraCrypt within Valgrind, it can be used regularly but slowed down similar to the use of Dr. Memory (cp. 5.2.1).

For the analysis, we compiled VeraCrypt with debug symbols. Debug symbols enable Valgrind to create a stack trace with specific method names for each finding. This helps to interpret the results.

As we expected to get a lot of false positives, we tried to reduce false positives as early as possible in the analysis process. We used a suppression file[71] to minimize the output of presumably irrelevant findings. For suppression, we targeted GUI libraries.

```
$valgrind /home/user/VeraCrypt/src/Main/veracrypt
```

Listing 12: Command and parameters for the Valgrind analysis.

5.2.2.2 Results

Valgrind produced a huge amount of findings, an example can be seen in Listing 13. Each finding has a brief description of the issue and the stack trace, which lead to the detected issue. Most findings include implementations of the *libwx* library, which implements GUI related functionality. By investigating the result log of Valgrind, it is especially difficult to conclude whether the reported issues actually have an impact on the security of VeraCrypt or not.

We split the complete result file to single files for each finding. This enabled us to search the findings more efficiently. Valgrind detected 260 issues, nearly all of them are uses of uninitialized values. 152 of those issues are caused by the method `aes_decrypt_key256` which is frequently used when creating a new, encrypted container file. In a source code review, we classified those issues as not security relevant.

```
==26460== HEAP SUMMARY:
    ==26460==     in use at exit: 304,094 bytes in 2,745 blocks
    ==26460==   total heap usage: 4,907 allocs, 2,162 frees, 1,301,852 bytes
allocated
    ==26460==
    ==26460== LEAK SUMMARY:
    ==26460==    definitely lost: 0 bytes in 0 blocks
    ==26460==    indirectly lost: 0 bytes in 0 blocks
    ==26460==      possibly lost: 1,736 bytes in 19 blocks
```

[70] http://valgrind.org/
[71] http://valgrind.org/docs/manual/manual-core.html\#manual-core.suppress

```
    ==26460==    still reachable: 302,358 bytes in 2,726 blocks
    ==26460==                       of which reachable via heuristic:
    ==26460==                         newarray          : 1,536 bytes in 16
blocks
    ==26460==         suppressed: 0 bytes in 0 blocks
    ==26460== Rerun with --leak-check=full to see details of leaked memory
    ==26460==
    ==26460== For counts of detected and suppressed errors, rerun with: -v
    ==26460== Use --track-origins=yes to see where uninitialised values come
from
    ==26460== ERROR SUMMARY: 38 errors from 4 contexts (suppressed: 0 from 0)
    ==26359==
    ==26359== HEAP SUMMARY:
    ==26359==     in use at exit: 2,756,118 bytes in 35,886 blocks
    ==26359==   total heap usage: 2,143,431 allocs, 2,107,545 frees, 163,664,408
bytes allocated
    ==26359==
    ==26359== LEAK SUMMARY:
    ==26359==    definitely lost: 23,820 bytes in 156 blocks
    ==26359==    indirectly lost: 89,676 bytes in 3,693 blocks
    ==26359==      possibly lost: 4,660 bytes in 53 blocks
    ==26359==    still reachable: 2,438,946 bytes in 30,435 blocks
    ==26359==                       of which reachable via heuristic:
    ==26359==                         length64          : 12,056 bytes in 173
blocks
    ==26359==                         newarray          : 2,240 bytes in 60
blocks
    ==26359==         suppressed: 0 bytes in 0 blocks
    ==26359== Rerun with --leak-check=full to see details of leaked memory
    ==26359==
    ==26359== For counts of detected and suppressed errors, rerun with: -v
    ==26359== Use --track-origins=yes to see where uninitialised values come
from
    ==26359== ERROR SUMMARY: 349367 errors from 260 contexts (suppressed: 566252
from 209)
```

Listing 13: Output from Valgrind giving a result overview.

5.3 Inspection of the Windows Kernel Driver

In Windows, VeraCrypt needs a device driver to mount devices and perform the device encryption and decryption. This kernel component is installed by the VeraCrypt setup. The VeraCrypt user space program interfaces to the driver to trigger actions to mount and unmount volumes (cp. 3.3.4).

5.3.1 Code Inspection

To start the communication with the driver, the VeraCrypt user space component uses the function `DriverAttach` in `src/Common/Dlgcode.c:4461`. This function opens a handle to the driver.

With the Windows function `DeviceIoControl` and the handle to the driver, a user-space program can communicate with the driver and issue various commands. This communication is done via IOCTL (Input/Output Control), a form of system call for device-specific input and output operations. The

`DeviceIoControl` function expects a buffer to exchange data between user space and kernel space, as well as an identifier pointing to the desired function of the driver. The corresponding functionality in the driver is selected using the identifier as parameter in a switch-case statement.

Windows-specific IO functionality is implemented in the method `ProcessVolumeDeviceControlIrp` found at `src/Driver/Ntdriver.c:631`, e.g., getting a device name or getting partition information. VeraCrypt-specific driver functionality is implemented in the method `ProcessMainDeviceControlIrp` in `src/Driver/Ntdriver.c:1674`.

Overall, there are 41 VeraCrypt-specific commands at the driver interface. Due to the specifications of the Windows Device Driver API, each command is required to inform the calling function about the execution results by setting the content of a system buffer with the data to return. Also, the driver returns for each command call an integer value indicating success or failure, as well as other status information depending on the command (`status` field).

The system buffer filled by the kernel functions with return data serves a further purpose: It is also used by the caller function to input data into the driver. This input mechanism is security-relevant because it bridges the user space and the higher privileged kernel space and can be an entry point for attackers (cp. 3.3.4). The VeraCrypt program in the user space does not necessarily run with administrative rights in Windows. An adversary with restrictive user rights may try to escalate her privileges by exploiting a security weakness in the VeraCrypt device driver.

In the following, three groups of commands with differing complexity of the input data types are discerned:

- Commands with C-style structs as complex input datatypes,
- Commands returning a primitive datatype in a system buffer provided by the user-space program, and
- Commands returning only a status value while not receiving any input data from the user-space program.

Of specific interest are the 22 commands that receive a complex datatype in form of a C-style struct by reference. To return data, the function modifies fields of this struct. The names of these commands and their respective data type for input and output operations are listed in Table 6.

Several of these commands handle the mounting and dismounting of volumes and allow reading properties or states of mounted volumes. Other commands enable the creation of encrypted boot devices and the creation of hidden volumes.

Two commands in Table 6 have the prefix `VC_IOCTL,` indicating that these commands were added by VeraCrypt.

Table 6: List of driver commands with complex parameter types.

Command Name	*IO Datatype*
TC_IOCTL_BOOT_ENCRYPTION_SETUP	BootEncryptionSetupRequest
TC_IOCTL_GET_BOOT_ENCRYPTION_STATUS	BootEncryptionStatus
VC_IOCTL_GET_BOOT_LOADER_FINGERPRINT	bootLoaderFingerprint
TC_IOCTL_GET_DECOY_SYSTEM_WIPE_STATUS	DecoySystemWipeStatus
VC_IOCTL_GET_DRIVE_GEOMETRY_EX	DISK_GEOMETRY_EX_STRUCT
TC_IOCTL_GET_DRIVE_GEOMETRY	DISK_GEOMETRY_STRUCT
TC_IOCTL_GET_DRIVE_PARTITION_INFO	DISK_PARTITION_INFO_STRUCT
TC_IOCTL_GET_BOOT_ENCRYPTION_ALGORITHM_NAME	GetBootEncryptionAlgorithmNameRequest
TC_IOCTL_GET_SYSTEM_DRIVE_CONFIG	GetSystemDriveConfigurationRequest

Command Name	*IO Datatype*
TC_IOCTL_GET_SYSTEM_DRIVE_DUMP_CONFIG	GetSystemDriveDumpConfigRequest
TC_IOCTL_GET_WARNING_FLAGS	GetWarningFlagsRequest
TC_IOCTL_GET_MOUNTED_VOLUMES	MOUNT_LIST_STRUCT
TC_IOCTL_MOUNT_VOLUME	MOUNT_STRUCT
TC_IOCTL_OPEN_TEST	OPEN_TEST_STRUCT
TC_IOCTL_PROBE_REAL_DRIVE_SIZE	ProbeRealDriveSizeRequest
TC_IOCTL_REOPEN_BOOT_VOLUME_HEADER	ReopenBootVolumeHeaderRequest
TC_IOCTL_GET_RESOLVED_SYMLINK	RESOLVE_SYMLINK_STRUCT
TC_IOCTL_DISMOUNT_VOLUME	UNMOUNT_STRUCT
TC_IOCTL_DISMOUNT_ALL_VOLUMES	UNMOUNT_STRUCT
TC_IOCTL_GET_VOLUME_PROPERTIES	VOLUME_PROPERTIES_STRUCT
TC_IOCTL_GET_BOOT_DRIVE_VOLUME_PROPERTIES	VOLUME_PROPERTIES_STRUCT
TC_IOCTL_START_DECOY_SYSTEM_WIPE	WipeDecoySystemRequest

The next group of commands uses a primitive type for communicating with the user space. To this end, they receive as input a system buffer of the size of the primitive type. Table 7 shows the eight commands with their respective type in this category.

Table 7: List of driver commands returning a primitive type in the given system buffer.

Command Name	*IO Datatype*
TC_IOCTL_GET_DEVICE_REFCOUNT	Int
TC_IOCTL_IS_DRIVER_UNLOAD_DISABLED	Int
TC_IOCTL_IS_ANY_VOLUME_MOUNTED	Int
TC_IOCTL_LEGACY_GET_MOUNTED_VOLUMES	uint32
TC_IOCTL_GET_BOOT_LOADER_VERSION	uint16
TC_IOCTL_IS_HIDDEN_SYSTEM_RUNNING	Bool
TC_IOCTL_GET_DRIVER_VERSION	Long
TC_IOCTL_LEGACY_GET_DRIVER_VERSION	Long

The last category with 11 entries shown in Table 8 does not exchange information with the user space program by means of a system buffer. The commands trigger actions like wiping caches, or return status information of different features of the device driver.

Table 8: List of driver commands only returning a status but no return value.

Command Name
TC_IOCTL_SET_PORTABLE_MODE_STATUS
TC_IOCTL_WIPE_PASSWORD_CACHE
TC_IOCTL_GET_PASSWORD_CACHE_STATUS
TC_IOCTL_GET_PORTABLE_MODE_STATUS

Command Name
TC_IOCTL_ABORT_BOOT_ENCRYPTION_SETUP
TC_IOCTL_GET_BOOT_ENCRYPTION_SETUP_RESULT
TC_IOCTL_ABORT_DECOY_SYSTEM_WIPE
TC_IOCTL_GET_DECOY_SYSTEM_WIPE_RESULT
TC_IOCTL_WRITE_BOOT_DRIVE_SECTOR
TC_IOCTL_SET_SYSTEM_FAVORITE_VOLUME_DIRTY
TC_IOCTL_REREAD_DRIVER_CONFIG

The code of each command was reviewed for weaknesses in the input processing. The focus was in particular on the commands using complex input types. Here, operations were checked as to whether they test the size of a potentially used input system buffer. For the two other classes of commands, it was checked whether and how an input system buffer was actually used.

This analysis did not reveal any security issues: The Windows driver code includes a high number of buffer size checks as well as checks if an operation was successful or not.

As a second test procedure, a dynamic analysis using fuzzing techniques was performed as described in the following.

5.3.2 Dynamic Analysis

To further assess the security of the device driver, a dynamic analysis was done. The idea was to trigger an error in the application to induce unexpected behavior, for example crashing the driver.

In a first step, each of the 41 VeraCrypt-specific IOCTL commands was executed with random data in the system buffer used to input data in the driver function. For this purpose, we modified the VeraCrypt user space application for Windows to execute the commands with random data. Multiple invocations with random bytes did not lead to unexpected results. The VeraCrypt device driver checks the input buffer for expected size and stops command execution gracefully.

As described in Section 5.3.1, some commands expect complex datatypes and others primitive data types in the shared system buffer. The driver checks the correct size of the buffer for each command. If the size is not correct, the driver does not accept the command.

In a second step, a selection of commands from Table 6 expecting complex structs as input was chosen for more sophisticated tests than just entering random data: Test cases were created to pass the size checks of the VeraCrypt device driver and to affect code executed later in the input processing pipeline.

The commands `TC_IOCTL_OPEN_TEST`, `TC_IOCTL_GET_RESOLVED_SYMLINK` and `TC_IOCTL_GET_DRIVE_PARTITION_INFO` underwent this additional vetting because they feature the most code of all command procedures. These commands expect structs with various flags and path strings. The fields of the struct were filled with random data to potentially trigger an error.

The VeraCrypt device driver did not run into an unexpected error state during these tests; all erroneous payloads were handled gracefully. The VeraCrypt device driver uses several checks to identify and deny input system buffers of unexpected size. Because of the multiple error handling and success checks for many of the in-driver operations, all our attempts to trigger driver failures by randomly inserting data failed.

5.4 Security of Header Parser for Container Files

VeraCrypt can create volumes within container files that can be stored at untrusted locations or sent over public communication channels while providing confidentiality through the data encryption. In an environment where containers are shared on a regular basis over untrusted channels, it cannot be prevented that a container file is altered by an attacker to cause malfunctions of VeraCrypt and, e.g., execute malicious code on the machine of a legitimate user of the container, as described in the security model in 3.3.5. VeraCrypt must ensure that invalid containers are detected correctly and are rejected without reaching unexpected error states.

A preferable entry point for an attacker is the header parser of VeraCrypt that processes the header data of a container file to retrieve the parameters necessary to mount the file. The robustness of this header parser is evaluated with fuzzing.

Fuzzing is a dynamic software testing technique. A permutation algorithm generates invalid inputs for a program based on valid input samples, the corpus. These inputs are fed into the program under test while monitoring the program's output values in response to the invalid input files. In case the program crashes, the cause of this crash can be further evaluated in order to find security bugs. Once set up, fuzzing can be performed without any supervision, thereby making it an efficient software testing technique.

5.4.1 Fuzzer

For this analysis, the fuzzer *american fuzzy lop plus plus (afl++)*[72] has been used which is an extension to the popular *american fuzzy lop (afl)*[73]. *afl* is the basis for a variety of research in the field of source code and binary fuzzing. Each research group that uses and extends *afl* for their purpose creates a new source code fork of *afl* adding new features or modifying existing ones.

afl++ is an actively supported fork of *afl* merging a considerable set of such new features and research results together. As a further advantage for this project, *afl++* compiles under current operating system versions. Hence, we were able to perform the evaluation on a server with an up-to-date Ubuntu 18.04 LTS version by compiling the current version 2.60c of *afl++* from its source, as shown in Listing 14.

```
git clone https://github.com/vanhauser-thc/AFLplusplus.git
git checkout 2.60c
make clean && make distrib
sudo make install
```

Listing 14: Commands for obtaining and compiling afl++ as used for the analysis.

5.4.2 Code Coverage

As for any dynamic testing technique, achieving a large code coverage is a challenge when fuzzing a program. When feeding random inputs to VeraCrypt, the program itself does not indicate which parts of the program logic has been executed nor how often. *afl++* solves this issue by instrumenting the program binary during compile time. This enables *afl++* to measure the code coverage and reason about the quality of generated input samples. With this knowledge, the generation of input samples can be guided to increase the code coverage.

afl++ provides compilers (`afl-gcc, afl-g++`) for compiling the application to apply the instrumentation. Because VeraCrypt is open source, these compilers can be used to create an instrumented

[72] https://github.com/vanhauser-thc/AFLplusplus
[73] http://lcamtuf.coredump.cx/afl/

binary from the available source code. The actual fuzzer (`afl-fuzz`) recognizes this instrumented binary and can use it for increasing the effectiveness of the fuzzing process.

Before compiling VeraCrypt with the *afl++* compilers, it was necessary to select the appropriate compilers with a suitable parametrization. Listing 15 shows the commands used. To execute the fuzzing on a high-performance machine, a binary of VeraCrypt was needed that can run in a text mode on a headless server. For this purpose, compilation was done with the `NOGUI` parameter. This parameter requires a statically linked `wxwidget`[74] library.

As VeraCrypt has a bug when compiling with `NOGUI` parameter in version 1.23, the current version at test time *VeraCrypt_1.24-Update3* was used for the fuzzing test.

```
export CC=afl-gcc
export CXX=afl-g++
make NOGUI=1 WXSTATIC=1 WX_ROOT=~/wxWidgets-3.0.4 wxbuild
make NOGUI=1 WXSTATIC=1
```

Listing 15: Commands with parameters for building and instrumenting VeraCrypt using the afl++ compilers.

5.4.3 Corpus

The corpus is a set of valid input files for the program under test. It is the starting point to generate invalid input files by slightly changing these valid base files. Fuzzing is a dynamic program testing technique that can only test those parts of the program under test that are reached in the control flow while processing the given input files. To increase this code coverage, a corpus with diverse files has been chosen for this test. The corpus for this evaluation is listed in the Appendix, see Section 9.2.2.

5.4.4 Modification to VeraCrypt

VeraCrypt's Linux version has been used for the fuzzing test as *afl++* and most other *afl* variants only run under Linux. The fuzzing process targets mainly the parsing logic for given input files of the program under test. All VeraCrypt versions for the various operating systems share the same input parsing logic, also allowing to use the same container files on different operating systems with VeraCrypt.

A fuzzer can generate and test multiple hundred input samples per second depending on the execution speed of the tested program. To reduce the time for one run, VeraCrypt was modified to not actually mount a given input file. This modification did not reduce the scope of the test in a relevant way, because the tests aimed on detecting errors in the parsing and processing logic of the volume header. The mounting procedures use primitives of the operating system and rely on the robustness of the operating system to check for erroneous inputs. These checks by the operating system are, however, not a subject of this analysis.

5.4.5 Fuzzing the Decryption Routine

For the first approach, the corpus presented in Section 9.2.2 was used as it is without any further modifications. The header format is explained in the VeraCrypt documentation[75]. The documentation shows that the complete container is encrypted and that there are no plaintext header parts besides an unencrypted *salt* that is publicly known random data. When the fuzzer derives input samples from the corpus, a single modification changes at least a complete cipher block or even more, depending on the modification of the fuzzer. When VeraCrypt decrypts such input samples, the structure of the original header is lost.

[74] https://www.wxwidgets.org/
[75] https://www.veracrypt.fr/en/VeraCrypt%20Volume%20Format%20Specification.html

An attacker with access to an encrypted volume file can modify a container file in arbitrary ways without knowing its decryption password. Exploitable bugs in the decryption routine and the subsequent parser logic are critical. To test its robustness, this fuzzing evaluation has been performed.

In a week of fuzzing with *afl++* on a dedicated high-performance test machine (see 9.2.1), no crash of VeraCrypt could be triggered.

5.4.6 Remove Encryption/Decryption Layer and CRC Checks

The second fuzzing approach was more sophisticated: By removing the decryption layer from the input processing pipeline, the fuzzing test could more directly aim on the header parsing logic. This was achieved by changing the source code of VeraCrypt to disable the encryption and decryption functions. The resulting binary was used to create a container file with an unencrypted plaintext header. Permutations derived from this file are more meaningful, i.e., changing single fields in the header does not lead to the change of a complete cipher block.

However, each VeraCrypt header contains two CRC-32 checksums for detecting erroneous container files. Random changes to the header applied by a fuzzer invalidate the CRC-32 checksums with high probability. When VeraCrypt detects wrong checksums, the processing is stopped and the fuzzed header fields are not processed further. These checks prevent the fuzzer from reaching more paths of the control flow.

There are multiple options to circumvent the CRC-32 checksum check: The fuzzer could calculate the correct checksums for each generated input sample. This is the most realistic approach as the generated input samples will be processed like a valid input container file. Modifying the fuzzer in order to generate valid checksums is, however, a time-consuming endeavor and was therefore not implemented.

Instead, for this evaluation, the global CRC-32 calculation method of VeraCrypt was modified to always return a constant value. When generating a volume file, this constant is inserted into the respective header fields regardless of the header's content. The same applies during the processing of a volume file and every CRC-32 correctness check will evaluate to true as always the same values are compared.

Because the encryption and decryption functions were disabled for this test, the PBKDF-2 key derivation functions also served no further purpose and were removed to lower the execution time for each run. Unfortunately, the execution is still rather slow (around 12 executions per second on a regular desktop computer). To increase the throughput, a server with high-performance hardware was used (see Section 9.2.1).

In three weeks of fuzzing, no crash could be triggered.

5.4.7 Separating the Header Parsing Routine

Even with a high-performance fuzzing server, the execution speed remained rather slow. To detect the performance bottleneck, the execution was profiled with Valgrind; the used commands are shown in Listing 16. The profiling results showed that the `wxEntry`[76] method consumes most of the execution time when starting VeraCrypt, even if it initializes a text user interface. To spare the initialization time, the header parsing logic was separated completely from the rest of the application. This increased the execution speed to around 450 executions per second on a regular desktop computer.

In total, 22 different control-flow paths were discovered within the first 24h of fuzzing. After further three weeks of fuzzing without any new paths nor crashes of VeraCrypt, we stopped the fuzzing test, because chances to reach new control flow paths seemed very low.

[76] https://docs.wxwidgets.org/trunk/group__group__funcmacro__appinitterm.html

```
valgrind --tool=callgrind ./veracrypt -t -k "" -p
ahy1eiV2eim3mohKNiefoC4kNiefoC4k --pim=0 --protect-hidden=no test.hc /tmp/volume
```

Listing 16: Commands and parameters for profiling VeraCrypt using Valgrind.

5.4.8 Summary

With all the fuzzing efforts, no input file could be created that caused a software failure within VeraCrypt.

5.5 Security of Third-Party Libraries

The VeraCrypt developers incorporate two third-party libraries in their code:

- *Libzip*[77] in its latest version 1.5.2 released on March 12, 2019, and
- *Zlib*[78] in version 1.2.11, which is also the latest version but was not updated for several years since January 15, 2017.

There exists no known vulnerabilities for these versions. These zip-archive-related libraries are apparently only used by the setup routines to extract setup files for the installer.

Other third-party libraries are not used in the project, but some implementations of cryptographic operations were taken from project-external sources (cp. 4.5.1). The VeraCrypt developers thank multiple individuals on their website[79] who provided them with code, mostly cryptography-related.

[77] https://libzip.org/
[78] http://www.zlib.net/
[79] https://www.veracrypt.fr/en/Acknowledgements.html

6 Code Quality and Documentation

6.1 Evaluation of VeraCrypt's Code Quality

In this section, we present results of an evaluation of VeraCrypt's code quality. The internal structuring and design of the source code is an indirect indicator of the software quality; it allows to reach conclusions, e.g., about the software's maintainability. Good code quality can also prevent that adding or modifying functionality introduces error and that fixing errors does not lead to further new errors.

6.1.1 Programming Guidelines and Best Practices

On VeraCrypt's GitHub page, there is no information about used programming guidelines available for third-party developers. This observation is probably caused by the fact that VeraCrypt is developed mainly by one developer alone, as documented in Section 2.2: when developing alone, a detailed documentation of programming guidelines is not important in order to build a uniformly styled code basis. The documentation also states that third-party developers are advised to contact the project's maintainer before developing a feature. This also supports the impression that collaborative development on VeraCrypt is an exception rather than the rule.

Function naming

Function names in the source code are mainly camel case, but in few cases they also contain underscores. This is a further indicator for missing programming guidelines. This observation can also be owed by the fact that VeraCrypt mainly consists of TrueCrypt's legacy code, which has not been refactored by the VeraCrypt developers.

Violation of generally accepted rules: GOTO statements

Existing research (52) and pioneers of computer science like Edsger W. Dijkstra have argued for decades against the use of `goto` statements. Exceptions might be cleanup code at the end of a function, which is still common practice in C code (52).

As in the TrueCrypt source code, the VeraCrypt source code makes heavy use of `goto` statements to jump to the end of a function to clean up memory in case of an error. There are 626 occurrences of a `goto` statement in VeraCrypt 1.23. A similar analysis done for the previous TrueCrypt study counted 388 `goto` statements (53).

Considering the diffs in all git commits in the repository between the first commit adding the TrueCrypt code and the commit tagged with version 1.23, 448 `goto s`tatements were removed, but 614 `goto` statements were also added. This increase in `goto` statements shows that this practice continued for the development of VeraCrypt.

Violation of generally accepted rules: multiple returns

Multiple return statements in a function are not considered good style except when improving readability (54). VeraCrypt's source code contains return statements to improve readability by exiting a function when certain preconditions are not met.

In other instances, functions return in the middle of their body, deep within control structures. An example is the function `DispatchControl` in file `src/Driver/Ntdriver.c:328`. The function consists of 151 lines of code and features 14 return statements scattered across the function's body.

The issue of using multiple returns was already present in the TrueCrypt source. Nevertheless, this practice apparently continues in VeraCrypt. For example, the function at `src/Setup/Setup.c:2572` introduced by the VeraCrypt project features multiple returns. Such code constructs hinder readability and code maintenance.

Violation of generally accepted rules: application logic in header files

In some cases, for instance `./Main/Forms/WaitDialog.h`, the declared functions are implemented in a header file. The C++ Core Guidelines advise against this practice[80] as it can lead to programming and compile errors, misunderstandings among developers, as well as longer compile times. Also new code developed in the VeraCrypt project shows this inappropriate coding style. For instance, in file `src/Volume/Volume.h`, two functions related to TrueCrypt support functionality are implemented in the header (lines 43 and 56).

Code Comments

The code is annotated with very few comments. For instance, only few functions contain a comment explaining their purpose and usage. This style is continued with VeraCrypt. When analyzing code diffs between VeraCrypt version 1.22 and 1.23, it is evident that no extensive efforts were undertaken to improve this situation. This lack of code comments, however, makes the code difficult to read and maintain.

6.1.2 Source Code Complexity

Code complexity should be as low as feasible to keep the project maintainable. Simple metrics for code complexity are function lengths (lines of code), the number of parameters a function takes and the control flow complexity. The Cyclomatic Complexity is a metric for control flow complexity. In (55), a Cyclomatic Complexity greater than 30 is stated as an indicator for error-prone code. Functions with values greater than 15 shall be refactored.

We measured the Cyclomatic Complexity from VeraCrypt with the tool *Lizard*[81]. It measures the Cyclomatic Complexity of functions from different programming languages and also from C/C++ projects. A warning for a function is issued when Cyclomatic Complexity is greater than 15, the function has more than 1000 lines of code or the function takes more than 100 parameters. For the VeraCrypt's source code, Lizard gives a warning for 286 functions, which is an increase by more than 100 warnings compared to TrueCrypt in version 7.1a (53).

6.1.3 Code Duplicates

Code duplicates are identical sequences of code found in multiple locations in the program. These duplicates should be refactored to one function to prevent multiple versions of the same functionality in the same code base and to improve maintainability.

We used the analysis tool *Duplo*[82] to measure the number of code duplicates. The tool analyzed 118,345 lines of code and found 19,558 duplicate lines of code. These duplicate lines of code were found in 3419 duplicate blocks, indicating a high quantity of code duplicates.

6.1.4 Build Process

To modify software features, the software's source code needs to be changed or extended and re-built. Building a project like VeraCrypt can be difficult due to dependencies to other projects or library versions. The build process requirements and their documentation indicate whether the build process itself is maintained and kept up to date. Hence, in the following, the build process for Linux and Windows is evaluated.

[80] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md\#sf2-a-h-file-may-not-contain-object-definitions-or-non-inline-function-definitions
[81] https://github.com/terryyin/lizard (version 1.16.6)
[82] https://github.com/dlidstrom/Duplo (version 0.3.0)

Linux

Building VeraCrypt on Linux (Ubuntu 18.04) requires to resolve some dependencies[83]. The dependencies can be installed with Ubuntu's packet manager[84]. Afterwards, VeraCrypt can be built and installed by running the `make` command. The overall building process is straight forward and does not require a deep understanding of the underlying source code.

Windows

Building VeraCrypt on Windows (Windows 10) requires to resolve multiple dependencies by manually downloading and installing further development software and software libraries[85].

However, direct links to the sources of the software are not provided. Required software versions are outdated. For example, only by subscribing to a Microsoft account and joining certain developer programs it is possible to acquire *Visual Studio 2010*. Current software versions are not compatible in every case.

The VeraCrypt GitHub page states that the required *Microsoft Visual C++ 1.52* compiler from 1995 should be available on *MSDN Subscriber Downloads*, but this portal has since presumably been replaced by the *Microsoft Developer Portal* in which we could not find the compiler in the specified version. Getting every component required for a working build environment to run on a modern Windows installation was an elaborate task. As we encountered many errors during this process, we were forced to de-install every installed Visual Studio, SDK, and Microsoft Development component from our test machine and to start again from scratch.

Furthermore, since no direct download links for required components are provided by VeraCrypt's maintainers, developers may tend to get the required software from unofficial sources because a quick internet search does not lead to official sources. This imposes a further risk to access web pages distributing malware when searching for a quick download of outdated software.

Overall, the build process for Windows does not seem to be well maintained. It is difficult to re-produce from the documentation, and considerable preparation efforts need to be taken into account.

6.1.5 Test Cases Review

The basic features are tested in a single batch file `test/bench.bat`. However, it seems more like a performance benchmark than functional tests. To assure the functioning of an extensive software like VeraCrypt, a broad set of test cases is required, but an actual test management process is not established in the VeraCrypt project.

6.1.6 Automated Analysis

As described in 5.1.1, we used the tool *CppCheck* to automatically analyze the source code for security issues. The tool also checks the style of the analyzed source code. The findings from these checks are presented in Table 9 that also shows how often a finding occurred.

Table 9: Findings and their number of occurrence from the CppCheck code quality check.

Finding	*Number of Occurrences*
Never used functions	379
Not explicit constructor	95
Scope can be reduced	89

[83] https://github.com/veracrypt/VeraCrypt#requirements-for-building-veracrypt-for-linux-and-mac-os-x
[84] https://wiki.ubuntuusers.de/APT/
[85] https://github.com/veracrypt/VeraCrypt#requirements-for-building-veracrypt-for-windows

Finding	*Number of Occurrences*
C-style pointer casting	3
Clarify calculation	6
Miscellaneous	6

Never used functions

In 379 cases, a function was defined but never used anywhere. 117 of those functions are found in the library *zlib*, 60 functions are found in the library *libzip*. The code of these libraries is located in VeraCrypt's source code directory. Those libraries are only required for the setup component of VeraCrypt, see 5.5. This explains the high number of unused functions in these libraries. 202 functions are defined in VeraCrypt and never used and are not part of a third-party library.

Unused functions indicate dead or deprecated code, which also has to be maintained and can introduce errors. Developers might be tempted to use these functions despite updated alternatives are available in the code base. Hence, it is advised to remove unused functions.

Not explicit constructor argument

When a constructor in C++ receives only a single argument, the compiler can make implicit conversions. This is illustrated in the example in Listing 17.

```
1    class Example {
2       public:
3           Example(int test) : m_test(test) {}
4       private:
5           int m_test;
6    };
7
8    void AnotherFunction (Example ex) {
9      [...]
10   }
11
12   int main() {
13      AnotherFunction(100);
14   }
```

Listing 17: Example for implicit type conversion in C++.

The class `Example` has a constructor receiving one argument. The function `AnotherFunction` takes an object of the class `Example` as an argument. In the main function, `AnotherFunction` is called but not directly with an `Example` object as argument but with an integer. The compiler does an implicit conversion and uses the integer argument in line 13 to create an object of type `Example` calling its constructor in line 3. To prevent the compiler to behave this way, the constructor can be made *explicit*. This forces the compiler to throw an error when compiling Listing 17.

CppCheck reports 95 not explicit constructors. This construct may allow unintended conversions and can induce programming errors. The C++ Core Guidelines discourage the use of non-explicit constructors[86].

Scope can be reduced

In 89 cases, the scope of variables can be reduced. In these cases, a variable is defined and used, but after a specific point in the current scope the variable is not used anymore. The variable is then accessible later in

[86] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md\#Rc-explicit

the program although this is not required for the code to function. Reducing the variable's scope can increase readability and prevent errors.

C-style pointer casting

In C++, multiple functions exist to safely cast between types. The C-style pointer casting using parenthesis should not be used in C++ as this practice can introduce several errors. The C++ Core Guidelines also advise against using C-style casts[87]. Three instances of this problem were found in VeraCrypt.

Clarify calculation

In six cases, a more complex calculation using different operators is reported by *CppCheck* because the intended precedence of the operators is not indicated using parentheses. This may introduce programming errors as developers might expect a different execution order for these calculations. Indicating precedence also improves readability.

Miscellaneous

Six miscellaneous cases are reported. These include scattered findings like an unused variable, not assigned variables or redundant conditions. As the number of findings in these categories are low, we do not further describe each case. Nevertheless, these issues should be fixed in order to prevent errors.

6.2 Evaluation of Documentation

The documentation available on the VeraCrypt website contains information for both developers and end users (56). It is largely a copy of the documentation of TrueCrypt (cp. (53) on p. 40), but changes and additions were made:

- TrueCrypt screenshots were swapped for screenshots of VeraCrypt.
- Support of further operating systems was documented.
- "VeraCrypt Rescue Disk on USB Stick" was described.
- New options for settings such as "Performance and Driver Options" and "Temporary Cache password" were added.
- "TrueCrypt Support", "Converting TrueCrypt Volumes & Partitions" and "Default Mount Parameters" were added as new sections.
- Language pack documentation was revised as functions are now simplified.
- Description of the Kuznyechik encryption algorithm was added as well as a refined description of cascaded encryption.
- Description of SHA-256 and Streebog algorithms were added.
- "Format.exe" as command line option was more extensively described.
- Description of "Wear Leveling" was added.
- The sections "Troubleshooting", "Incompatibilities", "Known Issues and Limitations" and "Frequently Asked Questions" were updated to match the current status of the software.
- Changes in the header key derivation algorithm were documented in the section "Header Key Derivation, Salt, and Iteration Count"; the newly added "PIM" functionality was documented.
- Section "Compliance with Standards and Specifications" now documents compatibility with FIPS 140-2.

All changes and additions were fitted into the structure of the former TrueCrypt documentation. The documentation, while on the one hand rather comprehensive, still has shortcomings already discussed for

[87] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md\#es48-avoid-casts

TrueCrypt (53). For instance, it comprises only limited information for end users. The documentation describes the general workflow for using VeraCrypt in multiple scenarios such as container or system encryption. But it lacks detailed recommendations such as a simplified user guideline outlining which encryption and hash algorithms are appropriate in certain scenarios. Also, incorrect descriptions of the content of the volume header and keypool mixing algorithm are still included in the documentation (cp. (53) on p. 41).

Overall it appears that the documentation was amended for describing the changes or improvements made in VeraCrypt. However, it does not seem that it has been revised as a whole.

A particular weakness of the documentation is the description of program settings. Occasionally settings are described in the documentation but do not exist at the user interface and vice versa. Examples with security relevance are:

- The "Cache Password in Driver Memory" setting is described in the documentation but does not appear at the user interface of the Linux version. There is a similar looking setting "Cache passwords and keyfiles in memory" in the user interface for mounting single volumes, but it is not clear whether its functionality matches the one described in the documentation. There exists a further general setting "Cache passwords in memory" available in the Linux version for which it is also unclear how its functionality relates to the other two mentioned options.
- The following application settings are provided by the user interface, but not described in the documentation:
  - "Do not use kernel cryptographic services",
  - "Preserve modification timestamp of file containers", and
  - "Close token session (log out) after a volume is successfully mounted".
- The following application settings are described in the documentation, but only provided by the user interface of the Windows version:
  - "Temporary Cache password during 'Mount Favorite Volumes' ",
  - "Auto-dismount volume after no data has been read/written to it for", and
  - "Force auto-dismount even if volume contains open files or directories".
- Differing from the general structure of the documentation, the setting "Do not accelerate AES encryption/decryption by using the AES instructions of the processor" is documented in a section dedicated to describe hardware-accelerated AES encryption.

For developers and security experts the documentation still contains many details about the software's functionalities, its security model and respective security considerations. In addition, the VeraCrypt project provides a publicly available development management system including:

- a version control system to track and document all changes made in the source code,
- a trouble ticket system allowing to contact VeraCrypt developers as well as getting informed of and involved into conversations of others with the VeraCrypt developers, and
- pull and merge functions of the version control system allowing other developers to contribute to the project.

This is a considerable improvement for developers over the TrueCrypt project, where contacting the TrueCrypt developers via e-mail was the only way to interact with and contribute to the project.

6.3 Summary

Overall, the VeraCrypt project continues the coding and documentation practices of the TrueCrypt project. The code still shows major quality problems already outlined in the previous TrueCrypt study (53). Moreover, newly added code introduces new instances of these problems, while it does not seem that a considerable effort was undertaken to solve the issues of TrueCrypt's legacy code. For instance, a refactoring of the source code was not performed.

As with the code, the documentation of VeraCrypt builds primarily on the documentation of TrueCrypt which is quite comprehensive, especially those sections intended for security experts and developers. The documentation was changed accordingly when changes in VeraCrypt were applied and new functions were introduced; it appears that the documentation is largely up to date.

A major improvement towards the TrueCrypt development project are the publicly available development management tools, especially the project's GitHub site. These tools allow a better tracking of code changes and manifold ways of interacting with the developers. By contrast, the development of TrueCrypt was opaque for people outside the project with very limited means to interact with the developers and the project.

7 Previous Work on the Security of VeraCrypt

This section summarizes our research on previous security evaluations of VeraCrypt. With a focus on collecting know weaknesses and vulnerabilities of VeraCrypt, we investigated academic publications (see 7.1), publications from the government, non-government and industry sector (see 7.2), as well as data from the development platforms of VeraCrypt (see 7.3). We collected security findings for both VeraCrypt as well as TrueCrypt, and analyzed their status with respect to version 1.23 of VeraCrypt. Furthermore, we created a brief collection of analysis methods applied for these previous security evaluations, see Appendix, Section 9.3.

7.1 Academic Publications

In academia, TrueCrypt or VeraCrypt are used as examples of hard disk encryption tools to investigate or discuss more general research questions in cryptography. However, only few peer-reviewed research publications exist actually discussing the security of TrueCrypt or VeraCrypt, and those focus mostly on features specific to both encryption tools, but not overall software security. In the following, two papers are presented reporting research on the security of the PBKDF2 key derivation algorithm and hidden volumes.

7.1.1 PBKDF2 Key Derivation

A recent study of Visconti et al. discusses the security of the PBKDF2 key stretching function. The study focuses on *LUKS* (Linux Unified Key Setup) (57). However, the results extend to TrueCrypt and VeraCrypt since both use PBKDF2. Moreover, the authors compared the costs to attack a VeraCrypt password to the costs of attacking a *Cryptsetup* password.

Attacks were executed with GPUs (Graphics Processing Units), thereby reducing the attack cost and overall accelerating the attack. This method of brute-forcing passwords is enabled by the design of PBKDF2: It is not a memory-intensive key derivation function, thereby allowing an efficient, massively parallel brute-force search on GPUs despite the memory restrictions of this computing architecture.

The authors found that costs of attacks against VeraCrypt and *Cryptsetup* 1.7.0 are comparable. This result is insofar significant as *Cryptsetup* adapts the number of PBKDF2 to the computing power of current desktop computer hardware while VeraCrypt uses a fixed iteration count between 327,661 and 655,331. The authors implicate that the fixed interaction count of VeraCrypt provides a security margin comparable to current state of the art of key management systems using PBKDF2 with a dynamic set up of iteration counts.

7.1.2 Hidden Volumes

A study of Kedziora et al. discusses the plausible deniability of hidden operating systems (58). The authors claim having been able to identify areas of lower entropy in an encrypted volume hosting a hidden encrypted volume with an operating system. However, while the authors present their observations made with a single test system, they do not further investigate the cause of low entropy regions. In addition, the observation is not verified with appropriate tests. The authors also discuss a further scenario of an attacker obtaining multiple copies of a drive containing encrypted data, all created at a different point in time. The authors demonstrate that by statistical analysis of changed data blocks on the drive, the existence of a hidden volume as well as its size can be estimated.

7.2 Security Analysis Reports

7.2.1 Open Crypto Audit Project: TrueCrypt - Security Assessment

In 2014, iSEC Partners performed a security assessment of TrueCrypt on behalf of the Open Crypto Audit Project (59). The Open Crypto Audit Project is a loosely connected community of security researchers. With a source code audit, hands on testing and fuzzing, eleven vulnerabilities were discovered. In (15), these vulnerabilities have been re-checked. In case a fix to a reported issue existed, it was cross-checked whether it was suitable to remediate the found security weakness. Most of the reported security issues have been fixed, which is explained in more detail in Section 7.2.3.

7.2.2 Open Crypto Audit Project: TrueCrypt - Cryptographic Review

In 2015, the NCC Group performed a cryptographic review of TrueCrypt, also on behalf of the Open Crypto Audit Project (60). For the purpose of this review, security experts performed a source code audit and dynamic analysis. The volume header format was checked for design and implementation flaws. Also, assumptions about API behavior were verified.

Four vulnerabilities were identified. Each found vulnerability was re-evaluated in (15). One out of four vulnerabilities got fixed. For more details see section 7.2.3.

7.2.3 VeraCrypt 1.18 Security Assessment

The security consultancy company Quarkslab performed a security assessment on VeraCrypt 1.18 (15). As reported by Quarkslab, the investigation took a total of 32 work-days and focused mainly on re-evaluating fixes of previously detected security issues of TrueCrypt and on investigating VeraCrypt features in regard to security.

Status of reported TrueCrypt vulnerabilities in VeraCrypt

Quarkslab's report states that multiple vulnerabilities found in TrueCrypt have been fixed in VeraCrypt. Quarkslab also verified that the fixes were applied correctly. However, Quarkslab also found that for a couple of vulnerabilities attempts to apply a fix would have caused incompatibilities to TrueCrypt or would require substantial changes to VeraCrypt's source code or architecture. This problem applies to the following reported security issues (excerpt from (15)):

- `TC_IOCTL_OPEN_TEST` multiple issues (need to change the application behavior),
- `EncryptDataUnits()` lacks error handling (need to design a new logic to retrieve errors),
- AES implementation susceptible to cache-timing attacks (need to fully rewrite the AES implementations),
- Keyfile mixing is not cryptographically sound, and
- Unauthenticated ciphertext in volume headers.

We checked whether at the time of writing this report these vulnerabilities had been addressed. We found no indications, neither in the release notes nor the source code history that this was the case.

New security issues reported by Quarkslab

The report describes three new issues (excerpt from (15)):

- The availability of GOST 28147-89, a symmetric block cipher with a 64-bit block size, is an issue. This algorithm must not be used in this context.
- Compression libraries are outdated or poorly written. They must be updated or replaced.

- If the system is encrypted, the boot password (in UEFI mode) or its length (in legacy mode) could be retrieved by an attacker.

The Quarkslab security experts also criticized the immaturity of the UEFI loader, however, did not identify this as a concrete security issue.

According to the VeraCrypt developers, all newly discovered issues have been fixed in Version 1.19 of the software, released on 17 October 2016, which was the same date when Quarkslab released their audit report.

7.2.4 Google Project Zero

Google maintains a team of security analysts named Project Zero[88]. Project Zero resulted from part-time research activities of Google staff, which eventually received a fixed structure in the company in 2014.

The Project Zero team investigated TrueCrypt and VeraCrypt, and found two vulnerabilities, which allow local privilege elevation. The vulnerabilities were submitted to the National Vulnerability Database and received a reference accordingly:

CVE-2015-7359: Truecrypt 7 Derived Code/Windows: Incorrect Impersonation Token Handling EoP

A user could impersonate another user and therefore inspect and manipulate a mounted container of another user on the same system[89]. The vulnerability has been fixed in VeraCrypt 1.15.

CVE-2015-7358: Truecrypt 7 Derived Code/Windows: Drive Letter Symbolic Link Creation EoP

By abusing functions intended to create symbolic links for drive letters, it was possible to remap the main system drive and thereby getting access to it[90]. In this way, unprivileged users could spawn processes with system account privileges. The vulnerability has been fixed in VeraCrypt 1.15.

7.2.5 Other CVEs

CVE-2016-1281: TrueCrypt and VeraCrypt Windows installers allow arbitrary code execution with elevation of privilege

The TrueCrypt/VeraCrypt installer loads and executes the Dynamic Link Libraries (DLL) `USP10.dll`, `RichEd20.dll`, `NTMarta.dll` and `SRClient.dll`. If one of those DLLs is located inside the same directory as the installer, this DLL is loaded from this directory. In case the TrueCrypt/VeraCrypt installer is downloaded with a browser, it may be stored in the user's `Downloads` directory. Per social engineering or drive-by download, an attacker can place a malicious DLL inside the same directory, which is then executed once the installer is executed. The installer runs with administrator privileges and therefore the DLLs as well[91].

The vulnerability has been fixed in VeraCrypt 1.17-BETA with commit `5872be28`[92] and `7a15ff20`[93].

CVE-2019-1010208: Minor information disclosure of kernel stack due to buffer overflow

A buffer overflow in the VeraCrypt NT Driver leads to a minor information disclosure of the kernel stack. The vulnerability has been fixed in VeraCrypt 1.23-Hotfix-1 with commit `f30f9339`[94].

[88] Background information on Project Zero is available here. https://googleprojectzero.blogspot.com/2014/07/announcing-project-zero.html
[89] https://bugs.chromium.org/p/project-zero/issues/detail?id=537
[90] https://bugs.chromium.org/p/project-zero/issues/detail?id=538
[91] https://seclists.org/fulldisclosure/2016/Jan/22
[92] commit `5872be28a243acb3b5aafdf13248e07d30471893`
[93] commit `7a15ff2083d75cdfe343de154715442dce635492`
[94] commit `f30f9339c9a0b9bbcc6f5ad38804af39db1f479e`

7.2.6 BSI Security Analysis of TrueCrypt

In 2015, BSI and a research team at Fraunhofer SIT performed a security review of TrueCrypt (53). The researchers reviewed the results from both OCAP security reviews (cp. 7.2.1 and 7.2.2), evaluated the cryptographic algorithms against reference implementations, executed automatic code analyses of the source code, and reviewed the architecture, threat model, as well as the code quality of TrueCrypt. In addition to already known weaknesses from other security audits, the researchers found a weakness in how random numbers are generated by TrueCrypt under Linux. Because entropy pools were instrumentalized wrongly, under particular circumstances, for instance in auto-deployment scenarios, TrueCrypt might create weak keys with insufficient entropy.

7.2.7 Summary of all Reported Security Issues

Overall, from the previously described security audits 38 security vulnerabilities and weaknesses were reported. An overview is given in Table 10.

The table uses short references to refer to the sources where the findings were reported. "OCAP1" and "OCAP2" refer to the two OCAP security audits (59) (60), "Quarkslab" refers to Quarkslab's security assessment of VeraCrypt 1.18 (15), "Project Zero" refers to findings reported by Google Project Zero, and "BSI" refers to BSI's security analysis of TrueCrypt.

We researched the status of the findings by reviewing the development history of VeraCrypt at GitHub and SourceForge, but also taking into consideration the re-evaluation Quarkslab performed. In Table 10, the status "Patched" marks findings addressed properly with a security patch, while the status "Not patched" indicates that no such security fix has been applied yet. In one case, the cryptographic function considered insecure had been partially removed.

Table 10 Overview of previous findings with reference to the report and patch-status.

Report-Ref.	*Description*	*Status*
OCAP1	Weak Volume Header key derivation algorithm	Patched
OCAP1	Sensitive information might be paged out from kernel stacks	Not patched
OCAP1	Multiple issues in the bootloader decompressor	Patched
OCAP1	Windows kernel driver uses memset() to clear sensitive data	Patched
OCAP1	TC_IOCTL_GET_SYSTEM_DRIVE_DUMP _CONFIG kernel pointer disclosure	Patched
OCAP1	IOCTL_DISK_VERIFY integer overflow	Not patched
OCAP1	TC_IOCTL_OPEN_TEST multiple issues	Patched
OCAP1	MainThreadProc() integer overflow	Patched
OCAP1	MountVolume() device check bypass	Patched
OCAP1	GetWipePassCount()/WipeBuffer() can cause BSOD	Patched
OCAP1	EncryptDataUnits() lacks error handling	Not patched
OCAP2	CryptAcquireContextmay silently fail in unusual scenarios	Patched
OCAP2	AES implementation susceptible to cache-timing attacks	Not patched

Report-Ref.	*Description*	*Status*
OCAP2	Keyfile mixing is not cryptographically sound	Not patched
OCAP2	Unauthenticated ciphertext in volume headers	Not patched
Quarkslab	The length of the password can be computed when encryption is activated	Patched
Quarkslab	Out-of-date inflate and deflate	Patched
Quarkslab	XZip and XUnzip need to be completely re-written	Patched
Quarkslab	Integer overflow when computing the number of iterations for PBKDF2 when PIM is used	Patched
Quarkslab	PIN code on command line	Not patched
Quarkslab	GOST 28147-89 Must Be Removed from VeraCrypt	Cipher deprecated
Quarkslab	Lack of test vectors for newly added algorithms	Patched
Quarkslab	Input and output parameters are swapped in GOST Magma	Patched
Quarkslab	The PBKDF2 implementation does not fully comply with the standard	Not patched
Quarkslab	Bad coding practice in the HMAC-SHA512 Computation	Patched
Quarkslab	Unused parameters in key derivation sub-functions	Patched
Quarkslab	Random Byte Generators in DCS Should Be Improved	Not relevant / not checked further
Quarkslab	Keystrokes are not erased after authentication	Patched
Quarkslab	Sensitive data is not correctly erased	Patched
Quarkslab	Memory corruption can occur when the recovery disk is read	Patched
Quarkslab	A null pointer can be de-referenced when encrypted blocks are written	Patched
Quarkslab	Dead code in DcsInt	Patched
Quarkslab	The function reading the configuration may read inconsistent data	Patched
Quarkslab	Bad pointer check in EfiGetHandles	Patched
Quarkslab	Potential de-reference of a null pointer in the graphic library	Patched
Project Zero	CVE-2015-7359: Truecrypt 7 Derived Code/Windows: Incorrect Impersonation Token Handling EoP	Patched
Project Zero	CVE-2015-7358: Truecrypt 7 Derived Code/Windows: Drive Letter Symbolic Link Creation EoP	Patched
Other	CVE-2016-1281: TrueCrypt and VeraCrypt Windows installers allow arbitrary code execution with elevation of privilege	Patched
Other	CVE-2019-1010208: Minor information disclosure of kernel stack due to buffer overflow	Patched
BSI	Random Number Generator in Linux	Not patched

7.3 Development Documentation

Modern platforms for hosting software repositories have additional community features allowing developers and users to interact and communicate on development issues. We searched the publicly accessible part of VeraCrypt's community features in order to find reports and communication between users and developers related to the security of the software. We analyzed what the reported security issues actually are and how VeraCrypt developers handled them. As it is an unpopular proceeding to publish security issues on public channels, chances to find current or previous vulnerabilities in development forums are, however, rather limited.

In detail, we researched two online communities identified as used to discuss VeraCrypt-related development issues:

- SourceForge *Forums*[95] and
- GitHub *Issues*[96].

7.3.1 Reports in the SourceForge Forums

The SourceForge Forums are mainly used for discussing software features and functional issues of VeraCrypt. We found no security-related issues in these forums.

7.3.2 Reports in the GitHub Issue Tracker

While the SourceForge Forums are the main platform for communication between developers and users, VeraCrypt's source is hosted on GitHub. GitHub also provides an issue-reporting function that can be used to reach out to the developers in order to report errors and problems with the software. We analyzed and filtered all reported issues for security-related topics. The following issues were identified as pointing to potential security weaknesses in VeraCrypt.

"Volume creation security issue?"

The user *emkey08* mentioned that large blocks of a created volume are zeroed out instead of randomly initialized when using the command line tool of VeraCrypt[97]. The main developer *idrassi* explained to the reporting user the cause underlying this behavior: Quick formatting is applied when using the command line tool. To address the report, he implemented a switch to disable quick formatting to avoid this kind of information leakage.

"[SECURITY!] 'Random Pool Enrichment' window doesn't fill anymore when I move my mouse"

The user *secerrorreporter* complained about the random pool enrichment behavior. There is no answer to the issue[98]. We were unable to reproduce this issue and hence decided not to follow up on this report.

"macOS: information leak"

The user *dmitryd* informed the developers that a file contains the location of the last opened directory. He considers this as an information leak since the directory may contain container files and potential investigators may use this information against their suspects[99]. There is no answer to this issue from the developers.

[95] https://sourceforge.net/p/veracrypt/discussion/
[96] https://github.com/veracrypt/VeraCrypt/issues
[97] https://github.com/veracrypt/VeraCrypt/issues/365
[98] https://github.com/veracrypt/VeraCrypt/issues/342
[99] https://github.com/veracrypt/VeraCrypt/issues/290

“Linux: Build results in binary with executable stack”

The user *lachs0r* reported that the VeraCrypt binary declares the program execution stack as executable thereby omitting state-of-the-art protection against attacks such as certain buffer overflow attacks[100]. Within two weeks after the report was submitted, the VeraCrypt developers addressed the issue with a fix.

“Security issue: mouse entropy isn’t collected in the password changing dialog”

The user *ghost* complained about missing entropy collection when changing the encryption password[101]. As the developers were unable to reproduce the report, the issue was not further addressed.

[100] https://github.com/veracrypt/VeraCrypt/issues/146
[101] https://github.com/veracrypt/VeraCrypt/issues/203

8 Bibliography

1. *Advanced Encryption Standard.* 2001, FIPS PUB 197.

2. Anderson, Ross, Biham, Eli and Knudsen, Lars. Serpent: A Proposal for the Advanced Encryption Standard. *First Advanced Encryption Standard (AES) Conference.* 1998.

3. Schneier, Bruce, et al. Twofish: A 128-Bit Block Cipher. *First Advanced Encryption Standard (AES) Conference.* 1998.

4. *Recommendation for Block Cipher Modes of Operation: the XTS-AES Mode for Confidentiality on Storage Devices.* Dworkin, Morris. 1 2010, NIST Special Publication 800-38E.

5. *Cascade Ciphers: The Importance of Being First.* Maurer, Ueli M. and Massey, James L. 1993, Journal of Cryptology, Vol. 6, pp. 55–61.

6. Liskov, Moses, Rivest, Ronald L. and Wagner, David. Tweakable Block Ciphers. *Journal of Cryptology.* 2011, Vol. 24, pp. 588–613.

7. Ehrsam, William Friedrich, et al. *Message verification and transmission error detection by block chaining. 4074066* US Patent, 2 1978.

8. *RIPEMD-160: A strengthened version of RIPEMD.* Dobbertin, Hans, Bosselaers, Antoon and Preneel, Bart. [ed.] Dieter Gollmann. s.l. : Springer, 1996. Fast Software Encryption – FSE 1996. Vol. 1039, pp. 71–82.

9. *Secure Hash Standard.* 2015, FIPS PUB 180-4.

10. Barreto, Paulo S.L.M. and Rijmen, Vincent. The W HIRLPOOL Hashing Function. 2003.

11. *A Description of the Camellia Encryption Algorithm.* Matsui, Mitsuru, Nakajima, Junko and Moriai, Shiho. 2004, RFC, Vol. 3713.

12. *GOST R 34.12-2015: Block Cipher ``Kuznyechik''.* Dolmatov, Vasily. 2016, RFC, Vol. 7801.

13. *GOST R 34.11-2012: Hash Function.* Dolmatov, Vasily and Degtyarev, Alexey. 2013, RFC, Vol. 6986.

14. *GOST 28147-89: Encryption, Decryption, and Message Authentication Code (MAC) Algorithms.* Dolmatov, Vasily. 2010, RFC, Vol. 5830.

15. Bédrune, Jean-Baptiste, et al. *VeraCrypt 1.18 Security Assessment.* Quarkslab. 2016. Tech. rep.

16. *Software Generation of Practically Strong Random Numbers.* Gutmann, Peter. [ed.] Aviel D. Rubin. Berkeley : USENIX Association, 1998. Proceedings of the 7th USENIX Security Symposium.

17. Intel. *Intel Digital Random Number Generator (DRNG) - Software Implementation Guide.* 2018. Tech. Report.

18. *PKCS #5: Password-Based Cryptography Specification, Version 2.0.* Kaliski, Burt. 2000, RFC, Vol. 2898.

19. *HMAC: Keyed-Hashing for Message Authentication.* Krawczyk, Hugo, Bellare, Mihir and Canetti, Ran. 1997, RFC, Vol. 2104.

20. *PKCS #5: Password-Based Cryptography Specification, Version 2.1.* Moriarty, Kathleen M., Kaliski, Burt and Rusch, Andreas. 2017, RFC, Vol. 8018.

21. Federal Office for Information Security, Germany. *BSI – Technical Guideline: "Cryptographic Mechanisms: Recommendations and Key Lengths".* 2020. BSI TR-02102-1.

22. Bhatkar, Sandeep and DuVarney, Daniel C and Sekar, Ron. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. USENIX Security Symposium : s.n., 2003. Vol. 12, 2.

23. *Lest We Remember: Cold Boot Attacks on Encryption Keys.* Halderman, J. Alex, et al. [ed.] Paul C. van Oorschot. San : USENIX Association, 2008. Proceedings of the 17th USENIX Security Symposium.

24. *I/O Attacks in Intel PC-based Architectures and Countermeasures.* Sang, Fernand Lone, Nicomette, Vincent and Deswarte, Yves. s.l. : IEEE Computer Society, 2011. 2011 First SysSec Workshop. pp. 19-26.

25. *Information technology – Security techniques – Encryption algorithms – Part 3: Block ciphers.* 2010, ISO/IEC 18033-3:2010.

26. *Kryptographische Verfahren: Empfehlungen und Schlüssellängen.* 2019, BSI TR-02102-1.

27. *Information technology. Cryptographic data security. Block ciphers.* 2015, National Standard of the Russian Federation GOST R 34.12–2015.

28. *Reverse-Engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1.* Biryukov, Alex, Perrin, Léo and Udovenko, Aleksei. [ed.] Marc Fischlin and Jean-Sébastien Coron. s.l. : Springer, 2016. Advances in Cryptology – EUROCRYPT 2016. Vol. 9665, pp. 372–402.

29. *Exponential S-Boxes: a Link Between the S-Boxes of BelT and Kuznyechik/Streebog.* Perrin, Léo and Udovenko, Aleksei. 2 2017, IACR Transactions on Symmetric Cryptology, Vol. 2016, pp. 99-124.

30. *Transitioning the Use of Cryptographic Algorithms and Key Lengths.* Barker, Elaine and Roginsky, Allen. 2019, NIST Special Publication 800-131A Revision 2.

31. *IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices.* 1 2019, IEEE Std 1619-2018 (Revision of IEEE Std 1619-2007).

32. *Folklore, Practice and Theory of Robust Combiners.* Herzberg, Amir. s.l. : IOS Press, 4 2009, J. Comput. Secur., Vol. 17, pp. 159–189.

33. Green, Matthew. Multiple encryption. *A Few Thoughts on Cryptographic Engineering.* [Online] 2012. https://blog.cryptographyengineering.com/2012/02/02/multiple-encryption/.

34. *Information technology. Cryptographic data security. Hashing function.* 2012, National Standard of the Russian Federation GOST R 34.11–2012.

35. *IT Security techniques – Hash-functions – Part 3: Dedicated hash-functions.* 2018, ISO/IEC 10118-3:2018.

36. 1999-12324, I. S. T. Final report of European project IST-1999-12324: New European Schemes for Signatures, Integrity, and Encryption. *Final report of European project IST-1999-12324: New European Schemes for Signatures, Integrity, and Encryption.* 2004.

37. *Recommendation for Password-Based Key Derivation – Part 1: Storage Applications.* Turan, Meltem Sönmez, et al. 2010, NIST Special Publication 800-132.

38. Group, S. O. G.-I.S. Crypto Working. *SOG-IS Crypto Evaluation Scheme – Agreed Cryptographic Mechanisms.* 2018. Tech. rep.

39. *A Future-adaptive Password Scheme.* Provos, Niels and Mazières, David. s.l. : USENIX Association, 1999. Proceedings of the 1999 USENIX Annual Technical Conference. pp. 81–91.

40. *Stronger Key Derivation Via Sequential Memory-Hard Functions.* Percival, Colin. 2009.

41. Biryukov, Alex, Dinu, Daniel and Khovratovich, Dmitry. Argon2: the memory-hard function for password hashing and other applications. *Argon2: the memory-hard function for password hashing and other applications.* 2017.

42. Laboratories, R. S. A. *PKCS #11 v2.20: Cryptographic Token Interface Standard.* 2004. Tech. rep.

43. Ellison, Carl. Cryptographic Random Numbers. *Cryptographic Random Numbers.* 1995.

44. FIPS PUB 186-2: Digital Signature Standard (DSS), Federal Information Processing Standards Publication 186-2. *FIPS PUB 186-2: Digital Signature Standard (DSS), Federal Information Processing Standards Publication 186-2.* 2000.

45. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* Barker, Elaine and Kelsey, John. 6 2015, NIST Special Publication 800-90A Rev. 1.

46. Müller, Stephan. *Documentation and Analysis of the Linux Random Number Generator.* Federal Office for Information Security. 2019. Tech. rep.

47. Edge, Jake. Random numbers from CPU execution time jitter. *Linux Weekly News.* 2015.

48. Bernstein, Daniel J. Cache-timing attacks on AES. *Cache-timing attacks on AES.* 2004.

49. *Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012.* Smyshlyaev, Stanislav, et al. 2016, RFC, Vol. 7836.

50. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* Bassham III, Lawrence E., et al. 4 2010, NIST Special Publication 800-22 Rev. 1a.

51. Dodis, Yevgeniy, et al. How to Eat Your Entropy and Have it Too – Optimal Recovery Strategies for Compromised RNGs. *How to Eat Your Entropy and Have it Too – Optimal Recovery Strategies for Compromised RNGs.* 2014.

52. *An empirical study of goto in C code from GitHub repositories.* Nagappan, Meiyappan, et al. New York, NY, USA : s.n., 2015. Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. pp. 404–414.

53. Baluda, Mauro, et al. Security Analysis of TrueCrypt. 2015.

54. McConnell, Steve. *Code complete, 2nd Edition.* Redmond : Microsoft Press, 2004. ISBN: 9780735619678.

55. *Software complexity and software maintenance: A survey of empirical research.* Kemerer, Chris F. 12 01, 1995, Annals of Software Engineering, Vol. 1, pp. 1–22. ISSN: 1573-7489.

56. VeraCrypt - Documentation. *VeraCrypt - Documentation.* 2019.

57. *Examining PBKDF2 security margin—Case study of LUKS.* Visconti, Andrea, et al. 2019, Journal of Information Security and Applications, Vol. 46, pp. 296-306. ISSN: 2214-2126.

58. *Defeating Plausible Deniability of VeraCrypt Hidden Operating Systems.* Kedziora, Michal, Chow, Yang-Wai and Susilo, Willy. [ed.] Lynn Batten, et al. Singapore : Springer Singapore, 2017. Applications and Techniques in Information Security. pp. 3–13. ISBN: 978-981-10-5421-1.

59. Junestam, Andreas and Guigo, Nicolas. *Open Crypto Audit Project TrueCrypt Security Assessment.* iSEC Partners. 2014. Tech. rep.

60. Balducci, Alex, Devlin, Sean and Ritter, Tom. *Open Crypto Audit Project TrueCrypt Cryptographic Review.* NCC Group. 2015. Tech. rep.

61. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* Barker, Elaine and Kelsey, John. 6 2006, NIST Special Publication 800-90.

62. *A Bit-Slice Implementation of the Whirlpool Hash Function.* Scheibelhofer, Karl. [ed.] Masayuki Abe. Berlin : Springer Berlin Heidelberg, 2006. Topics in Cryptology – CT-RSA 2007. pp. 385–401. ISBN: 978-3-540-69328-4.

9 Appendix

9.1 Detailed Results of Tagging the Source Code Commits

9.1.1 Overview

This section details the results from tagging the single source code commits to VeraCrypt as described in 2.4. Only the tags relevant for this study are shown.

One recorded change means that a file had been altered with one single commit regardless of how many lines of code had actually been changed. Only files in the `src/` directory are considered.

Table 11 shows the ten most modified files. Overall, most commits affected files in the `Mount`, `Common`, `Format` and `Driver` directory of the source code.

Table 11: Number of changes per file over all commits.

Changes	*File*
188	`src/Mount/Mount.c`
185	`src/Common/Dlgcode.c`
101	`src/Format/Tcformat.c`
95	`src/Common/Tcdefs.h`
86	`src/Driver/Ntdriver.c`
78	`src/Format/Format.rc`
76	`src/Common/Dlgcode.h`
75	`src/Common/BootEncryption.cpp`
62	`src/Setup/Setup.c`
50	`src/Driver/DriveFilter.c`

9.1.2 Tag-Specific Statistics

9.1.2.1 Tag “Boot”

Table 12 shows all files with three and more modifications. Most commits affected files in the `Boot` and `Common` directory of the source code.

Table 12: Number of changes per file over all commits tagged with “boot”.

Changes	*File*
26	`src/Boot/EFI/DcsInt.efi`
26	`src/Boot/EFI/DcsCfg.efi`
22	`src/Common/BootEncryption.cpp`
20	`src/Boot/EFI/DcsRe.efi`
20	`src/Boot/EFI/DcsInt32.efi`
20	`src/Boot/EFI/DcsCfg32.efi`
16	`src/Mount/Mount.c`

Changes	*File*
16	src/Boot/EFI/DcsRe32.efi
15	src/Common/BootEncryption.h
13	src/Common/Dlgcode.c
12	src/Boot/Windows/BootCommon.h
11	src/Driver/DriveFilter.c
10	src/Common/Volumes.c
10	src/Common/Language.xml
10	src/Common/Dlgcode.h
10	src/Boot/Windows/BootMain.cpp
9	src/Format/Tcformat.c
9	src/Common/Pkcs5.c
8	src/Mount/Mount.rc
8	src/Common/Crypto.c
8	src/Boot/EFI/DcsBoot.efi
7	src/Mount/Resource.h
7	src/Common/Crypto.h
7	src/Common/Common.rc
7	src/Boot/EFI/DcsBoot32.efi
6	src/Common/Resource.h
6	src/Boot/EFI/DcsBml.efi
6	src/Boot/EFI/DcsBml32.efi
5	src/Common/Apidrvr.h
5	src/Boot/Windows/Makefile
5	src/Boot/EFI/LegacySpeaker.efi
5	src/Boot/EFI/LegacySpeaker32.efi
5	src/Boot/EFI/DcsInfo.efi
5	src/Boot/EFI/DcsInfo32.efi
4	src/Setup/Setup.c
4	src/Mount/MainCom.idl
4	src/Mount/MainCom.cpp
4	src/Crypto/Rmd160.c
4	src/Boot/Windows/BootSector.asm
4	src/Boot/Windows/BootDefs.h
4	src/Boot/Windows/BootConsoleIo.cpp

Changes	*File*
4	src/Boot/EFI/Readme.txt
3	src/Format/FormatCom.idl
3	src/Format/FormatCom.cpp
3	src/ExpandVolume/WinMain.cpp
3	src/Driver/Ntdriver.c
3	src/Crypto/Twofish.h
3	src/Crypto/Twofish.c
3	src/Common/Volumes.h
3	src/Common/Pkcs5.h
3	src/Common/BaseCom.h
3	src/Common/BaseCom.cpp
3	src/Boot/Windows/Decompressor.c
3	src/Boot/EFI/siglists/MicWinProPCA2011_2011-10-19_SigList_Serialization.bin.p7
3	src/Boot/EFI/siglists/MicWinProPCA2011_2011-10-19_SigList_Serialization.bin
3	src/Boot/EFI/siglists/MicWinProPCA2011_2011-10-19_SigList.bin
3	src/Boot/EFI/siglists/MicCorUEFCA2011_2011-06-27_SigList_Serialization.bin.p7
3	src/Boot/EFI/siglists/MicCorUEFCA2011_2011-06-27_SigList_Serialization.bin
3	src/Boot/EFI/siglists/MicCorUEFCA2011_2011-06-27_SigList.bin
3	src/Boot/EFI/siglists/DCS_sign_SigList_Serialization.bin.p7
3	src/Boot/EFI/siglists/DCS_sign_SigList_Serialization.bin
3	src/Boot/EFI/siglists/DCS_platform_SigList_Serialization.bin.p7
3	src/Boot/EFI/siglists/DCS_platform_SigList_Serialization.bin
3	src/Boot/EFI/siglists/DCS_key_exchange_SigList_Serialization.bin.p7
3	src/Boot/EFI/siglists/DCS_key_exchange_SigList_Serialization.bin
3	src/Boot/EFI/sb_set_siglists.ps1
3	src/Boot/EFI/certs/Readme.txt

9.1.2.2 Tag “Crypto”

Table 13 shows all files with three and more modifications. Most commits affected files in the `Common` and `Mount` directory of the source code. This may be caused by that fact that cryptography is implemented (or copied from other open source projects) once, and then used in different parts of VeraCrypt.

Table 13: Number of changes per file over all commits tagged with "crypto". Only files with a minimum of 3 commits are listed.

Changes	File
54	src/Mount/Mount.c
41	src/Format/Tcformat.c
40	src/Common/Dlgcode.c
27	src/Common/Language.xml
26	src/Common/BootEncryption.cpp
22	src/Common/Pkcs5.c
22	src/Common/Crypto.c
21	src/Common/Volumes.c
21	src/Common/Dlgcode.h
21	src/Common/Common.rc
18	src/Mount/Mount.rc
18	src/Driver/DriveFilter.c
16	src/Common/Crypto.h
16	src/Common/BootEncryption.h
14	src/Mount/Resource.h
14	src/Common/Resource.h
12	src/Volume/Cipher.cpp
12	src/Main/Forms/TrueCrypt.fbp
12	src/Main/Forms/Forms.cpp
12	src/ExpandVolume/WinMain.cpp
12	src/Common/Tests.c
12	src/Common/Password.c
11	src/Volume/Volume.make
11	src/Main/Forms/VolumePasswordPanel.cpp
11	src/Format/Format.rc
11	src/Crypto/cpu.c
10	src/Main/TextUserInterface.cpp
10	src/Main/GraphicUserInterface.cpp
10	src/Main/Forms/Forms.h
10	src/Crypto/Sources
10	src/Crypto/misc.h
10	src/Common/Random.c
9	src/Volume/Pkcs5Kdf.h

Changes	*File*
9	src/Volume/EncryptionTest.cpp
9	src/Main/Forms/VolumePasswordPanel.h
9	src/Main/Forms/VolumeCreationWizard.cpp
9	src/Main/CommandLineInterface.cpp
9	src/Common/Pkcs5.h
8	src/Volume/Cipher.h
8	src/Mount/MainCom.cpp
8	src/Mount/Favorites.cpp
8	src/Main/UserInterface.cpp
8	src/Main/Forms/ChangePasswordDialog.cpp
8	src/Format/InPlace.c
8	src/Crypto/Whirlpool.c
8	src/Crypto/cpu.h
8	src/Common/Password.h
8	src/Boot/Windows/BootCommon.h
7	src/Volume/VolumeLayout.cpp
7	src/Volume/Hash.cpp
7	src/Volume/EncryptionAlgorithm.cpp
7	src/Main/Forms/MountOptionsDialog.cpp
7	src/Driver/Ntdriver.c
7	src/Crypto/Sha2.c
7	src/Crypto/Crypto.vcxproj.filters
7	src/Crypto/Crypto.vcxproj
7	src/Common/Tcdefs.h
7	src/Common/Format.c
7	src/Common/Apidrvr.h
7	src/Boot/Windows/BootMain.cpp
6	src/Volume/Volume.cpp
6	src/Volume/Pkcs5Kdf.cpp
6	src/Setup/Setup.c
6	src/Main/Forms/VolumePimWizardPage.cpp
6	src/Format/Resource.h
6	src/ExpandVolume/ExpandVolume.rc
6	src/ExpandVolume/ExpandVolume.c

Changes	*File*
6	src/Driver/Ntvol.c
6	src/Crypto/Twofish.h
6	src/Crypto/Twofish.c
6	src/Common/Volumes.h
6	src/Common/Keyfiles.c
6	src/Boot/EFI/DcsInt.efi
6	src/Boot/EFI/DcsCfg.efi
5	src/Volume/EncryptionAlgorithm.h
5	src/Mount/Mount.h
5	src/Mount/MainCom.idl
5	src/Main/CommandLineInterface.h
5	src/ExpandVolume/DlgExpandVolume.cpp
5	src/Crypto/Streebog.c
5	src/Crypto/GostCipher.c
5	src/Crypto/Crypto.vcproj
5	src/Crypto/config.h
5	src/Common/SecurityToken.cpp
5	src/Common/BaseCom.cpp
5	src/Boot/Windows/BootDefs.h
5	src/Boot/EFI/DcsInt32.efi
5	src/Boot/EFI/DcsCfg32.efi
4	src/Volume/Volume.h
4	src/Volume/Hash.h
4	src/Setup/ComSetup.cpp
4	src/Main/Main.make
4	src/Main/Forms/VolumePasswordWizardPage.cpp
4	src/Main/Forms/MainFrame.cpp
4	src/Crypto/Serpent.c
4	src/Crypto/Rmd160.c
4	src/Crypto/Makefile.inc
4	src/Crypto/GostCipher.h
4	src/Crypto/Camellia.c
4	src/Crypto/Camellia_aesni_x64.S
4	src/Core/Unix/Linux/CoreLinux.cpp

Changes	*File*
4	src/Common/SecurityToken.h
4	src/Common/Random.h
4	src/Common/Keyfiles.h
4	src/Common/EncryptionThreadPool.c
4	src/Common/Cache.c
4	src/Boot/Windows/Makefile
4	src/Boot/EFI/DcsRe.efi
4	src/Boot/EFI/DcsRe32.efi
3	src/Mount/MainCom.h
3	src/Mount/Favorites.h
3	src/Makefile
3	src/Main/Forms/VolumePimWizardPage.h
3	src/Main/Forms/VolumePasswordWizardPage.h
3	src/Main/Forms/KeyfileGeneratorDialog.h
3	src/Main/Forms/KeyfileGeneratorDialog.cpp
3	src/Main/Forms/EncryptionOptionsWizardPage.cpp
3	src/Format/Tcformat.h
3	src/Format/InPlace.h
3	src/Format/FormatCom.idl
3	src/Format/FormatCom.cpp
3	src/Driver/Driver.vcxproj.filters
3	src/Driver/Driver.vcxproj
3	src/Crypto/Twofish_x86.S
3	src/Crypto/Twofish_x64.S
3	src/Crypto/sha512-x64-nayuki.S
3	src/Crypto/Camellia_x64.S
3	src/Crypto/Aes_hw_cpu.asm
3	src/Common/Common.h
3	src/Common/Cmdline.c
3	src/Common/Cache.h
3	src/Common/BaseCom.h
3	src/Boot/Windows/BootSector.asm

9.1.2.3 Tag “Driver”

Table 14 shows all files with three and more modifications. Most commits affected files in the Boot and Common directory of the source code.

Table 14: Number of changes per file over all commits tagged with “driver”. Only files with a minimum of 3 commits are listed.

Changes	*File*
39	`src/Driver/Ntdriver.c`
27	`src/Release/Setup Files/veracrypt-x64.sys`
27	`src/Release/Setup Files/veracrypt.sys`
16	`src/Driver/Ntvol.c`
13	`src/Mount/Mount.c`
13	`src/Common/Dlgcode.c`
12	`src/Driver/DriveFilter.c`
10	`src/Common/Apidrvr.h`
6	`src/Mount/Mount.rc`
6	`src/Driver/Ntdriver.h`
5	`src/Common/Tcdefs.h`
5	`src/Common/Language.xml`
5	`src/Common/Dlgcode.h`
4	`src/Setup/Setup.c`
4	`src/Mount/Resource.h`
4	`src/Format/InPlace.c`
4	`src/Common/Volumes.c`
4	`src/Common/Password.c`
4	`src/Common/BootEncryption.cpp`
3	`src/Format/Tcformat.c`
3	`src/ExpandVolume/ExpandVolume.c`
3	`src/Common/Common.rc`
3	`src/Common/BootEncryption.h`
2	`src/Setup/Setup.h`
2	`src/Setup/ComSetup.cpp`
2	`src/Release/Setup Files/veracrypt-x64.cat`
2	`src/Release/Setup Files/veracrypt.Inf`
2	`src/Release/Setup Files/veracrypt.cat`
2	`src/Mount/MainCom.idl`
2	`src/Driver/VolumeFilter.c`

Changes	*File*
2	`src/Driver/EncryptedIoQueue.c`
2	`src/Driver/DumpFilter.c`
2	`src/Driver/BuildDriver.cmd`
2	`src/Common/Resource.h`
2	`src/Common/Common.h`
2	`src/Common/Cache.c`
2	`src/Boot/Windows/BootMain.cpp`

9.1.2.4 Tag "Security"

Table 15 shows all files with three and more modifications. Most commits affected files in the `Common`, `Driver`, `Format` and `Mount` directory of the source code.

Table 15: Number of changes per file over all commits tagged with "security". Only files with a minimum of 3 commits are listed.

Changes	*File*
27	`src/Common/Dlgcode.c`
26	`src/Mount/Mount.c`
17	`src/Driver/Ntdriver.c`
16	`src/Format/Tcformat.c`
13	`src/Common/Dlgcode.h`
11	`src/Common/Language.xml`
11	`src/Common/BootEncryption.cpp`
10	`src/Mount/Mount.rc`
10	`src/Driver/DriveFilter.c`
9	`src/Setup/Setup.c`
8	`src/Mount/Resource.h`
8	`src/Common/Volumes.c`
8	`src/Common/Random.c`
7	`src/Common/Password.c`
7	`src/Common/Apidrvr.h`
6	`src/Mount/MainCom.cpp`
6	`src/Format/InPlace.c`
6	`src/ExpandVolume/WinMain.cpp`
6	`src/Common/BootEncryption.h`
5	`src/Common/Language.c`
4	`src/Setup/SelfExtract.c`

Changes	*File*
4	src/Driver/Ntvol.c
4	src/Driver/Ntdriver.h
4	src/Common/Tcdefs.h
4	src/Common/Registry.c
4	src/Common/Format.c
4	src/Common/Cmdline.c
4	src/Boot/Windows/BootSector.asm
4	src/Boot/Windows/BootMain.cpp
3	src/ExpandVolume/DlgExpandVolume.cpp
3	src/Driver/EncryptedIoQueue.c
3	src/Common/Keyfiles.c
3	src/Common/Exception.h
3	src/Common/Crypto.h
3	src/Common/Combo.c
3	src/Common/BaseCom.cpp
3	src/Boot/Windows/Decompressor.c
3	src/Boot/Windows/BootCommon.h

9.1.2.5 Tag “Static”

Table 16 shows all files with three and more modifications. Most commits affected files in the `Boot` and `Common` directory of the source code.

Table 16: Number of changes per file over all commits tagged with “static”. Only files with a minimum of 3 commits are listed.

Changes	*File*
39	src/Driver/Ntdriver.c
27	src/Release/Setup Files/veracrypt-x64.sys
27	src/Release/Setup Files/veracrypt.sys
16	src/Driver/Ntvol.c
13	src/Mount/Mount.c
13	src/Common/Dlgcode.c
12	src/Driver/DriveFilter.c
10	src/Common/Apidrvr.h
6	src/Mount/Mount.rc
6	src/Driver/Ntdriver.h
5	src/Common/Tcdefs.h

Changes	*File*
5	`src/Common/Language.xml`
5	`src/Common/Dlgcode.h`
4	`src/Setup/Setup.c`
4	`src/Mount/Resource.h`
4	`src/Format/InPlace.c`
4	`src/Common/Volumes.c`
4	`src/Common/Password.c`
4	`src/Common/BootEncryption.cpp`
3	`src/Format/Tcformat.c`
3	`src/ExpandVolume/ExpandVolume.c`
3	`src/Common/Common.rc`
3	`src/Common/BootEncryption.h`

9.2 Technical Information on Fuzzing Tests of the Header Parser for Container Files

9.2.1 Machine Used for Fuzzing

A server with the following stats has been used to perform the fuzzing evaluation:

- **CPU Model:** Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
- **CPU Cores:** 144
- **Memory:** 3094916 Mb

9.2.2 Corpus

Hidden volume

A container file with an outer and a hidden volume. The outer volume has a size of 25mb, the hidden volume has 15mb. Both are formatted with FAT32, SHA512 has been used as Hash. The UI has been used to create this volume as it is not possible to create a hidden volume in the text interface in a convenient way. Even in the UI, the process of creating a hidden volume crashes frequently with misleading error messages. All properties are listed in the following:

- **Encryption:** AES
- **Filesystem:** FAT
- **Outer size:** 25MB
- **Inner size:** 15MB
- **Hash:** SHA-512
- **Dynamic:** False
- **Hidden:** True

Corpus 01

A container file with the following properties:

- **Encryption:** AES
- **Filesystem:** FAT
- **Size:** 1MB
- **Hash:** SHA-512
- **Dynamic:** False
- **Hidden:** False

Corpus 02

A container file with the following properties:

- **Encryption:** AES
- **Filesystem:** NTFS
- **Size:** 5MB
- **Hash:** SHA-512
- **Dynamic:** False
- **Hidden:** False

Corpus 03

A container file with the following properties:

- **Encryption:** AES
- **Filesystem:** None
- **Size:** 5MB
- **Hash:** SHA-512
- **Dynamic:** True
- **Hidden:** False

Corpus 04

A container file with the following properties:

- **Encryption:** Twofish
- **Filesystem:** NTFS
- **Size:** 1GB
- **Hash:** SHA-512
- **Dynamic:** False

  Hidden: False

Corpus 05

A container file with the following properties:

- **Encryption:** Camellia
- **Filesystem:** FAT
- **Size:** 1MB

- **Hash:** SHA-512
- **Dynamic:** False
- **Hidden:** False

Corpus 06

A container file with the following properties:

- **Encryption:** Kuznyechik
- **Filesystem:** FAT
- **Size:** 1MB
- **Hash:** RIPEMD-160
- **Dynamic:** False
- **Hidden:** False

Corpus 07

A container file with the following properties:

- **Encryption:** Kuznyechik
- **Filesystem:** ExFAT
- **Size:** 1MB
- **Hash:** RIPEMD-160
- **Dynamic:** False
- **Hidden:** False

Corpus 08

A container file with the following properties:

- **Encryption:** Serpent
- **Filesystem:** ReFS
- **Size:** 1GB
- **Hash:** SHA-256
- **Dynamic:** False
- **Hidden:** False

Corpus 09

A container file with the following properties:

- **Encryption:** Serpent
- **Filesystem:** FAT
- **Size:** 1MB
- **Hash:** SHA-512
- **Dynamic:** False
- **Hidden:** False

9.3 Evaluation Methods in Related Work

This chapter lists the evaluation methods used for previous security assessments of TrueCrypt and VeraCrypt (cp. 7.2). Note that for most of these previous assessments, the respective reports provide only fragmentary information about the methods and procedures applied. Hence this list cannot and will not be exhaustive.

Automatic code scanning

A method where software tools automatically scan the application source code for potential weaknesses and vulnerabilities. The results need to be reviewed by a security expert to sort out false positives. This procedure was applied by Fraunhofer SIT for the security assessment of TrueCrypt (53).

Comparison with reference implementations

This method can be applied to evaluate implementations of cryptographic functions. The related source code part is extracted and fed with large amounts of input data. The resulting output stream is compared to an output stream generated by another implementation of the same cryptographic function, usually a well-known reference implementation. Both output streams must be equal. Fraunhofer SIT executed such tests for the security assessment of TrueCrypt (53).

Measuring computational efforts

A method to test the security of cryptographic functions, especially key derivation or key stretching functions, is to quantify the computational efforts required for a brute-force attack to succeed. Such investigations require to identify combinations of algorithm and hardware for their execution that a particularly efficient, for instance, brute-force algorithms running on Graphics Processing Units (GPU). The research work of Visconti et al. makes uses of this method to discuss the security of PBKDF2 (57).

Manual source code review

Security analysts manually review the software code. Due to its cost, this method is usually not applied exhaustively. Security analysts use to focus on key portions of the software code, for instance, code implementing cryptographic algorithms. Manual review of cryptographic source code was the particular focus of the OCAP cryptographic review (60).

Fuzzing

Fuzzing aims on feeding software interfaces with random data in order to provoke insecure software behavior. The easiest approach is to input a stream of random data into an interface. However, approaches that are more sophisticated involve selectively injecting random data into an input data structure that otherwise conforms to the interface protocol. Fuzzing of software interfaces was applied during the OCAP TrueCrypt security assessment (59).

Manual testing

During manual testing, a security expert creates manual test cases to test the behavior of the software for potential security weaknesses and behavior. For instance, the OCAP TrueCrypt security assessment involved manual testing of the bootloader and kernel driver (59).

Graybox testing

A hybrid method using insights gathered from source code reviews to create test cases for dynamic testing. It appears that some form of graybox testing was used by iSEC for the OCAP TrueCrypt security assessment (59). However, the work is only vaguely described as “source code assisted security assessment”.

9.4 Patch for the NIST STS Software

```
diff --git a/src/assess.c b/src/assess.c
index cf41d4f..d251a5a 100644
--- a/src/assess.c
+++ b/src/assess.c
@@ -108,8 +108,9 @@ main(int argc, char *argv[])
      fprintf(summary, "------------------------------------------------------
---------------------\n");
      postProcessResults(option);
      fclose(summary);
+     free(streamFile);

-     return 1;
+     return 0;
 }

 void
@@ -118,7 +119,6 @@ partitionResultFile(int numOfFiles, int numOfSequences, int
option, int testName
      int         i, k, m, j, start, end, num, numread;
      float c;
      FILE  **fp = (FILE **)calloc(numOfFiles+1, sizeof(FILE *));
-     int         *results = (int *)calloc(numOfFiles, sizeof(int *));
      char  *s[MAXFILESPERMITTEDFORPARTITION];
      char  resultsDir[200];

@@ -180,6 +180,7 @@ partitionResultFile(int numOfFiles, int numOfSequences, int
option, int testName
            }
      }
      fclose(fp[numOfFiles]);
+     free(fp);
      for ( i=0; i<MAXFILESPERMITTEDFORPARTITION; i++ )
            free(s[i]);

@@ -284,7 +285,7 @@ computeMetrics(char *s, int test)
 {
      int         j, pos, count, passCount, sampleSize, expCount,
proportion_threshold_min, proportion_threshold_max;
      int         freqPerBin[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
-     double      *A, *T, chi2, proportion, uniformity, p_hat, tmp;
+     double      *A, *T, chi2, uniformity, p_hat;
      float c;
      FILE  *fp;

diff --git a/src/discreteFourierTransform.c b/src/discreteFourierTransform.c
index 61b52c2..73c31d6 100644
--- a/src/discreteFourierTransform.c
+++ b/src/discreteFourierTransform.c
@@ -18,7 +18,7 @@ DiscreteFourierTransform(int n)
```

```
      double      p_value, upperBound, percentile, N_l, N_o, d, *m = NULL, *X =
NULL, *wsave = NULL;
      int         i, count, ifac[15];

-     if ( ((X = (double*) calloc(n,sizeof(double))) == NULL) ||
+     if ( ((X = (double*) calloc(n+1,sizeof(double))) == NULL) ||
             ((wsave = (double *)calloc(2*n,sizeof(double))) == NULL) ||
             ((m = (double*)calloc(n/2+1, sizeof(double))) == NULL) ) {
                  fprintf(stats[7],"\t\tUnable to allocate working arrays for
the DFT.\n");
diff --git a/src/randomExcursions.c b/src/randomExcursions.c
index 010b65d..1422897 100644
--- a/src/randomExcursions.c
+++ b/src/randomExcursions.c
@@ -105,7 +105,7 @@ RandomExcursions(int n)
                  x = stateX[i];
                  sum = 0.;
                  for ( k=0; k<6; k++ )
-                       sum += pow(nu[k][i] - J*pi[(int)fabs(x)][k], 2) /
(J*pi[(int)fabs(x)][k]);
+                       sum += pow(nu[k][i] - J*pi[(int)abs(x)][k], 2) /
(J*pi[(int)abs(x)][k]);
                  p_value = cephes_igamc(2.5, sum/2.0);

                  if ( isNegative(p_value) || isGreaterThanOne(p_value) )
diff --git a/src/randomExcursionsVariant.c b/src/randomExcursionsVariant.c
index 4a2b4de..048af75 100644
--- a/src/randomExcursionsVariant.c
+++ b/src/randomExcursionsVariant.c
@@ -53,7 +53,7 @@ RandomExcursionsVariant(int n)
                  for ( i=0; i<n; i++ )
                        if ( S_k[i] == x )
                              count++;
-                 p_value = erfc(fabs(count-J)/(sqrt(2.0*J*(4.0*fabs(x)-2))));
+                 p_value = erfc(abs(count-J)/(sqrt(2.0*J*(4.0*abs(x)-2))));

                  if ( isNegative(p_value) || isGreaterThanOne(p_value) )
                        fprintf(stats[TEST_RND_EXCURSION_VAR], "\t\t(b) WARNING:
P_VALUE IS OUT OF RANGE.\n");
diff --git a/Makefile b/Makefile
new file mode 100644
index 0000000..faf18d6
--- /dev/null
+++ b/Makefile
@@ -0,0 +1,51 @@
+CC = clang-9
+CFLAGS = -Wall -Wextra -MMD -Wshadow-all
+LDFLAGS = -static
+ifeq ($(DEBUG),1)
+CFLAGS += -Og -g3 -fsanitize=address
+LDFLAGS += -fsanitize=address
+else
```

```
+CFLAGS += -O3
+endif
+
+ROOTDIR = .
+SRCDIR = $(ROOTDIR)/src
+OBJDIR = $(ROOTDIR)/obj
+
+OBJ := \
+     src/assess.o \
+     src/frequency.o \
+     src/blockFrequency.o \
+     src/cusum.o \
+     src/runs.o \
+     src/longestRunOfOnes.o \
+     src/serial.o \
+     src/rank.o \
+     src/discreteFourierTransform.o \
+     src/nonOverlappingTemplateMatchings.o \
+     src/overlappingTemplateMatchings.o \
+     src/universal.o \
+     src/approximateEntropy.o \
+     src/randomExcursions.o \
+     src/randomExcursionsVariant.o \
+     src/linearComplexity.o \
+     src/dfft.o \
+     src/cephes.o \
+     src/matrix.o \
+     src/utilities.o \
+     src/generators.o \
+     src/genutils.o \
+     src/assess.o
+
+assess: $(OBJ)
+     $(CC) $(LDFLAGS) -o $@ $^ -lm
+
+%.o: %.c
+     $(CC) $(CFLAGS) -c -o $@ $<
+
+clean:
+     rm -f assess $(OBJ) $(OBJ:.o=.d)
+
+rebuild: clean assess
+
+-include $(OBJ:.o=.d)
diff --git a/makefile b/makefile
deleted file mode 100644
index 7ff2cd5..0000000
--- a/makefile
+++ /dev/null
@@ -1,94 +0,0 @@
-CC = /usr/bin/gcc
-GCCFLAGS = -c -Wall
```

```
-ROOTDIR = .
-SRCDIR = $(ROOTDIR)/src
-OBJDIR = $(ROOTDIR)/obj
-VPATH  = src:obj:include
-
-OBJ = $(OBJDIR)/assess.o $(OBJDIR)/frequency.o $(OBJDIR)/blockFrequency.o \
-       $(OBJDIR)/cusum.o $(OBJDIR)/runs.o $(OBJDIR)/longestRunOfOnes.o \
-       $(OBJDIR)/serial.o $(OBJDIR)/rank.o $(OBJDIR)/discreteFourierTransform.o
\
-       $(OBJDIR)/nonOverlappingTemplateMatchings.o \
-       $(OBJDIR)/overlappingTemplateMatchings.o $(OBJDIR)/universal.o \
-       $(OBJDIR)/approximateEntropy.o $(OBJDIR)/randomExcursions.o \
-       $(OBJDIR)/randomExcursionsVariant.o $(OBJDIR)/linearComplexity.o \
-       $(OBJDIR)/dfft.o $(OBJDIR)/cephes.o $(OBJDIR)/matrix.o \
-       $(OBJDIR)/utilities.o $(OBJDIR)/generators.o $(OBJDIR)/genutils.o
-
-assess: $(OBJ)
-     $(CC) -o $@ $(OBJ) -lm
-
-$(OBJDIR)/assess.o: $(SRCDIR)/assess.c defs.h decls.h utilities.h
-     $(CC) -o $@ -c $(SRCDIR)/assess.c
-
-$(OBJDIR)/frequency.o: $(SRCDIR)/frequency.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/frequency.c
-
-$(OBJDIR)/blockFrequency.o: $(SRCDIR)/blockFrequency.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/blockFrequency.c
-
-$(OBJDIR)/cusum.o: $(SRCDIR)/cusum.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/cusum.c
-
-$(OBJDIR)/runs.o: $(SRCDIR)/runs.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/runs.c
-
-$(OBJDIR)/longestRunOfOnes.o: $(SRCDIR)/longestRunOfOnes.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/longestRunOfOnes.c
-
-$(OBJDIR)/rank.o: $(SRCDIR)/rank.c defs.h externs.h matrix.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/rank.c
-
-$(OBJDIR)/discreteFourierTransform.o: $(SRCDIR)/discreteFourierTransform.c \
-        defs.h externs.h utilities.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/discreteFourierTransform.c
-
-$(OBJDIR)/nonOverlappingTemplateMatchings.o: \
-        $(SRCDIR)/nonOverlappingTemplateMatchings.c defs.h externs.h
utilities.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/nonOverlappingTemplateMatchings.c
-
-$(OBJDIR)/overlappingTemplateMatchings.o: \
-        $(SRCDIR)/overlappingTemplateMatchings.c defs.h externs.h utilities.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/overlappingTemplateMatchings.c
```

```
-
-$(OBJDIR)/universal.o: $(SRCDIR)/universal.c defs.h externs.h utilities.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/universal.c
-
-$(OBJDIR)/approximateEntropy.o: $(SRCDIR)/approximateEntropy.c defs.h externs.h
utilities.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/approximateEntropy.c
-
-$(OBJDIR)/randomExcursions.o: $(SRCDIR)/randomExcursions.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/randomExcursions.c
-
-$(OBJDIR)/randomExcursionsVariant.o: $(SRCDIR)/randomExcursionsVariant.c defs.h
externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/randomExcursionsVariant.c
-
-$(OBJDIR)/serial.o: $(SRCDIR)/serial.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/serial.c
-
-$(OBJDIR)/linearComplexity.o: $(SRCDIR)/linearComplexity.c defs.h externs.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/linearComplexity.c
-
-$(OBJDIR)/dfft.o: $(SRCDIR)/dfft.c
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/dfft.c
-
-$(OBJDIR)/matrix.o: $(SRCDIR)/matrix.c defs.h externs.h utilities.h matrix.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/matrix.c
-
-$(OBJDIR)/genutils.o: $(SRCDIR)/genutils.c config.h genutils.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/genutils.c
-
-$(OBJDIR)/cephes.o: $(SRCDIR)/cephes.c cephes.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/cephes.c
-
-$(OBJDIR)/utilities.o: $(SRCDIR)/utilities.c defs.h externs.h utilities.h
config.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/utilities.c
-
-$(OBJDIR)/generators.o: $(SRCDIR)/generators.c defs.h externs.h utilities.h \
-        config.h generators.h
-     $(CC) -o $@ $(GCCFLAGS) $(SRCDIR)/generators.c
-
-clean:
-     rm -f assess $(OBJDIR)/*.o
-
-rebuild: clean assess
```