The CipherShed EFI Loader

- Concepts and Implementation -

Document Version: 0.6
Date: 5. Jan. 2016
Author: Falk Nedwal

Table of Contents

1 Necessary Background Information of UEFI

1.1 Useful Literature

The intention of this chapter is not to provide an introduction to the Unified Extensible Firmware Interface (UEFI). For this purpose, some books and a lot of articles exist on that topic. Unfortunately, there are not that many sources of information regarding the software development for UEFI. For an introduction in the topic, the author recommends the book “Beyond BIOS: Developing with the Extensible Firmware Interface” [BEYOND]. This book gives a good overview over the components which are interacting when the EFI based boot process of a PC is performed. This overview is enriched with code and pseudo-code examples in order to understand the complexity of the topic. To develop an UEFI device driver, the information are not detailed enough. For the software development, two papers are recommended that cover the required API:

- Driver Writer's Guide for UEFI 2.3.1 [DRVGUIDE]
- Unified Extensible Firmware Interface Specification [SPEC]

1.2 Necessity for an EFI approach for CipherShed

Regarding the CipherShed project, the first question that should be clarified is: Why the existing CipherShed boot loader can't be used for EFI based systems as well? There is no really short answer to that question. Of course, one answer is: The old loader is BIOS based! But what does this mean? The boot process of traditional, BIOS based PCs requires boot media (hard disks) that contain dedicated boot loader programs at defined locations. The BIOS is locating these loaders by looking for an Master Boot Record (MBR) at the hard disks of the PC. The BIOS analyzes the content of this sector and executes some code from it. The access to the boot media is done using well defined interrupt calls (INT 13) while the CPU is running in 16 bit real mode. On the other hand, modern systems do not use this procedure anymore. One reason for the change to EFI is that the specified data structures to address the data partitions on the hard disks (the partition table) became insufficient to handle the big sizes of actual hard disks. A new structure to define hard disk partitions was specified: GPT (GUID Partition Table). This new structure is more flexible and able to address the usual sizes of modern hard disks and storage media. But this structure is not compatible with the usual BIOS based boot process. Although a kind of backward compatibility was implemented in the GPT structures, a completely new approach for PC initialization and boot-up was specified and introduced at the beginning of the 21st century: The Extended Firmware Interface. With this interface, the concept of the MBR based booting and the INT 13 based access was replaced. The boot-up of Operation Systems (OS) now is controlled by executable files that are stored at a defined location at the boot media. The access to this file, including the installation of the filesystem driver, is now part of the EFI infrastructure that is provided by the platform vendor and replaces the old BIOS. Since the OS boot loader now is an executable file, the CipherShed loader also needs to be an executable file that has nothing to do with the old (BIOS based) TrueCrypt or

CipherShed boot loader.

1.3 Basic Terms of the EFI infrastructure

In order to be able to easily follow the following sections of this paper, some basic terms and principles of the UEFI infrastructure are explained, that are relevant for CipherShed: UEFI components are generally executable files with the extension ".efi", these are called EFI images. EFI images may contain different kinds of code, the relevant types are:

- EFI applications
- EFI drivers

EFI applications are intended to perform a finite job and are exited when the job is done. A typical example of an EFI application is a boot loader that performs its job and finishes as soon as the OS kernel is started. Another example is the EFI shell that provides an user interface to the EFI environment.

EFI drivers provide services. Depending on the kind of driver, they are available at boot time only or at boot time and at runtime of the OS. The former are called EFI boot services while the latter are called EFI runtime services. Drivers can also be distinguished by their responsibility:

Bus Drivers

Bus drivers access the system hardware and provide device nodes in order to communicate with the hardware. For that purpose, bus drivers usually scan the hardware and provide (logical) child device nodes, called controllers. These child objects can be referenced by unique handles. To communicate with the hardware, the handles must be known. To get the handles, the controllers are hierarchically structured and can be addressed by an unique device path (see [BEYOND] for details).

Device Drivers

Device Drivers usually interact with the hardware using controller handles. To be used by EFI applications or other device drivers, they offer so-called "protocols". EFI protocols are interfaces to the services that drivers offer. These interfaces are specified in [SPEC] and uniquely identifiable by a Global Unique Identifier (GUID). Consumers of EFI protocols need to get a protocol handle in order to access the interface functionality of the EFI protocol. These handles can be requested by the GUID of the protocol. For more details, see [BEYOND].

Hybrid Drivers

Hybrid Drivers are the combination of bus drivers and device drivers: They can offer child controller handles as well as protocol handles.

During boot-up of the PC, the platform dependent EFI firmware initializes the hardware and provides bus drivers and device drivers. The results of this initial scan process are stored in the EFI handle database. Every EFI application and every EFI driver gets a pointer to this handle database when its main function is called. This enables the application or driver to access all handles that are already initialized. The database can also be extended or modified in order to reflect the services of

the newly started driver.

2 Requirements to CipherShed Loader

The basic idea for a meaningful structure of the CipherShed loader was generated by the 3 main tasks of the loader, that are mapped to 3 different components which are described in the next sections.

2.1 CipherShed (crypto) driver

The CipherShed driver is the core component of the loader that handles the encrypted media. Since the OS is installed on an encrypted media, the OS loader needs to have access to the unencrypted data during the entire boot process. Hence, an additional driver is required that has access to the encrypted data and provides an interface to the OS loader to access the unencrypted data. Because the OS loader cannot be adjusted or modified, this driver must be transparent for the OS loader. Since the additional driver consumes an interface to the encrypted media and provides the same kind of interface to the OS loader (and other EFI applications), a so-called “filter driver” is required. In that case the block IO protocol is filtered: The driver accesses the encrypted media by the block IO protocol and at same time it provides the block IO protocol to the OS loader. The only activity “in-between” these two protocol interfaces is the encryption/decryption process that is applied to the read blocks or to the blocks to be written.

The initialization of the cipher engine that performs this encryption/decryption is done by another component of the CipherShed loader: The CipherShed controller (see below). This design decision was made because it does not belong to the usual tasks of a driver to provide a user interface and to perform other activities from the CipherShed service menu. This would make the driver more complex; although it would be possible from the UEFI point of view.

2.2 Controller Application

The CipherShed controller is an UEFI application and the most complex part of the CipherShed loader. It contains the user interface for the password input as well as the service menu. This can also be a graphical interface since UEFI supports graphical interactions with the user.

Further, the CipherShed controller implements the flow control of the entire initialization and authentication procedure. It reads the volume header file containing the media encryption keys and other data. Using the user password, this volume header data can be decrypted and used to initialize the CipherShed driver. In the usual case of booting the OS from the encrypted media, the controller starts and initializes the CipherShed driver with the data from the decrypted volume header. When this was performed successfully, the controller loads and executes the corresponding OS loader application.

To influence the behavior of the the controller application (for example in terms of the provided user interface), an optional configuration file is defined. The controller is responsible for the

handling and parsing of this configuration file. The CipherShed loader design intends to fall-back to a secure default configuration in case that this file does not exist of is syntactically wrong. In no case, the content of the configuration file shall cause an insecure state of the system.

2.3 Runtime Driver

The CipherShed runtime driver is the smallest component of the loader. Its only responsibility is to hand-over some data from the CipherShed loader to the CipherShed driver of the OS. Since the CipherShed EFI driver is only active at boot time, it cannot be used to hand-over these data. According to the UEFI driver model, the runtime driver is the only kind of driver which is available at runtime of the OS and provides access to UEFI data from the OS.

The data that is needed by the OS driver is the media encryption key and the algorithm. Also, some status data are provided to the CipherShed OS driver.

3 EFI Development Environment under Linux

The author has decided to develop the CipherShed EFI loader under Linux. The intended binary format of EFI files (applications and drivers) is somehow based on Windows calling conventions, it uses Microsoft's Application Binary Interface (ABI). As it can be seen from the output of the “file” command, the binaries of EFI applications differ from binaries of EFI drivers:

```
$ file CsDrv_64.efi
CsDrv_64.efi: PE32+ executable (EFI boot service driver) x86-64 (stripped
to external PDB), for MS Windows
$ file CsCtr_64.efi
CsCtr_64.efi: PE32+ executable (EFI application) x86-64 (stripped to
external PDB), for MS Windows
```

Here is the same output for 32 bit driver and application:

```
$ file CsDrv_32.efi
CsDrv_32.efi: PE32 executable (EFI boot service driver) Intel 80386
(stripped to external PDB), for MS Windows
$ file CsCtr_32.efi
CsCtr_32.efi: PE32 executable (EFI application) Intel 80386 (stripped to
external PDB), for MS Windows
```

3.1 Available Libraries

The following sections introduce the two available environments/libraries for EFI development.

3.1.1 EDK2

EDK II is a modern, feature-rich, cross-platform firmware development environment for the UEFI specification (see [EDK2]). It is available for free and uses the BSD license. The environment is quite big and not part of the standard Debian repository (at least not at the author's installation). For Linux, it supports the GCC compiler. The library has more high-level functionality then the GNU EFI library (see below).

The package contains a lot of example code, especially the source code of the EFI shell. This was extensively used by the author to get an understanding of the implementation of various procedures.

3.1.2 GNU EFI

GNU-EFI is a more lightweight development environment to create EFI applications (and drivers) based on the GNU toolchain (see [GNUEFI]). It is part of the standard Debian repository. Unfortunately, the documentation of this environment is not that large. As information source, the Wiki [EFIWIKI] is very helpful.

At some point, the author decided to use the GNU EFI environment to develop the CipherShed EFI loader. As advantage, the build process is controlled by an ordinary Makefile that easily can be understood and adjusted in case of some trouble. As disadvantage, the development process showed that the provided header files are incomplete in some situations (the author used version 3.0v-5 of GNU EFI).

3.2 Testing using Visualization

To test the generated binaries a virtualization using “Qemu” was used. Qemu does not have an EFI firmware, so it needs to be downloaded separately. This is provided by OVMF (see [OVMF]): OVMF is an EDKII based project to enable UEFI support for Virtual Machines. OVMF contains a sample UEFI firmware for Qemu and KVM.

Further, Qemu allows to define a directory that contains the content of an emulated media (a partition). The EFI files to test need to be copied to this directory before the Qemu environment is started:

A command to start this virtualization environment for a 64 bit system could look like:

```
$ qemu-system-x86_64 -L /home/fn/src/emu -bios bios.bin.x64 -vga cirrus
-hda fat:/home/fn/src/emu/hda-content
```

The file “bios.bin.x64” is the compiled OVMF EFI firmware. The corresponding call for the 32 bit system could look like:

```
$ qemu-system-i386 -L /home/fn/src/emu -bios bios.bin.ia32 -vga cirrus
-hda fat:/home/fn/src/emu/hda-content
```

Although this environment allows to execute the EFI applications and drivers and to use the EFI shell, the author was not able to emulate a GPT based hard disk with GUID based partitions. Hence, the functionality of the CipherShed crypto driver could not be tested in that environment. Instead, an installation of Windows 8.1 inside VMware Player was used to test the entire boot process.

4 Implementation

4.1 The Big Picture

The following figure gives an overview about the structure of the CipherShed EFI loader, its

components and their embedding in the operational environment.

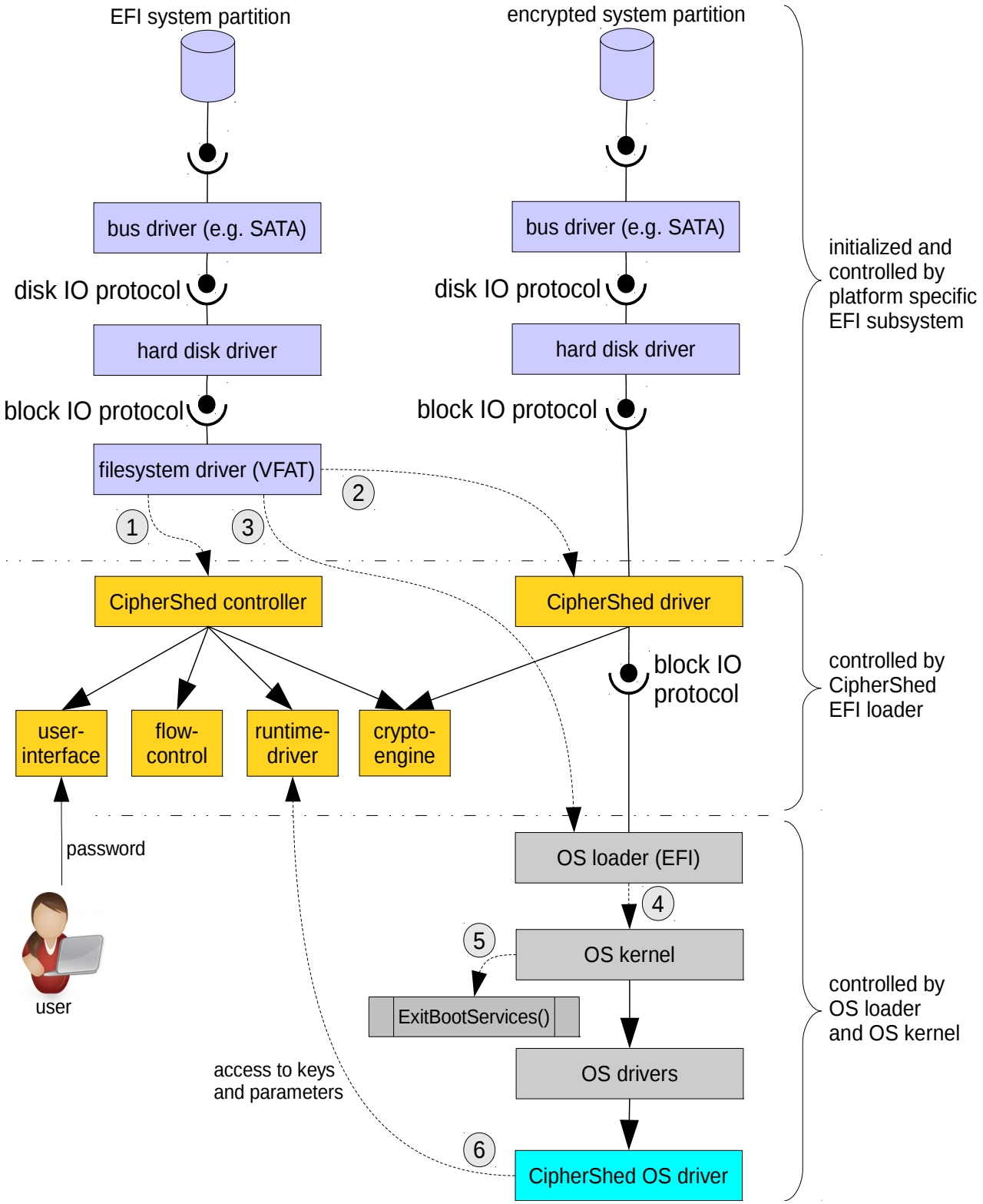

Figure 1: Embedding of CipherShed EFI loader in its Environment

The scenario in Figure 1 assumes the typical case that an GPT formatted hard disk contains (among others) one EFI system partition and one encrypted partition containing the OS. According to the EFI specification, the EFI system partition is required and is not encrypted. It contains a FAT32 file-system and can be identified by a dedicated GUID: `C12A7328-F81F-11D2-BA4B-00A0C93EC93B.` It contains a directory “/EFI” in its root that contains all UEFI related files. These files are further stored in individual directories, e.g. the Windows loader and related files are stored in “/EFI/Microsoft”. Further information is given in section 6.1 below.

Figure 1 shows a simplified sequence of device drivers that are connected to the partition controllers. The most important protocols are noted: Partitions as block devices are accessed by the block IO protocol that is provided by the corresponding EFI drivers that consume the disk IO protocol to access the media. The drivers and components in the upper part of the figure are initialized and provided by the EFI subsystem of the system firmware. This result is used by the CipherShed EFI loader as shown in the middle part of the figure. More details to that part are given section 4.2.1 below.

The lower part of Figure 1 shows the basic steps when the CipherShed EFI loader has started the OS loader. The only remaining task of the CipherShed EFI loader is the transfer of some data to the CipherShed driver of the OS. This process must be done in early boot stage since the CipherShed OS driver has to continue the decryption job of the CipherShed EFI driver to access the unencrypted data of the OS system partition.

4.2 Details to the CipherShed Components

4.2.1 CipherShed Controller

As shown in Figure 1, the CipherShed controller consists of the modules

- user interface,
- flow control,
- crypto engine, and
- runtime driver.

The concept of the runtime driver is further explained in section 4.2.3. The crypto engine is a collection of cryptographic functions that are used for the decryption of the volume header in a backward compatible way to TrueCrypt. In the current section, the flow control is described, which is the most important responsibility if the CipherShed controller.

The CipherShed controller is an almost linear sequence of steps that starts with the selection of the CipherShed loader entry in the EFI boot manager by the user. With this selection the EFI application containing the CipherShed controller is executed. When no dedicated EFI boot manager is installed, this execution step can also triggered automatically by start of the default boot entry in the EFI environment. This behavior is defined by the NVRAM variables that define the boot

sequence and the EFI applications to start.

The steps performed by the CipherShed controller are:

1. Initialization (shown as item (1) in Figure 1)
    - initialize the application state
    - get the current path in the FAT32 filesystem where the application was started
2. Load the settings
    - load and parse the configuration file
3. Load the volume header
    - load the file containing the volume header from its defined path relative to the current directory
4. Get the CipherShed driver locations
    - detect the path to the driver's “.efi” file in the EFI filesystem
5. Show the user dialog
    - ask for the user's password or other user decision
    - call the service menu if requested
6. Decrypt volume header
    - decrypt the loaded volume header
    - parse the decrypted volume header
7. If volume header could not decrypted (caused by wrong password), go back to step 5
8. Start the CipherShed driver (shown as item (2) in Figure 1)
    - load the driver's “.efi” file
    - start the driver
    - connect the driver to the configured controller device of the encrypted media
9. Boot the OS (shown as item (3) in Figure 1)
    - initialize the runtime service for data hand-over to the OS driver
    - load the configured OS boot loader “.efi” file
    - start the OS boot loader

The items (4) to (6) in Figure 1 are important steps after the CipherShed controller has finished its

job: In item (4) the OS loader is loading and executing the OS kernel from the (encrypted) system partition. When the kernel is started then the responsibility of the EFI boot services ends. The exact time when all EFI boot services are stopped and released from memory is defined by the ExitBootServices() call (see item (5)). This call is executed by the OS and triggers the cleanup procedure of the EFI boot services. Only the EFI runtime services remain active and available to the OS. In Figure 1 it is assumed that this call is executed before the OS drivers are initialized and started. In that case the CipherShed OS driver would also be started later and needs the runtime service to get its initialization data (item 6 in Figure 1).

Since the CipherShed EFI loader development is still ongoing and no practical tests with encrypted Windows OS were performed, the assumption about the call sequence may be wrong. It also might be possible, and makes sense, that the ExitBootServices() is called after initializing the CipherShed OS driver. In that case the runtime service to hand-over the initialization data would not required! Instead, an ordinary boot service could be established that hands over this data. Please note that this is a security relevant issue! Since the runtime service is available to the whole OS during runtime of the OS, the boot service is unavailable after ExitBootServices() is called. Hence, the sensitive key material that is subject of the data hand-over is not accessible anymore. In that case it's in the responsibility of the CipherShed OS driver to protect this data.

4.2.2 CipherShed Driver

The CipherShed is basically a filter driver for the block IO protocol.. That means that it consumes the block IO protocol from a controller where the driver is connected to. At the other hand it provides the block IO protocol to other EFI applications. That means that these other EFI applications need to connect to the CipherShed driver in order to use the provided block IO protocol. But when another application (like the Windows boot loader) connects to this driver, how does this work? Another application needs a controller to connect to. At this point there arises a problem: When this other application connects to the same controller where the CipherShed driver is connected, then this application get the same (encrypted) data when accessing the controller by the block IO protocol. The usual way to solve this issue is the creation of another controller handle by the driver as a child device of the controller where the driver is connected to. This new child handle could be accessed using the device path protocol of the parent controller: The path of the parent can simply be extended by a new node (“crypto”) to get the child path. When now an application connects to this child controller, then the block IO calls are handled by the CipherShed driver and provide the unencrypted data.

Why does this not work?

Unfortunately, the described behavior (access to the parent controller handle provides encrypted data and access to the child handle provides unencrypted data) does not work for the following reason: The Windows boot loader (“bootmgfw.efi”, tested using Windows 8.1) stores its boot device in the BCD data store that can be defined using the BCDEdit utility. The syntax expects something like “partition=C:” in the “device” section. The defined partition corresponds to a logically partition

device and cannot be modified to a child device to address the EFI child controller handle. Hence, the Windows boot loader always accesses the parent controller handle and gets the encrypted data instead of the unencrypted data provided by the EFI driver. Also, the Windows boot loader seems not to access the child handle automatically, it's simply ignored.

The desirable solution would be to exchange the roles of the parent and child controller: Access to the parent controller handle (the encrypted partition device) shall provide the unencrypted data provided by the CipherShed driver, and access to the child device can be used to access to the original (encrypted) data of the device. In fact, the child device might be omitted since no access to the original data is required (except when the disk should be permanently encrypted or decrypted). According to the opinion of the author, this behavior cannot be implemented by the normal use of the EFI API.

The “dirty” solution

In order to implement the desirable solution as described above, the author decided to go the following way: Since the block IO protocol of partition device (the parent handle) is provided by a foreign driver that is required by the CipherShed driver, it cannot simply be removed from the system or replaced by an own implementation. Instead, the CipherShed driver implements a runtime patch of this foreign driver code. The addresses of the interesting functions of the block IO protocol (for read and write access) can be detected by calling the OpenProtocol() function of the EFI API. The first few bytes of this code are now replaced by the code of an unconditional jump instruction towards an own function of the CipherShed driver. Now the accesses to the parent device are redirected to the own code. Since the own code strongly depends on the original functions to access the device, the driver works like this: When the original function need to be called, the replacement (patch) of this function is temporarily taken back, then the call is executed, and then the replacement is performed again. This works only for the reason that the EFI code is not multitasking or multithreaded code. Hence, the consistency of the patch is always ensured and controlled by the CipherShed driver. Now, every access to the encrypted device via the parent controller handle can be encrypted or decrypted as desired and read or written to the device using the corresponding original function. When the driver creates a child device then its block IO interface can provide access to the original (encrypted) data by simply calling the original functions as described above. The child device is only created by the CipherShed driver in case that a permanent encryption or decryption is requested by the user. This is controlled by an option to the driver.

One implication of this patch of the original block IO driver is that the CipherShed driver may also get access requests to other (not encrypted) partition or disk devices that are handled by the same driver. These accesses must be identified by the CipherShed driver and simply forwarded to the original code.

The implemented functionality was successfully tested using the EFI boot loader of Windows 8.1 running in VMware Player. Future versions of the Windows boot loader might behave differently

and may require the change of this concept. Also, the provided block IO driver of other systems might behave differently. Regarding the behavior of the EFI boot loader, the EFI specification version 2.5 [SPEC25] introduced a new feature: In section 12.11, a protocol EFI_BLOCK_IO_CRYPTO_PROTOCOL is specified that may solve the entire problem. When this protocol is established in the EFI environment of the PCs and used by the OS boot loaders then the CipherShed driver might be changed to support it.

Consumed and provided protocols

The block IO protocol is not the only protocol that is provided by the CipherShed driver. Figure 2 shows a more detailed view of the consumed and provided protocols of the driver (in fact this is still incomplete: only the most important consumed protocols are shown).

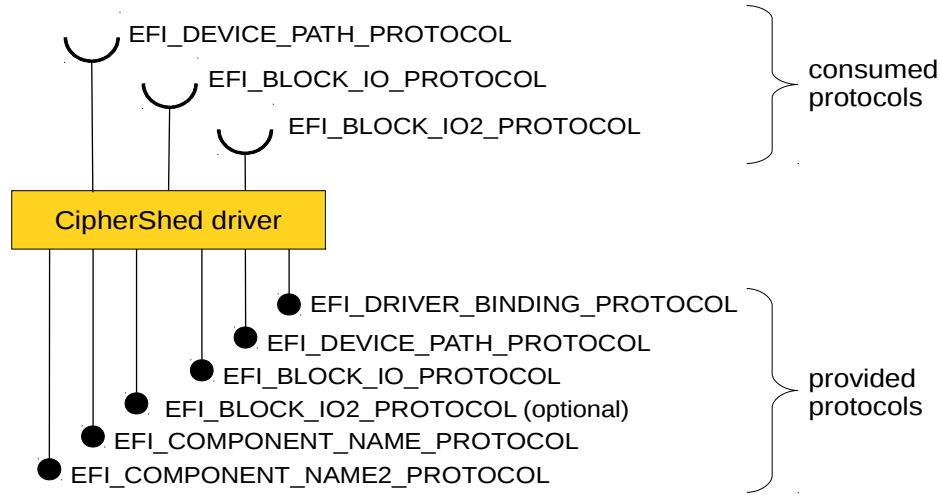

Figure 2: Protocols consumed and provided by the CipherShed Driver

In the following part, the meaning of the mentioned protocols are shortly explained, starting with the consumed protocols (further information can be found in [EFIWIKI] and [SPEC]):

EFI_DEVICE_PATH_PROTOCOL

This protocol is used to identify the intended controller. A correct identification is important because the encrypted media to connect to is identified by a GUID of the GPT partition.

EFI_BLOCK_IO_PROTOCOL

This is the most important protocol to access block devices. It provides the following interface:

```
typedef struct _EFI_BLOCK_IO_PROTOCOL {
  UINT64             Revision;
  EFI_BLOCK_IO_MEDIA *Media;
  EFI_BLOCK_RESET    Reset;
  EFI_BLOCK_READ     ReadBlocks;
  EFI_BLOCK_WRITE    WriteBlocks;
  EFI_BLOCK_FLUSH    FlushBlocks;
} EFI_BLOCK_IO_PROTOCOL;
```

Hence, application may directly read and write blocks at the controller. In the current use case the read and write access goes to the encrypted media that the driver has connected. Usually, filesystem drivers are using the block IO protocol to provide access to the file system.

EFI_BLOCK_IO2_PROTOCOL

The Block IO2 protocol defines an extension to the block IO protocol which enables the ability to read and write data at a block level in a non-blocking manner. It has a similar interface as the block IO protocol. In practice, this protocol seem not to be provided by the media controllers that the author has checked. In that sense the CipherShed driver always tries to open this protocol on the connected controller. In case that this protocol is not available, the driver recognizes this and does not provide this protocol by itself (see below).

Now to the provided protocols:

EFI_DRIVER_BINDING_PROTOCOL

This is a very important protocol that every driver needs to provide. It has the following interface:

```
typedef struct _EFI_DRIVER_BINDING_PROTOCOL {
  EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED Supported;
  EFI_DRIVER_BINDING_PROTOCOL_START     Start;
  EFI_DRIVER_BINDING_PROTOCOL_STOP      Stop;
  UINT32                                Version;
  EFI_HANDLE                            ImageHandle;
  EFI_HANDLE                            DriverBindingHandle;
} EFI_DRIVER_BINDING_PROTOCOL;
```

This protocol is used whenever a driver is connected to a controller. To check whether the intended controller is really supported by the driver, the Supported() function is called first. In this function, the driver developer has to implement rules that check the properties of the provided controller handle. In case that the driver supports that kind of controller, the function must return successfully, otherwise with an error code. In the successful case the next step is the call of the Start() function that really connects the driver with the controller. In this function, the driver is initialized and the consumed protocols are opened by the driver. At same time, the driver registers all its protocols at the EFI handle database that are provided by the driver. At the end, the provided protocols are initialized and their interfaces can be used by other EFI applications and drivers.

EFI_BLOCK_IO_PROTOCOL

As mentioned above, this protocol provides access to block devices. As described above, the provided block IO protocol is a patched version of the consumed block IO protocol.

On the parent device (the encrypted partition), the driver manages the encryption/decryption of the data before or after the disk is accessed: In case of read access, the request is forwarded to the consumed block IO protocol interface to get the encrypted blocks from the media device. After the decryption function is called, the decrypted data is returned to the caller. In case of a write request, the given block is encrypted first, then the resulting data is forwarded to the consumed block IO protocol for write access to the media. The Flush() and Reset() calls are simply forwarded to the corresponding interface of the consumed protocol.

On the child device, no encryption/decryption is implemented, the access is simply forwarded to the consumed protocol.

EFI_BLOCK_IO2_PROTOCOL

As already mentioned above, this protocol is only provided by the driver if the connected controller handle of the encrypted media provides this protocol. In that case the data handling is implemented similar to the block IO protocol.

EFI_COMPONENT_NAME_PROTOCOL

This protocol has no functional importance for the block data handling. It provides the human readable name of the CipherShed driver. For the current implementation, this protocol is used by the driver to identify whether the driver is already connected to a given controller. This helps to prevent from endless loops in the Supported() and Start() interface of the driver binding protocol.

EFI_COMPONENT_NAME2_PROTOCOL

This protocol is an alternative way to get a human readable name of the driver. Its implementation is equal to the the component name protocol.

4.2.3 Runtime Service

As already mentioned in the previous sections, the runtime service is only defined to hand-over key data, algorithm information and other status information from the EFI loader to the CipherShed OS driver.

The BIOS based loader of TrueCrypt used a fixed, hard coded physical address to hand-over the first data portion to the OS driver (TC_BOOT_LOADER_ARGS_OFFSET = 0x10). Although EFI applications and driver also have access to physical memory addresses, the author was unable to write data to the intended addresses from the EFI application: The memory map as maintained by the EFI environment can be shown by the “memmap” command in the EFI shell. The output looks like this:

```
Type       Start            End              # Pages          Attributes
BS_Code    0000000000000000-0000000000000FFF 0000000000000001 000000000000000F
BS_Data    0000000000001000-0000000000001FFF 0000000000000001 000000000000000F
BS_Code    0000000000002000-000000000000BFFF 000000000000000A 000000000000000F
Available  000000000000C000-0000000000057FFF 000000000000004C 000000000000000F
Reserved   0000000000058000-0000000000058FFF 0000000000000001 000000000000000F
Available  0000000000059000-0000000000068FFF 0000000000000010 000000000000000F
BS_Data    0000000000069000-000000000006AFFF 0000000000000002 000000000000000F
BS_Code    000000000006B000-000000000008BFFF 0000000000000021 000000000000000F
...
```

In this example the very first memory page (where the TC_BOOT_LOADER_ARGS_OFFSET is contained) is occupied by “BS_Code” which stands for EFI Boot Service Code. Hence, this area is not available to be overwritten with the data for the OS driver and another method of data hand-over to the OS is needed. This leads to the loss of backward compatibility with TrueCrypt in this area. On the other hand, this incompatibility should not be a problem due to the following two reasons:

1. Since the hard disk and partition structure of GPT partitions differ from the structure of MBR based systems, the OS driver needs to be adjusted to the EFI versions anyway.
2. The hand-over at fixed physical addresses seems to a be dirty workaround compared to the usage of EFI services. Hence, EFI services provide a better way to implement this task.

The service is implemented in the CipherShed controller. It uses an EFI environmental variable that contains the data to hand-over. This EFI variable can be accessed (read-only) by the OS using the corresponding API.

The EFI function to set the variable is as following:

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_VARIABLE)(
  IN CHAR16   *VariableName,
  IN EFI_GUID *VendorGuid,
  IN UINT32   Attributes,
  IN UINTN    DataSize,
  IN VOID     *Data
  );
```

It can be seen, that the access to this variable requires a GUID. This GUID and the name of the variable must be known by the OS driver. For this purpose, the following GUID (VendorGuid) was newly created:

```
#define CS_HANDOVER_VARIABLE_GUID     \
    { 0x16ca79bf, 0x55b8, 0x478a, {0xb8, 0xf1, 0xfe, 0x39, 0x3b, 0xdd,
0xa1, 0x06} }
```

As name of the variable “cs_data” is used.

With the “Attributes” parameter of the interface the scope of the variable can be defined: By choosing the attribute EFI_VARIABLE_RUNTIME_ACCESS, the access to the variable will be possible as runtime service, otherwise not. Hence, the scope can easily changed from an EFI runtime service to an EFI boot service.

5 The User Interface

The user interface is designed to look similar to the interface to TrueCrypt boot loader. It uses a text based interface. In contrast to the TrueCrypt interface, there is no distinct rescue menu anymore, since a dedicated rescue disc or other rescue media is supposed to be unnecessary. Instead, a service menu exists that provides basic operations that differ from the rescue menu of TrueCrypt. As special feature, some of the menu items can be influenced in the options file: Some items are disabled by default and need to be enabled in the option file if required (see section 6.2).

The user interface is completely implemented in the CipherShed controller. The following Figure 3 shows the actual process flow:

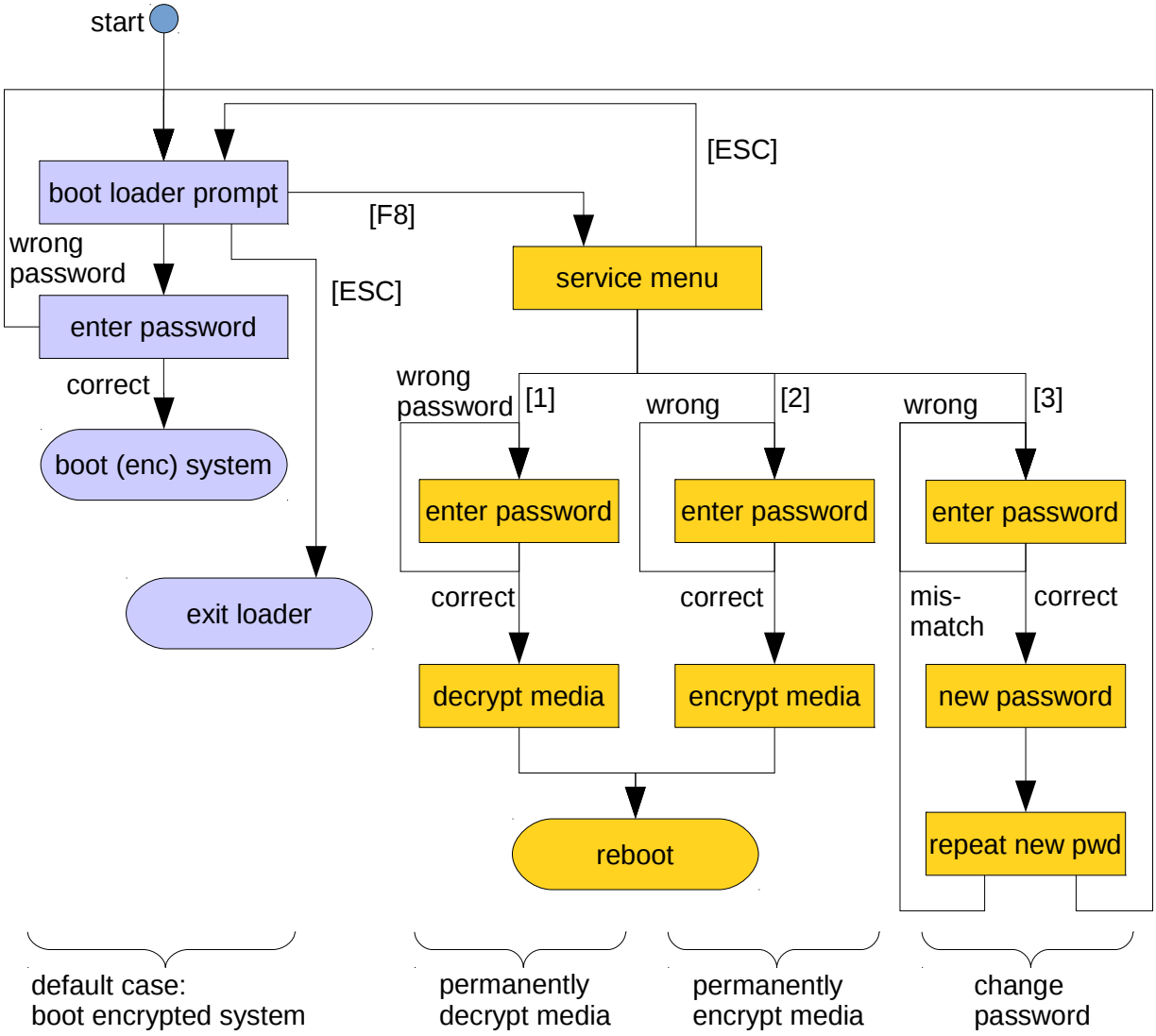

Figure 3: Process Flow of the User Interface

6 Configuration

6.1 Filesystem Structure

All relevant files for the CipherShed EFI loader are stored at the unencrypted EFI system partition in the directory /EFI/CipherShed/. The following files and directories are expected there:

- CsCtr_64.efi or CsCtr_32.efi
- CsDrv_64.efi or CsDrv_32.efi
- settings
  This is the configuration file for the CipherShed controller, see section 6.2.

- volume/
  This is a directory that contains the volume header file(s).

The *.efi files must match to the underlying system: The EFI files are built for i386 and x86_64 architecture.

6.2 Configuration file

The CipherShed EFI loader configuration is stored in the file /EFI/CipherShed/settings. If this file is missing or corrupt then the CipherShed controller falls back to default settings.

The configuration file is a simple text file containing key/value pairs. Lines starting with “#” are ignored. Also, unknown keywords are ignored. The keywords are not case-sensitive. The following table lists and explains the accepted keywords.

keyword/value	meaning
DEBUG=<number>	In case that a debug version of the loader (controller) is installed, this value defines the debug level (0->disabled). The values are interpreted according to the definitions in /usr/include/efi/efidebug.h
DRIVER_DEBUG=<number>	In case that a debug version of the driver is installed, this value defines the driver debug level (0->disabled). The values are interpreted according to the definitions in /usr/include/efi/efidebug.h
NOSILENT	This keyword disables the silent mode which is the default: When silent mode is enabled then no output is shown to the user. The loader only waits for the user password and continues after pressing the Enter key.
LANG=<language ID>	Definition of the language of the user interface. Currently supported are only English (“eng”) and German (“ger”).
ENABLE_SERVICE_MENU	If this keyword is present then the service menu is supported.
SHOW_SERVICE_MENU	If this keyword is present then the main menu contains a line indicating that and how the service menu can be called. If this is not enabled but ENABLE_SERVICE_MENU is, then the service menu can be called using the [F8] key, but no message about this option is shown.
ENABE_MEDIA_DECRYPTION	If this keyword is present then the service menu supports the decryption of the media triggered by the user.
ENABE_MEDIA_ENCRYPTION	If this keyword is present then the service menu supports the encryption of the media triggered by the user.
ENABLE_PASSWORD_CHANGE	If this keyword is present then the service menu supports the change of the user password.
ENABLE_PASSWORD_ASTERISK	If this keyword is present then asterisk characters are printed while inputting a passwords. This is enabled in all password input dialogs.

table 1: Available keywords for the Configuration File (need to be continued)

Here is a sample configuration file:

```
# CipherShed EFI loader
#
# sample configuration file
#

# set debugging: D_ERROR | D_WARN | D_LOAD | D_BLKIO | D_INIT | D_INFO
debug=0x80001047

# disable silent mode (which is the default)
nosilent

# language (supported only "eng" and "ger")
#lang=ger

# enable service menu
enable_service_menu

# show hint in main menu about service menu
show_service_menu

# enable permanant decryption option in service menu
enable_media_decryption

# enable permanant encryption option in service menu
enable_media_encryption

# enable change password option in service menu
enable_password_change

# show asterisk characters in password dialog (instead of no feedback)
enable_password_asterisk
```

6.3 Language Support

The user interface supports a basic support for different languages. That means that the strings for the text output in the user interface are available for multiple languages. Each available language can be selected in the configuration file using the keyword “LANG” (see section 6.2). The identifiers for the languages as well as the translated strings are compiled into the CipherShed controller.

To support additional languages or to change the translation of existing languages, the following information may be helpful: In file “cs_ui.h”, the enumeration “**enum** cs_enum_ui_language” is defined. Only these languages are supported. The corresponding strings and their translations are

defined in the file “cs_ui.c” in the data structure “`uiStrings ui_strings[]`”. Changes in this structure must comply with the set of available languages mentioned above and with the set of available keywords. The keywords are identified by an ID from the enumeration “`enum cs_enum_ui_stings`”. For additional languages, the corresponding strings in the configuration file are defined in the structure “`struct cs_language_name lang_strings[]`” in the file “cs_options.c”. Without adjusting this data, the (new) language cannot be selected.

6.4 Setup

The following sections give some hints about the expected actions for installation and deinstallation of the CipherShed EFI loader. These actions need to be done by the CipherShed setup application.

6.4.1 Installation of the EFI Loader

The installation of the CipherShed EFI loader at an ordinary GPT hard disk partition requires the following activities:

- The /EFI/CipherShed/ directory and the /EFI/CipherShed/volume/ directory need to be created in the EFI system partition.
- The CipherShed controller and the driver need to be stored in the CipherShed directory.
- The volume header need to be created and stored as a file (with the size of exactly 512 byte) in the /EFI/CipherShed/volume/ directory. The file name must follow one of the following two conventions:
  - The name of the file must be equal to the UUID string of the GPT system partition to be encrypted.
  - The file name can be arbitrary, but if the name is not equal to the UUID of the GPT system partition to be encrypted, a dedicated file “index” must be present in the /EFI/CipherShed/volume/ directory. This index file must contain an assignment between the UUID string of the GPT system partition to be encrypted and the corresponding file name of the file containing the volume header. In any case when an index file exists, the CipherShed loader first searches for the volume header in this file, in unsuccessful cases the loader checks the other file name convention (see above). This feature was implemented to support EFI systems that only allow “8.3” file names. Especially the test system of the author (VMware Player 7.1.0 for Windows) uses an EFI environment that was unable to write or create files with long file names.
- Create an entry in the EFI boot device list to start the CipherShed controller either as default boot entry or as an entry of the boot manager. The details for this procedure are not described here. Important is the correct call of the CipherShed controller: It expects some arguments:

```
CsCtr_64.efi <UUID of the encrypted system partition> <UUID of the
partition containing the EFI OS loader> <full pathname to the EFI OS
loader file> [<options to be handed over to the EFI OS loader>]
```

An example call of the loader looks like this:

```
CsCtr_64.efi 6932ABA3-1261-4912-A09A-D1DD8FCBB430 FBA1107F-5151-42BC-8A00-
35FD2E5133A2 \EFI\Microsoft\Boot\bootmgfw.efi
```

The design of the CipherShed EFI loader is created with the intention that this EFI loader may also be installed at a removable media (an USB memory stick). The precondition to this option is that the removable media is GPT formatted and is bootable. This can be achieved by creating an FAT32 formatted GPT partition on the media that contains the /EFI/CipherShed/ directory structure as described above. Further, the PC needs to support the booting from removable GPT based media and must be configured to do that. Under this preconditions the setup is expected to be as described above. The command line to the CsCtr_64.efi ensures that the required information regarding the intended encrypted system partition are provided to the CipherShed loader.

6.4.2 Deinstallation of the EFI Loader

The deinstallation simply needs to take back the steps described in section 6.4.1 above. The most important aspects to keep in mind are:

- Ensure that the system partition is decrypted before deinstallation,
- Ensure that the entry in the boot menu containing the CipherShed EFI loader is removed.

7 Some remaining Details

7.1 The Driver Development Environment for Windows

As described in the CipherShed Wiki, the current development environment uses Microsoft Driver Kit version 7.1.0. The author used this environment and tried to extend the CipherShed Windows driver with a new function to take over the cipher related data from the EFI loader using an EFI environment variable (see section 4.2.3). To implement this, a kernel mode function is required that reads values from EFI environment. Unfortunately, the author did not find this function in WDK 7.1.0. Instead, the function is available starting with Windows 8 (see MSDN documentation). Its signature looks like this:

```
NTSTATUS ExGetFirmwareEnvironmentVariable(
  _In_      PUNICODE_STRING VariableName,
  _In_      LPGUID          VendorGuid,
  _Out_opt_ PVOID           Value,
  _Inout_   PULONG          ValueLength,
  _Out_opt_ PULONG          Attributes
);
```

This function is part of WDK 8.1 and not WDK 7.1. WDK 8.1 supports the development of drivers for Windows 7, Windows 8 and Windows 8.1; Windows XP (and below) drivers seems not to be supported. Since the CipherShed driver project in Visual Studio is based on a makefile (and “nmake”), the author was not successful in building the CipherShed driver using WDK 8.1 (although it should be possible).

For the described reasons, it might be reasonable to switch the CipherShed development environment in the future towards Visual Studio Community 2013 including WDK 8.1. This might imply that Windows XP and older versions are not supported anymore, especially for the driver development. On the other hand, Windows XP is supposed to be insecure and outdated anyway, hence it might be acceptable not to provide new CipherShed versions for these systems.

7.2 Changes and Requirements to the CipherShed OS Driver

As already mentioned in section 4.2.3, the hand-over of data between the loader and the OS driver need to be adjusted. Also, the location of the volume header changes, that was stored in the first sector of the encrypted partition in MBR based media. The author expects the following aspects to be changed in the OS driver to support the EFI loader:

- replace the kind of data hand-over from a fixed memory location to an EFI service providing the data in an EFI variable
- adjust the structure of the hand-over data block from the EFI loader: the old data structure contains some pointers to further physical memory locations that make no sense when using an EFI service
- adjust the access to the volume header: The old OS driver keeps the user password and accesses the volume header data at the OS partition. This might be not necessary: The only situation when the volume header needs to be accessed is the change of the user password. For that purpose (and for deinstallation of course) the location of the volume header file (including the UUID of the partition where it is stored) needs to be handed over from the loader to the OS driver. This concept supports the storage of the volume header at an external media: In case of a requested password change the external media needs to be connected, the OS detects the right partition using the known UUID containing the volume header. This implies another change:
- change the password management (see above): Since the volume header is stored in another

way, the password change procedure must be reworked. Also, empty passwords need to be supported, especially in case when the volume header is stored on an external media that is used as security token (see section 7.3).

The boot behavior of the Windows 8.1 EFI loader was analyzed and documented in the paper [WINBOOT]. It might be an interesting source for troubleshooting of the OS boot process.

7.3 Booting from removable Media

Booting from a removable media means, that the CipherShed EFI loader and the volume header is not stored at the same hard disk as the encrypted OS. Since all key material is contained in the volume header, this approach implements a two-factor authentication of the user: Only if the user owns the removable media and knows the correct password, the system can be accessed. This option shall also support an empty user password: Although the security level is decreased by this measure, the user still needs to present the removable media as security token to the system.

The technical aspects for the booting from a removable media are already contained in section 6.4.1 and 7.2.

7.4 Current Development Status

At the time of writing this paper, the EFI related issues of the CipherShed controller and driver are implemented and partly tested. This comprises the flow control of the controller as well as the driver binding protocol providing the start and initialization of the driver. Also implemented are all described protocols that the driver provides.

The cryptographic part is also implemented and partly tested. This includes the decryption and parsing of the volume header as well as the functions to encrypt and decrypt sector data from encrypted media based on the master key from the volume header. The cryptographic functions are taken from the TrueCrypt source. They needed some modifications to adjust the sources to the EFI environment. All tests in this area were performed using a little-endian architecture (32 bit and 64 bit). In case of big-endian systems, the cryptographic part will probably not work and needs a special version of the EFI loader.

The user interface is implemented as well. It only supports the text based interface similar to the TrueCrypt loader. Although the EFI library also contains mouse support, different display resolution and even graphical output, the author decided to stay with text based interface using the default display resolution. Graphical support might be implemented later if somebody thinks that the effort is justified.

The service menu is implemented as described in chapter 5. It reflects the author's opinion of meaningful options.

The following aspects are still unfinished:

- No tests were performed with removable media.

The current version of EFI loader sourcecode is available at [SOURCECODE].

7.5 Open Problems

For the following issues, the author actually has no idea:

- TrueCrypt offers the feature of a “hidden volume”. It is still unclear whether this feature can be implemented in a compatible way using GPT based hard disks. If a hidden volume is present, a second volume header file would be required. In that case it is not hidden anymore: The second volume header file indicates the existence of a second volume. Hence, a new concept for hidden volumes need to be defined: The management sectors of GPT based media (the GPT headers) are not suitable to be misused to store hidden data. Maybe the volume header of a hidden partition can be stored somewhere inside an encrypted GPT partition (for example as its first sector).

References

BEYOND: Vincent Zimmer, Michael Rothman, Suresh Marisetty, Beyond BIOS: Developing withe the Unified Extensible Firmware Interface, 2010

DRVGUIDE: Driver Writer's Guide for UEFI 2.3.1, 2012

EDK2: EDK II, http://www.tianocore.org/edk2/

EFIWIKI: Phoenix UEFI Wiki,

GNUEFI: GNU EFI, http://sourceforge.net/projects/gnu-efi/

OVMF: OVMF, http://www.tianocore.org/ovmf/

SOURCECODE: Sourcecode of the CipherShed EFI Boot Loader, https://github.com/f-n/CipherShed/tree/efi/src/Boot/EFI

SPEC: Unified Extensible Firmware Interface Specification, 2014

SPEC25: Unified Extensible Firmware Interface Specification, 2015

WINBOOT: Windows UEFI startup – A technical overview, http://news.saferbytes.it/analisi/2013/10/windows-uefi-startup-a-technical-overview/