EINSTUFUNG WURDE AUFGEHOBEN

~~VS – NUR FÜR DEN DIENSTGEBRAUCH~~

[redacted]

[redacted]

Untersuchung TrueCrypt

Arbeitspaket 5

Prüfungen

[redacted]

Historie

Version	Autor	Kommentar	Datum
0.1	[redacted]	initiale Dokumentenversion erstellt	06.10.10
0.2		AES erweitert	21.10.10
0.3		AES, XTS, SHA, RMD erweitert	04.11.10
0.4	[redacted]/TE	Review v. 0.3	04.11.10
0.5		Schlüsselmaterial erweitert	12.11.10
0.6		Schlüsselmaterial, Zufallsquellen	19.11.10
0.7		Schlüsselmaterial	26.11.10
0.8		Schlüsselmaterial Windows hinzugefügt	01.12.10
0.9		Zus. Kap. 4, Kommentare BSI	15.12.10

Inhaltsverzeichnis

1 Algorithmische Implementierungen (AP5.1)

TrueCrypt unterstützt in der vorliegenden Version 7.0a die folgenden Kryptografiekomponenten:

Symmetrische Chiffren

- AES mit 256 Bit Schlüsseln
- Blowfish mit 448 Bit Schlüsseln
- CAST-256 mit 128 Bit Schlüsseln
- 3DES/TDEA mit 168 Bit Schlüsseln
- TWOFISH mit 256 Bit Schlüsseln
- SERPENT mit 256 Bit Schlüsseln

Betriebsmodi

- XTS
- LRW
- CBC

Hash Funktionen

- RIPEMD mit 160 Bit Digest Länge
- SHA1 mit 160 Bit Digest Länge
- SHA2 mit 512 Bit Digest Länge
- Whirlpool mit 512 Bit Digest Länge

Der Schwerpunkt der Prüfung liegt bei AES-256, XTS und SHA-512. Zusätzlich wird RIPEMD-160 untersucht, da diese Hashfunktion derzeit noch für die Full-Disk-Encryption unter Windows benötigt wird. Andere Algorithmen werden nicht nicht behandelt.

1.1 Methodik und Vorgehensweise

Bei der Analyse wird zunächst die **Erfüllung der Standards** analysiert. Hierfür werden zuerst die für die jeweiligen Algorithmen notwendigen Standards und Dokumente identifiziert. Anschließend wird durch Analyse gemäß der im Folgenden aufgezeigten Punkte kontrolliert, ob der Quelltext die spezifizierten Algorithmen umsetzt. Hierbei wurden die Quelltexte zeilenweise analysiert und die Funktionalität jeder Zeile auf den Algorithmus im Standard bzw. einer alternativen Darstellung (s. [1], Absch. 7) abgebildet. Insbesondere wurden die funktionalen Blöcke der Algorithmen identifiziert (AES: SubBytes(), ShiftRows(), MixColumns(), AddRoundKey() und die Key Expansion; XTS: Aufteilung in Sektoren und AES-Blöcke, Tweakerzeugung, Multiplikation in GF(2^{128}), Verschlüsselung; SHA-512 und RIPEMD-160: Initialisierung, Vorverarbeitung, Padding, Hashberechnung). Bei

kombinierter Darstellungsweise eines oder mehrerer Schritte wurde außerdem die Korrektheit der Kombination verifiziert. Danach wurde kontrolliert, dass der Quelltext die Operationen korrekt implementiert. Zusätzlich wurde der vom Compiler nach Makroexpansion verarbeitete Quelltext untersucht sowie die vom Compiler erzeugte Objektdatei disassembliert und analysiert. Hierdurch wurde verifiziert, dass im Übersetzungs-Prozeß der Objektcode korrekt erstellt wird. Insbesondere wird bestätigt, dass keine zusätzlichen Makros im Übersetzungs-Prozeß definiert werden, durch die die Übersetzung des Quelltextes beeinflusst wird.

Wenn in den Spezifikationen optionale Funktionalität enthalten ist, wird beschrieben, ob TrueCrypt diese unterstützt. Wenn TrueCrypt zusätzliche Funktionalität umsetzt, wird diese dokumentiert. Abschließend wird die Korrektheit der Implementierung durch Tests mit Testvektoren bestätigt.

Im nächsten Schritt wird überprüft, ob Programmierfehler vorliegen. Hierbei wird überprüft, ob die Basisoperationen der Algorithmen fehlerfrei sind. Insbesondere wurden hier die Operationen für Multiplikation und Potenzierung in $GF(2^n)$ untersucht. Außerdem findet eine Überprüfung der Größe von Datentypen, Größe von Speicherarrays, Verwendung von Zeigern und Zugriffe auf Speicherarrays statt.

Als letzte Komponente der Analyse wird der Quelltext auf mögliche inhärente Angriffspunkte untersucht. Relevant sind hier z.B. Timing-Angriffe, die positions- und zeitabhängige Speicherzugriffslatenzen oder datenabhängige Ausführungszeiten von Opcodes ausnutzen können.

1.2 Voraussetzungen und Annahmen

Es wird davon ausgegangen, dass TrueCrypt ausschließlich mit AES im Betriebsmodus XTS-AES-256 verwendet wird, sowie als Hashfunktion nur RIPEMD-160 und SHA-2 mit 512 Bit Digest-Länge (SHA-512) zum Einsatz kommen.

Es wird davon ausgegangen, dass der Compiler den Quelltext gemäß den aktuellen Standards C99 [2] bzw. C++98 [3] übersetzt. Diese spezifizieren die Semantik und Syntax des Quelltextes, aber nicht die genaue Ausführungsumgebung. Da diese aber insbesondere für Seitenkanalangriffe relevant ist, sollte diese in die Analyse eingeschlossen werden. Daher wird hier außerdem die Ausführung auf x86 und AMD64 Systemen und die relevanten Compiler (GCC, MSVC) mit entsprechenden Compiler-Optionen eingeschlossen. Als Referenz für das Programmcode-Verhalten werden die Handbücher von Intel herangezogen [4]. Diese stellen ebenfalls die Referenz für die Assemblerimplementierungen dar.

Es wird davon ausgegangen, dass der Programmcode nicht von externen Komponenten, die auf dem gleichen System ausgeführt werden, anders als über die vorgesehenen Schnittstellen angesprochen und/oder manipuliert wird. Eine genauere Analyse dieser Anforderung ist in AP3, insbesondere in Abs. 5.5 „Angriffsbaum System (W5)" dargestellt.

1.3 Analyse AES

Die AES-Chiffre wird in TrueCrypt im Modus XTS-AES-256 genutzt (s. Abs. 4.4).

Für die Prüfung der AES Implementierung wird der AES-Standard [5] als Referenz herangezogen. In allen Implementierungen wird ausschließlich eine Schlüssellänge von 256 Bit unterstützt.

TrueCrypt nutzt verschiedene Implementierungen der AES Chiffre. Es wird eine generische, auf Geschwindigkeit optimierte, Implementierung in C bereitgestellt, sowie zusätzlich auf Geschwindigkeit optimierte Assemblerimplementationen für x86 und AMD64, sowohl mit als auch ohne Unterstützung für die AES Befehle in neuen Intel Prozessoren. Für Geschwindigkeitssteigerungen werden insbesondere Tabellen mit vorher berechneten Ergebnissen von Teilen der Algorithmen, gleichzeitige Verarbeitung großer Blöcke und Loop Unrolling genutzt. Außerdem steht eine speicheroptimierte C- und x86-Assembler-Implementierung bereit, die nur sparsam Tabellen und Loop Unrolling verwendet. Die API der Funktionen ist unabhängig von der Implementierung (Ausnahme: AES-NI Implementierung).

1.3.1 C-Implementierung

TrueCrypt nutzt eine leicht modifizierte Version der AES-Implementierung von Dr. Brian Gladman (http://gladman.plushost.co.uk/oldsite/AES/index.php). Dieser Code ist vielen Leuten bekannt und hat bereits entsprechend viel Analyse erfahren. Er findet sich in den Dateien `Aescrypt.c`, `Aeskey.c`, `Aesopt.h`, `Aes.h` und `Aestab.h`. Modifikationen für TrueCrypt beschränken sich funktional auf die Möglichkeit, die im folgenden angeführten Tests zu deaktivieren.

Zunächst ist anzumerken, dass TrueCrypt ausschließlich AES-256 Verschlüsselung anbietet. AES Varianten mit einer Schlüssellänge von *128 und 192 Bit werden nicht unterstützt (s. `Aes.h`).* Es existieren Angriffe auf AES-256 und AES-192 [6]. Als Blockgröße wird 128 Bit unterstützt, was dem AES Standard entspricht. Es werden beide Verarbeitungsrichtungen unterstützt (Ver- und Entschlüsselung). Um die Verwendung der Algorithmen während der Laufzeit zu kontrollieren, wird getestet, ob der Verschlüsselungskontext eine gültige Anzahl Rundenschlüssel enthält. Eine mögliche Fehlfunktion der Implementierung wird nicht erkannt. Um diese zu gewährleisten, müsste regelmäßig die korrekte Verarbeitung von Testvektoren überprüft werden (diese Funktionalität ist derzeit nicht in TrueCrypt enthalten), was einen hohen zusätzlichen Aufwand und damit Verlangsamung der Verarbeitungsgeschwindigkeit darstellt. Einführen solcher Test ist nicht sinnvoll, da:

1. Die Sicherheit wird für den Fall erhöht, dass ein Fehler in der Ausführung des Programms auf dem Prozessor die Chiffre schwächt, ohne das Programm anderweitig zu stören. Ein solcher Fehler ist sehr unwahrscheinlich.
2. Der zusätzliche Aufwand bremst die Ausführung des Programmes deutlich.

Bewertung: Die Einführung solcher Tests ist nicht sinnvoll. TrueCrypt in der vorliegenden Version sollte nicht dahingehend geändert werden.

Die C-Implementierung der Chiffre verwendet ausgiebig Textersetzung von Makros durch den C-Präprozessor. So kann durch Setzen von Makro-Optionen die Implementierungsart ausgewählt werden. Relevante Optionen betreffen insbesondere die Anzahl und Größe von Lookup-Tabellen und Loop Unrolling. Die Makroexpansion ist zum Teil sehr tief verschachtelt, was die Analyse des Quelltextes

erschwert (siehe Beispiel in Abschnitt 5). Um Gewissheit über der Korrektheit des übersetzten Codes zu erlangen, wurde zusätzlich zu den Implementierungsdateien der Quelltext analysiert, der nach dem Präprozessordurchlauf vom Compiler übersetzt wird.

Durch diese Verwendung von Makros könnte undefiniertes Verhalten erzeugt werden: Einer Variable wird mehrfach ein Wert zugewiesen, ohne dass die Zuweisungen durch Sequenzpunkte getrennt sind (`Aeskey.c`, Z. 524-527, reduzierte Darstellung in Text 1.1, Zuweisungen zu `ss[4]`). Da die Zuweisung hier immer der gleiche Wert ist, wird der Quelltext von gängigen Compilern wie erwartet übersetzt. Ob diese Nutzung tatsächlich undefiniert ist, ist Frage der Interpretation des Standards.

Empfehlung: Da der Code von gängigen Compilern korrekt übersetzt wird, ist eine Änderung nicht notwendig.

```
cx->ks[52] = ( t_im[0][(ss[4] = a|b|c|d)]>> 0 \
             ^ t_im[1][(ss[4] = a|b|c|d)]>> 8 \
             ^ t_im[2][(ss[4] = a|b|c|d)]>>16 \
             ^ t_im[3][(ss[4] = a|b|c|d)]>>24 );
```

Text 1.1: Evtl. undefiniertes Verhalten nach Makroexpansion

Die C-Implementierung hat vor allem 32 Bit Maschinen als Zielplattform. So nutzt die Polynommultiplikation `gf_mulx` eine Schreibweise, bei der 4 Bytes des AES-Zustandes in einer 32 Bit Variable gespeichert und jeweils im $GF(2^8)$ (mod $x^8+x^4+x^3+x+1$ = m(x) aus [5]) multipliziert werden (`Aesopt.h`, Z. 584).

Die C-Implementierung alloziert an keiner Stelle Speicher. Der Speicher für Klartext, Chiffretext und Schlüsselkontext inkl. aller Rundenschlüssel muss von den aufrufenden Codestellen bereitgestellt werden. Diese Aufrufe wurden in Abs. 2 untersucht.

Key Schedule

AES-256 nutzt 15 Rundenschlüssel, die jeweils aus 16 Byte bestehen und somit einen gesamten Speicherbedarf von 15*16 Byte = 240 Byte aufweisen. Diese Schlüssel werden gemäß der AES Key-Schedule zu den Rundenschlüsseln expandiert und linear hintereinander im Arbeitsspeicher abgelegt. Es werden Tabellen für die Beschleunigung der Substitution eingesetzt, sowie Loop Unrolling zur weiteren Erhöhung der Ausführungsgeschwindigkeit.

Für die Entschlüsselung werden die Rundenschlüssel in umgekehrter Reihenfolge gespeichert, d.h. die 16 Bytes des letzten Rundenschlüssels werden an erster Position gespeichert, die 16 Bytes des vorletzten Rundenschlüssels an zweiter Position etc. Dies ist die Reihenfolge, in der sie bei der Entschlüsselung benötigt werden. Diese Form der Speicherung ist notwendig für Verwendung von AES-Hardware in VIA Chips. Bei keiner in TrueCrypt verwendeten Implementierung ist

dies notwendig, somit könnte der zusätzliche Speicherplatz für die umgekehrte Speicherung der Schlüssel gespart werden. Außerdem ist die Verwendung von Schlüsseln in umgekehrter Reihenfolge für Entschlüsselung in der `Aes_x64.asm` Assembler-Implementierung langsamer (s. Kommentar in `Aes_x64.asm`, Z. 133f). Über den Einfluss dieser Speicherweise auf die Geschwindigkeit in anderen Implementierungen wird keine Aussage getroffen.

Die einzige datenabhängige Operation, die in der Key Schedule durchgeführt wird, ist das Nachschlagen einer vorher berechneten Rotations-Operation mit Substitution aus zwei 4096-Byte-Tabelle und das Nachschlagen der Rundenkonstante aus einer 40-Byte Tabelle (entsprechend `SubWord(RotWord(temp))` und `Rcon[i/Nk]` aus [5], Figure 11).

Problem: Die temporäre Variable `ss[]` enthält Schlüsselmaterial und wir vor Beenden der Funktionen `aes_encrypt_key256()` und `aes_decrypt_key256()` nicht sicher überschrieben.

Behebung: Sicheres Überschreiben der Variable `ss[]` hinzufügen. **Aufwand:** 1 Personentag.

Endianess

Endianess wird in `Common/Endian.h` abstrahiert. Hierdurch wird sicher gestellt, dass Volumes von Big-Endian-Systemen auch auf Little-Endian-Systemen geöffnet werden können. Relevant ist die Endianess beim Zugriff auf Datenblöcke, die größer sind als ein Byte. Im betrachteten Quelltext sind die Größen 2, 4 und 8 Byte relevant.

Die TrueCrypt AES Implementierung kann auf allen Systemen sowohl Big- als auch Little-Endian nutzen, ist jedoch bei Nutzung der nativen Endianess schneller. Daher wird in den betrachteten Kryptographie-Implementierungen von TrueCrypt immer PLATFORM_BYTE_ORDER genutzt (s. `Aesopt.h`, Z. 40). Außerdem ist die native Byte-Ordnung für die Assembler-Implementierungen des Algorithmus notwendig. Im AES Standard ist Endianess nicht spezifiziert.

Um portable Container zu erstellen muss die Endianess der Maschine somit an anderer Stelle berücksichtigt werden. Sie erfolgt in der Implementierung des Betriebsmodus (s. Abs. 1.6).

Schnelle Implementierungen

Es gibt verschiedene Assemblerimplementierungen des AES sowie Kombinationen von C- und Assembler-Code.

Um die Geschwindigkeit des Codes zu steigern wird im `Aescrypt.c` Code volles Loop-Unrolling der AES-Runden durchgeführt. Als Nachteil ist hier der größere Speicherbedarf des Codes sowohl in der Objektdatei als auch zur Laufzeit zu beachten.

Ebenso werden 16 Tabellen (4 für die normale Verschlüsselungsrunden, 4 für die letzte Verschlüsselungsrunde, ebenso für Entschlüsselung) mit je 256*4 Bytes für Ver- und Entschlüsselung sowie 4 Tabellen mit 256*4 Bytes für die

Entschlüsselungs-Key-Schedule genutzt, um die Geschwindigkeit zu erhöhen. Die Tabellen kombinieren die `SubBytes()`, `ShiftRows()` und `MixColumns()` Operationen und werden wie in [1] dargestellt generiert und genutzt. Alternativ hierzu wäre eine Umsetzung mit nur einer oder ganz ohne Beschleunigungstabelle möglich. Nachteil von großen Tabelle ist der erhöhte Speicherbedarf. Die Tabellen werden statisch im Data-Segment bereitgestellt.

Außerdem ist zu erwarten, dass eine Implementierung mit großen Tabellen anfälliger ist für einen Seitenkanalangriff mittels Zeitmessungen (Timing)[7].

1.3.2 x86 CPU Assemblerimplementierung

In der Datei `Aes_x86.asm` befindet sich eine Assemblerimplementierung der `Aescrypt.c` Funktionen für CPUs mit Intel x86-Befehlsunterstützung. Es wird für Windows und Linux die C Calling Convention (*cdecl*, siehe [8]) genutzt, d.h. Funktionsparameter werden auf dem Stack übergeben. Es wird tatsächlich der identische Algorithmus wie in der C-Implementierung durchgeführt (gleiche Tabellen zur Beschleunigung, Reverse Decipher Key Schedule). Für die Key Schedule wird der C Code aus `Aeskey.c` genutzt. Somit werden auch hier die Schlüssel im Speicher in der Reihenfolge abgelegt, wie sie von der Entschlüsselungsoperation benötigt werden. Eine Speicherung in umgekehrter Reihenfolge, d.h. in Reihenfolge der Verarbeitung bei Verschlüsselung, wäre allerdings schneller („`Aes_x86.asm`", Z. 108).

Empfehlung: Durch Ändern der Key Schedule für Entschlüsselung in normale Reihenfolge kann ohne Nebenwirkungen ein Geschwindigkeitsvorteil erreicht werden. **Aufwand:** 1 Personentag.

Des weiteren ist anzumerken, dass verschiedene, für die Verarbeitung unterschiedlicher Schlüssellängen notwendige, Sprunganweisungen im Code enthalten sind. Da TrueCrypt allerdings nur eine Schlüssellänge unterstützt, ist diese Möglichkeit überflüssig. Durch Entfernung dieser Sprünge könnte eine geringe Verbesserung der Ausführungszeit und Codegröße erreicht werden. Diese Änderung kann durch Entfernen der Sprunganweisungen erfolgen.

Empfehlung: Entfernen der Sprunganweisungen. **Aufwand:** 1 Personentag.

1.3.3 x64 CPU Assemblerimplementierung

In der Datei `Aes_x64.asm` befindet sich eine Assemblerimplementierung der `Aescrypt.c` Funktionen für CPUs mit AMD64-Befehlsunterstützung. Hier wird unter Windows die Microsoft x64 Calling Convention (s. [9]), unter Linux die AMD64 ABI Calling Convention (s. [10]) genutzt. Damit werden bei beiden Versionen die drei Funktionsparameter in Registern übergeben.

Die x64-Implementierung verwendet die Key Schedule aus `Aeskey.c`. Sie nutzt nicht die Tabellen aus der C-Implementierung, sondern erzeugt selbst 4 Tabellen mit einer Größe von jeweils 2048 Byte.

Es wird der gleiche Algorithmus genutzt wie in der C-Variante, allerdings mit nur 4

Tabellen für Ver- und Entschlüsselung (normale Runde, letzte Runde, vorwärts und rückwärts). Die Tabellen haben Einträge der Größe 8 Byte und sind so gefüllt, dass die Shift-Operation der Operation `ShiftRows()` durch einen Adressoffset beim Speicherzugriff erreicht wird.

1.3.4 AES-NI Assemblerimplementierung

Eine Implementierung des AES mit AES-NI Instruktionen [4][11] findet sich in `Aes_hw_cpu.(asm|h)` für CPUs mit x86- und AMD64-Befehlssätzen, die diese neuen Instruktionen unterstützen. Es wird die dem Betriebsystem entsprechende Calling Convention genutzt (*cdecl*, AMD64, MS x86, s.o.). In der TrueCrypt-Implementierung werden vier der neuen Befehle genutzt (s. Tabelle 1.1). Es werde ausschließlich 256-Bit-Schlüssel unterstützt. Das Interface zu der Ver- und Entschlüsselungsfunktion unterscheidet sich von dem bisher verwendeten: Es werden nur zwei Parameter genutzt, wobei die unverschlüsselten Daten mit den verschlüsselten Daten überschrieben werden. Zusätzlich wird eine Funktion `aes_hw_cpu_(en|de)crypt_32_blocks()` angeboten, die 32 aufeinander folgende 128 Bit-Blöcke mit gleichem Schlüssel mittels AES verschlüsselt. Zugriffe auf diese unterschiedlichen Funktionen werden in `Common/Crypto.c` und `Volume/Cipher.cpp` abstrahiert.

Befehl	Funktion
AESENC	Normale AES Verschlüsselungsrunde
AESENCLAST	Letzte AES Verschlüsselungsrunde
AESDEC	Normale AES Entschlüsselungsrunde
AESDECLAST	Letzte AES Entschlüsselungsrunde

Tabelle 1.1: Von TrueCrypt verwendete AES-NI Befehle [4]

Die Implementierung nutzt die Key Schedule aus dem C-Code. Aussagen bzgl. Ausführungszeit und Angreifbarkeit der Key Schedule können somit übernommen werden.

Für die Verarbeitung der Daten erfordert die AES Implementierung mit Hardwarebeschleunigung durch den Verzicht auf Lookup-Tabellen deutlich weniger Speicher als alle anderen Implementierungen. Weiterhin werden deutlich weniger Instruktionen durchgeführt, die außerdem keine datenabhängige Ausführungszeit besitzen. Damit bietet diese Implementierung eine konstante, datenunabhängige Ausführungszeit, und ist zudem deutlich schneller als alle anderen Implementierungen. Als Angriffspunkt bleibt somit nur die Key Schedule, da diese von der Standard-C-Implementierung übernommen wird.

1.3.5 Unterschiede bei Bootloader-Implementierung

Wenn TrueCrypt im Full-Disk-Encryption (FDE) Mode, d.h. Verschlüsselung der Festplatte inklusive der Betriebssystem-Partition, genutzt wird, ist die Implementierung des AES speicheroptimiert. Diese Entscheidung ist darin begründet, dass der von TrueCrypt genutzte Bootloader im Master Boot Record der

Festplatte gespeichert werden muss. Dieser bietet allerdings nur 62 Sektoren Speicherplatz für den Bootloader, was bei einer Sektorgröße von 512 Byte ca. 32 KB entspricht. Die Tabellen aus der geschwindigkeitsoptimierten Version würden beispielsweise bereits einen großen Teil des zur Verfügung stehenden Speichers verwenden. Loop-Unrolling würde den Speicherbedarf der AES-Funktionen ebenfalls erhöhen.

Die Bootloader-Implementierung wird nur während des Bootvorgangs genutzt. Im Verlauf des Bootens wird das Volume-Passwort an eine Speicherstelle geschrieben, von wo der Windows-Treiber das Passwort ausliest. Dann wird der Windows-Treiber für Lesen und Schreiben genutzt (s. AP4, Abs. 1.2.4).

Die speicheroptimierten Bootloader-Implementierungen finden sich in `AesSmall.c` und `AesSmall_x86.asm`.

1.3.6 C small Implementierung

Die Implementierung `AesSmall.c` wird bei FDE im Bootloader genutzt.

Diese Implementierung führt den AES Algorithmus wie im Standard vorgeschlagen durch. Zur Beschleunigung der GF(2^8)-Multiplikation in der MixColumns-Operation wird eine vorberechnete Multiplikationstabelle genutzt. Es werden nur Byte- und Word-Typen genutzt, denen ausschließlich zulässige Werte zugewiesen werden. Die inverse Chiffre ist analog implementiert.

Die kompakte Implementierung bringt außerdem eine eigene Implementierung der Key Schedule mit. Diese setzt den Standard mit Byte-Datentypen um. Der verwendete Algorithmus ist identisch zu Fig. 11 in [5].

Problem: Die temporären Variablen `tt`, `t0`, `t1`, `t2` und `t3` enthalten Schlüsselmaterial und werden vor Beenden der Funktion nicht sicher überschrieben.

Behebung: Sicheres Überschreiben der temporären Variablen hinzufügen. **Aufwand:** 1 Personentag.

Auch in dieser Implementierung sind Reste der Originalimplementierung von Dr. B. Gladman [1] vorhanden, mit denen weitere Schlüssellängen implementiert wurden. Durch Entfernung dieser Codefragmente könnte ein minimal verringerter Speicherbedarf und eine schnellere Ausführungszeit erreicht werden. Gerade hier, im speicherbeschränkten Boot-Modus, ist eine solche Optimierung sinnvoll.

Empfehlung: Entfernen der Verarbeitung von Schlüsseln der Länge 128 und 192 Bits. **Aufwand:** 1 Personentag.

1.3.7 x86 Assembler small Assemblerimplementierung

Die Implementierung `AesSmall_x86.asm` wird alternativ bei FDE im Bootloader genutzt. Die Implementierung führt den im Standard vorgeschlagenen Algorithmus identisch durch. Hier werden für die Ver- und Entschlüsselung jeweils 8 Tabellen mit 256 Byte Einträgen genutzt. Diese Tabellen werden von `Aestab.c` importiert. `Aestab.c` wird für die Boot-Implementierung so übersetzt, dass die Tabellen zur Laufzeit erzeugt werden. Analog wird für die inverse Operation vorgegangen.

Tabellen werden nur für die normalen Runden genutzt, die Lookups der letzten Runde werden aus den Werten dieser Tabellen berechnet. Den verwendeten Byte- und Wort-Registern werden ausschließlich zulässige Werte zugewiesen.

Diese Assembler-Implementierung bringt eine eigene Funktion für die Key Schedule mit. Sie nutzt nur eine Tabelle für die Speicherung der Rundenkonstante Rcon und der S-Box.

1.4 Übersetzung des AES-Codes

Bei den C-Implementierungen werden in bestimmten Bereichen verschiedene Compiler unterschieden. Speziell berücksichtigt werden die in Tabelle 1.2 dargestellen Compiler und Versionen. Hierdurch werden Fehler in alten Compilerversionen umgangen, sowie zusätzliche Geschwindigkeitsoptimierungen durchgeführt.

Compiler	Makro	Versionen
GCC	__GNUC__	
Microsoft C	_MSC_VER	<=800, >=1300
Watcom C	__WATCOMC__	>=1100

Tabelle 1.2: Compiler und Versionen, die in TrueCrypt unterschieden werden

Die folgenden Unterscheidungen werden gemacht:

- Microsoft C: Es werden verschiedene Optimierungen eingeschaltet, wenn der Microsoft C Compiler eingesetzt wird (`Aestab.h`, `Aescrypt.c`, `Aesopt.h`, `AesSmall.c`). Für Compiler bis zur Versionsnummer 800 werden statische Tabellen deaktiviert.
- GCC und Microsoft C: Diese Compiler werden bei `Aes_x64.asm` unterschieden, um die entsprechende Calling Convention für Windows bzw. Linux zu verwenden.
- Watcom C: Calling Convention für bestimmte Funktionen zu *cdecl* definiert.

Die Unterscheidungen haben keinen Einfluß auf die korrekte Funktionsweise der Algorithmen.

1.5 Bewertung AES

Um die Ergebnisse der in den vorangegangenen Abschnitten beschriebenen Untersuchung auf einen Blick darzustellen, folgt nun eine Zusammenfassung der relevanten Eigenschaften der AES Implementierungen.

1.5.1 Standardkonformität

Die untersuchten Implementierungen setzen den AES Algorithmus standardkonform um.

1.5.2 Geschwindigkeit

Die Implementierungen in TrueCrypt sind sehr schnell und für verschiedene Befehlssätze optimiert. Außerdem wird eine generische, aber ebenfalls schnelle C Implementierung bereit gestellt, die auf Architekturen genutzt werden kann, für die noch keine Assemblerimplementierung existiert.

1.5.3 Vertrauenswürdigkeit

Die Implementierung der kryptografischen Algorithmen wird als vertrauenswürdig erachtet. Die Algorithmen sind korrekt implementiert und Daten werden gemäß der entsprechenden Standards verarbeitet. Es sind keine Hintertüren eingebaut. Es wurden keine Datenlecks gefunden.

1.5.4 Sicherheitsaspekte

Bei der Untersuchung hinsichtlich der Sicherheit der AES-Implementierungen wurde festgestellt, dass die hier verwendeten Implementierungen, mit Ausnahme der auf Hardwarebeschleunigung basierenden AES-NI Implementierung, anfällig für Timing-Angriffe sind. Die Anfälligkeit für Timing-Angriffe existiert bei allen schnellen AES-Implementierungen in Software [7]. Die Praktikabilität solcher Angriffer wird hier nicht bewertet. Die AES-NI Implementierung ist nicht anfällig für diesen Angriff.

Darüber hinaus wurde die Implementierung auf Speicherallokations- und Speicherzugriffsfehler untersucht. Solche Fehler wurden nicht entdeckt.

1.5.5 Empfehlung

Die Implementierungen können mit überschaubaren Änderungen übernommen werden. Wir empfehlen, die Implementierung mit AES-NI zu wählen, da diese keine Angriffsmöglichkeiten bezüglich Timing aufweist und die höchste Geschwindigkeit bietet.

1.6 Analyse XTS Encryption Mode

Der XTS-AES-256 Modus wird für die Verarbeitung der Daten im verschlüsselten Volume (s. Abs. 4.4) und für die Verschlüsselung des Headers (s. Abs. 4.3.4) genutzt.

Der XTS Encryption Mode ist von der IEEE in [12] spezifiziert und vom NIST in [13] mit einer kleinen Änderung übernommen worden: Das NIST verlangt eine Beschränkung der Größe einer Dateneinheit („data unit", s. [12], Abs. 4.3.1) auf maximal 2^{20} AES Blöcke, i.e. 16 MiB. Eine Dateneinheit ist ein Datenblock, für den der gleiche Schlüssel und Kontext („key scope") genutzt wird. Der Schlüsselkontext beinhaltet einen Tweak Wert, die Größe einer Dateneinheit und die Datenlänge. Alle Dateneinheiten besitzen eine identische Größe. Spezifiziert sind XTS-AES-128 und XTS-AES-256 mit 256 respektive 512 Bit Schlüsseln. XTS ist abgesehen von den Legacy-Modi (LRW, CBC) der einzige von TrueCrypt unterstützte Verschlüsselungmodus. Die Legacy-Modi können bei der Erstellung eines neuen Volumes nicht ausgewählt werden und existieren nur, um mit alten TrueCrypt-

Versionen erstellte Volumes öffnen zu können.

1.6.1 Standardkonformität

Im XTS Standard wird ein Datenträger zweistufig in kleinere Verarbeitungseinheiten aufgeteilt. So besteht ein Datenträger aus bis zu 2^{128}-2 Blöcken der Größe 128 Bit, aufgeteilt in Dateneinheiten von bis zu $2^{20} \cdot 128$ Bit (s. Abbildung 1.1). Ciphertext Stealing wird nicht betrachtet, da TrueCrypt keine Unterstützung dafür bietet.

TrueCrypt unterstützt XTS-AES-256 mit Dateneinheiten, die ausschließlich eine Größe von 512 Byte haben (`Crypto/Crypto.h`, Z. 37, und s. Text 1.3). Dies ist konform zu den Standards. Ciphertext Stealing, wie im Standard optional definiert, ist somit nicht notwendig und nicht implementiert. 512 Byte entspricht der üblichen Sektorgröße von Festplatten (Anmerkung: Mitlerweile gibt es Festplatten mit einer Sektorgröße von 4096 Bytes. Diese können ohne Nachteile in Blöcke von 512 Byte Größe aufgeteilt werden).

```
if (length % BYTES_PER_XTS_BLOCK)
      TC_THROW_FATAL_EXCEPTION;
```

Text 1.3: EncryptionModeXTS.cpp, Z. 65f. „length" ist die Länge des Klartextes in Bytes

XTS nutzt zwei unterschiedliche AES-256-Schlüssel, einen Tweak und einen Offset. Der Tweak i ist eine fortlaufende Nummer, die beliebig initialisiert wird. Die beiden AES Einheiten benötigen die Schlüssel Key_1 und Key_2. Die AES-Einheiten werden entsprechend ihren Schlüsseln im Folgenden als AES_1 und AES_2 bezeichnet.

Der Wert T wird hier als *Whitening*-Wert bezeichnet.

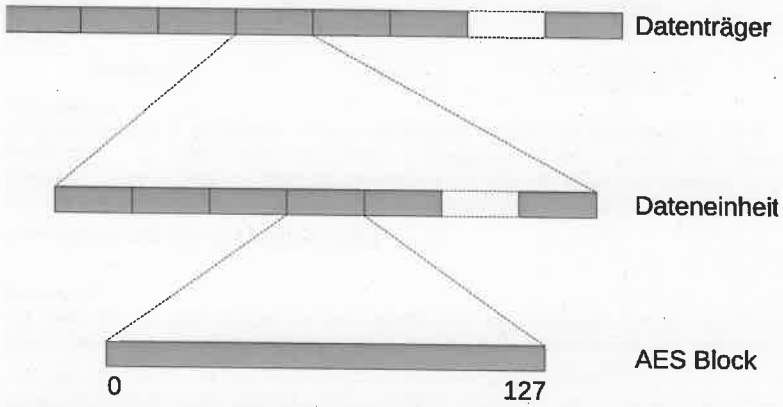

Abbildung 1.1: Aufteilung des Datenträgers im XTS Modus

Über die Anforderung des Standards, AES zu nutzen, hinaus unterstützt TrueCrypt

im XTS Modus auch Mehrfachverschlüsselung der Daten mit Serpent und (Twofish oder Blowfish) und AES sowie ausschließliche Nutzung anderer Chiffren ([12] und [13] verlangen und erlauben ausschließlich AES). Die Mehrfachverschlüsselung ersetzt beide AES Ver-/Entschlüsselungsinstanzen durch die alternative Chiffre.

1.6.2 Korrektheit der Implementierung

Der XTS Betriebsmodus ist gemäß dem im Standard vorgeschlagenen Algorithmus implementiert. j wird immer mit 0 initialisiert und die nachfolgende Multiplikationen mit α^j fortlaufend durch Shifts und XORs implementiert:

```
t = AES-enc(Key2, i)
for j = 0 to 512/8-1:
    MSB := t127
    t = t << 1
    t = t XOR MSB*135
next j
```

Dieser Algorithmus entspricht einer Multiplikation von t mit α^j für alle j von 0 bis 63 im GF(2^{128}). Da in TrueCrypt maximal eine Multiplikation mit α^{31} (512-Byte Dateneinheiten entsprechen 32 AES Blöcken) erforderlich ist, führt die Implementierung die Multiplikation im benötigten Rahmen korrekt durch. Die Funktionen zur Ver- und Entschlüsselung (`EncryptionModeXTS::EncryptBuffer/EncryptBufferXTS` bzw. `EncryptionModeXTS::DecryptBuffer/DecryptBufferXTS`) unterscheiden sich ausschließlich im Aufruf von AES_1, wo diese entsprechend der Ver- bzw. Entschlüsselungsrichtung aufgerufen wird. In beiden wird zunächst ein Array erstellt, in dem die notwendigen Whitening-Werte T (s. [12], Abs. 5.3.1, Figure 1) gespeichert werden. Anschließend werden diese Werte T mittels XOR mit dem Plaintext verknüpft, das Ergebnis PP dann verschlüsselt und die verschlüsselten Daten CC wiederum mit T verknüpft. Die Vorverarbeitung und das anschließende Übergeben von großen Blöcken an die AES-Routine ermöglicht die Nutzung einer für große Datenmengen optimierten AES-Ver- und Entschlüsselungsimplementierung.

Die Schnittstelle ist aufgrund der Verwendung von Byte-Arrays Endianess-unabhängig. Intern werden die Daten allerdings in 32- oder 64-Bit Variablen verarbeitet, weshalb die Endianess relevant ist. Sie wird entsprechend der Anforderung der AES-Implementierung in den Algorithmenblöcken angepasst.

Bewertung: Die Implementierung des XTS ist standardkonform.

1.6.3 Sicherheitsaspekte

Danach wurde der Algorithmus hinsichtlich Schwächen bzgl. Seitenkanalangriffen untersucht. Hierbei zeigt sich, dass die Potenzierung und Multiplikation in GF(2^{128}) solche Schwächen besitzt. Mögliche Schwächen in den verwendeten Chiffren werden hier nicht behandelt, setzen sich aber natürlich durch den XTS Modus hindurch fort.

Informationen, die sich hier extrahieren lassen, sind Informationen über die Position innerhalb von Sektoren, an denen Daten geschrieben werden. Der Code in Text 1.2 hat nach Übersetzung durch den Compiler typischerweise eine datenabhängige Laufzeit, da im „if"-Teil der Verzweigung mehr Code ausgeführt wird als im „else"-Zweig.

```
if (block >= startBlock)
{
      *whiteningValuesPtr64-- = *whiteningValuePtr64++;
      *whiteningValuesPtr64-- = *whiteningValuePtr64;
}
else
      whiteningValuePtr64++;
```

Text 1.2: Datenabhängige Ausführungszeit im XTS-Modus, abhängig von Schreibposition

Außerdem wird in TrueCrypt j für jeden Sektor mit 0 initialisiert, und somit können die Potenzen von α durch Shifts (α^0=0x1, α^1=0x2, α^2=0x4, α^3=0x8, etc), Multiplikation des MSB mit 135 und anschließende XOR-Addition erzeugt werden. Shifts innerhalb eines Registers sind in modernen CPUs (AMD ab K5, Intel ab Pentium) typischerweise nicht datenabhängig und somit nicht angreifbar durch Zeitmessung. Allerdings erfordert der XTS-Algorithmus ein Rotation über 128 Bit und damit über die Registergröße hinaus, so dass die Daten in zwei 64-Bit- oder vier 32-Bit-Registern gespeichert werden. Der Transfer des MSB über Registergrenzen hinaus wird wie in Text 1.3 dargestellt durchgeführt. Die Ausführungszeit von Verzweigungen ist nicht konstant, weshalb diese Passage keine konstante Ausführungszeit besitzt und somit eine Messung der Ausführungszeit Rückschlüsse auf den Whitening-Wert erlaubt.

```
finalCarry =
      (*whiteningValuePtr64 & 0x8000000000000000ULL) ?
      135 : 0;

*whiteningValuePtr64-- <<= 1;

if (*whiteningValuePtr64 & 0x8000000000000000ULL)
      *(whiteningValuePtr64 + 1) |= 1;

*whiteningValuePtr64-- <<= 1;
```

Text 1.3: Datenabhängige Laufzeit in der Potenzierung im XTS Modus

Bewertung: Der Unterschied der Laufzeit ist so gering, dass er von anderen Operationen währende der Ausführung verdeckt wird und praktisch nicht genutzt werden kann.

1.6.4 Geschwindigkeit

Für die Verschlüsselung eines einzelnen XTS-Blockes (128 Bit) werden zwei AES-Verschlüsselungen benötigt, für die Entschlüsselung eine AES-Ver- und eine AES-Entschlüsselung: Jeweils eine Verschlüsselung für die Erzeugung des Whitening-

Wertes, sowie eine Ver/Entschlüsselung für die Verarbeitung der mit den Whitening-Werten verknüpften Daten. Wenn eine ganze Dateneinheit von 512 Byte verschlüsselt wird, ist nur eine Berechnung von AES_2 notwendig.

Für die Ver-/Entschlüsselungen werden zwei Cipher-Instanzen an die Funktion übergeben. Somit ist es möglich, die Key Schedule für beide Instanzen genau einmal zu berechnen und nicht abwechselnd die Rundenschlüssel für AES_1 und AES_2 berechnen zu müssen.

Für Linux ist der XTS-Modus in der Datei `Volume/EncryptionModeXTS.cpp` umgesetzt. Hier wird, wie oben beschrieben, AES_1 mit großen Datenblöcken (bis zu 512 Byte, d.h. einer XTS data unit) aufgerufen. Alle Datentypten sind ausreichend groß, um standardkonform Volumes bis zu 2^64*16 Byte = 256 EiB verarbeiten zu können. Bei größeren Volumes muss für `startDataUnitNo` ein größerer Datentyp genutzt werden, sowie die darauf aufbauende Verarbeitung angepasst werden.

Unter Windows wird die Datei `Common/Xts.(c|h)` genutzt. Es existieren eine Funktion für parallele Ausführung der Verschlüsselung (`EncryptBufferXTSParallel`) sowie eine Version, in der blockparallele Ausführung der Verschlüsselung nicht unterstützt wird (`EncryptBufferXTSNonParallel`). So ist die erste, parallele Variante identisch zu der uriter Linux verwendeten Variante. Der serielle Algorithmus verschlüsselt die einzelnen AES-Blöcke nach Verknüpfung mit den Whitening-Werten direkt, ohne die Whitening-Werte vorher in einem Array zu speichern.

Bewertung: Der XTS-Modus ist in TrueCrypt effizient implementiert. Die Implementierung kann in der vorliegenden Form beibehalten werden.

1.7 Analyse Hashfunktionen

TrueCrypt bietet die Hashfunktionen RIPEMD-160, SHA-512, SHA-1 und Whirlpool an. Hashfunktionen werden in TrueCrypt in der Mischfunktion des Zufallszahlengenerator (s. Abs. 4.2.2) und der Schlüsselableitung mit PKCS#5-PBKDF2 genutzt (s. Abs. 4.3).

Untersucht werden die Hashfunktionen SHA-512 und RIPEMD-160. Derzeit unterstützt TrueCrypt ausschließlich RIPEMD-160 für den Full-Disk-Encrpytion Modus. Da RIPEMD-160 jedoch nicht mehr als sicher angesehen wird, untersuchen wir abschließend die Möglichkeit, diese Funktion durch SHA-512 zu ersetzen. Voraussetzung hierfür ist, dass der SHA-512-Code im TrueCrypt-Bootloader abgelegt werden kann.

1.7.1 SHA-512

SHA-512 ist in `Sha2.(h|c)` gemäß dem Standard [14] implementiert. Außer SHA-512 werden auch SHA-224, SHA-256, SHA-384 übersetzt und in die Objektdatei einbezogen. Diese werden von TrueCrypt allerdings nicht verwendet. Somit wäre hier eine Reduktion der Größe der Objektdatei durch Entfernen dieser Funktionen möglich. Dies ist insbesondere bei einer potenziellen Verwendung von SHA-512 im Boot-Modus sinnvoll. Die Implementierung basiert, wie auch die AES-

Implementierung, auf Code von Brian Gladman. Dieser Code ist nicht so extrem wie der AES Code auf Geschwindigkeit optimiert. Es wird nur beschränkt Loop-Unrolling durchgeführt (es werden 16 Schritte der 5 Runden innerhalb eines Schleifendurchlaufs berechnet, nicht die komplette Hashfunktion).

Der vorliegende Programmcode implementiert den Standard. Die hierzu durchgeführten Schritte entsprechen in der Reihenfolge größtenteils der Durchführung im Standard. Unterschiede sind im Abschnitt „Nachrichtenverarbeitung" aufgeführt. Es werden keine alternativen Darstellungsweisen wie bei der AES-Implementierung verwendet. Der Algorithmus ist in die Funktionen `sha512_begin()`, `sha512_hash()`, `sha512_compile()` und `sha512_end()` aufgeteilt. Diese Funktionen allozieren keinen Speicher. Sofern sie korrekt aufgerufen werden, greifen sie nur auf zulässige Speicherbereiche zu und halten den erlaubten Wertebereich von Variablen ein. Für die Daten wird durchgängig mit 64-Bit-Variablen gearbeitet.

Diese Implementierung unterstützt Nachrichten bis zu 2^{128}-1Bit Länge. Sollte eine Nachricht übergeben werden, die länger ist, wird der Fehler nicht erkannt. Dieses Problem ist praktisch allerdings nicht relevant.

Vorverarbeitung

In `sha512_begin()` wird der Hashwert mit den im Standard definierten Werten initialisiert (s. Abschnitt 5.3.5 in [14]). Der hier verwendete `memcpy`-Aufruf ist sicher, da die korrekte Größe des Datenfeldes als Parameter angegeben wird.

Nachrichtenverarbeitung

`sha512_hash()` erzeugt die Aufteilung der gepaddeten Eingangsnachricht in 1024-Bit-Blöcke, die anschließend mittels `sha512_compile()` in den Hashwert eingearbeitet werden. In diesem Teil erfolgt, sofern nötig, die Anpassung der Daten-Endianess an die Computer-Endianess.

`sha512_compile()` führt die Hashberechnung der einzelnen Blöcke durch. Hierbei werden die Schritte 1-3 des Abschnitts 6.4.2 des Standards kombiniert durchgeführt. Die acht Variablen a-h des Standards werden in einem Array `uint_64t v[8]` gespeichert. Das Umkopieren der Zwischenvariable a-h wird durch versetzten Zugriff auf die Variablen ersetzt. D.h. für `t = 0` entspricht `a = v[7]`, für `t = 1` gilt `a = v[6]`. Abschließend erfolgt die Aktualisierung des Hashwertes durch Addition des für dieses Datenpaket berechneten Resultats. Die Kombination dieser Schritte ist der einzige Unterschied zu den im Standard spezifizierten Schritten. Die in TrueCrypt durchgeführten Operationen sind äquivalent zum Standard.

Nachbearbeitung

Zum Schluß wird durch Aufruf von `sha512_end()` das Padding hinzugefügt (s. Abschnitt 5.1.2 in [14]) und der Hash-Wert der kompletten Nachricht extrahiert (s. Abschnitt 5.3.5 in [14]). Hier wird außerdem die Endianess korrigiert, falls die

ausführende Maschine nicht nativ die korrekte Endianess benutzt.

Bewertung: Die SHA512-Hashfunktion ist in TrueCrypt standardkonform implementiert. Die vorliegende Implementierung kann in der vorliegenden Form beibehalten werden.

1.7.2 RIPEMD-160

RIPEMD-160 ist in den Dateien `Rmd160.(c|h)` implementiert. Standardisiert ist der Algorithmus in [15]. Das Padding wurde aus MD4 übernommen [16].

Vorverarbeitung

In der Funktion `RMD160Init()` wird der RIPEMD-160-Kontext initialisiert.

Nachrichtenverarbeitung

In der Funktion `RMD160Update()` wird eine Nachricht in Blöcke aufgeteilt und an die Verarbeitung in den Hashwert an `RMD160Transform()` weiter gereicht. Wenn `RMD160Update()` für den Bootloader-Modus übersetzt wird, erfolgt die Zuweisung einer 32-Bit Variable an eine 16-Bit Variable („`unsigned __int32 lenArg`“ wird „`uint16 len`“ zugewiesen). Somit muss sicher gestellt werden, dass im Bootloader-Modus niemals Daten mit einer Länge von mehr als 2^{16}-1 Bit an RIPEMD-160 übergeben werden. Diese Grenze wird in TrueCrypt eingehalten.

Die Funktion `RMD160Transform()` führt den Kern des RIPEMD-160 Algorithmus durch. In der normalen Version werden 5 Runden komplett Loop-Unrolled ausgeführt. Die Rundenoperation $rol_{s(i)}$ wird mit einem Makro durchgeführt, das Verschieben der Variablen durch Variation des Zugriffsmuster auf die lokalen Variablen im Quelltext. Außerdem werden die beiden parallelen Rundenoperationen (einmal mit A, B, C, D, E; einmal mit A', B', C', D', E', s. [15]) hintereinander durchgeführt. Damit haben die ungestrichenen Variablen bereits ihren endgültigen Wert, bevor mit der Verarbeitung der gestrichenen Variablen begonnen wird. Anschließend werden die Variablen wie im Standard vorgesehen zum Hashwert kombiniert.

Wenn Rmd160.c speicherplatzoptimiert für den Bootloadermodus übersetzt wird, dann wird in der RIPEMD-160 Funktion `RMD160Update()` kein Loop-Unrolling genutzt. Die Funktion wird dann in 160 Schleifendurchläufen umgesetzt. Von diesen 160 Durchläufen verarbeiten die ersten 80 die ungestrichenen Variablen, die letzten 80 verarbeiten die gestrichenen Variablen. Anschließend werden die Variablen korrekt zum Hashwert kombiniert.

Nachverarbeitung

Die Nachricht wird in der Funktion `RMD160Pad()` durch Hinzufügen einer 1, Auffüllen der Nachricht mit Nullen auf eine Länge kongruent zu (448 mod 512) und Anfügen der Länge der Original-Nachricht in Bytes als 64-Bit Wert gepaddet. Dann wird der Hashwert mit den verbliebenen Daten und dem Pad aktualisiert. Mit der Funktion

`RMD160Final()` wird der Hashwert aus dem Hash-Kontext extrahiert und der Kontext zurückgesetzt.

Problem: Wenn der Programmcode mit Optimierung bzgl. Größe übersetzt wird (d.h. für Verwendung im Bootloader), wird nur das erste Byte des `PADDING`-Feldes initialisiert. Die weiteren 63 Byte sind nicht initialisert. Damit das Verhalten der Funktion `RMD160Pad()` gemäß der C-Standards [2][3] nicht definiert.

Behebung: Einfügen der korrekten Initialisierung der restlichen PADDING-Einträge.
Aufwand: 1 Personentag.

Bewertung: Die RIPEMD-160-Hashfunktion ist in TrueCrypt standardkonform implementiert. Aufgrund der mangelnden Sicherheit von RIPEMD-160 sollte sie auch für Full-Disk-Encryption durch den SHA-512-Algorithmus ersetzt werden. Siehe hierzu Abs. 1.7.3.

1.7.3 SHA-512 für Full Disk Encryption

SHA-512 wird derzeit nicht im Full Disk Encryption Modus unterstützt. Ein Grund hierfür ist die Größe des compilierten Codes: So ist die RIPEMD-160 Objektdatei bei Optimierung bzgl. Größe lediglich ~1700 Byte groß, die Sha-512 Objektdatei hingegen ist hier ~23500 Byte groß. Um SHA-512 im Full Disk Encryption Modus zu nutzen, ist somit eine signifikante Platzersparnis notwendig. Dies könnte zum Teil dadurch geschehen, dass kein Loop Unrolling genutzt wird und dass optionale Komponenten deaktiviert werden. Es ist wahrscheinlich nicht möglich, SHA-512 so klein zu implementieren wie RIPEMD-160, da allein die Tabellen mit Rundenkonstanten und Initialisierungswerten bei SHA-512 deutlich größer sind als bei RIPEMD-160.

Bewertung: Wir erachten die Verwendung von SHA-512 als Hashfunktion im Bootloader dennoch als möglich, da erwartet wird, dass der für eine auf speicheroptimierten Implementierung zusätzlich erforderliche Speicherplatz so gering ist, dass er an anderer Stelle im TrueCrypt-Bootloader eingespart werden kann.

Aufwand: Für eine auf Speicherverbrauch optimierte Implementierung der SHA-512 Funktion wird der Aufwand auf ca. 10 Personentage geschätzt.

1.8 Tests der Algorithmen

Um die verschiedenen Algorithmen testen zu können, wird eine Testframework bereitgestellt. Die Testbench führt die untersuchten Algorithmen mit vordefinierten Testvektoren durch und vergleicht die Ergebnisse mit den erwarteten Resultaten.

Die durchgeführten Tests treffen Aussagen darüber, ob die genannten kryptographischen Operationen korrekt funktionieren. Es wird keine Aussage darüber getroffen, ob die Funktionen innerhalb von TrueCrypt korrekt aufgerufen und angewandt werden.

1.8.1 Test AES

- Testvektoren aus [5]

- zufällige Daten mit zufälligen Schlüsseln
- Verschlüsselung
- Entschlüsselung
- Entschlüsselung vorher verschlüsselter Daten

1.8.2 Test XTS

- Testvektoren aus [12]
- zufällige Daten mit zufälligen Schlüsseln
- Verschlüsselung
- Entschlüsselung
- Entschlüsselung vorher verschlüsselter Daten
- Vergleich mit Referenzimplementierung

1.8.3 Test SHA-512

- Testvektoren aus [14]
- zufällige Daten

1.8.4 Test RIPEMD-160

- Testvektoren aus [15]
- zufällige Daten

2 Schutz von Schlüsselmaterial (AP5.2)

In diesem Kapitel wird der Schutz von Schlüsselmaterial in TrueCrypt untersucht. Effektiver Schutz und sichere Verarbeitung von Schlüsseln, Passwörtern und weiteren Parametern der Verschlüsselungsfunktionen stellt die Grundlage und damit die sensibelste Komponente für die Sicherheit der TrueCrypt-verschlüsselten Daten dar. Ein Angreifer darf keine Informationen erhalten, mit Hilfe derer die Komplexität eines Angriffes auf die verschlüsselten Daten reduziert werden kann.

2.1 Methodik und Vorgehensweise

Hierfür werden zunächst die Codestellen identifiziert, an denen Schlüsselmaterial, Initialisierungsvektoren (IV), Seed, Salt und Whitening-Werte verarbeitet werden. Aufgrund der Architektur wird hierbei zwischen der Windows- und Linux-Implementierung unterschieden, für die jeweils ein Datenflußgraph generiert wird.

Darauf folgend wird die Nutzung des Schlüsselmaterials untersucht. Zunächst ist die Verarbeitung während des normalen Betriebs relevant, d.h Datenhaltung im RAM. Insbesondere wird kontrolliert, ob das Schlüsselmaterial nur so lange wie notwendig gespeichert und bei Nichtverwendung sicher gelöscht und/oder überschrieben wird.

Dann wird die Speicherung während Suspend-To-RAM sowie Suspend-To-Disk untersucht. Hier ist relevant, ob das Schlüsselmaterial evtl. ungeschützt auf der Festplatte abgelegt wird und dadurch Rückstände entstehen, die ein Angreifer ausnutzen kann, um ein TrueCrypt-verschlüsseltes Volume anzugreifen.

Abschließend wird der Schutz des Schlüsselmaterials gegen Veränderungen überprüft. Es wird untersucht, ob die Daten mit Fehlerkorrektur gespeichert werden, was z.B. als Maßnahme gegen unbeabsichtigte Veränderungen durch Hardwaredefekte sinnvoll ist. Ein weiterer Schutz der Daten könnte durch kryptografische Methoden wie Signaturen oder HMAC geschehen, wodurch ein Schutz gegen beabsichtigte Veränderungen durch Angreifer erreicht werden könnte.

Zusätzlich wird an relevanten Stellen offensichtlicher Einfluss von Compiler-Optimierungen untersucht. So entfernen Compiler typischerweise Zuweisungen an Datenbereiche, die im weiteren Verlauf des Programms nicht mehr gelesen werden. Da sicheres Überschreiben von nicht mehr genutzten Daten aber voraussetzt, dass diese Schreibzugriffe trotzdem stattfinden, wird geprüft, ob der Code eine Schreiboperation sicherstellt.

2.2 Voraussetzungen und Annahmen

Es wird davon ausgegangen, dass der Programmcode nicht von externen Komponenten, die auf dem gleichen System ausgeführt werden, anders als über die vorgesehenen Schnittstellen angesprochen und/oder manipuliert wird. Insbesondere wird davon ausgegangen, dass die vom Betriebssystem bereitgestellten Schutzmechanismen wirksam sind und eine Umgehung nicht möglich ist. Sollte diese Anforderung verletzt werden, zeigt es eine Schwäche des Betriebssystems auf, nicht aber eine Schwäche der hier untersuchten Software. Eine genauere Analyse dieser

Anforderungen ist in AP3, insbesondere in Bas. 5.5 „Angriffsbaum System (W5)" dargestellt.

Allgemein ist anzumerken, dass über die Kommandozeile oder per GUI eingegebene Daten nicht vollständig der Kontrolle eines Benutzer-Programms unterliegen und es somit passieren kann, dass Daten in von den entsprechenden Bibliotheken oder dem Betriebsystemkernel angelegten Zwischenspeichern selbst bei sicherer Behandlung innerhalb TrueCrypts weiterhin existieren können. Beispielhaft sei hier für Linux die Befehls-Historie angegeben, welche die zuletzt in einem Terminal eingegebenen Befehle protokolliert. Eine solche Stelle für Windows ist die MFC-Bibliothek, die für die grafische Darstellung der Programmdialoge zuständig ist. Siehe hierzu auch AP3, Abs. 5.5 „Angriffsbaum System (W5)".

Es wird lediglich die Verarbeitung der Daten mit AES-256 im XTS-Modus behandelt.

2.3 Analysiertes Schlüsselmaterial

Es wird ausschließlich die Verarbeitung von Schlüsselmaterial kontrolliert. ***Schlüsselmaterial*** in diesem Sinne sind:

- Header-Schlüssel: Diese Schlüssel werden mit Hilfe der PKCS#5-Schlüsselableitungsfunktion PBKDF2 abgeleitet.
- Volume-Schlüssel K_V : Volumes werden mit einem Schlüssel verschlüsselt, der mit Material aus dem Zufallszahlengenerator erzeugt wird. Die Volume-Schlüssel werden mit den Header-Schlüsseln verschlüsselt auf der Festplatte abgelegt.
- Passwörter P_V : Dies sind vom Benutzer eingegebene Passwörter, aus denen mit der PKCS#5-PBKDF2 der Header-Schlüssel abgeleitet wird.
- Keyfiles: Dateien, die Schlüsselmaterial enthalten (s. [17]). Diese enthalten beliebige Daten und sind nicht durch Passworte geschützt. Die ersten 1024 kByte werden in das Schlüsselmaterial für die Entschlüsselung des Header eingepflegt.
- IV/Seed/Salt S : Diese Daten sind Initialisierungswerte für kryptographische Algorithmen. Insbesondere ist die Unvorhersagbarkeit von neuen Werten wichtig. Geheimhaltung dieser Daten spielt hingegen keine Rolle.
- Whitening-Werte T (s. [12], Abs. 5.3.1, Figure 1): Mit diesen Daten werden im XTS Modus die Klartexte verknüpft, um die Verschlüsselung identischer Klartexte zu unterschiedlichen Chiffretexten an anderen Positionen sicher zu stellen.

Die Bezeichnung ***„Sicherheitskritische Daten"*** bezieht sich in dieser Analyse ausschließlich auf die oben genannten Datenarten.

2.4 Analyse der Verarbeitung von Schlüsselmaterial

Um die Analyse sinnvoll zu ermöglichen und nachvollziehbar zu machen, wurde ein Graph aller Funktionen erstellt, die auf dem Aufrufpfad einer Funktion bis hin zur

AES Rundenschlüsselgenerierung liegen. Da die Architektur von TrueCrypt praktisch zweigeteilt ist, wurde ein Graph für die Linux-Version und ein zweiter für die Windows-Version erstellt. Diese Aufteilung spiegelt auch die Aufteilung für die Analyse wider. Die Graphen finden sich im Anhang.

Verarbeitung von Daten bedeutet hier, dass die Daten verändert, kopiert oder interpretiert werden. Übergeben von Referenzen auf Schlüsselmaterial an weitere Funktionen stellt keine Verarbeitung dar.

Sicherheit: Wenn die Verarbeitung der Daten ***sicher*** ist, bedeutet dies, dass es keine Hintertür in der Funktion gibt und keine Kopien oder anderweitige Daten erzeugt und einem Angreifer sichtbar gemacht werden, mit denen es möglich ist, den Schlüssel zu rekonstruieren.

Insbesondere wird Schlüsselmaterial nicht an Stellen kopiert, wo es nicht benötigt wird und es ist ausgeschlossen, dass eingelesenes Schlüsselmaterial durch schwaches ersetzt wird, wie z.B. einen immer gleichen Schlüssel.

2.5 Analyse Schlüsselverarbeitung Linux

Eine tabellarische Auflistung der Funktionen findet sich in Kap. 6. Wenn die Funktionsbezeichnungen nicht eindeutig sind, wird in diesem Dokument die Identifikationsnummer (s. Auflistung in Kap. 6) vorgestellt. Der alphabetische Funktions- und Klassenindex am Ende dieses Dokuments enthält die Windows- und Linux-Funktionen.

Makro burn()

Um sicheres Überschreiben von Daten im RAM in C-Code durchzuführen wird das Makro `burn()` genutzt. Unter Linux wird ein `volatile` deklarierter Zeiger auf die zu überschreibenden Speicherbereiche genutzt, um die Daten mit Nullen zu überschreiben.

Bewertung: Das Makro `burn()` überschreibt effektiv Speicherbereiche mit Nullen.

Klasse SecureBuffer

Diese Klasse stellt RAM-Speicher bereit, der bei Deallokation über `Memory::Erase()` per `memset()` mit Nullen überschrieben und dann erst freigegeben wird. Hiermit ist die Gefahr verbunden, dass der `memset()`-Aufruf vom Compiler im Zuge von Optimierungen entfernt wird. Beim von TrueCrypt verwendeten Übersetzungsprozeß ist dies allerdings nicht der Fall: Der Quelltext von `SecureBuffer` wird getrennt vom `Memory`-Quelltext übersetzt und die beiden haben keine weitergehenden Informationen über das interne Verhalten der jeweils anderen Übersetzungseinheit. Damit ist es dem Compiler nicht möglich, die Widerverwendung des Datenfeldes auszuschließen und damit kann der `memset()`-Aufruf nicht entfernt werden.

Problem: In Zukunft könnten Fortschritte in der Programmoptimierung dazu führen, dass Compiler `memset()`-Aufrufe entfernen könnten. Dies kann z.B. durch globale

Optimierung erreicht werden.

Behebung: Empfehlenswert ist die Veränderung der Überschreib-Funktion dahingehend, dass der Compiler keine Möglichkeit hat, den Aufruf zu entfernen. Dies könnte durch Einsatz der `burn()`-Funktion geschehen. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

Annahme: Für die weitere Betrachtung wird davon ausgegangen, dass diese Veränderung durchgeführt wird und damit die Klasse `SecureBuffer` sicher ist.

Klasse VolumePassword

`VolumePassword` ist ein Objekt, in dem ein Passwort mit zusätzlichen Informationen gespeichert wird. Das Passwort selbst wird schließlich in einem `SecureBuffer` gespeichert. Beim Zerstören des Objekts wird das Passwort somit überschrieben.

Problem: In `VolumePassword::Set()` wird eine Kopie des Passworts erzeugt, die in dem Byte-Array `byte passwordBuf[]` gespeichert und nach Benutzung nicht überschrieben wird (gilt für alle `Set()`-Varianten).

Behebung: Sicheres Überschreiben des `passwordBuf[]` Arrays im `VolumePassword`-Objekt. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

Annahme: Für die weiter Betrachtung wird davon ausgegangen, dass diese Veränderung durchgeführt wird und die Klasse `VolumePassword` sicher ist.

Klasse VolumePasswordPanel

Die Klasse `VolumePasswordPanel` wird genutzt, um ein Passwort in ein `VolumePassword` Objekt sowie eine Liste von Keyfiles in eine Liste einzulesen. Es ist ein GUI-Element, dass in der Regel in ein anderes Fenster eingebettet wird. Es werden ein Passwort und evtl. eine Bestätigung des Passworts abgefragt, die jeweils eine Instanz des `wxTextCtrl` sind (dies ist ein von der GUI-Bibliothek angebotener Datentyp zur Speicherung und Darstellung von Strings, s. [18]). Die beiden `wxTextCtrl` werden mit „X"en überschrieben, wenn `VolumePasswordPanel` zerstört wird (Anmerkung: Löschen mittels `burn()` ist aus programmtechnischen Gründen nicht möglich). Damit ist diese Klasse sicher.

Von den Keyfiles wird hier nur der Ort eingelesen, es werden keine weiteren Informationen extrahiert. Damit ist diese Verarbeitung hier unkritisch zu beurteilen.

Bewertung: Diese Klasse verarbeitet Schlüsselmaterial sicher.

Klasse MountOptionsDialog

`MountOptionsDialog` ist ein Objekt, in dem mittels eines `VolumePasswordPanel` ein Passwort für ein Volume und evtl. zusätzlich ein Passwort für ein geschütztes verstecktes Volume in das System eingelesen wird. Es werden lediglich Referenzen auf Schlüsselmaterial an weitere Funktionen übergeben.

Bewertung: Diese Klasse verarbeitet kein Schlüsselmaterial.

Klasse VolumePasswordWizardPage

Die `VolumePasswordWizardPage` liest mittels eines `VolumePasswordPanel` ein Passwort in ein `VolumePassword` ein. Die Daten werden hier nicht verarbeitet.

Bewertung: Diese Klasse verarbeitet kein Schlüsselmaterial.

Keyfiles

Keyfiles werden genutzt, um Schlüsselmaterial aus Dateien für TrueCrypt bereit zu stellen. Hierfür können zum einen gewöhnliche Dateien genutzt werden. Von solchen Dateien werden die ersten 1024 kByte zum Passwort hinzugefügt.

Zusätzlich kann TrueCrypt Keyfiles erstellen. Sie werden von der Funktion `CoreBase::CreateKeyfile()` erstellt. Diese Funktion wird aus `TextUserInterface::CreateKeyfile()` und `KeyfileGeneratorDialog::OnGenerateButtonClick()` aufgerufen und dort mit Zufallsdaten aus dem TrueCrypt-Zufallszahlengenerator gefüllt.

Über die Keyfiles erfolgt außerdem die Einbindung von Daten auf Security Tokens und Smartcards.

Keyfile::ApplyListToPassword()

Diese Funktion ist ein Wrapper, mit dem der Inhalt einer Liste von Keyfiles zu einem Passwort hinzugefügt wird. Die Keyfiles werden nacheinander an die `Keyfile::Apply()`-Funktion übergeben. Das Ergebnis ist der Header-Schlüssel, der in einem `SecureBuffer` gespeichert wird.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Keyfile::Apply()

In dieser Funktion werden die ersten 1024 kBytes eines Keyfiles eingelesen und der CRC32-Wert der einzelnen Bytes wird gemäß Abs. 4.3.2 zum Passwort hinzugefügt. Da ein Keyfile typischerweise länger ist als das Passwort, wird die Additionsposition im Passwort wieder auf den Anfang gesetzt, wenn das Ende erreicht ist. Die Addition erfolgt Byte-weise, somit (mod 2^8).

Diese Art der Anwendung von Keyfiles stellt sicher, dass es keinen Unterschied macht, in welcher Reihenfolge die Keyfiles auf das Passwort angewandt werden.

An dieser Stelle erfolgt auch die Integration von Dateien, die auf Security Tokens und Smartcards gespeichert sind.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher. Der verwendete Algorithmus stellt sicher, dass die Entropie aus den Keyfiles zum Schlüssel hinzugefügt wird.

Crc32::Crc32()

Diese Funktion speichert Daten in einer 32-Bit Variable, die nicht geschützt wird.

Problem: Schlüsselmaterial wird eventuell nicht überschrieben.

Behebung: Überschreiben der Daten im Destruktor hinzufügen. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das angesprochene Problem behoben wird.

TextUserInterface::AskPassword()

In `AskPassword()` werden Passwörter zunächst in einen `wxString passwordStr` eingelesen und von hier in ein `wchar_t passwordBuf[]` Array kopiert. Beim Kopieren wird jeder in `passwordStr` befindliche Buchstabe des Passworts mit „X" überschrieben, was effektiv das Passwort löscht.

Problem: Die Länge des Passwortes bleibt auf dem Heap im Speicher erkenntlich und lässt sich durch Zählen der „X" rekonstruieren.

Behebung: Änderung des Codes dahingehend, dass für das Passwort ein String konstanter Länge reserviert wird und immer dieser komplette für das Passwort zur Verfügung stehende Speicher mit „X" überschrieben wird. Mit dieser Änderung wäre diese Funktion sicher. **Aufwand**: 1 Personentag.

Schließlich wird das Passwort in ein `VolumePassword` gespeichert, welches dann an die aufrufende Funktion zurück gegeben wird. Wenn eine Verifizierung der Passworteingabe, d.h. eine Bestätigung des Passworts durch zweimalige Eingaben verlangt wird, wird das Passwort zusätzlich in einen zweiten `VolumePassword` gespeichert.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das angesprochene Problem behoben wird.

Pkcs5Kdf::DeriveKey()

In dieser Funktion wird der Header-Schlüssel mit der gewünschten Hashfunktion gemäß PBKDF2 aus PKCS#5 [19] abgeleitet (Verwendung: s. Abs. 4.3.4 und Abs. 4.3.5). Diese Funktion ist ein Prototyp, der von `Pkcs5HmacSha512::DeriveKey()` und `Pkcs5HmacRipemd160::DeriveKey()` implementiert wird.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Pkcs5HmacSha512::DeriveKey()

Dies ist der Eintrittspunkt der KDF mit Verwendung der HMAC-Funktion [20] mit SHA-512 Hashfunktion. Es werden nur Schlüsselmaterial-Referenzen an Unterfunktionen übergeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Pkcs5Ripemd160::DeriveKey()

Dies ist der Eintrittspunkt der KDF mit Verwendung HMAC-Funktion [20] mit der RIPEMD-160 Hashfunktion. Es werden nur Schlüsselmaterial-Referenzen an Unterfunktionen übergeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Pkcs5Kdf::ValidateParameters()

Hier wird die Korrektheit der an die PKCS#5-PBKDF2 übergebenen Parameter überprüft.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

derive_key_sha512()

In dieser Funktion wird die PBKDF2-Funktion mit SHA512 durchgeführt (s. [19], Kap. 5.2). Sicherheitskritische Daten werden abschließend mittels `burn()` sicher überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

derive_key_ripemd160()

In dieser Funktion wird die PBKDF2-Funktion mit RIPEMD160 durchgeführt (s. [19], Kap. 5.2). Sicherheitskritische Daten werden abschließend mittels `burn()` sicher überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

derive_u_sha512()

In dieser Funktion wird die Funktion `F(P, S, c, i)` der PBKDF2 implementiert (s. [19], Kap. 5.2). Alle sicherheitsrelevanten Daten werden sicher mittels `burn()` überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

derive_u_ripemd160()

In dieser Funktion wird die Funktion `F(P, S, c, i)` der PBKDF2 implementiert (s. [19], Kap. 5.2). Alle sicherheitsrelevanten Daten werden sicher mittels `burn()`

überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

hmac_sha512()

Hier wird die HMAC-Funktion mit SHA-512 durchgeführt. Hierzu werden die bereits in Abschnitt 1.7.1 analysierten SHA-512 Funktionen aufgerufen. Die verwendeten Daten werden abschließend sicher mit `burn()` überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

hmac_ripemd160()

Hier wird die HMAC-Funktion mit RIPEMD-160 durchgeführt. Hierzu werden die bereits in Abschnitt 1.7.2 analysierten RIPEMD-160 Funktionen aufgerufen. Die verwendeten Daten werden abschließend sicher mit `burn()` überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

GraphicUserInterface::OnInit ()

Diese Funktion ist der Einstiegspunkt der GUI unter Linux. Hier wird kein Schlüsselmaterial verarbeitet, sondern nur die GUI gestartet und die Verarbeitung der Kommandozeile aufgerufen.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial und ist nicht sicherheitsrelevant.

TextUserInterface::OnRun()

Diese Funktion ist der Einstiegspunkt des Textmode-Interfaces unter Linux. Hier wird kein Schlüsselmaterial verarbeitet, sondern nur die Verarbeitung der Kommandozeile aufgerufen. Eventuell auf der Kommandozeile eingegeben sicherheitskritische Daten, wie z.B. das Passwort, werden per Referenz an diese Funktion weitergereicht.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

UserInterface::ProcessCommandLine()

In der Funktion `UserInterface::ProcessCommandLine()`, die von `TextUserInterface::OnRun()` und `GraphicUserinterface::OnInit()` aufgerufen wird, werden die Kommandozeilenparameter interpretiert und verarbeitet. Dies betrifft insbesondere auch Passwörter, die auf der Kommandozeile übergeben werden. Diese Passwörter werden zunächst in der Variablen `options->Password`, später `cmdLine.ArgPassword` gespeichert.

Die Parameter der Kommandozeile werden mittels `parser.SetCmdLine(argc, argv)` und dann `CmdLine.reset(new CommandLineInterface (parser, InterfaceType))` in `UserInterface::Init()` initialisiert. Sofern ein Passwort

eingegeben wird, wird dieses in einem `VolumePassword` gespeichert. Dies gilt sowohl für normale als auch versteckte, zu schützende Volumes [21] (`password` und `protection-password`).

Die Auswertung der Parameter wird in `UserInterface::ProcessCommandLine()` durchgeführt. Hier werden auch die Aufrufe der entsprechenden Funktionen (Volume erstellen, einbinden, Passwort ändern, etc.) vorgenommen. Von hier werden die Funktionen entsprechend des Funktionsgraphen aufgerufen. Eine weitere Beschreibung der Datenverarbeitung findet sich bei den entsprechenden Funktionen.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

GraphicUserInterface::BackupVolumeHeaders()

In `GraphicUserInterface::BackupVolumeHeaders()` werden Passwörter in die Variable `Mountoptions *options` eingelesen. Sicherheitskritisches Material wird in den Attributen `VolumePassord Password`, `VolumePassword ProtectionPassword`, `shared_ptr <KeyfileList> Keyfiles`, `shared_ptr <KeyfileList> ProtectionKeyfiles` gespeichert. Das Einlesen geschieht über die Erzeugung eines `MountOptionsDialog`, in dem die Mount Optionen festgelegt werden. Zum Einlesen der Passwörter wird ein `VolumePasswordPanel` erzeugt, in welchem letztendlich die Passwörter eingelesen und dann über Referenzen an die betrachtete Funktion durchgereicht werden.

In den „`KeyfileList`"en werden Dateipfade von `Keyfiles` in Listen gespeichert. Diese Information ist nicht sicherheitsrelevant.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

TextUserInterface::BackupVolumeHeaders()

In `TextUserInterface::BackupVolumeHeaders()` werden analog Passwörter eingelesen, hier allerdings über die Funktion `TextUserInterface::AskPassword()`.

Zusätzlich ist anzumerken, dass alle Speicher für Passwörter selbst für den Fall, dass das Programm mit einer Exception unterbrochen wird, mittels Macros aus `finally.h` überschrieben werden.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

GraphicUserInterface::RestoreVolumeHeaders()

Diese Funktion ermöglicht das Neubeschreiben des Volume-Headers mit einem Header aus einem Backup. Das Backup kann im Volume gemäß der Format-Spezifikation (http://www.truecrypt.org/docs/?s=volume-format-specification) eingebettet sein oder von einer externen Datei wiederhergestellt werden.

Für die Wiederherstellung eines Volume-internen Backups wird zunächst das Volume geöffnet um das Header-Backup zu lesen. Dies erfolgt durch das Einlesen der notwendigen Informationen und Passwörter mit einem `MountOptionsDialog`. Mit diesen Informationen wird das Volume geöffnet und der Backup Header extrahiert. Dann wird für das Volume ein neuer Header mit neuem Salt, der aus dem Zufallszahlengenerator initialisiert wird, und dem Passwort aus dem Backup Header generiert.

Wenn ein Header aus einer externen Datei wiederhergestellt wird, so wird die verschlüsselte Backup-Datei zunächst mit einem über `MountOptionsDialog` eingelesenen Passwort entschlüsselt. Dann wird der entschlüsselte Header mit einem neuen Salt, das aus dem Zufallszahlengenerator gewonnen wird, verschlüsselt und als neuer Header im Volume gespeichert. Sofern das Volume-Format die Speicherung eines Backup-Headers erlaubt, wird auch dieses Backup neu geschrieben. Dies ist für das hier betrachtete Format mit XTS-AES-256 der Fall.

Beim Wiederherstellen eines Volume-Headers wird kein neues Passwort abgefragt. Ein Header und sein Backup unterscheiden sich im unterschiedlichen Salt, mit dem die PKCS#5-PBKDF aufgerufen wird, und dem daraus reusultierenden abgeleiteten Schlüssel für die Verschlüsselung der Headerdaten (s. Abs. 4.3.4).

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

TextUserInterface::RestoreVolumeHeaders()

Hier werden die gleichen Operationen durchgeführt wie in `GraphicUserInterface::RestoreVolumeHeaders()`, mit dem Unterschied, dass für die Passworteingabe und den Zufallszahlengenerator die textbasierten Funktionen `TextUserInterface::AskPassword()` und `TextUserInterface::AskKeyfiles()` aufgerufen werden.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CoreBase::ReEncryptVolumeHeaderWithNewSalt()

Hier wird mittels `pkcs5Kdf->DeriveKey()` ein neuer Header-Key generiert, der mit dem neuen Salt und der KDF durch `VolumeHeader::EncryptNew()` verschlüsselt gespeichert wird. Das Salt und der Header-Key werden in einem `SecureBuffer` gespeichert.

Das Neubeschreiben des Headers auf der Festplatte erfolg mittels `write()`. Nach jedem `write()` erfolgt ein `fsync()` auf die Datei. Hiermit ist sicher gestellt, dass die Daten die Festplatte erreichen.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

TextUserInterface::CreateVolume()

Hier werden neben allgemeinen Information für das Volume (normal/versteckt, Größe, Algorithmen etc.) auch das Passwort mittels `AskPassword()` in ein

`VolumePassword` eingelesen. Hierbei ist anzumerken, dass in `CreateVolume()` noch kein Schlüssel für die Verschlüsselung der Daten in dem Volume erzeugt wird, sondern nur das Passwort für die Verschlüsselung des Headers abgefragt wird. Die Erstellung des Volumes und des Datenschlüssels erfolgt in `VolumeCreator::CreateVolume()`.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeCreationWizard::ProcessPageChangeRequest()

Im `VolumeCreationWizard` wird über ein `VolumePasswordWizardPage` Objekt ein Passwort eingelesen, womit das Passwort in einem `VolumePassword` gespeichert wird.

Mit diesem Wizard wird die Erstellung eines neuen Volumes von der GUI durchgeführt. Im Schritt `CreationProgress` erfolgt die Erstellung des Volumes und des Datenschlüssels durch Aufruf von `VolumeCreator::CreateVolume()`.

Zusätzlich wird ein Passwort bei der Erstellung eines versteckten Volumes genutzt, um das externe Volume zu öffnen und die maximal erlaubte Größe des versteckten Volumes zu ermitteln. Hierfür wird das Passwort an `CoreBase::OpenVolume()` übergeben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeCreationWizard::OnProgressTimer()

Diese Funktion aktualisiert den den Fortschrittsindikator des `VolumeCreateWizard` Assistenten. Wenn der überwachte Prozess vollendet ist, wird die `OnVolumeCreatorFinished()` Funktion aufgerufen.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

VolumeCreationWizard::OnVolumeCreatorFinished()

Hier wird das Passwort aus dem `VolumeCreationWizard` Assistenten genutzt, um das erzeugte Volume zu öffnen, damit es formatiert werden kann. Dies geschieht durch Übergabe des Passworts an `CoreUnix::MountVolume()`. Das Passwort wird nicht weiter verarbeitet.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

WizardFrame::SetStep() (ID 51, 52)

In dieser Funktionen wird die Anzeige des nächsten Schrittes eines Wizards, i.e. des `VolumeCreationWizards`, initiiert.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

VolumeCreationWizard::GetPage()

Diese Funktion erzeugt die Dialoge des `VolumeCreationWizard`s, abhängig von dem aktuellen Schritt im Assistenten. Wenn ein verstecktes Volume erzeugt wird, dann wird im Schritt `OuterVolumeContents` das äußere Volume mit dem Passwort über die Funktion `CoreBase::MountVolume()` geöffnet.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MainFrame::OnMountVolumeMenuItemSelected()

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

MainFrame::MountVolume()

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

MainFrame::MountAllFavorites()

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

MainFrame::MountAllDevices()

`MountAllDevices()` ruft die Funktion `GraphicUserInterface::MountAllDeviceHostedVolumes()` auf.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

GraphicUserInterface::MountAllDeviceHostedVolumes()

Hier wird ein `MountOptionsDialog` erzeugt, über den die relevanten Daten zum Mounten von Volumes eingegeben werden (Passwort, Keyfiles etc.). Diese Information werden an `UserInterface::MountAllDeviceHostedVolumes()` weitergegeben. Das Passwort wird nicht weiter in dieser Funktion verarbeitet.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

TextUserInterface::MountAllDeviceHostedVolumes()

Hier wird ein Passwort und eine Liste von Keyfiles vom Benutzer mittels `TextUserInterface::AskPassword()` und `TextUserInterface::AskKeyfiles()` abgefragt. Diese werden in einer Optionen-Variable gespeichert und an `UserInterface::MountAllDeviceHostedVolumes()` übergeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

UserInterface::MountAllDeviceHostedVolumes()

`UserInterface::MountAllDeviceHostedVolumes()` testet alle Datenträger

im System, ob sie mittels TrueCrypt mit dem angegebenen Passwort verschlüsselt sind. Aus aufrufenden Funktionen erhält sie bereits das Passwort und Referenzen auf etwaige Keyfiles. Diese werden dann an `CoreUnix::MountVolume()` übergeben. Eine Liste aller geöffneter Volumes wird schließlich an die aufrufende Funktion zurück gegeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Main.cpp:main()

Die `Main`-Funktion ist der Einstiegspunkt für den `CoreService`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CoreService::Start()

Hier wird der `CoreService` gestartet.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CoreService::ProcessElevatedRequests()

Hier wird der `CoreService` mit erhöhten Rechten gestartet.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CoreService::ProcessRequests()

Hier werden Anforderungen des Benutzers verarbeitet. Im `MountRequest`-Handler wird das Passwort vom Kommunikationskanal, mit dem der CoreService mit dem Anwendungsprogramm kommuniziert (eine Unix-Pipe), eingelesen und an die `MountVolume()`-Funktion weitergegeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

UserInterface::MountAllFavoriteVolumes()

Diese Funktion hängt die FavoriteVolumes ein. Das evtl. in den Mount-Optionen übergebene Passwort wird hier nicht weiter verarbeitet.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

FavoriteVolume::ToMountOptions()

In dieser Funktion werden einzelne Mount-Optionen auf Standardwerte gesetzt.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

TextUserInterface::MountVolume()

In dieser Funktion wird, sofern in den Mount-Optionen noch kein Passwort gesetzt

ist, ein Passwort mittels `TextUserInterface::AskPassword()` in die Mount-Optionen eingelesen. Das Passwort wird nicht verarbeitet.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

GraphicUserInterface::MountVolume()

Hier wird, wie beim TextUserInterface, ein mite einem `MountOptionsDialog` ein Passwort eingelesen, wenn in den Mount-Optionen noch kein Passwort gesetzt ist.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

UserInterface::MountVolume()

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CoreUnix::MountVolume()

In dieser Funktion wird `OpenVolume()` mit dem Passwort als Option aufgerufen. Anschließend wird das Passwort gelöscht.

Problem: Sollte `OpenVolume()` eine Exception auslösen, so wird das Passwort nicht gelöscht bevor die Funktion verlassen wird.

Behebung: Löschen des Passworts im Exception-Handler. **Aufwand:** 1 Personentag.

Im weiteren Verlauf werden die Funktionen `CoreLinux::MountVolumeNative()` und `CoreUnix::MountAuxVolumeImage()` aufgerufen. Da das Passwort-Feld der Mount-Optionen bei Erreichen dieser Funktion aber schon gelöscht ist, kommen diese nicht mit sensiblen Daten in Berührung.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das angesprochene Problem behoben wird.

CoreLinux::MountVolumeNative()

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

CoreUnix::MountAuxVolumeImage()

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

CoreBase::ChangePassword() (ID 31)

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

CoreBase::ChangePassword() (ID 60)

In dieser Funktion wird mit Hilfe von Referenzen auf ein neues Passwort und ein offenes Volume ein `VolumePassword`-Objekt erzeugt, in dem das neue Volume-

Passwort gespeichert wird. Bei Verlassen der Funktion wird das Passwort bei Zerstörung des `VolumePassword`-Objekts sicher überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

TextUserInterface::ChangePassword()

Hier wird das Ändern eines Volume-Passwortes durchgeführt. Sofern das alte Passwort noch nicht auf der Kommandozeile übergeben wurde, wird es mit `AskPassword()` in ein `VolumePassword` eingelesen. Hiermit wird dann das existierende Volume geöffnet. Ebenso wird das neue Passwort mittels `AskPassword()` eingelesen, wenn es nicht auf der Kommandozeile übergeben wurde. Die verwendeten Datentypen sind sicher.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

ChangePasswordDialog::OnOKButtonClick()

Hier wird das Ändern eines Passworts in der GUI initiiert. Das alte und ein neues Passwort werden mit `VolumePasswordPanels` sicher eingelesen. Dann wird durch Aufruf von `CoreBase::ChangePassword()` das Volume-Passwort geändert.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

Volume::ReEncryptHeader()

Diese Funktion generiert einen neuen Header mit den Optionen, wie sie als Argumente übergeben werden (Salt, Header Key, KDF). Es wird ein `SecureBuffer` erzeugt, in dem der neue Header gespeichert wird. Dieser wird an Stelle des alten Headers in das Volume geschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CoreBase::OpenVolume()

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

Volume::Open() (ID 29)

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

Volume::Open() (ID 28)

Diese Funktion führt das tatsächliche Öffnen eines Volumes durch. So wird zunächst das Header-Passwort aus den Keyfiles und dem Passwort erzeugt. Dann wird der Header gelesen und mit den verwendeten Verschlüsselungsalgorithmen an die `VolumeHeader::Decrypt()`-Funktion übergeben. Danach wird optional der Schutz eines versteckten Volumes aktiviert und getestet, ob das Volume ein Betriebssystem enthält.

Schlüsselmaterial (der Header-Schlüssel) wird in dieser Funktion sicher behandelt.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeHeader::Decrypt()

Hier wird mit der PKCS#5-PBKDF2 aus dem Header-Passwort und dem Salt der Header-Schlüssel erzeugt. Mit diesem Schlüssel wird der Header entschlüsselt und mit der Funktion `VolumeHeader::Deserialize()` ein Verschlüsselungsobjekt erzeugt. Der entschlüsselte Header, in dem der Volume-Schlüssel enthalten ist, wird sicher in einem `SecureBuffer` gespeichert. Weiteres Schlüsselmaterial wird nicht verändert oder kopiert.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeHeader::Deserialize()

Die `VolumeHeader::Deserialize()`-Funktion liest einen einkommenden Datenstrom als Headerdaten und interpretiert die Daten entsprechend des Headerformats. Diese Daten werden mittels des `DeserializeEntry()`-Templates extrahiert. An dieses Template wird der entschlüsselte Headerinhalt und ein Offset übergeben. Das Template wird für den jeweiligen zu extrahierenden Datentyp erzeugt. Dieses Template verarbeitet ausschließlich die vorgesehenen Daten und es erfolgt kein unbeabsichtigter Zugriff auf Schlüsselmaterial. Der Volumeschlüssel wird in einen `SecureBuffer` kopiert und initialisiert dann mit diesem Schlüssel ein neu erstelltes Verschlüsselungsobjekt, das über Referenz an die aufrufende Funkion zurück gegeben wird. Da der Schlüssel in einem `SecureBuffer` gespeichert wird, erfolgt ein sicheres Überschreiben der Daten bei Zerstörung des Objekts.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeHeader::Serialize()

Die `VolumeHeader::Serialize()`-Funktion speichert die Header-Informationen in einem `SecureBuffer`. Hierfür wandelt er die Headerinformation in ein portables Format gemäß Tabelle 2.1, das dann auf der Festplatte abgelegt wird. Da Schlüsselmaterial in einem `SecureBuffer` gespeichert wird, erfolgt ein sicheres Überschreiben der Daten bei Zerstörung des Objekts.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeHeader::Create()

Diese Funktion bereitet die Erzeugung eine neuen Volume-Headers aus `VolumeHeaderCreationOptions` vor. Hierzu werden die Headerdaten in den Klassenvariablen gespeichert (`Headerversion`, `RequiredMinProgramVersion`, `DataAreaKey`, `HiddenVolumeDataSize`, `VolumeDataSize`, `EncryptedAreaStart`, `EncryptedAreaLength`, `SectorSize`). Anschließend wird die Funktion VolumeHeader::`EncryptNew()` aufgerufen. Dieser Funktion wird

der Speicherplatz für den Header, der Salt und Header-Schlüssel sowie die PKCS#5-PBKDF2 übergeben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeHeader::EncryptNew()

In dieser Funktion wird die Verschlüsselung des Headers durchgeführt. Hierzu wird ein neues Verschlüsselungsobjekt erzeugt und mit dem übergebenen Schlüssel und Salt im Modus, der für dieses Volume spezifiziert wurde, initialisiert. Dann werden mit diesem Verschlüsselungsobjekt die im Header befindlichen Daten (mit Ausnahme des Salt) verschlüsselt. Die Verarbeitung von Schlüsselmaterial, sowohl der Volume-Schlüssel als auch der Header-Schlüssel, erfolgt hier sicher.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

EncryptionAlgorithm::SetMode()

Diese Funktion überprüft, ob der gewählte Verschlüsselungsalgorithmus den gewünschten Modus unterstützt und, sofern möglich, der Modus mit dem Verschlüsselungsalgorithmus initialisiert. Abschließend wird die dem Modus entsprechende `SetCiphers()`-Funktion aufgerufen.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptionModeXTS::SetCiphers()

Hier wird AES_1 zugewiesen. AES_2 wird für den XTS-Modus ebenfalls initialisiert. Anschließend wird `SetSecondaryCipherKeys()` aufgerufen, um das Schlüsselmaterial zu laden.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptionModeXTS::SetKey()

In dieser Funktion werden die Schlüssel für AES_2 zugewiesen. Hierfür wird Speicher für einen `SecureBuffer` alloziert, in dem das Schlüsselmaterial gespeichert wird, und der Schlüssel wird in diesen Buffer kopiert. Mit einem Aufruf von `SetSecondaryCipherKeys()` wird die Initialisierung der Chiffre mt dem Schlüssel durchgeführt.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

EncryptionModeXTS::SetSecondaryCipherKeys()

Der Chiffre AES_2 wird in dieser Funktion der Schlüssel zugewiesen. Hierfür wird die Funktion `SetKey()` der Chiffre aufgerufen. Das Schlüsselmaterial wird nicht verarbeitet, sondern nur per Referenz an die Chiffren weiter gegeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

VolumeCreator::CreateVolume()

Hier wird der Schlüssel für die Verschlüsselung des Volumes (`MasterKey/DataKey`) und der Salt für die PKCS#5-PBKDF2 aus dem Zufallszahlengenerator gewonnen. Außerdem wird der Headerschlüssel gemäß Abs. 4.3.4 aus dem Passwort und den Keyfiles generiert. Diese Daten werden in `SecureBuffers` gespeichert. Mit diesen Informationen wird der komplette Header erzeugt (s. Abschnitt 2.8) und im Volume gespeichert. Außerdem werden die für ein verstecktes Volume reservierten Bereiche mit Zufallsdaten aus `CoreBase::RandomizeEncryptionAlgorithmKey()` beschrieben. Abschließend wird die Klassenvariable `Options` initialisiert und der `CreationThread()` erzeugt.

Die Klassenvariablen `HeaderKey`, `MasterKey`, `PasswordKey` und `Options` enthalten Schlüsselmaterial. Diese Variablen werden in sicheren Datentypen (`SecureBuffer`, `VolumePassword`, `VolumeCreationOptions`) gespeichert und ausschließlich von der `CreateVolume()` und `CreationThread()` Funktion referenziert.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

VolumeCreator::CreationThread()

In dieser Funktion wird die Erstellung des Volumes durchgeführt. So wird optional der freie Datenbereich mit Zufallsdaten initialisiert und ein Dateisystem angelegt.

Der Thread verwendet Schlüsselmaterial aus den Klassenvariablen `HeaderKey` und `PasswordKey`, die vorher in `VolumeCreator::CreateVolume()` intitialisiert wurden, um den Backup Header zu erzeugen. Dieser wird an der vorgesehenen Stelle im Volume gespeichert.

Um das Volume mit Zufallsdaten zu initialiseren wird zunächst mit `CoreBase::RandomizeEncryptionAlgorithmKey()` ein Verschlüsselungsalgorithmus `E(Schlüssel, Plaintext, Zähler)` initialisiert. Mit Hilfe dieser Chiffre werden Zufallsdaten gemäß Abs. 4.5 erstellt, mit denen das Volume initialisiert wird.

Bewertung PRNG: Der PRNG liefert pseudo-zufällige Zahlen, die ohne zusätzliches Wissen nicht von Zufallsdaten unterscheidbar sind.

Bewertung Funktion: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CoreBase::RandomizeEncryptionAlgorithmKey()

Diese Funktion generiert durch Aufruf der Zufallszahlen-Extraktionsfunktion `RandomNumberGenerator::GetData()` des TrueCrypt-RNGs einen zufälligen Schlüssel, mit dem ein Verschlüsselungsalgorithmus und -Modus initialisiert wird.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptionAlgorithm::SetKey()

Diese Funktion überprüft zunächst, ob die Chiffre initialisiert ist und ob die Schlüssellänge der Anforderung der verwendeten Chiffre entspricht. Wenn dies der Fall ist, wird der Schlüssel der Chiffre initialisiert. Schlüsselmaterial wird ausschließlich per Referenz an die Chiffre-spezifische `SetKey()`-Funktion übergeben und wird ansonsten nicht weiter verarbeitet.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

Cipher::SetKey()

In dieser Funktion wird der Schlüssel in den `SecureBuffer` Key des Chiffre-Objekts kopiert und die Generierung der Rundenschlüssel per `SetCipherKey()` durchgeführt.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CipherAES::SetCipherKey()

Hier wird die Key Schedule der AES-256 Chiffre aufgerufen. Es wird sowohl eine Ver- als auch eine Entschlüsselungs-Key-Schedule erzeugt und im dafür vorgesehenen Feld des Cipher-Objekts gespeichert. Es wird lediglich mit Referenzen gearbeitet und das Schlüsselmaterial wird nur an die Funktionen für die Erzeugung der Key Schedule übergeben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

aes_encrypt_key256() / aes_decrypt_key_256()

Dieses Funktionen erzeugen aus dem AES Schlüssel die Key Schedule für Ver- und Entschlüsselung. Die Ergebnisse werden in einem Speicherbereich abgelegt, der von der aufrufenden Funktion bereit gestellt wird. In beiden Funktionen wird eine Variable `ss[]` angelegt, in der Zwischenergebnisse der Key Schedule gespeichert.

Problem: Die temporäre Variable `ss[]` wird vor dem Verlassen der Funktion nicht überschrieben. Somit können Reste des Schlüsselmaterials im Speicher erhalten bleiben.

Behebung: Sicheres Überschreiben der temporären Variablen einfügen, z.B. durch Aufruf von `burn()`. **Aufwand:** 1 Personentag.

Ver- und Entschlüsselung

Mit dem oben dargestellten Datenfluß wird aus dem eingegebenen Passwort, den Keyfiles und Zufallsdaten der Header sowie die Schlüssel für die Chiffre erstellt. Die Schlüssel werden mit dem Verschlüsselungsalgorithmus in einer Instanz der `Cipher`-Klasse als geschützte Member gespeichert. Damit ist es nur den Klassenfunktionen und Klassenfunktionen, die von der `Cipher`-Klasse abgeleitet werden, erlaubt, auf diese Daten zuzugreifen. Es existiert keine Funktion, um

Schlüsselmaterial (Schlüssel, Key Schedule) aus diesem Objekt zu exportieren. Damit können nur die folgenden Funktionen auf das Schlüsselmaterial im Cipher-Objekt zugreifen:

CipherAES::Encrypt()

In dieser Funktion wird die `aes_encrypt()`- oder die `aes_hw_cpu_encrypt()`-Funktion aufgerufen. Schlüsselmaterial wird nur als Referenz an diese Funktion übergeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Cipher::EncryptBlock()

Diese Funktion überprüft, ob die Chiffre initialisiert wurde und führt dann mittels `Encrypt()` die Verschlüsselung des Datenblocks durch.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Cipher::EncryptBlocks()

Diese Funktion überprüft, ob die Chiffre initialisiert wurde und führt dann mittels Aufrufen von `Encrypt()` in einer Schleife für die übergebene Anzahl Datenblöcke die Verschlüsselung durch.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

Testfunktionen

Die Funktionen `EncryptionTest::*` und `EncryptionTestDialog:*` testen die Funktionen mit Referenzvektoren. Sie verarbeiten kein Schlüsselmaterial, sondern testen die Chiffren mit Testvektoren und -schlüsseln.

Bewertung: Diese Funktionen sind nicht sicherheitsrelevant.

2.5.1 Zusammenfassung und Gesamtbewertung Linux

Für die Analyse wurde untersucht, an welchen Stellen mit Schlüsselmaterial gearbeitet wird. Diese Codestellen wurden anschließend eingehend untersucht.

Die Verarbeitung von Schlüsselmaterial wird, mit der Ausnahme der in der Analyse angegebenen Probleme, sicher durchgeführt. Die Autoren der Software waren sich offensichtlich der Relevanz des Schutz von Schlüsselmaterials bewusst. So wird das Schlüsselmaterial meistens in `SecureBuffer`-Objekten gespeichert. Bei diesen Objekten werden die sensitiven Daten bei Zerstörung einer Instanz im Destruktor überschrieben. Die Beseitigung der Fehler ist mit vertretbarem Aufwad möglich und stellt somit keinen signifikanten Nachteil der Implementierung dar. Bei der Bewertung des Aufwandes war 1 Personentag die kleinste Einheit der Einschätzung, um ein konservatives Maß für den Aufwand der Behebung einzelner Probleme zu geben. Damit erwarten wir einen **Gesamtaufwand von 5 Personentagen, um den**

Programmcode zu korrigieren.

2.6 *Analyse Schlüsselverarbeitung Windows*

Unter Windows werden teilweise Daten vor dem Auslagern geschützt. Da sich aus verschiedenen Punkten in AP3 allerdings ergab, dass die Auslagerungsdatei genau so geschützt wird wie die sensitiven Daten (z.B. durch Full-Disk-Encryption mit verschlüsselter Betriebsystempartition, s. AP3: A.28.1, B.9.2.2.2.2, B.4.2.3), wurde der konsequente Schutz des Schlüsselmaterials vor Auslagerung nicht untersucht.

Wenn mehrere Funktionen mit gleichem Namen existieren, wird in Klammern das Subsystem der untersuchten Funktion angegeben. Der alphabetische Funktions- und Klassenindex am Ende dieses Dokuments enthält die Windows- und Linux-Funktionen.

Makro burn()

Um sicheres Überschreiben von Daten im RAM in C-Code durchzuführen wird das Makro `burn()` genutzt. Hierfür wird ein `volatile` deklarierter Zeiger auf die zu überschreibenden Speicherbereiche genutzt, um die Daten mit Nullen zu überschreiben. Zusätzlich wird die Betriebssystemfunktion `RtlSecureZeroMemory()` aufgerufen, die sicherstellt, dass der übergebene Speicherbereich überschrieben wird [22].

Bewertung: Das Makro `burn()` überschreibt effektiv Speicherbereiche mit Nullen.

AddPasswordToCache()

Diese Funktion kopiert ein übergebenes Passwort in den globalen Passwort-Cache [23]. Es wird zunächst geprüft, ob das Passwort schon im Cache vorhanden ist. Sollte dies nicht der Fall sein, wird das Passwort zum Cache hinzugefügt. Wenn der Cache bereits voll ist und ein neues Passwort hinzugefügt wird, dann wird der Zeiger auf ein altes Passwort überschrieben.

Problem: Wenn der Passwort-Cache voll ist und ein neues Passwort hinzugefügt wird, wird das alte Passwort nicht sicher überschrieben.

Behebung: Sicheres Überschreiben eines gelöschten Passworts mittels `burn()` hinzufügen. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion verarbeitet Passwörter sicher, wenn das genannte Problem behoben wird.

AskVolumePassword()

Diese GUI-Funktion reicht die Referenz auf das vom Benutzer zu erfragende Passwort an die Funktion `PasswordDlgProc()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

BackupVolumeHeader()

Es wird ein `OpenVolumeContext` in den lokalen Variablen `hiddenVolume` oder `volume` mittels `OpenVolume()` initialisiert. Wenn ein Volume erfolgreich geöffnet wurde, wird die Hashfunktion des Zufallszahlengenerators mittels `RandSetHashFunction()` auf die Hashfunktion des Volumes gesetzt. Diese Volumes werden vor Beenden der Funktion mit `CloseVolume()` geschlossen. Wenn kein Volume geöffnet werden kann, wird die Funktion ohne Erzeugen eines Backups verlassen.

Die Funktion `BackupVolumeHeader()` liest mittels `AskVolumePassword()` ein Passwort ein, das in der globalen Variable `VolumePassword` oder der lokalen Variable `hiddenVolPassword` gespeichert wird.

Das `hiddenPassword` und das `VolumePassword` werden vor Beendigung der Funktion sicher mit `burn()` überschrieben.

Ein neuer Header wird in `backup[]` erzeugt. `backup[]` stellt Platz für zwei Header zur Verfügung. Um diesen Header zu initialisieren, werden die folgende Schritte durchgeführt:

1. Eine Kopie des verwendeten AES-XTS-256-Schlüssels Key_2 wird in der lokalen Variable `byte originalK2[]` gespeichert. Diese Variable wird vor Beenden der Funktion mit `burn()` überschrieben.
2. Ein temporärer Schlüssel wird mit Zufallsdaten beschrieben. Dieser Schlüssel wird vor Beenden der Funktion mit `burn()` überschrieben.
3. Der verwendete AES-XTS-256-Schlüssels Key_2 wird mit Daten aus dem RNG beschrieben.
4. `backup[]` wird mit dem temporären Schlüssel verschlüsselt. Bemerkung: Damit enthält der Buffer pseudozufällige Daten
5. Ein neuer Header wird mit `ReEncryptVolumeHeader()` in `backup[]` gespeichert.

Wenn ein verstecktes Volume geöffnet ist, wird auch ein Backup des Hidden-Volume-Schlüssels erzeugt und in der zweiten Hälfte von `backup[]` gespeichert. Beide Header sind verschlüsselt.

Abschließend wird das Header-Backup in eine Datei geschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

BootEncryptedDrive()

Die Funktion bootet von einem verschlüsselten Volume. Dabei werden die globalen Variablen `BootCryptoInfo` für den Krypto-Kontext und die `BootArguments` (die an der festen Speicheraddresse `TC_BOOT_LOADER_ARGS_OFFSET` stehen und das Volume-Passwort enthalten) von der Funktion `MountVolume()` initialisiert. Diese Variablen werde am Ende (und in allen Fehlerfällen) sicher mit der Funktion

`crypto_close()` bzw. `EraseMemory()` geschlossen und überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

BootEncryption::ChangePassword()

Die Funktion ändert das Passwort mit dem der Header eines Boot-Volumes verschlüsselt ist. Dazu bekommt die Funktion eine Referenz auf das alte sowie das neue Passwort übergeben. Zunächst wird der aktuelle Header mittels der Funktion `ReadVolumeHeader()` mit dem alten Passwort entschlüsselt. Der dabei erzeugte Krypto-Kontext (`CRYPTO_INFO`) wird wieder sicher mit `crypto_close()` geschlossen und überschrieben. Der mittels `CreateVolumeHeaderInMemory()` neu erstelle Header existiert in dieser Funktion nur verschlüsselt und wird mittels `File::Write()` auf die Festplatte geschrieben. Ein beim Erzeugen des neuen Headers temporär entstandener Krypto-Kontext (`tmpCryptoInfo`) wird mittels `crypto_close()` sicher geschlossen und überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

BootEncryption::CreateVolumeHeader()

Die Funktion erzeugt zunächst mittels `CreateVolumeHeaderInMemory()` und dem übergebenen Passwort einen verschlüsselten Volume Header. Danach wird die Funktion `ReadVolumeHeader()` aufgerufen, um den eben erzeugten Header mittels des Passworts wieder zu entschlüsseln. Das dazu notwendige Schlüsselmaterial wird über diesen Aufruf ebenfalls zurückgeliefert und dann genutzt, um den Header selbst komplett mittels `DecryptBuffer()` zu entschlüsseln. Dann werden Änderungen am Header durchgeführt und der Header wieder mittels `EncryptBuffer()` verschlüsselt abgespeichert.

Das von `CreateVolumeHeaderInMemory()` bzw. `ReadVolumeHeader()` erhaltene Schlüsselmaterial `cryptoInfo` wird beim Verlassen der Funktion mittels `crypto_close()` sicher gelöscht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

BootEncryption::PrepareInstallation()

Diese Funktion leitet eine Passwort-Referenz lediglich weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

ChangePassword() (in Mount/Mount.c)

Diese GUI Funktion ruft `PasswordChangeDlgProc()` zum Ändern des Passwortes auf und hat selbst keine Referenz auf Schlüsselmaterial.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

ChangePwd()

Die Funktion `ChangePwd()` ändert das Passwort, mit dem ein Header verschlüsselt wird. Hierfür wird der Funktion das alte sowie das neue Passort übergeben. Mit dem alten Passwort wird der aktuelle Header entschlüsselt und damit die Informationen über das Volume extrahiert. Diese Informationen, die auch den Volume-Schlüssel enthalten, werden mit `crypto_close()` sicher geschlossen und überschrieben.

Ein zwischenzeitlich erstellter Krypto-Kontext `PCRYPTO_INFO ci` wird sicher mittels `crypto_close()` geschlossen und überschrieben.

Der neu erstellte Header existiert in dieser Funktion nur verschlüsselt und wird mittels `WriteEffectiveVolumeHeader()` auf die Festplatte geschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CheckDeviceTypeAndMount()

Die Funktion wird indirekt von Windows aufgerufen, wenn ein neues Laufwerk (`Drive`) zur Verfügung gestellt wird. Die Funktion prüft dann, ob es sich um ein Gerät mit TrueCrypt Bootloader handelt und veranlasst, dass das entsprechende Gerät mit den von `LoadBootArguments()` zur Verfügung gestellten Daten (Passwort aus Pre-Boot Authentifizierung und Salt) eingehängt wird. Hierzu wird die Funktion `MountDrive()` mit den entsprechenden Daten (Passwort, Salt) aufgerufen.

Es werden lediglich Referenzen auf Schlüsselmaterial an Funktion übergeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CheckPasswordCharEncoding()

Diese Funktion überprüft zum einen ein übergebenes Passwort auf nicht unterstützte Zeichen, zum anderen kann das Passwort direkt aus einem übergebenen Window Handle auf ein Texteingabefeld gelesen und lokal zwischengespeichert werden. Diese lokale Kopie wird mittels `burn()` sicher überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CheckPasswordLength()

Diese Funktion überprüft die Länge eines in einem Eingabefeld eingegebenen Passwortes.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CipherInit()

Diese Funktion leitet lediglich Referenzen auf Schlüsselmaterial an die Funktionen zur Erzeugung der Rundenschlüssel weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CloseVolume()

Die Funktion `CloseVolume()` schließt einen `OpenVolumeContext`. Hierfür wird zunächst geprüft, ob der `VolumeContext` sinnvollen Inhalt hat. Sollte dies der Fall sein, wird der mit dem OpenVolumeContext assoziierte `CRYPTO_INFO` Kontext mit `crypto_close()` sicher geschlossen und der VolumeContext wird entsprechend als geschlossen markiert. Somit wird an `crypto_close()` nur eine Referenz weitergegeben und diese Funktion selbst verarbeitet kein Schlüsselmaterial.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CompletionThreadProc()

Diese Funktion wartet in einer als Parameter übergebenen Warteschlange (`EncryptedIoQueue`) auf Arbeit und ruft gegebenenfals die Funktion `DecryptDataUnits()` mit einer Referenz auf Schlüsselmaterial auf, welche in der Warteschlange gespeichert ist.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

CopySystemPartitionToHiddenVolume()

Diese Funktion des Bootloaders ist dafür zuständig, den Inhalt der Systempartition in ein verstecktes Volume zu kopieren. Dazu liest sie Klartextdaten von der Systempartition, verschlüsselt diese mittels `EncryptDataUnits()` und der globalen Variable `BootCryptoInfo` und schreibt den verschlüsselten Inhalt in das Zielvolume. Bei Fehlern und Beendigung wird die `BootCryptoInfo` Struktur mittels `crypto_close()` sicher überschrieben. Weiterhin wird die globale `BootArgs` Variable mittels `EraseMemory()` überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

CreateVolumeHeaderInMemory()

Diese Funktion erzeugt einen Header mit den Daten gemäß Tabelle 2.1. Dieser wird in einem an sie übergebenen Speicherbereich gespeichert. Die Daten werden mittels der Makros `mputBytes()`, `mputLong()` und `mputWord()` in den Headerbuffer geschrieben. Das Header-Passwort, das als Argument übergeben wird, wird in das `userKey` Feld einer `keyInfo`-Struktur kopiert. Diese `keyInfo`-Struktur wird am Ende der Funktion sicher überschrieben.

Problem: Wenn Aufrufe an andere Funktionen aus dieser Funktion nicht erfolgreich sind, wird diese Funktion über `return` verlassen, ohne das Passwort sicher zu überschreiben.

Behebung: Hinzufügen einer `error`-Marke am Ende der Funktion, an der sicherheitskritische Daten überschrieben werden. Dann im Fehlerfall Beenden der Funktion über Sprung zur `error`-Marke. **Aufwand:** 2 Personentage.

Außerdem wird eine zweite Kopie dieses Schlüssels erstellt, die im Header

gespeichert wird. Dieser Header wird später mittels `EncrpytBuffer()` verschlüsselt. Damit ist die Verarbeitung des Headers sicher.

Mit dem Volume-Schlüssel wird eine `PCRYPTO`-Struktur `cryptoInfo` mittels `EAInit()` erzeugt. Der Schlüssel wird in dem `cryptoInfo->master_keydata` Feld gespeichert. Es wird ebenfalls der Operationsmodus mittels `EAInitMode()` initialisiert. Für diesen wird der verwendete XTS-AES-256 Key_2 in `cryptoInfo->k2` gespeichert. Abschließend wird die anfangs angelegte `keyInfo`-Struktur sicher mit `burn()` überschrieben.

Wenn `VOLFORMAT` definiert ist (d.h. wenn die Format-Anwendung übersetzt wird), dann wird der Volume-Schlüssel in einen String gespeichert. Eine zweite Kopie dieses Strings wird mit `strcat()` erzeugt. Das Gleiche geschieht für den Header Key. Diese Schlüssel werden dann optional in der GUI angezeigt.

Problem: Diese Strings, die eine ASCII-Darstellung des Volume und Header Schlüssels enthalten, werden nicht sicher überschrieben. Außerdem sind Teile der Schlüssel in der GUI sichtbar.

Behebung: Die Anzeige der Schlüssel und der dafür notwendigen Strings deaktivieren. Hiermit werden beide Probleme gelöst. **Aufwand:** 1 Personentag.

Es wird eine Verschlüsselungs-Struktur `PCRYPTO_INFO *retInfo` an die aufrufende Funktion zurück gegeben, die das Schlüsselmaterial für die Daten des Volumes enthält, welches auch im neuen Header selbst enthalten ist.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn die angesprochenen Probleme behoben werden.

crypto_open()

Die Funktion `crypto_open()` wird verwendet, um eine neue `PCRYPTO_INFO` Struktur zu erzeugen. Diese wird zunächst alloziert und mit 0 initialisiert, und dann mit `VirtualLock()` gegen Auslagerung geschützt.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

crypto_close()

Diese Funktion überschreibt einen übergebenen `CRYPTO_INFO`-Kontext sicher mittel `burn()`.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

crypto_loadkey()

Diese Funktion erzeugt eine Kopie des Passwortes in einer `PKEY_INFO` Struktur. Das Original-Passwort wird sicher mit `burn()` überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

DecipherBlock()

Diese Funktion ruft die Verschlüsselungsfunktion auf und reicht das Schlüsselmaterial an diese weiter. Falls Hardwaresupport für AES-NI-Befehle vorhanden ist wird die Funktion `aes_hw_cpu_decrypt()` verwendet, ansonsten `aes_decrypt()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DecipherBlocks()

Diese Funktion iteriert über mehrere AES-Blöcke und verschlüsselt sie. Falls Hardwaresupport für AES-NI-Befehle vorhanden ist wird die Funktion `aes_hw_cpu_decrypt_32_blocks()` verwendet, ansonsten `DecipherBlock()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DecryptBuffer()

Diese Funktion reicht das Schlüsselmaterial lediglich weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DecryptBufferXTS()

Diese Funktion übergibt lediglich Referenzen auf das Schlüsselmaterial an `DecryptBufferXTSNonParallel()` oder `DecryptBufferXTSParallel()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DecryptBufferXTSNonParallel()

Diese Funktion übergibt lediglich Referenzen auf das Schlüsselmaterial an `EncipherBlock()` oder `DecipherBlock()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DecryptBufferXTSParallel()

Diese Funktion übergibt lediglich Referenzen auf das Schlüsselmaterial an `EncipherBlock()` oder `DecipherBlock()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DecryptDataUnits()

Diese Funktion übergibt lediglich Referenzen auf das Schlüsselmaterial an `DecryptBufferXTS` oder `EncryptionThreadPoolDoWork ()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DecryptDataUnitsCurrentThread()

Diese Funktion reicht das Schlüsselmaterial lediglich an `DecryptBufferXTS ()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

DisplayPortionsOfKey()

Diese Funktion stellt das Schlüsselmaterial in einem GUI-Element dar, sofern Anzeige des Schlüssels gewünscht wird. Dann wird mit `MultiByteToWideChar()` eine Kopie des Schlüsselmaterials erstellt, das im Fenster dargestellt wird und nicht sicher gelöscht wird.

Problem: Schlüsselmaterial wird am Bildschirm dargestellt. Schlüsselmaterial wird nicht sicher überschrieben.

Behebung: Deaktivieren der Darstellung von Schlüsselmaterial behebt beide Probleme. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion sollte so geändert werden, dass sie kein Schlüsselmaterial mehr verarbeitet.

DumpFilterWrite()

Die Funktion holt eine Referenz auf das Schlüsselmaterial über die globale Variable `BootDriveFilterExtension->Queue.CryptoInfo` und gibt diese an die Funktion `EncryptDataUnitsCurrentThread()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EAInit()

Diese Funktion gibt lediglich Referenzen auf Schlüsselmaterial an `CipherInit()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EAInitMode()

Die Funktion gibt lediglich Referenzen auf Schlüsselmaterial an `EAInit()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncipherBlock()

Diese Funktion wählt die zu verwendende Verschlüsselungsfunktion aufgrund des Argumentes `cipher` aus und reicht das Schlüsselmaterial weiter. Falls Hardwaresupport für AES-NI-Befehle vorhanden ist wird die Funktion `aes_hw_cpu_encrypt()` verwendet, ansonsten `aes_encrypt()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncipherBlocks()

Diese Funktion iteriert über mehrere AES-Blöcke und verschlüsselt diese. Falls Hardwaresupport für AES-NI-Befehle vorhanden ist wird die Funktion `aes_hw_cpu_encrypt_32_blocks()` verwendet, ansonsten `EncipherBlock()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptBuffer()

Diese Funktion reicht das Schlüsselmaterial lediglich an `EncryptBufferXTS()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptBufferXTS()

Diese Funktion reicht das Schlüsselmaterial lediglich an EncryptBufferXTSParallel oder `EncryptBufferXTSNonParallel()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptBufferXTSNonParallel()

Diese Funktion reicht das Schlüsselmaterial lediglich an `EncipherBlock()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptBufferXTSParallel()

Diese Funktion reicht das Schlüsselmaterial lediglich an `EncipherBlock()` oder `EncipherBlocks()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptDataUnits()

Diese Funktion reicht das Schlüsselmaterial lediglich an `EncryptBufferXTS()` oder `EncryptionThreadPoolDoWork()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptDataUnitsCurrentThread()

Diese Funktion reicht das Schlüsselmaterial lediglich an `EncryptBufferXTS()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptedIoQueueStart()

Die Funktion `EncryptedIoQueueStart()` startet lediglich einen neuen Thread, der abhängig von der Art des Threads eine der Funktionen `CompletionThreadProc()`, `IoThreadProc()` oder `MainThreadProc()` ausführt. Dieser Thread erhält die Queue Struktur inklusive einer Referenz auf das Schlüsselmaterial.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptedIoQueueStop()

Diese Funktion greift nicht auf Schlüsselmaterial zu.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptionThreadPoolBeginKeyDerivation()

Die Funktion trägt als Parameter übergebene Referenzen auf das Passwort und den mittels PKCS#5-PBKDF2 abgeleiteten Schlüssel als `WorkItem` in die globale Warteschlange `WorkItemQueue` ein.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptionThreadPoolDoWork()

Die Funktion gibt eine Referenz auf des Schlüsselmaterial entweder direkt and die Funktionen `DecryptDataUnitsCurrentThread()` bzw. `EncryptDataUnitsCurrentThread()` weiter, oder sie speichert die Referenz in einem `WorkItem` in einer Warteschlange zur späteren Verarbeitung.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptionThreadPoolStart()

Diese Funktion initialisiert den Thread-Pool und startet die Arbeiter-Threads mit der Funktion `EncryptionThreadProc()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptionThreadProc()

Diese Funktion verarbeitet lediglich Referenzen auf Schlüsselmaterial und gibt diese Referenzen an weitere Funktionen weiter (`EncryptDataUnitsCurrentThread()`, `DecryptDataUnitsCurrentThread()`, `derive_key_sha512()`).

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

EncryptPartitionInPlaceBegin()

Die Funktion `EncryptPartitionInPlaceBegin()` erhält über den Parameter

`volParams` u.a. das Passwort für das zu erzeugende Volume. Dieses wird an die Funktion `CreateVolumeHeaderInMemory()` zur Erzeugung des verschlüsselten Headers übergeben. Weiterhin wird dann mit diesem Passwort der Backup Header mittels `OpenBackupHeader()` entschlüsselt. Die Funktion hält zwei lokale `PCRYPTO_INFO` Strukturen, die mittels `crypto_close()` bei Beendigung und im Falle eines Fehlers sicher gelöscht werden. Der Parameter `volParams` wird hier nicht überschrieben, dies wird von der aufrufenden Funktion `volTransformThreadFunction()` durchgeführt.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

EncryptPartitionInPlaceResume()

Zunächst wird der Backup-Header mittels des Passworts und der Funktion `OpenBackupHeader()` geöffnet und die enstprechenden `PCRYPTO_INFO` Strukturen lokal gespeichert. Mittels der `masterCryptoInfo` (Schlüsselmaterial für die Nutzdaten) und der Funktion `EncryptDataUnits()`, wird dann sukzessiv das Volume verschlüsselt. Bei einem I/O-Fehler wird mittels `DecryptDataUnits()` die Verschlüsselung des entsprechenden Blocks rückgängig gemacht und zurück in das Volume geschrieben. Nach jedem Schritt wird die Funktion `FastVolumeHeaderUpdate()` mit dem Schlüsselmaterial aufgerufen, um den Header zu aktualisieren.

Wurde das Volume vollständig verschlüsselt, so wird `CreateVolumeHeaderInMemory()` zur Erzeugung des Volume Headers mit dem entsprechenden Passwort aufgerufen. Im Fehlerfall und bei Beendigung wird zunächst die Funktion `FastVolumeHeaderUpdate()` nochmals aufgerufen, um den Header zu aktualisieren. Dann werden alle `PCRYPTO_INFO` Strukturen sicher mittels `crypto_close()` gelöscht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

ExtractCommandLine() (in Mount/)

Diese Funktion verarbeitet die Kommondozeilenoptionen, unter denen sich auch ein Passwort befinden kann. Zum Einlesen des Passwortes wird die Funktion `GetArgumentValue()` mit einer Referenz auf die globale Variable `CmdVolumePassword()` aufgerufen. Diese kopiert das Passwort in die globale Variable.

Problem: Das Passwort, das noch in der Orignal-Kommandozeile enthalten ist, wird nach dem Kopieren nicht sofort überschrieben, sondern erst beim Beenden des Programmes (durch `localcleanup()`)

Behebung: Das Passwort in der Original Kommandozeile sollte sofort nach Kopieren überschrieben werden. **Aufwand:** 1 Personentag

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

FastVolumeHeaderUpdate()

Die Funktion `FastVolumeHeaderUpdate()` liest einen verschlüsselten Header und entschlüsselt ihn im Speicher zunächst, um daran Änderungen vorzunehmen. Danach wird der Header wieder verschlüsselt und zurückgeschrieben. Die Operationen finden alle im gleichen Puffer statt, daher ist zusätzliches Überschreiben nicht erforderlich.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

FlushFormatWriteBuffer()

Diese Funktion verarbeitet lediglich Referenzen auf Schlüsselmaterial und gibt diese Referenzen an weitere Funktionen weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

FormatFat()

Diese Funktion erzeugt zunächst eine lokale Kopie eines Teils des Schlüsselmaterials und verwendet dann Zufallsdaten als Schlüssel für weitere Operationen. Das ursprüngliche Schlüsselmaterial wird bei Beendigung der Funktion wiederhergestellt. Lokale Kopien werden mittels `burn()` sicher überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

FormatNoFs()

Diese Funktion erzeugt zunächst eine lokale Kopie eines Teils des Schlüsselmaterials und verwendet dann Zufallsdaten als Schlüssel für weitere Operationen. Das ursprüngliche Schlüsselmaterial wird bei Beendigung der Funktion wiederhergestellt. Lokale Kopien werden mittels `burn()` sicher überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

GetArgumentValue() (in Mount/)

Diese Funktion liest ein Kommandozeilenarguemnt ein und kopiert es an die übergebene Referenz.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

GetCrc32()

Diese Funktion verarbeitet die erhaltenen Daten lediglich, um eine CRC32 Prüfsumme zu berechnen und diese zurückzuliefern.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

GetHeaderField16/32/64()

Diese Funktionen liefern Integerwerte der angeforderten Größe aus dem übergebenen Header zurück.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

GetSystemDriveCryptoInfo()

Diese Funktion liefert eine Referenz auf die `CryptoInfo` Struktur der Systemverschlüsselung zurück.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

GetWorkItemState()

Diese Funktion erhält ein `WorkItem` inklusive Schlüsselmaterial, verwendet das Schlüsselmaterial aber nicht.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

HiberDriverWriteFunctionAFilter0()

Diese Funktion reicht das Schlüsselmaterial lediglich weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

HiberDriverWriteFunctionAFilter1()

Diese Funktion greift nicht auf Schlüsselmaterial zu.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

HiberDriverWriteFunctionAFilter2()

Diese Funktion greift nicht auf Schlüsselmaterial zu.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

HiberDriverWriteFunctionBFilter0()

Diese Funktion greift nicht auf Schlüsselmaterial zu.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

HiberDriverWriteFunctionBFilter1()

Diese Funktion greift nicht auf Schlüsselmaterial zu.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

HiberDriverWriteFunctionBFilter2()

Diese Funktion greift nicht auf Schlüsselmaterial zu.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

HiberDriverWriteFunctionFilter()

Diese Funktion ist dafür verantwortlich, die Daten, die während Hibernation (Ruhezustand) auf die Festplatte geschrieben werden, zu verschlüsseln, da diese Daten nicht die normalen Disk I/O Filter durchlaufen (dazu ist ein spezieller Filter notwendig). Hierzu verschlüsselt die Funktion den übergebenen Puffer mit dem Schlüsselmaterial aus der globalen Variable `BootDriveFilterExtension->Queue.CryptoInfo`. Diese enthält das Schlüsselmaterial bei aktiver Systempartitions-Verschlüsselung. Danach wird das verschlüsselte Material an die regulären Windowsfunktionen zum Schreiben weitergegeben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

InitApp()

Die Funktion `InitApp()` dient lediglich der Initialisierung der Anwendung. Sie erhält die Kommandozeilenargumente, die potentiell Passwörter enthalten können. Diese werden verarbeitet, wenn eine neue Programminstanz mit erhöhten Rechten gestartet werden muss, und kein UAC verwendet werden kann („Portable Mode", s. [24]). Das Programm wird nach diesem Vorgang beendet.

Problem: Die Kommandozeile wird lokal kopiert, aber vor Beendigung des Programms nicht überschrieben.

Behebung: Kommandozeile vor Beendigung des Programms nach der Verarbeitung mit `burn()` überschreiben. **Aufwand:** 1 Personentag

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

KeyFilesApply()

Die Funktion `KeyFilesApply()` erhält als Parameter das Volume-Passwort als Referenz. Zunächst liest die Funktion die angeforderten Keyfiles ein und fügt das daraus gewonnene Schlüsselmaterial einem Pool hinzu. Die nicht länger benötigten Keyfiledaten werden im Speicher sofort mit `burn()` gelöscht. Ebenso wird der Pool bei Beendigung der Funktion sicher gelöscht, sowie mittels `VirtualLock()` vor Auslagerung geschützt. Das übergebene Passwort wird mittels der Daten im Keypool verändert und ansonsten nicht weiter verarbeitet. An dieser Stelle erfolgt auch die Einbindung von Security Tokens.

Problem: Der KeyPool wird nicht immer gelöscht, da im Fehlerfall die Funktion in Zeile 273 mit `return` verlassen wird.

Behebung: KeyPool vor Zeile 273 mittels `burn()` löschen. **Aufwand:** 1

Personentag.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

KeyFileProcess()

Die Funktion `KeyFileProcess()` liest aus dem übergebenen Keyfile Schlüsselmaterial und arbeitet dieses in den übergebenen Pool ein.

Problem: Puffer des Keyfiles wird nicht gelöscht.

Behebung: Puffer des Keyfiles mittels `burn()` löschen. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

LoadBootArguments()

Die Funktion `LoadBootArguments()` ist dafür zuständig, die Argumente (u.a. Passwort) der Pre-Boot Authentifizierung aus dem Speicher zu lesen (wo sie vom TrueCrypt Bootloader abgelegt wurden) und im Treiber selbst in einer globalen Variable (`BootArgs`) zu speichern. Sie wird bei der Initialisierung des Treibers einmalig aufgerufen. Das Passwort wird an den Cache übergeben, sofern die Caching-Funktionalität aktiviert ist.

Die Funktion löscht weiterhin die gelesenen Informationen an ihrer ursprünglichen Stelle im Speicher mittels eines Aufrufs an die Funktion `memset()`. Da der zu überschreibende Speicher durch die Funktion `MmMapIoSpace()` verfügbar gemacht wurde, besteht hier für den Compiler keine Möglichkeit, den Aufruf durch Optimierung ganz oder teilweise zu entfernen.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

LoadPage()

Diese GUI Funktion ist eine Funktion, die verschiedene Eingabefelder anzeigt, unter anderem auch zur Eingabe des Passwortes. Im Falle eines Passworteingabefeldes wird dieses beim Umschalten zu einem anderen Dialog durch „X" überschrieben. Die eigentliche Verarbeitung des Passwortes findet nicht in dieser Funktion statt.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

localcleanup() (Format)

Die Funktion `localcleanup()` in der Format-Anwendung löscht kritische GUI Elemente (Ausschnitte des Keys und Randomness Pools) sowie Passwörter und Keyfiles mittels Aufruf von `WipePasswordsAndKeyfiles()`.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

localcleanup() (Mount)

Die Funktion `localcleanup()` in der Mount Anwendung löscht kritische globale Variablen (`VolumePassword`, `CmdVolumePassword`, `mountOptions`, `defaultMountOptions`), sowie die Kommandozeile selbst.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

main() (Bootloader)

Die `main()` Funktion des Bootloaders verarbeitet selbst kein Schlüsselmaterial und ist nur zur Orientierung im Callergraphen enthalten.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

MainDialogProc() (Format)

Diese GUI Funktion steuert den Hauptdialog des TrueCrypt Format Programmes. Die Kommandozeile des Format Programms enthält kein Schlüsselmaterial. Die Funktion steuert auch Eingabefelder in die Passwörter eingegeben werden und setzt damit die globalen Variablen `volumePassword` und `szRawPassword`, die Schlüsselmaterial enthalten. Des weiteren reicht die Funktion Referenzen auf das Schlüsselmaterial an verschiedene Funktionen weiter (`volTransformThreadFunction()`, `VerifyPasswordAndUpdate()`, `CheckPasswordCharEncoding()`, `CheckPasswordLength()`, `MountHiddenVolumeHost()`, `WipePasswordsAndKeyfiles()`, `OpenVolume()`, `BootEncryption::PrepareInstallation()`). Als grafischen Effekt zeigt die Funktion während des Formatierens Teile des Schlüsselmaterials am Bildschirm an.

Problem: Schlüsselmaterial wird am Bildschirm dargestellt.

Behebung: Deaktivieren der Darstellung von Schlüsselmaterial. **Aufwand:** 2 Personentage.

Problem: Nach dem Einlesen des Passwortes wird dieses im Eingabefeld nicht gelöscht.

Behebung: Überschreiben des Eingabefeldes nach der Eingabe. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn die genannten Probleme behoben werden.

MainDialogProc_Mount() (Mount)

Die Funktion `MainDialogProc()` ist das Hauptfenster der Mount Anwendung. Wird diese Anwendung mit entsprechenden Kommandozeilenparametern gestartet, so wird automatisch aus dieser Funktion heraus die Funktion `MountVolume()` aufgerufen, um mittels des übergebenen Kommandozeilenpasswords oder eines erfragten Passworts (`AskVolumePassword()`) das Volume zu Mounten. Die Passwörter werden jeweils nach dem Mount-Vorgang mittels `burn()` überschrieben.

Kommandozeilenpasswörter werden auch überschrieben, wenn sie nicht verwendet wurden. Weitere aufgerufene Funktionen erhalten ihr Schlüsselmaterial nicht über diese Funktion, sondern über globale Variablen oder vom Benutzer selbst.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MainThreadProc()

Diese Funktion veranlasst die eigentliche Ent- und Verschlüsselung einzelner Datenblöcke der zu lesenden/schreibenden Daten innerhalb eines Volumes. Dazu ruft sie die Funktionen `EncryptDataUnits()` bzw. `DecryptDataUnits()` mit dem Schlüsselmaterial aus der Queue auf. Der so ver- bzw. entschlüsselte Puffer wird dann in die jeweilige Richtung als neuer I/O-Request weitergereicht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

Mount()

Die Funktion `Mount()` ruft mehrmals die Funktion `MountVolume()` auf und übergibt dabei Referenzen auf die globale Variable `VolumePassword`, sowie auf eine lokale Variable `emptyPassword`, die zuvor mit `KeyFilesApply()` durch ein Keyfile verändert wurde (wird verwendet, wenn ausschließlich ein Keyfile benötigt wird). Die Variable `emptyPassword` wird sofort nach Übergabe wieder mit `burn()` gelöscht. Weiterhin wird das Passwort vom Benutzer mittels `AskVolumePassword()` abgefragt und die Methode `KeyFilesApply()` wiederrum verwendet, um evtl. vorhandene Keyfiles zu verarbeiten. Sind nicht mehrere Mountvorgänge aktiv (durch `MountAllDevices()`), wird die globale Variable `VolumePassword` mittels `burn()` sicher gelöscht. Das Passwort für ein verstecktes Volume wird in jedem Falle mit `burn()` gelöscht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MountAllDevices()

Die Funktion entscheidet zunächst, ob die globale Variable `CmdVolumePassword` verwendet oder das Passwort mittels `AskVolumePassword()` vom Nutzer erfragt werden soll. Das zu verwendende Passwort wird in der globalen Variable `VolumePassword` gespeichert. Wurden Keyfiles angegeben, wird die Funktion `KeyFilesApply()` mit einer Referenz auf das Passwort zur weiteren Verarbeitung aufgerufen. Für jedes zu öffnende Volume wird dann weiterhin die Funktion `MountVolume()` mit einer Referenz auf das Passwort aufgerufen. Bei Beendigung der Funktion werden die Passwörter immer mittels `burn()` sicher gelöscht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MountDevice()

Die Funktion `MountDevice()` erhält u.a. eine `MOUNT_STRUCT` Struktur als Parameter. Diese enthält neben sonstigen Informationen auch das Passwort, das

zum Mounten verwendet werden soll (ggf. auch noch ein Passwort für ein verstecktes Volume, das geschützt werden soll, s. [25]). Im Verlauf der Funktion werden diese Informationen an die Funktionen `TCCreateDeviceObject()` und `MountManagerMount()` übergeben die jedoch keine der sicherheitskritischen Informationen aus der Struktur verwenden. Weiterhin wird die Struktur an die Funktion `TCStartVolumeThread()` übergeben, die diese Informationen weiterverarbeitet.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

MountDrive()

Die Funktion `MountDrive()` versucht mittels des übergebenen Passworts den Header des Laufwerks zu entschlüsseln und übergibt dazu das Passwort an die Funktion `ReadVolumeHeader()`. Weiterhin übergibt die Funktion eine Referenz auf die `EncryptedIoQueue()` des Treiberobjekts (`Extension->Queue`), das diesem Laufwerk zugeordnet ist. Dort wird von `ReadVolumeHeader()` das notwendige Schlüsselmaterial aus dem Header abgelegt. Weiterhin wird eine Referenz übergeben, die sonstige kryptographische Informationen (Algorithmus, etc.) speichert. Konnte der Header erfolgreich entschlüsselt und die nötigen Informationen abgelegt werden, so wird die `EncryptedIoQueue()` gestartet, die ab diesem Zeitpunkt für alle kryptografischen Lese- und Schreiboperationen mit diesem Laufwerk zuständig ist.

Die Funktion löscht außerdem nach erfolgreichem Öffnen des Volume-Headers zunächst die Key-Schedule, die durch die Pre-Boot Authentifizierung angelegt wurde, aus dem Speicher (mittels `memset()` auf einen Memory-Mapped Speicherbereich). Danach wird auch das `BootPassword` durch Überschreiben mittels `burn()` aus dem Speicher gelöscht. Der Header wird nur in verschlüsselter Version verarbeitet. Damit ist die Verarbeitung nicht sicherheitsrelevant.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MountFavoriteVolumes()

Diese Funktion erhält selbst kein Schlüsselmaterial, löscht jedoch vor Beendigung der Funktion die globale Variable `volumePassword`, um ein eventuell noch vorhandenes Passwort dort sicher zu überschreiben. Werden mittels der Funktion `Mount()` mehrere Volumes parallel mit dem gleichen Passwort gemountet, so kann die `Mount()` Funktion das Passwort selbst nicht löschen, daher wird dies nochmal in dieser Funktion durchgeführt.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MountHiddenVolHost()

Die Funktion `MountHiddenVolHost()` reicht lediglich das übergebene Passwort weiter an die Funktion `MountVolume()`.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

MountManagerMount()

Die Funktion `MountManagerMount()` erhält eine `MOUNT_STRUCT` Struktur als Parameter, die Passwörter enthält. Die Funktion verwendet jedoch keine kritischen Daten aus dieser Struktur.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

MountOptionsDlgProc()

Diese GUI-Funktion liest ggf. das Passwort für ein verstecktes Volume aus einem Eingabefeld in die globale Variable `mountOptions->ProtectedHidVolPassword`. Danach wird das Eingabefeld überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MountSelectedVolume()

Die Funktion `MountSelectedVolume()` ruft zunächst das entsprechende Dialogfenster mit `MountOptionsDlgProc()` auf, um die nötigen Optionen inkl. Passwörtern zu erhalten. Diese werden in der globalen Variable `mountOptions()` gespeichert. Sind Keyfiles aktiviert, so ruft die Funktion weiterhin `KeyFilesApply()` auf, um das Keyfile mit dem Passwort zu verarbeiten. Bei diesen Vorgängen werden lediglich Referenzen übergeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

MountVolume() (Common)

Die Funktion `MountVolume()` erzeugt zunächst eine `MOUNT_STRUCT` Struktur und speichert in dieser dann das übergebene Passwort. Dann wird die gesamte Struktur mittels der Funktion `DeviceIoControl()` an den Kerneltreiber weitergereicht. Nach Rückkehr dieser Funktion werden in der Struktur gespeicherte Passwörter mit `burn()` sicher gelöscht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

MountVolume() (Bootloader)

Die Funktion `MountVolume()` des Bootloaders erzeugt zunächst eine `BootArguments` Struktur an einer festen Stelle im Speicher und speichert in dieser das Passwort, das vom Benutzer eingegeben wird (Bemerkung: Diese Struktur wird später vom Kerneltreiber gelesen). Das Passwort wird zusätzlich an die Funktion `OpenVolume()` weitergeben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

OpenBackupHeader()

Diese Funktion liest mit Hilfe von anderen Funktionen einen Backup-Header ein und erzeugt einen Verschlüsselungskontext für diesen Header. Es werden lediglich Referenzen auf Schlüsselmaterial bzw. einen Verschlüsselungskontext übergeben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

OpenVolume() (Bootloader)

Diese Funktion wird vom Bootloader genutzt, um ein verschlüsseltes Volume zu öffnen.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

OpenVolume() (Common)

Diese Funktion gibt Referenzen auf Schlüsselmaterial an Funktionen weiter. Es wird ein verschlüsselter Volume Header gelesen, der aber zu keinem Zeitpunkt in dieser Funktion unverschlüsselt sichtbar ist. Ansonsten wird kein Schlüsselmaterial verarbeitet. Diese Funktion gibt einen `OpenVolumeContext` zurück, der eine `CRYPTO_INFO` Struktur mit Schlüsselmaterial enthält.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

PageDialogProc()

Diese Funktion stellt das Hauptfenster der Format-Anwendung dar. In diesem Dialog werden an verschiedenen Stellen Passwörter und Bestätigungspasswörter eingegeben. Diese werden mittels `VerifyPasswordAndUpdate()` auf Korrektheit überprüft und in den globalen Variablen `volumePassword` und `szVerify` gespeichert. Außerdem wird an verschiedenen Stellen `DisplayPortionsOfKeys()` aufgerufen.

Problem: Über die Funktion `SetWindowText()`, `DisplayPortionsOfKeys()` und `DisplayRandPool()` wird Schlüsselmaterial am Bildschirm dargestellt, wenn die Darstellung vom Benutzer gewünscht ist. Damit könnte Schlüsselmaterial von Angreifern ausgespäht werden.

Behebung: Deaktivieren der Möglichkeit, Schlüsselmaterial anzuzeigen. **Aufwand:** 2 Personentage.

Bewertung: Die Funktion verarbeitet Schlüsselmaterial sicher, wenn das angesprochene Problem behoben wird.

PasswordChangeDlgProc()

Die Funktion `PasswordChangeDlgProc()` implementiert die grafische Oberfläche zum Ändern des Passworts und ist für das Einlesen von Passwörtern und Keyfiles verantwortlich. Weiterhin übergibt diese Funktion die Passwörter zur Weiterverarbeitung an entsprechende andere Funktionen.

Bevor Eingabefelder für Passwörter gezeigt werden, werden eventuell vorhandene alte Eingaben mittels der Funktion `SetWindowText()` überschrieben.

Tritt ein Fehler in dieser Funktion auf, werden evtl. gespeicherte Passwörter mittels `burn()` überschrieben. Da Passwörter in den Eingabefeldern auf diese Weise nicht überschrieben werden können, wird hier wiederrum `SetWindowText` verwendet.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

PasswordChangeEnable()

Die Funktion PasswordChangeEnable führt Prüfungen vor dem Ändern der Passwörter durch. Dazu bezieht sie die Passwörter direkt aus den Eingabefeldern der GUI und speichert diese lokal zwischen. Bei Beendigung der Funktion werden die so gespeicherten Passwörter mittels `burn()` wieder gelöscht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

PasswordDlgProc()

Die Funktion zeigt einen Dialog an, in den der Benutzer das Passwort eingeben soll und speichert es gegebenenfalls an die als Referenz übergebene Position im Speicher ab. Nach dem Kopieren des Passwortes wird das Eingabefeld im Dialog mit „X"en überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

ProcessMainDeviceControlIrp()

Die Funktion `ProcessMainDeviceControlIrp()` behandelt alle IRPs (I/O Request Packet für Windows-Treiber), die direkt für den Treiber und nicht für ein bestimmtes Gerät oder Volume gedacht sind. Die Funktion nimmt bestimmte Prüfungen an den Eingangsdaten vor und leitet diese dann teilweise oder vollständig an andere Funktionen weiter, die die eigentliche Funktionalität umsetzen.

Für den Request `TC_IOCTL_MOUNT_VOLUME` wird hier die Passwortlänge geprüft und dann die Funktion `MountDevice()` aufgerufen. Nach Beendigung dieser Funktion werden die Passwörter aus dem Request mittels `burn()` im Speicher überschrieben.

Für den Request `TC_IOCTL_REOPEN_BOOT_VOLUME_HEADER` wird der Request direkt an die Funktion `ReopenBootVolumeHeader()` übergeben. Das Passwort wird hier nicht überschrieben, dies wird von der aufgerufenen Funktion erledigt.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

ProcessVolumeDeviceControlIrp()

Diese Funktion bekommt zwar eine Referenz auf eine Struktur, die auch Schlüsselmaterial enthält übergeben, greift aber selbst nicht darauf zu und gibt diese Teile auch nicht weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

RandgetBytes()

Die Funktion befüllt den übergebenen Puffer mit Zufallsdaten. Der Zufallszahlengenerator wird in AP4 und hier in Abs. 3 separat beschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

ReadEncryptedSectors()

Die Funktion nimmt das Schlüsselmaterial aus dem Speicher (über die globale Variable `BootCryptoInfo`) und reicht es an die Funktion `DecryptDataUnits()` weiter, um einen Puffer zu entschlüsseln. Der entschlüsselte Puffer wird an die angeforderte Stelle im Speicher zurückgeschrieben.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

ReadVolumeHeader()

Die Funktion `ReadVolumeHeader()` versucht mittels des übergebenen Passworts einen Volume Header zu entschlüsseln. Hierzu wird das Passwort zunächst an die Funktion `crypto_loadkey()` übergeben, die das Passwort in einer dafür übergebenen Schlüsselstruktur speichert. Diese ist lokal gespeichert und mittels der Funktion `VirtualLock()` vor Auslagerung geschützt. Mittels der PKCS#5-PBKDF2 wird dann der Header-Schlüssel abgeleitet. Mit diesem wird der Header entschlüsselt und das entsprechende Schlüsselmaterial aus dem Header zur Rückgabe in eine von der aufrufenden Funktion übergebene Variable geschrieben.

Von `crypto_loadkey()` wird eine Kopie des Schlüsselmaterials in `keyInfo` gespeichert. Vor Beenden der Funktion wird diese Kopie mittels `burn()` sicher überschrieben. Durch Aufruf von `DecryptBuffer()` wird eine Kopie des an diese Funktion übergebenen verschlüsselten Header entschlüsselt. D.h. ab diesem Zeitpunkt existiert der Volume-Schlüssel als Klartext im RAM. Aus diesem wird der Volume-Schlüssel in ein `KEY_INFO`- (XTS-AES-256 Key_1) und `PCRYPTO_INFO`-Struct (XTS-AES-256 Key_2) kopiert.

Problem: Die entschlüsselte Kopie des Headers wird nicht sicher überschrieben.

Behebung: Sicheres Überschreiben des entschlüsselten Headersmittels mittels `burn()` hinzufügen. **Aufwand:** 1 Personentag.

Bei Beendigung der Funktion wird das restliche lokal gespeicherte Schlüsselmaterial mittels `burn()` überschrieben.

Bewertung: Die Verarbeitung von Schlüsselmaterial erfolgt sicher, wenn das angesprochene Problem behoben wird.

ReadVolumeHeaderWCache()

Bei dieser Funktion handelt es sich um einen Wrapper um die Funktion

`ReadVolumeHeader()`. Erhält diese Funktion ein Passwort, so versucht sie mittels `ReadVolumeHeader()` dieses zum Entschlüsseln des angeforderten Headers zu nutzen. Ist dieser Vorgang erfolgreich, wird das Passwort im Cache abgelegt, sofern die Cachefunktionalität aktiviert ist. Erhält diese Funktion jedoch ein Passwort der Länge 0, so wird versucht, mittels der existierenden Passwörter im Cache den Header zu entschlüsseln (ebenfalls mittels `ReadVolumeHeader()`).

Der Passwortcache besteht aus einem Array von Passwortstrukturen, die im Kernelspeicher abgelegt sind. Er ist in `Common/Cache.c` als globale Variable definiert.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

ReEncryptVolumeHeader()

In dieser Funktion wird ein Volume Header mit einem neuen Passwort verschlüsselt. Um den neuen Header zu erzeugen wird die Funktion `CreateVolumeHeaderInMemory()` aufgerufen. Der von dieser Funktion zurück gegebene Verschlüsselungskontext `CRYPTO_INFO newCryptoInfo` wird mittels `crypto_close()` geschlossen und sicher überschrieben.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

ReopenBootVolumeHeader()

Die Funktion `ReopenBootVolumeHeader()` öffnet den Header der Full-Disk Encryption mit dem übergebenen (neuen) Passwort. Dazu wird das Passwort an die Funktion `ReadVolumeHeader()` weitergereicht. Der Request inklusive Passwort wird vor Abschluss der Funktion im Speicher überschrieben außer

- der Nutzer hat keinen Zugriff auf das Bootmedium oder
- der Request ist ungültig.

Beide Fälle werden im normalen Betrieb bereits vor Aufruf des Kerneltreibers durch TrueCrypt abgefangen.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

RestoreVolumeHeader()

Die Funktion `RestoreVolumeHeader()` ermöglicht die Wiederherstellung eines Volume Headers. Es kann ein Volume-internes Header-Backup oder ein externes Header-Backup genutzt werden.

- Um ein internes Backup wiederherzustellen wird zunächst das Header-Passwort mittels `AskVolumePasswort()` abgefragt und in der globalen Variable `VolumePassword` gespeichert. Dieses Passwort wird vor dem Beenden der Funktion mittels `burn()` überschrieben. Dann werden die Schlüsseldateien in das Passwort eingearbeitet und mit dem sich ergebenden Passwort über `OpenVolume()` das Volume geöffnet. `OpenVolume()` wird

mit dem Parameter `useBackupHeader = TRUE` aufgerufen, was zur Nutzung des Backup Headers im Volume führt. Ab diesem Zeitpunkt existiert ein `OpenVolumeContext` inkl. Schlüsselmaterial für das Volume, der vor Beenden der Funktion sicher mit `CloseVolume()` geschlossen wird. Aus diesem `OpenVolumeContext` wird mit `ReEncryptVolumeHeader()` ein neuer Header erzeugt, der in einem `buffer[]` gespeichert wird. Dieser neue Header ist in dieser Funktion nur verschlüsselt sichtbar. Abschließend wird der Header in das Volume geschrieben.

- Ein externes Header-Backup aus einer Datei wird wiederhergestellt, indem zuerst die Datei mit den Backup-Headern gelesen wird. Dann wird ein Header-Passwort mittels `AskVolumePasswort()` eingelesen und mit diesem Passwort wird dann über `ReadVolumeHeader()` ein `CRYPTO_INFO`-Kontext erzeugt. Ein evtl. vorhandenes verstecktes Volume wird ebenfalls berücksichtigt. Der `CRYPTO_INFO`-Kontext wird vor Beenden der Funktion sicher mit `crypto_close()` geschlossen und das Header-Passwort gelöscht. Im nächsten Schritt wird mit `ReEncryptVolumeHeader()`, dem geöffneten `CRYPTO_INFO`-Kontext und dem zuvor eingegebenen Passwort ein neuer Header erzeugt und in das Volume geschrieben.

  Problem: Der Rückgabewert von `ReEncryptVolumeHeader()` wird nicht überprüft, und die evtl. ungültige Daten im Header gespeichert. Damit wäre der Zugriff auf das betreffende Volume gestört.

  Behebung: Überprüfen des Rückgabewertes von ReEncryptVolumeHeader() und Behandlung des Rückgabewertes wie beim Wiederherstellen eines Volume-Internen Backups (Setzen des Fehlerstatus und Sprung zum Label `error`). **Aufwand:** 1 Personentag.

Bewertung: Die Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

SaveDriveVolumeHeader()

Die Funktion `SaveDriveVolumeHeader()` hat über die Referenz „`Extension`" Zugriff auf Schlüsselmaterial (`Extension->HeaderCryptoInfo`). Dies wird verwendet, um den Header der Systemverschlüsselung zunächst zu entschlüsseln, entsprechende Informationsfelder dort zu aktualisieren, und diesen dann wieder zu verschlüsseln und zurückzuschreiben. Dabei wird das Schlüsselmaterial und der Header an die Funktionen `EncryptBuffer()` bzw. `DecryptBuffer()` weitergegeben. Der entschlüsselte Header wird zur Prüfung außerdem an die Funktion `GetHeaderField32()` weitergegeben. Da die Funktion den entschlüsselten Header im Speicher wieder mit der verschlüsselten Version überschreibt, ist kein weiteres Löschen erforderlich.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

SendMessage()

Dies ist eine Windows API Funktion zur Übermittlung von Daten an ein Fenster. Sie wird hier nicht näher betrachtet.

SetupThreadProc()

Diese Funktion übergibt Referenzen auf Schlüsselmaterial (welche sie über die globale Variable `BootDriveFilterExtension->Queue` erhält) an die Funktionen `EncryptDataUnits()` und `DecryptDataUnits()`. Zum Speichern des Volume Headers wird außerdem die Funktion `SaveDriveVolumeHeader()` aufgerufen.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

SetWorkItemState()

Diese Funktion erhält ein `WorkItem` inklusive Schlüsselmaterial, verwendet dieses aber nicht.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

ShellExecute()

Dies ist eine Windows API Funktion, die hier zum Starten eines externen Programms verwendet wird.

Bewertung: Diese Funktion ist nicht sicherheitsrelevant.

StartBootEncryptionSetup()

Diese Funktion startet einen neuen Thread mit der Funktion `SetupThreadProc()`, ohne auf Schlüsselmaterial zuzugreifen.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

TCCloseVolume()

Die Funktion `TCCloseVolume()` erhält über `Extension` Schlüsselmaterial und löscht dieses sicher mittels `crypto_close()`.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

TCCreateDeviceObject()

Die Funktion `TCCreateDeviceObject()` erhält eine `MOUNT_STRUCT` Struktur als Parameter, die Passwörter enthält. Die Funktion verwendet jedoch keine kritischen Daten aus dieser Struktur.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

TCDispatchQueueIRP()

Die Funktion `TCDispatchQueueIRP()` ist der zentrale Eingangspunkt für IRPs (IO-Request Packets) aus dem Userspace oder von anderen Geräten. Die eingehenden Anfragen werden entsprechend ihrer Bestimmung an verschiedene Teile des Treibers weitergereicht. Lediglich die Anfragen `TC_IOCTL_MOUNT_VOLUME` und `TC_IOCTL_REOPEN_BOOT_VOLUME_HEADER` enthalten hierbei sicherheitskritische Daten, da diese Anfragen u.a. Passwörter enthalten. Diese Requests werden an `ProcessMainDeviceControlIrp()` weitergereicht.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

TCFormatVolume()

Die Funktion `TCFormatVolume()` erhält über die `FORMAT_VOL_PARAMETERS` Struktur das Passwort für das zu erstellende Volume. Das Passwort wird ausschließlich als Referenz an die Funktion `CreateVolumeHeaderInMemory()` weitergereicht. Der von dieser Funktion erzeugte Header ist gegen Auslagerung geschützt und wird bei Fehler und Beendigung der Funktion mittels `burn()` sicher überschrieben.

Weiterhin erhält die Funktion von `CreateVolumeHeaderInMemory()` eine `CryptoInfo` Struktur, die Schlüsselmaterial enthält. Diese Struktur wird ausschließlich als Referenz an die Funktionen `FormatNoFs()`, `FormatFat()`, und `WriteRandomDataToReservedHeaderAreas()` weitergegeben. Bei Beendigung der Funktion wird die `CryptoInfo` Struktur mittels `crypto_close()` sicher gelöscht.

Problem: Header wird nicht immer überschrieben (Zeile 174, `return` bei Fehler)

Behebung: `return` durch "`goto error`" ersetzen. **Aufwand:** 1 Personentag.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

TCOpenVolume()

Die Funktion `TCOpenVolume()` stellt zunächst ein neues Volume-Objekt bereit, über das später das entschlüsselte Volume ansprechbar sein soll. Hierzu wird der Kontext des aufrufenden Benutzers betrachtet, so dass das entstehende Volume vor Zugriffen durch fremde Benutzer auf dem gleichen System geschützt ist. Zur Entschlüsselung des Volume Headers wird dann die Funktion `ReadVolumeHeaderWCache()` aufgerufen, die u.a. das Passwort als Eingabe enthält (sofern kein Cache verwendet wird). Nach erfolgreichem Aufruf übergibt diese Funktion das Schlüsselmaterial, welches (ähnlich wie bei `MountDrive()`) in der `Extension`-Struktur des Volume-Objekts gespeichert und später durch die `EncryptedIoQueue` verwendet wird.

Mittels `crypto_close()` wird im Fehlerfall die `cryptoInfo` Struktur der Extension sicher gelöscht. Bei versteckten Volumes wird zusätzlich temporär eine lokale Kopie

gespeichert, welche jedoch ebenfalls mittels `crypto_close` gelöscht wird. Die übergebene Mount-Struktur wird nicht überschrieben, dies wird in der indirekt aufgerufenen Funktion `ProcessMainDeviceControlIrp()` durchgeführt, nachdem dort ein eventueller Fehlercode ausgelesen wurde.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

TCStartVolumeThread()

Die Funktion `TCStartVolumeThread()` startet lediglich einen neuen Thread mit der Funktion `VolumeThreadProc()` als Einstiegspunkt und übergibt dabei die Mountstruktur an diese Funktion.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

TrueCryptMainCom::BackupVolumeHeader()

Diese Funktion reicht lediglich Referenzen auf das Schlüsselmaterial an `BackupVolumeHeader()` weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

TrueCryptMainCom::ChangePassword()

Diese Funktion wird von Windows UAC über das Windows COM-Interface aufgerufen. Sie erhält ein altes und ein neues Passwort, mit denen ein Volume-Passwort geändert wird.

Problem: Die Passwörter werden vor Beenden der Funktion nicht sicher überschrieben.

Behebung: Sicheres Überschreiben der Passwörter hinzufügen. **Aufwand:** 1 Personentag.

Bewertung: Die Funktion verarbeitet Schlüsselmaterial sicher, wenn das genannte Problem behoben wird.

TrueCryptMainCom::RestoreVolumeHeader()

Diese Funktion reicht lediglich Referenzen auf das Schlüsselmaterial weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

VolumeThreadProc()

Die Funktion `VolumeThreadProc()` übergibt eine Mountstruktur ausschliesslich an die Funktion `TCOpenVolume()`, welches Schlüsselmaterial in der `Extension`-Struktur des Volumes speichert. Nach erfolgreichem Aufruf von `TCOpenVolume()` startet diese Funktion die `EncryptedIoQueue` mittels `EncryptedIoQueueStart()`, welche das Schlüsselmaterial über die Extension enthält. Tritt ein Fehler auf, wird die Funktion `TCCloseVolume()` aufgerufen.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

volTransformThreadFunction()

Die Funktion `volTransformThreadFunction()` erzeugt eine `FORMAT_VOL_PARAMETERS` Struktur, in der u.a. das Passwort aus der globalen Variable `volumePassword` gespeichert wird. Diese Struktur ist gegen Auslagerung geschützt und wird als Referenz an die Funktionen `TCFormatVolume()`, `EncryptPartitionInPlaceBegin()` und `EncryptPartitionInPlaceResume()` weitergegeben. Die erzeugte Struktur wird bei Beendigung und im Fehlerfall mittels `burn()` sicher überschrieben.

Weiterhin übergibt die Funktion die globale Variable `volumePassword` als Referenz an `MountHiddenVolHost()`.

Bewertung: Die Funktion verarbeitet Schlüsselmaterial sicher.

UacChangePwd()

Diese Funktion reicht lediglich Referenzen auf Schlüsselmaterial weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

UacBackupVolumeHeader()

Diese Funktion reicht lediglich Referenzen auf Schlüsselmaterial weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

UacRestoreVolumeHeader()

Diese Funktion reicht lediglich Referenzen auf Schlüsselmaterial weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

VerifyPasswordAndUpdate()

Diese GUI-Funktion bekommt die Handles auf zwei Eingabefelder (für das Passwort und das wiederholte Passwort) sowie zwei Referenzen, in denen die Eingabewerte gegebenenfalls gespeichert werden sollen. Unter Verwendung von temporären Puffern überprüft die Funktion zunächst die Gleichheit der in den Eingabefeldern enthaltenen Passwörter. Diese temporären Puffer werden nach Benutzung sicher mittels `burn()` überschrieben.

Bewertung: Die Funktion verarbeitet Schlüsselmaterial sicher.

WinMain() (Format)

Diese Hauptfunktion verarbeitet selbst kein Schlüsselmaterial kümmert sich aber um verschiedene Grundlagen für den sicheren Umgang mit globalen Variablen in der

Benutzeroberfläche, welche Schlüsselmaterial enthalten.

Zunächst werden die globalen Variablen `volumePassword`, `szVerify`, `szRawPassword`, `MasterKeyGUIView`, `HeaderKeyGUIView`, `randPool`, `lastRandPool`, `outRandPoolDispBuffer`, `szFileName` und `szDiskFile` durch einen Aufruf von `VirtualLock()` vor dem Auslagern geschützt. Desweiteren wird mit `atexit()` eine Cleanup-Funktion (`localcleanup()`) registriert die bei der normalen Terminierung des Programmes dafür sorgt, dass die oben genannten Variablen beim Programmende durch `burn()` sicher überschrieben werden. Im Falle einer abnormen Terminierung des Programmes (z.B. durch das Betriebssystem) wird diese Cleanup-Funktion allerdings nicht aufgerufen. Dafür gibt es aber unseres Wissens nach auch keine Möglichkeit.

Bewertung: Die Funktion verarbeitet Schlüsselmaterial sicher.

WinMain() (Mount)

Diese Hauptfunktion verarbeitet selbst kein Schlüsselmaterial, kümmert sich aber um einige Grundlagen für den sicheren Umgang mit globalen Variablen in der Benutzeroberfläche, welche Schlüsselmaterial enthalten.

Zunächst werden die globalen Variablen `VolumePassword`, `CmdVolumePassword`, `mountOptions` und `defaultMountOptions` durch einen Aufruf von `VirtualLock()` vor dem Auslagern geschützt. Desweiteren wird mit `atexit()` eine Cleanup-Funktion (`localcleanup()`) registriert, die bei der normalen Terminierung des Programmes dafür sorgt, dass die oben genannten Variablen beim Programmende durch `burn()` sicher überschrieben werden. Im Falle einer abnormen Terminierung des Programmes (z.B. durch das Betriebssystem) wird diese Cleanup-Funktion allerdings nicht aufgerufen. Dafür gibt es aber unseres Wissens nach auch keine Möglichkeit.

Bewertung: Die Funktion verarbeitet Schlüsselmaterial sicher.

WipePasswordsAndKeyfiles()

Diese Funktion überschreibt die Schlüsselmaterial enthaltenden globale Variablen sicher (`szVerify`, `volumePassword`, `szRawPassword`). Die Eingabebuffer der GUI werden mit "X" überschrieben.

Bewertung: Diese Funktion löscht Schlüsselmaterial effektiv.

WriteEncryptedSectors()

Die Funktion übergibt eine Referenz auf Schlüsselmaterial von der globalen Variablen `BootCryptoInfo` an die Funktion `EncryptDataUnits()` um damit den übergebenen Puffer zu verschlüsseln.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

WriteRandomDataToReservedHeaderAreas()

Die Funktion erhält zwar die komplette `CryptoInfo` Struktur, die auch das Schlüsselmaterial selbst enthält, sichert aber lediglich das Originalmaterial temporär und erzeugt zufällige neue Schlüssel, die dann verwendet werden. Diese werden verwendet, um Zufallsdaten zum Überschreiben von bestimmten Bereichen des Volumes zu erzeugen. Vor Beendigung der Funktion und im Fehlerfall werden die Originalschlüssel wieder in die Struktur zurückgeschrieben und lokale Kopien mittels `burn()` sicher gelöscht.

Bewertung: Diese Funktion verarbeitet Schlüsselmaterial sicher.

WriteSector()

Diese Funktion verarbeitet lediglich Referenzen auf Schlüsselmaterial und gibt diese an weitere Funktionen weiter.

Bewertung: Diese Funktion verarbeitet kein Schlüsselmaterial.

2.6.1 Zusammenfassung und Gesamtbewertung Windows

Für die Analyse wurde untersucht, an welchen Stellen mit Schlüsselmaterial gearbeitet wird. Diese Codestellen wurden anschließend eingehend untersucht.

Die Verarbeitung von Schlüsselmaterial wird, mit Ausnahme der in der Analyse aufgeführten Probleme, sicher durchgeführt. Die identifizierten Fehler lassen sich mit vertretbarem Aufwand beheben. Bei der Bewertung des Aufwandes war 1 Personentag die kleinste Einheit der Einschätzung, um ein konservatives Maß für den Aufwand der Entfernung einzelner Probleme zu geben. Damit erwarten wir einen **Gesamtaufwand von 15 Personentagen, um den Programmcode zu korrigieren.**

Bemerkung: Die Architektur der Windows-Software ist, wie bereits in AP4 angezeigt, nicht so strukturiert wie es bei der Linux-Version der Fall ist. Durch optionale Refaktorierung des Codes (Aufwand: 30 Personentage, genaue Analyse notwendig) oder Portierung der Linux-Version (Aufwand: 160 Personentage, s. Dokument „Aufwandsabschätzung für Portierung des Windowscodes") ließe sich eine Kontrolle des Daten- und Programmflusses mit geringerem Aufwand durchführen. Diese Änderung verbessert die Wartbarkeit der Software und würde eine zukünftige Re-Analyse oder Zertifizierung vereinfachen, ist aber nicht notwendig.

2.7 Analyse der Speicherung von Schlüsselmaterial im RAM

Während in Abs. 2.4-2.6 die sichere Datenhaltung innerhalb TrueCrypt betrachtet wurde, wird hier die Möglichkeit von Speicherlecks durch Ausführung von TrueCrypt im Betriebssystem und daraus folgenden externen Einflüssen aus der Betriebssystemumgebung betrachtet.

2.7.1 Datenhaltung während des Programmablaufs

Das Schlüsselmaterial wird nicht speziell geschützt. Es wird wie normale

Benutzerdaten im Arbeitsspeicher (RAM) abgelegt. Er ist somit durch den vom Betriebssystem und der CPU gemeinsam bereitgestellten Speicherschutz, der die Isolation verschiedener Programme während der Ausführung sicher stellt, vor Zugriffen anderer Programme geschützt (s. [4], Vol. 3).

Wenn das Betriebssystem die Auslagerung von Arbeitsspeicher auf die Festplatte unterstützt (Swap oder Auslagerungsdatei), muss zwischen Windows und Linux unterschieden werden. So werden unter Windows die Daten teilweise gegen Auslagerung auf die Festplatte geschützt (s. Abs. 2.6). Unter Linux hingegen erfahren sie keinen gesonderten Schutz und die Daten werden gegebenenfalls in der Auslagerungsdatei gespeichert. Wenn diese auf einem verschlüsselten Volume gelagert wird, dann ist die Datei durch die Verschlüsselung des Volumes geschützt. Dies ist gemäß Annahme aus AP3, Kap. 4, A28, der Fall.

Bewertung: Die Datenhaltung während des Programmablaufs wird aufgrund der Annahmen aus AP3 als sicher angesehen.

2.7.2 Datenhaltung während Suspend-To-RAM und Suspend-To-Disk

Wenn der Computer in den Suspend-To-RAM Modus versetzt wird, verbleibt das Schlüsselmaterial im Arbeitsspeicher. Wenn der Computer in den Suspend-To-Disk Modus versetzt wird, werden diese Daten, genau wie der restliche Arbeitsspeicher auch, auf der Festplatte in einer Hibernate-Datei gespeichert. Wenn diese Datei auf einem verschlüsselten Volume angelegt wird, dann ist das Schlüsselmaterial durch die Volume-Verschlüsselung geschützt. Dies ist gemäß Annahme aus AP3, Kap. 4, A28, der Fall.

Bewertung: Die Datenhaltung während Suspend-To-RAM und Suspend-To-Disk wird aufgrund der Annahmen aus AP3 als sicher angesehen.

2.8 Analyse der veränderungssicheren Schlüsselspeicherung

Im Arbeitsspeicher erfahren die Schlüssel abgesehen vom Speicherschutz des Betriebssystems keinen weiteren Schutz durch CRC oder Hashfunktionen.

Die einzige Komponente mit Schlüsselmaterial, die auf der Festplatte abgelegt wird, ist der Volume Header. Dieser enthält den unverschlüsselten 64 Byte langen Salt und 448 Byte verschlüsselte Volume-Information (s. Tabelle 2.1). Kombiniert nehmen diese Daten 512 Byte ein, was der kleinsten üblichen Sektorgröße entspricht. Die Volume-Informationen werden mittels mit einem Schlüssel, der mit der PKCS#5-PBKDF2 aus dem Passwort abgeleitet wird, im AES-XTS-Modus verschlüsselt.

Inhalt	Größe in Bytes	Start
String „TRUE"	4	0
Header Version	2	4
Minimale Programmversion	2	6
CRC32(Schlüsselbereich)	4	8
RESERVED	16	12
HiddenVolumeDataSize	8	28
VolumeDataSize	8	36
EncryptedAreaStart	8	44
EncryptedAreaLength	8	52
Flags	4	60
Sektorgröße	4	64
RESERVED	120	68
CRC32(0-187)	4	188
Schlüsselbereich	256	192

Tabelle 2.1: Volume-Informationen, die verschlüsselt im Volume Header gespeichert werden.

Durch den CRC32-Schutz der kompletten Volume-Informationen kann zufälliges Kippen einzelner Bits erkannt werden. Ein Angreifer ist nicht in der Lage die Volume-Information zu entschlüsseln und die CRC32-Information sinnvoll zu ändern. Dies ändert nichts an der Möglichkeit des Angreifers, diesen Block mit beliebigen Daten zu überschreiben und damit eine Entschlüsselung des Volumes zu verhindern. Wiederherstellen eines ehemals gültigen Headers ist ebenfalls möglich und kann nicht erkannt werden, wenn das Benutzerpasswort und die verwendeten Keyfiles nicht geändert wurden.

Bewertung: Das Header-Material wird sinnvoll gegen Veränderungen geschützt. Außer einem Replay alter Header-Daten kann ein Angreifer den Header nicht sinnvoll verändern.

3 Quellen für Schlüsselmaterial/Zufallszahlengenerator (AP5.3)

In diesem Abschnitt wird die Generierung von Zufallszahlen in TrueCrypt untersucht. Verwendung finden diese Daten u.a. bei der Erzeugung von Schlüsseln und Initialisierungsvektoren.

3.1 Methodik und Vorgehensweise

Um die Qualität des Zufallszahlengenerators zu bewerten, werden zunächst die Quellen für Zufallsmaterial identifiziert. Externe, vom Betriebssystem bereitgestellte Quellen werden hinsichtlich der Qualität beurteilt und es wird untersucht, ob der Zugriff von TrueCrypt auf diese Quellen sicher ist.

Interne, von TrueCrypt erzeugte Zufallsdaten, werden mittels verschiedener Tests aus der Testsuite nach FIPS140-2 untersucht und es wird eine Abschätzung für die minimal verfügbare Entropie ermittelt.

Darüber hinaus wird analysiert, wie aufwändig das Hinzufügen weiterer externer Quellen, wie z.B. eines USB-Tokens oder eines RNGs in einem TPM, ist. Hierbei wird davon ausgegangen, dass bereits Bibliotheken für den Zugriff auf diese Quellen existieren und diese nur in das Zufallsmaterial von TrueCrypt eingepflegt werden müssen.

3.2 Voraussetzungen und Annahmen

Hier wird wieder davon ausgegangen, dass die typischen Schutzfunktionen eines Betriebssystems greifen. Insbesondere wird davon ausgegangen, dass ein Angreifer keinen Zugriff auf den Pool von Zufallsdaten des Betriebssystems und TrueCrypt hat und diesen nicht verändern kann. Außerdem wird davon ausgegangen, dass die Zufallsquelle des Betriebssystems eine bestimmte Entropie besitzt, die dem BSI bekannt ist. Es wird zwischen der Windows- und Linux-Variante unterschieden.

3.3 Analyse

Der Zufallszahlengenerator ist ebenfalls auf zwei Arten implementiert, jeweils getrennt für Windows und Linux. Die Architektur des Generators ist in beiden Fällen identisch, jedoch gibt es Unterschiede in der Implementierung. Diese sind größtenteils gering (Mischfunktion, Hinzufügen von Entropie), unterscheiden sich aber stark bei den Quellen für Entropie.

3.3.1 Nutzung von Zufallsmaterial

Zufallsmaterial wird an den in Tabelle 3.1 dargestellten Funktionen genutzt.

ID	Aufgabe	Windows	Linux
R1	Festlegen der FAT Dateisystem-ID (4 Byte Seriennummer) [26]	FormatFat()	FatFormatter::Format()
R2	Volume mit Zufallsdaten initialisieren (s. Abs. 2.5)	FormatNoFs() FormatFat()	FatFormatter::Format()
R3	Keyfile mit Zufallsdaten erzeugen	KeyfileGeneratorDlgProc()	CoreBase::CreateKeyfile()
R4	Schlüsselerzeugung für Volume	CreateVolumeHeaderInMemory()	VoluemCreator::CreateVolume()
R5	Salt für einen Header erzeugen	CreateVolumeHeaderInMemory(), BackupVolumeHeader()	VolumeCreator::CreateVolume(), CoreBase::ReencryptVolumeHeaderWithNewSalt(), CoreBase::ChangePassword(), VolumeCreator::CreationThread()
R6	Schlüssel für PRNG (basierend auf AES in XTS Modus) festlegen (s. Abs. 2.5 und 4.5)	WriteRandomDataToReservedHeaderAreas()	CoreBase::RandomizEncryptionAlgorithmKey()
R7	Überschreiben des Köder-Betriebssystems mit PRNG-Daten [27]	BootEncryption::StartDecoyOSWipe()	-
R8	Überschreiben der Status Flags, wenn das Erstellen eines versteckten Betriebssystems beendet wird	BootEncryption::WipeHiddenOSCreationConfig()	-

Tabelle 3.1: Verwendung von Zufallsmaterial

Material aus dem Zufallszahlengenerator wird in den folgenden Bereichen eingesetzt:

- Erzeugung neuer Volumes (R4, R5)
- Erzeugung von Keyfiles (R3)
- Formatieren und Initialisieren von Volumes (R1, R2)
- Ändern von Schlüsseln/Headern (R5)
- Backup von Schlüsseln und Headern (R5)
- Löschen von Datenträgern (R6, R7)

Diese Aktionen sind alle interaktiv, d.h. ein Benutzer sitzt am Computer, startet diese Aktionen und führt sie durch. Während der Durchführung macht der Benutzer Eingaben, aus denen Entropie extrahiert werden kann (z.B. Mausbewegungen, s. 3.4.4, 3.4.5 und 3.5.4).

Im laufenden Betrieb verwendet die Festplattenverschlüsselung kein frisches Zufallsmaterial, d.h. schlechtes Zufallsmaterial ist irrelevant.

3.3.2 Unterscheidung Benutzer- und Serverbetrieb

Zufallsdaten werden lediglich dann genutzt, wenn von einem Benutzer Aktionen initiiert werden. Bei der Nutzung von Zufallsmaterial wird vorher sicher gestellt, dass der Zufallspool mit vom Benutzer erzeugten Zufallsdaten angereichert wird. Für den Serverbetrieb wird die Verschlüsselung bzw. Erzeugung von Volumes ebenfalls von einem Benutzer initiiert. Damit unterscheiden sich die Szenarien, in denen Zufallsmaterial genutzt wird, nicht nach Benutzer- und Serverbetrieb.

3.4 Analyse des Zufallszahlengenerator Linux

Der Zufallszahlengenerator ist in `Core/RandomNumberGenerator.(cpp|h)` implementiert.

3.4.1 Start und Stop des Entropiepool

Die Zufallsdaten werden in einem `SecureBuffer` gespeichert und somit beim Beenden sicher überschrieben. Dieser Buffer ist 320 Byte groß. Diese Größe ist ein vielfaches der Digest-Größe der Hashfunktion, die zum Mischen des Pools genutzt wird. Die Standardeinstellung für die hier verwendete Hashfunktion ist RipeMD-160. Sie kann allerdings vom Benutzer ausgewählt werden, so dass SHA-512 genutzt wird.

Der Zufallszahlengenerator wird durch Aufruf von `RandomNumberGenerator::Start()` gestartet. Dabei werden die folgenden Schritte durchgeführt:

- Reservierung eines Speicherbereichs von 320 Byte für den Entropiepool als `SecureBuffer`,
- Initialisieren des Entropiepools mit Daten aus `/dev/urandom` durch Aufruf von `RandomGenerator::AddSystemDataToPool (true)`.

Beim Beenden des Generators mittels `RandomNumberGenerator::Stop()` wird der allozierte Buffer freigegeben und damit durch den Destruktor von `SecureBuffer` sicher überschrieben.

3.4.2 Hinzufügen von Entropie

Um Daten zum Entropiepool hinzuzufügen, wird die Funktion `RandomNumberGenerator::AddToPool()` (s. Text 3.1) bereit gestellt. Das Hinzufügen der Daten zum Pool erfolgt durch Addition (mod 2^8) der Eingangsdaten

zu den Daten, die bereits im Pool vorhanden sind. Hierdurch ist sicher gestellt, dass die Entropie des Pools nicht veringert wird. Immer wenn 16 Bytes dem Pool hinzugefügt wurden, wird die Mischfunktion

```
void RandomNumberGenerator::AddToPool (const ConstBufferPtr &data)
{
	for (size_t i = 0; i < data.Size(); ++i)
	{
		Pool[WriteOffset++] += data[i];

		if (WriteOffset >= PoolSize)
			WriteOffset = 0;

		if (++BytesAddedSincePoolHashMix >= 16)
			HashMixPool();
	}
}
```

Text 3.1: Reduzierter C++-Code für das Hinzufügen von Entropiedaten zum Entropiepool.

`RandomNumberGenerator::HashMixPool()` aufgerufen.

3.4.3 Mischfunktion

Beim Hinzufügen von Daten in den Pool mittels `RandomNumberGenerator::AddToPool()` (s. Text 3.2) wird nach jeweils 16 Byte zusätzlicher Daten der Pool mittels `RandomNumberGenerator::HashMixPool()` gemischt. In dieser Funktion wird so oft ein Hash über den kompletten Buffer berechnet und dieser dann zum Pool addiert, bis der Pool komplett überschrieben wurde und die Eingangsdaten einen Einfluß auf den kompletten Pool haben (s. Abbildung 3.1). In der Abbildung stellt die Box-Plus-Operation die Addition von 8-Bit-Blöcken (modulo 2^8) dar.

```
void RandomNumberGenerator::HashMixPool ()
{
	BytesAddedSincePoolHashMix = 0;

	for (size_t poolPos = 0; poolPos < Pool.Size(); )
	{
		digest = Hash(Pool);
		for (size_t digestPos = 0; digestPos < digest.Size() &&
poolPos < Pool.Size(); ++digestPos)
		{
			Pool[poolPos++] += digest[digestPos];
		}
	}
}
```

Text 3.2: Reduzierter C++-Code für das Mischen des Entropiepools.

Bewertung: Die Mischfunktion stellt sicher, dass hinzugefügte Entropie erhalten bleibt und neu hinzugefügte Entropie sich auf den Zustand des ganzen Pools auswirkt.

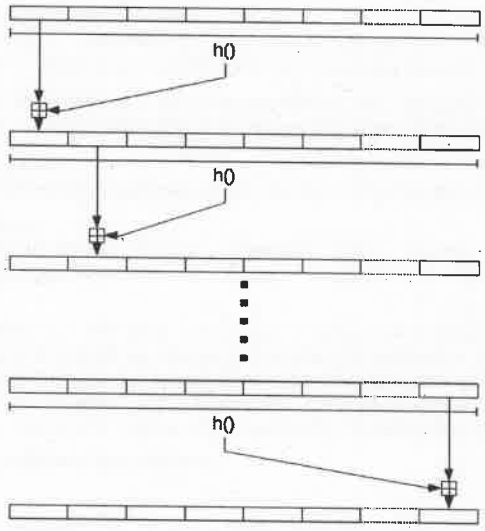

Abbildung 3.1: Mischfunktion des Zufallszahlengenerators mit der Hashfunktion h() unter Linux. Die Blöcke, in die der Pool aufgeteilt ist, haben die Länge des Digest der verwendeten Hashfunktion.

3.4.4 Entropiequelle Betriebssystem

Die Funktion `AddSystemDataToPool()` fügt Zufallsdaten dem Pool hinzu, die von der Zufallszahlenquelle des Betriebssystems mittels `read()` gelesen werden. Unter Linux sind es diese Quellen:

- `/dev/urandom`,
- `/dev/random`,

wobei `/dev/random` optional ist. Die gelesenen Daten werden mit `RandomNumberGenerator::AddToPool()` zum Entropiepool hinzugefügt.

Problem: Im `read()`-Aufruf wird als gewünschte Größe die komplette Poolgröße angegeben. Es wird allerdings nicht verifiziert, dass der Aufruf auch diese Anzahl von Bytes einliest (`read()` gibt die tatsächlich gelesene Anzahl Bytes zurück und

garantiert nicht, dass die gewünschte Anzahl gelesen wird). Es wird nur überprüft, dass der `read()`-Aufruf keine Fehlermeldung zurück gibt. Damit kann es passieren, dass bei diesem read()-Aufruf keine Zufallsdaten gelesen werden.

Behebung: Beim Hinzufügen von Betriebssystementropie muss so lange gewartet werden, bis die benötigte Menge an Entropie gelesen wurde und so lange geblockt werden, oder die Erzeugung mit einem Fehler abgebrochen werden.

Bewertung: Die Funktion fügt sicher Entropie aus dem Betriebssystem-RNG hinzu, wenn das genannte Problem behoben wird.

3.4.5 Entropiequelle Benutzer

Wenn im TextUserInterface Zufallsdaten erzeugt werden sollen, so werden diese von einer auf der Kommandozeile angegebenen Quelle eingelesen oder der Benutzer wird gebeten, zufällige Daten einzugeben, die der Länge des Zufallszahlen-Pools entsprechen. Das Verhalten lässt sich beim Start von TrueCrypt auswählen. Diese Daten werden dann mit `AddToPool()` dem Zufallspool hinzugefügt.

Im graphischen Userinterface erfolgt das Gewinnen von Entropie durch Hinzufügen der Maus-Position. Die Mausposition besitzt eine minimale Entropie von 0 Bit (im Fall, dass die Maus nicht bewegt wird), und eine maximale Entropie von 18 Bits (entsprechend der Anzahl Pixel im Dialogfenster). Die Daten sind unter Umständen stark korreliert. Die Dokumentation von TrueCrypt sagt aus, dass zusätzlich Tastatureingaben für die Entropiegewinnung genutzt wird. Dies ist aber in der Linux-Version nicht der Fall.

Es werden nur Daten in den Zufallspool eingepflegt, die in dem TrueCrypt-Fenster entstehen.

3.4.6 Zufallsdaten aus dem Pool extrahieren

~~VS—NUR FÜR DEN DIENSTGEBRAUCH~~

Daten werden mittels `RandomNumberGenerator::GetData()` aus dem Zufallszahlengenerator gelesen. Hier wird zusätzlich zu bereits existierenden Zufallsdaten nochmal die Funktion `RandomNumberGenerator::AddSystemDataToPool()` aufgerufen und der Pool gemischt. Dann wird die angeforderte Menge Zufallsmaterial aus dem Pool gelesen und im Ausgabebuffer gespeichert. Darauf folgend wird der Entropiepool bitweise invertiert, die `AddSystemDataToPool()`-Funktion aufgerufen und wiederum gemischt. Zum Abschluß wird ein zweites mal die gewünschte Menge Daten aus dem Pool extrahiert und mittels XOR mit dem Ausgabebuffer verknüpft. Der Ausgabebuffer wird an die aufrufende Funktion zurück gegeben. Der Vorgang ist in Abbildung 3.2 dargestellt.

Diese Art der Extraktion von Zufallsmaterial aus dem Pool stellt sicher, dass keine Information über den Zustand des Pools nach außen gelangt.

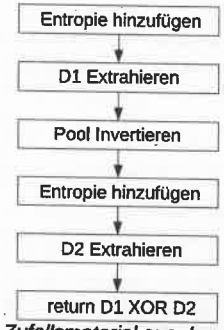

Abbildung 3.2: Extraktion von Zufallsmaterial aus dem Entropiepool.

```
void RandomNumberGenerator::GetData (const BufferPtr &buffer, bool fast)
{
	AddSystemDataToPool (fast);
	HashMixPool();
	for (size_t i = 0; i < buffer.Size(); ++i)
	{
		buffer[i] += Pool[ReadOffset++];
		if (ReadOffset >= PoolSize)
			ReadOffset = 0;
	}
	for (size_t i = 0; i < Pool.Size(); ++i)
	{
		Pool[i] = ~Pool[i];
	}
	AddSystemDataToPool (true);
	HashMixPool();
	for (size_t i = 0; i < buffer.Size(); ++i)
	{
		buffer[i] ^= Pool[ReadOffset++];
		if (ReadOffset >= PoolSize)
			ReadOffset = 0;
	}
}
```

Text 3.3: Reduzierter C++-Code für die Extraktion von Zufallsdaten aus dem Entropiepool.

3.4.7 Hineinschauen in den Pool

Um die Erzeugung von Zufallsmaterial zu visualisieren wird die Funktion `RandomNumberGenerator::PeekPool()` bereitgestellt. Diese Funktion liefert einen Zeiger auf die Pool-Daten an die aufrufende Funktion. Die aufrufenden Funktionen (`VolumeCreationProgressWizardPage::OnRandomPoolTimer()` und `VolumeCreationProgressWizardPage::SetKeyInfo()`) stellen diese Daten im Benutzerfenster dar und stehen einem Angreifer, der auf den Bildschirm schauen kann, zur Verfügung.

Empfehlung: Deaktivieren der `RandomNumberGenerator::PeekPool()` Funktion.

3.4.8 Test der Daten aus dem Zufallsdatengenerator

Es wurden ca. 1GB Daten aus dem Zufallszahlengenerator mit mit verschiedenen Tests aus der dieharder-Testsuite überprüft. Die Resultate der Tests sind in Abschnitt 7.1 wiedergegeben.

Bewertung: Die vom TrueCrypt-Zufallszahlengenerator gelieferten Zufallsdaten eignen sich für kryptografische Anwendungen.

3.5 Analyse des Zufallszahlengenerator Windows

Der Zufallszahlengenerator für Windows ist in `Common/Random.(c|h)` implementiert.

3.5.1 Start und Stopp des Entropiepool

Der Entropiepool wird mit der Funktion `RandInit()` gestartet. Hier wird der Speicher für den Entropiepool alloziert (wie unter Linux 320 Byte) und ein Thread `PeriodicFastPollThreadHandle()` gestartet, der periodisch die Funktion `PeriodicFastPollThreadProc()` aufruft, sowie einen Maus- und Tastatur-Handler installiert, damit Ereignisse dieser Eingabegeräte als Entropiequelle genutzt werden.

Beim Beenden mittel `RandStop()` wird der Thread `PeriodicFastPollThreadHandle()` gestoppt, die Maus- und Tastatur-Handler entfernt sowie optional der Pool sicher mit `burn()` überschrieben.

3.5.2 Mischfunktion

Es wird wie in der Linux-Version nach Hinzufügen von jeweils 16 Byte ein Mischen mit der Hashfunktion durchgeführt. Die Mischung erfolgt in der Funktion `Randmix()` vergleichbar zur Linux-Variante, allerdings wird statt der Addition von Blöcken (mod 2^{32}) die XOR-Verknüpfung gewählt. Bezüglich der Sicherheit macht dies aber keinen Unterschied. Beim Beenden des Zufallsgenerators wird der Pool mittels `burn()` sicher überschrieben (s. Abs. 2.4).

3.5.3 Hinzufügen von Entropie

Zum Hinzufügen von Entropie wird das Makro `RandaddByte()` sowie die Wrapperfunktionen `RandAddInt()` und `RandaddBuf()` aufgerufen. Neue Zufallsdaten werden (mod 2^8) zum bisherigen Poolinhalt addiert. Immer wenn 16 Bytes dem Pool hinzugefügt wurden, wird die Mischfunktion `Randmix()` aufgerufen.

3.5.4 Entropiequellen

Während in der Linux-Variante nur die Mausbewegungen und die vom Betriebssystem bereitgestellen Zufallsquellen als Quelle dienen, verarbeitet die Windows-Version deutlich mehr Daten.

So sammelt die `FastPoll()` Funktion Daten aus den in Tabelle 3.2 dargestellten Funktionen. Die `FastPoll()`-Funktion wird periodisch vom Thread `PeriodicFastPollThreadProc()` aufgerufen, um den Entropie-Pool anzureichern. Außerdem werden Callbacks für Tastatur- und Mauseingaben eingerichtet, die dem Pool Daten hinzufügen.

Die `SlowPoll()` Funktion fügt Statistiken aus dem Netzwerk- und dem Festplattensubsystem zum Pool hinzu. Statistiken für das Festplattensubsystem müssen außerhalb von TrueCrypt aktiviert werden, da diese ansonsten nicht gesammelt werden (mittels `diskperf -y`). Zusätzlich wird die Zufallszahlenfunktion `CryptGenRandom()` der Microsoft CryptoAPI aufgerufen und in den Pool eingefügt. Abschließend wird der Buffer mit `burn()` überschrieben und der Pool gemischt.

Es wird in beiden Funktionen nicht überprüft, ob `CryptGenRandom()` Daten zurück liefert. Nur in der `SlowPoll()` Funktion werden die von `CryptGenRandom()` gelieferten Daten sicher gelöscht.

Empfehlung: Hinzufügen von sicherem Löschen der abgefragten Daten in den `FastPoll()` und `SlowPoll()` Funktionen. Überprüfen, ob `CryptGenRandom()` die gewünschte Menge Zufallsmaterial liefert. **Aufwand:** 1 Personentag.

OS-Funktion	Bemerkung
`GetActiveWindow()`	Handle auf aktives Fenster
`GetCapture()`	Handle auf Fenster mit Maus
`GetClipboardOwner()`	Handle auf Besitzer der Zwischenablage
`GetClipboardViewer()`	Handle auf Betrachter der Zwischenablage
`GetCurrentProcess()`	Pseudohandle für aktuellen Prozeß
`GetCurrentProcessId()`	Prozeß ID
`GetCurrentThread()`	Pseudohandle auf Thread
`GetCurrentThreadId()`	Thread ID
`GetCurrentTime()`	Aktuelle Zeit
`GetDesktopWindow()`	Handle für das Desktop-Fenster
`GetFocus()`	Handle für Fenster, das den Fokus besitzt
`GetInputState()`	Befinden sich Maus- oder Tastaturereignisse im Eingabepuffer?
`GetMessagePos()`	Cursor für letzte Eingabe
`GetMessageTime()`	Zeit der letzten Eingabe
`GetOpenClipboardWindow()`	Handle auf Fenster mit offener Zwischenablage
`GetProcessHeap()`	Adresse des Prozeßheaps
`GetProcessWindowStation()`	Handle auf aktuelle Window Station
`GetQueueStatus(QS_ALLEVENTS)`	Art der Nachrichten im Nachrichtenpuffer
`GetCaretPos()`	Position des Einfügecursors
`GetCursorPos()`	Aktuelle Cursor-Position
`GlobalMemoryStatus()`	Informationen über Speicher. Nicht sicher [1]
`GetThreadTimes()`	Information über Ausführungszeit des Thread: User, Kernel, Start- und Stopzeit
`GetProcessTimes()`	Information über Ausführungszeit des Prozeß: User, Kernel, Start- und Stopzeit
`GetProcessWorkingSetSize()`	Informationen über Working Set (Speicher)
`GetStartupInfo()`	Start-Informationen. Einmalig ausgewertet
`QueryPerformanceCounter()`	Inhalt des Performance Counters
`CryptGenRandom()`	Zufallsfunktion der Microsoft CryptoAPI

Tabelle 3.2: Entropiequellen der SlowPoll()-Funktion unter Windows

1 Diese Funktion gibt auf Maschinen mit mehr als 4GB RAM aufgrund eines Overflows „-1" zurück

3.5.5 Zufallsdaten aus dem Pool extrahieren

`RandGetBytes()` generiert Zufallsdaten aus dem Zufalls-Pool. Zunächst werden Daten aus der `FastPoll()` und optional aus der `SlowPoll()` Funktion dem Zufallspool hinzugefügt und dieser dabei gemischt. Dann wird die angefragte Menge Zufallsmaterial aus dem Pool in den Ausgabepuffer kopiert. Darauf folgend wird der Pool bitweise invertiert, Zufallsdaten mit der `FastPoll()` Funktion hinzugefügt, und gemischt. Nun wird ein zweiter Datenblock aus dem Pool mit dem Ausgabepuffer per XOR verknüft. Diese Daten werden an die aufrufenden Funktion zurück gegeben.

Dies entspricht der Extraktion wie bei Linux.

3.5.6 Hineinschauen in den Pool

Die Funktion `RandpeekBytes()` ermöglicht einen Blick in den aktuellen Zustand des Zufalls-Pools. Es wird eine Kopie des aktuellen Poolinhalts erstellt und diese an die aufrufende Funktion zurück gegeben.

Problem: Diese Daten werden wie in der Linux-Variante dem Benutzer angezeigt und stellen somit ein Datenleck dar.

Behebung: Deaktivierung der `RandPeekBytes()` Funktion.

3.5.7 Test der Daten aus dem Zufallsdatengenerator

Es wurden ca. 1GB Daten aus dem Zufallszahlengenerator mit mit verschiedenen Tests aus der dieharder-Testsuite überprüft. Die Resultate der Tests sind in Abschnitt 7.2 wiedergegeben.

Bewertung: Die vom TrueCrypt-Zufallszahlengenerator gelieferten Zufallsdaten eignen sich für kryptografische Anwendungen.

3.6 Hinzufügen zusätzlicher Entropiequellen

Zusätzliche Entropiequellen können zum Zufallszahlengenerator hinzugefügt werden, indem Daten aus diesen Quellen mittels `AddToPool()` (Linux) und `RandaddBuf()` (Windows) in den Zufallspool eingespeist werden. Unter Linux erfolgt das Einfügen am einfachsten in der Funktion `AddSystemDataToPool()`, unter Windows in der Funktion `FastPoll()` oder `SlowPoll()`.

Aufwand: Es wird vorausgesetzt, dass die zuätzliche Entropiequelle über eine bereits existierende Bibliothek angesprochen werden kann. Dann kann die Integration in TrueCrypt in jeweils 2 Personentagen für Windows und Linux durchgeführt werden.

4 Algorithmische Darstellung verwendeter Algorithmen

In diesem Abschnitt werden die in TrueCrypt verwendeten Algorithmen zusammenhängend algorithmisch dargestellt.

4.1 Definitionen

4.1.1 Definitionen aus Standards

- IEEE 1619-2007 [12]
  „data unit": Abs. 4.3.1. Diese ist in TrueCrypt 512 Byte groß.
  $\text{XTS-AES-blockEnc}(Key,P,i,j)$: Abs. 5.3.1
  $C=\text{XTS-AES-Enc}(Key,P,i)$: Abs. 5.3.2
  $P=\text{XTS-AES-blockDec}(Key,C,i,j)$: Abs. 5.4.1
  $P=\text{XTS-AES-Dec}(Key,C,i)$: Abs. 5.4.2
  Implementiertung: S. Abs. 1.3-1.5
- FIPS PUB 198 [20]
  $m=\text{HMAC}(K,text)_t$: Verarbeitung einer Nachricht beliebiger Länge M gemäß HMAC Standard, Abs. 5. t ist die Länge des resultierenden MAC in Bytes. t ist in TrueCrypt für den XTS-AES-256-Modus 64 Byte.
  Implementierung: s. Abs. 4.1.7 und 4.1.8
- Secure Hash Standard [14]
  $h=\text{SHA512}(M)$: Verarbeitung einer Nachricht beliebiger Länge M gemäß Secure Hash Standard, Abs. 6.4, S. 23 ff. Der Digest h besitzt eine Länge von 64 Bytes.
  Implementierung: s. Abs. 1.7.1
- A Strengthened Version of RIPEMD [15]
  $h=\text{RipeMD160}(M)$: Verarbeitung einer Nachricht beliebiger Länge M gemäß [15]. Der Digest h besitzt eine Länge von 20 Bytes.
  Implementierung: s. Abs. 1.7.2
- PKCS#5 [19]
  $DK=\text{PBKDF2}(P,S,c,dkLen)$: Schlüsselableitung gemäß PBKDF2, Abs. 5.2, S. 9f.
  Implementiet in: `derive_key_rmd160()`, `derive_key_sha512()`
- IEEE802.3-2008
  $y=\text{CRC32}()$: CRC-Berechnung gemäß Abs. 3.2.9.
  Implementiert in: `Crc32::Crc32()`, `GetCrc32()`

4.1.2 Weitere Definitionen

1 Byte = 8 Bit

P_V	: Volume Passwort
S	: Salt, 64 Byte = 512 Bit
H	: Header, 512 Byte = 4096 Bit
K_V	: Volume-Schlüssel, 64 Byte = 512 Bit

P	: Klartext
C	: Chiffretext
FDE	: wahr, wenn die verarbeitete Partition als Betriebssystempartition für Full-Disk-Encryption genutzt wird, ansonsten falsch.
$Linux$	: wahr, wenn TrueCrypt unter Linux läuft
$Windows$	: wahr, wenn TrueCrypt unter Windows läuft

Operation $x\|\|y$: Konkatenation von Datenblöcken x und y.

Operation $x\oplus y$: Exklusiv-Oder-Verknüpfung der einzelnen Bits der Vektoren x und y. Es gelte: $\text{len}(x)=\text{len}(y)$

Operation $\neg x$: Negierung von x

Addition und Multiplikation werden stets explizit geschrieben, um Verwechslung von Variablennamen mit Multiplikation zu verhindern. Variablen werden mit kursiver Schrift dargestellt.

4.1.3 Funktion len()

$y=\text{len}(x)$ liefert die Länge des Array x in Bytes zurück.

4.1.4 Funktion readfile()

$y=\text{readfile}(file,len)$: liest len Bytes von Datei $file$ ein.

4.1.5 Funktion filesize()

$y=\text{filesize}(file)$: liefert die Länge von $file$ in Bytes zurück.

4.1.6 Funktion inv()

Bitweise Negierung eines Arrays x bestehend aus $8\cdot\text{len}(x)$ Bits, identifiziert durch x_i. Ebenso werden die einzelnen Bits von y durch y_i identifiziert.

$y=\text{inv}(x)$:
  for$(i=0,1,\ldots,8\cdot\text{len}(x)-1)$
    $y_i=\neg x_i$
  end for
  return y

4.1.7 Funktion HMAC_RMD160()

$m=HMAC-RMD160(K,text)_t$ Anwendung von $HMAC$ mit der $RipeMD160$ Hashfunktion. t ist in TrueCrypt für den XTS-AES-256-Modus 64 Byte. Diese Funktion wird in $PBKDF2$ als Funktion F genutzt, wenn für das Volume RIPEMD160 als Hashfunktion gewählt wurde.

Implementiert in: `derive_key_rmd160()`.

4.1.8 Funktion HMAC_SHA512()

$m = HMAC\text{-}SHA512(K, text)_t$ Anwendung von *HMAC* mit der *RipeMD160* Hashfunktion. t ist in TrueCrypt für den XTS-AES-256-Modus 64 Byte. Diese Funktion wird in *PBKDF2* als Funktion F genutzt, wenn für das Volume SHA512 als Hashfunktion gewählt wurde.

Implementiert in: `derive_key_sha512()`.

4.2 Zufallszahlengenerator

P_r : Leseposition (in Bytes) im Entropiepool
P_w : Schreibposition (in Bytes) im Entropiepool
E : Entropiepool der Größe 320 Bytes

Auf E kann zusätzlich über $(E_0 || E_1 || ... || E_{319})$ zugegriffen werden, wobei alle E_i eine Größe von 1 Byte besitzen. Zugriffe auf E und E_i verändern somit die jeweils korrespondierende Variable.

$E = FastPoll(E)$ stellt die Anwendung der `FastPoll()`-Funktion aus Abs. 3.5.4 dar.

$E = SlowPoll(E)$ stellt die Anwendung der `SlowPoll()`-Funktion aus Abs. 3.5.4 dar.

4.2.1 Start des RNG

$IV =$ `0xe892337d82adaeff4b874da5423d4200674cbd6b121a3b3f528ccf0 96aeba3ff32f14ffd49a59d1efa0d9cfeec008a9174bb6b7060b1b98e213be b3fb624385a77208b704080ed628690cfffd03d638405144fb4e12f69a39c5 de516d62fca9bb5f11d8e51d68e9e7a7cf17fe7547a2caaee8639064c27afc 60c9708c0c238089e05c2708f54410b42028f3434126978647b987042d4007 3a63313cedb4e0a9bc728eef693ed0f9712dcfd570bca0dfa7144734936616 0735d2835081d75977c5dca606856242bd91372554a13c7feb8bebd162cebe 64cbeab54b8c0a65eec9eea7187f0e1660af13f44456184a25c72edb94601e 57921f5f70149832dbd7951abc3506c65dbc32cf06246a06337574fec72a92 0b8f386753e7e068d00458ca11bba35bf7b52423545be01201fecad7ac2427 d4df0a35c0450e14f4d01a06587`

ist ein vordefinierter Wert der Länge 320 Bytes (Bemerkung: Er entsteht während des Tests des Zufallszahlengenerators und wird anschließend nicht überschrieben). Er stellt unter Linux einen bekannten Anfangszustand des Entropiepools her. Unter Windows wird der Entropiepool mit Nullen initialisiert.

RNGStart():
  $P_r = 0$
  $P_w = 0$
  if($Linux$)
  then
    $E = IV$
    $E = \text{AddSystemDataToPool}(E)$
  else
    $E = 0$
  end if

Unter Windows wird zusätzlich ein Thread gestartet, der periodisch Daten mittels `FastPoll()` zum Pool hinzufügt (2 mal pro Sekunde). Siehe Abs. 3.5.4 .
Implementiert in: `RandomNumberGenerator::Start()`, `RandInit()`.

4.2.2 Mischen des Entropiepools

$USE\text{-}SHA512$: wahr, wenn TrueCrypt für das verarbeitete Volume SHA512 verwendet.
$USE\text{-}RIPEMD160$: wahr, wenn TrueCrypt für das verarbeitete Volume RIPEMD160 verwendet.

$y = \text{Mix}(E)$:
  if($USE\text{-}SHA512$)
    then $step = 64$
  else if($USE\text{-}RIPEMD160$)
    then $step = 20$
  end if
  for($i = 0, step, 2 \cdot step, ..., \text{len}(E) - step$)
    if($USE\text{-}SHA512$)
      then $t = \text{SHA512}(E)$
    else if($USE\text{-}RIPEMD160$)
      then $t = \text{RipeMD160}(E)$
    end if
    for($j = 0, 1, ..., step - 1$)
      if($Linux$)
      then
        $E_{i+j} = E_{i+j} + t_j \pmod{2^8}$
      else
        $E_{i+j} = E_{i+j} \oplus t_j$
      end if
    end for
  end for
  return E

Implementiert in: `RandomNumberGenerator::HashMixPool()`, `Randmix()`.

4.2.3 Hinzufügen von Daten zum Entropiepool

y=AddToPool(P_r,P_w,D,E):
  for(i=0,1,...,len(L)−1)
    E_{P_v}=E_{P_v}+D_i(mod 2^8)
    P_w=P_w+1(mod len(E))
    if(P_w≡0(mod 16))
      E=Mix(E)
    end if
  return E

Implementiert in: `RandomGenerator::AddToPool()`, `RandaddByte()`/`RandAddInt()`/`RandaddBuf()`.

4.2.4 Hinzufügen von Entropie

Diese Funktion wird genutzt, um Entropie dem Pool hinzuzufügen. Dabei kann ein zusätzlicher Parameter *slow* angegeben werden, der entscheidet, ob langsam antwortende Entropiequellen genutzt werden.

y=AddEnt(E,*slow*):
  if(*Linux*)
    D=readfile(/dev/urandom,len(E))
    y=AddToPool(P_r,P_w,D,E)
    if(*slow*)
      D=readfile(/dev/random,len(E))
      y=AddToPool(P_r,P_w,D,E)
    end if
  else
    E=FastPoll(E)
    E=Mix(E)
    if(*slow*)
      E=SlowPoll(E)
      E=Mix(E)
    end if
  end if
  return E

Implementiert in: `RandomGenerator::AddSystemDataToPool()`, `RandgetBytes()`.

4.2.5 Extrahieren von Zufallsdaten aus dem Entropiepool

y=RNG(n):
  y=RNGget(E,n,P_r)
  return y

B : Puffer der Länge n Byte

B_i : Bytes des Puffers B

B=RNGget(E,n,P_r):
  E=AddEnt(E,*true*)
  E=Mix(E)
  for(i=0,1,...,n−1)
    B_i=E_{P_r}
    P_r=P_r+1(mod len(E))
  end for
  E=inv(E)
  E=AddEnt(E,*false*)
  E=Mix(E)
  for(i=0,1,...,n−1)
    B_i=B_i⊕E_{P_r}
    P_r=P_r+1(mod len(E))
  end for
  return B

Implementiert in: `RandomNumberGenerator::GetData()`, `RandGetBytes()`.

4.3 Header-Verarbeitung

4.3.1 Erzeugung eines Header-Passworts

Keyfiles : Liste mit Dateinamen von Keyfiles, vom Benutzer eingegeben.
P_v : Volume-Passwort, vom Benutzer eingegeben. Das Volume-Passwort wird durch Aufruf der readPasswordandKeyfiles() Funktion durch Anwendung der Keyfiles verändert.

P_v=readPasswordandKeyfiles():
  for(*file* in *Keyfiles*)
    P_v=KeyfileApply(P_v,*file*)
  end for
  return P_v

Implementiert in: `Keyfile::ApplyListToPassword()`, `KeyfilesApply()`.

4.3.2 Anwenden eines Keyfiles

t : Buffer der Größe 4 Byte

D : Buffer der Größe 1024 kByte

Auf D kann zusätzlich über $(D_0||D_1||...||D_{(1024\cdot1024-1)})$ zugegriffen werden, wobei alle D_i eine Größe von 1 Byte besitzen. Zugriffe auf D und D_i verändern somit die jeweils korrespondierende Variable.

$Pwd = \text{KeyfileApply}(Pwd, file)$:
  if (filesize($file$) > 1024·1024)
    then $max = 1024 \cdot 1024$
    else $max = \text{filesize}(file)$
  end if
  $D = \text{readfile}(file, max)$
  for $(i = 0, 1, \ldots, max-1)$
    $t = \text{CRC32}((D_0 \| \ldots \| D_i))$
    $Pwd_{4*i \,(mod\,64)} = Pwd_{4*i \,(mod\,64)} + t_0 \,(mod\, 2^8)$
    $Pwd_{4*i+1 \,(mod\,64)} = Pwd_{4*i+1 \,(mod\,64)} + t_1 \,(mod\, 2^8)$
    $Pwd_{4*i+2 \,(mod\,64)} = Pwd_{4*i+2 \,(mod\,64)} + t_2 \,(mod\, 2^8)$
    $Pwd_{4*i+3 \,(mod\,64)} = Pwd_{4*i+3 \,(mod\,64)} + t_3 \,(mod\, 2^8)$
  end for
  return Pwd

Implementiert in: `Keyfile::Apply()`, `KeyfilesApply()`.

4.3.3 Generierung eines neuen Headers

I_E : 192 Byte zusätzliche, kryptografisch nicht relevante Informationen, die im Volume Header gespeichert werden. Siehe: alle Daten aus Tabelle 2.1, mit Ausnahme des Schlüsselbereichs.

$H = \text{newheader}()$:
  $S = \text{RNG}(64)$
  $K_V = \text{RNG}(64)$
  $H = \text{genheader}(S, K_V, I_E)$
  return H

Implementiert in: `VolumeCreator::CreateVolume()`, `TCFormatVolume()/ReEncryptVolumeHeader()/ChangePwd()`.

4.3.4 Generierung eines Headers

I_P : 192 Byte Padding-Daten. Bemerkung: Benötigt, da der zur Speicherung des Schlüssels zur Verfügung stehende Datenbereich größer ist als die verwendeten Schlüssel.

t : 448 Byte temporärer Speicher. Bemerkung: Dies ist kleiner als der Speicherbereich einer XTS Data Unit in TrueCrypt, d.h. nur eine XTS Data Unit muß verschlüsselt werden.

Bemerkung: Wenn dieser Datenbereich nach der Verschlüsselung an den Salt angehängt wird, resultiert der Volume Header mit einer Größe von 512 Bytes.

K_H : Header-Schlüssel. Mit diesem Schlüssel wird der Header entschlüsselt.

$H = \text{genheader}(S, K_V, I_E)$:
  $Pwd = \text{readPasswordandKeyfiles}()$
  if (FDE)
    then $c = 1000$
    else $c = 2000$
  endif
  $K_H = \text{PBKDF2}(Pwd, S, c, 64)$
  $t = (I_E \| K_V \| I_P)$
  $t = \text{XTS-AES-Enc}(K_H, t, 0)$
  $H = (S \| t)$
  return H

Implementiert in: `VolumeHeader::Create()`, `CreateVolumeHeaderInMemory()`.

4.3.5 Öffnen eines Headers

Zunächst wird der Header H mit einer Länge von 512 Byte von der Festplatte gelesen.

Auf H kann zusätzlich über $(H_0 \| H_1 \| \ldots \| H_{511})$ zugegriffen werden, wobei alle H_i eine Größe von 1 Byte besitzen. Zugriffe auf H und H_i verändern somit die jeweils korrespondierende Variable.

$(K_V, I_E) = \text{openheader}(H)$:
  $S = (H_0 \| H_1 \| \ldots \| H_{63})$
  $Pwd = \text{readPasswordandKeyfiles}()$
  if (FDE)
    then $c = 1000$
    else $c = 2000$
  endif
  $K_H = \text{PBKDF2}(Pwd, S, c, 64)$
  $t = (H_{64} \| H_{65} \| \ldots \| H_{511})$
  $t = \text{XTS-AES-Dec}(K_H, t, 0)$
  $(I_E \| K_V \| I_P) = t$
  return (K_V, I_E)

Implementiert in: `Volume::Open()`, `OpenVolume()`

4.3.6 Erstellen eines Backup-Headers

$H = \text{backupheader}()$:
  $(K_V, I_E) = \text{openheader}(H)$
  $S = \text{RNG}(64)$
  $H = \text{genheader}(S, K_V, I_E)$
  return H

Implementiert in: `TextUserInterface::BackupVolumeHeader()/`

`GraphicUserInterface::BackupVolumeHeader()/Core::ReEncryptVolumeHeaderWithNewSalt(), BackupVolumeHeader()`

4.4 Verschlüsseln und Entschlüsseln von Daten

$Pos = 512 \cdot Sec + Off$: Lese- / Schreib-Position im Volume
Sec : Sektor
Off : Offset innerhalb eines Sektors

4.4.1 Verschlüsseln von Daten

Verschlüsseln von Daten P der Größe eines AES-Blocks (16 Byte) zum Chiffretext C für das TrueCrypt-Volume an Byte-Position $Pos \geq 0$ mit Volume-Schlüssel K_v.

$C = \text{TCEnc}(P, Pos, K_v)$:
  $Off = Pos \pmod{512}$
  $Sec = \lfloor Pos / 512 \rfloor$
  $C = \text{XTS-AES-blockEnc}(K_v, P, Sec, Off)$
  return C

Implementiert in: `EncryptionModeXTS::EncryptBuffer(), EncryptBufferXTS()`

4.4.2 Entschlüsseln von Daten

Entschlüsseln des Chiffretexts C im TrueCrypt-Volume an Byte-Position $Pos \geq 0$ mit der Größe eines AES-Blocks (16 Byte) zu Daten P mit dem Volume-Schlüssel K_v.

$P = \text{TCDec}(C, Pos, K_v)$:
  $Off = Pos \pmod{512}$
  $Sec = \lfloor Pos / 512 \rfloor$
  $P = \text{XTS-AES-blockDec}(K_v, C, Sec, Off)$
  return P

Implementiert in: `EncryptionModeXTS::EncryptBuffer(), DecryptBufferXTS()`

4.5 Pseudozufallszahlengenerator

Der Pseudozufallszahlengenerator wird für die Initialisierung eines neuen Volumes genutzt.

n : Anzahl gewünschter pseudozufälliger Daten in Bytes. $n/16$ entspricht der Anzahl der gewünschten mit AES-Blockgröße (128 Bit = 16 Bytes)
R : Ausgabe-Array für die pseudozufälligen Daten der Länge n Bytes
C : Zwischenspeicher für PRNG-Daten, 16 Byte lang.
Key : 64 Byte Schlüssel für XTS-AES-256.

$R = PRNG(n)$:
  $Key = RNG(64)$
  $C = 0$
  $for\ (i = 0, 1, 2, \ldots, n/16 - 1)$:
    $C = \text{XTS-AES-Enc}(Key, C, i)$
    $R_i = C$
  end for
  return R

Implementiert in: `VolumeCreator::CreationThread(), FormatFat()/FormatNoFs()`

5 Makroexpansion

In der AES Verschlüsselungs-Funktion wird folgender Aufruf von `round()` durchgeführt (auf relevante Element beschränkt):

```
AES_RETURN aes_encrypt(const unsigned char *in, unsigned char *out, const aes_encrypt_ctx cx[1
])
{   uint_32t        locals(b0, b1);
    const uint_32t  *kp;
    kp = cx->ks;
    state_in(b0, in, kp);
    round(fwd_rnd,  b1, b0, kp + 1 * N_COLS);
...
}
```

Hier ist `locals()` ein Makro, dass wie folgt definiert wird:

```
#if defined(ARRAYS)
#define locals(y,x)     x[4],y[4]
#else
#define locals(y,x)     x##0,x##1,x##2,x##3,y##0,y##1,y##2,y##3
#endif
```

In TrueCrypt ist ARRAYS definiert, somit gilt folgende Definition:

```
#define locals(y,x)     x[4],y[4]
```

`round()` ist ebenfalls ein Makro, das hier definiert wird:

```
#define round(rm,y,x,k) rm(y,x,k,0); rm(y,x,k,1); rm(y,x,k,2); rm(y,x,k,3)
```

Dann wird `fwd_rnd` expandiert, das an folgender Stelle definiert wird:

```
#if defined(FT4_SET)
#undef  dec_fmvars
#define fwd_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ four_tables(x,t_use(f,n),fwd_var,rf1,c))
#elif defined(FT1_SET)
#undef  dec_fmvars
#define fwd_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ one_table(x,upr,t_use(f,n),fwd_var,rf1,c))
#else
#define fwd_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ fwd_mcol(no_table(x,t_use(s,box),fwd_var,rf1,c)
))
#endif
```

In TrueCrypt ist FT4_SET definiert, somit gilt folgende Definition:

```
#define fwd_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ four_tables(x,t_use(f,n),fwd_var,rf1,c))
```

Das Makro

```
#define t_use(m, n) t_##m##n
```

expandiert zu `t_fn`

Und schließlich expandiert `four_tables` wie folgt:

```
#define four_tables(x, tab, vf, rf, c) (  tab[0][bval(vf(x,0,c),rf(0,c))] \
  ^ tab[1][bval(vf(x,1,c),rf(1,c))] \
  ^ tab[2][bval(vf(x,2,c),rf(2,c))] \
  ^ tab[3][bval(vf(x,3,c),rf(3,c))])
```

Dies sind Ausschnitte aus der Expansion der Makros, die Makroexpansion ist bei weitem noch unvollständig. Insgesamt wird `round()` in 322 Schritten expandiert, und je nachdem, welche `#defines` gesetzt sind, wird der AES mit unterschiedlichen Darstellungsweisen berechnet. Für die verschiedenen Optionen verweisen wir hier auf den Quelltext. Der resultierende, vom C Compiler zu verarbeitende Quelltext ergibt sich schließlich wie in Text 5.1 dargestellt.


```
(b1[0] = (kp + 1 * 4)[0] ^ ( t_fn[0][((uint_8t)((( 0 == 0 ? ( 0 == 0 ? b0[0] : 0 == 1 ? b0[1
      ] : 0 == 2 ? b0[2] : b0[3]) : 0 == 1 ? ( 0 == 0 ? b0[1] : 0 == 1 ? b0[2] : 0 == 2 ?
      b0[3] : b0[0]) : 0 == 2 ? ( 0 == 0 ? b0[2] : 0 == 1 ? b0[3] : 0 == 2 ? b0[0] : b0[1]
      ) : ( 0 == 0 ? b0[3] : 0 == 1 ? b0[0] : 0 == 2 ? b0[1] : b0[2])) >> (8 * ((0))))))] ^
      t_fn[1][((uint_8t)((( 1 == 0 ? ( 0 == 0 ? b0[0] : 0 == 1 ? b0[1] : 0 == 2 ? b0[2] :
      b0[3]) : 1 == 1 ? ( 0 == 0 ? b0[1] : 0 == 1 ? b0[2] : 0 == 2 ? b0[3] : b0[0]) : 1 == 2
      ? ( 0 == 0 ? b0[2] : 0 == 1 ? b0[3] : 0 == 2 ? b0[0] : b0[1]) : ( 0 == 0 ? b0[3] : 0
      == 1 ? b0[0] : 0 == 2 ? b0[1] : b0[2])) >> (8 * ((1))))))] ^ t_fn[2][((uint_8t)((( 2
      == 0 ? ( 0 == 0 ? b0[0] : 0 == 1 ? b0[1] : 0 == 2 ? b0[2] : b0[3]) : 2 == 1 ? ( 0 == 0
      ? b0[1] : 0 == 1 ? b0[2] : 0 == 2 ? b0[3] : b0[0]) : 2 == 2 ? ( 0 == 0 ? b0[2] : 0 ==
      1 ? b0[3] : 0 == 2 ? b0[0] : b0[1]) : ( 0 == 0 ? b0[3] : 0 == 1 ? b0[0] : 0 == 2 ?
      b0[1] : b0[2])) >> (8 * ((2))))))] ^ t_fn[3][((uint_8t)((( 3 == 0 ? ( 0 == 0 ? b0[0] :
      0 == 1 ? b0[1] : 0 == 2 ? b0[2] : b0[3]) : 3 == 1 ? ( 0 == 0 ? b0[1] : 0 == 1 ? b0[2
      ] : 0 == 2 ? b0[3] : b0[0]) : 3 == 2 ? ( 0 == 0 ? b0[2] : 0 == 1 ? b0[3] : 0 == 2 ?
      b0[0] : b0[1]) : ( 0 == 0 ? b0[3] : 0 == 1 ? b0[0] : 0 == 2 ? b0[1] : b0[2])) >> (8 *
      ((3))))))]));
(b1[1] = (kp + 1 * 4)[1] ^ ( t_fn[0][((uint_8t)((( 0 == 0 ? ( 1 == 0 ? b0[0] : 1 == 1 ? b0[1
      ] : 1 == 2 ? b0[2] : b0[3]) : 0 == 1 ? ( 1 == 0 ? b0[1] : 1 == 1 ? b0[2] : 1 == 2 ?
      b0[3] : b0[0]) : 0 == 2 ? ( 1 == 0 ? b0[2] : 1 == 1 ? b0[3] : 1 == 2 ? b0[0] : b0[1]
      ) : ( 1 == 0 ? b0[3] : 1 == 1 ? b0[0] : 1 == 2 ? b0[1] : b0[2])) >> (8 * ((0))))))] ^
      t_fn[1][((uint_8t)((( 1 == 0 ? ( 1 == 0 ? b0[0] : 1 == 1 ? b0[1] : 1 == 2 ? b0[2] :
      b0[3]) : 1 == 1 ? ( 1 == 0 ? b0[1] : 1 == 1 ? b0[2] : 1 == 2 ? b0[3] : b0[0]) : 1 == 2
      ? ( 1 == 0 ? b0[2] : 1 == 1 ? b0[3] : 1 == 2 ? b0[0] : b0[1]) : ( 1 == 0 ? b0[3] : 1
      == 1 ? b0[0] : 1 == 2 ? b0[1] : b0[2])) >> (8 * ((1))))))] ^ t_fn[2][((uint_8t)((( 2
      == 0 ? ( 1 == 0 ? b0[0] : 1 == 1 ? b0[1] : 1 == 2 ? b0[2] : b0[3]) : 2 == 1 ? ( 1 == 0
      ? b0[1] : 1 == 1 ? b0[2] : 1 == 2 ? b0[3] : b0[0]) : 2 == 2 ? ( 1 == 0 ? b0[2] : 1 ==
      1 ? b0[3] : 1 == 2 ? b0[0] : b0[1]) : ( 1 == 0 ? b0[3] : 1 == 1 ? b0[0] : 1 == 2 ?
      b0[1] : b0[2])) >> (8 * ((2))))))] ^ t_fn[3][((uint_8t)((( 3 == 0 ? ( 1 == 0 ? b0[0] :
      1 == 1 ? b0[1] : 1 == 2 ? b0[2] : b0[3]) : 3 == 1 ? ( 1 == 0 ? b0[1] : 1 == 1 ? b0[2
      ] : 1 == 2 ? b0[3] : b0[0]) : 3 == 2 ? ( 1 == 0 ? b0[2] : 1 == 1 ? b0[3] : 1 == 2 ?
      b0[0] : b0[1]) : ( 1 == 0 ? b0[3] : 1 == 1 ? b0[0] : 1 == 2 ? b0[1] : b0[2])) >> (8 *
      ((3))))))]));
(b1[2] = (kp + 1 * 4)[2] ^ ( t_fn[0][((uint_8t)((( 0 == 0 ? ( 2 == 0 ? b0[0] : 2 == 1 ? b0[1
      ] : 2 == 2 ? b0[2] : b0[3]) : 0 == 1 ? ( 2 == 0 ? b0[1] : 2 == 1 ? b0[2] : 2 == 2 ?
      b0[3] : b0[0]) : 0 == 2 ? ( 2 == 0 ? b0[2] : 2 == 1 ? b0[3] : 2 == 2 ? b0[0] : b0[1]
      ) : ( 2 == 0 ? b0[3] : 2 == 1 ? b0[0] : 2 == 2 ? b0[1] : b0[2])) >> (8 * ((0))))))] ^
      t_fn[1][((uint_8t)((( 1 == 0 ? ( 2 == 0 ? b0[0] : 2 == 1 ? b0[1] : 2 == 2 ? b0[2] :
      b0[3]) : 1 == 1 ? ( 2 == 0 ? b0[1] : 2 == 1 ? b0[2] : 2 == 2 ? b0[3] : b0[0]) : 1 == 2
      ? ( 2 == 0 ? b0[2] : 2 == 1 ? b0[3] : 2 == 2 ? b0[0] : b0[1]) : ( 2 == 0 ? b0[3] : 2
      == 1 ? b0[0] : 2 == 2 ? b0[1] : b0[2])) >> (8 * ((1))))))] ^ t_fn[2][((uint_8t)((( 2
      == 0 ? ( 2 == 0 ? b0[0] : 2 == 1 ? b0[1] : 2 == 2 ? b0[2] : b0[3]) : 2 == 1 ? ( 2 == 0
      ? b0[1] : 2 == 1 ? b0[2] : 2 == 2 ? b0[3] : b0[0]) : 2 == 2 ? ( 2 == 0 ? b0[2] : 2 ==
      1 ? b0[3] : 2 == 2 ? b0[0] : b0[1]) : ( 2 == 0 ? b0[3] : 2 == 1 ? b0[0] : 2 == 2 ?
      b0[1] : b0[2])) >> (8 * ((2))))))] ^ t_fn[3][((uint_8t)((( 3 == 0 ? ( 2 == 0 ? b0[0] :
      2 == 1 ? b0[1] : 2 == 2 ? b0[2] : b0[3]) : 3 == 1 ? ( 2 == 0 ? b0[1] : 2 == 1 ? b0[2
      ] : 2 == 2 ? b0[3] : b0[0]) : 3 == 2 ? ( 2 == 0 ? b0[2] : 2 == 1 ? b0[3] : 2 == 2 ?
      b0[0] : b0[1]) : ( 2 == 0 ? b0[3] : 2 == 1 ? b0[0] : 2 == 2 ? b0[1] : b0[2])) >> (8 *
      ((3))))))]));
(b1[3] = (kp + 1 * 4)[3] ^ ( t_fn[0][((uint_8t)((( 0 == 0 ? ( 3 == 0 ? b0[0] : 3 == 1 ? b0[1
      ] : 3 == 2 ? b0[2] : b0[3]) : 0 == 1 ? ( 3 == 0 ? b0[1] : 3 == 1 ? b0[2] : 3 == 2 ?
      b0[3] : b0[0]) : 0 == 2 ? ( 3 == 0 ? b0[2] : 3 == 1 ? b0[3] : 3 == 2 ? b0[0] : b0[1]
      ) : ( 3 == 0 ? b0[3] : 3 == 1 ? b0[0] : 3 == 2 ? b0[1] : b0[2])) >> (8 * ((0))))))] ^
      t_fn[1][((uint_8t)((( 1 == 0 ? ( 3 == 0 ? b0[0] : 3 == 1 ? b0[1] : 3 == 2 ? b0[2] :
      b0[3]) : 1 == 1 ? ( 3 == 0 ? b0[1] : 3 == 1 ? b0[2] : 3 == 2 ? b0[3] : b0[0]) : 1 == 2
      ? ( 3 == 0 ? b0[2] : 3 == 1 ? b0[3] : 3 == 2 ? b0[0] : b0[1]) : ( 3 == 0 ? b0[3] : 3
      == 1 ? b0[0] : 3 == 2 ? b0[1] : b0[2])) >> (8 * ((1))))))] ^ t_fn[2][((uint_8t)((( 2
      == 0 ? ( 3 == 0 ? b0[0] : 3 == 1 ? b0[1] : 3 == 2 ? b0[2] : b0[3]) : 2 == 1 ? ( 3 == 0
      ? b0[1] : 3 == 1 ? b0[2] : 3 == 2 ? b0[3] : b0[0]) : 2 == 2 ? ( 3 == 0 ? b0[2] : 3 ==
      1 ? b0[3] : 3 == 2 ? b0[0] : b0[1]) : ( 3 == 0 ? b0[3] : 3 == 1 ? b0[0] : 3 == 2 ?
      b0[1] : b0[2])) >> (8 * ((2))))))] ^ t_fn[3][((uint_8t)((( 3 == 0 ? ( 3 == 0 ? b0[0] :
      3 == 1 ? b0[1] : 3 == 2 ? b0[2] : b0[3]) : 3 == 1 ? ( 3 == 0 ? b0[1] : 3 == 1 ? b0[2
      ] : 3 == 2 ? b0[3] : b0[0]) : 3 == 2 ? ( 3 == 0 ? b0[2] : 3 == 1 ? b0[3] : 3 == 2 ?
      b0[0] : b0[1]) : ( 3 == 0 ? b0[3] : 3 == 1 ? b0[0] : 3 == 2 ? b0[1] : b0[2])) >> (8 *
      ((3))))))]));
```

Text 5.1: Expandiertes Makro `round(fwd_rnd,  b1, b0, kp + 1 * N_COLS)`

6 Schlüsselverarbeitende Funktionen Linux

ID	ruft	Funktionsname
1		AES_RETURN aes_encrypt_key256(const unsigned char *key, aes_encrypt_ctx cx[1])
2	1	void CipherAES::SetCipherKey (const byte *key)
3	2	void Cipher::SetKey (const ConstBufferPtr &key)
4	3	void EncryptionAlgorithm::SetKey (const ConstBufferPtr &key)
5	4, 66	void CoreBase::RandomizeEncryptionAlgorithmKey (shared_ptr <EncryptionAlgorithm> encryptionAlgorithm) const
6	5, 58, 30	void GraphicUserInterface::BackupVolumeHeaders (shared_ptr <VolumePath> volumePath) const
7	6, 10, 12, 26, 27, 37, 38, 40, 41, 43, 49, 56	bool UserInterface::ProcessCommandLine ()
8	7	bool GraphicUserInterface::OnInit ()
9	7	int TextUserInterface::OnRun()
10	20, 30, 5, 58	void TextUserInterface::BackupVolumeHeaders (shared_ptr <VolumePath> volumePath) const
11	5, 66, 4, 64, 14, 19, 61, 48	void VolumeCreator::CreateVolume (shared_ptr <VolumeCreationOptions> options)
12	30, 11, 20	void TextUserInterface::CreateVolume (shared_ptr <VolumeCreationOptions> options) const
13	11, 27, 30, 53	WizardFrame::WizardStep VolumeCreationWizard::ProcessPageChangeRequest (bool forward)
14	5, 57, 19	void VolumeCreator::CreationThread ()
15	4	void EncryptionTest::TestLegacyModes ()
16	4	void EncryptionTest::TestXts ()
17	4	void EncryptionTest::TestXtsAES ()
18	4	void EncryptionTestDialog::EncryptOrDecrypt (bool encrypt)
19	24	void Pkcs5Kdf::DeriveKey (const BufferPtr &key, const VolumePassword &password, const ConstBufferPtr &salt) const
20		shared_ptr <VolumePassword> TextUserInterface::AskPassword (const wxString &message, bool verify) const
21		void FavoriteVolume::ToMountOptions (MountOptions &options) const
22		void CoreUnix::MountAuxVolumeImage (const DirectoryPath &auxMountPoint, const MountOptions &options) const
23	48	void CoreLinux::MountVolumeNative (shared_ptr <Volume> volume, MountOptions &options, const DirectoryPath &auxMountPoint) const
24	70, 71	void Pkcs5HmacSha512::DeriveKey (const BufferPtr &key, const VolumePassword &password, const ConstBufferPtr &salt, int iterationCount) const
25	4, 64, 65, 66	bool VolumeHeader::Decrypt (const ConstBufferPtr &encryptedData, const VolumePassword &password, const Pkcs5KdfList &keyDerivationFunctions, const EncryptionAlgorithmList &encryptionAlgorithms, const EncryptionModeList &encryptionModes)
26	25, 30, 58, 48	void GraphicUserInterface::RestoreVolumeHeaders (shared_ptr <VolumePath> volumePath) const
27	20, 25, 58, 48	void TextUserInterface::RestoreVolumeHeaders (shared_ptr <VolumePath> volumePath) const
28	25, 29, 48	void Volume::Open (shared_ptr <File> volumeFile, shared_ptr <VolumePassword> password, shared_ptr <KeyfileList> keyfiles, VolumeProtection::Enum protection, shared_ptr <VolumePassword> protectionPassword, shared_ptr <KeyfileList> protectionKeyfiles, VolumeType::Enum volumeType, bool useBackupHeaders, bool partitionInSystemEncryptionScope

ID	ruft	Funktionsname
29	28	void Volume::Open (const VolumePath &volumePath, bool preserveTimestamps, shared_ptr <VolumePassword> password, shared_ptr <KeyfileList> keyfiles, VolumeProtection::Enum protection, shared_ptr <VolumePassword> protectionPassword, shared_ptr <KeyfileList> protectionKeyfiles, bool sharedAccessAllowed, VolumeType::Enum volumeType, bool useBackupHeaders, bool partitionInSystemEncryptionScope)
30	29	shared_ptr <Volume> CoreBase::OpenVolume (shared_ptr <VolumePath> volumePath, bool preserveTimestamps, shared_ptr <VolumePassword> password, shared_ptr <KeyfileList> keyfiles, VolumeProtection::Enum protection, shared_ptr <VolumePassword> protectionPassword, shared_ptr <KeyfileList> protectionKeyfiles, bool sharedAccessAllowed, VolumeType::Enum volumeType, bool useBackupHeaders, bool partitionInSystemEncryptionScope) const
31	30, 60, 48	void CoreBase::ChangePassword (shared_ptr <VolumePath> volumePath, bool preserveTimestamps, shared_ptr <VolumePassword> password, shared_ptr <KeyfileList> keyfiles, shared_ptr <VolumePassword> newPassword, shared_ptr <KeyfileList> newKeyfiles, shared_ptr <Pkcs5Kdf> newPkcs5Kdf) const
32	30, 22, 23	shared_ptr <VolumeInfo> CoreUnix::MountVolume (MountOptions &options)
33	32	void CoreService::ProcessRequests (int inputFD, int outputFD)
34	33	void CoreService::ProcessElevatedRequests ()
35	34, 36	int main (int argc, char **argv)
36	33	void CoreService::Start ()
37	32	VolumeInfoList UserInterface::MountAllDeviceHostedVolumes (MountOptions &options) const
38	37	VolumeInfoList GraphicUserInterface::MountAllDeviceHostedVolumes (MountOptions &options) const
39	38	void MainFrame::MountAllDevices ()
40	20, 37	VolumeInfoList TextUserInterface::MountAllDeviceHostedVolumes (MountOptions &options) const
41	32, 49, 43, 21	VolumeInfoList UserInterface::MountAllFavoriteVolumes (MountOptions &options)
42	41	void MainFrame::MountAllFavorites ()
43	32	shared_ptr <VolumeInfo> UserInterface::MountVolume (MountOptions &options) const
44	43	shared_ptr <VolumeInfo> GraphicUserInterface::MountVolume (MountOptions &options) const
45	44	void MainFrame::MountVolume ()
46	45	void OnMountVolumeMenuItemSelected (wxCommandEvent& event) { MountVolume(); }
47		void Keyfile::Apply (const BufferPtr &pool) const
48	47	shared_ptr <VolumePassword> Keyfile::ApplyListToPassword (shared_ptr <KeyfileList> keyfiles, shared_ptr <VolumePassword> password)
49	20, 43	shared_ptr <VolumeInfo> TextUserInterface::MountVolume (MountOptions &options) const
50	32	WizardPage *VolumeCreationWizard::GetPage (WizardStep step)
51	50	void WizardFrame::SetStep (WizardStep newStep, bool forward)
52	51	void WizardFrame::SetStep (WizardStep newStep)
53	32, 52	void VolumeCreationWizard::OnVolumeCreatorFinished ()
54	53	void VolumeCreationWizard::OnProgressTimer ()
55	60	void ChangePasswordDialog::OnOKButtonClick (wxCommandEvent& event)

ID	ruft	Funktionsname
56	30, 60, 20	void TextUserInterface::ChangePassword (shared_ptr <VolumePath> volumePath, shared_ptr <VolumePassword> password, shared_ptr <KeyfileList> keyfiles, shared_ptr <VolumePassword> newPassword, shared_ptr <KeyfileList> newKeyfiles, shared_ptr <Hash> newHash) const
57	4, 64, 66	void VolumeHeader::EncryptNew (const BufferPtr &newHeaderBuffer, const ConstBufferPtr &newSalt, const ConstBufferPtr &newHeaderKey, shared_ptr <Pkcs5Kdf> newPkcs5Kdf)
58	57, 48	void CoreBase::ReEncryptVolumeHeaderWithNewSalt (const BufferPtr &newHeaderBuffer, shared_ptr <VolumeHeader> header, shared_ptr <VolumePassword> password, shared_ptr <KeyfileList> keyfiles) const
59	57	void Volume::ReEncryptHeader (bool backupHeader, const ConstBufferPtr &newSalt, const ConstBufferPtr &newHeaderKey, shared_ptr <Pkcs5Kdf> newPkcs5Kdf)
60	59, 19, 20	void CoreBase::ChangePassword (shared_ptr <Volume> openVolume, shared_ptr <VolumePassword> newPassword, shared_ptr <KeyfileList> newKeyfiles, shared_ptr <Pkcs5Kdf> newPkcs5Kdf) const
61	57,64	void VolumeHeader::Create (const BufferPtr &headerBuffer, VolumeHeaderCreationOptions &options)
62	4	void EncryptionModeXTS::SetSecondaryCipherKeys ()
63	62	void EncryptionModeXTS::SetCiphers (const CipherList &ciphers)
64	63	void EncryptionAlgorithm::SetMode (shared_ptr <EncryptionMode> mode)
65	64, 66, 4	bool VolumeHeader::Deserialize (const ConstBufferPtr &header, shared_ptr <EncryptionAlgorithm> &ea, shared_ptr <EncryptionMode> &mode)
66	62	void EncryptionModeXTS::SetKey (const ConstBufferPtr &key)
67	4	void EncryptionTest::TestCiphers ()
68	4	static void TestCipher (Cipher &cipher, const CipherTestVector *testVector, size_t testVectorCount)
69	72	derive_u_sha512 (pwd, pwd_len, salt, salt_len, iterations, u, b);
70	69	void derive_key_sha512 (char *pwd, int pwd_len, char *salt, int salt_len, int iterations, char *dk, int dklen)
71		void Pkcs5Kdf::ValidateParameters (const BufferPtr &key, const VolumePassword &password, const ConstBufferPtr &salt, int iterationCount) const
72		void hmac_sha512(char *k, int lk, char *d, int ld, char *out, int t)

7 Ergebnisse Zufallszahlentest

7.1 Linux

```
#=============================================================#
#          dieharder version 3.29.4beta Copyright 2003 Robert G. Brown      #
#=============================================================#
   rng_name    |           filename             |rands/second|
       mt19937|                rnddata-linux.txt|  4.85e+07  |
#=============================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================#
    diehard_birthdays|  0|       100|     100|0.61152636|  PASSED
       diehard_operm5|  5|   1000000|     100|0.15366634|  PASSED
   diehard_rank_32x32|  0|     40000|     100|0.28487112|  PASSED
     diehard_rank_6x8|  0|    100000|     100|0.26173492|  PASSED
    diehard_bitstream|  0|   2097152|     100|0.64591561|  PASSED
         diehard_opso|  0|   2097152|     100|0.65876193|  PASSED
         diehard_oqso|  0|   2097152|     100|0.07346781|  PASSED
          diehard_dna|  0|   2097152|     100|0.21572522|  PASSED
diehard_count_1s_str|  0|    256000|     100|0.31747927|  PASSED
diehard_count_1s_byt|  0|    256000|     100|0.69622284|  PASSED
 diehard_parking_lot|  0|     12000|     100|0.95542793|  PASSED
    diehard_2dsphere|  2|      8000|     100|0.28080108|  PASSED
    diehard_3dsphere|  3|      4000|     100|0.87178678|  PASSED
     diehard_squeeze|  0|    100000|     100|0.50686363|  PASSED
        diehard_sums|  0|       100|     100|0.00151380|   WEAK
        diehard_runs|  0|    100000|     100|0.64318812|  PASSED
        diehard_runs|  0|    100000|     100|0.52461963|  PASSED
      diehard_craps|  0|    200000|     100|0.61617538|  PASSED
      diehard_craps|  0|    200000|     100|0.97640767|  PASSED
 marsaglia_tsang_gcd|  0|  10000000|     100|0.42207304|  PASSED
 marsaglia_tsang_gcd|  0|  10000000|     100|0.42009107|  PASSED
         sts_monobit|  1|    100000|     100|0.55063994|  PASSED
            sts_runs|  2|    100000|     100|0.57454672|  PASSED
          sts_serial|  1|    100000|     100|0.37410514|  PASSED
          sts_serial|  2|    100000|     100|0.00070498|   WEAK
          sts_serial|  3|    100000|     100|0.02017541|  PASSED
          sts_serial|  3|    100000|     100|0.13919817|  PASSED
          sts_serial|  4|    100000|     100|0.12411560|  PASSED
          sts_serial|  4|    100000|     100|0.88330485|  PASSED
          sts_serial|  5|    100000|     100|0.50708990|  PASSED
          sts_serial|  5|    100000|     100|0.97840754|  PASSED
```



```
          sts_serial|  6|    100000|     100|0.35264433|  PASSED
          sts_serial|  6|    100000|     100|0.14751585|  PASSED
          sts_serial|  7|    100000|     100|0.38867639|  PASSED
          sts_serial|  7|    100000|     100|0.46288072|  PASSED
          sts_serial|  8|    100000|     100|0.59623603|  PASSED
          sts_serial|  8|    100000|     100|0.02572512|  PASSED
          sts_serial|  9|    100000|     100|0.66951936|  PASSED
          sts_serial|  9|    100000|     100|0.17823232|  PASSED
          sts_serial| 10|    100000|     100|0.81192881|  PASSED
          sts_serial| 10|    100000|     100|0.99277456|  PASSED
          sts_serial| 11|    100000|     100|0.54746459|  PASSED
          sts_serial| 11|    100000|     100|0.22490202|  PASSED
          sts_serial| 12|    100000|     100|0.66199187|  PASSED
          sts_serial| 12|    100000|     100|0.65496350|  PASSED
          sts_serial| 13|    100000|     100|0.96895514|  PASSED
          sts_serial| 13|    100000|     100|0.99625477|   WEAK
          sts_serial| 14|    100000|     100|0.63088615|  PASSED
          sts_serial| 14|    100000|     100|0.88963560|  PASSED
          sts_serial| 15|    100000|     100|0.81472805|  PASSED
          sts_serial| 15|    100000|     100|0.65041661|  PASSED
          sts_serial| 16|    100000|     100|0.55579354|  PASSED
          sts_serial| 16|    100000|     100|0.94033206|  PASSED
           rgb_bitdist|  1|    100000|     100|0.12162856|  PASSED
           rgb_bitdist|  2|    100000|     100|0.10909540|  PASSED
           rgb_bitdist|  3|    100000|     100|0.02846925|  PASSED
           rgb_bitdist|  4|    100000|     100|0.50237803|  PASSED
           rgb_bitdist|  5|    100000|     100|0.46069109|  PASSED
           rgb_bitdist|  6|    100000|     100|0.54586463|  PASSED
           rgb_bitdist|  7|    100000|     100|0.32735124|  PASSED
           rgb_bitdist|  8|    100000|     100|0.85894577|  PASSED
           rgb_bitdist|  9|    100000|     100|0.89058156|  PASSED
           rgb_bitdist| 10|    100000|     100|0.54904559|  PASSED
           rgb_bitdist| 11|    100000|     100|0.31790813|  PASSED
           rgb_bitdist| 12|    100000|     100|0.73674389|  PASSED
rgb_minimum_distance|  2|     10000|    1000|0.20164083|  PASSED
rgb_minimum_distance|  3|     10000|    1000|0.41707827|  PASSED
rgb_minimum_distance|  4|     10000|    1000|0.20119899|  PASSED
rgb_minimum_distance|  5|     10000|    1000|0.69505547|  PASSED
     rgb_permutations|  2|    100000|     100|0.28910875|  PASSED
     rgb_permutations|  3|    100000|     100|0.44061443|  PASSED
     rgb_permutations|  4|    100000|     100|0.37100759|  PASSED
     rgb_permutations|  5|    100000|     100|0.39597323|  PASSED
       rgb_lagged_sum|  0|   1000000|     100|0.95471099|  PASSED
       rgb_lagged_sum|  1|   1000000|     100|0.16221012|  PASSED
```

```
rgb_lagged_sum|  2| 1000000|     100|0.33839626|  PASSED
rgb_lagged_sum|  3| 1000000|     100|0.86224934|  PASSED
rgb_lagged_sum|  4| 1000000|     100|0.23685657|  PASSED
rgb_lagged_sum|  5| 1000000|     100|0.61649143|  PASSED
rgb_lagged_sum|  6| 1000000|     100|0.41138088|  PASSED
rgb_lagged_sum|  7| 1000000|     100|0.42465666|  PASSED
rgb_lagged_sum|  8| 1000000|     100|0.50623970|  PASSED
rgb_lagged_sum|  9| 1000000|     100|0.47656665|  PASSED
rgb_lagged_sum| 10| 1000000|     100|0.83371148|  PASSED
rgb_lagged_sum| 11| 1000000|     100|0.83048839|  PASSED
rgb_lagged_sum| 12| 1000000|     100|0.84021725|  PASSED
rgb_lagged_sum| 13| 1000000|     100|0.11856788|  PASSED
rgb_lagged_sum| 14| 1000000|     100|0.72407324|  PASSED
rgb_lagged_sum| 15| 1000000|     100|0.35079339|  PASSED
rgb_lagged_sum| 16| 1000000|     100|0.10628962|  PASSED
rgb_lagged_sum| 17| 1000000|     100|0.95097049|  PASSED
rgb_lagged_sum| 18| 1000000|     100|0.01838061|  PASSED
rgb_lagged_sum| 19| 1000000|     100|0.04617444|  PASSED
rgb_lagged_sum| 20| 1000000|     100|0.36573493|  PASSED
rgb_lagged_sum| 21| 1000000|     100|0.15023610|  PASSED
rgb_lagged_sum| 22| 1000000|     100|0.66422633|  PASSED
rgb_lagged_sum| 23| 1000000|     100|0.64397251|  PASSED
rgb_lagged_sum| 24| 1000000|     100|0.95301172|  PASSED
rgb_lagged_sum| 25| 1000000|     100|0.66126938|  PASSED
rgb_lagged_sum| 26| 1000000|     100|0.48005684|  PASSED
rgb_lagged_sum| 27| 1000000|     100|0.01337962|  PASSED
rgb_lagged_sum| 28| 1000000|     100|0.28523687|  PASSED
rgb_lagged_sum| 29| 1000000|     100|0.76692790|  PASSED
rgb_lagged_sum| 30| 1000000|     100|0.81067958|  PASSED
rgb_lagged_sum| 31| 1000000|     100|0.30917200|  PASSED
rgb_lagged_sum| 32| 1000000|     100|0.69453544|  PASSED
rgb_kstest_test| 34|   10000|    1000|0.37469552|  PASSED
```


VS – NUR FÜR DEN DIENSTGEBRAUCH

7.2 Windows

```
#=============================================================================#
#        dieharder version 3.29.4beta Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
       mt19937|            rnddata-windows.txt|  6.02e+07  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|  0|       100|     100|0.15496536|  PASSED
      diehard_operm5|  5|   1000000|     100|0.60917223|  PASSED
  diehard_rank_32x32|  0|     40000|     100|0.45116823|  PASSED
    diehard_rank_6x8|  0|    100000|     100|0.46273756|  PASSED
   diehard_bitstream|  0|   2097152|     100|0.70032968|  PASSED
        diehard_opso|  0|   2097152|     100|0.76252604|  PASSED
        diehard_oqso|  0|   2097152|     100|0.20616559|  PASSED
         diehard_dna|  0|   2097152|     100|0.02775301|  PASSED
diehard_count_1s_str|  0|    256000|     100|0.22104274|  PASSED
diehard_count_1s_byt|  0|    256000|     100|0.88748908|  PASSED
 diehard_parking_lot|  0|     12000|     100|0.01918028|  PASSED
    diehard_2dsphere|  2|      8000|     100|0.20258183|  PASSED
    diehard_3dsphere|  3|      4000|     100|0.69876974|  PASSED
     diehard_squeeze|  0|    100000|     100|0.26971247|  PASSED
        diehard_sums|  0|       100|     100|0.09301586|  PASSED
        diehard_runs|  0|    100000|     100|0.79483188|  PASSED
        diehard_runs|  0|    100000|     100|0.38214689|  PASSED
      diehard_craps|  0|    200000|     100|0.76257308|  PASSED
      diehard_craps|  0|    200000|     100|0.35483283|  PASSED
 marsaglia_tsang_gcd|  0|  10000000|     100|0.98473365|  PASSED
 marsaglia_tsang_gcd|  0|  10000000|     100|0.74776501|  PASSED
         sts_monobit|  1|    100000|     100|0.39305798|  PASSED
            sts_runs|  2|    100000|     100|0.69729173|  PASSED
          sts_serial|  1|    100000|     100|0.72466083|  PASSED
          sts_serial|  2|    100000|     100|0.88747081|  PASSED
          sts_serial|  3|    100000|     100|0.55055625|  PASSED
          sts_serial|  3|    100000|     100|0.33711410|  PASSED
          sts_serial|  4|    100000|     100|0.63242519|  PASSED
          sts_serial|  4|    100000|     100|0.40552313|  PASSED
          sts_serial|  5|    100000|     100|0.65127518|  PASSED
          sts_serial|  5|    100000|     100|0.97038934|  PASSED
          sts_serial|  6|    100000|     100|0.07364636|  PASSED
          sts_serial|  6|    100000|     100|0.24867564|  PASSED
```

sts_serial	7	100000	100|0.66054627	PASSED
sts_serial	7	100000	100|0.15257945	PASSED
sts_serial	8	100000	100|0.96762867	PASSED
sts_serial	8	100000	100|0.87000272	PASSED
sts_serial	9	100000	100|0.53784691	PASSED
sts_serial	9	100000	100|0.90155334	PASSED
sts_serial	10	100000	100|0.24583441	PASSED
sts_serial	10	100000	100|0.40729527	PASSED
sts_serial	11	100000	100|0.20209435	PASSED
sts_serial	11	100000	100|0.80027968	PASSED
sts_serial	12	100000	100|0.07382532	PASSED
sts_serial	12	100000	100|0.22404509	PASSED
sts_serial	13	100000	100|0.33128300	PASSED
sts_serial	13	100000	100|0.71498603	PASSED
sts_serial	14	100000	100|0.18870893	PASSED
sts_serial	14	100000	100|0.41488177	PASSED
sts_serial	15	100000	100|0.13344070	PASSED
sts_serial	15	100000	100|0.30664819	PASSED
sts_serial	16	100000	100|0.48075939	PASSED
sts_serial	16	100000	100|0.54853059	PASSED
rgb_bitdist	1	100000	100|0.84940300	PASSED
rgb_bitdist	2	100000	100|0.01848302	PASSED
rgb_bitdist	3	100000	100|0.21153539	PASSED
rgb_bitdist	4	100000	100|0.54591084	PASSED
rgb_bitdist	5	100000	100|0.69633977	PASSED
rgb_bitdist	6	100000	100|0.91491904	PASSED
rgb_bitdist	7	100000	100|0.95599644	PASSED
rgb_bitdist	8	100000	100|0.94132929	PASSED
rgb_bitdist	9	100000	100|0.86657295	PASSED
rgb_bitdist	10	100000	100|0.63998709	PASSED
rgb_bitdist	11	100000	100|0.35330877	PASSED
rgb_bitdist	12	100000	100|0.24141182	PASSED
rgb_minimum_distance	2	10000	1000|0.30393249	PASSED
rgb_minimum_distance	3	10000	1000|0.39645980	PASSED
rgb_minimum_distance	4	10000	1000|0.34183476	PASSED
rgb_minimum_distance	5	10000	1000|0.68736960	PASSED
rgb_permutations	2	100000	100|0.49365221	PASSED
rgb_permutations	3	100000	100|0.53930310	PASSED
rgb_permutations	4	100000	100|0.79707783	PASSED
rgb_permutations	5	100000	100|0.56691882	PASSED
rgb_lagged_sum	0	1000000	100|0.13180741	PASSED
rgb_lagged_sum	1	1000000	100|0.97209765	PASSED
rgb_lagged_sum	2	1000000	100|0.20188789	PASSED
rgb_lagged_sum	3	1000000	100|0.69964489	PASSED

rgb_lagged_sum	4	1000000	100|0.94688759	PASSED
rgb_lagged_sum	5	1000000	100|0.94986888	PASSED
rgb_lagged_sum	6	1000000	100|0.86133682	PASSED
rgb_lagged_sum	7	1000000	100|0.25144461	PASSED
rgb_lagged_sum	8	1000000	100|0.85869529	PASSED
rgb_lagged_sum	9	1000000	100|0.75268546	PASSED
rgb_lagged_sum	10	1000000	100|0.35813421	PASSED
rgb_lagged_sum	11	1000000	100|0.70181889	PASSED
rgb_lagged_sum	12	1000000	100|0.06809806	PASSED
rgb_lagged_sum	13	1000000	100|0.01667634	PASSED
rgb_lagged_sum	14	1000000	100|0.56568339	PASSED
rgb_lagged_sum	15	1000000	100|0.57681897	PASSED
rgb_lagged_sum	16	1000000	100|0.87032423	PASSED
rgb_lagged_sum	17	1000000	100|0.07485213	PASSED
rgb_lagged_sum	18	1000000	100|0.98142206	PASSED
rgb_lagged_sum	19	1000000	100|0.32014612	PASSED
rgb_lagged_sum	20	1000000	100|0.32868372	PASSED
rgb_lagged_sum	21	1000000	100|0.16516644	PASSED
rgb_lagged_sum	22	1000000	100|0.98132622	PASSED
rgb_lagged_sum	23	1000000	100|0.78177375	PASSED
rgb_lagged_sum	24	1000000	100|0.88033051	PASSED
rgb_lagged_sum	25	1000000	100|0.21001742	PASSED
rgb_lagged_sum	26	1000000	100|0.21525636	PASSED
rgb_lagged_sum	27	1000000	100|0.40385238	PASSED
rgb_lagged_sum	28	1000000	100|0.88546433	PASSED
rgb_lagged_sum	29	1000000	100|0.64523333	PASSED
rgb_lagged_sum	30	1000000	100|0.05506138	PASSED
rgb_lagged_sum	31	1000000	100|0.27876346	PASSED
rgb_lagged_sum	32	1000000	100|0.88956245	PASSED
rgb_kstest_test	32	10000	1000|0.18146671	PASSED

Bibliografie

1: Gladman, Brian, A Specification for Rijndael, the AES Algorithm,
2: ISO/IEC, Programming languages -- C, 1999
3: ISO/IEC, Programming Languages -- C++, 1998
4: Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual,
5: FIPS, Anouncing the ADVANCED ENCRYPTION STANDARD (AES),
6: Biryukov, Alex; Khovratovich, Dmitry, Related-key Cryptanalysis of the Full AES-192 and AES-256,
7: Bernstein, Daniel J., Cache-timing attacks on AES, 2005
8: , Argument Passing and Naming Conventions: __cdecl, , http://msdn.microsoft.com/en-us/library/zkwh89ks.aspx
9: , Calling Convention, , http://msdn.microsoft.com/en-us/library/9b372w95.aspx
10: , System V Application Binary Interface, AMD64 Architecture Processor Supplement,
11: Gueron, Shay, White Paper: Intel Advanced Encryption Standard (AES) Instructions Set, 2010
12: IEEE, IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices, 2007
13: Dworkin, Morris, Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices, 2010
14: , Secure Hash Standard (SHS), 2008
15: Dobbertin, Hans; Bosselaers, Antoon; Preneel, Bart, RIPEMD-160: A Strengthened Version of RIPEMD, 1996
16: Rivest, R.L., The MD4 Message Digest Algorithm, 1990
17: TrueCrypt Foundation, Keyfiles,
18: wxWidget Developers, wxWidgets 2.6 wxTextCtrl,
19: RSA Laboratories, PKCS #5 v2.0: Password-Based Cryptography Standard, 1999
20: FIPS, The Keyed-Hash Message Authentication Code (HMAC),
21: , TrueCrypt Manual,
22: , RtlSecureZeroMemory, ,
23: TrueCrypt Foundation, Program Menu,
24: TrueCrypt Foundation, Portable Mode,
25: TrueCrypt Foundation, Hidden Volume Protection,
26: Microsoft Corporation, Microsoft Extensible Firmware Initiative FAT32 File System Specification , 2000
27: TrueCrypt Foundation, Hidden Operating System,

Funktionen und Klassen