Forensic Science International: Digital Investigation, Volume 39, December 2021, Article number 301269
DOI:10.1016/j.fsidi.2021.301269

Purple Dawn: Dead disk forensics on Google's Fuchsia Operating System

Matt Jarrett 1, Sarah Morris 1

1: Centre for Electronic Warfare, Information, and Cyber, Cranfield University, Defence Academy of the United Kingdom, SN6 8LA

Corresponding Author: DigitalForensics@Cranfield.ac.uk

Abstract

Fuchsia is the project name for a "modular, capability-based" operating system currently being developed by Google. There is speculation that Fuchsia may be a successor to the Android OS or a replacement for several other operating systems currently supported by the organisation. This paper examines the filesystems found in this operating system and provides a breakdown of the content and structure of the unique volume manager and other partitions found on system. The findings outlined in this paper should allow digital investigators to expedite their understanding of the underlying data found on the platform. This paper also highlights how the zxcrypt encryption subsystem may inhibit the ability of practitioners to carry out an investigation of the MinFS partition. As Fuchsia is still in development, these findings are reliant on there not being significant changes made to structure of partitions examined. There remain unanswered questions regarding the content of the BootFS disk image found in the ZIRCON partition and the structure of entries within the Slice Allocation Table in the FVM.

Keywords

Google Fuchsia; Fuchsia Volume Manager, FVM, Zircon, ZBI, MinFS, BlobFS, Zxcrypt

Section 1: Introduction

Fuchsia the project name for an open source "modular, capability-based" (Google Git, 2020) operating system currently being developed by Google. What is unusual about Fuchsia is that unlike Google's other operating systems, it does not use a Linux kernel (Fuchsia Project, 2019a) but rather a custom microkernel called Zircon. Such a move represents a significant investment in terms of time and effort as well as a shift to a theoretically more secure platform (Setapa et al., 2011). As of January 2020, Google has not made an official statement regarding Fuchsia's purpose, although Google's Senior Vice President for Android, Hiroshi Lockheimer, has spoken of the "increasing number of devices that require operating systems" (Statt, 2019), such as those found within the Internet-of-Things (IoT) space, suggesting that Fuchsia may be an alternative for these. Key personnel from the Android Project

team (Rahman, 2018) and former senior engineers at Apple (Li, 2019) have publicly acknowledged that they are working on the project with the aim of bringing it to market. The presence of ex-Android leads within the Fuchsia team has led to some speculation that Fuchsia may be in fact a replacement for Android OS as a whole, or at least an attempt to reduce the number of operating systems supported by the organisation (Mark Gurman, 2018). With a wide range of potential devices being targeted and an unknown deployment timeframe, digital investigators may soon find themselves dealing with a range of devices utilizing an unfamiliar operating system with unknown data partitions and storage methodologies.

The purpose of this paper is to present an initial investigation into the operating system, and to provide the necessary supplementary material for investigators and researchers to begin to be able to understand the disk structures utilised by Fuchsia. This paper outlines the disk-level identifiers and data storage structures of Fuchsia's custom volume manager, the partitions found within this volume manager and a brief description of their purpose within the wider operating system. Should Fuchsia see market deployment in a similar state to its current configuration, these findings should expedite digital investigators understanding of the underlying data and permit extraction of useful material.

The remainder of this paper is structured as follows: Section 2 provides background material for the operating system whilst Section 3 highlights related work. Section 4 outlines the objectives and Section 5 contains the methodology for the investigation. In Section 6 the results of this research can be found which is followed by a discussion in Section 7. In Section 8 areas for further investigation are highlighted and Section 9 concludes this work.

Section 2: Background on Fuchsia

Public knowledge of Fuchsia originated from the appearance of software repositories containing the early versions of OS in August 2016. The appearance of these repositories was done so without official announcement, and early inspection of the codebase indicated numerous codeword references to various devices (Barth et al., 2019), including some not previously known about (Bradshaw, 2019). These repos contain instructions for deploying Fuchsia onto a range of test devices from workstations to routers (Robinson et al., 2019) with build options for both Intel x86_64 and ARM architectures (Hockett et al., 2019). As such, if Fuchsia ever comes to market forensic investigators may start to encounter it across a variety of devices.

Fuchsia's kernel, Zircon, is based on the LK microkernel (Fuchsia Project, 2019b), and extends its functionality by introducing features more akin to traditional kernels such as system calls and support for user space operations. The drive to utilise this alternative kernel seems in part to stem from the desire to implement a capability-

based security model, where each system service and user application has only the minimum privileges required to execute successfully (Gusmeroli, Piccione and Rotondi, 2013). The implications of this shift in security model is that it is also necessitates a change in the way the OS handles, regulates access to and stores underlying user data. This has ramifications for digital forensic investigations in that the ability for investigators to attribute which user or application has created, had access to or modified specific sets of data may be hindered through a lack of available filesystem metadata.

Fuchsia defines its own logical volume manager, referred to as the Fuchsia Volume Manager (FVM) (Fuchsia Project, 2019c), which creates and serves portions of storage (or 'slices') and allocates them to Fuchsia specific filesystems. FVM maintains data regarding the allocation and virtual position of these slices within a designated filesystem. Specific documentation on FVM is relatively high level (Fuchsia Project, 2019d), although the codebase responsible for defining the data structures is open source (Google Git, 2019a). The kernel for the operating system is held within a custom container format referred to as Zircon Boot Image or ZBI (McGrath et al., 2020). This format reportedly holds hardware specific information and kernel command line instructions (Fuchsia Project, 2019e), although documentation of the structure of ZBI is limited to high level descriptions and raw codebase dumps. As such work is required to determine how this information is represented on disk or otherwise identified.

Fuchsia defines a number of custom file systems (Fuchsia Project, 2019d) for use within FVM. These fulfil a variety of roles within the OS; BlobFS provides a flat file system for 'write-once then read only' data such as application packages for system services (Fuchsia Project, 2019f), whilst MinFS, a simple, Unix-like filesystem (Fuchsia Project, 2019g) provides persistent storage for user data**.** Given the stated roles of these filesystems, understanding of their composition may be of significant interest to investigators.

Fuchsia also utilises a custom, transparent disk encryption system referred to as zxcrypt (Zircon Crypt) (Fuchsia Project, 2019h). In terms of operation this appears to function in a similar way to dm-crypt (Device Mapper crypt) as found in the Linux kernel (Broz, 2019), through the use of a device mapper service which decrypts read requests and encrypts write requests to the underlying storage device. The implications of such a system from a digital forensics perspective are obvious in that without a method to bypass or acquisition of the keys used in the encryption, investigators may struggle to complete an investigation.

Section 3: Related works

As indicated in Section 2, high-level documentation is available (Fuchsia Project, 2019i), which outlines the justification for some of the approaches taken within the operating system as well as providing an indication as to what should be found on disk. This however lacks detail, or a clear breakdown of the actual structures expected to be found on the storage medium. The United States (US) National Security Agency (NSA) has previously presented on the fundamentals of Fuchsia (Carter, 2018) and how it handles certain data primitives. This work however did not delve into data storage and was more focused on the security features of the kernel. Naumann has published material highlighting the system calls utilised by the operating system's microkernel Zircon (Naumann, 2018), although this was to serve the development of another operating system and was not targeted at features of direct concern by forensic practitioners. The United Kingdom's (UK) Defence Science and Technology Laboratory (DSTL) wrote a bulletin providing an initial look at the operating system with the aim of identifying issues from a forensic perspective (DSTL, 2019). This work serves as a broad introduction to Fuchsia and discusses many of its features, but openly stated that more work would be required to analyse the data structures found on target devices (DSTL, 2019).

The underlying code base (Google Git, 2019a) for Fuchsia is available for inspection. This is useful for understanding the actual implementation of the filesystems and provides an indication as to what should be found on disk. Given MinFS appears to take inspiration from Unix-based file systems, examination of prior work in the examination of Unix file systems such as Ext3 (Fairbanks, Xia and Owen, 2009; Narváez, 2007; Piper et al., 2006) and Ext4 (Fairbanks, 2012; Pomeranz, 2010) file systems can be utilised to suggest what kind of data structures are expected to be found. Much work has previously been carried out in the investigation of the logical volume managers utilised for both the Linux and Windows operating systems (Carrier, 2005a, 2005b; Prokop, 2013; Rocha, 2017) which may be used to compare and contrast against FVM.

Section 4: Objectives

As seen in Section 3, there is a gap in understanding regarding how data is fundamentally stored, structured and organised within the Fuchsia platform. This includes both the partitions utilised by the operating system and the filesystems within the custom volume manager itself. As a result, the objectives of this research were to:

- Investigate the partitions found on Fuchsia devices and determine the unique digital identifiers.

- Examine the FVM and determine what information is held regarding the filesystems it allocates storage space to.
- Determine the role of these partitions in the context of the wider operating system.

The first question indicated the need to determine and document the Globally Unique IDentifiers (GUIDs) associated with Fuchsia partitions as well as the position and layout of these partitions found. This involved the examination of the storage media utilised by target Fuchsia devices to determine GUID Partition Table (GPT) entries and allocation of storage space to various system partitions. The magic numbers and other identifiers associated with each Fuchsia partition were also examined and documented as well as significant metadata structures. The second question focused on Fuchsia's unique volume manager, and this research aimed to examine and document the schema FVM utilises for the allocation of storage. This involved the examination of the data structures for FVM and documenting how storage space is distributed, tracked, and served to filesystems within the volume manager. The third question was expected to be answered through examination of the content of the data section within each partition alongside examination of the codebase and available documentation.

Section 5: Methodology

For this examination an Ubuntu 18.04 machine was utilised as the scripts and tooling within the Fuchsia codebase are targeted towards a Unix-development environment, and a Debian-based distribution was specifically recommended. The source code for the operating system was downloaded via the Fuchsia public software repository (Fuchsia Project, 2020a), which outlined the compiling process and pre-requisites. This involved utilising developer provided bash scripts (Fuchsia Project, 2017; Google Git, 2015) to pull the requisite files and generate the directories for the source code as well as install *jiri*, a custom tool for multiple repository development and management (Google Git, 2019b). In addition to *jiri*, these scripts a new command line tool, *fx* (Fuchsia Project, 2019j), for use in the building, configuration and deployment of Fuchsia.

Fuchsia was compiled using *fx* with the target architecture set as 'x64' and the target product as 'core'. It was compiled using the latest public build at the time of testing, on the 20/05/21 at 0900hrs UTC. The 'x64' build option was chosen as only platforms utilizing this architecture were available for this research. The 'core' build was selected as the documentation indicated that compiling the OS using other options only appeared to affect higher-level features in the OS (Fuchsia Project, 2019j), and as such noteworthy differences to the partition data structures was not expected. This was confirmed with preliminary testing. From this preliminary testing it was determined that regardless of build selected, Fuchsia development builds for

physical devices are approximately 8.02 GiB in size with any remaining space on the storage medium going unutilised.

Three Intel New Unit Computing (NUC) PCs were utilised as test devices for examination. The model of Intel NUCs (BOXNUC7i5BNH) were chosen as the developer documentation specifically stated support for this model (Fuchsia Project, 2019k) and there was a desire to minimise any potential hardware compatibility issues during the deployment process. The test devices were built with 250-GB Non-Volatile Memory express (NVMe) Solid State Drives (SSD) and 16-GBs of RAM. Other platforms were investigated, for example a Google Pixelbook device, however it was found that the deployment process left ChromeOS recovery and kernel partitions on the system to allow users to easily rollback to a production operating system. As such it was felt that utilising these as a target platform would be less representative of any potential future example. Hardware compatibilities issues prevent the successful deployment of Fuchsia onto other platforms.

QEMU-based virtual machines (VMs) were built by issuing a '*fx emu'* command. This process created a live boot version of Fuchsia and generated a temporary raw disk file. Unlike the builds of Fuchsia found on the physical devices, these VMs only utilise about 770-MiB of storage space, and the temporary files only contain the FVM partition. These VMs were utilised as the built process did not include the zxcrypt functionality and as such this provided a way to examine the unencrypted content of a MinFS partition within the FVM.

The actual deployment of Fuchsia onto the physical platforms involved formatting an external USB storage drive using the *'fx mkzedboot'* command. This allowed the USB drive to be used to live-boot the target platform, which would then query for a Fuchsia development server over the local network. A listener service was run (via the '*fx pave*' command) which would serve the latest build of the operating system to the querying platform. The bootloader would then 'pave' the target device with a Fuchsia system image using the onboard storage.

Once booted, the Fuchsia devices presented a simple user shell akin to those found on Unix distributions, although with reduced functionality due to the stripped-down nature of the core build and permissions issues. Various debugging tools within this terminal were utilised for this investigation. This included '*gpt*' (Yip et al., 2019), which when used with the '*dump*' argument provided a summary of the content of the GPT on a target system. The tool '*fvm-check'* (Klein et al., 2019) was utilised to provide an on-system interpretation of the content and the allocation of storage within FVM. 'lsblk' (Fuchsia Project, 2020b) provided a breakdown of the allocation of block devices viable from the user shell, whilst 'df' (Fuchsia Project, 2020c) was utilised to confirm the mount points for the partitions inside FVM. Over the course of examination, fifty files of varying sizes (1KB-100MB) with unique identifiers were created within the writable directories prior to acquisition. Some of the larger files were deleted prior to the creation of smaller files to determine how the OS handled

deleted data and reallocation of storage. This was done on both the physical and virtual devices.

From the physical devices, images of the hard drives were taken using FTK Imager v3.1.4.6. These images were remounted as read-only and examined in WinHex 19.0 hexadecimal editor. For the virtual machines, copies of the storage files were taken and loaded directly into WinHex for examination. The content of the GPT header and partition table was inspected and cross-referenced against the output of the *gpt* command within the target system. Understanding of this data was facilitated utilizing prior work (Nikkel, 2009) and allowed for the identification of the starting locations for various partitions on disk as well as their type GUIDs.

Through the examination of the specific sections of Fuchsia's codebase responsible for generating the volume manager and partitions found on system, the content and purpose of various fields within different metadata structures were identified. Confirmation testing allowed for verification of these findings. Other elements of the codebase for Fuchsia were inspected, for example the Software Development Kit (SDK) (Fuchsia Project, 2019l), with the aim of determining references for suspected static values, such as GUIDs for partition types. Confirmatory testing through modification of these values and then inspecting how the OS description of them was changed allowed for corroboration of this.

Section 6: Findings

6.1 Overall Disk Structure

Sector(s)	Physical Offset	Description
0	0 - 511	Protective MBR
1	512 - 1,023	GPT Header
2	1,024 - 1,471	GPT Entries
2 - 34	1,472 - 17,407	Unused space
34 - 16,777,249	17,408 - 8,589,951,999	FVM Partition
16,777,250 - 16,818,209	8,589,952,000 - 8,610,923,519	EFI bootloader
16,818,210 - 16,883,745	8,610,923,520 - 8,644,477,951	ZIRCON-A
16,883,746 - 16,982,049	8,644,477,952 - 8,694,809,087	ZIRCON-R
16,982,049 - 488,397,134	8,694,809,088 - 250,059,333,119	UNUSED SPACE
488,397,135 - 488,397,166	250,059,333,120 - 250,059,349,503	Backup GPT entries
488,397,167	250,059,349,504	Backup GPT header

Table 1: Partition distribution across a sample Fuchsia GPT based disk.

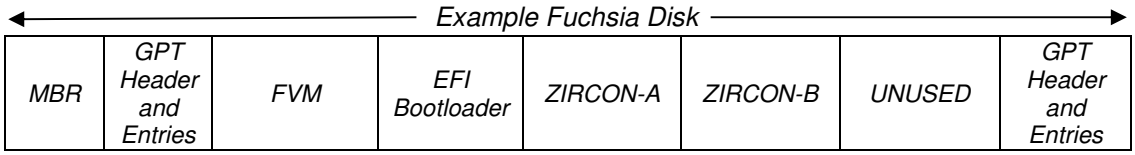


Figure 1: Basic distribution of partitions on Fuchsia Disk

Table 1 documents the number of sectors and physical offsets allocated to each partition found on a target device utilising the Fuchsia OS, whilst Figure 1 shows a basic outline of their order of allocation. Several Fuchsia-unique partition type GUIDs were identified. These have been documented in Table 2. These were initially identified after examining the GPT header and entries. An extended list of Fuchsia-specific GUIDs was identified through examination of the SDK (Fuchsia Project, 2019l) and this was used to classify other GUIDs found throughout experimentation. This list was further augmented with additional GUIDs identified for the Zircon Boot Image format (McGrath et al., 2020). Compliance testing through physically overwriting of these values using test partitions confirmed the EFI-name definitions.

The EFI Gigaboot partition seen in Table 1 was a FAT-32 EFI partition where a bootloader service was operating. This service queries for a Fuchsia development server on the local network to check for new updates. Once complete, the boot loader loads the kernel and other boot items (kernel memory, commandline instructions, etc) from the active Zircon partition and transfers control to the kernel (Fuchsia Project, 2019e).

EFI Name	Partition Type GUID	Description
efi-system	{C12A7328-F81F-11D2-BA4B-00A0C93EC93B}	EFI bootloader
Fuchsia-system	{606B000B-B7C7-4653-A7D5-B737332C899D}	BootFS
fuchsia-data	{08185F0C-892D-428A-A789-DBEEC8F55E6A}	MinFS partition
crashlog0	{b25e3082-9ed3-7545-4C84-D072206CC8A0}	Zircon Boot Image
fuchsia-blob	{2967380E-134C-4CBB-B6DA-17E7CE1CA45D}	BlobFS partition
fuchsia-fvm	{41D0E340-57E3-954E-8C1E-17ECAC44CFF5}	FVM partition
zircon-a	{DE30CC86-1F4A-4A31-93C4-66F147D33E05}	ZIRCON Kernel image
zircon-b	{23CC04DF-C278-4CE7-8471-897D1A4BCDF7}	ZIRCON Kernel disk image secondary
zircon-r	{A0E5CF57-2DEF-46BE-A80C-A2067C37CD49}	Zircon recovery kernel image
zxcrypt	{00F8E85F-6DB3-E711-807A-786372797074}	Zxcrypt encrypted partition

Table 2: Partition Type GUIDs defined within the Fuchsia SDK.

6.2 ZIRCON-A and ZIRCON-R Partitions

Both the ZIRCON-A and ZIRCON-R partitions utilised the same overarching data structure and as such only the data structures specifically found on a ZIRCON-A partition have been documented below. As part of Fuchsia's boot sequence, data from this partition such as kernel images, hardware-specific information and specific command line instructions for the kernel (Fuchsia Project, 2019e) is loaded into memory. This information is held within custom data structures referred to as 'boot items' in the Zircon Boot Image (ZBI) format. These boot items consist of a header and payload. The first boot item within a ZIRCON partition is the ZBI Container. All other boot items within a ZIRCON partition are held within the payload this ZBI Container boot item. At a minimum a ZBI Kernel and ZBI Storge boot items should always be found within this payload. This is depicted in Figure 2 alongside the order in which these boot items should be found.

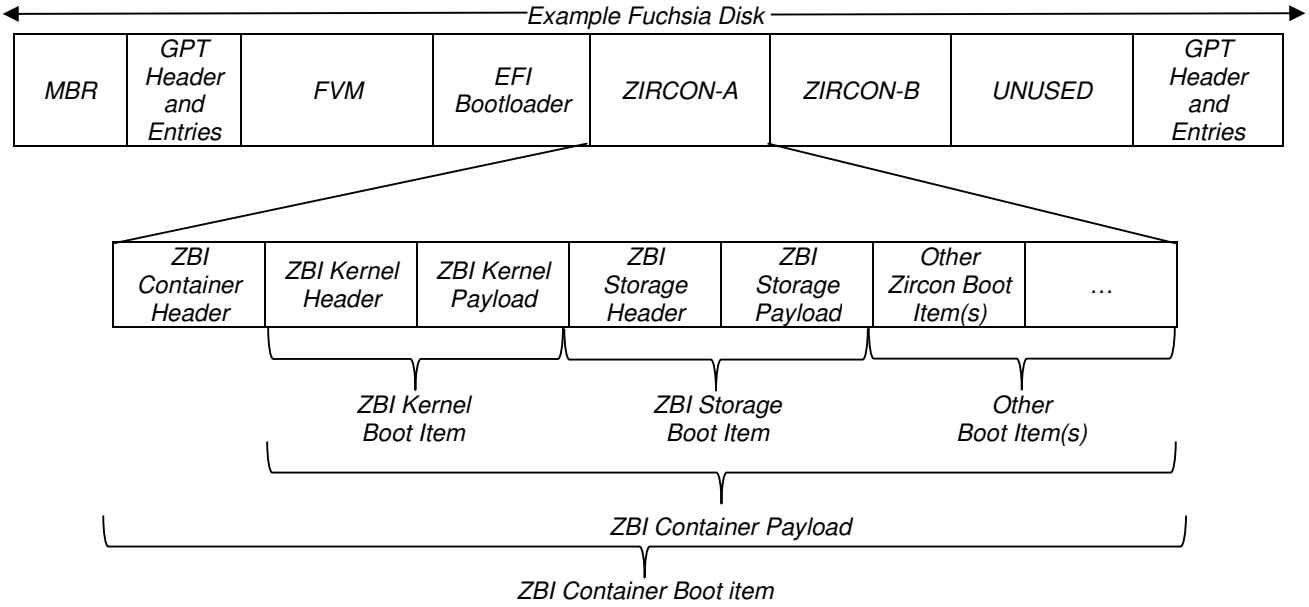


Figure 2: Content of Zircon-A Partition.

The headers of each boot item found in a ZIRCON partition is broadly similar. The structure of this header is outlined in Figure 3. This was determined through cross-examination of the source code (McGrath et al., 2020) with information recovered from Fuchsia test devices. The specific header used by the ZBI Container boot item is shown in Figure 4.

Relative offsets	0	4	8	12
0	**ZBI Type Prefix 4 Bytes**	**Length of Payload uint32**	**ZBI Extra 4 Bytes**	**ZBI Flags 4 Bytes**
16	**Reserved(0) 4 Bytes**	**Reserved(1) 4 Bytes**	**ZBI Item Magic 4 Bytes**	**Payload CRC-32 4 Bytes**

Figure 3: *General format of boot item headers.*

Relative offsets	0	4	8	12
0	**ZBI Type Prefix** **4 Bytes** 0x42 4f 54 (BOOT)	**Length of Payload** **uint32** 9,753,136 bytes	**ZBI Extra** **4 Bytes** 0xE6 F7 8C 86	**ZBI Flags** **4 Bytes** 0x00 00 01 00
16	**Reserved(0)** **4 Bytes** 0x00 00 00 00	**Reserved(1)** **4 Bytes** 0x00 00 00 00	**ZBI Item Magic** **4 Bytes** 0x29 17 78 B5	**Payload CRC-32** **4 Bytes** 0xD6 E8 87 4A

Figure 4: Format and content of an example ZBI Container boot item header.

The ‘ZBI Type Prefix’ field is populated via a static list and can be used to identify the boot item and infer what should be found within its payload. A listing of the various ZBI Types and their associated ZBI Type prefixes can be found in source code (McGrath et al., 2020). As this boot item is effectively a container for all other boot items, the length of the payload indicates the complete size (omitting the first 32-bytes of the ZBI container header) that is utilised for the given partition. The ZBI Extra field seen in the Container header is a static value - the last 4-bytes of SHA-256 of the string “bootdata” represented in little endian. This value is reused by several boot items to populate this field. The flag value shown in the header is the version flag (*0x00 00 01 00*) and is seen in all headers within the Zircon partition. Two other flags are defined within the code base; *0x01 00 00 00* which indicates the payload is compressed, and *0x00 00 02 00*, which indicates no CRC-32 of the payload has been generated, and a placeholder value has been utilised. The ZBI Item Magic is another static value - the last 4-bytes of a SHA-256 of the string “bootitem”, again represented in little endian.

The ZBI Container header is followed by ZBI Kernel boot item. This boot item contains an image of the system kernel in its payload. This is loaded into memory at runtime by the bootloader. This boot item’s header differs from others in that it utilises an additional two fields appended to the end. This is shown in Figure 5 below.

Relative offsets	0	4	8	12
0	**ZBI Type Prefix** **4 Bytes** 0x4B 52 4E 4C (KRNL)	**Length of Payload** **uint32** 1,979,672 bytes	**ZBI Extra** **4 Bytes** 0x00 00 00 00	**ZBI Flags** **4 Bytes** 0x00 00 01 00
16	**Reserved(0)** **4 Bytes** 0x00 00 00 00	**Reserved(1)** **4 Bytes** 0x00 00 00 00	**ZBI Item Magic** **4 Bytes** 0x29 17 78 B5	**Payload CRC-32** **4 Bytes** 0xD6 E8 87 4A
32	**Physical Entry Point Address** **uint64** 1,048,696		**Minimum Reserved Memory** **uint64** 595,816 (bytes)	

Figure 5: Format and content of an example ZBI Kernel boot item header.

The last byte of the ZBI Type prefix indicates which architecture the boot item has been built for; 0x4c or ‘L’ indicates x86_64, 0x38 or ‘8’ indicates ARM-64. The payload length indicates the size of the compressed image in the payload. The ZBI Kernel boot item utilises the same static value seen in the ZBI Container Header for

the ZBI Item Magic field. Similarly, As the CRC-32 flag (*0x00 00 02 00*) has not been set, a placeholder hash value is reused. The additional two fields utilised in the header of this boot item relate to where the kernel memory image should be loaded into the device's RAM at boot and the amount of additional space to reserve for use by the kernel. The ZBI Storage boot item is found after the ZBI Kernel boot item. The payload of this boot item contains the boot filesystem to be utilised by Fuchsia at runtime. Figure 6 shows an example of the header.

Relative offsets	0	4	8	12
0	**ZBI Type Prefix** **4 Bytes** 0x42 46 53 42 (BFSB)	**Length of Payload** **uint32** 7,773,279 bytes	**ZBI Extra** **uint32** 30,154,752 bytes	**ZBI Flags** **4 Bytes** 0x01 00 03 00
16	**Reserved(0)** **4 Bytes** 0x00 00 00 00	**Reserved(1)** **4 Bytes** 0x00 00 00 00	**ZBI Item Magic** **4 Bytes** 0x29 17 78 B5	**Payload CRC-32** **4 Bytes** 0x77 19 B4 74

Figure 6: Format and content of an example ZBI Storage header.

ZBI Storage boot items utilise several different prefixes to indicate the content of the payload; In this example, the ZBI Type prefix field indicates the presence of a BootFS filesystem image based on the '*BFSB*' value. At boot time the image within the payload is uncompressed, loaded into memory, and mounted under the /boot path on the system. An indication that the payload is compressed can be seen in the differences between the payload length value, and the ZBI Extra field value (where the latter indicates the true, uncompressed size).The first byte (*0x01*) of the ZBI Flags field in this header indicates that the payload is compressed. Unlike the previous headers the CRC32 flag has been set – as such a genuine CRC-32 checksum can be seen at the end of the header. As with the prior two headers, the ZBI Item Magic field utilises the same static value.

The payload for the Storage boot item is prefixed by a header of unknown length. Examination of Fuchsia's source code (McGrath, 2019), indicates that the data within the payload of this boot item contains information to be utilised for decompression of a disk image (for example compression algorithm used). According to Fuchsia's documentation, the payload for this boot item should be compressed using LZ4 compression (Fuchsia Project, 2019e), however the devices examined for testing were found to be utilising Zstandard (Zstd) compression. The compression can be determined by examining the first four bytes of the boot item's payload where *0x28 B5 2F FD* indicates Zstd and *0x04 33 4D 18* indicates LZ4. This is followed by two bytes of unknown purpose and then another four bytes which indicate the uncompressed size of the image (7,773,279 bytes). This value corresponds to the ZBI Extra value seen in the header of this boot item. Further work is required to validate the content and structure of the disk image within the payload.

Other boot items may be found which relate to specific hardware and architectural details or build configurations for the specific built of Fuchsia being examined. The

devices utilised for this research were found to have created a single additional boot item. The on-disk representation of this boot item can be seen in Figure 7 whilst the header is outlined in Figure 8.

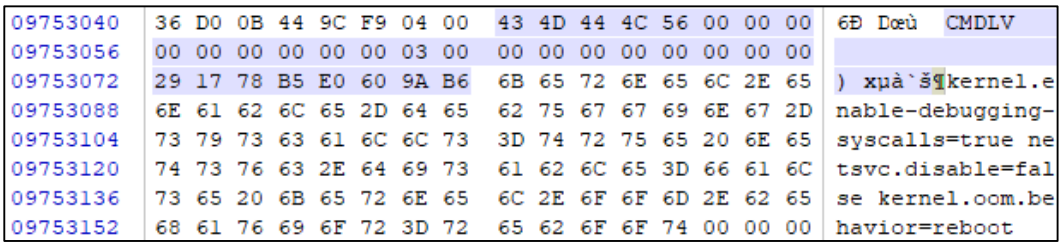


Figure 7: Example Kernel Command Line Fragment boot item (header highlighted).

Relative offsets	0	4	8	12
0	**ZBI Type Prefix** **4 Bytes** 0x43 3D 44 4C (CMDL)	**Length of Payload** **uint32** 86 bytes	**ZBI Extra** **uint32** 0x00 00 00 00	**ZBI Flags** **4 Bytes** 0x00 00 03 00
16	**Reserved(0)** **4 Bytes** 0x00 00 00 00	**Reserved(1)** **4 Bytes** 0x00 00 00 00	**ZBI Item Magic** **4 Bytes** 0x29 17 78 B5	**Payload CRC-32** **4 Bytes** 0xE0 60 9A B6

Figure 8: Format of an example ZBI Type Command line Fragment header.

The header in Figure 8 indicates the boot item is a Command Line fragment – instructions to be passed by the bootloader to the kernel at system start-up. These may alter system behaviour, such as whether Address Space Layout Randomization (ASLR) is utilised to protect system memory (Fuchsia Project, 2020d) or may contain instructions for the kernel to subsequently pass onto userspace processes and/or the device manager, such as whether verbose logging is utilised by the device manager (Fuchsia Project, 2020d). The identity of this boot item can be confirmed by the ZBI Type Prefix at the start of the header. The ZBI Extra field for this specific boot item is not utilised, although under the ZBI Flags field the CRC-32 flag is set (*0x00 00 02 00*) and a legitimate CRC-32 can be found. Within the payload of this boot item, kernel command line instructions are found in ASCII format. These instructions are delineated by white spaces (*0x20*) and are null terminated.

6.3 FVM Partition

Fuchsia's Logical Volume manager makes up the other unique partition with a GPT entry, the general layout of which is depicted in Figure 9. FVM utilises blocks of 8,192 bytes in length and the partition starts with a superblock. The structure of this superblock is depicted in Figure 10. These superblocks can be identified by the 'FVM PART' magic identifier (in ASCII).

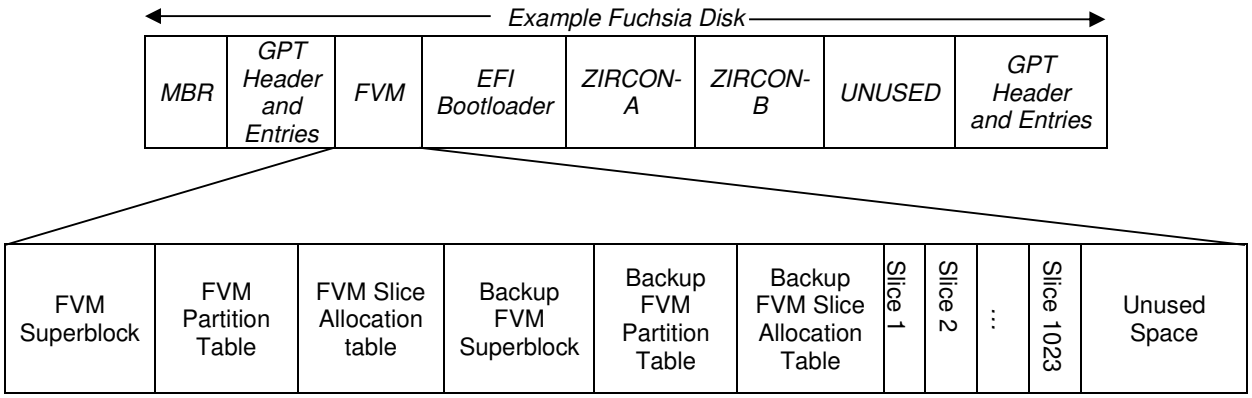


Figure 9: FVM Partition structure

Relative offsets	0	4	8	12
0	**FVM Magic identifier** **8 bytes** 0x46 56 4D 20 50 41 52 54 (FVM PART)		**Version** **uint64** 1	
16	**Max no, of slices** **uint64** 1,023 slices		**size of each slice** **uint64** 8,388,608 bytes (8 MiB)	
32	**size of partition** **uint64** 8,289,934,592 bytes (7.72 GiB)		**size of partition table** **uint64** 65,536 bytes (64 KiB)	
48	**size of allocation table** **uint64** 8,192 bytes (8 KiB)		**Generation** **uint64** 158	
64 80	**Checksum (SHA-256)** **32 bytes** 0x36 99 6A A3 C8 63 59 17 85 D9 CE A3 70 1F F6 CD 63 92 4E EC 7B DC 9D E5 E8 D5 27 DE B8 18 67 79			
96 … 8176	**Reserved** **8,096 Bytes**			

Figure 10: Format and example content of the FVM superblock

Within the superblock, the slice count denotes the maximum number of chunks of storage space that partitions within FVM can be allocated. The given size of each slice (8-MiB in size) is hardcoded. Default development builds of Fuchsia utilise an FVM partition of 8-GiB in length, regardless if the underlying storage device has more space available. This appears to be a result of the build configuration used for Fuchsia rather than a size limitation. Within the superblock, the amount of storage space allocated to both the FVM Partition Table and the FVM Slice Allocation Table is static regardless of the number of entries found in each. The generation number found in the superblock is used to determine which superblock (Primary or Backup) is the latest version. After the SHA-256 checksum, the remainder of the first block in the partition is reserved, possibly as space for future expansion to the superblock. Unlike LVM or LDM, the FVM does not appear to have provisions for providing distributed storage across multiple physical disks (no pointers for other FVM partitions, or notions of Volume Groups or Physical Volumes).

The start of the second block within the FVM contains the FVM Partition Table, which is made up of 64-byte entries, the first entry of which is. A breakdown of the actual structure of the Partition Table can be seen in Figure 11. The example Partition Type GUIDs shown in Figure 11 represent examples found on test devices and correspond to the entries outlined in Table 2. None of the devices examined had any flags set, although the source code indicates that a partition inactive flag (*0x00 00 00 01*) is defined (Klein et al., 2020).

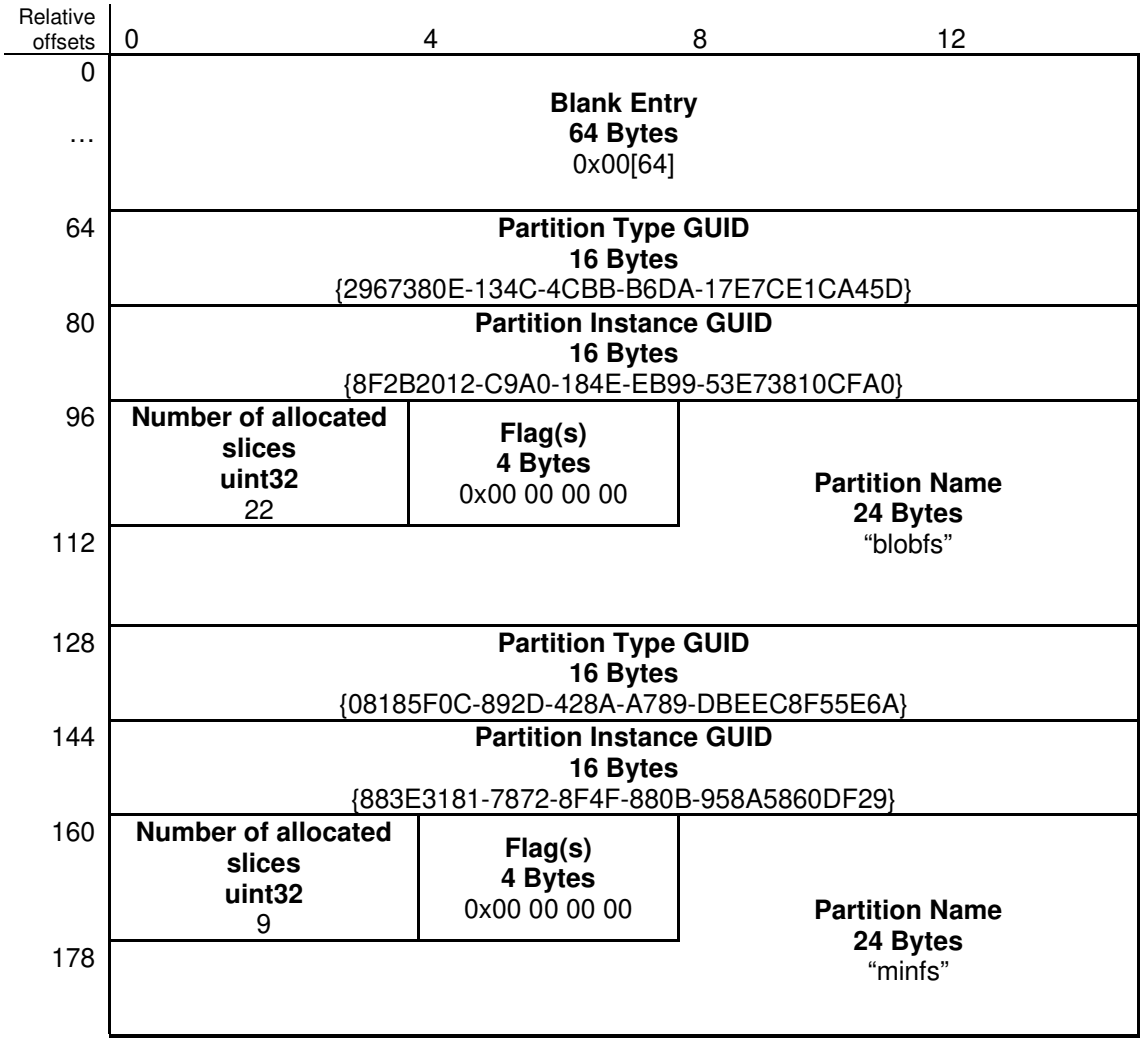


Figure 11: FVM Partition Table with the first (blank) entry highlighted.

Immediately after the space allotted to the Partition Table, the Slice Allocation Table can be found. The entries within the Allocation Table are laid out in an array, with the position of each entry within the array indicating which slice it corresponds to on the storage device. As with the Partition Table, the first entry in this table is blank. Each entry is 8-bytes in length, containing one 16-bit and one 32-bit value with two bytes of padding. The first value signals that the slice is allocated and to which partition. The 32-bit value specifies the virtual slice identity of this entry and indicates the offset (in terms of number of blocks) that the slice starts at within the specified partition. As the entries are required to be 8-byte aligned, the remaining two bytes

within each allocated entry appear to be padding (although this has not been confirmed).

Relative offsets	0	4		8	12	
0	**Blank entry** **8 bytes** 0x00 00 00 00 00 00 00 00			**Partition allocation uint16** **1**	**Virtual slice ID uint32** 0	**Padding 2 bytes** 0x00 00
16	**Partition allocation uint16** 1	**Virtual slice ID uint32** 64	**Padding 2 bytes** 0x00 00	**Partition allocation uint16** 1	**Virtual slice ID uint32** 128	**Padding 2 bytes** 0x00 00
32	**Partition allocation uint16** 1	**Virtual slice ID uint32** 192	**Padding 2 bytes** 0x00 00	**Partition allocation uint16** 1	**Virtual slice ID uint32** 256	**Padding 2 bytes** 0x00 00
48	***Entry for physical slice 7***			…		

Figure 12: First 6 entries of an example FVM Allocation Table.

The FVM Superblock, Primary Partition Table and Allocation Table take up 10 blocks worth of space (81,920 bytes). These data structures are followed by the Backup FVM Superblock, Secondary Partition table and Secondary Allocation table (for a total of 81,920 bytes). These are followed by the first data slice (starting from offset 163,840). In testing, this was always allocated to the BlobFS filesystem. Using the Fuchsia developer tool *fvm-check*, it was possible to confirm this (Figures 13 and 14).

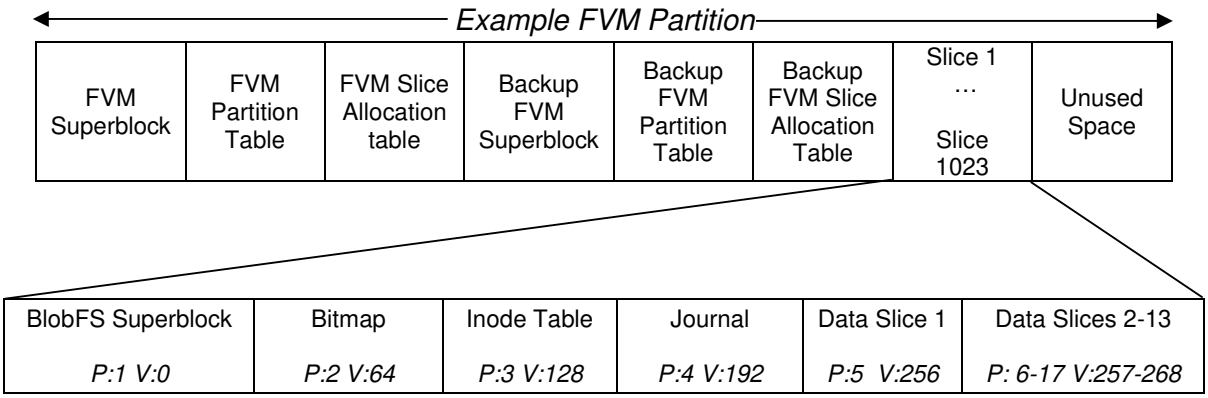
```
$ fvm-check /dev/sys/pci/00:02.0/ahci/sata0/block
[  FVM Info  ]
Version: 1
Generation number: 0
Generation number: 0 (invalid copy)

[  Size Info  ]
Device Length:   788856832
Block size:           512
Slice size:       8388608
Slice count:           94

[  Metadata  ]
Valid metadata start:     0x0000000000000000
Metadata start:           0x0000000000000000
Metadata size:                         81920 (for each copy)
Metadata count:                            2
Metadata end:             0x0000000000028000

[  All Subsequent Offsets Relative to Valid Metadata Start  ]

[  Virtual Partition Table  ]
VPartition Entry Start:   0x0000000000002000
VPartition entry size:                    64
VPartition table size:                 65536
VPartition table end:     0x0000000000012000

[  Slice Allocation Table  ]
Slice table start:        0x0000000000012000
Slice entry size:                          8
Slice table size:                        752
Slice table end:          0x00000000000122f0
```

Figure 13: Part 1 of fvm-check detailing metadata block sizes and starting location.

```
[  Slice Info  ]
Physical Slice 1 allocated
  Allocated as virtual slice 0
  Allocated to partition 1
Physical Slice 2 allocated
  Allocated as virtual slice 64
  Allocated to partition 1
Physical Slice 3 allocated
  Allocated as virtual slice 128
  Allocated to partition 1
Physical Slice 4 allocated
  Allocated as virtual slice 192
  Allocated to partition 1
13 Physical Slices [5, 17] allocated
  Allocated as virtual slices [256, 268]
  Allocated to partition 1
Physical Slice 18 allocated
  Allocated as virtual slice 0
  Allocated to partition 2
Physical Slice 19 allocated
  Allocated as virtual slice 64
  Allocated to partition 2
Physical Slice 20 allocated
  Allocated as virtual slice 128
  Allocated to partition 2
Physical Slice 21 allocated
  Allocated as virtual slice 192
  Allocated to partition 2
2 Physical Slices [22, 23] allocated
  Allocated as virtual slices [256, 257]
  Allocated to partition 2
Physical Slice 24 allocated
  Allocated as virtual slice 320
  Allocated to partition 2
$
```

Figure 14: Part 2 of fvm-check detailing slice allocation.

6.3.1 BlobFS



P: Physical Slice Identifier
V: Virtual Slice Identifier

Figure 15: Makeup of BlobFS partition in relation to FVM

The BlobFS partition is the one of two Fuchsia-unique file systems found within the FVM. A basic representation of the content of the BlobFS partition can be seen in Figure 15. As stated in Section 2, BlobFS is a flat file system and is mounted under the /blob path. This filesystem comprises of a series of 'blobs' – immutable, custom objects containing files and/or other components to support system services and applications. The names of these blobs are deterministically derived from their content (Fuchsia Project, 2020e), using Merkle Trees; a hierarchical data structure consisting of a series of SHA-256 cryptographic hashes generated from the value of lower level nodes within the tree (Fuchsia Project, 2019m). The lowest level 'leaf' nodes compute their hashes from individual data blocks utilised by the blob. This construct continues upwards until the tree terminates with a single hash, referred to

as the Merkle Root. This hash value is used to name the blob, examples of which can be seen in Figure 16.

Figure 16: Example Content of /blob file path on Fuchsia device

By default, a single BlobFS file system is made up of at least six FVM slices which are allocated in the order as seen in Figure 15. At least two slices are always allocated to the data section of a BlobFS partition. BlobFS also utilises a significant degree of virtual padding between its various data structures. This appears to be to facilitate potential growth for the various fields and whilst this does not (initially) affect the actual data distribution it is something that needs considering when calculating offsets. The slices allocated to the data section do not utilise this padding and have sequential virtual IDs. This can be seen in the physical slices 5-17 in Figure 14. As with FVM, the first block of a BlobFS partition contains the partition's superblock. A breakdown of the structure of the BlobFS Superblock can be seen in Figure 17.

Relative offsets	0	4	8	12
0	**BlobFS Magic Identifier Part 1** **8 Bytes** 0x21 4D 69 9E 47 53 21 AC (!MižGS!¬)		**BlobFS Magic Identifier Part 2** **8 Bytes** 0x14 D3 D3 D4 D4 00 50 98 (ÓÓÔÔ p~)	
16	***Version*** ***uint32*** 8	**Flags** **uint32** 4	**Block size** **uint32** 8,192 Bytes	
32	**No. of Data Blocks** **uint64** 13,312		**No. of Journal Blocks** **uint64** 1,024	
48	**No. of Inode entries** **uint64** 131,072		**No. of blocks allocated** **uint64** 328	
64	**No. of allocated Inode entries** **uint64** 358		**Next BlobFS Partition Location** **uint64** 0	
80	**Current Slice size** **uint64** 8,388,608		**No. of slices in this partition** **uint64** 17	
96	***Slices Allocated to bitmap*** ***uint32*** 1	**Slices allocated to Inode table** **uint32** 1	**Slices allocated to Data section** **uint32** 1	**Slices allocated to Journal** **uint32** 1

Figure 17: Structure and example content of the BlobFS superblock.

The magic identifiers are statically defined and can be used as indicators for the start of the partition, although in testing representations of these identifiers were found in text files within the MinFS partition (Section 6.3.2). Within BlobFS only two flags are defined; *0x00 00 00 04*, the FVM flag, which indicates that this BlobFS partition is a part of an FVM partition and *0x00 00 00 01*, the clean flag which at the time of writing

is not used. If the FVM flag is not set, the last six entries within the superblock are not populated.

As with FVM, BlobFS uses blocks of 8,192 bytes in length. In terms of total size, the number of blocks allocated to the data section should equal the size of the number of slices allocated to the same section as stated within the superblock. Similarly, the number of Journal blocks and the number of inodes entries (64-bytes in length) should also be consistent with the number of FVM slices (in terms of size) allocated to each section. The null value for the location of the next BlobFS partition depicted in Figure 17 indicates that is the last (and only) BlobFS partition. The size of slices and the total number allocated to the partition should be consistent with the information stored in the FVM Superblock and Allocation Table.

The slice following the BlobFS superblock contains a simple bitmap for the data section, with each bit indicating whether a block within the data slices is free or not. These entries start immediately from the beginning of the slice; unlike other data structures within the Fuchsia environment, there is no padding / empty entries at the start. The number of allocated bits should match the number of allocated data blocks in offset 56 of the BlobFS superblock.

The third slice allocated to the BlobFS partition contains the Inode Table, which is made up of 64-byte entries. The number of these entries should correspond to the value defined at offset 64 within the BlobFS superblock. Unlike other data structures within FVM, the first entry in The Inode Table is populated with a legitimate entry. A breakdown of the structure of an example inode entry can be seen in Figure 18.

<table>
<tr><td>Relative offsets</td><td colspan="2">0</td><td colspan="2">4</td><td colspan="2">8</td><td>12</td></tr>
<tr><td>0</td><td>Flags
uint16
0000 0000
0000 0001</td><td>Version
uint16
0</td><td colspan="2">Next inode containing blob extent
uint32</td><td colspan="3" rowspan="2">Merkle root hash (SHA-256)
32 bytes
FF CA 3B 16 0B EA 2B 6B E1 A3 30 E1 C9
54 9A FB A3 0E 2B 1F 80 40 48 12 59 20
7C FC 08 E5 FB A6</td></tr>
<tr><td>16</td><td colspan="4" rowspan="2"></td></tr>
<tr><td>32</td><td colspan="3">Size of blob
uint64
54 bytes</td></tr>
<tr><td>48</td><td colspan="2">No. of Data blocks for this entry
uint32
1</td><td>Number of extents
uint16
1</td><td>Reserved
uint16
0</td><td>Extent offset for data start
uint32
32,797</td><td>Extent length
uint32
65,536 bits
(8,192 bytes)</td></tr>
</table>

Figure 18: Breakdown of an example BlobFS inode entry.

Each BlobFS inode entry is prefixed by one or more bitwise shifted flags (Table 3). From the devices examined all inodes entries which referenced compressed data utilised the Zstd compression flag rather than the LZ4. The inode entry shown indicates it is an uncompressed blob 54-bytes in length using a total of 1 data block (8,192 bytes) for storage. As this data is not fragmented, it is found in a single,

contiguous blob within the data partition, hence the single extent value at offset 017,004,980 and null value for ‘next inode’. The extent offset value indicates the data starts at block offset 32,797 from the start of the data slice and the length of this fragment is 65,536 bits or 8,192 bytes (i.e. one block length or 8,192 byes).

Flag	Description
0000 0001	Inode is allocated
0000 0010	Data is LZ4 compressed
0000 0100	Inode is an Extent container
0000 1000	Data is Zstd Compressed

Table 3: Bitwise shifted flags used in BlobFS Inode Table.

None of the Inode tables of Fuchsia devices examined contained an inode entry utilizing the Extent Container flag. However, the unique structure of an Extent container entry was identified through examination of Fuchsia’s codebase. Figure 19 contains the modified entry structure Extent containers utilised. In an Extent Container Entry, a flag of at least 0000 0000 0000 0101 must be set to indicate the entry is both allocated and Extent Container status. The next and previous inode values depend on whether the underlying blob data was fragmented across the data slices (necessitating multiple entries within the Inode table). The number of these fragments are outlined by the extent count. Up to six extent entries can be included in a single Extent Container (for a total size of 64-bytes).

Relative offsets	0		4	8	12	
0	**Flags uint16**	**Version uint16**	**Next inode containing blob extent uint32**	**Previous inode containing blob extent uint32**	**Extent count uint16**	**Reserved 2 bytes**
16	**Extent 1 offset for data start uint32**		**Extent 1 length (in bits) uint32**	**Extent 2 offset for data start uint32**	**Extent 2 length (in bits) uint32**	
32	**Extent 3 offset for data start uint32**		**Extent 3 length (in bits) uint32**	**Extent 4 offset for data start uint32**	**Extent 4 length (in bits) uint32**	
48	**Extent 5 offset for data start uint32**		**Extent 5 length (in bits) uint32**	**Extent 4 offset for data start uint32**	**Extent 6 length (in bits) uint32**	

Figure 19: Breakdown of an Inode Extent Container Entry.

The Inode Table slice is followed by the Journal slice. As both BlobFS and MinFS utilise the same journaling scheme, this has been documented separately in Section 6.3.3. The remaining slices allocated to BlobFS are used for the storage of blob data. The first block (8,192-bytes) of this slice is empty. The allocation of this space is utilized as per the values outlined in the Inode Table and are aligned on the block boundaries.

6.3.2 MinFS Partition

The MinFS partition is a “traditional Unix-like file system” (Fuchsia Project, 2019g). Within an FVM partition, it is made up of at least seven slices; A breakdown of the overall structure of the partition is shown in Figure 20.

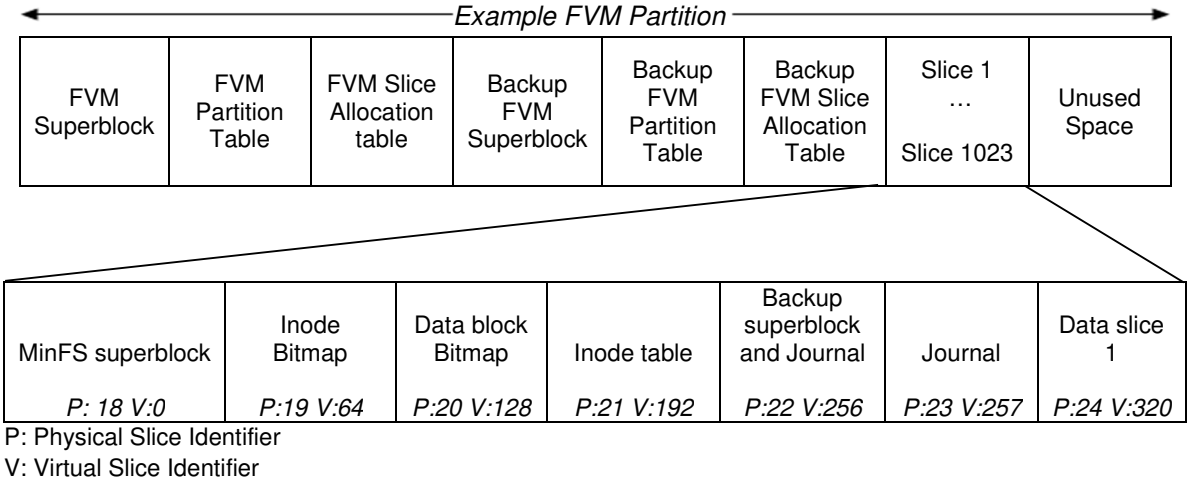


Figure 20: Basic layout of MinFS partition.

On current development builds, the MinFS partition is encrypted by default using the previously mentioned zxcrypt subsystem. Exploring methods for bypassing zxcrypt encryption was out-of-scope for this research, although the presence of zxcrypt encryption can be identified by the existence of a zxcrypt GUID at the start of the MinFS partition (Figure 21). It was possible to build Fuchsia without zxcrypt via the use of the QEMU virtual machines, which allowed for the generation of material used to populate the rest of this section. The first block of an unencrypted MinFS partition contains the partition’s superblock. A breakdown of the structure of the MinFS superblock can be seen in Figure 22.

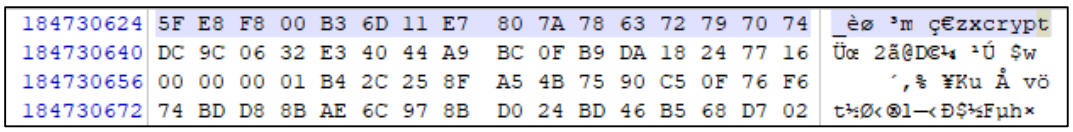


Figure 21: zxcrypt encrypted MinFS superblock.

The magic identifiers are statically defined and can be used as indicators for the start of partition, although in testing some plain text files contained representations of these identifiers were found within the data section of this partition. The CRC-32 checksum is calculated using the content of the superblock alone. The generation field is used to determine if the primary or backup superblock holds the latest content and should be updated with each write. Within MinFS, only two flags are defined; *0x00 00 00 02*, the FVM flag, which indicates that this MinFS partition is a part of an FVM partition and *0x00 00 00 01*, the clean flag which is currently unused. If the FVM flag is set, there are several additional fields populated from relative offset 80 onwards. These fields indicate the number of slices allocated for the bitmaps, Inode Table, Backup Superblock, Journal, and file data. These data structures are virtually

offset from one another by 64 slices. This allows FVM to allocate additional physical slices on demand whilst keeping any growing data structures within MinFS logically contiguous. Regardless of whether the FVM field is populated or not, the remainder of the block (8,076 bytes) from offset 114 onwards is reserved by the superblock and should be empty.

Relative offsets	0	4	8	12
0	**MinFS Magic Identifier Part 1** **8 Bytes** 0x21 4D 69 6E 46 53 21 00 (!MinFS!)		**MinFS Magic Identifier Part 2** **8 Bytes** 0x04 D3 D3 D3 D3 00 50 38 (ÓÓÓÓ P8)	
16	**Version (major)** **uint32** 9	**Version (minor)** **uint32** 0	**CRC-32 checksum** **uint32** 0x D0 AC 8b 7A	**Generation** **uint32** 1,472
32	**Flags** **uint32** 2	**Block size** **uint32** 8,192	**Inode size** **uint32** 256	**No. of data blocks** **uint32** 1,024
48	**No. of inode records** **uint32** 32,768	**No. of allocated data blocks** **uint32** 100	**No. of allocated inodes** **uint32** 91	**First block no. of inode bitmap** **uint32** 65,536
64	**First block no. of data block bitmap** **uint32** 131,072	**First block no. of inode table** **uint32** 196,608	**First block no. of backup superblock & journal** **uint32** 262,144	**First block no. of file data** **uint32** 327,680
80	**FVM slice size** **uint32** 8,388,608	**No. of slices allocated to MinFS** **uint32** 7	**No. of slices allocated to inode bitmap** **uint32** 1	**No. of slices allocated data block bitmap** **uint32** 1
96	**No. of slices allocated to inode table** **uint32** 1	**No. of slices allocated to backup superblock and journal** **uint32** 2	**No. of slices allocated to file data** **uint32** 1	**Index to first unlinked (but open) inode** **uint32** 0
112	**Index to last unlinked (but open) inode.** 0	**Remainder of block reserved** **8,076 Bytes** 0x00[8076]		
…				
8,176				

Figure 22: Example MinFS superblock with additional FVM fields utilised.

Following the superblock, the slices for the Inode and Data bitmaps can be found. Both bitmaps found in MinFS use the same simple scheme used by BlobFS where each bit represents the relative block allocation status. These are immediately followed by the MinFS Inode Table. An example of an entry within this structure can be seen in Figure 23. Each inode entry is 256 bytes in length and start with one of

two potential entry identifiers, depending on whether the inode is referencing a folder (0x04) or file (0x08). Due to the use of an unsigned 32-bit integer for the length, MinFS cannot support files bigger than 4GiB in size. As with Unix file systems (Hal Pomeranz, 2009), the link count refers to the number of directory entries that associate a specific name for the data specified (i.e. a Hard link). The timestamps used in MinFS are recorded in the number of nanoseconds since the start of Unix epoch.

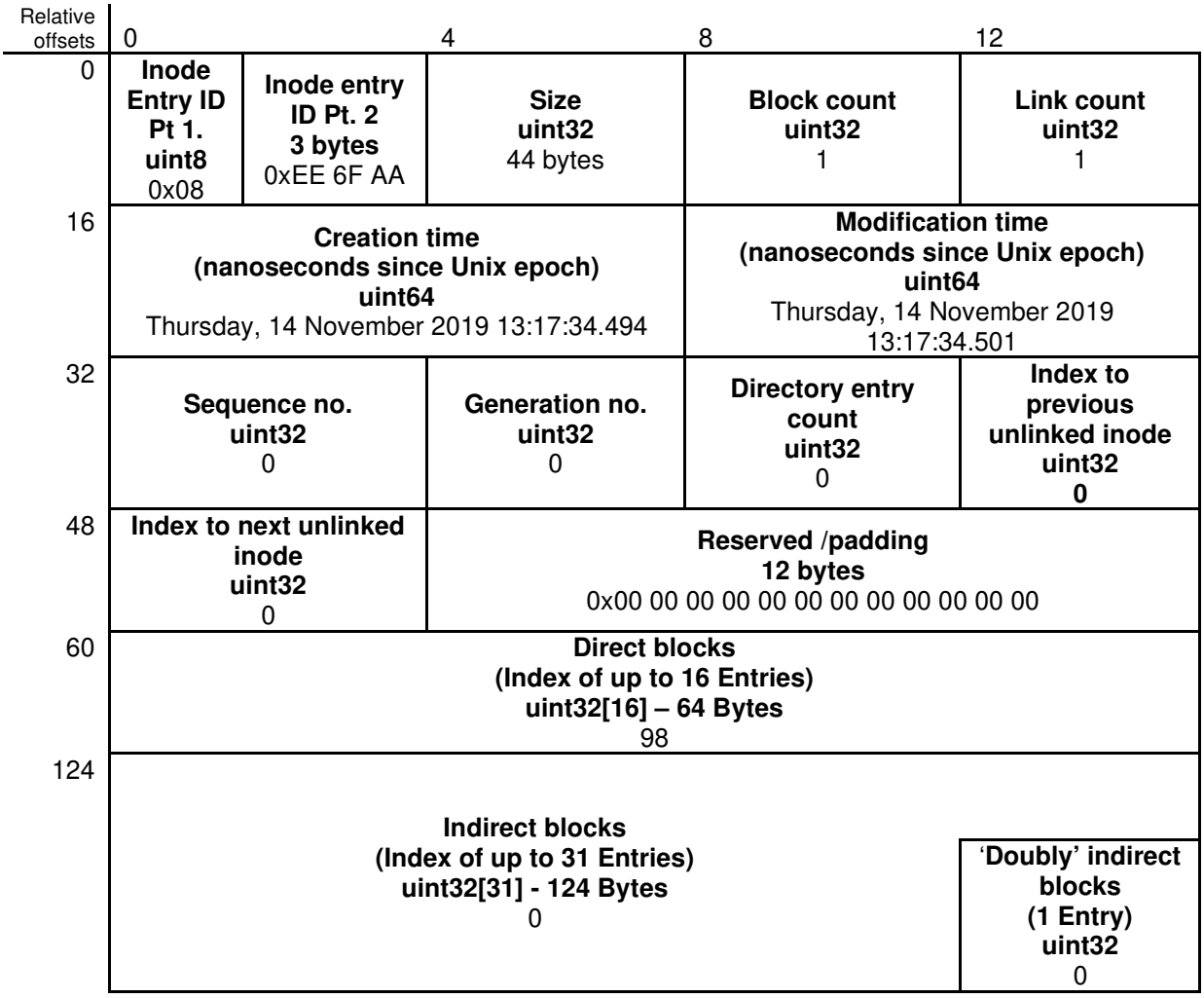


Figure 23: Example MinFS Inode Entry.

The sequence number should increment up each time the record is modified, however in testing this was found to be unreliable. The Generation number increments when the inode data is deleted. The directory entry count field is only populated if the inode record refers to a folder, and this field indicates the number of items within the directory entry. The next two fields relate to inode record(s) which no longer appear in directory entries but that still exist within the Inode Table. This is followed by 12-bytes of padding/reserved space and then an array of up to 16 individual pointers indicating the location of data on disk. Each pointer details the physical offset (in number of blocks) from the start of the data slice to the referenced block of data. If the data for the record is sufficiently large, MinFS has provisions for

a number indirect inode records and (if required) a single doubly indirect inode record within each entry. Unlike Unix-based filesystems, MinFS does not store information related to who the owner or which user groups can carry out read, write, or execute operations on files and folders.

The Journal slice(s) immediately follow the Inode Table, although the first 8,192-bytes of the first Journal slice is utilized for the backup MinFS superblock. For more information on the Journal, refer to Section 6.3.3. After the Journal, the data section(s) contains both directory entries and the raw data content of individual files. These are aligned on 8,192-byte blocks with the first entry left unutilised. The first populated blocks of the data slice contain directory entries. Examination of these entries can facilitate the identification of the actual name and individual inodes numbers of specific files. In Figure 24 a Directory Entry is shown, containing the record referenced in Figure 23 as the last entry. A breakdown of the structure of this record can be seen in Table 4.

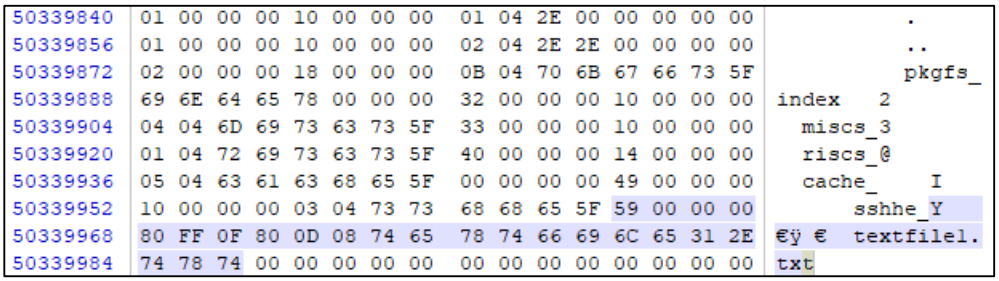


Figure *24: Example MinFS Directory Entry with final record highlighted.*

Offset	Length	Description	Value
0	uint32	Inode number (0 if record should be ignored)	89
4	uint32	Record length Low: 28 bits = Length High 4 bits = Record Flag	0x80 FF 0F 80
8	uint8	Name length	13
9	uint8	Record type: (0x04 = folder, 0x08 = file)	0x08
10	1-255 byte char	File/folder name (maximum of 255 characters)	textfile1.txt

Table 4: Highlighted MinFS Directory Entry record structure.

As seen in Figure 24, both the first two records within the entry have an inode number of '0x01' due to this being the root directory of the MinFS partition. This field is also used to indicate if a record in an entry is active; If the inodes numbers were set to 0x00, the relevant record would be considered free and skipped over during lookups. As such previously deleted records may be identified by inspecting these entries. The length of these records must be a multiple of four, which results in the

use of null characters or characters taken from the previous record's name for padding and/or alignment. This can be seen in Figure 24 where the folder names "*ssh*" and "*r*" have been padded with excess characters for alignment. As with an item's inode, the record in a directory entry indicates whether it refers to a file or folder. The length of these records are 28-bit numbers, with the last 4 bits of the field being used to set flags for the record. Only one flag for directory entries is defined, *0x08*, which indicates the record is the last one in this entry. When this flag is set, the true length of the record entry is overwritten with the maximum length of a directory (128 blocks or 1,048,576 bytes) minus the offset from the start of the directory entry to the start of that field. Using Figure 24/Table 4 as the example, the offset to that field is 128-bytes which equals 1,048,448 bytes or *0x80 FF 0F* in little endian (as seen in Table 4). Using the information from the Directory Entry and the Inode Table, it is possible to correctly attribute file content to specific files within the data slices (Figure 25).

51134464	2A 2A 2A 73 74 61 72 74 5F 6F 66 5F 74 65 78 74	***start_of_text
51134480	5F 66 69 6C 65 2A 2A 2A 65 6E 64 5F 6F 66 5F 74	_file***end_of_t
51134496	65 78 74 5F 66 69 6C 65 2A 2A 2A 0A 00 00 00 00	ext_file***

Figure *25: Data for referenced file in Table 4 and Figure 24.*

Further examination of the data section indicated that previously deleted data may be found in unallocated data blocks. In testing it was observed that deleted data appeared to persist after creation of additional files, indicating that previously allocated blocks data are not prioritised for reallocation. However, when a previously allocated block was reutilised, any previous data within that block appeared to be purged prior to the write operation, resulting in no data persisting in the unused space of the newly allocated block.

6.3.3 Journaling for BlobFS and MinFS

Relative offsets	0	4	8	12
0	**Journal Info block magic identifier** **8 Bytes** 0x 6C 6E 72 6A 62 6F 6C 62 (lnrjbolb)		**Starting block** **Uint64** 340	
16	**Reserved** **8 bytes** 0x00 00 00 00 00 00 00 00		**Sequence Number** **8 bytes** 1,274	
32	**CRC32 Checksum** **8 bytes** 0x8F B7 56 D0	**Unused**		

Figure 26: Structure of Journal Info Block.

Both BlobFS and MinFS utilise the same schema for journaling. Both filesystems utilise it as a log of filesystem write operations to ensure overall filesystem integrity, in the event of a loss of power or device reboot (Fuchsia Project, 2019f) (Fuchsia Project, 2019f). The first FVM block within the journal contains the Journal Information block, effectively the Superblock. An example of the Journal Information

block can be seen in Figure 26. This block can be identified by the Journal Info block magic identifier, as seen in the first eight bytes of the entry in Figure 26.

The starting block field indicates the first entry (in number of blocks) relative to the start of the journal entries. The Sequence number effectively acts as a timestamp, indicating which entry within the Journal was current when the Journal Information Block was last updated. The entry referenced by the starting block field should contain the same Sequence number as that found at offset 24 in the Journal Information Block. The checksum is calculated based on the preceding 32 bytes of the Journal Information block. Between the Journal Information Block and the indicated current entry, historic entries can be found. These should contain sequence number(s) of a value less than the one found in the Journal Information block. Each entry within the Journal starts at an FVM block boundary and contains one Journal Entry Header (first block in the entry) and one Journal Entry Commit block (final block of the entry within the Journal). The structure of the Entry header can be seen in Figure 27.

Relative offsets	0	4	8	12
0	**Journal entry block magic identifier** **8 Bytes** 0x6C 6E 72 75 6A 61 6D 69 (lnrujami)		**Sequence no.** **Uint64** 1,274	
16	**Flag** **Uint64** 0x00 00 00 00 00 00 00 01		**Reserved** **8 bytes** 0x00 00 00 00 00 00 00 00	
32	**No. of blocks between this and commit block** **Uint64** 3		**Actual location of data on disk** **Uint64 [679]** [Block: 196,610 | Block: 131,072 | Block: 0]	
...				
5,472	**Flags for blocks** **Uint32 [679]** [0 | 0 | 0]			
…			**Reserved** **4 Bytes** 0x00 00	
8176				

Figure 27: Structure of Journal Entry Header.

Whether a record is an Entry Header or Entry Commit Block can be quickly identified by the flag value (0x1 = header, 0x2 = commit). Two other values are defined (3=revocation, 0= unknown), however these were not seen in testing and no other information was identified within the code base. The true location(s) on disk for the payload as seen in Figure 27 are held in an array. This array can hold until 679 records. The subsequent Flags for block field also stores values in an array with each value corresponding to the block in the same position within the array within the

field above. The only block flag defined is an escape character to indicate that the referenced block starts with Journal entry block magic identifier. In these situations, these values are replaced with zeros, and a Flag value of 1 is set.

The payload data of the entry can be found in the block(s) between these two blocks. In testing, only metadata related to filesystem operations was found (different superblock versions, inode table records, and bitmap table). Within the codebase for Fuchsia there are indications that the Journal should be capable of storing actual file data as well partition metadata, so it is not clear why it was not found in testing. Following the payload of the entry, the Journal Entry commit block can be found on an FVM block boundary. The structure of the Journal Entry Commit block is similar, utilizing the same first four entries as a header, however the remaining fields are instead replaced with a single CRC32 checksum of the entire journal entry except the content of the commit block. This structure is shown in Figure 28.

Relative offsets	0	4	8	12
0	**Journal entry block magic identifier** **8 Bytes** 0x6C 6E 72 75 6A 61 6D 69 (lnrujami)		**Sequence no.** **Uint64** 1,274	
16	**Flag** **Uint64** 0x00 00 00 00 00 00 00 02		**Reserved** **8 bytes** 0x00 00 00 00 00 00 00 00	
32	**CRC32 Checksum** **8 bytes** 0x35 45 87 06	**Unused**		

Figure 28: Structure of Journal Entry Commit

Section 7: Discussion

Within this work the data structures found on Fuchsia storage disks were examined and their unique identifiers were determined. The content of the superblocks for Fuchsia partitions were found to detail key structural features including the requisite identifiers needed for quick identification. In the case of the ZIRCON-A partition, other unique identifiers denoting various ZBI containers within the partition were determined. Within the FVM partition the sizes of other structures within the partition were also found. This information alongside the GUIDs shown in Table 2 highlights several key features found which can be identified to Fuchsia disks and begin to map the high-level structure.

The material presented in this research is based off the default behaviour of current development builds for Fuchsia, compiled to the core x64 specification. As such, one of the gaps in this research is the lack of results for a platform utilising the ARM architecture. Given there are indications that the ZBI Kernel header (Figure 5) found within the ZIRCON partition utilises a different unique identifier based on architecture, there may be other differences in the number of headers and types utilised between the different architectures.

Key information related to the filesystems within FVM was determined through the examination of the superblock, FVM Partition Table and FVM Slice Allocation Table. The maximum theoretical amount of storage which may be allocated to the filesystems within FVM and how this data is broken up was identified along with the amount of diskspace (including number of slices) that was actively being utilised at the time of capture. This is seen in the FVM Partition table (Figure 11) where this information is specified in the form of a GUID, given name and a total number for allocated slices. Lastly, much of the content of the Slice Allocation Table was determined (Figure 12). This allows for the identification of the physical location on disk of the slices as well as provides context to their virtual position within the individual filesystems. With these findings, it is possible to determine the layout of FVM partitions and examine the content of them.

The ambiguity over the ‘extra’ bytes seen in each slice entry within the FVM Slice Allocation Table remains a possible source of confusion. Given what was seen in the source code related to the creation of new entries (Klein et al., 2020) compared to what was found on disk (Figure 12) and how the content of this table was represented within fvm-check (Figure 14), it is unclear why one 16-bit, one 32-bit and 2-bytes of padding appears to be utilised for each slice entry rather than two 32-bit values.

The data structures utilised by the partitions within FVM were examined and the associated metadata was identified. The content of the Superblock for both BlobFS and MinFS was found to outline key structural features for each partition. As seen in Figures 17 and 22, the sizes and location of other data structures is clearly documented, alongside data primitives such as the block size used within the partition. This facilitated the mapping of the distribution of data within each partition. It is also possible to determine what file metadata is store within each filesystem. This is seen in the Inode records (Figures 18 and 23) and in the case of MinFS, directory entries (Table 4). This allows for the identification of the sizes and physical locations of referenced data on disk, and for MinFS, the creation and modification times and file names of specific objects. As MinFS has provisions for indicating which records should be skipped during lookups for the directory entries (Table 4), these may be utilised to identify previously deleted file within specific directories. As indicated in Section 6.3.2, deleted file data may be found in the unallocated blocks within the data section for MinFS, however no data from previous utilisation was found to persist in the slack space reallocated data blocks.

The role of the partitions in the context of the wider operating system were also partially identified through the examination of the data slices. The content of these slices indicates the separation of user space files from files and data for use in system services. MinFS clearly performs the role of the former with a format similar to that of some Unix filesystems (although it is lacking fields for storing information regarding user ownership or group privileges within its inode entries). This allows for the inferences that the MinFS partition may be of most interest to forensic

investigators when found within FVM on a Fuchsia device. In the case of BlobFS we can see how the content of the filesystem information is represented to the user in Figure 16, however further work is required to clarify if inspecting the content of these blobs to verify installed applications and/or other software.

The ambiguity over the lack of data entries within the Journals for both MinFS and BlobFS remains a source of confusion. As indicated in Section 6.3.3, the entries found clearly indicate that the journal appears to function, so it remains unclear why no data entries were found. In the case of MinFS this may have been a consequence of relying on the virtual machine to see the content of the partition. Further work is required to determine why; this may be a result of the relatively small sizes of each partition or that the underlying codebase is not final. Despite not being an original objective for this research, this work has highlighted how zxcrypt may pose a significant hinderance to further research and investigations. The default usage of zxcrypt on MinFS partitions means that the area of greatest interest to forensic practitioners may be encrypted. There are references within the codebase for Fuchsia that a null key is currently used (Fisher, 2019a), with comments (Fisher, 2019b) indicating an intended future reliance on using Trusted Platform Modules (TPMs) or Trusted Execution Environments (TEE) for hardware backed key storage and attestation. This potentially represents a significant hurdle for future investigations as without a method to bypass or extract the keys from the TPM, investigators may be forced to work off off-device backups or on the live system.

Section 8: Future Work

Further examination of the compressed BootFS filesystem found within the ZIRCON partition(s) should be carried out. This may be of useful in investigating and understanding Fuchsia's kernel and the implementation of its security model. Further experimentation with additional devices or alternative architectures may further indicate what other types of headers are utilised within the ZIRCON partitions.

Analysis of the codebase and experimentation with the operating system should take place to confirm the true purpose of the extra two bytes seen within each FVM slice table entry. Similarly, work is required in the examination of the blobs found on BlobFS partition with the aim of determining the content of these blobs. This could provide further understanding of the OS as well as additional methods for determining what software and/or system service is found on the target platform. This would allow for a judgement to be made on their usefulness to forensic practitioners.

Exploration of methods for mitigating or bypassing zxcrypt on MinFS data partitions may provide crucial for further investigations. Whilst its functionality has been documented previously (DSTL, 2019), and methods for overcoming similar implementations on older OSs has been researched (Bell, 2018) a method applicable for zxcrypt is not available at this time. Without a suitable approach,

practitioners may be denied access to the filesystem theoretically containing the most relevant information. Finally, as Fuchsia is still in development, confirmatory work is required to clarify whether the findings here still hold true on any eventual production device.

Further system wide analysis is required to enable reliable extraction of data relevant to forensic investigators. Whilst it has been determined that creation and modified times can be attributed to specific sets of data, current research has not determined a way to attribute this activity to specific users or system services. Additional investigation of Fuchsia data block allocation procedure is needed to determine viability of reliable data recovery through file carving or other means. As the OS continues to evolve, examination and identification of system-level artifacts of forensic relevance (system logs, users list and user applications, etc) will need to be carried out to enable investigator to put together a more complete picture of user and system activity.

Should Fuchsia see real-world use in the future, digital forensic tooling will need to be developed or updated to support their analysis. Given the range of potential devices that is provisionally being targeted by the OS, alongside the prevalence of Google-based products across the smart device market, digital investigators lack sufficient tooling to address this problem. As with many other digital forensic tools, this tooling will need to be semi-automated in the indexing and categorising of data on a target device as digital investigators are likely to lack the time to manually analysis each individual device to the level of detail documented here.

Section 9: Conclusions

This research has identified the unique identifiers found on the disks of Fuchsia devices and documented the various data structures utilised by the unique partitions found on this platform. As Fuchsia is still in development, these findings are reliant on there not being any significant changes to structure of the partitions examined. There remain unanswered questions regarding the content of the BootFS disk image found in the kernel partition and the structure of entries within the Slice Allocation Table in the FVM. Questions also remain regarding the content of the blobs found within the BlobFS filesystem and their usefulness to forensic examiners. This research has also highlighted the potential difficulties posed to forensic investigations due to the usage of zxcrypt. Building on the material generated by this research and its companion piece, further exploration of the Fuchsia operating system, such as the system services, should be possible.

Acknowledgements

This research was conducted as part of the award of MSc Digital Forensics on the NCSC certified track, at Cranfield University. Cranfield University maintain copyright over the conducted research.

Section 10: References

Barth, A., Kamar, A., Kell, B., Voydanoff, M. and Bauman, J. (2019) *fuchsia / fuchsia / master / . / zircon / system / dev / board*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/dev/board/ (Accessed: 23 November 2019).

Bell, P. (2018) *Bruteforcing Linux Full Disk Encryption (LUKS) With Hashcat.*, *Forensic Focus* Available at: https://articles.forensicfocus.com/2018/02/22/bruteforcing-linux-full-disk-encryption-luks-with-hashcat/ (Accessed: 15 December 2019).

Bradshaw, K. (2019) *The newly-launched Google Home Hub is 'Astro,' a known Fuchsia OS test device*. Available at: https://9to5google.com/2018/10/10/google-home-hub-fuchsia-os/ (Accessed: 7 July 2019).

Broz, M. (2019) *dm-crypt: Linux kernel device-mapper crypto target.*, *Gitlab* Available at: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt (Accessed: 14 December 2019).

Carrier, B. (2005a) 'Linux LVM and Windows LDM', in Pearson Education, I. (ed.) *File System Forensic Analysis*. Crawfordsville: Addison Wesley Professional, pp. 120–121.

Carrier, B. (2005b) 'Volume analysis of disk spanning logical volumes', *Digital Investigation*, 2(2), pp. 78–88.

Carter, S.S.& J. (2018) *Security in Zephyr and Fuchsia.*, *The Linux Foundation* Available at: https://www.youtube.com/watch?v=Jov4dTnjm2o (Accessed: 7 July 2019).

DSTL (2019) *Google Fuchsia: A first look at the Fuchsia operating system.*, *DSTL Digital Forensics Bulletin* Available at: https://mailchi.mp/ef10dc6a5a9f/digital-crime-scene-bulletin-edition-8 (Accessed: 7 July 2019).

Fairbanks, K.D. (2012) 'An analysis of Ext4 for digital forensics', *Proceedings of the Digital Forensic Research Conference, DFRWS 2012 USA*.

Fairbanks, K.D., Xia, Y.H. and Owen, H.L. (2009) 'A method for historical Ext3 inode to filename translation on honeypots', *Proceedings - International Computer Software and Applications Conference*.

Fisher, D. (2019a) *fuchsia / fuchsia / refs/heads/master / . / build / images / zxcrypt.gni*. Available at: https://fuchsia.googlesource.com/fuchsia/+/refs/heads/master/build/images/zxcrypt.gni (Accessed: 12 January 2020).

Fisher, D. (2019b) *fuchsia / fuchsia / master / . / zircon / system / ulib / zxcrypt / include / zxcrypt / fdio-volume.h*. Available at: https://fuchsia.googlesource.com/fuchsia/+log/boot-migration/zircon/system/ulib/zxcrypt/include/zxcrypt/fdio-volume.h (Accessed: 12 June 2020).

Fuchsia Project (2019a) *Fuchsia is not Linux*. Available at: https://fuchsia.dev/fuchsia-src/concepts#zircon_kernel (Accessed: 12 January 2020).

Fuchsia Project (2019b) *Zircon and LK*. Available at: https://fuchsia.dev/fuchsia-src/concepts/kernel/zx_and_lk (Accessed: 23 November 2019).

Fuchsia Project (2019c) *Filesystem Architecture*. Available at: https://fuchsia.dev/fuchsia-src/the-book/filesystems.md (Accessed: 14 July 2019).

Fuchsia Project (2019d) *Filesystems*. Available at: https://fuchsia.dev/fuchsia-src/the-book/filesystems.md (Accessed: 14 July 2019).

Fuchsia Project (2019e) *Zircon kernel to userspace bootstrapping (userboot)*. Available at: https://fuchsia.dev/fuchsia-src/concepts/booting/userboot#boot_loader_and_kernel_startup (Accessed: 23 November 2019).

Fuchsia Project (2019f) *Blobfs: An immutable, integrity-verifying package storage filesystem*. Available at: https://fuchsia.dev/fuchsia-src/concepts/filesystems/blobfs (Accessed: 14 December 2019).

Fuchsia Project (2019g) *MinFS*. Available at: https://fuchsia.dev/fuchsia-src/concepts/filesystems/minfs (Accessed: 14 December 2019).

Fuchsia Project (2019h) *Zxcrypt*. Available at: https://fuchsia.dev/fuchsia-src/concepts/filesystems/zxcrypt (Accessed: 14 December 2019).

Fuchsia Project (2019i) *Fuchsia Documentation*. Available at: https://fuchsia.dev/fuchsia-src (Accessed: 12 January 2020).

Fuchsia Project (2020a) *Fuchsia Source*. Available at: https://fuchsia.dev/fuchsia-src/development/source_code (Accessed: 12 January 2020).

Fuchsia Project (2017) *Fuchsia development environment bootstrap*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/scripts/bootstrap?format=TEXT (Accessed: 12 January 2020).

Fuchsia Project (2019j) *fx workflows*. Available at: https://fuchsia.dev/fuchsia-src/development/build/fx (Accessed: 12 January 2020).

Fuchsia Project (2019k) *Install Fuchsia on a NUC*. Available at: https://fuchsia.dev/fuchsia-src/development/hardware/developing_on_nuc (Accessed: 23 November 2019).

Fuchsia Project (2020b) *fuchsia / fuchsia / master / . / zircon / system / uapp / lsblk*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/uapp/lsblk/ (Accessed: 12 January 2020).

Fuchsia Project (2020c) *fuchsia / fuchsia / master / . / zircon / system / uapp / df*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/uapp/df/ (Accessed: 12 January 2020).

Fuchsia Project (2019l) *Fuchsia SDK*. Available at: https://fuchsia.dev/fuchsia-src/development/sdk (Accessed: 12 January 2020).

Fuchsia Project (2020d) *Zircon Kernel Commandline Options*. Available at: https://fuchsia.dev/fuchsia-src/reference/kernel/kernel_cmdline (Accessed: 28 June 2020).

Fuchsia Project (2019m) *Fuchsia Merkle Roots*. Available at: https://fuchsia.dev/fuchsia-src/concepts/storage/merkleroot (Accessed: 14 December 2019).

Fuchsia Project (2020e) *Blobfs*. Available at: https://fuchsia.dev/fuchsia-src/concepts/filesystems/blobfs (Accessed: 28 June 2020).

Google Git (2020) *Fuchsia*. Available at: https://fuchsia.googlesource.com/fuchsia/ (Accessed: 12 January 2020).

Google Git (2019a) *fuchsia / fuchsia / master / . / zircon / system*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system (Accessed: 13 December 2019).

Google Git (2015) *Jiri deployment script*. Available at: https://fuchsia.googlesource.com/jiri/+/master/scripts/bootstrap_jiri?format=TEXT (Accessed: 12 January 2020).

Google Git (2019b) *Jiri*. Available at: https://fuchsia.googlesource.com/jiri/ (Accessed: 12 January 2020).

Gusmeroli, S., Piccione, S. and Rotondi, D. (2013) 'A capability-based security approach to manage access control in the Internet of Things', *Mathematical and Computer Modelling*

Hal Pomeranz (2009) *Directory Link Counts and Hidden Directories*., *SANS Digital Forensics and Incident Response Blog* Available at: https://www.sans.org/blog/directory-link-counts-and-hidden-directories/ (Accessed: 10 January 2020).

Hockett, J., Jurka, M., McGrath, R., Auradkar, V. and Wilkinson, C. (2019) *build arguments - base packet labels*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/docs/gen/build_arguments.md#base_package_labels (Accessed: 23 November 2019).

Klein, S., McGrath, R., Barth, A. and Brittain, B. (2019) *fuchsia / fuchsia / master / . / zircon / system / uapp / fvm-check /*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/uapp/fvm-check/main.cc (Accessed: 23 November 2019).

Klein, S., Robinson, J., Landers, T., Kulakowski, G., Valentino, G., Adam, B., Green, A. and Vikram, A. (2020) *FVM Format*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/ulib/fvm/include/fvm/format.h (Accessed: 16 June 2020).

Li, A. (2019) *Google poaches 14-year Mac veteran from Apple to bring Fuchsia to market*. Available at: https://9to5google.com/2019/01/22/google-fuchsia-poaches-mac-veteran/ (Accessed: 23 November 2019).

Mark Gurman, M.B. (2018) *Project 'Fuchsia': Google Is Quietly Working on a Successor to Android.*, *Bloomberg* Available at: https://www.bloomberg.com/news/articles/2018-07-19/google-team-is-said-to-plot-android-successor-draw-skepticism (Accessed: 7 July 2019).

McGrath, R. (2019) *fuchsia / fuchsia / master / . / zircon / system / public / zircon / boot / bootfs.h*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/public/zircon/boot/bootfs.h (Accessed: 12 January 2020).

McGrath, R., Kalsi, G., Moradshahi, P., Voydanoff, M., Geiselbecht, T., Coyne, E., Barth, A., Malhotra, S., Graham, S., Kulakowski, G. and Tabaka, C. (2020) *fuchsia / fuchsia / master / . / zircon / system / public / zircon / boot / image.h*. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/public/zircon/boot/image.h (Accessed: 14 June 2020).

Narváez, G. (2007) *Taking advantage of Ext3 journaling file system in a forensic investigation*., *SANS Institute Information Security Reading Room* Available at: https://www.sans.org/reading-room/whitepapers/forensics/advantage-ext3-journaling-file-system-forensic-investigation-2011 (Accessed: 14 December 2019).

Naumann, S. (2018) *PylotOS - an interpreted Operating System*. Technische Universität Chemnitz.

Nikkel, B.J. (2009) 'Forensic analysis of GPT disks and GUID partition tables', *Digital Investigation*

Piper, S., Davis, M., Manes, G. and Shenoi, S. (2006) 'Detecting hidden data in Ext2/Ext3 file systems', *IFIP International Federation for Information Processing*

Pomeranz, H. (2010) *Understanding EXT4 (Part 1): Extents*., *SANS Digital Forensics and Incident Response Blog* Available at: https://digital-forensics.sans.org/blog/2010/12/20/digital-forensics-understanding-ext4-part-1-extents (Accessed: 14 December 2019).

Prokop, M. (2013) *ldmtool: accessing Microsoft Windows dynamic disks from Linux*. Available at: ldmtool: accessing Microsoft Windows dynamic disks from Linux (Accessed: 26 November 2019).

Rahman, M. (2018) *Google's Former Head of Android Platform Security is now working on Fuchsia*. Available at: https://www.xda-developers.com/google-head-of-android-platform-security-fuschia/ (Accessed: 23 November 2019).

Robinson, J., Barth, A., Vongsouvanh, A., Meschkat, S., Puryear, M. and Laligand, P.. (2019) *Fuchsia product definitions*. Available at: https://fuchsia.googlesource.com/fuchsia/+/refs/heads/master/products/README.md (Accessed: 14 June 2020).

Rocha, L. (2017) *Intro to Linux Forensics*., *Count upon Security* Available at: https://countuponsecurity.com/tag/linux-lvm-forensics/ (Accessed: 25 November 2019).

Setapa, S., Isa, M.A.M., Abdullah, N. and Manan, J.-L.A. (2011) '*Trusted computing based microkernel*'

Statt, N. (2019) *Google is starting to reveal the secrets of its experimental Fuchsia OS*. Available at: https://www.theverge.com/2019/5/9/18563521/google-fuchsia-os-android-chrome-hiroshi-lockheimer-secrets-revealed (Accessed: 7 July 2019).

Yip, Y., Swetland, B., Kilbourn, T., Evans, D., Mattson, J., McGrath, R., Kulakowski, G., Klein, S., Tucker, J., Geiselbecht, T., Voydanoff, M., Todd, E., Barth, A., Auradkar, V., Mitchener, B., Brittain, B. and Ramachandra, P. (2019) *fuchsia / fuchsia / master / . / zircon / system / uapp / gpt*. Available at: https://fuchsia.googlesource.com/fuchsia/+/boot-migration/zircon/system/uapp/gpt/gpt.cc (Accessed: 14 June 2020).

2021-09-20

Purple dawn: dead disk forensics on Google's Fuchsia operating system

Jarrett, Matt

Elsevier

Jarrett M, Morris S. (2021) Purple dawn: dead disk forensics on Google's Fuchsia operating system. Forensic Science International: Digital Investigation, Volume 39, December 2021, Article number 301269

https://doi.org/10.1016/j.fsidi.2021.301269