FsVerify

Xenia

Reflexion zum
AP Informatik

Inhaltsverzeichnis

1 Idee

Die Idee einer Dateisystemverifizierung ist nichts Neues; oft wird sie in embedded Geräten oder Handys implementiert, wo die Sicherheit und Integrität des Systems von großer Bedeutung sind. Jedoch ist sie bei Desktop-Betriebssystemen wie Windows, macOS oder Linux Distributionen nicht sehr herkömmlich. Dies liegt meist daran, dass eine effektive Verifizierung des Dateisystems sich darauf verlässt, dass das Dateisystem sich nicht über Zeit verändert, welches man bei Desktop-Betriebssystemen nicht gewährleisten kann, da Programme oft direkt in den Root des Betriebssystems schreiben (/usr in *nix, C:/Program Files in Windows).
Mittlerweile gibt es jedoch in der Linux-Welt eine neue Art von Distribution, die sogenannten 'Immutable' Distributionen, welche sich darin unterscheiden, dass der Root schreibgeschützt ist oder wie bei NixOS oder Gnu GUIX gar nicht erst richtig existiert. Dadurch können Programme nur direkt in den Homeordner des Nutzers installiert werden, und das eigentliche System bleibt original bestanden.

Hierdurch wird Dateisystem-Verifizierung möglich, da der originale Zustand sich nie ändern wird, kann ein Programm ohne Probleme die Partition verifizieren, dass sich nichts verändert hat.

1.1 Implementierungen

Für die Verifizierung eines Dateisystems gibt es verschiedene Methoden:

Per-Datei verifizierung

Bei der Per-Datei Verifizierung wird der Hash von jeder Datei die verifiziert werden soll mit einem vorbestimmten, vertrauten, Hash verglichen, falls der Hash übereinstimmt ist Datei unmodifiziert, wenn sie jedoch abweichen, ist die Datei modifiziert und kann nicht vertraut werden.

Festplattenverifizierung

Hier wird ein Hash von der ganzen Festplatte oder Partition mit einem vorgegebenen Wert verglichen. Im Vergleich zu der Per-Datei-Verifizierung werden hier auch neue Dateien erkannt, welche eine Per-Datei-Verifizierung ignoriert hätte. Jedoch kann dies auch erheblich langsamer sein, da die ganze Partition, welche sehr groß werden kann, in einem Thread gehasht wird.

Blockverifizierung

Dies ist ähnlich zu der Festplattenverifizierung, jedoch werden hier nur einzelne Blöcke gehasht und verifiziert. Dies ermöglicht es, die Verifizierung durch Multithreading zu beschleunigen, während man weiterhin die ganze Festplatte/Partition verifiziert.

Alle drei Arten der Verifizierung haben eine Sache gemeinsam, sie brauchen eine vertraute quelle, von der sie den korrekten Hash für eine Datei/Partition/Block lesen können.

1.2 Hash Quellen

Wie bereits gesagt, braucht das Verifizierungsprogramm eine vertraute Quelle für die korrekten Hashes. Hier gibt es auch verschiedene Lösungsansätze, was jedoch alle gemeinsam haben ist, dass sie eine Quelle und eine sichere Methode, um diese Quelle zu verifizieren, brauchen.
Für die Quellen gibt es viele verschiedene Möglichkeiten; bei der Entwicklung von fsverify hatte ich die Wahl auf zwei Möglichkeiten begrenzt, da beide sehr einfach zu implementieren sind und dadurch die Verifizierung der Quellen auch einfach ist.

Externe Partition

Hier wird eine Datenbank an Hashes zusammen mit allen Metadaten in eine extra Partition geschrieben; diese Partition kann auf ein externes Medium geschrieben werden und nur dann angeschlossen sein, wenn das System die Verifizierung durchführt. Jedoch braucht dies entweder eine separate Partition auf der Festplatte, wodurch die nutzbare Speicherkapazität sich verringert, oder ein externes Medium, welches nicht immer vorhanden ist.

Einfache Datei

Hier wird die Datenbank einfach in einem Ort gespeichert, auf den das Programm während der Verifizierung zugreifen kann. Dies ist sehr einfach zu implementieren und benötigt keine externen Partitionen oder Speichermedien. Das Problem ist es jedoch, die Datei an einem Ort zu speichern, bei der man nicht unverifizierte Dateisysteme anhängen muss oder ungeschützt ohne Schreibschutz offen ist.

Um die Quelle zu schützen beziehungsweise zu verifizieren, gibt es zwei Methoden:

Kryptographische Verifizierung

Die Entwickler des Betriebssystems müssen hierbei bei dem Aufsetzen des Verifizierungsprogramms die Hash Quelle kryptografisch mit ihren privaten Schlüsseln signieren (zum Beispiel mit GnuPG oder Minisgin), das Verifizierungsprogramm erhält den öffentlichen Schlüssel der Entwickler, die Signatur und die Quelle, wodurch es anhand der Signatur verifizieren kann, dass die Quelle von den Entwicklern stammt und nicht modifiziert wurde.

Hierbei ist das größte Problem, dass der öffentliche Schlüssel gut geschützt werden muss, damit die Signatur und Schlüssel nicht mit der eines Attackers ersetzt werden kann.

Verschlüsselung

Die Quelle ist mit einem zufällig generierten Schlüssel verschlüsselt, welcher in den Quellcode des Verifizierungsprogramms geschrieben wird, um somit den Schlüssel direkt im Programm zu speichern. Dadurch können keine Schlüssel ersetzt werden, jedoch ist es immer möglich, den Schlüssel aus dem Programm zu extrahieren, ohne überhaupt auf das System zugreifen zu müssen, da man das Betriebssystem selber installieren kann. Sobald der Schlüssel bekannt ist, kann die Datei einfach verschlüsselt und ohne Probleme modifiziert werden.

1.3 Gewählte Implementation

In Anbetracht existierender Dateiverifizierungsprogramme wie Androids dm-verity und mein vorheriges, ähnliches Projekt FsGuard.

Für die Implementation habe ich die Blockverifizierung ausgewählt, da sie durch Multithreading sehr schnell sein kann, aber auch neue Dateien bemerkt, welche die Per-Datei-Verifizierung nicht gewährleistet.

Um die Hashes zu speichern, wird ein eigenes Partitionsschema benutzt, welches alle Metadaten und die Datenbank beinhaltet. Der minisign öffentliche Schlüssel kann durch mehrere Methoden gespeichert werden, wie einer Textdatei oder einem Gerät, welches über USB-Serial den Schlüssel übergibt.

Weitere Entscheidungen für die Implementation sind:

- Programmiersprache: go
  go ist mir vertraut und Memory Safe, welches für die Sicherheit des Programmes eine große Rolle spielt.

- Datenbank: bbolt
  bbolt ist eine Datenbank, welche direkt in go geschrieben wurde und somit eine robustere API als sqlite hat; zudem ist bbolt unter einer richtigen Lizenz lizenziert und wirkt moderner.

2 Realisierung

Das Projekt kann in drei Unterprojekte eingeteilt werden. Fsverify, also die Verifizierung selber, verifysetup, ein Programm, um das System richtig zu konfigurieren, um die Nutzung von fsverify möglich zu machen und fbwarn, ein Programm, welches den Nutzer grafisch über eine fehlgeschlagene Verifizierung informiert.

2.1 fsverify

Da das Konzept der Festplattenverifizierung nichts Neues ist, habe ich mir erstmals bereits existierende Projekte angeschaut, um zu sehen, wie es in anderen Betriebssystemen realisiert ist. Hierbei war Google's dm-verity, welches in Android und ChromeOS Geräten genutzt wird, die beste Hilfe, da es am besten dokumentiert und ausgetestet ist.

2.1.1 Partitionslayout

Inspiriert an dm-verity, entschied ich mich dafür, die Datenbank auf eine eigene Partition zu speichern; also war das erste Ziel, ein gutes Partitionslayout zu entwickeln, in der die Datenbank und Metadaten so gut wie möglich gespeichert werden können.
Die erste Version des Layouts war recht simpel, es hatte alles, was wirklich wichtig war, eine magic number, die Signatur, Größe des Dateisystems und Größe der Datenbank:

```
<magic number> <signature> <filesystem size> <table size>
```

Feld	Größe	Nutzen	Wert
magic number	2 bytes	Sanity check	0xACAB
signature	302 bytes	minisign signatur	-
filesystem size	4 bytes	größe des originalen Dateisystems in GB	-
table size	4 bytes	größe der Datenbank in MB	-

In der Implementierung dieses Layouts fiel dann auf, dass es keinen Sinn macht, die Datenbankgröße in MB festzulegen. Die zweite Version fügt aus diesem Grund ein weiteres Feld hinzu, um die Einheit der Datenbankgröße festzulegen:

```
<magic number> <signature> <filesystem size> <table size> <table unit>
```

Feld	Größe	Nutzen	Wert
magic number	2 bytes	Sanity check	0xACAB
signature	302 bytes	minisign signatur	-
filesystem size	4 bytes	größe des originalen Dateisystems in GB	-
table size	4 bytes	größe der Datenbank in MB	-
table unit	1 byte	datentyp des Feld “table size”	-

Die nächste Version teilte die Signatur in zwei Teile auf. Da minisign Signaturen aus einem Kommentar, einer vertrauten Signatur, einem weiteren Kommentar und einer nicht vertrauten Signatur.

```
<magic number> <untrusted signature hash> <trusted signature hash>
<filesystem size> <table size> <table unit>
```

Feld	Größe	Nutzen	Wert
magic number	2 bytes	Sanity check	0xACAB
untrusted signature	100 bytes	nicht vertrauter signatur	-
trusted signature	88 bytes	vertraute signatur	-
filesystem size	4 bytes	größe des originalen Dateisystems in GB	-
table size	4 bytes	größe der Datenbank in MB	-
table unit	4 bytes	datentyp des Feld “table size”	-

2.1.2 Datenbanklayout

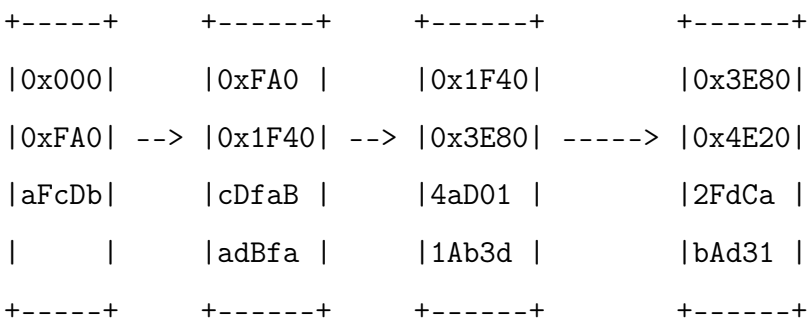
Nachdem der Header der Partition festgelegt wurde, muss festgelegt werden, wie die Datenbank festgelegt ist. bbolt, die Datenbankbibliothek, die fsverify nutzt, hat ein key/value System, das heißt, dass jeder Wert mit einem Schlüssel verbunden ist. Zudem benutzt bbolt das Konzept von "Buckets", einem Eimer, in dem Datenpaare sortiert werden können.

Das erste Layout war für eine Implementation von fsverify, die nur auf einem Thread läuft, besteht aus einem Bucket "Nodes", in dem jede Node gespeichert wird. Eine Node sieht wie folgt aus:

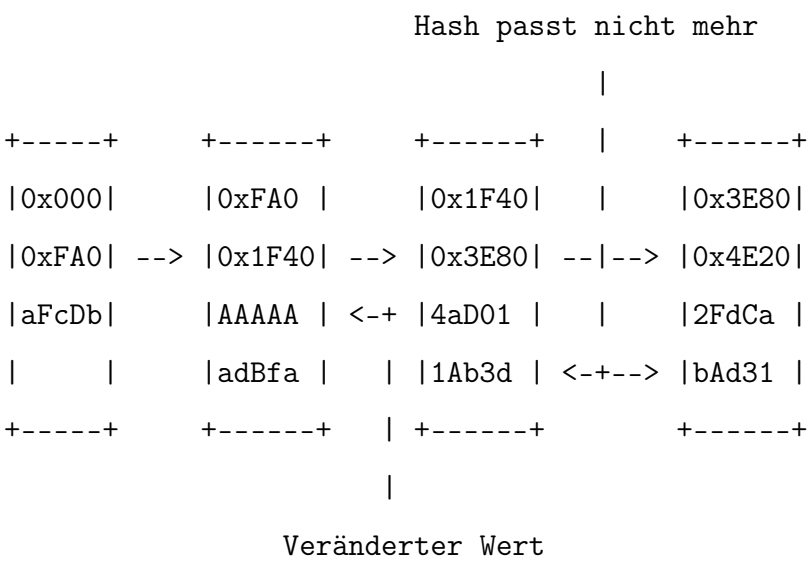
```
// Node.go
type Node struct {
  BlockStart int
  BlockEnd int
  BlockSum string
  PrevNodeSum string
}
```

Feld	Nutzen
BlockStart	Der hex offset and dem der Block anfängt
BlockEnd	Der hex offset and dem der Block ended
BlockSum	Der sha1 hash des Blocks
PrevBlockSum	Der sha1 hash aus allen Feldern der vorherigen Node

Jeder Block bekommt eine Node zugewiesen; diese Nodes werden in der Datenbank aneinandergereiht, mit dem Wert von PrevBlockSum als den Key. Der Wert PrevBlockSum erlaubt es, während der Verifizierung Fehler in der Datenbank zu finden. Wird eine Node verändert, stimmt der PrevBlockSum der nächsten Node nicht mehr, das heißt, dass es nicht mehr möglich ist, den Key zu der nächsten Node zu berechnen, wodurch die Verifizierung fehlschlägt.



Wird hier eine Node verändert, stimmt die restliche Kette nicht mehr.



Da die erste Node keinen Vorgänger hat, von dem es PrevNodeSum berechnen kann, wird ihr der Wert "Entrypoint" gegeben.

Diese Datenbankstruktur hat ohne Probleme funktioniert, jedoch war fsverify viel zu langsam, wenn es auf einem Thread läuft. Das Problem bei dem Multithreading jedoch ist, dass man Nodes nicht wahrlos aufgreifen kann, sondern eine vorherige Node oder die entrypoint Node braucht. Die Lösung ist recht einfach, die Anzahl der Threads wird in verifysetup bereits angegeben und somit in fsverify fest einprogrammiert. Somit gibt es in der Datenbank mehrere entrypoint Nodes, die sich durch eine hinzugefügte Nummer unterscheiden. Dadurch ist es weiterhin möglich, die Datenbank zu verifizieren, während es multithreaded läuft.

2.1.3 Datenbanksignatur

Um sicherzustellen, dass die Datenbank nicht modifiziert wurde, wird eine Signatur generiert, die mit der gelesenen Datenbank verglichen wird.

Wie bereits erwähnt, wird für die Signatur das Programm minisign benutzt. Minisign beruht auf einem public/private key System, wodurch eine Signatur von dem privaten Schlüssel generiert wird und durch den öffentlichen Schluss verifiziert werden kann.
Die Signatur wurde bereits im Partitionsheader gespeichert. Was übrig bleibt, ist der öffentliche Schlüssel.
Da der öffentliche Schlüssel und die Signatur gebraucht werden, um eine Datenbank zu verifizieren, muss sichergestellt werden, dass beide separat gespeichert werden und zumindest der öffentliche Schlüssel nicht bearbeitet werden kann.
Die erste Idee, um dies zu lösen, wäre, dass man einfach den Schlüssel in eine Datei schreibt und die Datei schreibgeschützt speichert. Jedoch ist bei diesem Weg der Speicherort der Datei das Problem. Wie soll man sicher sein, dass nicht das ganze Dateisystem verändert wurde, um einen neuen Schlüssel zu beinhalten?

Das heißt, dass man ein schreibgeschütztes, möglichst separates und immer vertrautes Speichermedium braucht, auf dem man den Schlüssel speichert.
Die Lösung: Mikrocontroller. Sie können über usb-serial (also /dev/ttyACM* in Linux) Daten übertragen, können durch das Modifizieren bestimmter Sektoren permanent schreibgeschützt werden und sind sehr klein, also können sie von dem Nutzer mitgetragen werden oder in dem Gerät direkt verbaut sein.
Um dieses Konzept zu testen, habe ich einen Arduino UNO genutzt. Dieser ist zwar immer schreibbar, hat aber keine technischen Unterschiede, die die Datenübertragung ändern würden.
Der Code für den Arudino sieht wie folgt aus:

```
// publicKey.c
void setup() {
  Serial.begin(9600); // set up a serial tty with the baud rate 9600
  Serial.print("\tpublic key\t"); // Write the public key to the tty
}
void loop() {}
```

Es wird eine serielle Konsole auf einer Baudrate von 9600 geöffnet, auf der einmalig der öffentliche Schlüssel ausgegeben wird. Es ist wichtig zu beachten, dass der Schlüssel mit Tabstopp (\ t) ausgegeben wird, diese benutzt fsverify um zu wissen, ob der volle

Schlüssel aufgenommen wird, fehlt der Tabstopp am Anfang oder am Ende, ist es sehr wahrscheinlich, dass der Schlüssel auch nicht vollständig aufgenommen wurde.

2.1.4 Optimierung

Wie bereits gesagt, lief die erste Version von fsverify auf einem Thread, dies führte zu einer Laufzeit von über einer Stunde bei einer Partitionsgröße von 1 GB. Da fsverify beim Booten des Systems ausgeführt werden soll, ist eine Laufzeit von einer Stunde nicht akzeptabel.
Die ersten Schritte der Optimierung war es, die Größe der Blocks zu verringern und von sha256 zu sha1 zu wechseln. Da das Lesen von daten viel schneller erfolgt als das hashen von daten, ist es besser mehr zu lesen und dadurch kleinere Datenmengen zu hashen, der wechsel von sha256 zu sha1 mag erstmal schlecht wirken, jedoch macht dies keine Probleme, da hier keine Passwörter oder ähnliches verschlüsselt werden, also sind Bruteforceattacken hier kein Risiko.
Mit diesen Optimierungen hat sich die Laufzeit etwas verbessert, von 60 Minuten zu ungefähr 50. Nicht viel besser.
Der nächste Schritt war es, fsverify mit Multithreading zu implementieren; die dafür notwendigen Änderungen in der Datenbank wurden bereits erklärt. In fsverify selber hat sich die Art geändert, wie die Daten von der Partition gelesen werden. Anstatt alles auf einmal zu lesen und durchzugehen, wird die Größe der Partition genommen, durch die Anzahl der Threads geteilt, und somit für jeden Thread genau die Anzahl an Bytes gelesen, die für die Node-kette nötig ist. Dies stellt sicher, dass keine Kette sich überlappt und korrupte Nodes in Ketten auffallen, da sie durch Korruption versuchen könnten, Bytes zu lesen, die sie gar nicht lesen sollten.
Durch das Multithreading hat sich die Laufzeit von den singlethreaded 50 Minuten zu nur 6 Sekunden verringert.

Fsverify hatte eine Laufzeitoptimierung von 60000% in einer Woche:

```
10.02.2024:
fsverify takes 60minutes to complete for 1gb
optimizations: none
```

```
12.02.2024:
fsverify takes 52minutes to complete for 1gb
optimizations: block size 2k, sha1 instead of sha256

17.02.2024:
fsverify takes ~6 seconds to complete for 1gb with 12 threads (p7530)
optimizations: block size 2k, sha1 instead of sha256,
               multithreaded, db batch operations
unoptimizations: manual connecting of arduino, ~1 second penalty
```

2.2 verifysetup

Nachdem fsverify vollständig implementiert war und alle Speicherkonzepte vollständig entwickelt sind, braucht fsverify auch ein Programm, um alles richtig aufzusetzen.
Das Programm muss eine Datenbank von Nodes anhand der zu verifizierenden Partition erstellen, den Header entsprechend konfigurieren und alles auf eine Datei schreiben, die der Nutzer (oder eher Distributions-Entwickler) auf die für fsverify vorgesehene Partition schreiben kann.

2.2.1 Optimierung

Genauso wie fsverify benutzt verifysetup erstmal nur einen Thread, um die Datenbank zu erstellen. Dies führte zu einer Laufzeit von über 2 Stunden für 1 GB.
Die Schritte zur Optimierung sind die gleichen wie bei fsverify. Jedoch verbesserte sich die Laufzeit um einiges, bereits bei dem Wechsel zu 2 KB Blocks und sha1 hashing, von 2 Stunden zu einer Stunde.
Mit dem Wechsel zu Multithreading ging dies dann runter zu 19 Sekunden mit 12 Threads.
Die Laufzeit von verifysetup verbesserte sich um 33846% in einer Woche.

```
10.02.2024:
fsverify setup takes 110minutes to complete for 1gb
optimizations: none

12.02.2024:
fsverify setup takes 71minutes to complete for 1gb
optimizations: block size 2k, sha1 instead of sha256
```

```
12.02.2024:
fsverify setup takes ~9.54 seconds to complete for 1gb with 12 threads
optimizations: block size 2k, sha1 instead of sha256,
               multithreaded, db batch operations

17.02.2024:
fsverify setup takes ~19.50 seconds to complete for 1gb with 12 threads
optimizations: block size 2k, sha1 instead of sha256,
               multithreaded, db batch operations
unoptimizations: enable database signing, header generation,
                 fsverify partition generation
```

2.3 fbwarn

Falls die Festplattenverifizierung fehlschlägt, muss der Nutzer gewarnt werden (es ist auch möglich, das Gerät einfach auszuschalten, jedoch sollte dies aus UX Gründen nicht gemacht werden).

2.3.1 Grafischer Output

Die Warnung ist am besten grafisch zu tun, jedoch gibt es keinen Display-Server wie Wayland oder X11, deshalb muss direkt auf den Framebuffer /dev/fb* geschrieben werden, um grafischen Output zu zeigen.
Da jeder verfügbare framebuffer als /dev/fbX verfügbar ist, kann man ganz einfach die Datei öffnen und mit mmap manipulieren:

```c
// imports: stdlib, linux/fb.h, stdio
int main()
{
    int fbfd = 0;
    struct fb_var_screeninfo vinfo;
    struct fb_fix_screeninfo finfo;

    // Open the framebuffer file for reading and writing
    fbfd = open("/dev/fb0", O_RDWR);

    // Get fixed screen information
```

```
    ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo);
    // Get variable screen information
    ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo);

    // Map the device to memory
    fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fbfd, 0);

    location = (300+vinfo.xoffset) * (vinfo.bits_per_pixel/8) + \
               (100+vinfo.yoffset) * finfo.line_length;
    *(fbp + location) = 100;         // Some blue
    *(fbp + location + 1) = 50;      // A little green
    *(fbp + location + 2) = 200;     // A lot of red
    *(fbp + location + 3) = 0;       // No transparency

    munmap(fbp, screensize);
    close(fbfd);
    return 0;
}
```

Dies genügt, wenn man einfache Formen in den Framebuffer zeichnen möchte, wie im Bild zu sehen ist.

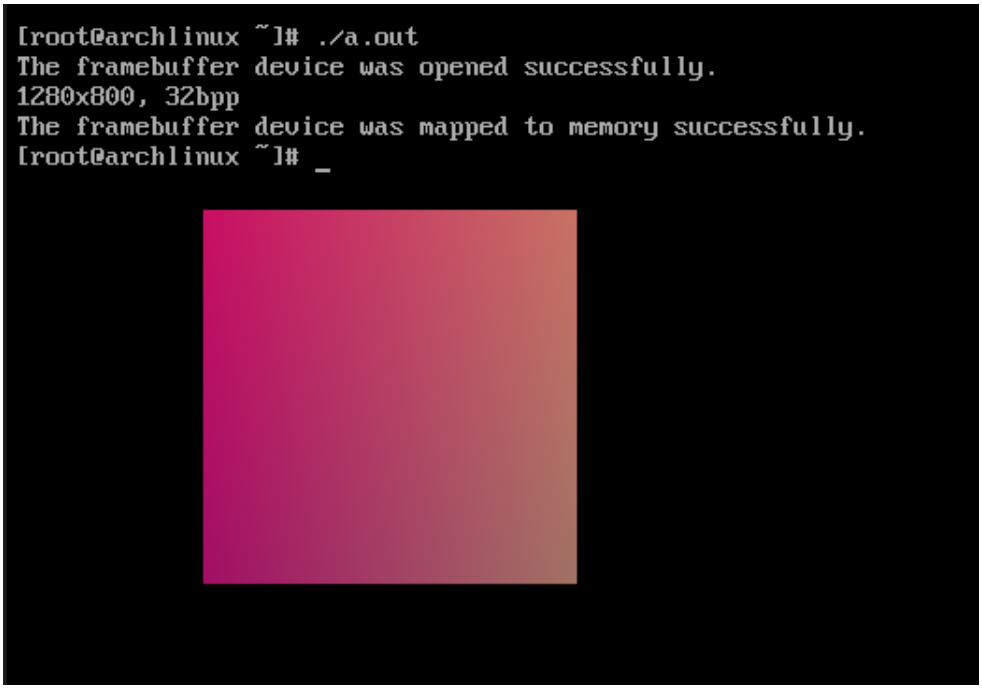


Abbildung 1: C Programm welches einen Quadrat mit Gradient direkt in den Framebuffer schreibt

Jedoch wird es komplizierter, wenn man auch Text anzeigen möchte, da man jedes Pixel manuell schreiben muss, welches bei Sätzen wie “System Verification Failed” bereits sehr umständlich ist.
Deshalb ist eine bessere Lösung nötig, eine Bibliothek, die in den Framebuffer schreiben kann und alle Rendering-Funktionen abstrahiert. Die Bibliothek, die ich benutzt habe, ist Raylib, eine C-Bibliothek, welche hauptsächlich für die Entwicklung von Spielen gedacht ist, jedoch, nicht wie andere Engines, keine besonderen Features hat, sondern lediglich verschiedene Aspekte wie Grafik, Physik und Audio abstrahiert. Glücklicherweise kann mit der Compilerflag `-DEGL_NO_X11` raylib so kompiliert werden, dass es direkt in den Framebuffer schreibt, anstatt versucht, ein Fenster zu öffnen.

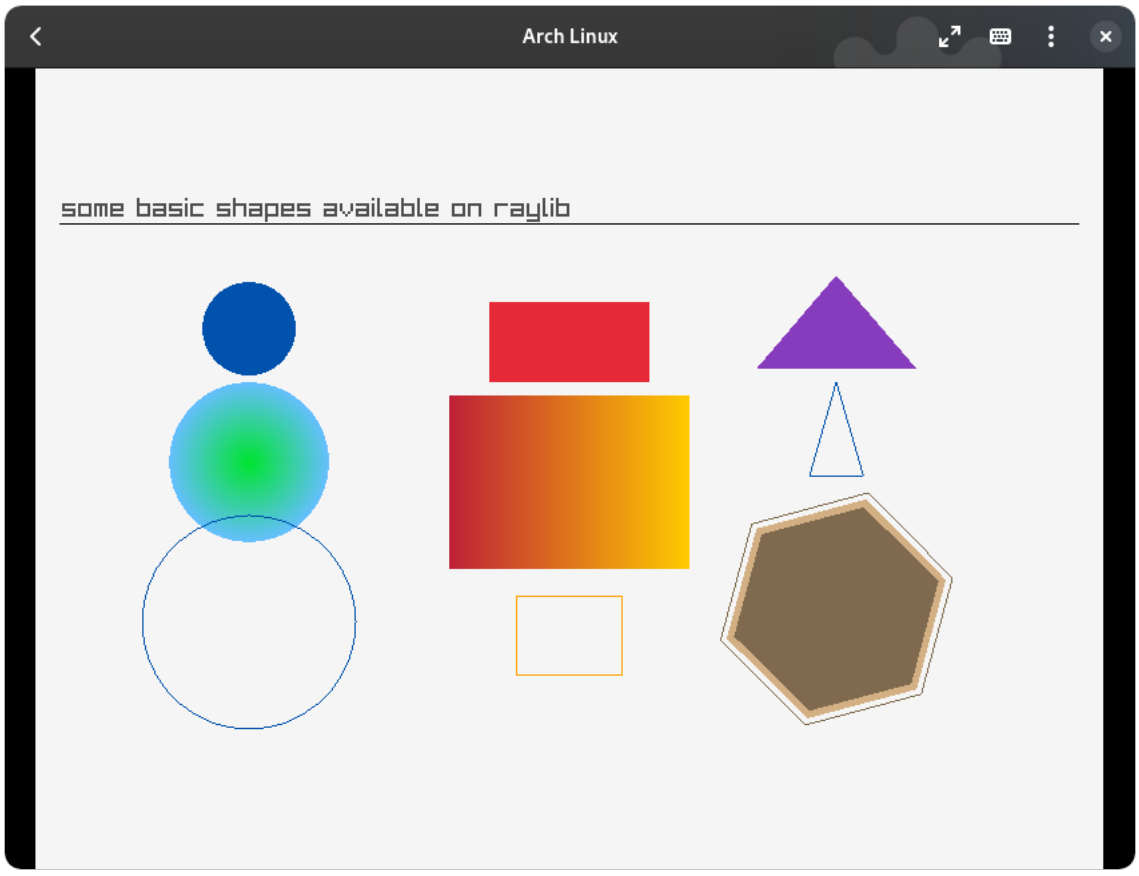


Abbildung 2: Raylib Beispielprogramm `shapes_basic_shapes` in einer VM mit output direkt zum Framebuffer

Hiermit wird es um einiges einfacher, eigene Bilder zu erstellen, die angezeigt werden, wenn nötig.

2.3.2 Grafikformat

Da es recht umständlich wär, das Bild immer manuell in C zu programmieren, ist es am besten, ein Grafikformat zu benutzen, welches extern geladen werden kann. Der erste Gedanke wäre, einfach JPG-XL oder PNG-Bilder zu benutzen, jedoch sind rasterbasierte Grafikformate hier nicht sehr nützlich, da die Warnung auf viele verschiedene Bildschirmgrößen angezeigt werden muss, also sind vektorbasierte Grafikformate nötig. Die bekannteste wäre SVG, ein XML-basiertes Format, mit dem Bilder geschrieben werden können, die unendlich groß skaliert werden können. SVG ist jedoch sehr komplex und hat Features, die hier nicht nötig sind.
Da ich kein vektorbasiertes Grafikformat gefunden habe, welches auch sehr simpel gehalten ist, habe ich mich entschlossen, ein eigenes Format zu entwickeln.

Das Format hat eine funktionsbasierte Syntax, das heißt, dass im Gegensatz zu SVG man einfach Funktionen aufruft, um Formen zu zeichnen oder Text zu schreiben:

```
rectangle (x=100,y=100,height=100,width=100,color="#FFFFFFFF",fill=true)
```

Da diese Art von Syntax sehr simpel zu parsen ist, kann es alles direkt in POSIX-C implementiert werden, ohne externe Bibliotheken verwenden zu müssen.
Der nächste Schritt ist, festzulegen, welche Funktionen benötigt werden. Mit Betracht auf die unterstützten Funktionen in raylib habe ich die folgenden Funktionen implementiert:

- IMG

  Funktion, um ein Bild zu initialisieren, muss immer die erste Funktion in einem Bild sein.

- rectangle

  Ein Rechteck, unterstützt ausgefüllte und nicht ausgefüllte Rechtecke.

- roundedrectangle

  Ein Rechteck, aber mit abgerundeten Ecken.

- circle

  Ein Kreis.

- circlesegment

  Ein Kreissegment.

- ring

  Ein Ring, kann genutzt werden um nicht ausegfüllte Kreise zu zeichnen.

- ellipse

  Eine Ellipse.

- triangle

  Ein Dreieck.

- text

  Text.

Mit diesen Funktionen kann ein Bild ungefähr so aussehen:

```
// The IMG function is always required
// it initializes the image with its size
IMG (height=100, width=100)

// A rectangle
rectangle (x=0, y=0, height=100, width=100, color="#5BCEFA", fill=true, thickness=0)

/*
 Rectangle but multiline
*/
rectangle (x=20, y=0,
height=60, width=100,
color="#F5A9B8",
fill=true, thickness=0)

circle (x=100,y=100,radius=10,color="#CfCfCf")
```

Trotz des sehr simplen Aufbaus und kleinen Arsenals an Funktionen ist es möglich, viele verschiedene Dinge zu zeichnen, perfekt um Grafiken für die Warnung von Nutzern zu erstellen.

Abbildung 3: Haskell Logo in bvg

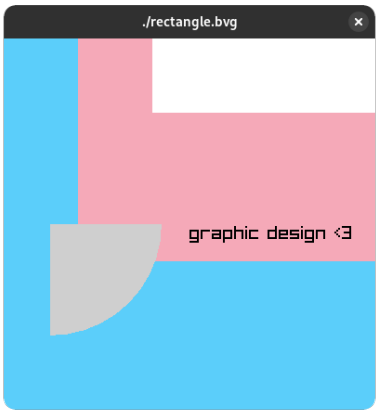


Abbildung 4: Rechtecke und Text

3 Reflexion

Insgesamt gibt es keine großen Mängel, die mir während der Implementierung aufgefallen sind. Die meisten Kritikpunkte liegen in Warnen, welche teilweise der kurzen Entwicklungszeit (7 Tage) zuschulden sind.

3.1 Bessere Datenbank

Die Datenbank welche ich zurzeit nutze, bbolt, hat mehrere Probleme, zum einen ist die Lese und Schreibgeschwindigkeit nicht sehr schnell, trotz bestmöglicher Optimierungen durch Nutzen von Batch-Operation und einmaligen öffnen und Schreiben der Datenbank, fügt die Datenbank ungefähr 2 Sekunden an Laufzeit zu verifysetup, 22% der ganzen Laufzeit.

Dazu kommt auch, das bbolt es zurzeit nicht unterstützt, eine Datenbank direkt aus einer Variable zu lesen, die Datenbank muss als Pfad im Dateisystem angegeben werden, welches dazu führt, das fsverify die Datenbank von der Partition in eine Variable liest, und die Variable direkt wieder in einer Datei in `/tmp` schreibt. Dies führt zu unnötigen Write-cycles, die durch das Verwenden einer anderen Datenbank oder einem Patch für den bbolt Quellcode gelöst werden könnte.

3.2 Nutzung vom TPM2 für öffentliche Schlüssel

Dieses Feature war geplant, und ich hatte bereits einen Schlüssel durch verschiedene Linux Tools in den TPM geschrieben, jedoch konnte ich keine gute go Bibliothek für TPMs finden, weshalb ich das Feature auslassen musste, hätte ich dies noch bevor ich mit der Implementierung gewusst, hätte ich entweder eine andere Programmiersprache für fsverify gewählt, oder eine eigene Bibliothek für TPMs als teil des Projekts entwickelt.

3.3 Besserer Parser für fbwarn

Zurzeit benutzt fbwarn einfaches String Matching mit Funktionen aus `stdlib.h` und `strings.h`, dies Funktioniert, jedoch bringt es viele Probleme mit sich, sodass zum Beispiel ein Leerzeichen am falschen Platz bereits vieles Zerstören kann, welches sehr schwer zu debuggen ist, da man Fehler solcher Art nicht sofort erkennt.
Hätte ich mir für fbwarn mehr Zeit gegeben, hätte ich Programme benutzt, die speziell für das Parsen von Dateien in C gedacht sind, wie `yacc(1)` und `lex(1)`.

3.4 Mehr Funktionen in bvg

bvg unterstützt zurzeit neun Funktionen, wie bereits gezeigt ist dies zwar genug, um recht viel zu zeichnen, jedoch unterstützen die Funktionen alle nur solide Farben, also keine Farbübergänge oder ähnliches, welches das Design der Bilder einschränkt und recht “alt” erscheinen lässt, da Farbübergänge für Elemente wie Schatten in modernen Designs sehr oft genutzt werden.
Zudem unterstützt bvg keinen Bézier Kurven, die das Zeichnen von beinahe jeder Form erlauben. Das Fehlen ist jedoch ein Zeitproblem, da raylib bereits Funktionen für Bézier Kurven hat und die Implementierung in bvg recht simple wäre.